Profitina: Building an AI-Visibility Platform in the Open

Irfan Subakti



Profitina

profitina.com

Profitina

Building an AI-Visibility Platform in the Open

Irfan Subakti



Copyright © 2026 Irfan Subakti. All rights reserved.

Published by Profitina · profitina.com

First edition, September 2026.

Every program in this book was compiled and run before its output was printed. The complete source code is in the accompanying Code folder. The programs are offered for teaching and learning; Irfan Subakti and the publisher accept no liability for their use elsewhere.

Contents

Preface ..... **xxvii**
Who this book is for xxvii
What is in it xxvii
How to read it xxvii
About the author xxviii
Acknowledgments xxviii

CHAPTER 1 Origins: From a Lecturer's Classroom to a Platform **1**
1.1 A Platform with a Classroom Behind It 1
1.2 The Founder and Three Decades of Teaching 2
1.2.1 A lecturer in Surabaya 2
1.2.2 Where the teaching began 2
1.2.3 Research in several countries and several branches of AI 2
1.3 The Classroom and the Code Base Are the Same Place 3
1.3.1 Three courses a platform is made of 3
1.3.2 A library of course books around one real system 3
1.3.3 The platform in a real class 4
1.4 From Idea to Production: A Dated Timeline 4
1.4.1 Before July 2026 4
1.4.2 Twelve weeks, week by week 4
1.4.3 Counting the days: a first look at code 6
1.5 The Founding Ideas 7
1.5.1 The mission 7
1.5.2 Visibility and participation in one system 7
1.5.3 Local AI, by choice 7
1.5.4 Tested before it ships 8
1.5.5 Indonesia first 8
1.6 Who This Book Is For 8
1.7 How to Read This Book 9
1.7.1 The shape of the book 9
1.7.2 Every chapter has the same parts 10
1.7.3 Programs, outputs and the code folder 10
1.7.4 Boxes, labels and dates 11
1.8 Setting Up VS Code for Python and Java on Windows, macOS and Linux 11
1.8.1 Three words first 11
1.8.2 Visual Studio Code 12
1.8.3 Java: JDK 25 12
1.8.4 Python 3.14 13
1.8.5 The two VS Code extensions 13
1.8.6 A folder for the book 14
1.8.7 Checking the whole bench 14
1.8.8 Your first program, in two languages 17
1.8.9 A second program: days since the launch 18
1.8.10 Running and debugging in VS Code 21
1.9 Profitina in Practice 22
1.10 Common Misunderstandings 22
1.11 Chapter Summary 23
1.12 Exercises 23
1.13 Solutions to the Exercises 24
References 28

CHAPTER 2 From Search to Answers: The Rise of GEO 29
2.1 Where We Left Off 29
2.2 Twenty-Five Years of the List of Links 29
2.2.1 Crawl, index, rank 29
2.2.2 The person chooses 30
2.3 The Answer Engine Arrives 30
2.3.1 One answer instead of ten links 30
2.3.2 What changes for a business 31
2.4 The Numbers, 2025–2026 31
2.4.1 Fewer clicks when an answer appears 32
2.4.2 More answers, more often 33
2.4.3 The scale of the engines 33
2.4.4 Indonesia 33
2.4.5 Treat forecasts with care 34
2.5 What GEO Is: The Academic Origin 34
2.5.1 The paper that named it 34
2.5.2 Measuring visibility inside an answer 34
2.5.3 A benchmark and nine methods 35
2.6 Profitina's Definition of GEO 35
2.6.1 In one sentence, and in a comparison 35
2.6.2 What Profitina measures, and what it does not 36
2.6.3 The loop: Scan, Optimize, Monitor 36
2.6.4 Honest expectations 37
2.7 The Machine-Readable Layer: Structured Data, robots.txt and llms.txt 37
2.7.1 Structured data: schema.org in JSON-LD 37
2.7.2 Keep the facts true 38
2.7.3 robots.txt: who may read what 38
2.7.4 The AI crawlers 39
2.7.5 llms.txt: a short map for language models 40
2.8 Lab G6: A JSON-LD, robots.txt and llms.txt Generator 41
2.8.1 What we will build 41
2.8.2 The facts file 41
2.8.3 The generator in Python 42
2.8.4 The three generated files 46
2.8.5 The generator in Java 49
2.8.6 Reading it back: the checker 59
2.8.7 When things are wrong 69
2.9 Profitina in Practice 70
2.10 Common Misunderstandings 71
2.11 Chapter Summary 72
2.12 Exercises 72
2.13 Solutions to the Exercises 73
References 77

CHAPTER 3 What Profitina Does: The Visibility Loop and a User's Journey 80
3.1 Where We Left Off 80
3.2 The Visibility Loop 81
3.2.1 Four verbs, three tabs 81
3.2.2 Why a loop and not a line 82
3.2.3 What is measured, and what is not 82
3.3 Measure: What a Scan Looks At 83
3.3.1 The input 83
3.3.2 Three families of signals 83
3.3.3 Progress you can read 84
3.3.4 The number, and what it means 84

3.4 Explain: From a Number to a List of Fixes 85
3.5 Fix: The Content Engine and the Technical Assistant 85
3.5.1 What the Content Engine drafts 85
3.5.2 An AI draft only when it is better 86
3.5.3 The Technical Assistant 86
3.6 Monitor: History, Schedules and Alerts 86
3.6.1 A weekly trend 86
3.6.2 What "significant" means 86
3.7 Beyond Websites: Researcher and Organization Reports 87
3.7.1 The Researcher Visibility Report 87
3.7.2 Organization Visibility 87
3.7.3 The Academic Citation Scan 88
3.8 One Text, Many Platforms 88
3.9 A User's Journey: Profitina Laundry, Step by Step 88
3.9.1 Step 1: Arrive 89
3.9.2 Step 2: Try without an account 89
3.9.3 Step 3: Register 89
3.9.4 Step 4: Scan 90
3.9.5 Step 5: Optimize 91
3.9.6 Step 6: Monitor 92
3.9.7 Step 7: Participate 93
3.9.8 Step 8: Upgrade, if she wants to 93
3.9.9 Step 9: Exercise her rights 94
3.9.10 The same journey for a researcher 94
3.9.11 One Text, Many Platforms, from the inside 95
3.10 Lab G1: A Mini AI-Readiness Checker 96
3.10.1 What we will build, and whose score it is 96
3.10.2 The sample sites 97
3.10.3 A tiny local server 101
3.10.4 Reading robots.txt by the rules 104
3.10.5 Collecting the facts of a page 110
3.10.6 Eight checks 117
3.10.7 The report and the command line 126
3.10.8 Running the lab: before and after 133
3.10.9 Choosing your own weights 137
3.10.10 When the checker stops 139
3.10.11 What the checker cannot see 139
3.11 Profitina in Practice 139
3.12 Common Misunderstandings 140
3.13 Chapter Summary 141
3.14 Exercises 141
3.15 Solutions to the Exercises 143
References 148

CHAPTER 4 Community by Design 150
4.1 Where We Left Off 150
4.2 Why Build a Community into a Visibility Tool 151
4.2.1 People already live in feeds 151
4.2.2 What Profitina does differently 151
4.3 A Map of the Community 151
4.4 Real Time Without Magic: Polling, WebSockets and Server-Sent Events 153
4.4.1 The problem: the web was built for asking 153
4.4.2 Polling: asking again and again 153
4.4.3 WebSockets: a two-way pipe 154
4.4.4 Server-Sent Events: one long answer 154

4.4.5 Thin events 156
4.5 The Feed, Threads and Quote-Repost 156
4.5.1 A feed is a list; a thread is a tree 156
4.5.2 Reposts and where they belong 156
4.5.3 The fifteen-minute edit window 157
4.6 Channels and the PROFIT-Balance Gate 157
4.6.1 Rooms with a purpose 157
4.6.2 The gate 157
4.6.3 Per-channel leaderboards 158
4.7 Following Channels and the “N new” Counter 159
4.7.1 What a follower wants to know 159
4.7.2 The definition 159
4.7.3 One word, one number 159
4.7.4 Keeping the counter live 160
4.8 Direct Messages and Profiles: Private by Default 160
4.8.1 Direct messages 160
4.8.2 Profiles with per-item consent 160
4.8.3 Neutral handles 160
4.9 The Dashboard as a Cockpit 161
4.10 Instant (Optimistic) Posting 162
4.10.1 Waiting feels longer than it is 162
4.10.2 Show it first, confirm it later 162
4.11 When Real Time Goes Wrong: Three Engineering Lessons 163
4.11.1 Lesson 1: the read path must never call a model 163
4.11.2 Lesson 2: invalidate, do not refetch everything 163
4.11.3 Lesson 3: count your queries 164
4.11.4 Lesson 4 (a bonus): late answers 165
4.12 Lab G2: A Live Feed with Server-Sent Events 165
4.12.1 What we will build 165
4.12.2 The Python server 166
4.12.3 The Java server 171
4.12.4 The browser page 180
4.12.5 The console clients 185
4.12.6 Running it 189
4.12.7 Checking the page in a headless browser 190
4.12.8 What the lab leaves out 191
4.13 Profitina in Practice 191
4.14 Common Misunderstandings 192
4.15 Chapter Summary 192
4.16 Exercises 193
4.17 Solutions to the Exercises 194
References 195

CHAPTER 5 Keeping the Community Safe: Link Safety, Moderation, Fair Play 197
5.1 Where We Left Off 197
5.2 Why a Community Needs Protecting 197
5.2.1 Three numbers 197
5.2.2 Three threats, three defenses 198
5.3 Anatomy of a Link, and Where the Tricks Hide 199
5.3.1 The parts of a URL 199
5.3.2 The usual tricks 199
5.3.3 Hard signals and soft signals 200
5.3.4 Five seconds of checking 201
5.4 Link Safety as Layers 201
5.4.1 The two layers 201

5.4.2 Why the floor comes first 202
5.4.3 Never on the read path 202
5.4.4 A second, quieter layer: showing text as text 203
5.5 Measuring a Screener: The Confusion Matrix 203
5.5.1 Four kinds of outcome 203
5.5.2 Five measures from four counts 203
5.5.3 Profitina's public benchmark 204
5.5.4 What zero out of thirty proves 205
5.5.5 Why the false positive rate matters more than it seems 205
5.6 Moderators: Human Judgment, Narrow Powers 206
5.6.1 What software cannot do 206
5.6.2 Channel-scoped moderators 206
5.6.3 Reporting, and the frozen copy 207
5.6.4 Rules that can be moderated 207
5.7 Fair Play: Keeping Rewards for Humans 208
5.7.1 Rewards attract automation 208
5.7.2 The layers 208
5.7.3 Why layers multiply: a worked illustration 209
5.7.4 What is deliberately not published 209
5.8 Lab G3: Our Own Rule-Based Link Screener 210
5.8.1 What we will build 210
5.8.2 The decision rule 210
5.8.3 How far is "one letter away"? 211
5.8.4 The Python screener 211
5.8.5 The Java screener 216
5.8.6 Trying it on single messages 220
5.8.7 The sample set and the evaluation 221
5.8.8 Reading the results 227
5.8.9 Another view: points on an ROC plot 228
5.8.10 What the lab leaves out 229
5.9 Profitina in Practice 229
5.10 Common Misunderstandings 229
5.11 Chapter Summary 230
5.12 Exercises 230
5.13 Solutions to the Exercises 231
References 233

CHAPTER 6 Local AI: The Content Engine and the Two-Lane Model 234
6.1 Where We Left Off 234
6.2 What a Language Model Does, in Plain Words 234
6.2.1 Predicting the next piece of text 234
6.2.2 Parameters, memory and quantization 235
6.3 Why Local-Only Inference: Privacy, Cost, Control 236
6.3.1 Privacy 236
6.3.2 Cost 236
6.3.3 Control 237
6.3.4 The honest costs of going local 237
6.4 Ollama and the Qwen Family 238
6.4.1 Ollama: a model server on your own machine 238
6.4.2 The Qwen family, as of September 2026 238
6.4.3 Why Profitina still runs Qwen2.5 7B 239
6.5 The Two-Lane Model 240
6.5.1 Two kinds of work 240
6.5.2 Why the lengths matter: a little arithmetic 241
6.5.3 One long job at a time, and an honest queue 241

6.5.4 Work that belongs in neither lane 242
6.6 The Content Engine and the Quality Gate 242
6.6.1 What the Content Engine produces 242
6.6.2 The gate: serve AI only when it is better 242
6.6.3 Reading the public result: 5 of 6 243
6.7 Honest "No Model Available" Behavior 244
6.8 Lab G5: Talking to a Local Model over HTTP 245
6.8.1 What we will build, and where the output comes from 245
6.8.2 A stub that imitates `/api/generate` 245
6.8.3 Starting the stub 258
6.8.4 Looking at the wire 258
6.8.5 The client 259
6.8.6 Two lanes in code 267
6.8.7 A quality gate in code 274
6.8.8 The same code against a real Ollama 280
6.9 Profitina in Practice 280
6.10 Common Misunderstandings 281
6.11 Chapter Summary 282
6.12 Exercises 282
6.13 Solutions to the Exercises 284
References 287

CHAPTER 7 The Feedback Loop: Humans Decide, Models Learn 289
7.1 Where We Left Off 289
7.2 Why Feedback, and Why Humans Decide 289
7.2.1 Models learn from people 289
7.2.2 Why a human must decide 290
7.2.3 The app advises, the user decides 290
7.2.4 A story in five steps 291
7.3 The Five-Step Pipeline 291
7.4 The Life of One Feedback Item 292
7.5 Curating: The Machine Sorts, a Human Decides 293
7.5.1 Why sort before a person looks 293
7.5.2 Finding duplicates with word overlap 293
7.5.3 The human decision 294
7.5.4 Ordering the queue 294
7.6 The Dataset: Preference Pairs in JSON Lines 294
7.6.1 What a training example looks like 294
7.6.2 JSON Lines 294
7.6.3 What must not go into a dataset 295
7.6.4 Versioning and provenance 295
7.7 Train and Measure: Before and After 295
7.7.1 A held-out set 295
7.7.2 Pass rate and regressions 295
7.7.3 Measuring honestly 296
7.7.4 How precise is a pass rate? 296
7.8 Release: The Pipeline Advises, a Human Decides 297
7.9 PROFIT for Improving AI Responses 298
7.9.1 Why reward feedback at all 298
7.9.2 The public earning rules 298
7.9.3 The arithmetic of a capped reward 299
7.9.4 Bounding what one account can earn 300
7.10 Lab: A Mini Feedback Pipeline 300
7.10.1 The scenario and the data 300
7.10.2 The program 301

7.10.3 Running the pipeline 313
7.10.4 Which day is it? UTC versus WIB 315
7.11 Profitina in Practice 316
7.12 Common Misunderstandings 317
7.13 Chapter Summary 318
7.14 Exercises 318
7.15 Solutions to the Exercises 319
References 321

CHAPTER 8 The Business Model 323
8.1 Where We Left Off 323
8.2 What a Business Model Is 324
8.2.1 Freemium, in one picture 324
8.2.2 What Profitina deliberately does not sell 325
8.3 The Market Profitina Serves 325
8.3.1 Who is online 325
8.3.2 How people pay: QRIS 326
8.3.3 What a small business compares the price with 326
8.4 Two Tiers: Free and Pro 327
8.4.1 What Free includes, and why so much 327
8.4.2 What Pro adds 327
8.4.3 The price, per day 328
8.4.4 A rule for plan cards 328
8.5 Why the Agency Tier Was Removed 328
8.5.1 The hidden cost of a tier 329
8.5.2 Simplicity for the customer 329
8.5.3 What happens to agencies now 330
8.6 No Ads, Permanently 330
8.6.1 The conflict of interest 330
8.6.2 The privacy argument 330
8.6.3 The credibility argument 330
8.7 Paying for Pro: QRIS and PROFIT 331
8.7.1 QRIS, step by step 331
8.7.2 Redeeming PROFIT 331
8.7.3 Auto-renew and the reminder 332
8.8 Measure Interest Before Building Payment 332
8.8.1 The zero-purchase finding 332
8.8.2 Why the rule is right 332
8.8.3 A ladder of evidence 333
8.8.4 The same rule elsewhere 333
8.9 Daily Quotas and the 00:00 WIB Rule 334
8.9.1 Whose day? 334
8.9.2 The UTC-date trap 334
8.9.3 The day as a formula 334
8.9.4 How many scans are left 335
8.10 Lab: Plans as Data and a Quota Calculator 335
8.10.1 Design 335
8.10.2 The Python program 336
8.10.3 The Java program 340
8.11 Lab: The Subscription as a State Machine 345
8.11.1 States, events and transitions 345
8.11.2 The Python program 346
8.11.3 The Java program 351
8.12 Lab: Tests That Pin the Rules Down 356
8.12.1 What to test 356

8.12.2 The Python tests 356
8.12.3 The test that caught a real bug 359
8.12.4 The Java tests, with a tiny runner 360
8.13 Profitina in Practice 365
8.14 Common Misunderstandings 365
8.15 Chapter Summary 366
8.16 Exercises 366
8.17 Solutions to the Exercises 367
References 371

CHAPTER 9 Architecture, Reliability, and the Release Gate 372
9.1 Where We Left Off 372
9.2 Reliability, in Plain Words 373
9.2.1 Availability 373
9.2.2 What the nines mean in minutes 373
9.2.3 Reliability includes being right 374
9.3 The Architecture at Box Level 374
9.3.1 The two front doors 374
9.3.2 The edge and the tunnel 375
9.3.3 One host, kept alive by NSSM 375
9.3.4 The application: FastAPI under Uvicorn 375
9.3.5 Real time: Server-Sent Events 376
9.3.6 The database: PostgreSQL, with separation enforced 376
9.3.7 The AI lanes: local only 376
9.3.8 What is deliberately not in the picture 377
9.4 Why One Machine Is Enough, for Now 377
9.5 The Test Suite as a Release Gate 378
9.5.1 The size of the suite 378
9.5.2 Why a gate, and why now 378
9.5.3 Two lessons the suite taught 378
9.6 A Health-Verified, Staged Release 379
9.6.1 The gate that lied 379
9.6.2 The fix: success means the service says so 379
9.6.3 Six steps 379
9.6.4 What the health endpoint reports 380
9.7 The Performance Story: Switching Channels 381
9.7.1 The symptom 381
9.7.2 Three root causes 381
9.7.3 The fix and the result 382
9.7.4 The rule, and the race 382
9.8 Scheduling Re-scans: A Priority Queue 383
9.8.1 The problem 383
9.8.2 Cadence and how many runs to expect 384
9.8.3 Three details that separate a toy from a tool 384
9.9 Lab G4, Part 1: The Scheduler 385
9.9.1 The Python program 385
9.9.2 The Java program 389
9.9.3 The tests 393
9.10 Lab G4, Part 2: The Health Endpoint 397
9.10.1 The Python program 397
9.10.2 The Java program 400
9.11 Lab G4, Part 3: The Release Gate 403
9.11.1 How the lab maps onto the real release 403
9.11.2 The Python gate 404
9.11.3 The Java gate 409

9.12 Profitina in Practice 415
9.13 Common Misunderstandings 415
9.14 Chapter Summary 416
9.15 Exercises 416
9.16 Solutions to the Exercises 417
References 422

CHAPTER 10 The Marketing Site as a GEO Case Study 423
10.1 Where We Left Off 423
10.2 Two Surfaces, One Brand 424
10.2.1 Why a separate marketing site at all 424
10.2.2 Why WordPress 424
10.2.3 The stack, from the visitor inward 425
10.2.4 What lives where 426
10.3 A Bilingual Theme Done Properly 426
10.3.1 Two languages, two sets of pages 426
10.3.2 The `$id` bug: a lesson in shared names 427
10.4 robots.txt from First Principles 427
10.4.1 What Chapter 2 established 427
10.4.2 From lines to groups 427
10.4.3 Choosing the groups: merging and the fallback 428
10.4.4 Choosing the rule: longest match, allow on a tie 429
10.4.5 Wildcards 429
10.4.6 When the file cannot be read 429
10.4.7 Reading and access are different things 430
10.5 The robots.txt Lesson: When the Served File Is Not Your File 430
10.5.1 What happened 430
10.5.2 Why prepending is enough to block 431
10.5.3 Defaults move, too 432
10.5.4 The file profitina.com serves today 432
10.5.5 The general lesson 432
10.6 llms.txt: What It Is, and What It Is Not 433
10.6.1 The proposal 433
10.6.2 Adoption versus use 433
10.6.3 Profitina's position: cheap hygiene, no promises 434
10.7 Two Cache Layers and the Purge Order 434
10.7.1 Why caches exist 434
10.7.2 Freshness in HTTP 435
10.7.3 What each layer caches by default 435
10.7.4 The purge order 436
10.7.5 Choosing lifetimes 437
10.8 Structured Data on the Marketing Site 437
10.8.1 What the pages carry 437
10.8.2 After May 2026: FAQ markup without FAQ results 438
10.8.3 Freshness belongs in the data 438
10.8.4 Three faults that make structured data useless 438
10.9 Lab: A GEO Hygiene Auditor in Python and Java 438
10.9.1 What we will build 438
10.9.2 The sample sites and their server 440
10.9.3 A robots.txt reader that follows RFC 9309 449
10.9.4 The auditor 452
10.9.5 Running the auditor 471
10.9.6 A surprise in Python's standard library 473
10.9.7 Pointing the auditor at a real site 474
10.10 Profitina in Practice 474

10.11 Common Misunderstandings 475
10.12 Chapter Summary 475
10.13 Exercises 476
10.14 Solutions to the Exercises 477
References 480

CHAPTER 11 Privacy and Compliance by Design (UU PDP) 482
11.1 Where We Left Off 482
11.2 UU PDP No. 27/2022 in Plain Language 483
11.2.1 One law, sixteen chapters 483
11.2.2 What is personal data? 483
11.2.3 The principles 483
11.2.4 Lawful bases 484
11.2.5 The rights of the data subject 484
11.2.6 The controller's duties 484
11.2.7 Sanctions 485
11.3 Where the Law Stands in 2026 485
11.3.1 PP 33/2026, the implementing regulation 485
11.3.2 The supervisory agency 486
11.3.3 What this means for a small team 486
11.4 Privacy by Design as an Engineering Method 486
11.4.1 The idea 486
11.4.2 Minimize first 487
11.4.3 Write it down: the record of processing 487
11.4.4 Private by default, everywhere 487
11.4.5 One home for the legal text 487
11.5 Passwords: Why Argon2id 488
11.5.1 Never store the password 488
11.5.2 Why a *fast* hash is the wrong one-way function 488
11.5.3 Salt 489
11.5.4 Argon2id 489
11.5.5 A stored hash names its own recipe 489
11.6 Consent That Means Something 490
11.6.1 From Pasal 22 to a checkbox 490
11.6.2 Designing the consent log 490
11.6.3 Withdrawal 491
11.6.4 Children 491
11.7 Login Rate Limiting 491
11.7.1 The attacks 491
11.7.2 Fixed windows and sliding windows 491
11.7.3 Four details that matter 492
11.8 Rights That Are Actually Reachable 493
11.8.1 From articles to buttons 493
11.8.2 An honest correction 493
11.8.3 Export: everything, and only mine 494
11.8.4 Erasure: purge, and leave a tombstone only where others need one 494
11.8.5 When something goes wrong 495
11.9 HTML Legal Pages and a Single Source of Truth 495
11.10 Lab G7: Consent Log, Export and Erasure 495
11.10.1 What we will build 495
11.10.2 Password hashing 496
11.10.3 The sliding-window limiter 500
11.10.4 The application 501
11.10.5 The demo 515
11.10.6 The tests 523

11.11 Profitina in Practice 532
11.12 Common Misunderstandings 533
11.13 Chapter Summary 534
11.14 Exercises 534
11.15 Solutions to the Exercises 535
References 540

CHAPTER 12 PROFIT: The Utility Coin on Solana 541
12.1 Where We Left Off 541
12.2 What a Utility Coin Is, and What It Is Not 542
12.2.1 Hollow pride 542
12.2.2 Four things PROFIT is not 542
12.2.3 The Indonesian frame 543
12.3 Solana in Plain Words 543
12.3.1 A shared notebook that nobody can quietly edit 543
12.3.2 Accounts, programs, transactions 544
12.3.3 Three clusters, one of them for practice 544
12.3.4 The state of the network in 2026 544
12.4 Tokens on Solana: Mints, Accounts and Authorities 545
12.4.1 The mint and the token accounts 545
12.4.2 Decimals, and why amounts are whole numbers 546
12.4.3 The two authorities 549
12.5 PROFIT's Public Facts 549
12.6 Emission That Decays 550
12.6.1 The problem a decaying schedule solves 550
12.6.2 The public rule 551
12.6.3 The hidden power law 551
12.6.4 Lab G8, step 2: reproducing the table 553
12.6.5 From the schedule to real rewards 555
12.7 Four Ways PROFIT Leaves: The Burns 555
12.7.1 What burning means 555
12.7.2 The four burns 556
12.8 Supply as a Geometric Series 556
12.8.1 A model simple enough to solve 556
12.8.2 Unrolling the recurrence 557
12.8.3 The steady state and how fast it is reached 557
12.8.4 Lab G8, step 3: a ledger in micro-PROFIT against the closed form 558
12.9 When Does Burning Overtake Emission? 564
12.9.1 Per user, the question becomes simple 564
12.9.2 Lab G8, step 4: closed form, bisection and a toy oracle 564
12.10 The Oracle: A Floor, Not a Price 569
12.10.1 What the oracle computes 569
12.10.2 What moves the floor, and what does not 569
12.11 Buying, Spending and Transferring: The Guardrails 570
12.11.1 Buying through QRIS 570
12.11.2 Spending on real services 571
12.11.3 Transferring between users 571
12.11.4 Administrator grants, in public 571
12.12 One Tier of Referral, and Why Not More 572
12.12.1 The rule 572
12.12.2 Why a second tier would be a different thing 572
12.13 Live Today, and on the Roadmap 572
12.14 Profitina in Practice 573
12.15 Common Misunderstandings 574
12.16 Chapter Summary 575

12.17 Exercises 575
12.18 Solutions to the Exercises 576
References 578

CHAPTER 13 The Profitina Group and the Road Ahead 580
13.1 Where We Left Off 580
13.2 One Umbrella, Several Houses 581
13.2.1 The shape of the group 581
13.2.2 Six shared layers 581
13.2.3 One rule for sharing 582
13.3 Profitina Capital: A Quantitative Desk 582
13.3.1 What it is 582
13.3.2 Five measures on ten days 583
13.3.3 What Capital teaches 584
13.4 Profitina TTM: From Text to a Short Film 584
13.4.1 The pipeline 584
13.4.2 Painting in two lanes 585
13.4.3 Fail loudly 585
13.5 Profitina AI: A Local-First Assistant 585
13.5.1 What it does 585
13.5.2 Never stop silently 586
13.6 Profitina Laundry: The Teaching Ground 586
13.7 The Principles That Bind Them 587
13.7.1 Local-first AI 587
13.7.2 The app advises; the user decides 588
13.7.3 Tests as a gate 588
13.7.4 And one more: tell the truth about what happened 588
13.8 Lab G9: Checking a Researcher Record with OpenAlex 589
13.8.1 The feature behind the lab 589
13.8.2 OpenAlex in brief 589
13.8.3 What the checker checks 589
13.8.4 Step 1: counting spellings with a set 591
13.8.5 Step 2: the checker 593
13.8.6 Running it, and what happened to the network 606
13.8.7 Step 3: a local stand-in for the API 608
13.8.8 Taking the checker further 612
13.9 What Comes Next 613
13.9.1 The road behind 613
13.9.2 The road ahead 613
13.9.3 What will not change 614
13.9.4 The labs, one last time 614
13.10 Profitina in Practice 614
13.11 Common Misunderstandings 615
13.12 Chapter Summary 616
13.13 Exercises 616
13.14 Solutions to the Exercises 617
13.15 A Closing Word 618
References 619

List of Figures

Figure 1.1 The first twelve weeks of the current code base on a time axis. The amber point is the production launch of 17 July 2026. 6
Figure 1.2 The thirteen chapters in four groups, with the hands-on lab of each chapter in italics. The labs marked G1 to G9 are the book's numbered programs. 10
Figure 1.3 The bench used throughout the book. You work in VS Code's editor and terminal; two extensions teach VS Code about Python and Java; the terminal finds Python and the JDK through the operating system's PATH. 11
Figure 2.1 The same question on a search results page (left) and in an answer engine (right). On the left the person scans a ranked list and chooses; on the right the engine has already chosen, and names only one or two sources. 31
Figure 2.2 Three measurements from **Table 2.1**: (a) the share of Google visits that clicked a result, with and without an AI summary (Pew, March 2025); (b) the share of Google searches showing an AI Overview a year apart; (c) visits to Google AI Mode (both Similarweb, reported July 2026). 32
Figure 2.3 How a crawler applies `robots.txt` under RFC 9309: choose the groups that name it (or the `*` group), then let the longest matching rule decide, with ties going to `Allow`. 39
Figure 2.4 Lab G6. One facts file feeds a generator that writes three files; a separate checker reads them back and reports; a person reviews the result before anything is published. 41
Figure 3.1 The visibility loop: measure, explain, fix and monitor, grouped into the three tabs of the application, Scan, Optimize and Monitor (Pindai, Optimasi, Pantau). 81
Figure 3.2 The nine steps of a user's journey. Steps 1 and 2 need no account; steps 4 to 6 are the SOM/POP loop, and the dashed arrow shows that the loop repeats. 89
Figure 3.3 Registration, as a wireframe. (1) The consent box starts empty; (2) only an email and a password are asked for; (3) a code sent by email activates the account. 90
Figure 3.4 The Scan tab, as a wireframe. (1) Location and customer questions are filled in automatically for the owner to check; (2) when the category is uncertain, the system says "not sure yet" and asks; (3) progress is shown with its sub-step and the time remaining. 90
Figure 3.5 A scan result, as a wireframe with illustrative values. (1) The readiness estimate; (2) the link to the method; (3) the prioritized fixes; (4) the way into the Content Engine; (5) feedback on the scan. 91
Figure 3.6 The Optimize tab, as a wireframe. (1) An answer-first draft with the entity named at once; (2) the quality gate's verdict; (3) the matching JSON-LD; (4) the Technical Assistant's crawler choices; (5) the owner publishes on her own site. 92
Figure 3.7 The Monitor tab, as a wireframe with illustrative values. (1) The weekly trend; (2) an alert for a significant change; (3) the re-scan cadence of each plan; (4) CSV export, a Pro feature. 93
Figure 3.8 The Account & Personal Data panel, as a wireframe. (1) Export as JSON; (2) the consent history; (3) permanent deletion, which only the user can start; (4) these rights are free on every plan. 94
Figure 3.9 The Researcher Visibility Report, as a wireframe. (1) A fictional name is not found, and no score is given; (2) for a real record, the report lists closeable gaps; (3) only bibliographic records are used. 95
Figure 3.10 One Text, Many Platforms, as a wireframe. (1) One platform is chosen at a time; (2) the queue position and wait are shown; (3) the length is checked against the platform's limit; (4) Share to X opens X in the user's own account. 95
Figure 3.11 The architecture of lab G1. A tiny server publishes a sample site on 127.0.0.1; `gather()` fetches at most four resources, `RobotsRules` and `PageFacts` turn them into a `Site`, eight checks score it, and `report()` prints the result. 96
Figure 3.12 Points of each check for the two sample sites, taken from the real output of `demo.py`: before the fixes (score 13) and after them (score 100). 137
Figure 4.1 The community at box level. One live event bus feeds every part of the public square and the private rooms; the safety layer of Chapter 5 screens what is written before it is shown. 152

Figure 4.2 Three ways for news to reach a browser. Polling asks every five seconds and often gets nothing; Server-Sent Events keep one response open and push each post as it happens; WebSockets upgrade the connection so that both sides can send at any time. 153
Figure 4.3 A thread is a tree of replies. A quote-repost (#5) points back to the post it quotes and, unless its writer chooses otherwise, inherits the original's channel; a plain repost shared deliberately to General (#6) goes there instead. 156
Figure 4.4 Creating a channel. A balance of at least 50 PROFIT is required but never deducted; at most two new channels per day and five active channels per member. 158
Figure 4.5 The "N new" counter. Only top-level posts after the last-seen marker are counted, so six new items of which three are replies give the badge "3 new". 159
Figure 4.6 Wireframe of the dashboard cockpit. One request fills a three-level priority strip and three cards; every item links to its own section. 161
Figure 4.7 The life of an optimistic post. It is shown at once as pending; a success confirms it with its real number, and a failure removes it and gives the text back with the reason. 162
Figure 4.8 What a page does with a thin event. Events for other channels only invalidate a cache and update a counter; an event for the channel on screen fetches one post; an event for the member's own optimistic post is ignored. 164
Figure 4.9 One feed page of 30 posts before and after the September 2026 fixes (log scale; a value of zero is drawn as a short stub). Time fell from 868 ms to 11 ms and queries from 335 to 12. 164
Figure 4.10 A stale-response race. Without a guard, a slow reply for channel A overwrites channel B; with a request sequence number, the late reply is recognized and thrown away. 165
Figure 4.11 The lab. Either server keeps a list of posts and one queue per listener; browsers and console clients send ordinary requests and receive thin events over one open stream. 166
Figure 4.12 The lab page in headless Chromium after the automated check. Marker 1: the "1 new" badge of a channel not on screen; 2: a reply, indented; 3: the connection status; 4: the composer, cleared at once when a post is sent. 191
Figure 5.1 Three measures of the threat environment in 2026: phishing attacks observed worldwide in one quarter, scam reports received by Indonesia's Anti-Scam Centre since it opened, and the share of web traffic that is automated. 198
Figure 5.2 A link that hides its real host. Everything before `@` is user information, a decoy; the host is `receipt-view.click`. The bottom line lists three other places where tricks hide in a host. 199
Figure 5.3 Link safety as layers. Both layers run once, on the write path, and the verdict is stored with the post; rendering the feed only reads the stored verdict and never calls a model. 202
Figure 5.4 Two confusion matrices. Left: Profitina's published benchmark for its rule floor (27 of 28 scams caught, 0 false positives on 30 normal texts). Right: the lab screener of Section 5.8 on this book's own 60 samples with $k = 2$. 204
Figure 5.5 What a channel moderator can and cannot do. Three powers inside one channel, every action logged and the poster notified; no access to PROFIT, private data, direct messages or other channels. 206
Figure 5.6 Fair play at box level. A contribution earns a small reward only if it comes from a verified human, stays within the daily cap, is not a duplicate and is judged useful; abuse found later is clawed back and burned. 208
Figure 5.7 The lab screener. Links are extracted and split, eleven textbook rules are applied, and the verdict is BLOCK if any hard rule fires or at least k soft rules fire. 210
Figure 5.8 Precision, recall and $F1$ of the lab screener as the soft threshold k varies from 1 to 5. Lower k catches more scams and blocks more honest messages; from $k = 3$ upwards there are no false positives, but most scams get through. 228
Figure 6.1 Download sizes of Qwen models in Ollama's library, in gigabytes, as listed on September 24, 2026 [15], [5], [14]. The two teal bars are the models behind Profitina's quality lane and fast lane. 239
Figure 6.2 The two-lane model. Long jobs go to the quality lane, short jobs to the fast lane, both served by Ollama on the same machine. Each lane has its own honest failure path. 240
Figure 6.3 A quality gate as built in this chapter's lab. A draft must pass every hard check and then score strictly higher than the template; when no model is available the user is told so. 243

Figure 6.4 A streaming call to `/api/generate`. The client sends one JSON request; the server answers with one JSON object per line, the last one marked `"done": true` and carrying the token counts and durations. 259
Figure 7.1 The five-step feedback pipeline. Dark boxes are the steps; grey boxes show what the giver sees; the amber notes mark the two places where a human decides. After a release, the loop starts again with new feedback. 291
Figure 7.2 The life of one feedback item. Dashed moves are automatic and explainable; amber moves are human decisions. Only accepted items can ever be shipped. 292
Figure 7.3 The lab's held-out questions before and after the dataset. Six questions are fixed, one keeps passing, and E2 regresses: the pass rate rises from 2/8 to 7/8, and the pipeline still advises a hold. 296
Figure 7.4 A lab daily cap of 3 rewarded items per WIB day for user u1. F10 at 23:30 WIB hits the cap; F11 at 00:10 WIB falls on a new WIB day. A cap keyed on the UTC date would reset only at 07:00 WIB. 300
Figure 8.1 The market in 2026. (a) Internet penetration by region from the APJII survey, with the national figure of 81.72% dashed; the axis starts at 60%. (b) Internet users (APJII, 2026) beside QRIS users and QRIS merchants (Bank Indonesia, June 2026). 326
Figure 8.2 The tier structure before and after 6–7 August 2026. Agency's three features had been shown to work; the tier was removed for simplicity, and the permanent no-ads stance was kept. 329
Figure 8.3 Measure interest before building payment, as a ladder. Each step is built only when the one below it shows real demand; the August 2026 finding of zero PROFIT purchases kept the platform on the lower steps. 333
Figure 8.4 One day on two clocks. The correct reset is at 00:00 WIB; code that uses the UTC date resets at 07:00 WIB instead, leaving users with yesterday's exhausted quota for the first seven hours of their day. 334
Figure 8.5 The subscription state machine of the lab. Teal arrows lead to Pro, red arrows return to Free, and the loop on Pro is the reminder and the auto-renewal; every state also accepts the clock's tick. 345
Figure 9.1 Profitina's architecture at box level. Browsers and the marketing site reach one Windows 11 host through the Cloudflare edge and an outbound-only tunnel; inside, NSSM keeps Uvicorn and FastAPI alive, with PostgreSQL and local AI beside them. Answer engines are measured targets, never called. 374
Figure 9.2 The six-step, health-verified release. Any failing test or a payload that changed during the gate refuses the release with production untouched; a new version that does not answer the health check in time is rolled back. 380
Figure 9.3 Switching channels in the community feed, before and after the September 2026 fix: time for a 30-post page, database queries, new connections and AI calls per page. 382
Figure 9.4 The stale-response race. A late reply to an earlier request must not overwrite the reply to the latest one; a sequence number on each request, checked when the reply arrives, discards the stale reply. 383
Figure 9.5 A min-heap of scheduled scans ordered by due time. The root is the next job to run; the dashed entry is a stale one left behind by a reschedule, skipped when it reaches the root. 384
Figure 10.1 The marketing site's stack. A request passes the Cloudflare edge (cache layer 2) and the host's LiteSpeed Cache (cache layer 1) before WordPress and its bilingual theme build a page; the application is reached only through the embedded widget and the legal links. 425
Figure 10.2 A managed block prepended at the edge. The origin's `*` group still allows everyone, but a crawler named in the managed block now has a group of its own and never looks at `*`; a crawler that is not named still falls back to `*`. 431
Figure 10.3 Adoption versus use of `llms.txt` in 2026. About one in eleven top sites publishes the file, but in May 2026 almost none of the files received a single request. 434
Figure 10.4 Purging two cache layers. In the wrong order, a visitor who arrives between the purges makes the edge refill itself from the still-stale inner cache, and the stale copy lives on for a full edge lifetime; in the right order, the stale window ends with the second purge. 436
Figure 10.5 The auditor's requests for one site. Two comparisons tell a stale cached copy apart from a file that a layer rewrites; the last request fetches a fresh home page to compare `dateModified`. 439

Figure 11.1 UU PDP from enactment to the implementing regulation, with the milestones of Profitina's own compliance work. The agency's date is an expert's estimate, not a schedule. 486
Figure 11.2 Storing a password. A random salt and the cost settings go into a slow key-derivation function; the stored string records the recipe, the cost, the salt and the result, so that login can recompute it and an outdated recipe can be upgraded. 489
Figure 11.3 A sliding-window limiter with the lab's values. Three failures at 09:00:05, 09:00:10 and 09:00:15 block the fourth attempt at 09:00:20, even with the right password; the block lifts when the first failure leaves the window at 09:01:05. 492
Figure 11.4 The life of personal data at Profitina: minimal collection, protected use, rights exercised from one panel, erasure followed by the production purge, and the breach-notification duty that stands behind it all. 493
Figure 11.5 Erasure in lab G7. The user row goes; the post that another user replied to becomes an anonymous shell; the consent events keep only an anonymous identifier, a version and a time, and a `withdrawn` event is added. 494
Figure 12.1 A token on Solana: one mint account describes the token (supply, decimals and two authorities), each holder has a token account pointing to the mint, and a metadata account gives it a name and logo. The Token Program enforces the rules for every change. 545
Figure 12.2 Daily emission under a linear schedule and under the whitepaper's decaying schedule, on logarithmic axes. The whitepaper's three example points lie on the decaying line; at a million users the gap is a factor of 64. 552
Figure 12.3 How PROFIT enters and leaves circulation. Earning and QRIS purchases add to circulation; four burns remove PROFIT permanently; the oracle floor divides the Rupiah reserve by what circulates. 555
Figure 12.4 Circulating supply under a constant emission of 3,125 PROFIT per day and three daily burn rates, from the integer ledger of Lab G8. Each curve approaches its own steady state E/b; a higher burn rate gives a lower steady state that is reached sooner. Illustrative values only. 563
Figure 12.5 Per-user emission $m(u)$ falls as the community grows; a constant per-user burn β is a horizontal line. Burning overtakes emission to the right of each crossing $u*$. Illustrative values of β. 569
Figure 13.1 The Profitina Group. Shared layers live at the top and are copied down into each product, never back up. Profitina Laundry, the teaching ground, borrows only ideas and patterns, never code or data. 581
Figure 13.2 The three principles as one loop. Data goes to a local model; the model gives advice with reasons; a person decides; tests stand between every change and its release, which counts as done only when the health check confirms it. 587
Figure 13.3 Lab G9. A name is searched; only an exact match after normalization passes the guard, and anything else is reported as not found with no score. When the network is blocked, a saved sample or a local stand-in answers the same three calls. 590
Figure 13.4 The road behind (solid) and the road ahead (dashed). Everything to the left of September 2026 has happened; everything to the right is planned and gated. 613

List of Tables

Table 1.1 Profitina's dated milestones from 3 July to 24 September 2026, with the chapter of this book that treats each one. The last row is a plan, not a fact. 5
Table 1.2 Five kinds of reader, what each will find in this book, and which chapters to read first. 9
Table 1.3 Installing VS Code on each operating system. 12
Table 1.4 Installing JDK 25 on each operating system. 13
Table 1.5 Installing Python on each operating system. 13
Table 1.6 The VS Code extensions used in this book. 13
Table 1.7 The verification commands, what each should print, and the usual fix when it does not. 14

Table 2.1 Measurements of the shift from lists to answers, 2025–2026, with the date and source of each. “M” means million. 32
Table 2.2 Selected results of the GEO paper on the position-adjusted word count: the best three methods and keyword stuffing against the unmodified baseline [2]. 35
Table 2.3 SEO and GEO compared. The foundations are shared; what differs is who chooses the source and what counts as success. 36
Table 2.4 Crawler names used in lab G6’s `robots.txt`, with each company’s own description of the crawler’s job. 40
Table 3.1 The four verbs of the loop and the three tabs of the application: SOM in English, POP in Indonesian. 82
Table 3.2 The three families of observable signals behind a readiness estimate, and the public standards through which each can be observed. 84
Table 3.3 The two plans as described by the application’s plan cards in September 2026. 94
Table 3.4 The eight checks of lab G1 and their rules. The thresholds are teaching choices of this book. 96
Table 4.1 The parts of Profitina’s community and when each reached production (dates from the public whitepaper and the release history). 152
Table 4.2 Polling, WebSockets and Server-Sent Events compared. 155
Table 4.3 The rules for direct messages and the reason for each. 160
Table 4.4 The lab server’s interface (identical in Python and Java). 166
Table 5.1 Common tricks in malicious links, with made-up examples. 200
Table 5.2 The lab screener’s eleven rules. The lists behind them are this book’s own examples and are meant to be edited. 210
Table 5.3 The edit-distance table for `pay` and `pya`, filled row by row with Equation (5.10); the answer is in the bottom-right cell. 211
Table 5.4 A few of the 60 labeled samples (30 scam, 30 normal) in `data/samples.tsv`. 221
Table 6.1 Qwen generations relevant to local inference, as of September 24, 2026. “MoE” (mixture of experts) models such as 30B-A3B have 30 billion parameters in total but use only about 3 billion for each token. 239
Table 6.2 The quality lane and the fast lane in the September 2026 release. 241
Table 7.1 The states of a feedback item in this chapter’s lab, who moves an item into each state, and what the giver sees. 292
Table 7.2 Public PROFIT earning rules (whitepaper v24, September 2026). Amounts are further multiplied by the decaying emission multiplier and by an AI usefulness judgment. 298
Table 7.3 The eleven lab feedback items in `submissions.tsv`. Times are in UTC; add seven hours for WIB. 301
Table 8.1 The two tiers as the plan cards describe them in the September 2026 release. 327
Table 8.2 What is live and what is not, for everything in the business model that involves payment. 331
Table 8.3 The transition table of the subscription machine. Any event not listed for the current state is refused with an error. 346
Table 8.4 Feature-and-tier combinations and directed tier transitions for $F = 12$. 370
Table 9.1 Allowed down-time at common availability targets, from Equation (9.2). 373
Table 9.2 The components of **Figure 9.1**, the choice made for each, and the main reason. 377
Table 9.3 The automated suite that gates every release, at the time of writing (24 September 2026). 378
Table 9.4 How the lab gate imitates each step of the real release. 403
Table 10.1 The two public surfaces of the Profitina brand and what each one is responsible for. 426
Table 10.2 How RFC 9309 tells crawlers to treat each kind of response to a request for `robots.txt`. Google additionally treats a 429 like a server error. 430
Table 10.3 The HTTP caching directives and headers the lab’s auditor reads, as RFC 9111 defines them for shared caches. 435
Table 10.4 The four sample sites of the lab. Two of them also expose their origin on a second port, so that the auditor can compare what visitors receive with what the origin holds. 439

Table 11.1 Measured speed of four ways to hash a password on one CPU core of the machine used for this book (September 2026). A real attacker is much faster with specialized hardware; the ratios are what matter. 488
Table 11.2 Data-subject rights under UU PDP and the self-service feature, or channel, through which a Profitina user exercises each one. 493
Table 12.1 Solana's three public clusters, as described in the official documentation in September 2026. 544
Table 12.2 PROFIT's public facts, from the public whitepaper v24 and the app, as of September 2026. 550
Table 12.3 The whitepaper's illustrative emission table: the multiplier halves for every tenfold growth of users. 551
Table 12.4 Earning actions, base rewards and caps as listed in the public whitepaper. Every base reward is multiplied by the emission multiplier and by the usefulness score; daily caps reset at 00:00 WIB. 555
Table 12.5 What is live and what is on the roadmap for PROFIT, as of September 2026. 573
Table 13.1 The labs of the book, the idea each teaches, and the principle it serves best. 614

List of Listings

Listing 1.1 `check_setup.py`: reports the operating system, the Python that runs the program, and which tools of the bench the PATH can reach. 14
Listing 1.2 `CheckSetup.java`: the same report in Java, with a small hand-written search of the PATH. 16
Listing 1.3 `hello.py`: the first program; it proves that the editor, the terminal and Python work together. 17
Listing 1.4 `Hello.java`: the first program in Java. 17
Listing 1.5 `milestones.py`: reads the milestones, prints their distance from the launch, and applies Equation (1.1) to the number of days live. 19
Listing 1.6 `Milestones.java`: the same table in Java, with `LocalDate` and `ChronoUnit.DAYS`. 20
Listing 1.7 `sol_countdown.py`: days, weeks and weekdays between two dates. 25
Listing 1.8 `SolCountdown.java`: the same computation with `LocalDate` and `ChronoUnit`. 26
Listing 2.1 `business.json`: the facts that Profitina Laundry's owner maintains, the single source for all three generated files. 41
Listing 2.2 `geo_generator.py`: reads `business.json`, validates it, and writes `jsonld.html`, `robots.txt` and `llms.txt` into a folder. 42
Listing 2.3 `MiniJson.java`: a minimal JSON parser and pretty-printer for this lab, with error positions reported as line and column. 49
Listing 2.4 `GeoGenerator.java`: the same generator in Java, using `MiniJson` and ordered maps. 55
Listing 2.5 `geo_check.py`: parses the JSON-LD, applies `robots.txt` to a list of crawlers and paths, and checks the structure of `llms.txt`. 59
Listing 2.6 `Robots.java`: a small `robots.txt` reader implementing RFC 9309's group selection and longest-match rule (without the `*` and `$` wildcards) 63
Listing 2.7 `GeoCheck.java`: the same checks in Java, using `MiniJson` and `Robots`. 65
Listing 2.8 `sol_rule_order.py`: the first-match and longest-match readings of one `robots.txt` group, side by side. 74
Listing 2.9 `SolRuleOrder.java`: the same comparison in Java. 75
Listing 3.1 `sites/before/robots.txt`: a panel-managed file that blocks two AI crawlers and one other crawler by name, and allows everyone else. 97
Listing 3.2 `sites/before/index.html`: the home page before the fixes. The title is "Home", there are two `h1` headings, the JSON-LD has a trailing comma, and the business name appears only in the footer. 97

Listing 3.3 `sites/after/robots.txt`: every crawler may read the public pages; `/admin/` and `/api/` are kept out, `CCBot` is blocked by choice, and the sitemap is named. 99
Listing 3.4 `sites/after/llms.txt`: an H1 title, a summary, and a list of the site's main pages. 99
Listing 3.5 `sites/after/index.html`: the home page after the fixes, with an answer-first opening, a clean heading outline and one valid JSON-LD graph. 99
Listing 3.6 `serve_site.py`: a static file server for one sample site, with one log line per request. 101
Listing 3.7 `ServeSite.java`: the same server with `com.sun.net.httpserver`, including the path-traversal guard. 102
Listing 3.8 `robots_rules.py`: parsing `robots.txt` into groups and deciding by status code, group choice and longest match. 105
Listing 3.9 `RobotsRules.java`: the same parser and decision procedure in Java, with records for rules, groups and decisions. 107
Listing 3.10 `page_facts.py`: an event-based collector of titles, meta tags, headings, JSON-LD blocks and visible words, built on `html.parser`. 110
Listing 3.11 `PageFacts.java`: the same collector in Java, with a small regular-expression tokenizer and raw-text handling for `script` and `style`. 112
Listing 3.12 `checks.py`: the eight checks. Each takes a `Site` and returns points, evidence and advice. 117
Listing 3.13 `Checks.java`: the `Site` and `CheckResult` records, the `Check` interface and the eight checks. 121
Listing 3.14 `readiness.py`: fetching politely, running the checks, printing the report and computing the teaching score. 126
Listing 3.15 `Readiness.java`: the same fetching, reporting and scoring with `java.net.http.HttpClient`. 129
Listing 3.16 `demo.py`: serve both sample sites on ports 7030 and 7031, check them, and always stop the servers. 133
Listing 3.17 `Demo.java`: the same demonstration in Java. 134
Listing 3.18 `robots_compare.py`: the RFC 9309 decisions of `RobotsRules` beside those of `urllib.robotparser`. 143
Listing 3.19 `RobotsCompare.java`: the RFC 9309 decisions for the same six cases. 144
Listing 3.20 `robots_patterns.py`: matching rule paths with `*` and a final `$`. 146
Listing 3.21 `RobotsPatterns.java`: the same matching in Java, with `Pattern.quote` for the ordinary parts. 147
Listing 4.1 `feed_server.py`: a live-feed server with Server-Sent Events, standard library only. 166
Listing 4.2 `FeedServer.java`: the same server with `com.sun.net.httpserver`, using one blocking queue per listener. 171
Listing 4.3 `MiniJson.java`: just enough JSON for the lab, compatible with Python's output. 177
Listing 4.4 `feed.html`: the live page with channel badges, optimistic posting, one-item refetch and a stale-reply guard. 180
Listing 4.5 `feed_client.py`: listening to the stream and posting, in Python. 185
Listing 4.6 `FeedClient.java`: the same client with `java.net.http.HttpClient` and `BodyHandlers.ofLines()`. 186
Listing 5.1 `link_screener.py`: a rule-based link screener with eleven textbook rules and the decision rule of Equation (5.9). 212
Listing 5.2 `LinkScreener.java`: the same screener in Java, producing byte-identical output. 216
Listing 5.3 `evaluate.py`: the confusion matrix, five measures, the list of mistakes and a sweep over k. 221
Listing 5.4 `Evaluate.java`: the same evaluation in Java. 223
Listing 6.1 `stub_ollama.py`: a stand-in for Ollama's `/api/version`, `/api/tags` and `/api/generate`, with streaming as newline-delimited JSON. 245
Listing 6.2 `StubOllama.java`: the same stub with `com.sun.net.httpserver`; `sendResponseHeaders(200, 0)` selects chunked streaming. 249
Listing 6.3 `MiniJson.java`: a minimal recursive-descent JSON reader and a writer, enough for this lab. 254

Listing 6.4 `ollama_client.py`: a standard-library client for Ollama's HTTP interface, with one honest exception for every way a model can be missing. 259
Listing 6.5 `OllamaClient.java`: the same client with `java.net.http.HttpClient`; streaming uses `BodyHandlers.ofLines()` 262
Listing 6.6 `two_lanes.py`: a quality lane that runs one long job at a time and reports queue positions, a fast lane for labels, and a rule floor underneath. 267
Listing 6.7 `TwoLanes.java`: the two-lane router in Java, with `record`, `ReentrantLock` and threads. 270
Listing 6.8 `quality_gate.py`: hard checks, a soft score, and a strict comparison against a correct template 274
Listing 6.9 `QualityGate.java`: the same gate in Java; `show` reproduces Python's `textwrap.fill` so both print identical lines. 276
Listing 6.10 `json_label.py`: ask for JSON, validate the shape and the allowed values, retry once, then hand over to a human. 284
Listing 6.11 `JsonLabel.java`: the same validation and retry in Java, using `MiniJson`. 285
Listing 7.1 `feedback_pipeline.py`: submit, curate with a human decision, export a JSONL dataset, reward under a WIB daily cap, measure before and after, and wait for a human release decision. 301
Listing 7.2 `FeedbackPipeline.java`: the same pipeline in Java, with records, `java.time` for WIB, integer micro-units, and a hand-written JSON Lines writer. 306
Listing 7.3 `day_key.py`: the UTC date and the WIB date of four instants. 315
Listing 7.4 `DayKey.java`: the same comparison with `OffsetDateTime` and `ZoneOffset.ofHours(7)` 315
Listing 8.1 `tiers.py`: the plans as data, a quota book that counts scans per WIB day, same-day top-ups, and the UTC-date bug for comparison. 336
Listing 8.2 `Tiers.java`: the same plans, quota book and demonstration in Java, using `java.time`. 340
Listing 8.3 `subscription.py`: the subscription state machine, with QRIS orders, verification, PROFIT redemption, reminders and auto-renewal. 346
Listing 8.4 `Subscription.java`: the same state machine in Java, with an `EnumMap` as the transition table. 351
Listing 8.5 `test_tiers.py`: eight tests for the plan table, the WIB day, the daily limit, top-ups, monitors and history. 357
Listing 8.6 `test_subscription.py`: ten tests for the subscription state machine. 358
Listing 8.7 `MiniTest.java`: a test runner in fifty lines, so the Java tests need nothing but the JDK. 360
Listing 8.8 `TestTiers.java`: the quota tests in Java. 361
Listing 8.9 `TestSubscription.java`: the subscription tests in Java. 363
Listing 8.10 `until_reset.py`: seconds until the next 00:00 WIB. 368
Listing 8.11 `UntilReset.java`: the same computation with `Duration.between`. 368
Listing 8.12 `late_renewal.py`: the old rule and a fair rule for very late renewals. 369
Listing 8.13 `LateRenewal.java`: the same comparison in Java. 369
Listing 9.1 `scheduler.py`: a min-heap scheduler with per-tier cadence, lazy deletion, skip-not-burst catch-up and a simulated clock. 385
Listing 9.2 `Scheduler.java`: the same scheduler with `java.util.PriorityQueue` and records. 389
Listing 9.3 `test_scheduler.py`: seven tests for the scheduler. The release gate runs exactly this file. 393
Listing 9.4 `TestScheduler.java`: the same seven tests in Java. 395
Listing 9.5 `health.py`: a health endpoint that reports the running version and three checks, distinguishing down from degraded. 397
Listing 9.6 `HealthServer.java`: the same endpoint with `com.sun.net.httpserver`. 400
Listing 9.7 `release_gate.py`: a six-step release gate that refuses on failing tests or a changed payload, and declares success only when `/api/health` answers the new version. 404
Listing 9.8 `ReleaseGate.java`: the same gate in Java, with `ProcessBuilder`, `MessageDigest` and `java.net.http.HttpClient` 409
Listing 9.9 `weekly_trend.py`: ISO-week means of one monitor's scores, with a week-over-week alert. 418

Listing 9.10 `WeeklyTrend.java`: the same trend in Java. 418
Listing 9.11 `latest_only.py`: the sequence-number guard against stale replies. 420
Listing 9.12 `LatestOnly.java`: the same guard in Java. 420
Listing 10.1 `site_server.py`: serves the sample sites, simulating an edge layer (`_edge/`) and a stale cache (`_cache/`) in front of each origin. 440
Listing 10.2 `SiteServer.java`: the same sample-site server on the JDK's built-in HTTP server. 443
Listing 10.3 `robots_rules.py`: a reader that implements Equation (10.1) (merging, fallback to `*`) and Equation (10.2) (longest match, `Allow` on a tie), with the `*` and `$` wildcards. 449
Listing 10.4 `RobotsRules.java`: the same RFC 9309 reader in Java. 450
Listing 10.5 `auditor.py`: checks `robots.txt`, `llms.txt`, cache headers and JSON-LD of each site and prints a report and a summary table. 452
Listing 10.6 `MiniJson.java`: a tiny strict JSON reader for the Java auditor; anything that is not valid JSON, such as a trailing comma, throws an exception. 460
Listing 10.7 `Auditor.java`: the same auditor in Java; it prints exactly the same report as **Listing 10.5**. 462
Listing 10.8 `robots_quirk.py`: the same two questions put to `urllib.robotparser` and to the RFC 9309 reader of **Listing 10.3**. 473
Listing 11.1 `passwords.py`: PBKDF2-HMAC-SHA256 from the standard library, and Argon2id through the optional `argon2-cffi` package; each stored hash names its recipe. 496
Listing 11.2 `Pbkdf2Hasher.java`: the same PBKDF2 recipe with the JDK's `SecretKeyFactory` 498
Listing 11.3 `rate_limit.py`: a sliding-window limiter for failed logins, with a clock passed in so that tests can control time. 500
Listing 11.4 `SlidingWindowLimiter.java`: the same limiter in Java. 500
Listing 11.5 `privacy_app.py`: registration with explicit consent, rate-limited login with hash upgrade, export, correction, and erasure with a tombstone. 501
Listing 11.6 `PrivacyApp.java`: the same application in Java. 506
Listing 11.7 `Json.java`: a small JSON writer with the same layout as Python's `json.dumps(indent=2)`. 514
Listing 11.8 `demo.py`: consent, hashing, rate limiting, export, correction and erasure in one scenario. 515
Listing 11.9 `Demo.java`: the same scenario in Java. 518
Listing 11.10 `test_privacy_app.py`: ten tests, including a known-answer test for PBKDF2 and a search of every table after an erasure. 523
Listing 11.11 `PrivacyAppTest.java`: the same ten properties in Java, with a minimal runner. 527
Listing 11.12 `sol_policy.py`: asking for consent again after a policy change, on top of lab G7. 536
Listing 11.13 `SolPolicy.java`: the same in Java (the field `policyVersion` of `PrivacyApp` is deliberately not `final`) 537
Listing 11.14 `sol_sqlite.py`: lab G7's tables in SQLite, with erasure as one transaction. 539
Listing 12.1 `01_units.py`: PROFIT amounts as whole numbers of micro-PROFIT, with Equation (12.1) applied in both directions. 546
Listing 12.2 `Units.java`: the same conversion in Java, with a `long` count of micro-PROFIT. 547
Listing 12.3 `02_emission.py`: the decaying multiplier of Equation (12.2), the daily emission of Equation (12.3) and its power-law form Equation (12.5). 553
Listing 12.4 `Emission.java`: the same table in Java, formatted identically. 553
Listing 12.5 `03_supply_sim.py`: the recurrence Equation (12.6) run as an integer ledger, checked against the closed form Equation (12.8), with the steady state Equation (12.9) and $t90$ from Equation (12.10) 558
Listing 12.6 `SupplySim.java`: the same ledger simulation in Java, with arrays of `long` micro-PROFIT. 559
Listing 12.7 `04_breakeven_oracle.py`: the break-even user count by Equation (12.11) and by bisection, and a toy oracle floor. 564
Listing 12.8 `BreakEvenOracle.java`: the same two computations in Java. 566
Listing 13.1 `normalize.py`: two normalizers and a set that counts the spellings that really differ. 591
Listing 13.2 `Normalize.java`: the same normalizers in Java. 592

Listing 13.3 `researcher_check.py`: the OpenAlex researcher-record checker, with the exact-name guard, the rates of Equation (13.6) and an honest network failure. 593
Listing 13.4 `Json.java`: a small recursive-descent JSON reader and writer, enough for the lab. 597
Listing 13.5 `ResearcherCheck.java`: the same checker in Java, with a `Source` interface and two implementations. 600
Listing 13.6 `sample_server.py`: a local stand-in for the three OpenAlex calls, answering from the saved samples. 608
Listing 13.7 `SampleServer.java`: the same stand-in server in Java. 609

List of Pseudocode

Pseudocode 3.1 Deciding whether a crawler may fetch a path under RFC 9309: status codes first, then group choice, then the longest matching rule, with ties going to `Allow`. 104

List of Equations

(1.1) 6
(2.1) 33
(2.2) 34
(3.1) 86
(3.2) 97
(3.3) 105
(3.4) 110
(3.5) 120
(3.6) 120
(3.7) 120
(4.1) 153
(4.2) 153
(4.3) 154
(4.4) 157
(4.5) 157
(4.6) 159
(4.7) 164
(5.1) 203
(5.2) 204
(5.3) 204
(5.4) 204
(5.5) 204
(5.6) 205
(5.7) 205
(5.8) 209
(5.9) 211
(5.10) 211
(6.1) 235
(6.2) 237
(6.3) 237

(6.4) 241
(6.5) 241
(6.6) 241
(6.7) 243
(6.8) 244
(7.1) 293
(7.2) 294
(7.3) 295
(7.4) 296
(7.5) 297
(7.6) 297
(7.7) 299
(7.8) 299
(7.9) 299
(7.10) 299
(7.11) 300
(8.1) 324
(8.2) 328
(8.3) 329
(8.4) 332
(8.5) 335
(8.6) 335
(8.7) 335
(9.1) 373
(9.2) 373
(9.3) 382
(9.4) 382
(9.5) 384
(9.6) 384
(9.7) 384
(9.8) 403
(10.1) 428
(10.2) 429
(10.3) 435
(10.4) 435
(10.5) 436
(10.6) 437
(11.1) 488
(11.2) 491
(11.3) 492
(12.1) 546
(12.2) 551
(12.3) 551
(12.4) 552
(12.5) 552
(12.6) 556
(12.7) 557
(12.8) 557
(12.9) 557
(12.10) 557
(12.11) 564

(12.12) 569
(12.13) 570
(12.14) 572
(13.1) 583
(13.2) 583
(13.3) 583
(13.4) 583
(13.5) 584
(13.6) 590

Preface

Ask an AI assistant which laundry in Surabaya is open late, which clinic takes walk-in patients, or which textbook explains algorithms best, and it answers in a sentence or two. It does not give you ten blue links. It gives you a *name*. Whether that name is yours depends on something almost no business owner in Indonesia had heard of two years ago: how ready your website is to be read, understood and quoted by an answer engine. Profitina exists to measure that readiness, explain it in plain words, and help fix it.

This book tells how Profitina was built, in the open, from a lecturer's classroom in Surabaya to a platform running in production at app.profitina.com. It is the story of a product, but it is written as a textbook: every idea is explained from the ground up, and every chapter has a hands-on lab that you build yourself, in both **Python** and **Java**, so that you understand the engineering and not only the headlines.

Who this book is for

Students of informatics who want to see how the subjects they study — data structures, web programming, databases, security, artificial intelligence — come together in a real product. Business owners and marketers who want to understand why AI answer engines matter and what can actually be done about them. Developers who want an honest account of the decisions, trade-offs and mistakes behind a small platform with real users. You need no prior knowledge: Chapter 1 starts with an empty computer and ends with your first programs running.

What is in it

Chapter 1 tells the origin story and sets up your workbench: VS Code, Python and Java on Windows, macOS or Linux. Chapter 2 explains how search turned into answers and what Generative Engine Optimization (GEO) is. Chapter 3 walks through what Profitina does, from the first scan to the monitored result, and lets you build your own AI-readiness checker. Chapters 4 and 5 treat the real-time community and how it is kept safe. Chapters 6 and 7 cover local AI — the Content Engine, the two-lane model and the feedback loop in which humans decide and models learn. Chapter 8 is the business model. Chapter 9 is the architecture, reliability and the release gate. Chapter 10 studies the marketing site as a GEO case of its own. Chapter 11 is privacy and compliance by design under Indonesia's personal data protection law. Chapter 12 explains PROFIT, the utility coin on Solana, stating plainly what is live and what is roadmap. Chapter 13 places Profitina within the Profitina Group and looks ahead.

Every lab was run before its output was printed. The programs are collected, chapter by chapter, in the accompanying `Code` folder, with a README in every chapter folder that explains how to run each one in a terminal and in VS Code on Windows, macOS and Linux.

How to read it

In order, the first time; after that, any chapter stands on its own. Type the labs in rather than only reading them, change one thing, and run them again. The exercises at the end of each chapter all have complete solutions — use them after your own attempt, not instead of it.

Numbered figures, tables, listings and equations are live links: in the electronic edition, click "Listing 3.2" or "[4]" anywhere in the text and you are taken straight to it. The contents and the lists at the front of the book work the same way.

About the author

Irfan Subakti describes himself in one line — *Father | Educator | Entrepreneur | Bridging Tech & Nature: CS Lecturer × Crypto/Digital Business × AI/ML Specialist × Fruit Farmer | Storyteller* — and every part of that line has found its way into this book. He has taught in the Department of Informatics at Institut Teknologi Sepuluh Nopember (ITS), Surabaya, since 1998. He supervises and examines undergraduate theses, teaches MLOps for the IBM SkillsBuild program, and builds products: he founded Profitina and the Profitina Group. Away from the keyboard he tends the family fruit and durian orchard, where he learned that good software, like a good orchard, is grown patiently, one well-tended piece at a time. The storyteller in him is the reason this book prefers a story to a list whenever a story will do.

Acknowledgments

To the members of the Profitina community, whose questions, reports and patience shaped the product more than any plan did; to the students at ITS who tried its earliest versions and told the truth about them; and to the colleagues with whom the ideas in this book were discussed and sharpened. The book owes its standard to the published research on answer engines and to the official documentation of every technology it uses, which is cited wherever it is used.

Surabaya, September 2026



Irfan Subakti · yifana@gmail.com

Profitina

profitina.com

Chapter 1

Origins: From a Lecturer's Classroom to a Platform

By the end of this chapter you will know where Profitina came from and why it looks the way it does: who built it, what he had been teaching for almost thirty years before he wrote its first line, how an idea became a live platform on 17 July 2026 and grew week by week afterwards, and which ideas were there from the start. You will know whether this book is written for you, and which chapters to read first if your time is short. And you will have a working programmer's bench on your own computer, whether it runs Windows, macOS or Linux: Visual Studio Code, a Java Development Kit and Python, checked one by one, with your first two programs written, run and understood in both Java and Python. Every later chapter ends in a hands-on lab, and every lab assumes that the bench of Section 1.8 is standing.

1.1 A Platform with a Classroom Behind It

Most technology products are introduced with a founding legend: a garage, a dormitory room, a sleepless weekend, a sudden flash of insight written on a napkin. Profitina does not have one, and this chapter will not invent one. What it has instead is less dramatic and, for a reader who wants to learn something, far more useful: a long and well-documented trail that runs from a university classroom in Surabaya, through almost three decades of teaching and research in artificial intelligence, to a production web application that anyone can open today at `app.profitina.com`.

Profitina is Indonesia's AI Visibility platform. The discipline it practices has a name, GEO, short for *Generative Engine Optimization*, and Chapter 2 explains it carefully. In one sentence: Profitina measures how ready a business, a website, a researcher or an organization is to be *found and quoted* by AI answer engines such as ChatGPT, Gemini and Perplexity, and then helps fix what it finds [1]. It does this from signals anyone can observe on the public web, such as whether a page's text can be extracted cleanly, whether it carries structured data in the schema.org vocabulary, and whether its technical foundations are sound. Its Indonesian tagline calls it *aplikasi manajemen & optimalisasi visibilitas AI nomor 1 di Indonesia*, and its three-word motto is *Gather · Grow · Give* (in Indonesian, *Bersama · Berkembang · Berbagi*) [1].

Two facts about Profitina's origin shape everything that follows in this book.

The first is that it was built by a lecturer who has never stopped lecturing. The same person who designs Profitina's architecture teaches algorithms, web programming and object-oriented programming to undergraduates every semester. The product is therefore not only a business; it is also a teaching case, examined from many angles in a library of course books, and since 31 August 2026 it has been a place where a class of about 250 students actually works.

The second is that it was built in the open and at speed. The current code base begins with version 0.11.0 on 3 July 2026. Two weeks later, on 17 July 2026, the platform went into production. By 24 September 2026, the day this chapter was finished, it had reached version 0.14.5 and its public whitepaper had reached its twenty-fourth version. Every one of those steps is dated, and Section 1.4 lays them out.

This chapter tells that story in order. Section 1.2 introduces the founder and his academic path. Section 1.3 shows why the classroom and the code base are, in a precise sense, the same place. Section 1.4 walks through the dated timeline. Section 1.5 names the founding ideas that the rest of the book keeps returning to. Sections 1.6 and 1.7 say who the book is for and how to read it. Section 1.8, the longest, sets up your computer. Then come the chapter's regular closing sections: Profitina in practice, common misunderstandings, a summary, exercises and full solutions.

1.2 The Founder and Three Decades of Teaching

1.2.1 A lecturer in Surabaya

Profitina was founded, and is still built, by Ir. M.M. Irfan Subakti, S.Kom., M.Sc.Eng., M.Phil., IPM, a lecturer in the Department of Informatics of Institut Teknologi Sepuluh Nopember (ITS) in Surabaya, one of Indonesia's leading engineering universities. Everywhere else in this book he is simply "the founder".

The string of letters around his name is worth decoding once, because each part records a different kind of training. *S.Kom.* (Sarjana Komputer) is the Indonesian bachelor's degree in computer science. *M.Sc.Eng.* is a Master of Science in Engineering and *M.Phil.* a Master of Philosophy, the research degree. *Ir.* is the professional engineer's title awarded through a university's professional engineering program (Program Profesi Insinyur), and *IPM* (Insinyur Profesional Madya) is a mid-level professional engineer certification. Together they describe a career built around computing and artificial intelligence, recognized both academically and professionally.

1.2.2 Where the teaching began

The earliest documented course the founder taught at ITS ran from September 1997 to January 1998: a unit on a *general inference engine* within a course on knowledge-based systems. An *inference engine* is the part of an expert system that takes a set of rules, such as "if the fever is high and the rash is red, suspect measles", and a set of facts, and derives new facts from them, one rule at a time, until it reaches a conclusion. It is one of the oldest ideas in applied artificial intelligence, and it is still, in modern clothing, at the heart of systems that must *explain* their decisions. His own undergraduate thesis, in 1998–1999, was a simulator for Flexible Manufacturing Systems, a program that models a factory's machines, queues and schedules so that one can experiment with them without stopping a real production line.

It is tempting to draw straight lines from those beginnings to Profitina, and some of them hold. A readiness scan that produces a score *together with the reasons for it* is, in spirit, a descendant of the expert system that must show its rules. A platform whose release process, re-scan scheduler and community feed are all queues and schedules is a cousin of the manufacturing simulator. The book will not overstate these links, but it will point them out where they genuinely help you understand a design choice.

1.2.3 Research in several countries and several branches of AI

Between 2007 and 2010 the founder spent research periods abroad, at National Chiao Tung University (NCTU) in Hsinchu, Taiwan, from 2007 to 2009, and at Comenius University in Bratislava, Slovakia, from 2009 to 2010. His publications over the years cover expert systems, ontologies, fuzzy logic, genetic algorithms, neural networks, and image processing and retrieval. Taken together, that list is close to a table of contents of classical artificial intelligence: reasoning

with rules, representing knowledge as structured concepts, reasoning with degrees of truth, searching by imitating evolution, learning from examples, and making sense of pictures.

This matters for how the rest of the book should be read. Profitina is not the story of a business person who hired AI expertise to decorate a product idea. It is closer to the reverse: a career spent building and teaching AI systems from their foundations, applied to a real product that the same person also operates. When Chapter 6 explains why Profitina runs its language models *locally*, on its own hardware, instead of renting a commercial AI service, you will see that the decision was made by someone who is used to understanding a model rather than merely calling one.

> **KEY INSIGHT.** Profitina's unusual feature is not that it uses AI; almost every new product does. It is that the platform's designer has taught the foundations of AI, algorithms and software construction for about three decades, and that the product is deliberately built so that it can be *taught*: its ideas are explained in public, its numbers are dated, and its hands-on labs can be rebuilt by a student with nothing but a laptop.

1.3 The Classroom and the Code Base Are the Same Place

1.3.1 Three courses a platform is made of

The most concrete evidence of the link between the founder's teaching and the platform is simple and checkable. In the 2025/2026 academic year he teaches three undergraduate courses at ITS: EF234405 *Design and Analysis of Algorithms* (DAA), EF234301 *Web Programming* (WebPro) and EF234302 *Object-Oriented Programming* (OOP). These are not loosely related electives. They are, respectively, the course about thinking in algorithms and their costs, the course about building web applications, and the course about structuring software into cooperating objects. A platform like Profitina is made of exactly those three materials: the costs of its queries and schedules are DAA questions, its application and its embedded community widget are WebPro artifacts, and the scanner's many independent checks, each reporting its own result, are an OOP design.

A small detail shows that this is a maintained record rather than a line on a curriculum vitae. The course codes changed from an older scheme beginning with "IF" (for example IF184301 in 2022/2023) to the current scheme beginning with "EF" (for example EF234405 in 2025/2026) as part of an ITS curriculum revision.

1.3.2 A library of course books around one real system

This book belongs to a series, the Profitina Course Library. Its books include *Design and Analysis of Algorithms*, *Web Programming*, *Object-Oriented Programming*, *Data Structure*, *Framework-Based Programming* and *Fundamental Programming*, together with books on competitive programming, intelligent systems, numerical methods and human-computer interaction. Most of them use Profitina as a running, real-world case. A few examples show the range: login rate limiting is a *sliding-window* algorithm in the algorithms book; password hashing with Argon2id is an example of a deliberately *memory-hard* function; the decaying emission of the PROFIT coin is an exponential-decay recurrence in the numerical-methods book; and the platform's leaderboards are a practical exercise in sorting and ranking.

The Web Programming and Framework-Based Programming books go one step further and build a small but complete application, **Profitina Laundry**, as their running case study. Profitina Laundry is a fictional laundry business in Surabaya with three outlets, a database of customers,

services and bills, and versions in several languages and frameworks. It has no connection to the real product's code or data; it exists so that every hands-on exercise has a friendly, concrete business to work on. You will meet it in this book too: in Chapter 2 we write its structured data and its `robots.txt`, and later labs scan it, schedule it and protect it.

1.3.3 The platform in a real class

On 31 August 2026 the relationship between classroom and platform stopped being only a matter of examples. The first ITS class of about 250 students began using Profitina's community for their course work, organized in course channels. Several features that are now part of the product were built for exactly that situation, in the weeks before and after it:

- **Channel-scoped moderators** (13 August 2026). A moderator, for example a lecturer or a student coordinator, can hide a post, pin an announcement or temporarily mute a participant *in their own channel only*. Moderators never touch anyone's PROFIT balance or private data, every action is logged, and the person affected is notified.
- **Neutral handles** (14 August 2026). New accounts receive a random handle, the letter `u` followed by eight hexadecimal digits, which the user can change. Handles are checked so that a student cannot accidentally publish a student identity number as a public name.
- **Self-service rights** (19 August 2026). The panel *Akun & Data Pribadi* (Account and Personal Data) lets each user export their data, change their email or password, or delete their account permanently, without asking anyone. Chapter 11 tells the instructive story of why this panel was urgent.

A platform used by a real class receives real feedback, very quickly. One decision from that period is a good example of the discipline the book returns to again and again. In mid-September 2026 the team evaluated an idea to offer free AI-generated content to all 250 students. It was measured and, on the evidence, *frozen* on 15 September 2026. An attractive idea was set aside because the measurement said so. Chapter 7 explains the general rule: humans decide, and they decide on evidence.

1.4 From Idea to Production: A Dated Timeline

1.4.1 Before July 2026

Profitina's ideas are older than its current code base. Before July 2026 the project went through twelve versions of its whitepaper, several versions of a custom WordPress theme for the marketing site, and an early methodology called "3P". The product at that time also had a third subscription tier, *Agency*, for consultancies managing many clients; as the timeline shows, it was later removed. The earlier history lives outside the code base the book describes, so the dated story below begins on 3 July 2026.

1.4.2 Twelve weeks, week by week

Table 1.1 lists the milestones from the first version of the current code base to the day this chapter was written, and **Figure 1.1** draws the same period on a time axis. Read the table slowly: it is also a map of the book, because almost every row is the subject of a later chapter.

Table 1.1 Profitina's dated milestones from 3 July to 24 September 2026, with the chapter of this book that treats each one. The last row is a plan, not a fact.

Date (2026)	Milestone	Chapter
3 Jul	Version 0.11.0: GEO scan, PROFIT concept, theme and whitepaper v12 carried into the current code base	1
Jul	Phase 1 (accounts, scan) and Phase 1.5 (personal-data protection: Argon2id, consent, rate limiting, rights, legal pages)	3, 11
17 Jul	**Production launch.** The production database is reset to zero just before the first public users ("launch-fresh"), so no test data is carried over	9
late Jul	PROFIT economy: email verification by one-time code, fair-play checks, Link Safety, the PROFIT-to-rupiah oracle and QRIS purchase, light and dark themes, full ID/EN interface, PROFIT as a token on Solana's devnet	5, 8, 12
29 Jul	Historical dashboard, re-scan scheduler and alerts; version 0.12.0	9
31 Jul	The X (Twitter) API is removed; sharing moves to X's Web Intent, posted by the user from their own account; granular progress bars	3
1–5 Aug	Two local-AI lanes: a quality lane on the CPU and a fast lane on the GPU	6
6–7 Aug	Two plans only, Free and Pro: the Agency tier is removed; no advertising, permanently	8
12 Aug	Posts can be edited for 15 minutes, with the reward fully recalculated	4
13 Aug	Channel-scoped moderators; administrator grants with a public history; separate database roles for development and production	4, 9
19 Aug	Self-service privacy-rights panel	11
20 Aug	One-tier referral program, profile pages with per-item consent, direct messages	4, 8
30 Aug	PROFIT can be spent on real services; one single home for the Privacy Policy	11, 12
31 Aug	First ITS class of about 250 students; Researcher Visibility Report (OpenAlex)	3, 13
1 Sep	*One Text, Many Platforms*	3
9 Sep	Organization Visibility	3, 13
14 Sep	The project joins the Profitina Group umbrella; health-verified release gate	9, 13
15–16 Sep	Dashboard as a "cockpit", instant posting, administrator command center	4, 9
23–24 Sep	Instant channel switching (a feed page from 868 ms to 11 ms); versions 0.14.3 to 0.14.5-testnet; whitepaper v24	9
Q4 (plan)	Phase 3: PROFIT on Solana's main network and a non-custodial exchange, gated by licensing and audit	12

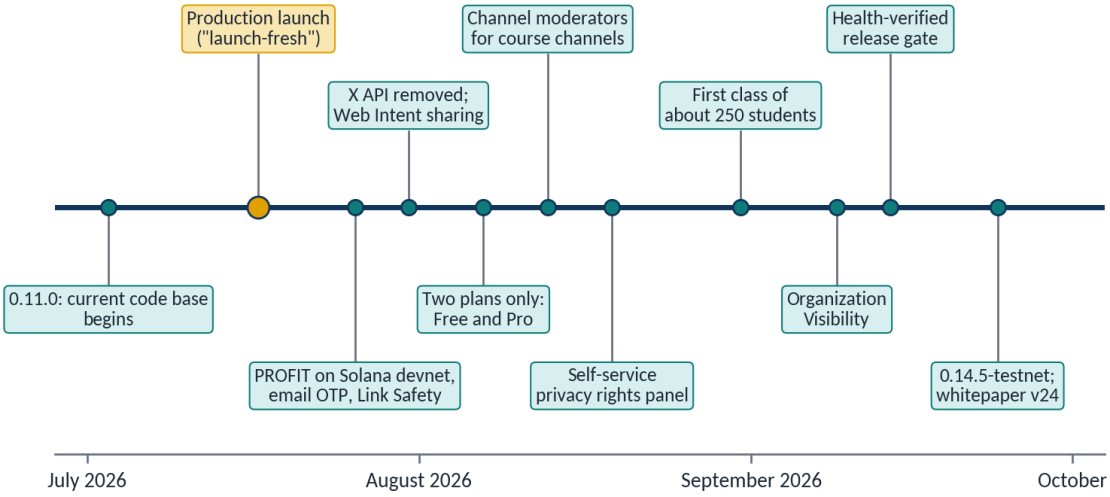


Figure 1.1 The first twelve weeks of the current code base on a time axis. The amber point is the production launch of 17 July 2026.

Three patterns stand out in **Figure 1.1**.

The first is **pace**. Between 3 July and 24 September 2026 the code base received 391 commits, on 35 different days, and moved from version 0.11 to 0.14. A new capability reached users roughly every week.

The second is **subtraction**. Several of the milestones *remove* something: the X API, the Agency tier, advertising. A young product is tempted to promise everything. Profitina's timeline shows the opposite habit: when a feature could not be verified, could not be sustained, or distracted from the main job, it was taken out, and the reason was written down. Chapter 8 quotes a house rule that follows from this habit: *never write a promise of an unbuilt feature next to a paid price.*

The third is **dating everything**. Each row of **Table 1.1** carries a date because the platform keeps a written log of every change, and the whitepaper is versioned in the same way. This book adopts the same discipline. Every number that can change, such as a version, a test count or a statistic, is printed with the date on which it was true, usually "at the time of writing (24 September 2026)". When you read this later, some of those numbers will have moved. That is expected, and the dates tell you by how much the world has moved since.

1.4.3 Counting the days: a first look at code

How long has Profitina been live? On 24 September 2026 the answer is 69 days. Counting days by hand across months of different lengths is exactly the kind of small, error-prone task that a program does better than a person, and it makes a good second program for Section 1.8.9: it reads the milestones of **Table 1.1** from a file and prints each one's distance from the launch. If d is a number of days, the number of whole weeks w and the remaining days r are

$$w = \left\lfloor \frac{d}{7} \right\rfloor, \qquad r = d - 7w \tag{1.1}$$

where $\lfloor x \rfloor$, read "floor of x", is the largest whole number not greater than x. For $d = 69$: $69/7 = 9.857\ldots$, so $w = 9$ and $r = 69 - 63 = 6$. Profitina had been live for 9 weeks and 6 days. Equation (1.1) is what the operators `//` and `%` compute in Python, and `/` and `%` on whole numbers in Java, for any $d \geq 0$.

1.5 The Founding Ideas

1.5.1 The mission

Profitina states its mission in one sentence:

> **MISSION.** "Democratize AI visibility for every Indonesian business and creator through a transparent, inclusive, self-reliant platform — so anyone (from a warung to a SaaS startup to a solo creator) can be found and quoted by AI, without paid APIs or gatekeepers."

Three words in it carry the weight. *Transparent* means that the method is shown: a scan does not only give a score between 0 and 100 but also the reasons and a prioritized list of fixes, and the platform says openly that its results are "methodological estimates, not guarantees of actual third-party AI engine output" [1], [2]. *Inclusive* means the same product serves a *warung* (a small neighborhood shop or food stall), a software start-up and a single creator, under the same method. *Self-reliant* means that the platform's AI runs on infrastructure it operates itself and does not depend on renting a third party's AI service.

1.5.2 Visibility and participation in one system

The second founding idea is less obvious and more original. For two decades, social platforms have collected enormous value from ordinary people's participation, such as posts, replies, recommendations and reviews, and paid it back mostly in likes and followers. Profitina's founding thesis, expressed in its design rather than in a slogan, is that useful contribution can be *measured*, and that the measurement can be tied to a real, if small, reward.

In the product this idea takes three forms. A community, with channels, threads and direct messages, lives inside the same application as the visibility tools (Chapter 4). Contributions that help, such as feedback that improves a scan, are recognized and credited, and every decision about what ships is made by a person (Chapter 7). And a utility coin, PROFIT, is earned for useful contribution and can be spent on the platform's own services (Chapter 12). The founder's academic environment was asking related questions at the same time: undergraduate theses he supervised in 2026 studied how large holders move the volatility of a token on Ethereum, how non-transferable "soulbound" tokens could represent a professional identity, and how trend-following and mean-reversion strategies behave on the Solana spot market. No document claims that those theses shaped PROFIT's design, and this book does not claim it either; they show that the classroom and the product were working on neighboring problems in the same months.

> **LIVE AND ROADMAP.** PROFIT exists today as a token on Solana's *devnet*, a public test network whose tokens have no monetary value; its contract can be inspected by anyone in a block explorer. Buying PROFIT with rupiah through QRIS, earning it and spending it inside the platform are live. Selling PROFIT is disabled, and a launch on Solana's main network is a Phase 3 plan that depends on licensing and an audit. Nothing in this book is investment advice or a promise about any price.

1.5.3 Local AI, by choice

The third founding idea is architectural. Every AI feature in Profitina runs on language models that execute on Profitina's own machine, through a runtime called Ollama, in two lanes: a *quality lane* on the processor with the Qwen2.5 7B model for long texts such as content drafts, and a *fast lane* on the graphics processor with a very small model, qwen3:1.7b, for short classification tasks [1]. Code paths that once called hosted AI services were removed entirely. Two consequences follow. User data never leaves the server for AI processing, which matters under Indonesia's

personal-data protection law (Chapter 11). And the platform can be honest about its limits: when no model is available, the application *says so* instead of silently serving a template as if a model had written it.

A related principle is easy to misunderstand, so it is worth stating early. Profitina does **not** measure a site's visibility by asking ChatGPT, Gemini or Perplexity questions about it. Those engines are the *targets* whose readiness is estimated, never components the platform depends on. The reasons are independence, privacy, and a house rule against paid third-party APIs. Chapter 2 explains what the platform measures instead.

1.5.4 Tested before it ships

The fourth founding idea is discipline about releases. A new version of Profitina reaches users only through a release gate that runs the full automated test suite first. At the time of writing (24 September 2026, version 0.14.5) that suite had **1,577 tests passing and 1 skipped** on its SQLite test path, and a priority subset of **440 tests** passed against PostgreSQL 16, the production database. The gate declares success only after the newly deployed server answers its health endpoint with the new version number. Chapter 9 tells the story of how the gate came to be this strict, including the day a release script reported success for a version that had not actually been deployed.

1.5.5 Indonesia first

The last founding idea is geographical. Profitina is built for Indonesia first in concrete ways, not as a slogan: its interface is fully bilingual, Indonesian and English; its handling of personal data follows Law No. 27 of 2022 on Personal Data Protection (UU PDP); and its payments run on QRIS, Indonesia's national standard for QR-code payments. Expansion to the region is a Phase 4 plan that comes only after leadership at home [1]. The home market is large: according to the 2026 survey of the Indonesian Internet Service Providers Association (APJII), Indonesia has about 235.3 million internet users out of a population of about 287.3 million, a penetration of 81.72 percent [3].

1.6 Who This Book Is For

This book is written to be read from the first page by someone who knows nothing about GEO, web crawlers, language models or blockchains, and to leave that reader able to explain each of them and to build a small working example of each. It is also written for readers who already know some of these things and want to see how they fit together in one real system. **Table 1.2** names five kinds of reader and suggests where each can start.

Table 1.2 Five kinds of reader, what each will find in this book, and which chapters to read first.

If you are…	You will find…	Start with
A student of informatics or a related field	A real production system examined end to end, with every lab in Python and Java, and exercises with full solutions	1, 2, 3, then any order
A software developer or engineer	Architecture, reliability, local AI, real-time feeds, testing as a release gate, and the reasoning behind each choice	2, 3, 6, 9
A business owner or a creator	What GEO is, why AI answers change how customers find you, and what you can do this week	2, 3, 10
A partner or someone evaluating Profitina	What is live and what is roadmap, the business model, compliance, and the PROFIT coin described factually	2, 8, 11, 12
A lecturer	A case study that connects algorithms, web programming, object-oriented design, security and privacy law	1, 3, 9, 11

Whichever row describes you, read Chapters 1 and 2 in full. They are the foundation that every later chapter assumes: Chapter 1 gives you the story and the working bench, Chapter 2 the vocabulary and the reasons.

Two groups deserve a direct word. If you are a **business owner** with no programming background, you can read the story sections of every chapter and skip the listings; the text around each listing always says in plain words what the program does and what its output means. If you are considering Profitina as a **partner or investor**, this book is deliberately not a sales document. It says plainly where the platform stands today and where the roadmap points, including that the PROFIT coin is not tradable today and that its main-network launch depends on regulators and an audit. Nothing here is financial advice.

1.7 How to Read This Book

1.7.1 The shape of the book

The book has thirteen chapters, grouped by the question they answer. **Figure 1.2** shows the groups: *why* Profitina exists (Chapters 1 and 2), *what* it does (Chapters 3 to 5), *how* it works (Chapters 6 to 10), and why it can be *trusted*, together with where it is going (Chapters 11 to 13). Under each chapter title the figure names its hands-on lab.

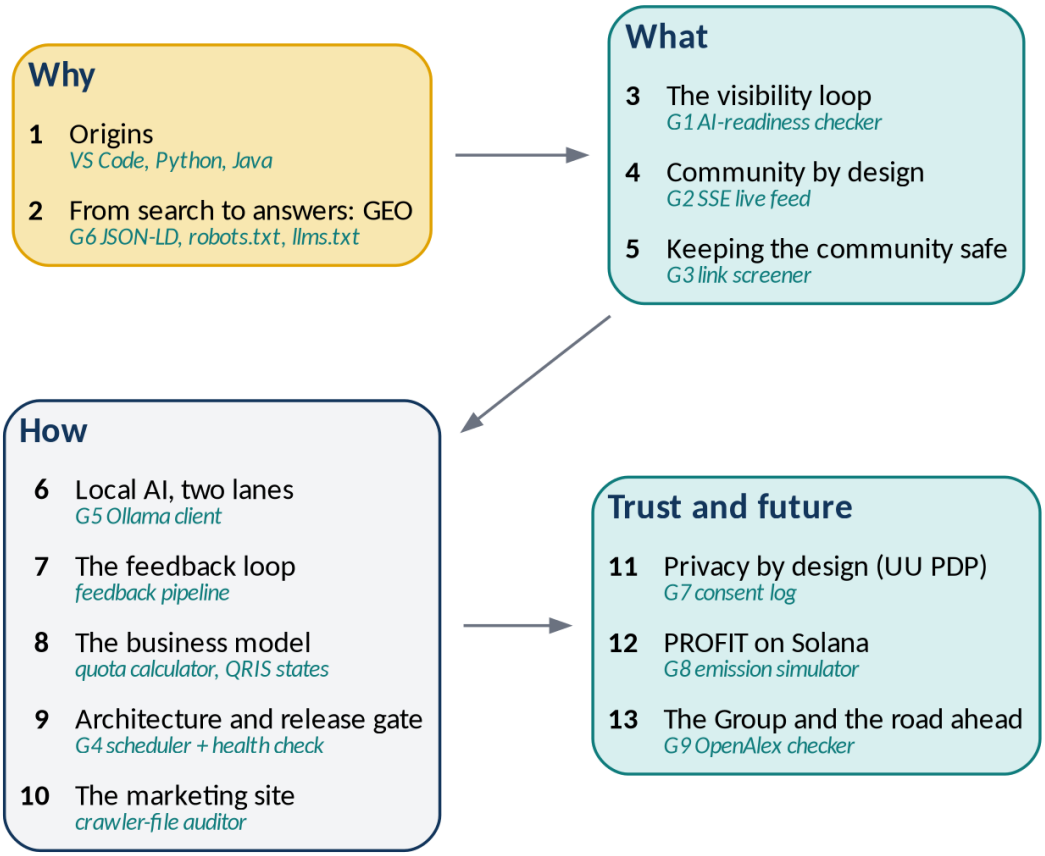


Figure 1.2 The thirteen chapters in four groups, with the hands-on lab of each chapter in italics. The labs marked G1 to G9 are the book's numbered programs.

1.7.2 Every chapter has the same parts

Every chapter opens with a paragraph that says what you will be able to do at the end, and closes with the same five sections: *Profitina in Practice* connects the chapter to the live product; *Common Misunderstandings* collects the mistakes readers most often make with the chapter's ideas; *Chapter Summary* restates the essentials; *Exercises* give you practice; and *Solutions to the Exercises* work every exercise out in full. A chapter's *References* list its sources, numbered in the order in which the text first cites them, like this [1].

1.7.3 Programs, outputs and the code folder

Every lab is written twice, in **Python** and in **Java**, with the same behavior and, wherever possible, byte-for-byte the same output, so that you can read the language you know and learn the other by comparison. The listings use only each language's standard library, so there is nothing extra to install. Every listing is a complete program, printed exactly as it is in the book's code folder, and every block of output under a listing is what that program really printed when it was run for this book. The code folder has one subfolder per chapter, `Chapter01`, `Chapter02` and so on, each with `python` and `java` subfolders and a `README.md` that repeats the run instructions for Windows, macOS and Linux.

The programs were run for this book on Linux with Python 3.11 and JDK 21, and they are written for the current releases, Python 3.14 and JDK 25. Where a program prints something that depends on your machine, such as a folder name or the version of your Java, the text says so.

1.7.4 Boxes, labels and dates

Four kinds of box appear in the text. A **definition** box introduces a term precisely; a **key insight** box states an idea worth remembering; a **warning** box marks a trap, and also every place where the difference between *live* and *roadmap* matters; and an **example** box works through a case or quotes a statement word for word. Mathematics is written in standard notation and every symbol is explained the first time it appears; the important formulas are numbered, like Equation (1.1).

Profitina changes weekly. When the book gives a version, a count or a price, it gives the date as well. When it describes something planned, it says so in the same sentence. And when a number comes from outside Profitina, such as a survey or an industry statistic, the reference list says who measured it and when.

1.8 Setting Up VS Code for Python and Java on Windows, macOS and Linux

Everything in this book runs on an ordinary laptop, and nothing has to be bought. This section installs the complete bench once and carefully: an editor, a Java Development Kit and Python. Then it proves that each part works by running the same first program in both languages. **Figure 1.3** shows how the parts fit together.

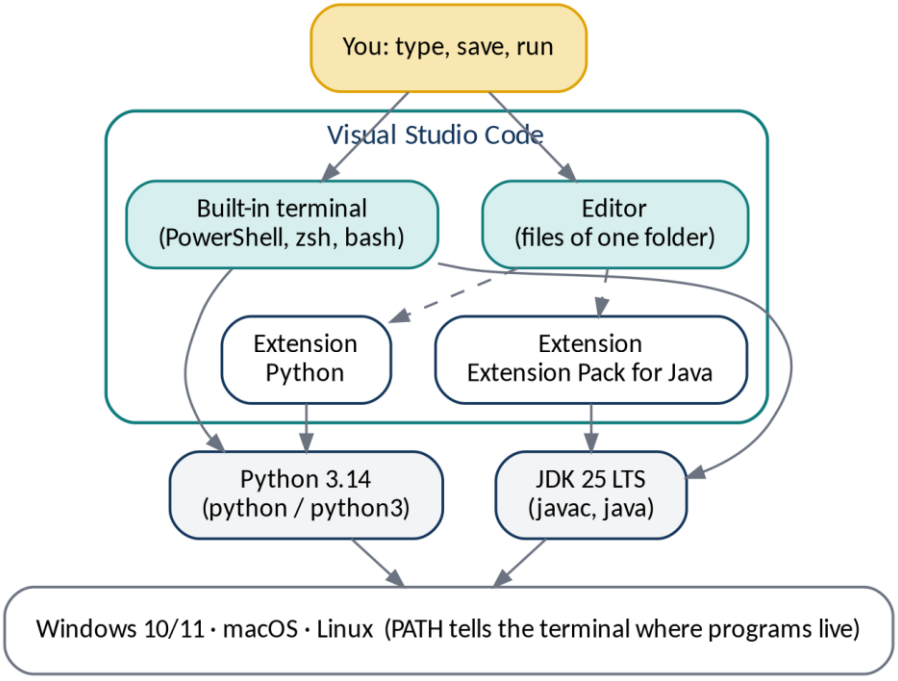


Figure 1.3 The bench used throughout the book. You work in VS Code's editor and terminal; two extensions teach VS Code about Python and Java; the terminal finds Python and the JDK through the operating system's PATH.

1.8.1 Three words first

A **terminal** is a window in which you type commands and read their answers. On Windows it is **PowerShell** (or the older Command Prompt); on macOS it is the **Terminal** application, running a shell called `zsh`; on Linux it is also called Terminal and usually runs `bash`. VS Code has a terminal built in, opened with **Terminal → New Terminal**. In this book, a line to type into a terminal is printed in a monospaced font, and you press Enter at its end.

A **package manager** installs software by command instead of by clicking through an installer: `winget` on Windows 10 and 11, **Homebrew** (the `brew` command, from `https://brew.sh`) on macOS, and `apt` on Ubuntu and Debian. Each step below gives both the package-manager command and the graphical alternative; either is fine.

The **PATH** is the list of folders in which a terminal looks for programs. When you type `javac`, the terminal searches each folder on the PATH, in order, for a program called `javac` (on Windows, `javac.exe`). "Command not found" or "is not recognized as an internal or external command" almost always means one of two things: the program is installed but its folder is not on the PATH, or the terminal was opened *before* the folder was added. Opening a **new** terminal is the first cure to try.

> **DEFINITION.** The *PATH* is an environment variable holding a list of folders, separated by `;` on Windows and by `:` on macOS and Linux. To run a command given by name, the terminal tries each folder of the list in turn and runs the first program with that name it finds.

1.8.2 Visual Studio Code

Visual Studio Code, VS Code for short, is a free editor from Microsoft that looks and works the same on all three systems and becomes a complete environment for a language through **extensions**. It is updated every month; the newest release at the time of writing is version 1.139, released on 23 September 2026 [4]. Any recent version will do. **Table 1.3** gives the installation.

Table 1.3 Installing VS Code on each operating system.

	Windows 10/11	MacOS	Linux (Ubuntu/Debian)
Install	Installer from `https://code.visualstudio.com`; tick **Add to PATH**. Or `winget install Microsoft.VisualStudioCode`	Download, unzip and drag *Visual Studio Code* into **Applications**. Or `brew install --cask visual-studio-code`	The `.deb` package from the website, then `sudo apt install ./code_*.deb`. Or `sudo snap install code --classic`
Make `code` work in a terminal	Done by the installer	In VS Code: **Cmd+Shift+P**, then *Shell Command: Install 'code' command in PATH*	Done by the package
Check	`code --version`	`code --version`	`code --version`

1.8.3 Java: JDK 25

Java programs are compiled with a **JDK**, a Java Development Kit, which contains the compiler `javac` and the launcher `java`. Install **JDK 25**, the long-term-support (LTS) release of September 2025, which receives updates for years. JDK 27, released on 15 September 2026, is the newest feature release, but it is supported only until the next feature release replaces it in March 2027 [5], [6]. For a book, an LTS release is the calm choice. The **Eclipse Temurin** builds from the Adoptium project are free, and identical on all three systems [7]. The programs in this book need Java 17 or newer and were compiled here with JDK 21. **Table 1.4** gives the installation.

Table 1.4 Installing JDK 25 on each operating system.

	Windows 10/11	MacOS	Linux (Ubuntu/Debian)
Install	`winget install EclipseAdoptium.Temurin.25.JDK`, or the `.msi` installer from `adoptium.net` with **Set JAVA_HOME** ticked	`brew install --cask temurin@25`, or the `.pkg` installer from `adoptium.net`	`sudo apt install openjdk-25-jdk`, or Temurin from the Adoptium apt repository
Check	`javac -version` and `java -version`	same	same

Both checks must answer, and with the same version number. If `java` answers and `javac` does not, only a Java *runtime* was installed, which can run Java programs but not compile them: install the JDK. If both answer with *different* versions, two JDKs are installed and an older one comes first on the PATH; move the newer one's `bin` folder to the top of the PATH, or uninstall the older one.

1.8.4 Python 3.14

Install **Python 3.14**, the current series; its newest release at the time of writing is 3.14.7, of 5 August 2026 [8]. The book's Python programs need Python 3.10 or newer and were run here with Python 3.11. **Table 1.5** shows how to install it and how the command differs between systems.

Table 1.5 Installing Python on each operating system.

	Windows 10/11	MacOS	Linux (Ubuntu/Debian)
Install	Installer from `https://www.python.org`; on its first screen tick **Add python.exe to PATH**. Or `winget install Python.Python.3.14`	Installer from `python.org`, or `brew install python@3.14`	The system `python3`: `sudo apt install python3`; recent Ubuntu releases already include it
Run a program	`python hello.py`	`python3 hello.py`	`python3 hello.py`
Check	`python --version`	`python3 --version`	`python3 --version`

On Windows the command is `python`; on macOS and Linux it is `python3`, because on older systems the name `python` meant the long-retired Python 2. Where this book writes `python3`, Windows users type `python`.

1.8.5 The two VS Code extensions

Open the Extensions view (the icon of four squares in the left bar, or **Ctrl+Shift+X**, **Cmd+Shift+X** on a Mac), search for each name in **Table 1.6**, and press **Install**. Or type the two commands below the table into any terminal.

Table 1.6 The VS Code extensions used in this book.

Extension	Publisher	Identifier	Gives you
Python	Microsoft	`ms-python.python`	Python editing, running, debugging and virtual environments [9]
Extension Pack for Java	Microsoft	`vscjava.vscode-java-pack`	Java editing, running, debugging, testing and project views [10]

```
code --install-extension ms-python.python
code --install-extension vscjava.vscode-java-pack
```

The Python extension also installs *Pylance*, which underlines mistakes as you type, and the Java pack installs the *Language Support for Java* extension, a debugger and a test runner. The first time

you open a Java file, the Java extension may take a minute to start; the status bar at the bottom of the window shows its progress.

1.8.6 A folder for the book

Make one folder for this book, for example `Profitina` in your Documents folder, and copy the book's code folder into it, so that you have `Profitina/Chapter01/python`, `Profitina/Chapter01/java`, and so on. Always open a **folder** in VS Code (**File → Open Folder…**), never a single file. VS Code's terminal then starts inside that folder, so that commands such as `python3 milestones.py` find the files next to them. When VS Code asks whether you trust the files in the folder, answer yes for the book's folder: it is your own copy.

1.8.7 Checking the whole bench

Open a *new* terminal, so that it sees the updated PATH, and type each command of **Table 1.7**. On Windows, type `python` where the table says `python3`.

Table 1.7 The verification commands, what each should print, and the usual fix when it does not.

Command	A good answer begins with	If it fails
`code --version`	`1.`	Reinstall VS Code with **Add to PATH**, or run *Shell Command: Install 'code' command in PATH*
`javac -version`	`javac 25`	Only a runtime is installed, or an older JDK is earlier on the PATH
`java -version`	`openjdk version "25`	Install the JDK; check `JAVA_HOME`
`python3 --version`	`Python 3.14`	Windows: rerun the installer and tick **Add python.exe to PATH**

On the Linux machine on which this book's programs were run, the checks printed the following. The machine is a little behind the recommended versions, with JDK 21 and Python 3.11, which shows that the programs of the book run on these too.

```
$ javac -version
javac 21.0.10
$ java -version
openjdk version "21.0.10" 2026-01-20
$ python3 --version
Python 3.11.15
```

Typing four commands is quick, but a program can do the same check and explain its answer, and writing that program is a good way to see the PATH at work. **Listing 1.1** asks the operating system which of six tools it can find, exactly as a terminal would.

Listing 1.1 `check_setup.py`: reports the operating system, the Python that runs the program, and which tools of the bench the PATH can reach.

```python
# check_setup.py - report which tools of the workbench this terminal can see
import os
import platform
import shutil

TOOLS = ["code", "python3", "python", "java", "javac", "git"]


def main():
    print("Operating system :", platform.system(), platform.release())
    print("Python running   :", platform.python_version())
    print("Working folder   :", os.getcwd())
    print()
    missing = 0
    for tool in TOOLS:
        where = shutil.which(tool)       # searches the PATH, like a terminal does
        if where is None:
            missing += 1
            print(f"  {tool:<8} NOT FOUND on the PATH")
        else:
            print(f"  {tool:<8} {where}")
    print()
    if missing == 0:
        print("Every tool was found.")
    else:
        print(f"{missing} tool(s) not found; see Chapter 1 for the usual fixes.")


if __name__ == "__main__":
    main()
```

The function `shutil.which(tool)` does what the terminal does when you type a command: it walks the folders of the PATH in order and returns the full path of the first matching program, or `None` if there is none. The format `{tool:<8}` pads the name with spaces to eight characters, left-aligned, so that the second column lines up. Run it with `python3 check_setup.py` in the `Chapter01/python` folder:

```
Operating system : Linux 6.18.44-fc-v37
Python running   : 3.11.15
Working folder   : /home/claude/kit/Profitina/Code_new/Chapter01/python

  code     NOT FOUND on the PATH
  python3  /usr/bin/python3
  python   /usr/bin/python
  java     /usr/bin/java
  javac    /usr/bin/javac
  git      /usr/bin/git

1 tool(s) not found; see Chapter 1 for the usual fixes.
```

Your output will differ in the details, and it should. The machine used for the book is a server without a graphical desktop, so it has no VS Code, and the program says so honestly rather than pretending. On your laptop `code` should be found, and on Windows the paths look like `C:\Program Files\...`. The tool `git`, a version-control system, is not needed for this book, but the check shows whether you have it.

Java has no ready-made `which`, so **Listing 1.2** writes one. It reads the PATH with `System.getenv("PATH")`, splits it with `File.pathSeparator`, which is `;` on Windows and `:` elsewhere, and on Windows also tries the endings `.exe`, `.cmd` and `.bat`, because a Windows command named `javac` is really a file named `javac.exe`.

Listing 1.2 `CheckSetup.java`: the same report in Java, with a small hand-written search of the PATH.

```
// CheckSetup.java - report which tools of the workbench this terminal can see
import java.io.File;
import java.util.List;

public class CheckSetup {
    static final List<String> TOOLS =
            List.of("code", "python3", "python", "java", "javac", "git");

    // Search every folder of the PATH for the tool, as a terminal does.
    static String which(String tool) {
        String path = System.getenv("PATH");
        if (path == null) {
            return null;
        }
        boolean windows = System.getProperty("os.name").startsWith("Windows");
        String[] endings = windows ? new String[] {".exe", ".cmd", ".bat"}
                                   : new String[] {""};
        for (String dir : path.split(File.pathSeparator)) {
            for (String end : endings) {
                File f = new File(dir, tool + end);
                if (f.isFile() && f.canExecute()) {
                    return f.getPath();
                }
            }
        }
        return null;
    }

    public static void main(String[] args) {
        System.out.println("Operating system : " + System.getProperty("os.name")
                + " " + System.getProperty("os.version"));
        System.out.println("Java running     : " + Runtime.version());
        System.out.println("Working folder   : " + System.getProperty("user.dir"));
        System.out.println();
        int missing = 0;
        for (String tool : TOOLS) {
            String where = which(tool);
            if (where == null) {
                missing++;
                System.out.printf("  %-8s NOT FOUND on the PATH%n", tool);
            } else {
                System.out.printf("  %-8s %s%n", tool, where);
            }
        }
        System.out.println();
        if (missing == 0) {
```

Listing 1.2 (Continued) *CheckSetup.java: the same report in Java, with a small hand-written search of the PATH.*

```
            System.out.println("Every tool was found.");
        } else {
            System.out.println(missing
                    + " tool(s) not found; see Chapter 1 for the usual fixes.");
        }
    }
}
```

Compile and run it in the `Chapter01/java` folder with `javac CheckSetup.java` and then `java CheckSetup`:

```
Operating system : Linux 6.18.44-fc-v37
Java running     : 21.0.10+7-Ubuntu-124.04
Working folder   : /home/claude/kit/Profitina/Code_new/Chapter01/java

  code     NOT FOUND on the PATH
  python3  /usr/bin/python3
  python   /usr/bin/python
  java     /usr/bin/java
  javac    /usr/bin/javac
  git      /usr/bin/git

1 tool(s) not found; see Chapter 1 for the usual fixes.
```

The two programs agree, as they should: they ask the same question of the same PATH. The line `Java running` shows the full version string of the running Java; with Temurin 25 it begins with `25`.

1.8.8 Your first program, in two languages

Create the file `hello.py` in your `Chapter01/python` folder, type in **Listing 1.3**, and save it.

Listing 1.3 `hello.py`: the first program; it proves that the editor, the terminal and Python work together.

```
# hello.py - the first program of the Profitina book
import sys

try:
    name = input("What is your name? ")
except EOFError:                          # no keyboard, no file: nothing to read
    name = ""
name = name.strip()
if not name:
    name = "friend"                       # no input: greet a friend
print(f"Hello, {name}! Welcome to Profitina.")
print(f"Language version: Python {sys.version_info.major}.{sys.version_info.minor}")
```

Open the terminal in VS Code with **Terminal → New Terminal** and run the program with `python3 hello.py` (`python hello.py` on Windows). It asks for your name; type one and press Enter:

```
What is your name? Sari
Hello, Sari! Welcome to Profitina.
Language version: Python 3.11
```

Read the program line by line. `import sys` gives access to the `sys` module, which knows, among other things, which version of Python is running. `input(...)` prints its argument as a prompt and waits for a line from the keyboard. If there is no keyboard at all, for example when the program's input comes from an empty file, `input` cannot read anything and raises the exception `EOFError` (end of file); the `try` and `except` catch it and use an empty name instead. `strip()` removes spaces at both ends, so that a name typed as " Sari " is greeted as "Sari". An empty name becomes "friend". The *f-string* `f"Hello, {name}! ..."` builds a string with the value of `name` inserted between the braces. With Python 3.14 the last line says `Python 3.14`.

Now the Java version, **Listing 1.4**. Save it as `Hello.java` in the `Chapter01/java` folder. In Java, a public class must live in a file with exactly its name, capital letter included.

Listing 1.4 `Hello.java`: the first program in Java.

```java
// Hello.java - the first program of the Profitina book
import java.util.Scanner;

public class Hello {
    public static void main(String[] args) {
        Scanner in = new Scanner(System.in);
        System.out.print("What is your name? ");
        String name = in.hasNextLine() ? in.nextLine().strip() : "";
        if (name.isEmpty()) {
            name = "friend";                  // no input: greet a friend
        }
        System.out.println("Hello, " + name + "! Welcome to Profitina.");
        System.out.println("Language version: Java " + Runtime.version().feature());
    }
}
```

Compile and run it in two steps:

```
javac Hello.java
java Hello
```

```
What is your name? Sari
Hello, Sari! Welcome to Profitina.
Language version: Java 21
```

The first command runs the compiler, which reads `Hello.java` and writes `Hello.class`, a file of *bytecode* that the Java virtual machine understands. If `javac` prints nothing, it succeeded: silence is good news. The second command starts the virtual machine and asks it to run the class `Hello`; notice that you write the class name, without `.class`. Since Java 11 you may also type `java Hello.java`, and the launcher compiles and runs a single-file program in one step.

The Java program mirrors the Python one. A `Scanner` reads from `System.in`, the keyboard. `in.hasNextLine()` asks whether there is a line to read at all, which plays the role of Python's `EOFError` test; the expression `condition ? a : b` chooses `a` if the condition holds and `b` otherwise. `strip()` removes spaces at both ends, as in Python. `Runtime.version().feature()` is the major version of the running Java, 21 here and 25 with JDK 25.

Both programs handle the case of no input at all. Run `python3 hello.py < empty.txt` with an empty file, or on macOS and Linux simply `python3 hello.py < /dev/null`, and the program greets "friend". A program that behaves sensibly when its input is missing is a small example of a habit that runs through the whole book, and through Profitina: *fail loudly or fail gracefully, but never fail silently.*

1.8.9 A second program: days since the launch

The first program proved that the tools work. The second does something useful: it reads the milestones of **Table 1.1** from a small file and prints, for each one, the weekday and the number of days from the launch on 17 July 2026. The file `milestones.csv` is a *CSV* file (comma-separated values): the first line names the columns, and each following line is one record.

```
date,event
2026-07-03,First commit of the current code base (version 0.11.0)
2026-07-17,Production launch at app.profitina.com
2026-07-31,Sharing to X moves to Web Intent; no X API
2026-08-07,Two plans only: Free and Pro
2026-08-19,Self-service privacy rights panel
2026-08-31,First university class of about 250 students
2026-09-01,One Text Many Platforms
2026-09-09,Organization Visibility
2026-09-24,Version 0.14.5-testnet
```

Listing 1.5 reads it with Python's `csv` module, turns each date into a `date` object, sorts the records by date, and prints a table. An optional command-line argument gives the "as of" date; without one, the program uses today's date.

Listing 1.5 `milestones.py`: reads the milestones, prints their distance from the launch, and applies Equation (1.1) to the number of days live.

```python
# milestones.py - read Profitina's milestones and count days from the launch
import csv
import sys
from datetime import date

LAUNCH = date(2026, 7, 17)


def load(filename):
    rows = []
    with open(filename, encoding="utf-8", newline="") as f:
        for row in csv.DictReader(f):
            rows.append((date.fromisoformat(row["date"]), row["event"]))
    rows.sort()                                 # oldest first, whatever the file order
    return rows


def main():
    if len(sys.argv) > 1:
        as_of = date.fromisoformat(sys.argv[1])
    else:
        as_of = date.today()
    rows = load("milestones.csv")
    print(f"{'Date':<11}{'Day':<5}{'From launch':>12}  Event")
    for when, event in rows:
        offset = (when - LAUNCH).days       # negative: before the launch
        day = when.strftime("%a")           # Mon, Tue, ...
        print(f"{when.isoformat():<11}{day:<5}{offset:>+12}  {event}")
    live = (as_of - LAUNCH).days
    print(f"\nAs of {as_of.isoformat()}, Profitina has been live for {live} days"
          f" ({live // 7} weeks and {live % 7} days).")


if __name__ == "__main__":
    main()
```

```
Date       Day   From launch  Event
2026-07-03 Fri           -14  First commit of the current code base (version 0.11.0)
2026-07-17 Fri            +0  Production launch at app.profitina.com
2026-07-31 Fri           +14  Sharing to X moves to Web Intent; no X API
2026-08-07 Fri           +21  Two plans only: Free and Pro
2026-08-19 Wed           +33  Self-service privacy rights panel
2026-08-31 Mon           +45  First university class of about 250 students
2026-09-01 Tue           +46  One Text Many Platforms
2026-09-09 Wed           +54  Organization Visibility
2026-09-24 Thu           +69  Version 0.14.5-testnet

As of 2026-09-24, Profitina has been live for 69 days (9 weeks and 6 days).
```

That is the output of `python3 milestones.py 2026-09-24`. A few details are worth noticing.

- `csv.DictReader` turns each line into a dictionary whose keys come from the header line, so the code can write `row["date"]` instead of remembering that the date is column 0.
- `date.fromisoformat("2026-07-17")` turns text in the international format *year-month-day* (ISO 8601) into a date. Subtracting two dates gives a `timedelta`, and its `.days` is the difference in

whole days. The calendar arithmetic, with months of 28 to 31 days and leap years, is done by the library, correctly.

- `rows.sort()` sorts a list of pairs by their first element, the date, and uses the second only to break ties. The program therefore prints the milestones in date order even if the file is edited carelessly.
- The format `{offset:>+12}` right-aligns the number in twelve characters and always shows its sign, so a milestone before the launch shows `-14`.
- `live // 7` and `live % 7` are the w and r of Equation (1.1).

Listing 1.6 is the same program in Java.

Listing 1.6 `Milestones.java`: the same table in Java, with `LocalDate` and `ChronoUnit.DAYS`.

```java
// Milestones.java - read Profitina's milestones and count days from the launch
import java.io.IOException;
import java.nio.charset.StandardCharsets;
import java.nio.file.Files;
import java.nio.file.Path;
import java.time.LocalDate;
import java.time.format.TextStyle;
import java.time.temporal.ChronoUnit;
import java.util.ArrayList;
import java.util.List;
import java.util.Locale;

public class Milestones {
    static final LocalDate LAUNCH = LocalDate.of(2026, 7, 17);

    record Milestone(LocalDate when, String event) { }

    static List<Milestone> load(String filename) throws IOException {
        List<Milestone> rows = new ArrayList<>();
        List<String> lines = Files.readAllLines(Path.of(filename),
                StandardCharsets.UTF_8);
        for (String line : lines.subList(1, lines.size())) {   // skip the header
            if (line.isBlank()) {
                continue;
            }
            int comma = line.indexOf(',');
            rows.add(new Milestone(LocalDate.parse(line.substring(0, comma)),
                    line.substring(comma + 1)));
        }
        rows.sort((a, b) -> a.when().compareTo(b.when()));  // oldest first
        return rows;
    }

    public static void main(String[] args) throws IOException {
        LocalDate asOf = args.length > 0 ? LocalDate.parse(args[0])
                                         : LocalDate.now();
        List<Milestone> rows = load("milestones.csv");
        System.out.printf("%-11s%-5s%12s  %s%n",
                "Date", "Day", "From launch", "Event");
        for (Milestone m : rows) {
            long offset = ChronoUnit.DAYS.between(LAUNCH, m.when());
            String day = m.when().getDayOfWeek()
                    .getDisplayName(TextStyle.SHORT, Locale.ENGLISH);
            System.out.printf("%-11s%-5s%+12d  %s%n",
                    m.when(), day, offset, m.event());
        }
```

Listing 1.6 (Continued) *Milestones.java: the same table in Java, with LocalDate and ChronoUnit.DAYS.*

```
        long live = ChronoUnit.DAYS.between(LAUNCH, asOf);
        System.out.printf("%nAs of %s, Profitina has been live for %d days"
                + " (%d weeks and %d days).%n", asOf, live, live / 7, live % 7);
    }
}
```

Run with `javac Milestones.java` and `java Milestones 2026-09-24`, it prints exactly the same eleven lines as the Python version. The Java program shows three features you will meet again. A `record` declares a small class whose only job is to hold values, here a date and an event, in one line. `Files.readAllLines` reads a whole text file into a list of strings, and `lines.subList(1, lines.size())` skips the header. And `ChronoUnit.DAYS.between(a, b)` counts the days from `a` to `b`, negative when `b` comes first.

The program is short, but it is not a toy. It has every part of the programs in later chapters: it reads data from a file, turns text into typed values, computes something that is easy to get wrong by hand, and prints a result that a person can check.

1.8.10 Running and debugging in VS Code

Running from the editor. With the extensions of **Table 1.6** installed, a Python or Java file shows a **Run** triangle at the top right of the editor. Pressing it runs the file in VS Code's terminal. For a Java file the extension compiles it for you first. For a Python file, the first time, VS Code asks which Python interpreter to use; choose the 3.14 you installed. The status bar shows the chosen interpreter, and clicking it lets you change it. Programs that take command-line arguments, such as `milestones.py 2026-09-24`, are simplest to run from the terminal.

Feeding input from a file. A program that reads the keyboard can read a file instead through *redirection*: `python3 hello.py < name.txt` or `java Hello < name.txt` on macOS, Linux and the Windows Command Prompt, and `Get-Content name.txt | python hello.py` in PowerShell, which does not understand `<`.

Debugging: watching the program think. When a program gives a wrong answer, the fastest way to find out why is to *watch* it. Click in the margin to the left of a line number: a red dot appears, a **breakpoint**. Press **F5** and choose *Python Debugger* or *Java*. The program runs until it reaches the red dot and stops. The **Variables** panel then shows the value of every variable at that moment, and you can hover over any name in the code to see its value. **F10** runs one line, **F11** steps into a function call, **Shift+F11** steps out again, and **F5** continues to the next breakpoint. Put a breakpoint on the `print` inside the loop of **Listing 1.5** and watch `when` and `offset` change from row to row. It is the best way to understand a loop that someone else wrote.

> **WHEN SOMETHING GOES WRONG.** *"python is not recognized"* (Windows): Python was installed without **Add python.exe to PATH**; rerun the installer, choose *Modify*, and tick it; then open a new terminal. *Typing `python` opens the Microsoft Store* (Windows): turn off the *App execution aliases* for `python.exe` in Settings, or install Python from python.org. *`javac` is not found but `java` is*: you installed a runtime; install the JDK. *`error: class Hello is public, should be declared in a file named Hello.java`*: rename the file so that it matches the class exactly, capital letters included. *`Error: Could not find or load main class Hello.class`*: you typed `java Hello.class`; type `java Hello`. *`FileNotFoundError: milestones.csv`*: the terminal is in a different folder from the file; `cd` into the folder, or open the folder in VS Code with **File → Open Folder**. *Red squiggles everywhere in a Java file, but it compiles*: wait for the Java extension to finish starting, or run *Java: Clean Java Language Server Workspace* from the Command Palette (**Ctrl+Shift+P**).

With the bench standing, every listing in the rest of this book is one copy and one command away from being yours.

1.9 Profitina in Practice

The story of this chapter is visible in the product itself, and a few places are worth opening in a browser as you read.

The public faces. `profitina.com` is the bilingual marketing site; `app.profitina.com` is the application, whose window title reads "Profitina — Dasbor GEO". The application's interface switches between Indonesian and English and between light and dark themes, and the same choices apply on the site. The marketing site publishes a file `/llms.txt`, a short machine-readable summary for language models, which Chapter 2 builds for Profitina Laundry and Chapter 10 examines in detail.

The versioned whitepaper. The public whitepaper, version 24 in September 2026, is served by the application itself and exists in Indonesian and English [1]. It is the safest single source for the facts in this book, and every chapter that quotes it says so. Its disclaimer is a founding idea in a single sentence: visibility results are methodological estimates, not guarantees of what any third-party AI engine will actually say.

The honest version string. The application's version at the time of writing is `0.14.5-testnet`. The *0.x* says that the platform is young and still changing its surface; *testnet* says that the PROFIT coin lives on a test network. Both are statements of fact that the product makes about itself, in the place where every user can see them.

The course channels. The community's course channels, with their moderators, pinned announcements and neutral handles, are the part of the platform that a student is most likely to meet first. The rules that govern them, such as what a moderator may and may not do and why a handle is not a student number, are described in Chapters 4 and 11.

A local habit in the code. Even a program as small as `milestones.py` touches a Profitina habit: time zones. The platform's daily quotas and caps reset at 00:00 WIB (Western Indonesian Time, UTC+7), not at midnight UTC, which would be 07:00 in Jakarta. A date without a time zone, like the ones in `milestones.csv`, is fine for counting days between milestones; a *moment* such as "the quota resets now" needs a zone. Chapter 9's scheduler lab makes that difference concrete.

Where to write. For questions about the platform, the official contact is `info@profitina.com` [2].

1.10 Common Misunderstandings

"Profitina asks ChatGPT about my business and reports what it says." It does not. The answer engines are the targets whose *readiness* to cite you is estimated from observable signals; Profitina does not query them and does not depend on them. Chapter 2 explains what is measured instead, and Chapter 3 shows the scan.

"A score of 100 guarantees that AI will recommend me." No score can guarantee what a third-party engine will say, and the platform says so in its terms [2]. The score estimates readiness; the fixes improve the odds.

"Profitina's AI is a wrapper around a commercial chatbot." Every AI feature runs on local models in two lanes on Profitina's own machine; hosted-AI code paths were removed.

"PROFIT is a cryptocurrency I can trade." PROFIT lives on Solana's devnet, a test network whose tokens have no monetary value. Selling is disabled. A main-network launch is a plan that depends on licensing and an audit.

"The founder built the product first and wrote the teaching material afterwards to promote it." The dates run the other way: the teaching career began in 1997, and the course books use the product as one case among many because the courses are what the product is made of.

"If `java -version` works, Java is installed." Only the runtime may be installed. You need `javac` as well; check both, and check that they report the same version.

"The PATH is updated as soon as the installer finishes." A terminal reads the PATH when it starts. After installing anything, open a *new* terminal, and on Windows sometimes a new VS Code window, before concluding that something failed.

"I can open a single `.py` or `.java` file in VS Code and work from there." You can, but the terminal will then start in some other folder and programs will not find their data files. Open the chapter's folder.

1.11 Chapter Summary

- **Profitina** is Indonesia's AI Visibility (GEO) platform. It estimates how ready a business, site, researcher or organization is to be found and quoted by AI answer engines, from observable signals, and helps fix what it finds. It has been live at `app.profitina.com` since **17 July 2026**.
- It was founded and is built by a **lecturer in ITS's Department of Informatics** who has taught since 1997–1998 and whose research spans expert systems, ontologies, fuzzy logic, genetic algorithms, neural networks and image processing.
- The founder teaches **DAA, Web Programming and OOP**, the three subjects the platform is made of, and a **library of course books** uses Profitina as a case; **Profitina Laundry** is the fictional business used for hands-on practice. A class of about **250 students** has used the platform since 31 August 2026.
- The **timeline** of **Table 1.1** and **Figure 1.1** shows pace (versions 0.11 to 0.14 in twelve weeks), subtraction (the X API, the Agency tier and advertising removed) and dating everything.
- The **founding ideas** are a transparent, inclusive, self-reliant mission; visibility and useful participation in one system; local AI in two lanes; tested releases (1,577 tests passing at version 0.14.5); and Indonesia first (bilingual, UU PDP, QRIS).
- **PROFIT** is live on Solana's devnet; selling is disabled; the main-network launch is roadmap. Nothing in the book is investment advice.
- The book has **thirteen chapters** in four groups (**Figure 1.2**); every lab is in **Python and Java** with real output; every changing number carries its date.
- The **bench** is VS Code with the Python extension and the Extension Pack for Java, **JDK 25** and **Python 3.14**, checked with the commands of **Table 1.7** or with `check_setup`. Days split into weeks and days by Equation (1.1).

1.12 Exercises

Exercise 1.1. In your own words and in at most three sentences each, explain (a) what Profitina measures, (b) what it does *not* use in order to measure it, and (c) why the platform calls its results "estimates".

Exercise 1.2. Using **Table 1.1**, list the milestones that *removed* something from the product, and for each one give the reason suggested in this chapter. What do they have in common?

Exercise 1.3. Match each of the three courses EF234405, EF234301 and EF234302 with one concrete part of Profitina described in this chapter, and explain the match in one sentence each.

Exercise 1.4. Which of the following statements are live facts on 24 September 2026, and which are roadmap? (a) PROFIT can be bought with rupiah through QRIS. (b) PROFIT can be sold for rupiah. (c) PROFIT exists on Solana's main network. (d) PROFIT can be spent on scan top-ups. (e) Regional expansion beyond Indonesia.

Exercise 1.5. Apply Equation (1.1) by hand to $d = 45$, $d = 7$, $d = 0$ and $d = 183$. Show w and r for each, and check that $7w + r = d$.

Exercise 1.6. A friend on Windows reports three messages. Explain each and give the fix. (a) `'python' is not recognized as an internal or external command`, although the installer finished without error. (b) `java -version` prints `openjdk version "25.0.1"`, but `javac -version` prints `javac 17.0.2`. (c) After typing `java Hello.class`: `Error: Could not find or load main class Hello.class`.

Exercise 1.7. Run `hello.py` and `Hello.java` with an empty input file, then with a file containing only spaces. Predict both outputs before running them, and explain which line of each program decides the result.

Exercise 1.8. Write a program, in Python and in Java, that takes two dates in the form `YYYY-MM-DD` on the command line and prints how many days lie between them, the weekday of each, and the difference in weeks and days by Equation (1.1). Use it to find how many days after Profitina's launch the implementing regulation of Indonesia's personal-data protection law, PP No. 33 of 2026, takes effect on 16 January 2027.

Exercise 1.9. Add the milestone "2026-09-14,Health-verified release gate" to the *end* of `milestones.csv` and run `milestones.py 2026-09-24` again. Where does the new line appear in the output, and which line of the program puts it there? What would happen without that line?

Exercise 1.10. Modify `check_setup.py` or `CheckSetup.java` so that it also reports the *version* of Python and Java that the terminal would run, by running `python3 --version` and `javac -version` as child processes. (Hint: Python's `subprocess.run` with `capture_output=True, text=True`; Java's `ProcessBuilder` with `redirectErrorStream(true)`.) Why might the version the terminal runs differ from the version running `check_setup` itself?

Exercise 1.11. Using **Table 1.2**, choose the row that describes you best and write a reading plan for this book: which chapters, in which order, and which labs you will run. Then write one question you hope the book will answer; check at the end of Chapter 13 whether it did.

Exercise 1.12. The chapter says that a *date* is enough for counting days between milestones but a *moment* needs a time zone. Give a concrete example, using WIB (UTC+7) and UTC, of a daily quota that would reset at the wrong local hour if the program used the UTC date.

1.13 Solutions to the Exercises

Solution 1.1. (a) Profitina measures how *ready* a business, website, researcher or organization is to be found and quoted by AI answer engines, from signals anyone can observe on the public web: whether text can be extracted cleanly, whether structured data describes the business, and whether the technical foundations are sound. It then suggests prioritized fixes. (b) It does not ask

the answer engines themselves; ChatGPT, Gemini and Perplexity are targets, not components, for reasons of independence, privacy and a rule against paid third-party APIs. (c) Because no one outside an engine can know what it will say tomorrow; engines change constantly. A signal-based score estimates the odds and can be explained, but it cannot be a guarantee, and saying so is part of the transparent mission.

Solution 1.2. The X API was removed on 31 July 2026, because the platform could not verify sharing through it and did not want to depend on a paid third-party API; sharing moved to Web Intent, where the user posts from their own account. The Agency tier was removed on 6–7 August 2026, for simplicity: two plans are easier to understand, to support and to keep honest. Advertising was ruled out permanently at the same time. In common: each removal reduced what the product *promises* to what it can *deliver and verify*, and each came with a written reason.

Solution 1.3. EF234405, *Design and Analysis of Algorithms*: the costs of queries and schedules, for example the feed page that went from 868 ms to 11 ms once the number of database queries per page fell. EF234301, *Web Programming*: the application itself and the community widget embedded in the marketing site. EF234302, *Object-Oriented Programming*: the scanner's many independent checks, each an object that reports its own result and explanation behind one common interface.

Solution 1.4. (a) Live. (b) Roadmap; selling is disabled until the main network and regulatory clarity. (c) Roadmap; today PROFIT is on devnet. (d) Live. (e) Roadmap (Phase 4).

Solution 1.5. For $d = 45$: $45/7 = 6.43\ldots$, $w = 6$, $r = 45 - 42 = 3$, and $7 \cdot 6 + 3 = 45$. For $d = 7$: $w = 1$, $r = 0$, and $7 + 0 = 7$. For $d = 0$: $w = 0$, $r = 0$. For $d = 183$: $183/7 = 26.14\ldots$, $w = 26$, $r = 183 - 182 = 1$, and $182 + 1 = 183$. The check $7w + r = d$ holds by the definition of r in Equation (1.1); what the floor guarantees is the other fact, $0 \le r < 7$.

Solution 1.6. (a) The installer's **Add python.exe to PATH** box was not ticked, or the terminal was opened before the installation. Open a new terminal; if the message remains, rerun the installer, choose *Modify* and tick the box, or add Python's folder to the user Path by hand. (b) Two JDKs are installed, and an older JDK 17 comes earlier on the PATH for `javac` than the JDK 25 does for `java`. Put JDK 25's `bin` folder first in the PATH (and set `JAVA_HOME` to JDK 25), or uninstall JDK 17, then open a new terminal. (c) The launcher takes a *class name*, not a file name; type `java Hello`.

Solution 1.7. With an empty file, Python's `input` raises `EOFError`, the `except` sets `name` to `""`, and the program prints `Hello, friend! Welcome to Profitina.`; Java's `in.hasNextLine()` returns `false`, the name is `""`, and the output is the same. With a file containing only spaces, both programs *do* read a line, but `strip()` turns it into `""`, so both again greet "friend". The deciding lines are `name = name.strip()` followed by `if not name:` in Python, and `in.nextLine().strip()` followed by `if (name.isEmpty())` in Java. Without `strip()`, both would greet a name made of spaces: `Hello,  ! Welcome to Profitina.`

Solution 1.8. **Listing 1.7** and **Listing 1.8** check the number of arguments, parse both dates (reporting a bad one instead of crashing), and apply Equation (1.1) to the absolute difference. Python's `divmod(a, 7)` returns the pair $(\lfloor a/7 \rfloor, a \bmod 7)$ in one call.

Listing 1.7 `sol_countdown.py`: days, weeks and weekdays between two dates.

```python
# sol_countdown.py - days, weeks and weekday between two dates (Exercise 1.8)
import sys
from datetime import date


def main():
    if len(sys.argv) != 3:
        print("usage: python3 sol_countdown.py FROM TO   (dates as YYYY-MM-DD)")
        sys.exit(1)
    try:
        start = date.fromisoformat(sys.argv[1])
        end = date.fromisoformat(sys.argv[2])
    except ValueError as e:
        print("Not a date:", e)
        sys.exit(1)
    days = (end - start).days
    weeks, rest = divmod(abs(days), 7)       # floor division and remainder
    word = "after" if days >= 0 else "before"
    print(f"{end.isoformat()} ({end.strftime('%A')}) is {abs(days)} days {word}"
          f" {start.isoformat()} ({start.strftime('%A')}),")
    unit = "day" if rest == 1 else "days"
    print(f"that is {weeks} weeks and {rest} {unit}.")


if __name__ == "__main__":
    main()
```

Listing 1.8 `SolCountdown.java`: the same computation with `LocalDate` and `ChronoUnit`.

```java
// SolCountdown.java - days, weeks and weekday between two dates (Exercise 1.8)
import java.time.LocalDate;
import java.time.format.DateTimeParseException;
import java.time.format.TextStyle;
import java.time.temporal.ChronoUnit;
import java.util.Locale;

public class SolCountdown {
    static String weekday(LocalDate d) {
        return d.getDayOfWeek().getDisplayName(TextStyle.FULL, Locale.ENGLISH);
    }

    public static void main(String[] args) {
        if (args.length != 2) {
            System.out.println("usage: java SolCountdown FROM TO"
                    + "   (dates as YYYY-MM-DD)");
            System.exit(1);
        }
        LocalDate start;
        LocalDate end;
        try {
            start = LocalDate.parse(args[0]);
            end = LocalDate.parse(args[1]);
        } catch (DateTimeParseException e) {
            System.out.println("Not a date: " + e.getParsedString());
            System.exit(1);
            return;
        }
        long days = ChronoUnit.DAYS.between(start, end);
        long weeks = Math.abs(days) / 7;          // floor division ...
        long rest = Math.abs(days) % 7;           // ... and remainder
        String word = days >= 0 ? "after" : "before";
        System.out.println(end + " (" + weekday(end) + ") is " + Math.abs(days)
                + " days " + word + " " + start + " (" + weekday(start) + "),");
        String unit = rest == 1 ? "day" : "days";
        System.out.println("that is " + weeks + " weeks and " + rest + " "
                + unit + ".");
    }
}
```

Running `python3 sol_countdown.py 2026-07-17 2027-01-16` and `java SolCountdown 2026-07-17 2027-01-16` gives the same answer:

```
2027-01-16 (Saturday) is 183 days after 2026-07-17 (Friday),
that is 26 weeks and 1 day.
```

PP No. 33 of 2026 takes effect 183 days after Profitina's launch. With the dates in the other order, `2026-09-24 2026-07-17`, both programs print `2026-07-17 (Friday) is 69 days before 2026-09-24 (Thursday),` followed by `that is 9 weeks and 6 days.`

Solution 1.9. The new line appears between "Organization Visibility" (+54) and "Version 0.14.5-testnet" (+69), with the offset +59 and the weekday `Mon`, because `rows.sort()` in Python and `rows.sort(...)` in Java order the records by date before printing. Without the sort, the line would be printed last, out of order, and a reader scanning the table would conclude that the gate came after the release of 24 September.

Solution 1.10. In Python, for each command:

```
import subprocess
result = subprocess.run([tool, "--version"], capture_output=True, text=True)
print(tool, (result.stdout or result.stderr).strip())
```

(`javac` understands `-version`; older versions print it on the error stream, which is why both streams are read.) In Java, `new ProcessBuilder("javac", "-`

`version").redirectErrorStream(true).start()` and then read `process.getInputStream()`. The versions can differ because `check_setup` reports the interpreter that is *running it*, which may have been started by full path or by VS Code's interpreter picker, while the terminal runs whichever program comes *first on the PATH*. When they differ, VS Code and the terminal are using two different installations, a frequent cause of "it works in the terminal but not in VS Code".

Solution 1.11. Answers are personal. An example for a student: Chapters 1 and 2 in full; Chapter 3 and its lab G1; then Chapters 4 and 5 with labs G2 and G3; then Chapter 9 with G4 and Chapter 11 with G7; the rest in order. A question to carry: "How can a platform know that a post is useful without reading the writer's mind?" (Chapter 7).

Solution 1.12. Suppose a quota must reset at 00:00 WIB. The moment 00:00 WIB on 25 September is 17:00 UTC on 24 September. A program that asks for "today's date" in UTC keeps answering 24 September until 00:00 UTC, which is 07:00 WIB on 25 September. So a user in Surabaya who uses up their quota at 23:30 WIB on 24 September would find it *not* renewed at midnight, and renewed only at seven in the morning. The fix is to compute the date in the zone `Asia/Jakarta`, with `zoneinfo.ZoneInfo("Asia/Jakarta")` in Python or `ZoneId.of("Asia/Jakarta")` in Java.

References

[1] Profitina, *Profitina Whitepaper — Public Version v24*, Sep. 2026. [Online]. Available: https://app.profitina.com/static/whitepaper/

[2] Profitina, "Kebijakan Privasi" and "Syarat dan Ketentuan" (Privacy Policy and Terms and Conditions, v1.1, effective Aug. 6, 2026). [Online]. Available: https://app.profitina.com/legal/privacy

[3] ANTARA, "Survei APJII: Penetrasi internet di Indonesia 2026 capai 81,7 persen," May 21, 2026. [Online]. Available: https://www.antaranews.com/berita/5576225/survei-apjii-penetrasi-internet-di-indonesia-2026-capai-817-persen

[4] Microsoft, "Visual Studio Code 1.139," *Visual Studio Code Release Notes*, Sep. 23, 2026. [Online]. Available: https://code.visualstudio.com/updates/v1_139

[5] Inside.java, "JDK 27 is now available," Sep. 15, 2026. [Online]. Available: https://inside.java/2026/09/15/jdk-27-available/

[6] Oracle, "Oracle Java SE Support Roadmap." [Online]. Available: https://www.oracle.com/java/technologies/java-se-support-roadmap.html (accessed Sep. 2026).

[7] Eclipse Foundation, "Eclipse Temurin," *Adoptium*. [Online]. Available: https://adoptium.net (accessed Sep. 2026).

[8] Python Software Foundation, "Python 3.14.7," Aug. 5, 2026. [Online]. Available: https://www.python.org/downloads/latest/python3.14/

[9] Microsoft, "Getting Started with Python in VS Code," *Visual Studio Code Documentation*. [Online]. Available: https://code.visualstudio.com/docs/python/python-tutorial (accessed Sep. 2026).

[10] Microsoft, "Getting Started with Java in VS Code," *Visual Studio Code Documentation*. [Online]. Available: https://code.visualstudio.com/docs/java/java-tutorial (accessed Sep. 2026).

Chapter 2

From Search to Answers: The Rise of GEO

By the end of this chapter you will be able to explain, to a shop owner or to an engineer, why being *found* online is no longer the same thing as being *ranked*: how a search engine builds its list of links, how an AI answer engine instead reads sources and writes one answer, and what the latest measurements of 2025 and 2026 say about the size of that shift. You will know where the term GEO, *Generative Engine Optimization*, comes from, what the research paper that coined it actually measured, and how Profitina defines the discipline for its own users. You will understand the three small machine-readable files and blocks that every GEO effort begins with: schema.org structured data in JSON-LD, `robots.txt`, and the newer `llms.txt`, including what each one can and cannot do. And you will have built lab G6, a generator that turns a small file of business facts for Profitina Laundry into all three, plus a checker that reads them back and proves they are valid, in Python and in Java.

2.1 Where We Left Off

Chapter 1 told Profitina's story and set up the bench: VS Code, Python and a JDK, tested with two small programs. It also made a claim it did not yet explain. Profitina, it said, measures how ready a business is to be *found and quoted by AI answer engines*. Why would a business need that? For twenty-five years, a business that wanted to be found online worked on its position in Google's list of results, and an entire profession, search engine optimization, grew up to help it. What changed?

This chapter answers that question in five steps. Section 2.2 recalls how the list of links works and why it shaped the web. Section 2.3 explains what an answer engine does differently, in plain words. Section 2.4 puts numbers on the change, all from 2025 and 2026 and all with their sources. Section 2.5 goes back to the research paper that named the new discipline, and Section 2.6 gives Profitina's own definition. Then Section 2.7 introduces the machine-readable layer of the web, and Section 2.8 builds it for Profitina Laundry.

2.2 Twenty-Five Years of the List of Links

2.2.1 Crawl, index, rank

A web search engine does three jobs, and it helps to keep them apart.

It **crawls**. A program called a *crawler*, or *bot*, fetches a page, finds the links in it, and adds the linked pages to a queue of pages to fetch next. Starting from a few well-known sites and following links, a crawler eventually visits a large part of the public web. Before fetching pages from a site, a polite crawler reads the site's `robots.txt` file, which says what the site's owner allows; Section 2.7.3 explains how.

It **indexes**. For every page it fetched, the engine records which words appear where, much as the index at the back of a book records on which pages each term appears. An *inverted index* maps

each word to the list of pages that contain it, so that the engine can answer "which pages contain *laundry* and *Rungkut*?" in a fraction of a second without reading the web again.

It **ranks**. Many pages contain the words of a query; the engine must decide which to show first. Google's early answer, PageRank, treated a link from page A to page B as a vote by A for B, and votes from important pages as worth more [1]. Modern ranking combines hundreds of signals, but the output has kept the same shape for a quarter of a century: an ordered **list of links**, ten to a page, each with a title and a snippet of text.

2.2.2 The person chooses

The list of links has one property that shaped the whole commercial web: *the person chooses*. The engine proposes an order, but a human being reads the titles and snippets, decides which look promising, clicks one or several, and reads the pages themselves. The website receives a visit; the business on it gets its chance to persuade.

Search engine optimization (SEO) is the discipline of improving a page's position in that list. Its tools are familiar: pages that load fast and can be crawled, titles and headings that contain the words people search for, content that deserves links from other sites. SEO is not going away, and Profitina never says it is. But its unit of success, a *position in a list*, assumes that there is a list and that a person will look at it.

2.3 The Answer Engine Arrives

2.3.1 One answer instead of ten links

An **answer engine** accepts the same kind of question, but it does not return a list for the person to sort through. It returns an *answer*: a few sentences or paragraphs of fluent text, written for this question, often with small numbered references that point to the sources it used. ChatGPT with search, Google's AI Overviews and AI Mode, Gemini, Microsoft Copilot and Perplexity are the best-known examples in 2026.

> **DEFINITION.** A *generative engine*, or *answer engine*, is a system that combines a conventional search step, which retrieves documents relevant to a query, with a large language model that reads those documents and generates a single response grounded in them, usually citing some of the sources inline [2].

Inside, most answer engines follow a pattern known as *retrieval-augmented generation* [3]. The question is used to *retrieve* a handful of relevant passages, either from the engine's own web index or from a partner's search index. Those passages are placed in front of a language model together with the question. The model then *generates* an answer, and in doing so it may quote a passage, restate a fact from it, or ignore it altogether. The model is not searching when it writes; it is summarizing what the retrieval step handed it, plus what it learned during training.

Figure 2.1 puts the two experiences side by side for a question a Surabaya customer might ask.

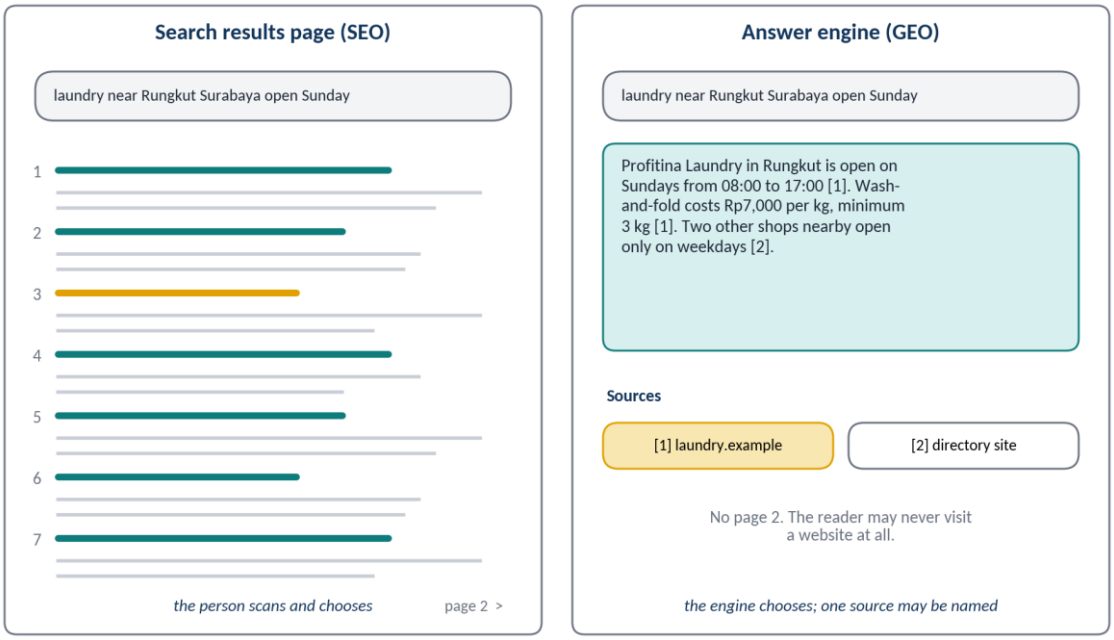


Figure 2.1 The same question on a search results page (left) and in an answer engine (right). On the left the person scans a ranked list and chooses; on the right the engine has already chosen, and names only one or two sources.

2.3.2 What changes for a business

Look at the right-hand side of **Figure 2.1** from the point of view of the laundry's owner. Three things have changed.

The engine chooses. In a list, position three still gets some clicks. In an answer, a business is either *in the sentence* or it is not. Profitina's marketing site puts it this way: GEO "competes for a sentence inside the answer itself — where there is no page two, and often only one source gets named" [4].

The reader may never visit. If the answer already says that the laundry opens on Sundays at 08:00 and charges Rp7,000 per kilogram, the customer has what they wanted without opening any website. The visit, which SEO was built to win, may never happen. Being *quoted correctly* becomes the goal.

The engine must be able to extract. A language model can only restate a fact that its retrieval step found and could read. A price hidden in a picture of a menu, opening hours written only in a promotional video, or an address split across three decorative boxes may be invisible to it. A page that states its facts plainly, near the top, in sentences a machine can lift out, has an advantage that has little to do with its position in any list.

> **KEY INSIGHT.** In a list of links, the unit of success is a *position*. In an answer, it is a *citation*: being the source whose fact, quotation or recommendation the engine repeats. GEO is the discipline of earning citations; SEO remains the discipline of earning positions. They complement each other.

2.4 The Numbers, 2025–2026

A shift in behavior is only interesting if it is large. This section collects the most useful measurements available in September 2026, each with its date and its source, and explains how

to read them. **Table 2.1** gathers them in one place; **Figure 2.2** draws the three that are easiest to compare.

Table 2.1 Measurements of the shift from lists to answers, 2025–2026, with the date and source of each. "M" means million.

Measurement	Value	When	Source
Visits that clicked a traditional result, when an AI summary appeared / did not appear	8% / 15%	March 2025 (U.S., 900 adults, 68,879 searches)	Pew Research Center [5]
Visits that clicked a link *inside* the AI summary	1%	March 2025	Pew Research Center [5]
Google searches showing an AI summary	18%	March 2025	Pew Research Center [5]
Google searches showing an AI Overview	from 15% to 43% over one year	reported July 2026	Similarweb via TechCrunch [6]
Visits to Google AI Mode	126 M (June 2025) → 279 M (May 2026)	2025–2026	Similarweb via TechCrunch [6]
U.S. ChatGPT desktop queries whose answer included citations	6.8%	May 2026	Similarweb via TechCrunch [6]
ChatGPT weekly active users	more than 900 M	February 2026	OpenAI, reported by Search Engine Land [7]
Gemini monthly active users	950 M (750 M at Q4 2025)	Q2 2026, reported 22 July 2026	Alphabet, reported by Forbes [8]
Perplexity queries per month	780 M, as last disclosed	May 2025	Perplexity's CEO, reported by a secondary source [9]
Internet users in Indonesia	235.3 M of 287.3 M people (81.72%)	2026 survey, released May 2026	APJII, reported by ANTARA [10]

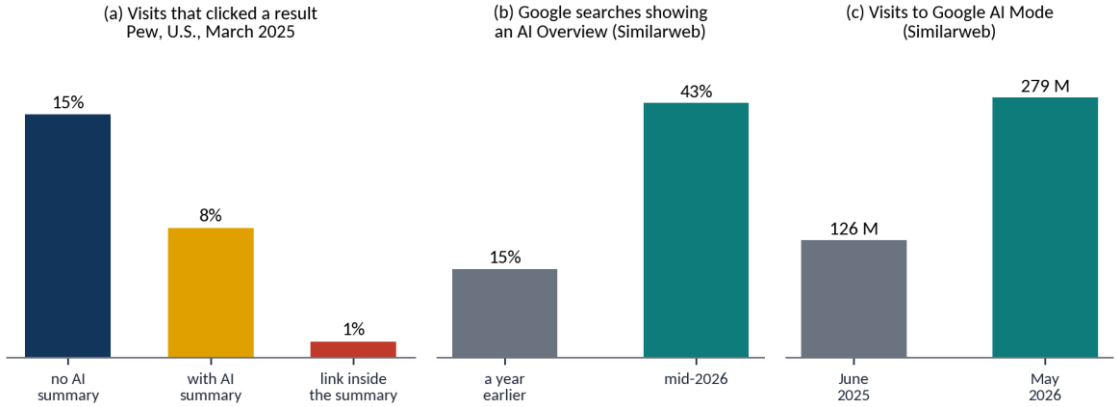


Figure 2.2 Three measurements from **Table 2.1**: (a) the share of Google visits that clicked a result, with and without an AI summary (Pew, March 2025); (b) the share of Google searches showing an AI Overview a year apart; (c) visits to Google AI Mode (both Similarweb, reported July 2026).

2.4.1 Fewer clicks when an answer appears

The most careful measurement comes from the Pew Research Center, which in March 2025 recorded the actual browsing of 900 U.S. adults who agreed to take part, and analyzed 68,879 Google searches they made [5]. In the visits where Google showed an AI summary, people clicked a traditional search result 8% of the time; in the visits without one, 15% of the time. They clicked a link inside the AI summary itself in only 1% of visits. And they were more likely to end their

browsing session altogether after a page with an AI summary (26% of such visits) than after one without (16%).

How big is a fall from 15% to 8%? A useful way to compare two rates is the *relative change*: if a rate moves from a to b, the relative change is

$$\Delta_{\text{rel}} = \frac{b - a}{a} \tag{2.1}$$

For Pew's numbers, $a = 15$ and $b = 8$, so by Equation (2.1) $\Delta_{\text{rel}} = (8 - 15)/15 = -7/15 \approx -0.47$: when a summary appeared, the chance of a click on a traditional result fell by about 47 percent, almost by half. Keep the difference between this and the *absolute* change clear: the rate fell by 7 *percentage points*, from 15 to 8, which is a fall of 47 *percent* of the original rate. Newspapers often confuse the two; this book does not.

2.4.2 More answers, more often

The second group of numbers measures how *often* answers appear. According to Similarweb data reported by TechCrunch in July 2026, Google's AI Overviews went from appearing in 15% of searches to 43% within one year, and visits to Google's separate AI Mode more than doubled, from 126 million in June 2025 to 279 million in May 2026 [6]. Pew's own count a year earlier, 18% of searches with an AI summary in March 2025 [5], fits the lower end of that trend, although the two studies measured different samples in different ways, so the numbers should not be combined into one curve.

The same Similarweb report contains a number that matters a great deal for GEO: as of May 2026, only 6.8% of U.S. ChatGPT desktop queries produced an answer that included citations [6]. Most answers name no source at all. When an answer does cite, being among the few sources named is correspondingly valuable, and the report notes that the number of answers with a citation had risen more than fivefold in a year.

2.4.3 The scale of the engines

The third group measures the size of the audience. OpenAI announced in February 2026 that ChatGPT had more than 900 million weekly active users [7]. Alphabet reported in July 2026 that its Gemini app had 950 million monthly active users, up from 750 million two quarters earlier [8]. Perplexity's last specific company-wide figure was 780 million queries in May 2025, then growing by more than 20% a month [9]; no comparable figure had been published by September 2026, so this book quotes it as "last disclosed". Weekly and monthly active users are different measures, and a single person may use several engines, so these numbers cannot be added together. What they show is that each of the large answer engines reaches an audience of the same order as the largest social networks.

2.4.4 Indonesia

For an Indonesian business, the relevant audience is large and still growing. The 2026 survey of the Indonesian Internet Service Providers Association (APJII), released in May 2026, counts about 235.3 million internet users in a population of about 287.3 million: a penetration of 81.72%, up from 80.66% in 2025 [10]. With Equation (2.1), the rise from 80.66 to 81.72 is a relative change of $(81.72 - 80.66)/80.66 \approx 0.013$, about 1.3 percent in a year. The market is no longer growing fast in *people*; the growth now is in what those people do online, and asking an assistant instead of scanning a list is one of the fastest-growing things they do.

2.4.5 Treat forecasts with care

> **A FORECAST IS NOT A MEASUREMENT.** In February 2024 the research firm Gartner predicted that "by 2026, traditional search engine volume will drop 25%", as AI chatbots and virtual agents took over [11]. It is now 2026, and the measurements above tell a more complicated story: answers are appearing far more often *inside* search engines, and clicks on links fall when they do, but "search volume dropped by a quarter" is not what any of the studies in **Table 2.1** measured. Commentary in 2026 has argued that the prediction did not come true as stated. The lesson for a reader, and for a platform that sells to businesses, is the one Profitina applies to itself: quote measurements with their dates and methods, label forecasts as forecasts, and never build a promise on someone else's prediction.

2.5 What GEO Is: The Academic Origin

2.5.1 The paper that named it

The term *Generative Engine Optimization* was introduced by a research team from Princeton University, Georgia Tech, the Allen Institute for AI and IIT Delhi: Pranjal Aggarwal, Vishvak Murahari, Tanmay Rajpurohit, Ashwin Kalyan, Karthik Narasimhan and Ameet Deshpande. Their paper, "GEO: Generative Engine Optimization", first appeared as a preprint on 16 November 2023 and was published at KDD 2024, the ACM conference on knowledge discovery and data mining [2].

The paper asked a question that website owners had not been able to ask of search engines in the same way: if a generative engine writes one answer from several sources, what makes one source more *visible* in that answer than another, and can the owner of a source improve it?

2.5.2 Measuring visibility inside an answer

To answer, the team first had to define what "visible" means when there is no list. Their main measure looks at the sentences of the generated answer that cite a given source, counts their words, and gives more weight to sentences that appear earlier, because readers pay more attention to the beginning of an answer. Written out, for a source c_i and a generated response r made of the sentences S:

$$\mathrm{Imp}_{\mathrm{pwc}}(c_i, r) = \frac{\sum_{s \in S_{c_i}} |s| \cdot e^{-\mathrm{pos}(s)/|S|}}{\sum_{s \in S} |s|} \tag{2.2}$$

Read Equation (2.2) from the inside out. S_{c_i} is the set of sentences of the response that cite source c_i. $|s|$ is the number of words in sentence s, so the numerator adds up the words spent on the source. $\mathrm{pos}(s)$ is the position of sentence s in the response (first, second, …), and $|S|$ is the number of sentences, so $\mathrm{pos}(s)/|S|$ runs from near 0 at the start to near 1 at the end. The factor $e^{-\mathrm{pos}(s)/|S|}$ is therefore close to 1 for an early sentence and falls to about $e^{-1} \approx 0.37$ for the last one: an early mention counts almost fully, a late one about a third. The denominator is the total number of words in the response, so the result is the *share* of the answer that belongs to the source, adjusted for position. The paper calls this the *position-adjusted word count* and complements it with a *subjective impression* score judged by a language model along seven dimensions, such as relevance, influence and the likelihood that a reader would click [2].

A small example makes Equation (2.2) concrete. Suppose an answer has $|S| = 4$ sentences of 10 words each, 40 words in total, and the laundry's page is cited by the first sentence only, whose

position we number pos = 0 (the paper's exact convention for numbering positions does not change the idea). Then the numerator is $10 \cdot e^{0} = 10$ and the measure is $10/40 = 0.25$. If instead the laundry is cited only by the last sentence, pos = 3, the numerator is $10 \cdot e^{-3/4} \approx 10 \cdot 0.47 = 4.7$ and the measure is about 0.12: the same ten words, worth half as much because they come last.

2.5.3 A benchmark and nine methods

The team built **GEO-bench**, a benchmark of 10,000 queries drawn from nine existing datasets and spanning 25 domains, with training, validation and test splits of 8,000, 1,000 and 1,000 queries [2]. For each query they took the sources a search step returned, rewrote *one* source with an optimization method, regenerated the answer, and measured whether that source's visibility went up or down compared with the unmodified version.

Nine methods were tested: a more *Authoritative* (persuasive) tone, *Statistics Addition*, *Keyword Stuffing*, *Cite Sources*, *Quotation Addition*, *Easy-to-Understand* language, *Fluency Optimization*, *Unique Words* and *Technical Terms*. **Table 2.2** summarizes the headline results on the position-adjusted word count.

Table 2.2 Selected results of the GEO paper on the position-adjusted word count: the best three methods and keyword stuffing against the unmodified baseline [2].

Method	What it changes in the source	Result on Imp_{pwc}
No optimization (baseline)	nothing	19.3
Quotation Addition	adds relevant quotations from credible sources	27.2 (about +41%)
Statistics Addition	replaces vague claims with numbers	25.2 (about +32%)
Cite Sources	adds citations to credible sources	24.6 (about +28%)
Keyword Stuffing	repeats query words, the old SEO trick	17.7 (worse than baseline)

The relative gains in the last column come from Equation (2.1): for Quotation Addition, $(27.2 - 19.3)/19.3 = 7.9/19.3 \approx 0.41$. The paper summarizes its findings as visibility improvements "up to 40%" in generative engine responses, and reports gains of up to 37% in a test on Perplexity, a deployed commercial engine [2].

Two lessons from **Table 2.2** have shaped practical GEO ever since. The methods that worked add **verifiable substance**: quotations, numbers, references. The method that failed is the one that tries to game the machine: repeating keywords, which once helped pages rank, made them *less* visible to a model that is trying to write a useful answer. For an honest business this is good news. The best thing a laundry can do for its visibility in an answer is to state its real prices, hours and policies precisely, with numbers.

> **READ THE PAPER'S RESULTS AS THE PAPER STATES THEM.** GEO-bench measured changes to one source among several, on the engines and models of 2023–2024, with the paper's own metrics. The percentages are evidence that substance helps and stuffing hurts, not a guarantee of a particular gain on any engine today. Engines change their retrieval and their models constantly.

2.6 Profitina's Definition of GEO

2.6.1 In one sentence, and in a comparison

Profitina explains the discipline to its users in two sentences that are worth learning by heart [4]:

> **GEO, IN PROFITINA'S WORDS.** "SEO optimizes your ranking on search results (a list of links). GEO optimizes so your content is understood, extracted, and cited directly inside AI answers. They complement each other."

The three verbs in the middle are chosen with care. Content must be **understood**: the engine must know what the page is about and which business it describes, which is what structured data is for. It must be **extracted**: the facts must be in text a machine can lift out, not in images or scripts. And it must be **cited**: of all the sources that say similar things, this one should be the one the engine names. **Table 2.3** sets the two disciplines side by side.

Table 2.3 SEO and GEO compared. The foundations are shared; what differs is who chooses the source and what counts as success.

	SEO	GEO
The engine returns	a ranked list of links	one generated answer, sometimes with citations
Who chooses the source	the person, by clicking	the engine, while writing
Unit of success	a position in the list	a citation or a correct mention inside the answer
Success is visible as	a visit to the site	the business's facts repeated correctly, whether or not anyone visits
What helps most	crawlability, relevant titles and headings, links from other sites	the same foundations, plus extractable answer-first text, precise facts and numbers, structured data, and crawler access for AI bots
What hurts	slow, uncrawlable, thin pages	the same, plus facts locked in images, vague marketing language, keyword stuffing

2.6.2 What Profitina measures, and what it does not

Profitina's AI Visibility Engine estimates a site's readiness to be cited from signals it can observe directly on the public web: how extractable the content is, what schema.org structured data the site publishes, and how sound its technical foundations are [12]. It combines several signals into a readiness score from 0 to 100, shows its method openly, and lists prioritized fixes. It also says plainly what the score is: "Visibility results are methodological estimates, not guarantees of actual third-party AI engine output" [12].

Just as important is what the platform does *not* do. It does not measure a business's visibility by asking ChatGPT, Gemini or Perplexity about it. Chapter 1 gave the reasons: independence (a measurement that depends on a third party's service breaks when that service changes its terms), privacy (brand and competitor names would be sent to an outside provider), and a house rule against paid third-party APIs. The answer engines are the *targets* whose readiness is estimated, never components the platform depends on. The research of Section 2.5 explains why this is a reasonable choice: the factors that make a source visible, extractable substance, clear structure, precise facts, are properties of the *source*, and they can be observed at the source.

2.6.3 The loop: Scan, Optimize, Monitor

Profitina organizes its work as a loop with the same three steps in both of its languages: **Scan · Optimize · Monitor** in English, **Pindai · Optimasi · Pantau** in Indonesian, abbreviated SOM and POP. The *scan* estimates readiness and finds gaps; *optimize* turns the gaps into fixes, such as answer-first text, an FAQ section, structured data and crawler access; *monitor* re-scans on a schedule and shows the trend. Chapter 3 walks through the loop screen by screen. This chapter's lab builds the smallest possible piece of the *optimize* step: the structured data and crawler files.

2.6.4 Honest expectations

A GEO platform is tempted to promise too much: "become the answer ChatGPT gives", "rank number one in AI". Profitina's documents refuse that language for a simple reason: no one outside an engine controls what it will say, no single leaderboard of AI citations exists, and engines change their retrieval and models continuously. What a platform *can* do is raise the odds, legitimately and measurably, and show its work. The same stance runs through this book. When a claim is an estimate, it is called one.

2.7 The Machine-Readable Layer: Structured Data, robots.txt and llms.txt

A web page is written for people, but it is read by programs first. Three small pieces of text, none of them visible on the page itself, tell those programs what the page is about and what they may do with it. Together they are the foundation of any GEO work, and they are exactly what lab G6 generates.

2.7.1 Structured data: schema.org in JSON-LD

schema.org is a shared vocabulary for describing things on the web: businesses, products, events, articles, people, questions and answers. It was founded by the major search engines, and it is maintained in the open; its current release at the time of writing is version 30.1, of 16 September 2026 [13]. Each kind of thing is a *type*, such as `LocalBusiness`, and types form a hierarchy. `DryCleaningOrLaundry`, for instance, is a more specific kind of `LocalBusiness` [14]. Each type has *properties*, such as `name`, `address`, `telephone` or `openingHours`.

JSON-LD (JSON for Linked Data) is the most common way to write schema.org data into a page [15]. It is ordinary JSON, placed in the page's HTML inside a `<script type="application/ld+json">` element, which browsers do not display but crawlers read. Three special keys, all beginning with `@`, turn plain JSON into linked data:

- `@context` says which vocabulary the names come from, here `https://schema.org`;
- `@type` says what kind of thing an object is, such as `DryCleaningOrLaundry` or `FAQPage`;
- `@id` gives an object a global name, a URL, so that other objects can refer to it; `"about": {"@id": "https://laundry.example/#business"}` says "this FAQ is about the business named by that identifier".

A fourth key, `@graph`, holds a list of several objects in one block, so that a page can describe the business and its FAQ together and link them with `@id`.

> **DEFINITION.** *Structured data* is information about a page written in a machine-readable vocabulary. In *JSON-LD* it is a JSON object whose `@context` names the vocabulary, whose `@type` names the kind of thing described, and whose other keys are that type's properties.

Why does this help an answer engine? Because it removes guessing. A page may say "Open daily except Sunday afternoons, call us!" in a decorative font; the JSON-LD says `"openingHours": ["Mo-Sa 07:00-21:00", "Su 08:00-17:00"]` and `"telephone": "+62-812-0000-1234"`. An FAQ written as `FAQPage` with `Question` and `Answer` objects hands the engine exactly the kind of short, self-contained, quotable statement that Section 2.5 found to be most visible. Structured data does not guarantee that anything will be quoted; it makes the facts unambiguous for any program that wants them.

2.7.2 Keep the facts true

Structured data is a promise about the page, and it must match the page. A `priceRange` in the JSON-LD that disagrees with the price list on the page confuses every reader, human and machine alike, and search engines' own guidelines treat structured data that misrepresents the visible content as a problem. The rule for lab G6 is therefore that every fact comes from one file the owner maintains, `business.json`, and the same file feeds both the page text and the structured data.

2.7.3 robots.txt: who may read what

`robots.txt` is a plain text file at the root of a site, `https://laundry.example/robots.txt`, that tells crawlers which paths they may fetch. It is one of the oldest conventions of the web, from 1994, and it was finally written down as an Internet standard, the *Robots Exclusion Protocol*, RFC 9309, in September 2022 [16]. The file is a list of **groups**. Each group begins with one or more `User-agent:` lines naming crawlers, followed by `Allow:` and `Disallow:` rules, each with a path prefix. A special group `User-agent: *` applies to every crawler that no other group names. A `Sitemap:` line, anywhere in the file, points to the site's list of pages.

RFC 9309 fixes precisely how a crawler decides, and **Figure 2.3** draws the procedure. First the crawler finds the groups whose `User-agent` names it; if there are none, it uses the `*` group. A crawler that has its own group *ignores* the `*` group entirely, which surprises many site owners. Then, among the rules of those groups whose path is a prefix of the path it wants, the **longest** one wins; if an `Allow` and a `Disallow` rule are equally long, `Allow` wins; and if no rule matches, the path is allowed.

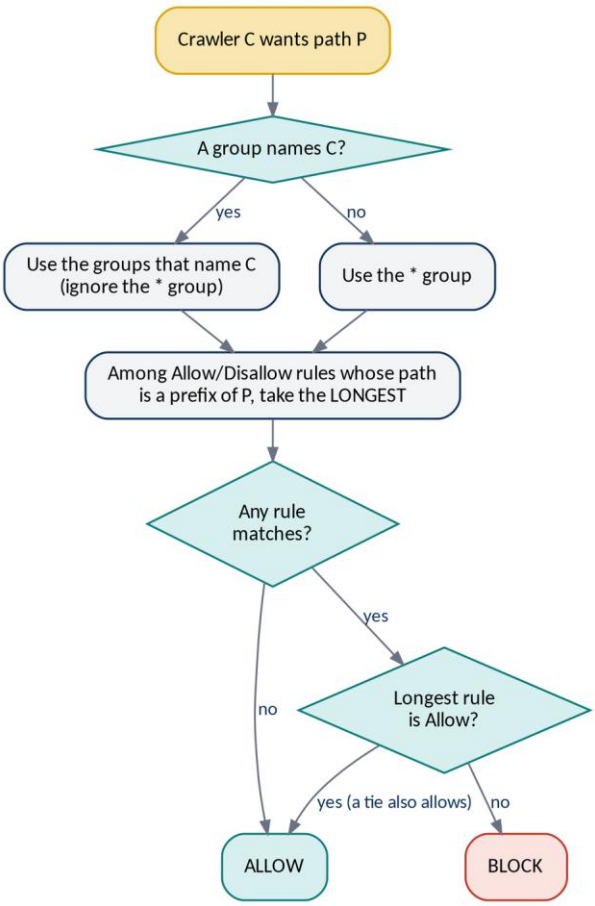


Figure 2.3 How a crawler applies `robots.txt` under RFC 9309: choose the groups that name it (or the `*` group), then let the longest matching rule decide, with ties going to `Allow`.

Two warnings belong here. First, `robots.txt` is a *request*, not a lock. Well-behaved crawlers obey it; anything private must be protected by passwords and permissions, never only by a `Disallow` line, which in fact *advertises* the path to anyone who reads the file. Second, the file matches *prefixes of paths*, character by character, not folder names: `Disallow: /admin` blocks `/admin/`, `/admin.html` and `/administrator`, and `Allow: /admin/help` also allows `/admin/helpdesk`. Exercise 2.9 explores both traps.

2.7.4 The AI crawlers

Each large AI company runs several crawlers with different jobs, and each publishes their names so that site owners can decide separately. **Table 2.4** lists the ones lab G6 names, with each company's own description.

Table 2.4 Crawler names used in lab G6's `robots.txt`, with each company's own description of the crawler's job.

Company	Crawler (User-agent)	What it does, according to the company
OpenAI	`OAI-SearchBot`	Finds sites to show in ChatGPT's search answers; sites that block it are not shown there [17]
OpenAI	`GPTBot`	Crawls content that may be used to train OpenAI's foundation models [17]
OpenAI	`ChatGPT-User`	Visits a page when a ChatGPT user's request needs it; not automatic crawling [17]
Anthropic	`ClaudeBot`	Collects web content to train and improve Claude models [18]
Anthropic	`Claude-SearchBot`	Crawls to improve the quality of Claude's search results [18]
Anthropic	`Claude-User`	Fetches a page when a Claude user asks a question that needs it [18]
Perplexity	`PerplexityBot`	Indexes sites to surface them in Perplexity's answers [19]
Google	`Google-Extended`	Not a separate crawler but a control token: whether Google may use content for training Gemini models and for grounding in Gemini apps; it does not affect Google Search [20]
Google, Microsoft	`Googlebot`, `Bingbot`	The classic search crawlers, whose indexes also feed AI features such as AI Overviews and Copilot

The table shows why a single "allow AI" or "block AI" switch is too coarse. A laundry that wants to appear in ChatGPT's search answers must allow `OAI-SearchBot`; whether it also allows `GPTBot`, which collects training data, is a separate decision. Many publishers block training crawlers and allow search crawlers; a small business that wants to be widely known usually allows both. Profitina's own advice puts crawler access *first* among the technical fixes, because an engine cannot quote a page it was not allowed to read [12]. Microsoft's Bing deserves a special mention in that advice: its index grounds Copilot, and registering a site and its sitemap in Bing Webmaster Tools is one of the cheapest high-value steps a small business can take.

2.7.5 llms.txt: a short map for language models

`llms.txt` is the newest of the three and the least settled. It was proposed on 3 September 2024 by Jeremy Howard of Answer.AI as a file at a site's root, `/llms.txt`, written in Markdown, that gives language models a concise, curated overview of the site: an H1 title with the site's name, a one-line summary in a block quote, a few paragraphs of essential detail, and then sections headed with `##` containing lists of links, each written `- [title](url): note` [21]. A section named `## Optional` marks links a model may skip when it is short of space.

Is anyone reading it? The honest answer in September 2026 is: not much yet. Adoption has grown quickly. A scan on 18 September 2026 of the Tranco list of top domains found an `llms.txt` or `llms-full.txt` on 9.3% of the top 1,000 domains and 8.3% of the top 10,000 [22], and one tracker counted 8.8 times more sites with the file in May 2026 than a year earlier [23]. But use lags far behind. An analysis of server logs from 137,000 domains found that 97% of `llms.txt` files received no requests at all in May 2026, and that most of the requests that did arrive came from SEO audit tools rather than AI assistants [23]. The team that ran the scan adds that publishing the file "does not establish search-ranking improvements, AI citations, or training permissions" [22].

> **KEY INSIGHT.** `llms.txt` is cheap hygiene, not a lever. Profitina publishes one on its own marketing site because it costs little and describes the site honestly, and lab G6 generates one for the same reason. Neither claims that it causes citations. The file that *does* decide whether an engine may read a page at all is `robots.txt`.

2.8 Lab G6: A JSON-LD, robots.txt and llms.txt Generator

2.8.1 What we will build

Lab G6 takes the facts a business owner knows and produces the three machine-readable pieces of Section 2.7, then checks them by reading them back. **Figure 2.4** shows the whole pipeline.

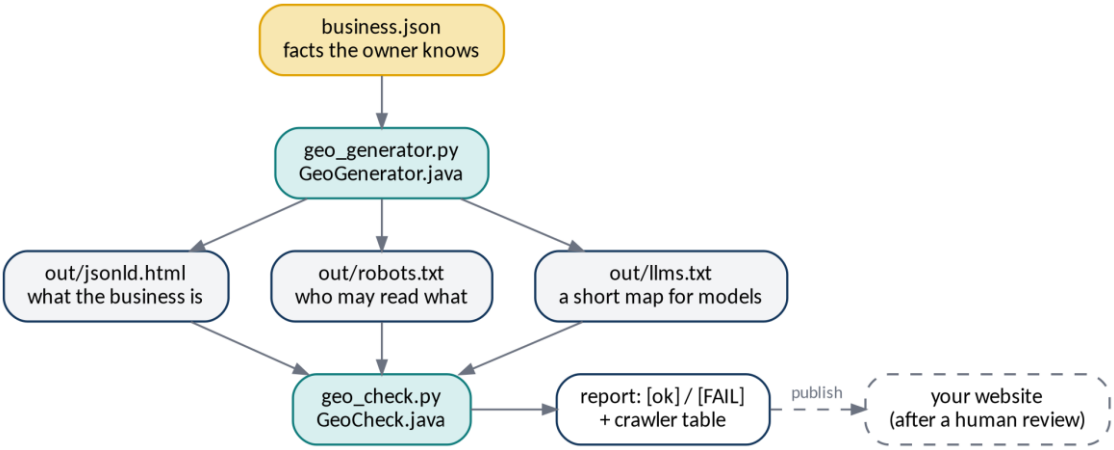


Figure 2.4 Lab G6. One facts file feeds a generator that writes three files; a separate checker reads them back and reports; a person reviews the result before anything is published.

The design follows three rules, each of which the chapter has already motivated.

1. **One source of truth.** Every fact lives in `business.json`. Nothing is typed twice, so the structured data, the crawler file and the model summary cannot disagree.
2. **Generate, then verify independently.** The checker does not trust the generator. It reads the written files from disk, parses them with a real parser, and applies the rules of RFC 9309 and of the `llms.txt` proposal. A generator bug therefore shows up as a failed check, not as a silently broken website.
3. **A person publishes.** The programs write files into a folder called `out`; they never upload anything. A human reviews them and copies them to the site. The `robots.txt` even says so in its second line.

Both versions, Python and Java, use only their language's standard library, read the same `business.json`, and write byte-for-byte identical files.

2.8.2 The facts file

Listing 2.1 is Profitina Laundry's facts file. The address, telephone and web address are those of the fictional business used throughout the Profitina Course Library; `laundry.example` uses the `.example` top-level domain, which is reserved for documentation and can never belong to a real site.

Listing 2.1 `business.json`: the facts that Profitina Laundry's owner maintains, the single source for all three generated files.

```
{
  "name": "Profitina Laundry",
  "type": "DryCleaningOrLaundry",
  "description": "Wash-and-fold, ironing and dry cleaning in Surabaya.",
  "url": "https://laundry.example",
  "telephone": "+62-812-0000-1234",
  "email": "hello@laundry.example",
  "priceRange": "Rp7,000 - Rp75,000",
  "address": {
    "street": "Jl. Penjaringan Sari No. 65, Rungkut",
    "city": "Surabaya",
    "region": "Jawa Timur",
    "postalCode": "60297",
    "country": "ID"
  },
  "openingHours": ["Mo-Sa 07:00-21:00", "Su 08:00-17:00"],
  "faq": [
    {"q": "How much does wash-and-fold cost?",
     "a": "Rp7,000 per kilogram, with a minimum of 3 kg per order."},
    {"q": "How long does a normal order take?",
     "a": "Two days. Express orders are ready the same day for 50% more."},
    {"q": "Can I pay with QRIS?",
     "a": "Yes. Every outlet accepts cash and QRIS."},
    {"q": "How do I check the status of my order?",
     "a": "Enter your bill number and phone number on the status page."}
  ],
  "pages": [
    {"title": "Services and prices", "path": "/services",
     "note": "prices of every service"},
    {"title": "Outlets", "path": "/outlets",
     "note": "addresses and opening hours"},
    {"title": "Order status", "path": "/status",
     "note": "find an order by bill number"}
  ],
  "allowCrawlers": ["Googlebot", "Bingbot", "GPTBot", "OAI-SearchBot",
                    "ChatGPT-User", "ClaudeBot", "Claude-SearchBot",
                    "Claude-User", "PerplexityBot", "Google-Extended"],
  "blockCrawlers": ["CCBot"],
  "privatePaths": ["/admin/", "/api/"]
}
```

The fields fall into four groups. The first ten describe the business itself. `faq` holds question-and-answer pairs, written the way Section 2.5 recommends: short, self-contained, with numbers. `pages` lists the site's important pages for `llms.txt`. The last three fields are *policy*: which crawlers the owner allows, which one they block (here `CCBot`, the Common Crawl crawler, as an example of an owner's choice, not a recommendation), and which paths are private.

2.8.3 The generator in Python

Listing 2.2 reads the facts, checks them, and writes the three files.

Listing 2.2 `geo_generator.py`: reads `business.json`, validates it, and writes `jsonld.html`, `robots.txt` and `llms.txt` into a folder.

```python
# geo_generator.py - business facts in, JSON-LD + robots.txt + llms.txt out
import json
import os
import sys

REQUIRED = ["name", "type", "description", "url", "telephone", "address", "faq"]


def load_facts(filename):
    with open(filename, encoding="utf-8") as f:
        facts = json.load(f)
    for field in REQUIRED:
        if not facts.get(field):
            raise ValueError(f"{filename}: field '{field}' is missing or empty")
    if not facts["url"].startswith("https://"):
        raise ValueError(f"{filename}: url must start with https://")
    facts["url"] = facts["url"].rstrip("/")      # one spelling, no final slash
    return facts


def build_jsonld(facts):
    home = facts["url"] + "/"
    a = facts["address"]
    business = {
        "@type": facts["type"],
        "@id": home + "#business",
        "name": facts["name"],
        "description": facts["description"],
        "url": home,
        "telephone": facts["telephone"],
        "email": facts.get("email", ""),
        "priceRange": facts.get("priceRange", ""),
        "address": {
            "@type": "PostalAddress",
            "streetAddress": a["street"],
            "addressLocality": a["city"],
            "addressRegion": a["region"],
            "postalCode": a["postalCode"],
            "addressCountry": a["country"],
        },
        "openingHours": facts.get("openingHours", []),
    }
    questions = []
    for pair in facts["faq"]:
        questions.append({
            "@type": "Question",
```

Listing 2.2 (Continued) *geo_generator.py: reads business.json, validates it, and writes jsonld.html, robots.txt and llms.txt into a folder.*

```
            "name": pair["q"],
            "acceptedAnswer": {"@type": "Answer", "text": pair["a"]},
        })
    faq_page = {
        "@type": "FAQPage",
        "@id": home + "#faq",
        "about": {"@id": home + "#business"},
        "mainEntity": questions,
    }
    return {"@context": "https://schema.org", "@graph": [business, faq_page]}


def jsonld_script(doc):
    text = json.dumps(doc, indent=2, ensure_ascii=False)
    text = text.replace("</", "<\\/")      # "</script>" cannot end it early
    return ('<script type="application/ld+json">\n' + text + "\n</script>\n")


def build_robots(facts):
    private = facts.get("privatePaths", [])
    lines = [f"# robots.txt for {facts['name']}",
             "# Generated from business.json; review before publishing.", ""]
    allowed = facts.get("allowCrawlers", [])
    if allowed:
        lines += [f"User-agent: {bot}" for bot in allowed]
        lines += [f"Disallow: {p}" for p in private] + ["Allow: /", ""]
    for bot in facts.get("blockCrawlers", []):
        lines += [f"User-agent: {bot}", "Disallow: /", ""]
    lines += ["User-agent: *"]
    lines += [f"Disallow: {p}" for p in private] + ["Allow: /", ""]
    lines += [f"Sitemap: {facts['url']}/sitemap.xml"]
    return "\n".join(lines) + "\n"


def build_llms(facts):
    a = facts["address"]
    lines = [f"# {facts['name']}", "", f"> {facts['description']}", "",
             f"Address: {a['street']}, {a['city']}, {a['region']} "
             f"{a['postalCode']}, {a['country']}.",
             f"Telephone: {facts['telephone']}. "
             f"Opening hours: {'; '.join(facts.get('openingHours', []))}.",
             "", "## Pages", ""]
    for page in facts.get("pages", []):
        lines.append(f"- [{page['title']}]({facts['url']}{page['path']}):"
                     f" {page['note']}")
    lines += ["", "## Optional", "",
```

Listing 2.2 (Continued) *geo_generator.py: reads business.json, validates it, and writes jsonld.html, robots.txt and llms.txt into a folder.*

```
            f"- [FAQ]({facts['url']}/#faq):"
            f" {len(facts['faq'])} questions and answers"]
    return "\n".join(lines) + "\n"


def write(folder, name, text):
    path = os.path.join(folder, name)
    with open(path, "w", encoding="utf-8", newline="\n") as f:
        f.write(text)
    lines = text.count("\n")
    size = len(text.encode("utf-8"))
    print(f"Wrote {path:<18} {lines:>3} lines {size:>5} bytes")


def main():
    source = sys.argv[1] if len(sys.argv) > 1 else "business.json"
    folder = sys.argv[2] if len(sys.argv) > 2 else "out"
    try:
        facts = load_facts(source)
    except (OSError, ValueError) as e:   # JSONDecodeError is a ValueError
        print("Cannot generate:", e)
        sys.exit(1)
    print(f"Read {source}: {facts['name']}, {len(facts['faq'])} FAQ pairs,"
          f" {len(facts.get('pages', []))} pages")
    os.makedirs(folder, exist_ok=True)
    write(folder, "jsonld.html", jsonld_script(build_jsonld(facts)))
    write(folder, "robots.txt", build_robots(facts))
    write(folder, "llms.txt", build_llms(facts))


if __name__ == "__main__":
    main()
```

Walk through it function by function.

- `load_facts` reads the JSON file with `json.load` and refuses to continue if a required field is missing or empty, or if the web address does not begin with `https://`. A generator that silently produced structured data without a URL would be worse than no generator. It also normalizes the address by removing a trailing slash, so that later code can add `/` exactly once.
- `build_jsonld` builds two Python dictionaries, the business and the FAQ page, and wraps them in `@context` and `@graph`. Python dictionaries keep the order in which keys are inserted, so the JSON comes out in a readable, predictable order. The FAQ's `about` property points to the business's `@id`, linking the two objects as Section 2.7.1 described.
- `jsonld_script` turns the dictionary into text with `json.dumps(doc, indent=2, ensure_ascii=False)`: two-space indentation for readability, and non-ASCII characters kept as they are rather than escaped. Then it replaces every `</` with `<\/`. In JSON, `\/` is simply another way of writing `/`, so the data does not change; but in HTML, the text `</script>` inside a string would end the script element early and break the page. This one line is a small but real security habit.
- `build_robots` writes one group naming every allowed crawler, with the private paths disallowed and everything else allowed; then one group per blocked crawler with `Disallow: /`; then the `*` group with the same rules as the allowed crawlers; and finally the `Sitemap:` line. Note the order inside each group: the specific `Disallow:` lines come *before* `Allow: /`. Under RFC 9309 order does not matter, but some older readers apply the first matching rule, and this order gives the same answer under both readings (Section 2.8.7).

- `build_llms` follows the structure of the proposal: H1 title, block-quote summary, detail lines, a `## Pages` section of links, and an `## Optional` section.
- `write` saves each file as UTF-8 with `newline="\n"`, so that the files are identical on Windows, where Python would otherwise write `\r\n` line endings, and prints the number of lines and bytes.

Run it in the `Chapter02/python` folder with `python3 geo_generator.py`:

```
Read business.json: Profitina Laundry, 4 FAQ pairs, 3 pages
Wrote out/jsonld.html     70 lines  2043 bytes
Wrote out/robots.txt      26 lines   513 bytes
Wrote out/llms.txt        16 lines   560 bytes
```

On Windows the paths are printed as `out\jsonld.html`, because `os.path.join` uses the system's separator.

2.8.4 The three generated files

The JSON-LD block, ready to paste into the `<head>` of the home page:

```
<script type="application/ld+json">
{
  "@context": "https://schema.org",
  "@graph": [
    {
      "@type": "DryCleaningOrLaundry",
      "@id": "https://laundry.example/#business",
      "name": "Profitina Laundry",
      "description": "Wash-and-fold, ironing and dry cleaning in Surabaya.",
      "url": "https://laundry.example/",
      "telephone": "+62-812-0000-1234",
      "email": "hello@laundry.example",
      "priceRange": "Rp7,000 - Rp75,000",
      "address": {
        "@type": "PostalAddress",
        "streetAddress": "Jl. Penjaringan Sari No. 65, Rungkut",
        "addressLocality": "Surabaya",
        "addressRegion": "Jawa Timur",
        "postalCode": "60297",
        "addressCountry": "ID"
      },
      "openingHours": [
        "Mo-Sa 07:00-21:00",
        "Su 08:00-17:00"
      ]
    },
    {
      "@type": "FAQPage",
      "@id": "https://laundry.example/#faq",
      "about": {
        "@id": "https://laundry.example/#business"
      },
      "mainEntity": [
        {
          "@type": "Question",
          "name": "How much does wash-and-fold cost?",
          "acceptedAnswer": {
            "@type": "Answer",
            "text": "Rp7,000 per kilogram, with a minimum of 3 kg per order."
          }
        },
        {
          "@type": "Question",
          "name": "How long does a normal order take?",
          "acceptedAnswer": {
            "@type": "Answer",
            "text": "Two days. Express orders are ready the same day for 50% more."
          }
        },
        {
          "@type": "Question",
          "name": "Can I pay with QRIS?",
```

```
          "acceptedAnswer": {
            "@type": "Answer",
            "text": "Yes. Every outlet accepts cash and QRIS."
          }
        },
        {
          "@type": "Question",
          "name": "How do I check the status of my order?",
          "acceptedAnswer": {
            "@type": "Answer",
            "text": "Enter your bill number and phone number on the status page."
          }
        }
      ]
    }
  ]
}
</script>
```

Seventy lines is more than a person would type by hand, and every line is mechanical, which is exactly why a generator is worth writing. Notice the two `@id` values and the `about` link between them, the `PostalAddress` object nested inside the business, and that the FAQ answers are complete sentences with numbers in them.

The `robots.txt`:

```
# robots.txt for Profitina Laundry
# Generated from business.json; review before publishing.

User-agent: Googlebot
User-agent: Bingbot
User-agent: GPTBot
User-agent: OAI-SearchBot
User-agent: ChatGPT-User
User-agent: ClaudeBot
User-agent: Claude-SearchBot
User-agent: Claude-User
User-agent: PerplexityBot
User-agent: Google-Extended
Disallow: /admin/
Disallow: /api/
Allow: /

User-agent: CCBot
Disallow: /

User-agent: *
Disallow: /admin/
Disallow: /api/
Allow: /

Sitemap: https://laundry.example/sitemap.xml
```

And the `llms.txt`:

```
# Profitina Laundry

> Wash-and-fold, ironing and dry cleaning in Surabaya.

Address: Jl. Penjaringan Sari No. 65, Rungkut, Surabaya, Jawa Timur 60297, ID.
Telephone: +62-812-0000-1234. Opening hours: Mo-Sa 07:00-21:00; Su 08:00-17:00.

## Pages

- [Services and prices](https://laundry.example/services): prices of every service
- [Outlets](https://laundry.example/outlets): addresses and opening hours
- [Order status](https://laundry.example/status): find an order by bill number

## Optional

- [FAQ](https://laundry.example/#faq): 4 questions and answers
```

2.8.5 The generator in Java

Java's standard library has no JSON parser or writer, so the Java version begins with a small one. **Listing 2.3** is about 230 lines: a *recursive-descent parser*, which reads a value by looking at its first character and calling the method for that kind of value (`object()`, `array()`, `string()`, and so on), which in turn call `value()` for the values they contain; and a writer that pretty-prints with two-space indents exactly as Python's `json.dumps(..., indent=2)` does.

Listing 2.3 `MiniJson.java`: a minimal JSON parser and pretty-printer for this lab, with error positions reported as line and column.

```java
// MiniJson.java - just enough JSON for this lab: parse text, write it back
import java.util.ArrayList;
import java.util.LinkedHashMap;
import java.util.List;
import java.util.Map;

public class MiniJson {
    /** Thrown for text that is not JSON; line and column count from 1. */
    public static class JsonError extends RuntimeException {
        private static final long serialVersionUID = 1L;
        final int line;
        final int column;

        JsonError(String message, String text, int pos) {
            super(message);
            int nl = text.lastIndexOf('\n', pos - 1);
            int count = 1;
            for (int i = 0; i < pos; i++) {
                if (text.charAt(i) == '\n') {
                    count++;
                }
            }
            line = count;
            column = pos - nl;                  // nl is -1 on the first line
        }
    }

    private final String s;
    private int pos = 0;

    private MiniJson(String text) {
        s = text;
    }

    /** Objects become LinkedHashMap (order kept), arrays ArrayList. */
    public static Object parse(String text) {
        MiniJson p = new MiniJson(text);
        Object value = p.value();
        p.skipSpace();
        if (p.pos != text.length()) {
            throw new JsonError("Extra data", text, p.pos);
        }
        return value;
    }
```

Listing 2.3 (Continued) *MiniJson.java: a minimal JSON parser and pretty-printer for this lab, with error positions reported as line and column.*

```
    private void skipSpace() {
        while (pos < s.length() && " \t\r\n".indexOf(s.charAt(pos)) >= 0) {
            pos++;
        }
    }

    private Object value() {
        skipSpace();
        if (pos >= s.length()) {
            throw new JsonError("Expecting value", s, pos);
        }
        char c = s.charAt(pos);
        if (c == '{') {
            return object();
        } else if (c == '[') {
            return array();
        } else if (c == '"') {
            return string();
        } else if (s.startsWith("true", pos)) {
            pos += 4;
            return Boolean.TRUE;
        } else if (s.startsWith("false", pos)) {
            pos += 5;
            return Boolean.FALSE;
        } else if (s.startsWith("null", pos)) {
            pos += 4;
            return null;
        } else if (c == '-' || Character.isDigit(c)) {
            int start = pos;
            while (pos < s.length()
                    && "+-0123456789.eE".indexOf(s.charAt(pos)) >= 0) {
                pos++;
            }
            return Double.parseDouble(s.substring(start, pos));
        }
        throw new JsonError("Expecting value", s, pos);
    }
```

Listing 2.3 (Continued) *MiniJson.java: a minimal JSON parser and pretty-printer for this lab, with error positions reported as line and column.*

```
    private Map<String, Object> object() {
        Map<String, Object> map = new LinkedHashMap<>();
        pos++;                                      // the '{'
        skipSpace();
        if (pos < s.length() && s.charAt(pos) == '}') {
            pos++;
            return map;
        }
        while (true) {
            skipSpace();
            if (pos >= s.length() || s.charAt(pos) != '"') {
                throw new JsonError("Expecting property name", s, pos);
            }
            String key = string();
            skipSpace();
            if (pos >= s.length() || s.charAt(pos) != ':') {
                throw new JsonError("Expecting ':' delimiter", s, pos);
            }
            pos++;
            map.put(key, value());
            skipSpace();
            if (pos < s.length() && s.charAt(pos) == ',') {
                pos++;
            } else if (pos < s.length() && s.charAt(pos) == '}') {
                pos++;
                return map;
            } else {
                throw new JsonError("Expecting ',' delimiter", s, pos);
            }
        }
    }

    private List<Object> array() {
        List<Object> list = new ArrayList<>();
        pos++;                                      // the '['
        skipSpace();
        if (pos < s.length() && s.charAt(pos) == ']') {
            pos++;
            return list;
        }
        while (true) {
            list.add(value());
            skipSpace();
            if (pos < s.length() && s.charAt(pos) == ',') {
                pos++;
            } else if (pos < s.length() && s.charAt(pos) == ']') {
```

Listing 2.3 (Continued) *MiniJson.java: a minimal JSON parser and pretty-printer for this lab, with error positions reported as line and column.*

```
                pos++;
                return list;
            } else {
                throw new JsonError("Expecting ',' delimiter", s, pos);
            }
        }
    }

    private String string() {
        StringBuilder out = new StringBuilder();
        int start = pos;
        pos++;                                  // the opening quote
        while (pos < s.length()) {
            char c = s.charAt(pos++);
            if (c == '"') {
                return out.toString();
            }
            if (c != '\\') {
                out.append(c);
                continue;
            }
            char e = s.charAt(pos++);
            switch (e) {
                case 'n' -> out.append('\n');
                case 't' -> out.append('\t');
                case 'r' -> out.append('\r');
                case 'b' -> out.append('\b');
                case 'f' -> out.append('\f');
                case 'u' -> {
                    String hex = s.substring(pos, pos + 4);
                    out.append((char) Integer.parseInt(hex, 16));
                    pos += 4;
                }
                default -> out.append(e);       // \" \\ and \/
            }
        }
        throw new JsonError("Unterminated string", s, start);
    }

    /** Pretty-print with two-space indents, like Python's json.dumps(indent=2). */
    public static String write(Object value) {
        StringBuilder out = new StringBuilder();
        write(value, 0, out);
        return out.toString();
    }
```

Listing 2.3 (Continued) *MiniJson.java: a minimal JSON parser and pretty-printer for this lab, with error positions reported as line and column.*

```
    private static void write(Object v, int depth, StringBuilder out) {
        String pad = "  ".repeat(depth + 1);
        String end = "  ".repeat(depth);
        if (v instanceof Map<?, ?> map) {
            if (map.isEmpty()) {
                out.append("{}");
                return;
            }
            out.append("{\n");
            int i = 0;
            for (Map.Entry<?, ?> e : map.entrySet()) {
                out.append(pad).append(quote(e.getKey().toString())).append(": ");
                write(e.getValue(), depth + 1, out);
                out.append(++i < map.size() ? ",\n" : "\n");
            }
            out.append(end).append('}');
        } else if (v instanceof List<?> list) {
            if (list.isEmpty()) {
                out.append("[]");
                return;
            }
            out.append("[\n");
            for (int i = 0; i < list.size(); i++) {
                out.append(pad);
                write(list.get(i), depth + 1, out);
                out.append(i + 1 < list.size() ? ",\n" : "\n");
            }
            out.append(end).append(']');
        } else if (v instanceof String str) {
            out.append(quote(str));
        } else {
            out.append(v);                      // numbers, true, false, null
        }
    }

    static String quote(String str) {
        StringBuilder out = new StringBuilder("\"");
        for (char c : str.toCharArray()) {
            switch (c) {
                case '"' -> out.append("\\\"");
                case '\\' -> out.append("\\\\");
                case '\n' -> out.append("\\n");
                case '\t' -> out.append("\\t");
                case '\r' -> out.append("\\r");
                case '\b' -> out.append("\\b");
                case '\f' -> out.append("\\f");
```

Listing 2.3 (Continued) *MiniJson.java: a minimal JSON parser and pretty-printer for this lab, with error positions reported as line and column.*

```
                default -> {
                    if (c < 0x20) {
                        out.append(String.format("\\u%04x", (int) c));
                    } else {
                        out.append(c);
                    }
                }
            }
        }
        return out.append('"').toString();
    }
}
```

Three design choices in **Listing 2.3** deserve a word. Objects are parsed into `LinkedHashMap`, which, unlike `HashMap`, remembers the order in which keys were inserted; without it the written JSON would come out in an unpredictable order and would not match Python's. When the parser meets something that is not JSON, it throws a `JsonError` that records the *line and column* of the problem, computed the same way Python does, so the checker in Section 2.8.6 can report errors identically in both languages. And numbers are read as `double`, which is enough for this lab, whose data contains no numbers at all; a general-purpose library must do better, for instance to write `3` rather than `3.0`. For real projects a library such as Jackson or Gson is the right choice; here, writing the parser once is a lesson in how every JSON library works inside.

Listing 2.4 is the generator itself, a line-for-line counterpart of **Listing 2.2**.

Listing 2.4 `GeoGenerator.java`: the same generator in Java, using `MiniJson` and ordered maps.

```
// GeoGenerator.java - business facts in, JSON-LD + robots.txt + llms.txt out
import java.io.IOException;
import java.nio.charset.StandardCharsets;
import java.nio.file.Files;
import java.nio.file.Path;
import java.util.ArrayList;
import java.util.LinkedHashMap;
import java.util.List;
import java.util.Map;

public class GeoGenerator {
    static final List<String> REQUIRED = List.of("name", "type", "description",
            "url", "telephone", "address", "faq");

    @SuppressWarnings("unchecked")
    static Map<String, Object> map(Object o) {
        return (Map<String, Object>) o;
    }

    @SuppressWarnings("unchecked")
    static List<Object> list(Object o) {
        return o == null ? List.of() : (List<Object>) o;
    }

    static String str(Map<String, Object> m, String key) {
        Object v = m.get(key);
        return v == null ? "" : v.toString();
    }

    static boolean empty(Object v) {             // Python's "not v"
        return v == null || v.equals("") || (v instanceof List<?> l && l.isEmpty())
                || (v instanceof Map<?, ?> m && m.isEmpty());
    }

    static Map<String, Object> loadFacts(String filename) throws IOException {
        String text = Files.readString(Path.of(filename), StandardCharsets.UTF_8);
        Map<String, Object> facts = map(MiniJson.parse(text));
        for (String field : REQUIRED) {
            if (empty(facts.get(field))) {
                throw new IllegalArgumentException(filename + ": field '" + field
                        + "' is missing or empty");
            }
        }
        String url = str(facts, "url");
        if (!url.startsWith("https://")) {
            throw new IllegalArgumentException(filename
```

Listing 2.4 (Continued) *GeoGenerator.java: the same generator in Java, using MiniJson and ordered maps.*

```
                    + ": url must start with https://");
        }
        while (url.endsWith("/")) {              // one spelling, no final slash
            url = url.substring(0, url.length() - 1);
        }
        facts.put("url", url);
        return facts;
    }

    static Map<String, Object> node(Object... keyValues) {
        Map<String, Object> m = new LinkedHashMap<>();   // keeps insertion order
        for (int i = 0; i < keyValues.length; i += 2) {
            m.put((String) keyValues[i], keyValues[i + 1]);
        }
        return m;
    }

    static Map<String, Object> buildJsonLd(Map<String, Object> facts) {
        String home = str(facts, "url") + "/";
        Map<String, Object> a = map(facts.get("address"));
        Map<String, Object> business = node(
                "@type", str(facts, "type"),
                "@id", home + "#business",
                "name", str(facts, "name"),
                "description", str(facts, "description"),
                "url", home,
                "telephone", str(facts, "telephone"),
                "email", str(facts, "email"),
                "priceRange", str(facts, "priceRange"),
                "address", node(
                        "@type", "PostalAddress",
                        "streetAddress", str(a, "street"),
                        "addressLocality", str(a, "city"),
                        "addressRegion", str(a, "region"),
                        "postalCode", str(a, "postalCode"),
                        "addressCountry", str(a, "country")),
                "openingHours", list(facts.get("openingHours")));
        List<Object> questions = new ArrayList<>();
        for (Object o : list(facts.get("faq"))) {
            Map<String, Object> pair = map(o);
            questions.add(node(
                    "@type", "Question",
                    "name", str(pair, "q"),
                    "acceptedAnswer",
                    node("@type", "Answer", "text", str(pair, "a"))));
        }
```

Listing 2.4 (Continued) *GeoGenerator.java: the same generator in Java, using MiniJson and ordered maps.*

```
        Map<String, Object> faqPage = node(
                "@type", "FAQPage",
                "@id", home + "#faq",
                "about", node("@id", home + "#business"),
                "mainEntity", questions);
        return node("@context", "https://schema.org",
                "@graph", List.of(business, faqPage));
    }

    static String jsonLdScript(Map<String, Object> doc) {
        String text = MiniJson.write(doc);
        text = text.replace("</", "<\\/");   // "</script>" cannot end it early
        return "<script type=\"application/ld+json\">\n" + text + "\n</script>\n";
    }

    static String buildRobots(Map<String, Object> facts) {
        List<Object> priv = list(facts.get("privatePaths"));
        List<String> lines = new ArrayList<>(List.of(
                "# robots.txt for " + str(facts, "name"),
                "# Generated from business.json; review before publishing.", ""));
        List<Object> allowed = list(facts.get("allowCrawlers"));
        if (!allowed.isEmpty()) {
            allowed.forEach(bot -> lines.add("User-agent: " + bot));
            priv.forEach(p -> lines.add("Disallow: " + p));
            lines.addAll(List.of("Allow: /", ""));
        }
        for (Object bot : list(facts.get("blockCrawlers"))) {
            lines.addAll(List.of("User-agent: " + bot, "Disallow: /", ""));
        }
        lines.add("User-agent: *");
        priv.forEach(p -> lines.add("Disallow: " + p));
        lines.addAll(List.of("Allow: /", ""));
        lines.add("Sitemap: " + str(facts, "url") + "/sitemap.xml");
        return String.join("\n", lines) + "\n";
    }
```

Listing 2.4 (Continued) *GeoGenerator.java: the same generator in Java, using MiniJson and ordered maps.*

```
    static String buildLlms(Map<String, Object> facts) {
        Map<String, Object> a = map(facts.get("address"));
        List<String> hours = new ArrayList<>();
        list(facts.get("openingHours")).forEach(h -> hours.add(h.toString()));
        List<String> lines = new ArrayList<>(List.of(
                "# " + str(facts, "name"), "", "> " + str(facts, "description"), "",
                "Address: " + str(a, "street") + ", " + str(a, "city") + ", "
                        + str(a, "region") + " " + str(a, "postalCode") + ", "
                        + str(a, "country") + ".",
                "Telephone: " + str(facts, "telephone") + ". Opening hours: "
                        + String.join("; ", hours) + ".",
                "", "## Pages", ""));
        for (Object o : list(facts.get("pages"))) {
            Map<String, Object> page = map(o);
            lines.add("- [" + str(page, "title") + "](" + str(facts, "url")
                    + str(page, "path") + "): " + str(page, "note"));
        }
        lines.addAll(List.of("", "## Optional", "",
                "- [FAQ](" + str(facts, "url") + "/#faq): "
                        + list(facts.get("faq")).size()
                        + " questions and answers"));
        return String.join("\n", lines) + "\n";
    }

    static void write(String folder, String name, String text) throws IOException {
        Path path = Path.of(folder, name);
        Files.writeString(path, text, StandardCharsets.UTF_8);
        long lines = text.chars().filter(c -> c == '\n').count();
        int size = text.getBytes(StandardCharsets.UTF_8).length;
        System.out.printf("Wrote %-18s %3d lines %5d bytes%n",
                folder + "/" + name, lines, size);
    }

    public static void main(String[] args) throws IOException {
        String source = args.length > 0 ? args[0] : "business.json";
        String folder = args.length > 1 ? args[1] : "out";
        Map<String, Object> facts;
        try {
            facts = loadFacts(source);
        } catch (IOException | RuntimeException e) {  // JsonError is one too
            System.out.println("Cannot generate: " + e.getMessage());
            System.exit(1);
            return;
        }
        System.out.println("Read " + source + ": " + str(facts, "name") + ", "
                + list(facts.get("faq")).size() + " FAQ pairs, "
```

Listing 2.4 (Continued) *GeoGenerator.java: the same generator in Java, using MiniJson and ordered maps.*

```
                + list(facts.get("pages")).size() + " pages");
        Files.createDirectories(Path.of(folder));
        write(folder, "jsonld.html", jsonLdScript(buildJsonLd(facts)));
        write(folder, "robots.txt", buildRobots(facts));
        write(folder, "llms.txt", buildLlms(facts));
    }
}
```

The small helper methods at the top absorb the differences between the languages. `map` and `list` cast the parser's `Object` results to the types we know they have; `str` returns an empty string for a missing key, as Python's `facts.get(key, "")` does; `empty` reproduces Python's rule that `None`, `""`, an empty list and an empty dictionary all count as "missing". The method `node(Object... keyValues)` takes its arguments in pairs, key then value, and builds an ordered map, which keeps

`buildJsonLd` almost as readable as the Python dictionary literals. `Files.writeString` writes UTF-8 with exactly the `\n` characters the program put in, on every operating system.

Compile and run it in the `Chapter02/java` folder:

```
javac *.java
java GeoGenerator
```

```
Read business.json: Profitina Laundry, 4 FAQ pairs, 3 pages
Wrote out/jsonld.html     70 lines  2043 bytes
Wrote out/robots.txt      26 lines   513 bytes
Wrote out/llms.txt        16 lines   560 bytes
```

The three files are byte-for-byte identical to the Python version's; on macOS or Linux, `cmp java/out/robots.txt python/out/robots.txt` prints nothing, which means "no difference".

2.8.6 Reading it back: the checker

A generator that says "done" has proved only that it did not crash. **Listing 2.5** reads the three files from disk and checks them against the rules of Section 2.7, printing `[ok]` or `[FAIL]` for each check.

Listing 2.5 `geo_check.py`: parses the JSON-LD, applies `robots.txt` to a list of crawlers and paths, and checks the structure of `llms.txt`.

```
# geo_check.py - read the generated files back and check them
import json
import os
import re
import sys
import urllib.robotparser

BUSINESS_TYPES = {"LocalBusiness", "DryCleaningOrLaundry", "Store", "Restaurant"}
PATHS = ["/", "/services", "/admin/", "/api/orders"]
LINK = re.compile(r"^- \[[^\]]+\]\(https://[^)\s]+\)(: .+)?$")

results = []                                    # (passed, message) pairs


def check(passed, message):
    results.append((passed, message))
    print(("  [ok]   " if passed else "  [FAIL] ") + message)


def read(folder, name):
    with open(os.path.join(folder, name), encoding="utf-8") as f:
        return f.read()


def check_jsonld(text):
    m = re.search(r'<script type="application/ld\+json">(.*?)</script>',
                  text, re.DOTALL)
    check(m is not None, "jsonld.html contains one JSON-LD script block")
    if m is None:
        return
    try:
        doc = json.loads(m.group(1))          # "<\/" is a legal JSON escape of "/"
    except json.JSONDecodeError as e:
        check(False, f"JSON-LD is not valid JSON (line {e.lineno},"
                     f" column {e.colno})")
        return
    check(True, "JSON-LD parses as JSON")
    check(doc.get("@context") == "https://schema.org",
          "@context is https://schema.org")
    nodes = doc.get("@graph", [])
    ids = {n.get("@id") for n in nodes}
    shops = [n for n in nodes if n.get("@type") in BUSINESS_TYPES]
    check(len(shops) == 1, f"one business node ({len(shops)} found)")
    for shop in shops:
        missing = [k for k in ("name", "url", "address", "telephone")
                   if not shop.get(k)]
```

Listing 2.5 (Continued) *geo_check.py: parses the JSON-LD, applies robots.txt to a list of crawlers and paths, and checks the structure of llms.txt.*

```
        check(not missing, f"{shop['@type']} has name, url, address, telephone"
              + (f" (missing: {', '.join(missing)})" if missing else ""))
    faqs = [n for n in nodes if n.get("@type") == "FAQPage"]
    check(len(faqs) == 1, f"one FAQPage node ({len(faqs)} found)")
    for faq in faqs:
        qs = faq.get("mainEntity", [])
        good = [q for q in qs if q.get("@type") == "Question" and q.get("name")
                and q.get("acceptedAnswer", {}).get("text")]
        check(len(qs) > 0 and len(good) == len(qs),
              f"FAQPage has {len(qs)} questions, {len(good)} with an answer")
        target = faq.get("about", {}).get("@id")
        check(target in ids, "FAQPage.about points to a node in the graph")


def check_robots(text):
    rp = urllib.robotparser.RobotFileParser()
    rp.parse(text.splitlines())
    bots = [line.split(":", 1)[1].strip() for line in text.splitlines()
            if line.lower().startswith("user-agent:")]
    bots = [b for b in bots if b != "*"] + ["SomeNewBot"]
    maps = rp.site_maps() or []
    check(len(maps) == 1, f"robots.txt names one sitemap ({len(maps)} found)")
    print()
    print((f"    {'Crawler':<18}" + "".join(f"{p:<13}" for p in PATHS)).rstrip())
    for bot in bots:
        row = ["allow" if rp.can_fetch(bot, p) else "block" for p in PATHS]
        print((f"    {bot:<18}" + "".join(f"{r:<13}" for r in row)).rstrip())
    print()
    readers = [b for b in bots if rp.can_fetch(b, "/")]
    check(len(readers) >= 1, f"{len(readers)} of {len(bots)} crawlers may read /")
    leaks = [b for b in bots if rp.can_fetch(b, "/admin/")]
    check(not leaks, "no crawler may read /admin/")


def check_llms(text):
    lines = text.splitlines()
    check(len(lines) > 0 and lines[0].startswith("# "),
          "llms.txt starts with an H1 title")
    check(any(line.startswith("> ") for line in lines),
          "llms.txt has a one-line summary (> ...)")
    items = [line for line in lines if line.startswith("- ")]
    bad = [line for line in items if not LINK.match(line)]
    check(len(items) > 0 and not bad,
          f"{len(items)} link lines, {len(bad)} malformed")
```

Listing 2.5 (Continued) *geo_check.py: parses the JSON-LD, applies robots.txt to a list of crawlers and paths, and checks the structure of llms.txt.*

```
def main():
    folder = sys.argv[1] if len(sys.argv) > 1 else "out"
    try:
        files = {n: read(folder, n) for n in ("jsonld.html", "robots.txt",
                                              "llms.txt")}
    except OSError as e:
        print("Cannot check:", e)
        sys.exit(1)
    print(f"Checking {folder}/jsonld.html")
    check_jsonld(files["jsonld.html"])
    print(f"Checking {folder}/robots.txt")
    check_robots(files["robots.txt"])
    print(f"Checking {folder}/llms.txt")
    check_llms(files["llms.txt"])
    passed = sum(1 for ok, _ in results if ok)
    print(f"\n{passed} of {len(results)} checks passed.")
    sys.exit(0 if passed == len(results) else 1)


if __name__ == "__main__":
    main()
```

`check_jsonld` finds the script block with a regular expression. The pattern `(.*?)` is *non-greedy*: it matches as little as possible, so it stops at the first `</script>`. With the flag `re.DOTALL`, `.` also matches line breaks. The block is then parsed with `json.loads`; if that fails, the exception carries `lineno` and `colno`, and the check reports them. Then come the structural checks: the right `@context`, exactly one business node with its four essential properties, exactly one `FAQPage` whose every question has an answer, and an `about` link that points to an `@id` that really exists in the graph.

`check_robots` hands the file to Python's own `urllib.robotparser` [24] and asks it, for every crawler named in the file plus an unknown `SomeNewBot`, whether each of four paths may be fetched. The table it prints is the most useful output of the lab: it shows the owner, in one glance, what every crawler will do. `check_llms` checks the three structural rules of the proposal, using a regular expression for the link lines: `- [`, a title without `]`, `](https://`, an address without spaces or `)`, `)`, and optionally `:` followed by a note.

Run it after the generator with `python3 geo_check.py`:

```
Checking out/jsonld.html
  [ok]   jsonld.html contains one JSON-LD script block
  [ok]   JSON-LD parses as JSON
  [ok]   @context is https://schema.org
  [ok]   one business node (1 found)
  [ok]   DryCleaningOrLaundry has name, url, address, telephone
  [ok]   one FAQPage node (1 found)
  [ok]   FAQPage has 4 questions, 4 with an answer
  [ok]   FAQPage.about points to a node in the graph
Checking out/robots.txt
  [ok]   robots.txt names one sitemap (1 found)

    Crawler           /           /services    /admin/      /api/orders
    Googlebot         allow       allow        block        block
    Bingbot           allow       allow        block        block
    GPTBot            allow       allow        block        block
    OAI-SearchBot     allow       allow        block        block
    ChatGPT-User      allow       allow        block        block
    ClaudeBot         allow       allow        block        block
    Claude-SearchBot  allow       allow        block        block
    Claude-User       allow       allow        block        block
    PerplexityBot     allow       allow        block        block
    Google-Extended   allow       allow        block        block
    CCBot             block       block        block        block
    SomeNewBot        allow       allow        block        block

  [ok]   11 of 12 crawlers may read /
  [ok]   no crawler may read /admin/
Checking out/llms.txt
  [ok]   llms.txt starts with an H1 title
  [ok]   llms.txt has a one-line summary (> ...)
  [ok]   4 link lines, 0 malformed

14 of 14 checks passed.
```

Read the crawler table row by row against **Figure 2.3**. The ten named crawlers find their own group and may read everything except the private paths. `CCBot` finds its own group, which disallows `/`, the prefix of every path, so it is blocked everywhere. `SomeNewBot` is named nowhere, so it falls back to the `*` group, which has the same rules as the allowed crawlers. The program also sets an *exit code*, 0 when every check passed and 1 otherwise, so that another program, or a release gate like the one in Chapter 9, can run it and act on the result without reading the text.

The Java checker needs its own `robots.txt` reader, and writing one is the clearest way to understand RFC 9309. **Listing 2.6** parses the file into groups and implements the decision of **Figure 2.3** in `canFetch`.

Listing 2.6 `Robots.java`: a small `robots.txt` reader implementing RFC 9309's group selection and longest-match rule (without the `*` and `$` wildcards).

```java
// Robots.java - a small robots.txt reader following RFC 9309 (no * or $ patterns)
import java.util.ArrayList;
import java.util.List;

public class Robots {
    record Rule(boolean allow, String path) { }

    static class Group {
        final List<String> agents = new ArrayList<>();
        final List<Rule> rules = new ArrayList<>();
    }

    final List<Group> groups = new ArrayList<>();
    final List<String> sitemaps = new ArrayList<>();

    Robots(String text) {
        Group current = null;
        boolean lastWasAgent = false;
        for (String raw : text.split("\n")) {
            String line = raw.replaceAll("#.*", "").strip();  // drop comments
            int colon = line.indexOf(':');
            if (colon < 0) {
                continue;
            }
            String key = line.substring(0, colon).strip().toLowerCase();
            String value = line.substring(colon + 1).strip();
            switch (key) {
                case "user-agent" -> {
                    if (current == null || !lastWasAgent) {  // a new group begins
                        current = new Group();
                        groups.add(current);
                    }
                    current.agents.add(value);
                    lastWasAgent = true;
                }
                case "allow", "disallow" -> {
                    if (current != null && !value.isEmpty()) {
                        current.rules.add(new Rule(key.equals("allow"), value));
                    }
                    lastWasAgent = false;
                }
                case "sitemap" -> sitemaps.add(value);
                default -> lastWasAgent = false;
            }
        }
    }
```

Listing 2.6 (Continued) *Robots.java: a small robots.txt reader implementing RFC 9309's group selection and longest-match rule (without the * and $ wildcards).*

```
    // The groups that name this crawler; if none does, the "*" groups.
    List<Group> groupsFor(String agent) {
        List<Group> named = new ArrayList<>();
        List<Group> star = new ArrayList<>();
        for (Group g : groups) {
            for (String a : g.agents) {
                if (a.equalsIgnoreCase(agent)) {
                    named.add(g);
                } else if (a.equals("*")) {
                    star.add(g);
                }
            }
        }
        return named.isEmpty() ? star : named;
    }

    // The longest matching rule wins; on a tie, allow wins; no rule means allow.
    boolean canFetch(String agent, String path) {
        Rule best = null;
        for (Group g : groupsFor(agent)) {
            for (Rule r : g.rules) {
                if (!path.startsWith(r.path())) {
                    continue;
                }
                int len = r.path().length();
                if (best == null || len > best.path().length()
                        || (len == best.path().length() && r.allow())) {
                    best = r;
                }
            }
        }
        return best == null || best.allow();
    }
}
```

The constructor reads line by line. It removes comments with the regular expression `#.*`, splits each line at its first colon, and lowercases the key, because RFC 9309 makes field names case-insensitive. The variable `lastWasAgent` implements the grouping rule: consecutive `User-agent` lines belong to one group, and a `User-agent` line that follows a rule starts a new group. `groupsFor` chooses the groups that name the crawler, case-insensitively, or else the `*` groups. `canFetch` keeps the best rule seen so far and replaces it when a longer one matches, or an equally long `Allow`. The one simplification is stated in the first line of the file: the wildcards `*` and `$` inside paths, which RFC 9309 also defines, are not supported, because lab G6's files do not use them.

Listing 2.7 is the checker itself, the Java counterpart of **Listing 2.5**.

Listing 2.7 `GeoCheck.java`: the same checks in Java, using `MiniJson` and `Robots`.

```java
// GeoCheck.java - read the generated files back and check them
import java.io.IOException;
import java.nio.charset.StandardCharsets;
import java.nio.file.Files;
import java.nio.file.Path;
import java.util.ArrayList;
import java.util.HashSet;
import java.util.List;
import java.util.Map;
import java.util.Set;
import java.util.regex.Matcher;
import java.util.regex.Pattern;

public class GeoCheck {
    static final Set<String> BUSINESS_TYPES =
            Set.of("LocalBusiness", "DryCleaningOrLaundry", "Store", "Restaurant");
    static final List<String> PATHS =
            List.of("/", "/services", "/admin/", "/api/orders");
    static final Pattern LINK =
            Pattern.compile("^- \\[[^\\]]+\\]\\(https://[^)\\s]+\\)(: .+)?$");
    static final Pattern SCRIPT = Pattern.compile(
            "<script type=\"application/ld\\+json\">(.*?)</script>",
            Pattern.DOTALL);

    static int passed = 0;
    static int total = 0;

    static void check(boolean ok, String message) {
        total++;
        if (ok) {
            passed++;
        }
        System.out.println((ok ? "  [ok]   " : "  [FAIL] ") + message);
    }

    static String text(Object o) {              // "" for a missing value
        return o instanceof String s ? s : "";
    }

    @SuppressWarnings("unchecked")
    static Map<String, Object> map(Object o) {
        return o instanceof Map<?, ?> ? (Map<String, Object>) o : Map.of();
    }
```

Listing 2.7 (Continued) *GeoCheck.java: the same checks in Java, using MiniJson and Robots.*

```
    @SuppressWarnings("unchecked")
    static List<Object> list(Object o) {
        return o instanceof List<?> ? (List<Object>) o : List.of();
    }

    static void checkJsonLd(String html) {
        Matcher m = SCRIPT.matcher(html);
        boolean found = m.find();
        check(found, "jsonld.html contains one JSON-LD script block");
        if (!found) {
            return;
        }
        Map<String, Object> doc;
        try {
            doc = map(MiniJson.parse(m.group(1)));  // "<\/" is a legal "/"
        } catch (MiniJson.JsonError e) {
            check(false, "JSON-LD is not valid JSON (line " + e.line
                    + ", column " + e.column + ")");
            return;
        }
        check(true, "JSON-LD parses as JSON");
        check("https://schema.org".equals(doc.get("@context")),
                "@context is https://schema.org");
        List<Object> nodes = list(doc.get("@graph"));
        Set<String> ids = new HashSet<>();
        List<Map<String, Object>> shops = new ArrayList<>();
        List<Map<String, Object>> faqs = new ArrayList<>();
        for (Object o : nodes) {
            Map<String, Object> n = map(o);
            ids.add(text(n.get("@id")));
            if (BUSINESS_TYPES.contains(text(n.get("@type")))) {
                shops.add(n);
            } else if ("FAQPage".equals(n.get("@type"))) {
                faqs.add(n);
            }
        }
        check(shops.size() == 1, "one business node (" + shops.size() + " found)");
        for (Map<String, Object> shop : shops) {
            List<String> missing = new ArrayList<>();
            for (String k : List.of("name", "url", "address", "telephone")) {
                Object v = shop.get(k);
                if (v == null || v.equals("") || (v instanceof Map<?, ?> mm
                        && mm.isEmpty())) {
                    missing.add(k);
                }
            }
```

Listing 2.7 (Continued) *GeoCheck.java: the same checks in Java, using MiniJson and Robots.*

```
            check(missing.isEmpty(), shop.get("@type")
                    + " has name, url, address, telephone"
                    + (missing.isEmpty() ? ""
                       : " (missing: " + String.join(", ", missing) + ")"));
        }
        check(faqs.size() == 1, "one FAQPage node (" + faqs.size() + " found)");
        for (Map<String, Object> faq : faqs) {
            List<Object> qs = list(faq.get("mainEntity"));
            int good = 0;
            for (Object o : qs) {
                Map<String, Object> q = map(o);
                String answer = text(map(q.get("acceptedAnswer")).get("text"));
                if ("Question".equals(q.get("@type"))
                        && !text(q.get("name")).isEmpty() && !answer.isEmpty()) {
                    good++;
                }
            }
            check(!qs.isEmpty() && good == qs.size(),
                    "FAQPage has " + qs.size() + " questions, " + good
                    + " with an answer");
            String target = text(map(faq.get("about")).get("@id"));
            check(ids.contains(target),
                    "FAQPage.about points to a node in the graph");
        }
    }

    static String row(String first, List<String> cells) {
        StringBuilder sb = new StringBuilder(String.format("    %-18s", first));
        cells.forEach(c -> sb.append(String.format("%-13s", c)));
        return sb.toString().stripTrailing();
    }

    static void checkRobots(String txt) {
        Robots rp = new Robots(txt);
        List<String> bots = new ArrayList<>();
        for (Robots.Group g : rp.groups) {
            g.agents.stream().filter(a -> !a.equals("*")).forEach(bots::add);
        }
        bots.add("SomeNewBot");
        check(rp.sitemaps.size() == 1,
                "robots.txt names one sitemap (" + rp.sitemaps.size() + " found)");
        System.out.println();
        System.out.println(row("Crawler", PATHS));
        for (String bot : bots) {
            List<String> cells = new ArrayList<>();
            PATHS.forEach(p -> cells.add(rp.canFetch(bot, p) ? "allow" : "block"));
```

Listing 2.7 (Continued) *GeoCheck.java: the same checks in Java, using MiniJson and Robots.*

```
            System.out.println(row(bot, cells));
        }
        System.out.println();
        long readers = bots.stream().filter(b -> rp.canFetch(b, "/")).count();
        check(readers >= 1,
                readers + " of " + bots.size() + " crawlers may read /");
        boolean leak = bots.stream().anyMatch(b -> rp.canFetch(b, "/admin/"));
        check(!leak, "no crawler may read /admin/");
    }

    static void checkLlms(String txt) {
        List<String> lines = txt.lines().toList();
        check(!lines.isEmpty() && lines.get(0).startsWith("# "),
                "llms.txt starts with an H1 title");
        check(lines.stream().anyMatch(l -> l.startsWith("> ")),
                "llms.txt has a one-line summary (> ...)");
        List<String> items =
                lines.stream().filter(l -> l.startsWith("- ")).toList();
        long bad = items.stream().filter(l -> !LINK.matcher(l).matches()).count();
        check(!items.isEmpty() && bad == 0,
                items.size() + " link lines, " + bad + " malformed");
    }

    static String read(String folder, String name) throws IOException {
        return Files.readString(Path.of(folder, name), StandardCharsets.UTF_8);
    }

    public static void main(String[] args) {
        String folder = args.length > 0 ? args[0] : "out";
        String html;
        String robots;
        String llms;
        try {
            html = read(folder, "jsonld.html");
            robots = read(folder, "robots.txt");
            llms = read(folder, "llms.txt");
        } catch (IOException e) {
            System.out.println("Cannot check: " + e);
            System.exit(1);
            return;
        }
        System.out.println("Checking " + folder + "/jsonld.html");
        checkJsonLd(html);
        System.out.println("Checking " + folder + "/robots.txt");
        checkRobots(robots);
        System.out.println("Checking " + folder + "/llms.txt");
```

Listing 2.7 (Continued) *GeoCheck.java: the same checks in Java, using MiniJson and Robots.*

```
        checkLlms(llms);
        System.out.println("\n" + passed + " of " + total + " checks passed.");
        System.exit(passed == total ? 0 : 1);
    }
}
```

Run `java GeoCheck` after `java GeoGenerator`, and it prints exactly the same 34 lines as the Python checker, including the crawler table and the closing line `14 of 14 checks passed.`

2.8.7 When things are wrong

A checker earns its keep on bad input. The code folder contains two deliberately damaged cases.

The first is `broken.json`, a copy of the facts file whose address begins with `http://` instead of `https://`. The generator refuses it, in both languages, and exits with code 1:

```
Cannot generate: broken.json: url must start with https://
```

The second is a folder `damaged` with three hand-written files of the kind that are common on real sites: JSON-LD with a trailing comma after the last property (legal in JavaScript, illegal in JSON), a `robots.txt` that disallows everything to everyone, and an `llms.txt` without its title marker or summary. Running `python3 geo_check.py damaged` or `java GeoCheck damaged` prints:

```
Checking damaged/jsonld.html
  [ok]   jsonld.html contains one JSON-LD script block
  [FAIL] JSON-LD is not valid JSON (line 5, column 67)
Checking damaged/robots.txt
  [FAIL] robots.txt names one sitemap (0 found)

    Crawler           /           /services    /admin/      /api/orders
    SomeNewBot        block       block        block        block

  [FAIL] 0 of 1 crawlers may read /
  [ok]   no crawler may read /admin/
Checking damaged/llms.txt
  [FAIL] llms.txt starts with an H1 title
  [FAIL] llms.txt has a one-line summary (> ...)
  [FAIL] 1 link lines, 1 malformed

2 of 8 checks passed.
```

Line 5, column 67 of the script block is the `}` right after the trailing comma: the parser expected another property name and found the end of the object. (The block's text begins right after the opening tag, on the tag's own line, so its line numbers are the same as the file's.) Every other failure names the rule it broke. A site owner who reads this report knows exactly what to fix, and in what order: the JSON-LD cannot be read at all, and no crawler may read the site.

There is one more thing to learn from the checker, and it is about the tools rather than the files. Python 3.11's `urllib.robotparser`, used here, does not implement RFC 9309's longest-match rule: it applies the *first* rule in the group that matches. A change to make it follow RFC 9309 was proposed for CPython in September 2025 [25], so newer versions may behave differently. For the file our generator writes, the two readings agree, because it puts the narrow `Disallow:` lines before the broad `Allow: /`. For a file that says `Allow: /` first and `Disallow: /admin/` second, they do not: the first-match reading lets every crawler into `/admin/`. Exercise 2.8 measures the difference with a small program in both languages. The lesson is general: when two programs must agree on a standard, test them on the cases where the standard's rules actually bite.

2.9 Profitina in Practice

Everything in lab G6 has a counterpart in the live product, at the level of what it does, not how it scores.

The Content Engine and the Technical Assistant. After a scan, Profitina's *Optimize* tab turns visibility gaps into publish-ready drafts: answer-first text that states the direct answer in its first sentences, entity names, an FAQ section, and schema.org JSON-LD [12]. The Technical Assistant helps with `robots.txt`, in particular allowing the AI crawlers of **Table 2.4**, and with submitting a sitemap to Bing. The drafts are written by the platform's local language model, and a quality gate serves an AI draft only when it beats a built-in template; after the September 2026 revision, AI

drafts won in 5 of 6 benchmark cases. When no model is available, the application says so rather than passing a template off as AI output.

Readiness, not querying. The scan estimates readiness from the site's own signals: extractability, structured data, technical soundness. It never sends the business's name to ChatGPT, Gemini or Perplexity to see what they say, for the reasons of Section 2.6.2. Its results carry the same disclaimer as Section 2.6.4: estimates, not guarantees [12].

Profitina's own marketing site. `profitina.com` practices what the platform recommends. It publishes `/llms.txt`, FAQ structured data and a `dateModified` date on its pages. Chapter 10 examines it as a case study, including a lesson the team learned the hard way: the content delivery network in front of the site once served a "managed" `robots.txt` that silently disallowed `GPTBot` and `ClaudeBot`, overriding the file the team had written. The file in the repository was right; the file the world saw was wrong. The rule that came out of it applies directly to lab G6: *always verify the live URL, bypassing any cache*, rather than the file you think you published.

The loop. Lab G6 is the smallest piece of *Optimize*. Chapter 3's lab G1 builds a small readiness checker for *Scan*, which can be pointed at the files G6 writes, and Chapter 9's lab G4 schedules re-scans for *Monitor*. Chapter 10's lab adds a live-URL auditor for exactly the cache problem above.

2.10 Common Misunderstandings

"GEO replaces SEO." It builds on it. An engine cannot cite a page its crawler could not fetch, and many answer engines retrieve from classic search indexes. The foundations of **Table 2.3** are shared.

"If I add JSON-LD, AI will recommend my business." Structured data removes ambiguity; it does not create preference. It makes a correct citation possible, not certain. The paper of Section 2.5 found that *substance*, such as numbers, quotations and references, increased visibility, and that gaming the text decreased it.

"Keyword stuffing still works, just more subtly." In GEO-bench it made sources *less* visible (**Table 2.2**). A model writing an answer has no use for repetition.

"`llms.txt` is how AI finds my site." In May 2026, 97% of the files in one large sample received no requests at all [23]. It is hygiene. The file that decides access is `robots.txt`.

"`Disallow` keeps a page private." It is a request that polite crawlers honor, and it publicly lists the path. Private pages need authentication.

"Blocking `GPTBot` removes my site from ChatGPT." `GPTBot` collects training data; ChatGPT's *search* answers rely on `OAI-SearchBot` [17]. The two are controlled separately, which is the whole point of **Table 2.4**.

"`Google-Extended` controls whether I appear in Google's AI Overviews." According to Google, it controls use for training Gemini models and grounding in Gemini apps, and "does not impact a site's inclusion in Google Search" [20].

"A crawler with its own group still obeys the `*` rules too." Under RFC 9309 it uses only the groups that name it. If a crawler's own group forgets `Disallow: /admin/`, the `*` group's rule does not protect `/admin/` from that crawler.

"A 47% fall and a 7-point fall are the same thing." They describe the same data, 15% to 8%, in different units: relative change by Equation (2.1) versus absolute change in percentage points. Say which one you mean.

"Profitina guarantees a position in AI answers." No one can. The platform says so in its terms and in every result [12].

2.11 Chapter Summary

- A **search engine** crawls, indexes and ranks, and returns a **list of links** from which the *person* chooses. **SEO** earns positions in that list.
- An **answer engine** retrieves sources and has a language model **generate one answer**, sometimes with citations; the *engine* chooses. Success becomes a **citation**, and the reader may never visit the site.
- The **numbers** of **Table 2.1** and **Figure 2.2**: with a Google AI summary, clicks on traditional results fell from 15% to 8% of visits, a relative change of about −47% by Equation (2.1) (Pew, March 2025); AI Overviews rose from 15% to 43% of searches in a year, and AI Mode visits from 126 M to 279 M (Similarweb, 2025–2026); ChatGPT passed 900 M weekly users, Gemini 950 M monthly users. Indonesia has 235.3 M internet users (81.72%). Forecasts are not measurements.
- **GEO** was named by Aggarwal et al. (KDD 2024). Visibility inside an answer is measured by the **position-adjusted word count** of Equation (2.2). On GEO-bench, quotations, statistics and citations raised visibility by roughly 30–40%; keyword stuffing lowered it (**Table 2.2**).
- **Profitina's definition:** SEO optimizes ranking in a list; GEO optimizes so content is *understood, extracted and cited* inside answers; they complement each other (**Table 2.3**). Profitina **estimates readiness** from observable signals and never queries the engines; results are estimates, not guarantees. Its loop is **Scan · Optimize · Monitor** (Pindai · Optimasi · Pantau).
- The **machine-readable layer**: schema.org **JSON-LD** (`@context`, `@type`, `@id`, `@graph`) makes facts unambiguous; `robots.txt` (RFC 9309) decides which crawler may read which path, by group choice and longest match (**Figure 2.3**), with separate names for training, search and user-triggered crawlers (**Table 2.4**); `llms.txt` is a Markdown map for models, widely published but rarely read.
- **Lab G6** (**Figure 2.4**) generates all three from one `business.json` and checks them independently, in Python and Java with identical files and reports; a person publishes.

2.12 Exercises

Exercise 2.1. In your own words, explain the difference between *crawling*, *indexing* and *ranking*, and say which of the three an answer engine still needs and which it replaces or changes.

Exercise 2.2. Using Equation (2.1), compute the relative change for (a) AI Overviews from 15% to 43% of searches; (b) AI Mode visits from 126 M to 279 M; (c) Gemini monthly users from 750 M to 950 M. For (a), also give the absolute change in percentage points.

Exercise 2.3. An answer has $|S| = 5$ sentences with 12, 8, 10, 6 and 14 words, in that order, numbered with positions 0 to 4. Source A is cited by sentences 0 and 3; source B by sentences 2 and 4. Compute $\mathrm{Imp}_{\mathrm{pwc}}$ of Equation (2.2) for both sources. Which is more visible, and why, although B has more words?

Exercise 2.4. A shop's website says "We are the best laundry in Surabaya! Best service, best price, best laundry Surabaya." Rewrite it as two or three sentences that follow the lessons of **Table 2.2**, using Profitina Laundry's facts from **Listing 2.1**.

Exercise 2.5. Classify each statement as a *measurement* or a *forecast*, and name the source: (a) "43% of searches show an AI Overview"; (b) "Search volume will drop 25% by 2026"; (c) "97% of llms.txt files received zero requests in May 2026".

Exercise 2.6. Add a field `"sameAs"` to `business.json`, holding a list of the laundry's social-media profile addresses, and change `build_jsonld` (Python) or `buildJsonLd` (Java) so that the business node includes it. Where in the node should it go, and what does the property mean in schema.org?

Exercise 2.7. The owner decides to block training crawlers (`GPTBot`, `ClaudeBot`, `Google-Extended`) but to allow search and user-triggered crawlers. Change `business.json` accordingly, run the generator and the checker, and show the new crawler table. Is `Google-Extended` really a crawler? What does blocking it change, according to Google?

Exercise 2.8. For the group `User-agent: *` with the rules `Allow: /`, `Disallow: /admin/` and `Allow: /admin/help`, in that order, decide by hand, first with the first-match reading and then with RFC 9309's longest-match reading, whether each of the paths `/`, `/services`, `/admin/`, `/admin/help` and `/admin/helpdesk` is allowed. Then write a program in both languages that prints the two answers side by side.

Exercise 2.9. A site's `robots.txt` contains `User-agent: *` followed by `Disallow: /admin`. Which of `/admin`, `/admin/`, `/admin.html`, `/administrator/` and `/Admin/` are blocked? How would you block exactly the folder `/admin/` and nothing else?

Exercise 2.10. Create a damaged `jsonld.html` in which one FAQ question has no `acceptedAnswer`, and another whose `about` points to `https://laundry.example/#shop`. Predict which checks of `geo_check.py` fail and what they print, then run both checkers to confirm.

Exercise 2.11. Why does `jsonld_script` replace `</` with `<\/`? Construct a `business.json` whose FAQ answer would break the page without that line, and show what the resulting HTML would look like.

Exercise 2.12. The Similarweb report says only 6.8% of U.S. ChatGPT desktop queries included citations in May 2026. Explain, in a short paragraph for a business owner, why this makes structured data and extractable facts *more* important rather than less.

2.13 Solutions to the Exercises

Solution 2.1. *Crawling* is fetching pages by following links; *indexing* is recording which words and facts appear on which pages so that they can be found quickly; *ranking* is ordering the pages that match a query. An answer engine still needs crawling and indexing, its own or a partner's, because its retrieval step can only return what was fetched and indexed. It changes ranking: instead of presenting an ordered list to a person, it passes the top passages to a language model, which *chooses* what to use and writes one answer. Ranking still matters, because only retrieved passages can be cited, but the last word belongs to the model.

Solution 2.2. (a) $(43 - 15)/15 = 28/15 \approx 1.87$, a rise of about 187%, and an absolute rise of 28 percentage points. (b) $(279 - 126)/126 = 153/126 \approx 1.21$, about +121%, more than doubling. (c) $(950 - 750)/750 = 200/750 \approx 0.27$, about +27% in two quarters.

Solution 2.3. The total is $12 + 8 + 10 + 6 + 14 = 50$ words and $|S| = 5$. The weights are $e^{-0/5} = 1$, $e^{-1/5} \approx 0.819$, $e^{-2/5} \approx 0.670$, $e^{-3/5} \approx 0.549$ and $e^{-4/5} \approx 0.449$. For A: $12 \cdot 1 + 6 \cdot 0.549 = 12 + 3.29 = 15.29$, so $\mathrm{Imp}_{\mathrm{pwc}}(A) \approx 15.29/50 \approx 0.31$. For B: $10 \cdot 0.670 + 14 \cdot 0.449 = 6.70 + 6.29 = 12.99$, so $\mathrm{Imp}_{\mathrm{pwc}}(B) \approx 12.99/50 \approx 0.26$. A is more visible although B has 24 words against A's 18, because A's main mention is the *first* sentence, with full weight, while B's words come in the second half, where the exponential factor has fallen below 0.7.

Solution 2.4. For example: "Profitina Laundry in Rungkut, Surabaya, washes and folds clothes for Rp7,000 per kilogram, with a minimum of 3 kg per order. Normal orders are ready in two days; express orders are ready the same day for 50% more. All three outlets accept cash and QRIS." The rewrite replaces superlatives, which no engine can verify, with statistics (price, minimum, time), states the entity and place in the first words, and gives self-contained sentences that can be quoted one at a time.

Solution 2.5. (a) Measurement, Similarweb data reported by TechCrunch in July 2026 [6]. (b) Forecast, Gartner, February 2024 [11]. (c) Measurement, an analysis of server logs of 137,000 domains, reported in July 2026 [23].

Solution 2.6. In `business.json` add, for example, `"sameAs": ["https://www.instagram.com/profitinalaundry.example", "https://www.facebook.com/profitinalaundry.example"]`. In Python, add the line `"sameAs": facts.get("sameAs", []),` to the `business` dictionary after `"openingHours"`; in Java, add `"sameAs", list(facts.get("sameAs"))` as a further pair in the `node(...)` call (with a comma after the `openingHours` pair). Any position inside the business node is valid JSON-LD; placing it last keeps the existing output unchanged above it. In schema.org, `sameAs` gives URLs of other pages that unambiguously identify the same entity, which helps a machine connect the website, the social profiles and a map listing as one business.

Solution 2.7. Set `"allowCrawlers": ["Googlebot", "Bingbot", "OAI-SearchBot", "ChatGPT-User", "Claude-SearchBot", "Claude-User", "PerplexityBot"]` and `"blockCrawlers": ["GPTBot", "ClaudeBot", "Google-Extended", "CCBot"]`. The generator then writes one group of seven allowed crawlers and four one-line blocked groups; in the checker's table the four blocked crawlers show `block` in every column, and the check line becomes `8 of 12 crawlers may read /` (seven named plus `SomeNewBot`). `Google-Extended` is not a crawler that fetches pages; it is a *product token* that Google's ordinary crawlers honor. Blocking it tells Google not to use the content for training Gemini models or for grounding in Gemini apps; according to Google it does not affect inclusion or ranking in Google Search [20].

Solution 2.8. By hand, first match (the first rule whose path is a prefix wins): `Allow: /` is a prefix of *every* path, so every path is allowed and the later rules are never reached. Longest match: `/` and `/services` match only `Allow: /`, so they are allowed; `/admin/` matches `Allow: /` (length 1) and `Disallow: /admin/` (length 7), and the longer one blocks it; `/admin/help` also matches `Allow: /admin/help` (length 11), the longest, so it is allowed; `/admin/helpdesk` begins with `/admin/help` too, so it is allowed as well, a reminder that rules match prefixes, not folder names. **Listing 2.8** and **Listing 2.9** print both readings.

Listing 2.8 `sol_rule_order.py`: the first-match and longest-match readings of one `robots.txt` group, side by side.

```python
# sol_rule_order.py - first-match versus longest-match on one robots.txt group
RULES = [("allow", "/"), ("disallow", "/admin/"), ("allow", "/admin/help")]
PATHS = ["/", "/services", "/admin/", "/admin/help", "/admin/helpdesk"]


def first_match(path):
    for kind, prefix in RULES:              # the order of the file decides
        if path.startswith(prefix):
            return kind
    return "allow"


def longest_match(path):
    best = None
    for kind, prefix in RULES:              # RFC 9309: longest prefix decides
        if path.startswith(prefix):
            if (best is None or len(prefix) > len(best[1])
                    or (len(prefix) == len(best[1]) and kind == "allow")):
                best = (kind, prefix)
    return "allow" if best is None else best[0]


def main():
    for kind, prefix in RULES:
        print(f"{kind.capitalize()}: {prefix}")
    print()
    print(f"{'Path':<18}{'first match':<14}longest match")
    for p in PATHS:
        a, b = first_match(p), longest_match(p)
        flag = "  <- differ" if a != b else ""
        print(f"{p:<18}{a:<14}{b:<14}".rstrip() + flag)


if __name__ == "__main__":
    main()
```

Listing 2.9 `SolRuleOrder.java`: the same comparison in Java.

```java
// SolRuleOrder.java - first-match versus longest-match on one robots.txt group
import java.util.List;

public class SolRuleOrder {
    record Rule(String kind, String prefix) { }

    static final List<Rule> RULES = List.of(new Rule("allow", "/"),
            new Rule("disallow", "/admin/"), new Rule("allow", "/admin/help"));
    static final List<String> PATHS =
            List.of("/", "/services", "/admin/", "/admin/help",
                    "/admin/helpdesk");

    static String firstMatch(String path) {
        for (Rule r : RULES) {                    // the order of the file decides
            if (path.startsWith(r.prefix())) {
                return r.kind();
            }
        }
        return "allow";
    }

    static String longestMatch(String path) {
        Rule best = null;
        for (Rule r : RULES) {                    // RFC 9309: longest prefix decides
            if (!path.startsWith(r.prefix())) {
                continue;
            }
            int len = r.prefix().length();
            if (best == null || len > best.prefix().length()
                    || (len == best.prefix().length()
                        && r.kind().equals("allow"))) {
                best = r;
            }
        }
        return best == null ? "allow" : best.kind();
    }
```

Listing 2.9 (Continued) *SolRuleOrder.java: the same comparison in Java.*

```java
    public static void main(String[] args) {
        for (Rule r : RULES) {
            String k = r.kind();
            System.out.println(Character.toUpperCase(k.charAt(0)) + k.substring(1)
                    + ": " + r.prefix());
        }
        System.out.println();
        System.out.printf("%-18s%-14s%s%n", "Path", "first match", "longest match");
        for (String p : PATHS) {
            String a = firstMatch(p);
            String b = longestMatch(p);
            String flag = a.equals(b) ? "" : "  <- differ";
            System.out.println(String.format("%-18s%-14s%-14s", p, a, b)
                    .stripTrailing() + flag);
        }
    }
}
```

Both print:

```
Allow: /
Disallow: /admin/
Allow: /admin/help

Path              first match   longest match
/                 allow         allow
/services         allow         allow
/admin/           allow         disallow  <- differ
/admin/help       allow         allow
/admin/helpdesk   allow         allow
```

Only `/admin/` differs, and it differs in the worst direction: the first-match reading exposes the private area. This is why lab G6 writes the narrow rules first.

Solution 2.9. `Disallow: /admin` blocks every path that *begins* with the six characters `/admin`: `/admin`, `/admin/`, `/admin.html` and `/administrator/`. It does not block `/Admin/`, because path matching is case-sensitive. To block exactly the folder, write `Disallow: /admin/`, with the trailing slash; this blocks `/admin/` and everything below it, but not `/admin.html` or `/administrator/`. (The bare path `/admin` without a slash then stays allowed; if the server redirects it to `/admin/`, the crawler will be stopped there.)

Solution 2.10. The checks `JSON-LD parses as JSON`, `@context`, `one business node` and the business-properties check still pass. The FAQ check fails and prints `[FAIL] FAQPage has 4 questions, 3 with an answer`, because the question without `acceptedAnswer` is not counted as good. The link check fails with `[FAIL] FAQPage.about points to a node in the graph`, because `#shop` is not the `@id` of any node. Both checkers print the same lines, since they apply the same rules, and exit with code 1.

Solution 2.11. Inside an HTML `<script>` element the browser does not understand JSON; it looks only for the characters `</script>` to know where the element ends. A FAQ answer such as `"Type </script><script>alert(1)</script> to test"` would therefore end the JSON-LD block in the middle of a string, and the rest would be read as HTML, a broken page at best and an injected script at worst. With the replacement the text becomes `"Type <\/script><script>alert(1)<\/script> to test"`: still the same JSON string for any JSON parser, since `\/` means `/`, but no longer a closing tag for the browser. The owner's facts file should never contain such text, but a generator must be safe even when its input is not.

Solution 2.12. When most answers name no source, the few that do name one choose it carefully, and the engine can only name what it could read and understand. If your hours, prices and address are stated plainly on the page and repeated in structured data, an engine that decides to be specific can be specific *about you*, and correctly. If they are hidden in images or contradicted by an old listing, even an engine that would like to mention you may get the facts wrong or pick a competitor whose facts are clearer. And even an answer that cites nothing is still built from what the engine read: correct, extractable facts are what make that uncited answer say the right thing about your business.

References

[1] S. Brin and L. Page, "The anatomy of a large-scale hypertextual Web search engine," *Computer Networks and ISDN Systems*, vol. 30, no. 1–7, pp. 107–117, 1998.

[2] P. Aggarwal, V. Murahari, T. Rajpurohit, A. Kalyan, K. Narasimhan, and A. Deshpande, "GEO: Generative Engine Optimization," in *Proc. 30th ACM SIGKDD Conf. Knowledge Discovery and Data Mining (KDD '24)*, 2024. arXiv:2311.09735. [Online]. Available: https://arxiv.org/abs/2311.09735

[3] P. Lewis *et al.*, "Retrieval-augmented generation for knowledge-intensive NLP tasks," in *Advances in Neural Information Processing Systems 33 (NeurIPS 2020)*, 2020.

[4] Profitina, "Profitina — GEO platform," marketing site. [Online]. Available: https://profitina.com (accessed Sep. 24, 2026).

[5] A. Chapekis and A. Lieb, "Google users are less likely to click on links when an AI summary appears in the results," Pew Research Center, Jul. 22, 2025. [Online]. Available: https://www.pewresearch.org/short-reads/2025/07/22/google-users-are-less-likely-to-click-on-links-when-an-ai-summary-appears-in-the-results/

[6] S. Perez, "Google's AI search is rapidly becoming the default, new data shows," *TechCrunch*, Jul. 27, 2026 (Similarweb data). [Online]. Available: https://techcrunch.com/2026/07/27/googles-ai-search-is-rapidly-becoming-the-default-new-data-shows/

[7] Search Engine Land, "OpenAI: ChatGPT now has 900 million weekly active users," Feb. 2026. [Online]. Available: https://searchengineland.com/chatgpt-900-million-weekly-active-users-470492

[8] T. Roush, "950 million people now use Gemini each month as Alphabet posts earnings beat," *Forbes*, Jul. 22, 2026. [Online]. Available: https://www.forbes.com/sites/tylerroush/2026/07/22/950-million-people-now-use-gemini-each-month-as-alphabet-posts-earnings-beat/

[9] Perplexity AI Magazine, "Perplexity AI monthly queries," secondary report of A. Srinivas's statement at Bloomberg Tech, Jun. 5, 2025. [Online]. Available: https://perplexityaimagazine.com/perplexity-hub/perplexity-ai-monthly-queries/

[10] ANTARA, "Survei APJII: Penetrasi internet di Indonesia 2026 capai 81,7 persen," May 21, 2026. [Online]. Available: https://www.antaranews.com/berita/5576225/survei-apjii-penetrasi-internet-di-indonesia-2026-capai-817-persen

[11] Gartner, "Gartner predicts search engine volume will drop 25% by 2026, due to AI chatbots and other virtual agents," press release, Feb. 19, 2024. [Online]. Available: https://www.gartner.com/en/newsroom/press-releases/2024-02-19-gartner-predicts-search-engine-volume-will-drop-25-percent-by-2026-due-to-ai-chatbots-and-other-virtual-agents

[12] Profitina, *Profitina Whitepaper — Public Version v24*, Sep. 2026. [Online]. Available: https://app.profitina.com/static/whitepaper/

[13] Schema.org, "Schema.org releases: Version 30.1," Sep. 16, 2026. [Online]. Available: https://schema.org/docs/releases.html

[14] Schema.org, "DryCleaningOrLaundry." [Online]. Available: https://schema.org/DryCleaningOrLaundry (accessed Sep. 2026).

[15] W3C, "JSON-LD 1.1: A JSON-based Serialization for Linked Data," W3C Recommendation, Jul. 16, 2020. [Online]. Available: https://www.w3.org/TR/json-ld11/

[16] M. Koster, G. Illyes, H. Zeller, and L. Sassman, "Robots Exclusion Protocol," IETF RFC 9309, Sep. 2022.

[17] OpenAI, "Overview of OpenAI crawlers." [Online]. Available: https://developers.openai.com/api/docs/bots (accessed Sep. 2026).

[18] Anthropic, "Does Anthropic crawl data from the web, and how can site owners block the crawler?" [Online]. Available: https://support.claude.com/en/articles/8896518 (accessed Sep. 2026).

[19] Perplexity, "Perplexity crawlers." [Online]. Available: https://docs.perplexity.ai/docs/resources/perplexity-crawlers (accessed Sep. 2026).

[20] Google, "Google's common crawlers," *Google Search Central*. [Online]. Available: https://developers.google.com/search/docs/crawling-indexing/google-common-crawlers (accessed Sep. 2026).

[21] J. Howard, "/llms.txt—a proposal to provide information to help LLMs use websites," Answer.AI, Sep. 3, 2024. [Online]. Available: https://www.answer.ai/posts/2024-09-03-llmstxt.html

[22] Rankability, "llms.txt adoption," scan of Sep. 18, 2026 (Tranco list of Sep. 17, 2026). [Online]. Available: https://www.rankability.com/data/llms-txt-adoption/

[23] PPC Land, "llms.txt adoption rises 8.8x but 97% of files get zero AI requests," Jul. 2, 2026. [Online]. Available: https://ppc.land/llms-txt-adoption-rises-8-8x-but-97-of-files-get-zero-ai-requests/

[24] Python Software Foundation, "urllib.robotparser — Parser for robots.txt," *The Python Standard Library*. [Online]. Available: https://docs.python.org/3/library/urllib.robotparser.html

[25] Python Software Foundation, "Support RFC 9309 in robotparser," CPython issue #138907, Sep. 15, 2025. [Online]. Available: https://github.com/python/cpython/issues/138907

Chapter 3

What Profitina Does: The Visibility Loop and a User's Journey

By the end of this chapter you will be able to describe Profitina's working method as one loop with four verbs, *measure, explain, fix, monitor*, and map those verbs onto the three tabs a user actually sees, **Scan · Optimize · Monitor** in English and **Pindai · Optimasi · Pantau** in Indonesian. You will know what a scan looks at and what it deliberately does not do, how a single number becomes a list of fixes that a shop owner can act on, what the Content Engine and the Technical Assistant produce, and why monitoring turns a snapshot into evidence. You will meet the two reports that carry the same loop from websites to scholarship, the Researcher Visibility Report and Organization Visibility, and the distribution tool called *One Text, Many Platforms*. You will then follow the owner of a small fictional business, Profitina Laundry in Surabaya, through the whole journey, from her first visit to the moment she exports or deletes her own data, screen by screen, in wireframes drawn for this book. Finally you will build lab G1, the flagship program of this book: a mini AI-readiness checker that fetches a page, its `robots.txt` and its `llms.txt` from a tiny local server, inspects the structured data, the headings, the amount of extractable text and the basic meta tags, and reports eight checks and a transparent teaching score. The score is our own, built in the open so that you can see and change every rule; it is not Profitina's score. It is written in Python and in Java, and both versions print exactly the same report.

3.1 Where We Left Off

Chapter 2 explained why being *found* online is no longer the same as being *ranked*. A search engine returns a list of links and lets a person choose; an answer engine reads sources and writes a single answer, in which a business is either cited or absent. Research named the new discipline *Generative Engine Optimization*, GEO, and measured that sources written with precise, quotable substance are more visible inside generated answers [1]. Chapter 2 also introduced the machine-readable layer that every GEO effort begins with, schema.org structured data in JSON-LD, `robots.txt` and `llms.txt`, and its lab G6 *generated* those three pieces for Profitina Laundry.

This chapter turns from the discipline to the product. Profitina calls itself Indonesia's AI Visibility (GEO) management and optimization platform, and it describes its job in one sentence: it measures how ready a brand, a website, a researcher or an organization is to be cited by AI answer engines such as ChatGPT, Gemini and Perplexity, from real and observable signals, and then helps to fix what it finds [2]. That sentence contains the whole of this chapter. *Measures* is the scan. *From observable signals* is a design decision with consequences. *Helps to fix* is the Content Engine and the Technical Assistant. And the word *ready* is chosen with care: Profitina estimates readiness, it does not promise results. Its own terms say so plainly: "Visibility results are methodological estimates, not guarantees of actual third-party AI engine output" [2], [3].

We proceed in three movements. Sections 3.2 to 3.8 describe *what* the product does, feature by feature, in the order in which a user meets them. Section 3.9 then shows the same features *in use*, as a journey of nine steps through the life of one small business. Section 3.10 is the lab: having seen what a readiness scan is for, we write a small one ourselves, so that the ideas stop being

abstract. Where the book describes Profitina, it describes the *what* and the *why* at the level of boxes and arrows; where it wants to show the *how*, it builds its own teaching version, with rules we choose and publish.

3.2 The Visibility Loop

3.2.1 Four verbs, three tabs

Every feature in this chapter serves one of four verbs, and the verbs form a loop. **Figure 3.1** draws it.

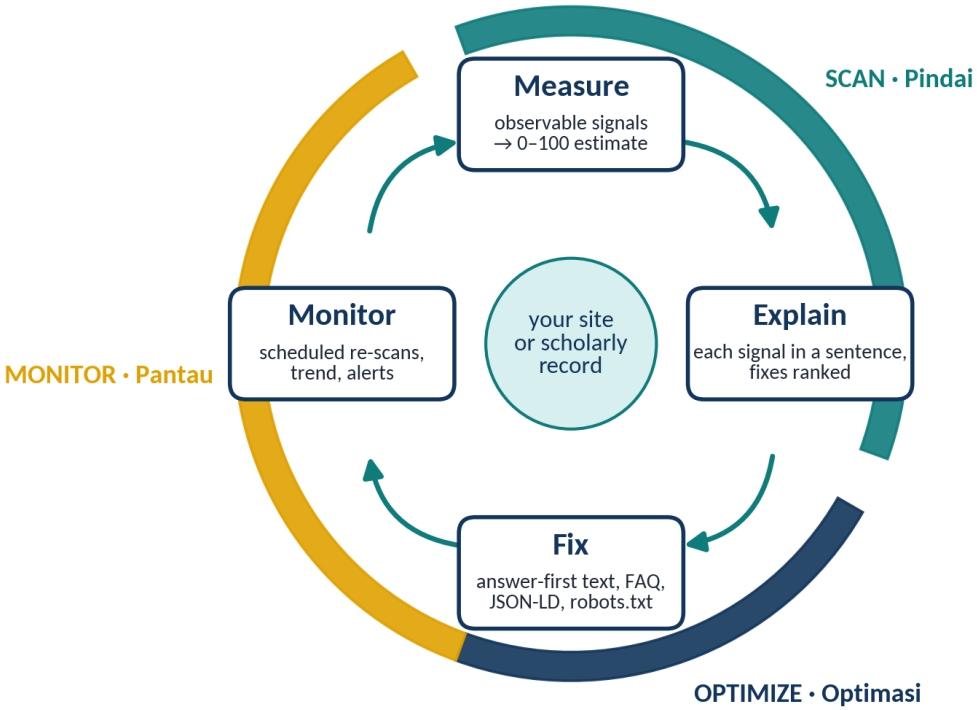


Figure 3.1 The visibility loop: measure, explain, fix and monitor, grouped into the three tabs of the application, Scan, Optimize and Monitor (Pindai, Optimasi, Pantau).

Measure. Look at the thing to be cited, a business website or a scholarly record, and collect signals that can be observed directly: can the AI crawlers read the site, does it publish structured data, is the content written so that a program can extract a self-contained answer from it? Combine the signals into an estimate of readiness on a scale from 0 to 100.

Explain. A number alone helps nobody. Turn every signal into a sentence a non-specialist understands ("two AI crawlers are blocked by your `robots.txt`"), and order the resulting fixes so that the most useful comes first.

Fix. Close the gaps. Some fixes are text: a first paragraph that says who the business is and what it does, an FAQ with the questions customers really ask. Some are technical: a JSON-LD block, a `robots.txt` that admits the crawlers the owner wants, a sitemap registered with the search engines. Profitina drafts these; the owner publishes them on her own site.

Monitor. Scan again, on a schedule, and keep the history. A single scan is a photograph; a weekly series is evidence that the fixes worked, or a warning that something broke.

The application groups the four verbs into three tabs, because *measure* and *explain* happen on the same screen, in the same moment: the result of a scan *is* its explanation. The three tabs carry a

name in each of the application's two languages, and the names were chosen so that both form an acronym of their own (**Table 3.1**).

Table 3.1 The four verbs of the loop and the three tabs of the application: SOM in English, POP in Indonesian.

Verb of the loop	English tab	Indonesian tab	What the user gets
Measure	**S**can	**P**indai	a 0–100 readiness estimate from observable signals
Explain	**S**can	**P**indai	each signal in one sentence; a prioritized fix list
Fix	**O**ptimize	**O**ptimasi	answer-first drafts, FAQ, JSON-LD, `robots.txt` and sitemap help
Monitor	**M**onitor	**P**antau	scheduled re-scans, a weekly trend, alerts, history

SOM reads Scan · Optimize · Monitor; *POP* reads Pindai · Optimasi · Pantau. An earlier version of the product called the same method the "3P" methodology; the loop has not changed, only its name. The marketing site adds a three-word motto that belongs to the whole platform rather than to the loop, *Gather · Grow · Give*, in Indonesian *Bersama · Berkembang · Berbagi* [2].

3.2.2 Why a loop and not a line

It would be simpler to sell GEO as a one-off service: scan, fix, done. The loop exists because readiness decays, for three reasons that every site owner eventually meets.

First, **sites change**. A new theme, a plugin update or a migration to another host can remove the structured data or rewrite `robots.txt` without anyone noticing. Profitina's own marketing site learned this the hard way: a content-delivery network's "managed `robots.txt`" feature once silently disallowed `GPTBot` and `ClaudeBot` on profitina.com, although the file in the site's own source was correct. The lesson the team wrote down is short: always verify the live URL, bypassing the cache [2]. Chapter 10 builds an auditor for exactly that problem; in this chapter it is the reason monitoring exists.

Second, **the engines change**. Crawlers are renamed and split; the major AI companies now publish separate names for their training crawlers, their search crawlers and the fetchers that act for a user, and they revise the list [4], [5], [6]. A `robots.txt` written two years ago may admit one of them and not the other.

Third, **the owner learns**. The first scan of a small business usually finds the obvious gaps. Once those are closed, the next scan finds subtler ones, such as a first paragraph that never names the business, or prices written only inside an image. Improvement comes in rounds.

A loop is also the honest shape for a product that cannot promise outcomes. No one outside an answer engine controls what it will say, so what a platform can offer is a disciplined, repeated way of raising the odds and checking that the raised odds stay raised.

3.2.3 What is measured, and what is not

The most important design decision in the product is a refusal. Profitina does **not** measure a business's visibility by asking ChatGPT, Gemini or Perplexity about it. The answer engines are the *targets* whose readiness to cite a business is estimated; they are never components that Profitina depends on [2].

There are three reasons, and each would suffice alone. *Independence*: a measurement that calls a third party's service breaks when that service changes its terms, its prices or its output, and an

engine's answer to the same question varies from day to day. *Privacy*: querying an outside provider with a business's name, its competitors and its customers' questions sends exactly the data a small business would rather keep. *Cost and principle*: a house rule of the platform forbids depending on paid third-party APIs. The research of Chapter 2 makes the choice reasonable rather than merely convenient. The properties that make a source visible inside answers, extractable substance, clear structure, precise facts, open access for crawlers, are properties of the *source*, and they can be observed at the source [1].

The same refusal shapes the local AI. Wherever Profitina uses a language model, to draft content, to classify a short text, to sort feedback, it uses one that runs on its own server, a model of the Qwen family served by Ollama, so that user data never leaves the machine for AI processing [2]. Chapter 6 explains the two lanes of that local AI.

3.3 Measure: What a Scan Looks At

3.3.1 The input

A scan starts from very little. A business enters its name and its domain; a creator enters a name and a social-media account. From that, the AI Visibility Engine **auto-fills** the rest of what a scan needs: the likely category of the business, its location, and a handful of realistic questions that a customer might put to an answer engine, such as "laundry kiloan Surabaya" or "dry cleaning near Rungkut" [2]. The owner checks the filled-in values and may change them. The design principle is that the person with the least technical knowledge should be able to start, and that the machine should do the typing.

Auto-fill raises a question that every classifier faces: what should it do when it is unsure? Many systems guess silently and present the guess as if it were a fact. Profitina's category detection does the opposite. When it cannot decide, it **says "not sure yet"** and asks the user to choose from a short list [2]. That small behavior is a principle of the whole platform in miniature: an honest "I don't know" is worth more than a confident mistake, because a wrong category would silently skew every later step of the scan.

3.3.2 Three families of signals

What does a scan observe? At the level at which the product describes itself publicly, the signals fall into three families [2]. **Table 3.2** lists them together with the open standards through which each can be observed. The table names families and standards, not the product's internal signal list, its weights, or how it combines them; those are the product's own and are not the subject of this book. What matters for understanding is that every family is made of things anyone can check.

Table 3.2 The three families of observable signals behind a readiness estimate, and the public standards through which each can be observed.

Family	The question it asks	Where the evidence lives (open standards)
Access	May the engines' crawlers read the site at all? Can they find its pages?	`robots.txt`, RFC 9309 [7]; sitemaps [8]; crawler documentation of each company [4], [5], [6], [9], [10]
Structure	Does the site describe itself in machine-readable form?	schema.org types and properties [11] written as JSON-LD [12]; `llms.txt` [13]
Extractability	Can a program lift a self-contained, correct answer from the text?	HTML headings and document structure, titles, meta descriptions, language [14]; the visible text itself

Access comes first because nothing else matters without it. An engine cannot quote a page its crawler was not allowed to fetch, and it cannot fetch pages it never discovers. Chapter 2 explained how `robots.txt` groups and rules work [7]; a scan asks the practical question for each crawler that matters.

Structure is what makes facts unambiguous. A price written in a decorative banner is a guess for a program; a `priceRange` in JSON-LD is a fact. An FAQ marked up as `FAQPage` hands an engine exactly the kind of short, self-contained statement that research found to be most visible [1], [11], [12].

Extractability is the quality of the text itself. Is there one clear main heading, with sections nested under it? Does the page state *who* and *what* in its first lines, or does it open with "Welcome!" and a slideshow? Is there enough real text, or is the page mostly scripts, styles and images? These are old virtues of good web writing; answer engines have made them measurable consequences.

3.3.3 Progress you can read

A scan takes time: it fetches pages, parses them, runs the checks, and consults the local model for some steps. A spinner that turns for forty seconds tells the user nothing and invites her to close the tab. Profitina shows a **granular progress bar from 0 to 100%** with labels for the sub-steps, such as "query 3/8", and an estimate of the time remaining [2]. It is a small thing, and it is the difference between a user who trusts the tool and one who suspects it has hung.

3.3.4 The number, and what it means

The result is a readiness estimate on a scale from 0 to 100, with its methodology shown openly and a list of prioritized fixes beside it. Three properties of that number deserve to be stated precisely, because they are exactly the properties a careless reader assumes away.

It is an **estimate of readiness**, not a measurement of citations. A site can score well and still not be quoted today, because an engine's choice depends also on the question, on competing sources and on the engine's own changes.

It is **relative to observable signals**. Anything that cannot be observed from the outside, such as a business's reputation among its customers, lies outside it.

It is **for comparison over time**, above all. The same site scanned before and after a fix, by the same method, gives the owner a meaningful difference. That is why the product is a loop.

> **KEY INSIGHT.** A readiness score answers "how well does this site make itself readable, understandable and quotable by machines?" It does not answer "will ChatGPT mention me

> tomorrow?" No honest tool can answer the second question, and Profitina says so in every result [2], [3].

3.4 Explain: From a Number to a List of Fixes

A number is a verdict; a list is a plan. The scan's result page therefore spends most of its space on the list. Each observed gap becomes one line in plain language, with the evidence that produced it and a next step. The owner of a laundry does not need to know what RFC 9309 is; she needs to read "GPTBot and ClaudeBot are blocked by your `robots.txt`; remove those two `Disallow` lines if you want to be read by them", and to know that it matters more than the missing `lang` attribute.

Two ideas govern the list, and both can be stated without describing how the product ranks.

Every line is explained. A line that says only "structured data: 40%" fails the owner. A line that says "your JSON-LD block is not valid JSON, so programs ignore it; the error is a comma after the last property" gives her something to do. In the lab of Section 3.10 every check returns exactly three things for this reason: its points, its *evidence* and its *advice*.

Order by consequence. Some gaps block everything behind them: a disallowed crawler cannot read the perfect FAQ. Others are refinements. A useful list puts the blockers first. A simple way to think about it, which is ours and not the product's, is to ask of each fix two questions, "what does it unblock?" and "how much work is it?", and to prefer fixes that unblock much for little work. Removing two lines from `robots.txt` scores high on both.

The result page also carries a small link that is easy to overlook and important to the platform: **send feedback on this scan**. A user who thinks a result is wrong can say so. Chapter 7 follows that feedback through a five-step pipeline in which a human, never an algorithm, decides what changes in the next release [2].

3.5 Fix: The Content Engine and the Technical Assistant

3.5.1 What the Content Engine drafts

The *Optimize* tab turns gaps into publishable material. Its **Content Engine** writes GEO drafts: an *answer-first* text that states in its first two to four sentences who the business is, what it does and where, backed by specific and checkable facts; a set of entities and key points written to be extracted; an FAQ phrased as the questions customers actually type; and the matching schema.org JSON-LD [2]. The owner reviews, corrects and publishes on her own site. Profitina never publishes on a user's behalf.

What does *answer-first* mean in practice? Compare two openings for the same laundry.

Welcome! Clean clothes, happy life. We are the best and the fastest. Your satisfaction is our priority.

Profitina Laundry is a laundry in Rungkut, Surabaya, that washes, irons and dry-cleans clothes for households and small businesses. Wash-and-fold costs Rp7,000 per kilogram with a minimum of 3 kg, a normal order is ready in two days, and every outlet accepts cash and QRIS.

The first says nothing that a program could quote in answer to any question. The second answers at least five: what is Profitina Laundry, where is it, what does it do, how much does it cost, how long does it take. Each of its facts can be checked, and none of them is a superlative. The lab of

Section 3.10 contains a check, which we call *answer-first*, that asks one narrow version of this question: does the name of the business appear in the first sixty words of the main content?

3.5.2 An AI draft only when it is better

The Content Engine runs on the platform's local model. A model can write fluent text that is worse than a plain, correct template: vaguer, longer, or confidently wrong about a fact. Profitina therefore puts a **quality gate** in front of every AI draft. The draft is compared with what a built-in template would produce for the same input, and it is served only if it is genuinely better; in the benchmark after the September 2026 revision, the AI draft won in 5 of 6 cases [2]. How the gate scores drafts belongs to the product and is not described here. The principle is what matters, and it can be copied into any system: *use the model only where it beats the simple thing.*

A related rule covers failure. When no model is available, because the process is restarting or the machine is busy, the application **says so**, rather than silently serving the template as if it were an AI draft [2]. A user who is told "the model is unavailable, here is the template, marked as such" can decide what to do. A user who is deceived cannot.

3.5.3 The Technical Assistant

The second half of the *Optimize* tab handles the machine-readable layer. The **Technical Assistant** produces the JSON-LD block for the business, helps the owner write a `robots.txt` that admits the crawlers she chooses (the list typically includes `GPTBot`, `OAI-SearchBot`, `ClaudeBot`, `PerplexityBot`, `Google-Extended` and the classic search crawlers), and gives guidance for submitting the sitemap to Bing Webmaster Tools [2]. Bing gets a line of its own for a practical reason: in Profitina's guidance, Bing's index is where Microsoft's Copilot and ChatGPT's search draw many of their sources, so registering there, and letting `Bingbot` in, is one of the cheapest high-value steps a small business can take [2], [10].

Lab G6 in Chapter 2 generated those same three artifacts from a file of facts. The Technical Assistant does the same job interactively, for a business that has never heard of any of them.

3.6 Monitor: History, Schedules and Alerts

3.6.1 A weekly trend

The *Monitor* tab keeps every scan of a brand and draws the **weekly score trend** as a chart; the history can be exported as CSV [2]. Scan history is **private to its owner**: nobody else, including other users, sees which sites a business watches or how they scored. A **scheduler** re-scans automatically, at a cadence that depends on the plan: weekly on the Free plan, daily on Pro. When a re-scan finds a **significant change**, up or down, the platform raises an alert, optionally by email [2].

3.6.2 What "significant" means

Let s_t be the score of a brand in week t, for $t = 1,2,3, \ldots$ The change from one week to the next is

$$\Delta_t = s_t - s_{t-1} \tag{3.1}$$

and a simple alert rule, which we choose for teaching, fires when $\Delta_t \geq \delta$ or $\Delta_t \leq -\delta$ for a threshold δ that the owner picks. The absolute value makes the rule symmetric: a drop matters as much as a rise, and usually more, because a drop is often the first sign that an update broke

something. With $\delta = 10$ and the series 41,44,58,63,61, Equation (3.1) gives the changes $+3, +14, +5, -2$, and only the week with $\Delta_t = +14$ triggers an alert. If the site had been broken by a theme update in the fifth week and scored 45, the change -18 would have triggered one too.

A rule of this kind has a weakness that every monitoring system must face: noise. If a score jumps around by a few points from week to week for reasons nobody controls, a small δ produces alerts that people learn to ignore. A threshold that is too large misses real breakage. The product's own threshold is its own business; the lesson for any system is to choose δ larger than the ordinary week-to-week variation and small enough to catch a broken `robots.txt`, which usually moves a score by a large amount at once. Exercise 3.9 works through an example.

3.7 Beyond Websites: Researcher and Organization Reports

The loop was built for businesses, but its idea, measure what can be observed, explain the gaps, help close them, applies equally to scholarship. A lecturer's publications are cited by answer engines only if they can be *found*: under one consistent name, linked to a persistent identifier, with open copies that a crawler can read. Since the end of August 2026, Profitina carries the loop into that world with two reports [2].

3.7.1 The Researcher Visibility Report

The **Researcher Visibility Report** measures how complete and retrievable a scholarly record is. Its data source is **OpenAlex**, an open catalog of scholarly works, researchers, venues and institutions that absorbs Crossref's metadata and is free to query [15]. For a researcher's record it checks the things that decide whether the works can be found and attributed: whether the works carry a **DOI**, whether the researcher is linked to an **ORCID iD**, the persistent identifier for researchers [16], how the **affiliation** is spelled across the works, whether copies are **open access**, and how the works are cited. Each gap is named concretely, with a way to close it: "No ORCID linked", or "Your affiliation is written in *N* different spellings" [2].

The report has a feature that deserves a name of its own: a **permanent negative control**. Fictional names always return "not found", never a plausible-looking score. In the product's own test, 8 of 8 fictional subjects stayed at 0.0 [2]. Why does that matter so much? Because a system that returns a number for *any* input cannot be trusted with the inputs that matter. If a made-up name received a score of 34, every real score would be suspect: how much of it is the real record, and how much is the system's habit of producing numbers? A negative control is the cheapest proof that the numbers come from the data.

The report processes *works and their metadata*, not the person: no address, no photograph, no social links, nothing from a Community profile. A person who is not found receives no score at all [2].

3.7.2 Organization Visibility

Organization Visibility applies the same data at the level of an institution. It is deliberately *not* a leaderboard of people; it reports only aggregate gaps that the institution can close. The product's test on Institut Teknologi Sepuluh Nopember (ITS) is instructive. In 500 sampled works, the institution's affiliation appeared in **466 different spellings**, and **244 of 607 researchers** had no linked ORCID [2]. Neither number says anything about the quality of anyone's research. Both say something precise about why that research is harder than it should be to find and attribute, and both suggest a concrete institutional fix: one canonical spelling of the affiliation in every

submission, and an ORCID drive. The report covers organizations with works indexed in OpenAlex.

3.7.3 The Academic Citation Scan

A third service, the **Academic Citation Scan**, assesses the citation-readiness of a site from the signals available on it, and can be paid for with PROFIT, the platform's utility coin of Chapter 12. The product distinguishes it explicitly from Scopus and Google Scholar: it is not a citation index and does not claim to count citations [2].

The Organization Visibility idea, counting distinct spellings of one name, is simple enough to program in a few lines: normalize every spelling (case, spaces, punctuation) and count the members of the resulting set. Chapter 13's lab G9 does exactly that against the public OpenAlex API, with a negative control of its own.

3.8 One Text, Many Platforms

Visibility in answer engines is one channel; a small business also lives on social media, where each platform has its own tone, its own conventions and its own length limits. **One Text, Many Platforms** addresses that chore. The user pastes one source text and picks one platform from six, X, Instagram, LinkedIn, YouTube, TikTok or Facebook, and receives a ready-to-paste version that fits that platform's tone and is **guaranteed to fit its character limit** [2].

Three details of its design repay attention.

One request at a time, honestly queued. The adaptation runs on the local model, and the machine runs one such request at a time for the whole system. Rather than hiding that, the interface shows the user's **position in the queue and an estimated wait** [2]. A queue that is visible is tolerable; a queue that is hidden looks like a broken product.

A guarantee that can be kept. The feature guarantees the character limit, which a program can check exactly, and it explicitly does *not* promise that the result will be "more engaging" or "viral", which no program can check [2]. Promise what you can verify.

Sharing without an API. For X, a *Share to X* button opens X's web intent in the user's own account, with the text prefilled; the user presses "post" herself. Profitina's server only composes the text and never contacts X; there is no X API in the product, and sharing earns no PROFIT, because a share cannot be verified [2]. The feature also works without an account, which makes it one of the first things a visitor can try.

3.9 A User's Journey: Profitina Laundry, Step by Step

The features of Sections 3.2 to 3.8 are easier to understand in use. This section follows one user through nine steps. The business is **Profitina Laundry**, the fictional laundry that serves as this book's case study: an outlet in Rungkut, Surabaya, run by its owner, whom we shall call Sari, with a small website at `laundry.example`. **Figure 3.2** is the map of her journey; the wireframes that follow were drawn for this book to show the shape of each screen, and they simplify the real interface.

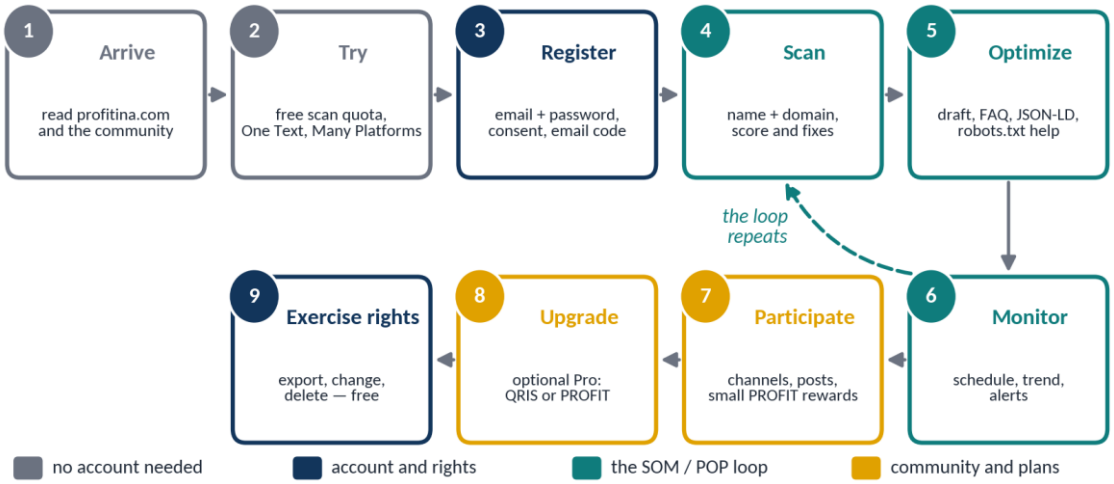


Figure 3.2 The nine steps of a user's journey. Steps 1 and 2 need no account; steps 4 to 6 are the SOM/POP loop, and the dashed arrow shows that the loop repeats.

3.9.1 Step 1: Arrive

Sari hears about Profitina from a friend who sells siomay online and visits profitina.com. The marketing site is written the way it advises its users to write: answer-first pages, an FAQ marked up as structured data, and an `llms.txt` at its root [2]. Embedded in the pages is the platform's **community widget**, the real-time feed of Chapter 4, which an anonymous visitor may read but not join. She reads a few posts from other small businesses and a thread about `robots.txt`, and decides to try.

3.9.2 Step 2: Try without an account

Two features work without registering. A visitor has a small **free scan quota**, enough to see what a scan is; and **One Text, Many Platforms** works for anyone [2]. Sari pastes the announcement she wrote for her regulars, "From 1 October we open at 06:30 on weekdays", chooses Instagram, and gets back a caption that sounds like Instagram. It is a small, immediate benefit, and it is deliberate: a tool should prove its worth before it asks for an email address.

3.9.3 Step 3: Register

Registration asks for the minimum: an **email address and a password**, nothing else. The consent checkbox for the Privacy Policy and the Terms is **never pre-ticked**, and a short bilingual notice says what is collected and why (**Figure 3.3**). The password is stored only as a one-way Argon2id hash [2], [17].

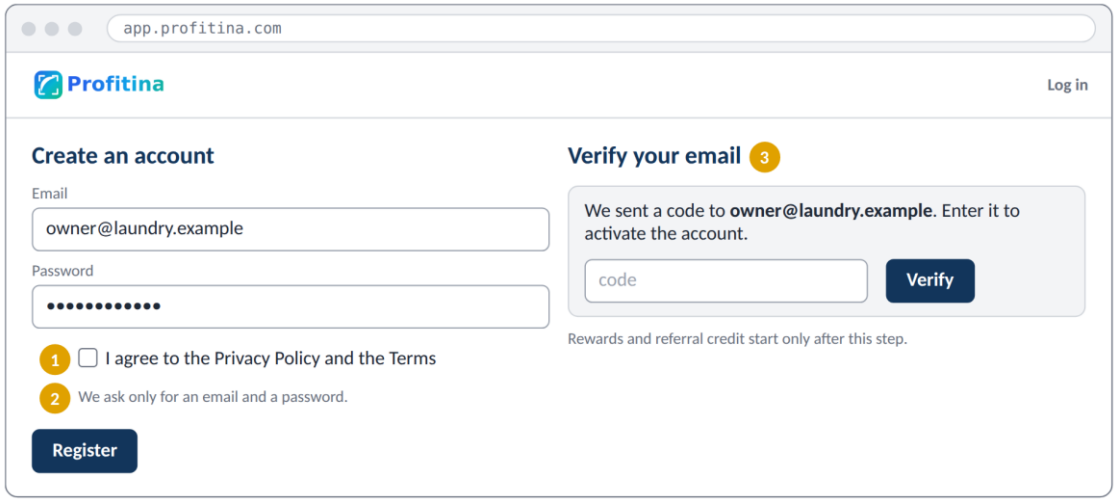


Figure 3.3 Registration, as a wireframe. (1) The consent box starts empty; (2) only an email and a password are asked for; (3) a code sent by email activates the account.

After registering, Sari receives a **one-time code by email** and types it in. Email verification is the platform's first layer of defense against bots: an automated sign-up that cannot read a mailbox gets no further. It also protects the reward system of later chapters, because rewards and referral credit start only once the email is verified [2].

3.9.4 Step 4: Scan

Sari opens the *Scan* tab (*Pindai* in her Indonesian interface) and types her business name and domain. **Figure 3.4** shows what happens next.

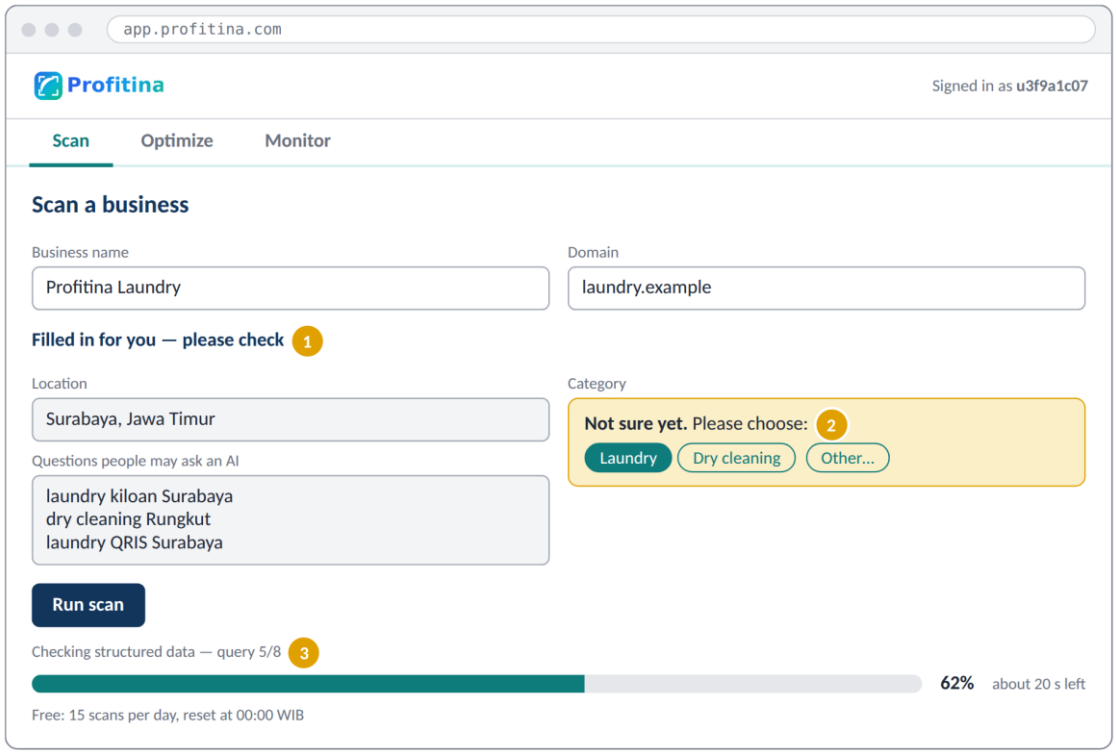


Figure 3.4 The Scan tab, as a wireframe. (1) Location and customer questions are filled in automatically for the owner to check; (2) when the category is uncertain, the system says "not sure yet" and asks; (3) progress is shown with its sub-step and the time remaining.

The location and three customer questions appear filled in (marker 1). The category does not: her site mentions washing, ironing *and* dry cleaning, and the system, unsure whether this is a laundry or a dry cleaner, says **"not sure yet"** and offers a choice (marker 2). She taps *Laundry*. The scan runs with a progress bar that names its current step (marker 3). The Free plan allows 15 scans a day, and the counter resets at 00:00 WIB, midnight in Jakarta's time zone [2].

The result (**Figure 3.5**) is sobering, and useful.

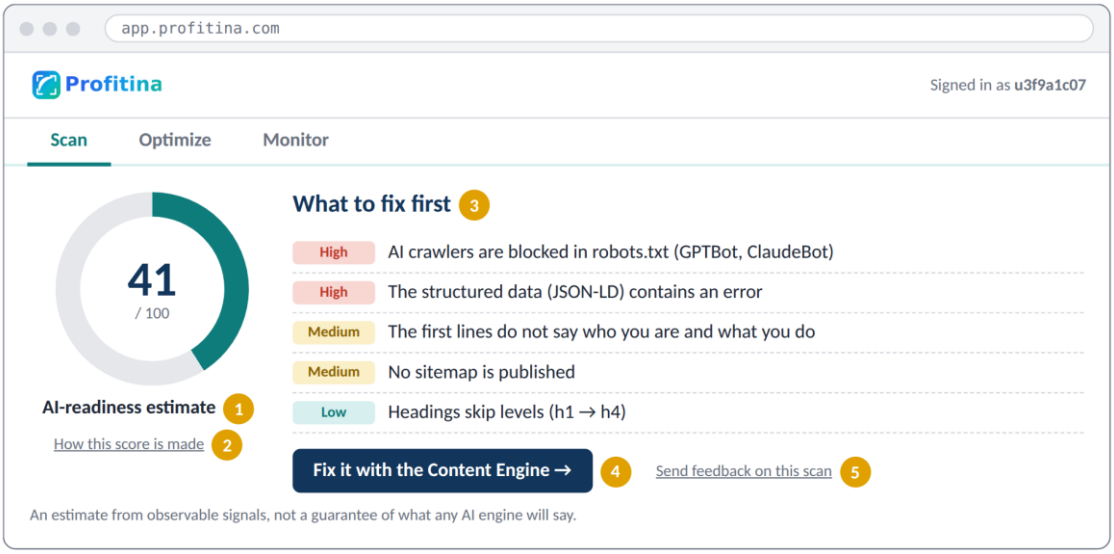


Figure 3.5 A scan result, as a wireframe with illustrative values. (1) The readiness estimate; (2) the link to the method; (3) the prioritized fixes; (4) the way into the Content Engine; (5) feedback on the scan.

The estimate is low (marker 1), and a link explains how it was made (marker 2). The list of fixes (marker 3) is what she will actually use. At the top, with high priority: two AI crawlers are blocked by her `robots.txt`, and her structured data contains an error. With medium priority: her page does not say who she is in its first lines, and she has no sitemap. At the bottom: her headings skip from `h1` to `h4`. Every line is a sentence she can understand. Two links lead onward: into the Content Engine (marker 4), and to feedback about the scan itself (marker 5).

Section 3.10's lab reproduces this experience on a copy of Sari's old site, and its checker finds the same gaps, by our own rules.

3.9.5 Step 5: Optimize

The *Optimize* tab (*Optimasi*) turns the list into material (**Figure 3.6**).

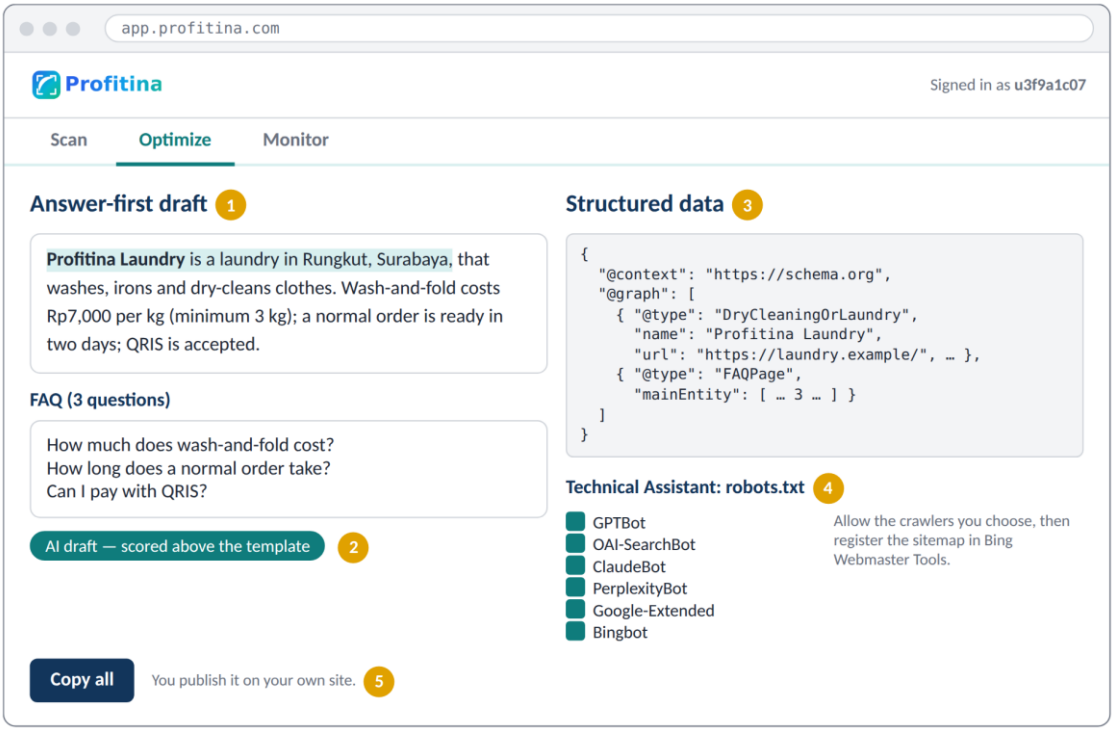


Figure 3.6 The Optimize tab, as a wireframe. (1) An answer-first draft with the entity named at once; (2) the quality gate's verdict; (3) the matching JSON-LD; (4) the Technical Assistant's crawler choices; (5) the owner publishes on her own site.

The draft (marker 1) opens with the sentence Section 3.5.1 praised: *Profitina Laundry is a laundry in Rungkut, Surabaya, that washes, irons and dry-cleans clothes.* A label says that this AI draft passed the quality gate (marker 2). Beside it are the JSON-LD for her business and her FAQ (marker 3), and the Technical Assistant's list of crawlers, which she leaves all ticked (marker 4). She corrects one price, copies everything, and asks the young man who built her website to paste the draft into the home page, replace the broken JSON-LD, and replace `robots.txt`. The platform does not touch her site; she publishes (marker 5). Then she registers her sitemap in Bing Webmaster Tools, following the assistant's guidance.

3.9.6 Step 6: Monitor

In the *Monitor* tab (*Pantau*), Sari turns on the one weekly monitor her Free plan includes (**Figure 3.7**).

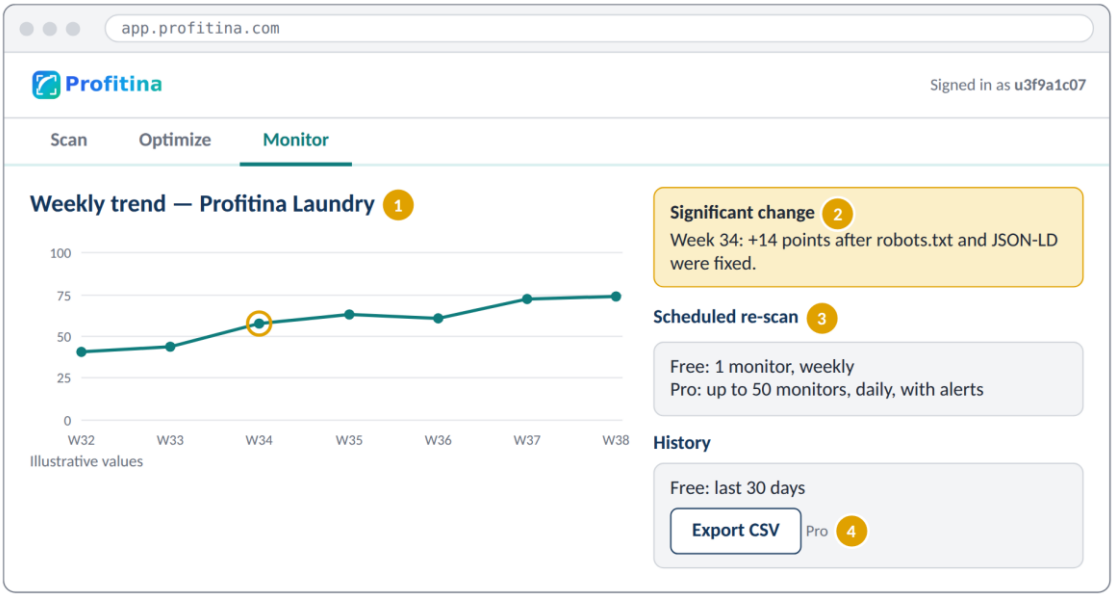


Figure 3.7 The Monitor tab, as a wireframe with illustrative values. (1) The weekly trend; (2) an alert for a significant change; (3) the re-scan cadence of each plan; (4) CSV export, a Pro feature.

Over the following weeks the trend line climbs (marker 1). The week in which the fixes went live produces an alert (marker 2), the rise of +14 points that Equation (3.1) computed in Section 3.6.2. The Free plan keeps 30 days of history and re-scans weekly; Pro would re-scan daily, allow up to 50 monitors, and export the history as CSV (markers 3 and 4) [2].

3.9.7 Step 7: Participate

Sari posts in the community about what she learned ("my hosting panel was blocking GPTBot!") and follows a channel about small-business websites. Her post earns a small amount of PROFIT, the platform's utility coin, because it was judged useful; rewards are small by design, are paid only to email-verified people, and have daily caps that reset at 00:00 WIB [2]. Chapters 4, 5 and 7 describe the community, its safety measures and its reward loop; Chapter 12 describes PROFIT itself.

3.9.8 Step 8: Upgrade, if she wants to

Sari may stay on the Free plan for as long as she likes. If she wants daily monitoring, more monitors, full history and CSV export, she can upgrade to **Pro**, which costs **Rp50,000 for 30 days**, paid by QRIS from any Indonesian e-wallet or mobile bank, or by redeeming PROFIT; it can renew automatically from her balance, with a reminder before it expires [2]. **Table 3.3** compares the two plans as the application's plan cards describe them in September 2026. There are exactly two plans and no advertising, permanently; an earlier third tier, Agency, was removed in August 2026 for simplicity [2]. Chapter 8 tells the story of those decisions.

Table 3.3 The two plans as described by the application's plan cards in September 2026.

	Free	Pro
Scans and auto-fill	15 per day	unlimited scans and content
Content Engine, schema, Technical Assistant	yes	yes
Scheduled monitoring	1 monitor, weekly	50 daily monitors, with alerts
History	30 days	full history and trends, CSV export
Term and price	free	30 days, Rp50,000 by QRIS, or redeemed with PROFIT

3.9.9 Step 9: Exercise her rights

The last step is the one most products hide. In the *Monitor* tab, a panel called **Akun & Data Pribadi** (*Account & Personal Data*) gives Sari every right Indonesia's Personal Data Protection Law, UU PDP No. 27/2022, grants her, as buttons she can press herself (**Figure 3.8**): export all her data as JSON, change her email or password, see which version of the policy she consented to and when, and delete her account permanently [2], [17].

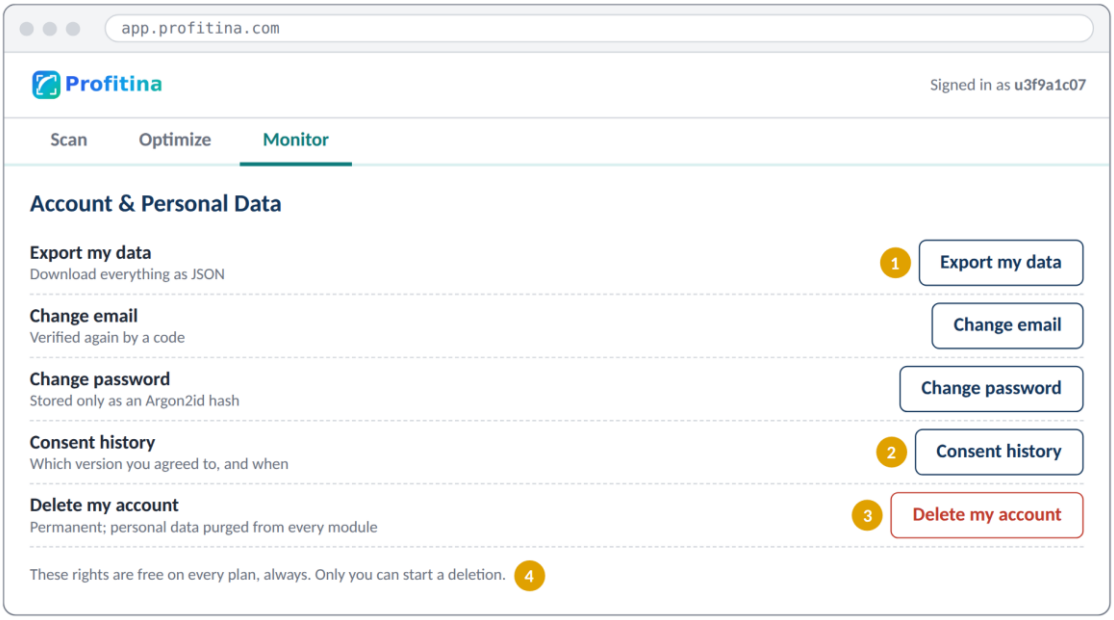


Figure 3.8 The Account & Personal Data panel, as a wireframe. (1) Export as JSON; (2) the consent history; (3) permanent deletion, which only the user can start; (4) these rights are free on every plan.

Deletion purges her personal data from every module and leaves only an anonymous placeholder where shared content must keep its shape; only she can start it, and an administrator never deletes "on her behalf". These rights, together with safety and account recovery, stay **free forever**, on every plan [2]. The product's own history contains an honest correction worth remembering: the deletion mechanism existed before 19 August 2026, but there was no button for it in the interface, so "self-service" was not yet true. It became true the day the panel shipped. Chapter 11 is devoted to privacy by design.

3.9.10 The same journey for a researcher

A lecturer's journey has the same shape with different nouns. She enters her name as it appears on her publications, and the Researcher Visibility Report looks her up in OpenAlex (**Figure 3.9**).

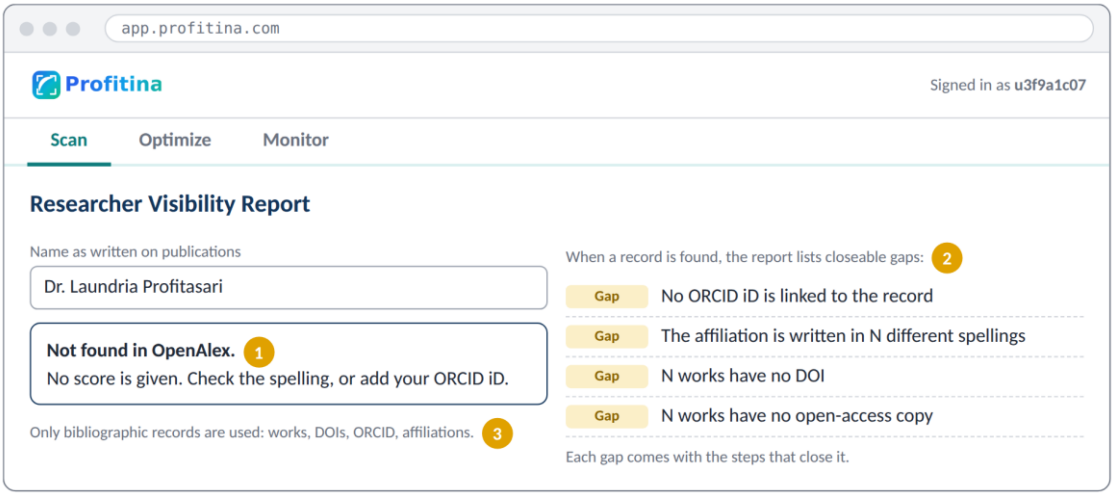


Figure 3.9 The Researcher Visibility Report, as a wireframe. (1) A fictional name is not found, and no score is given; (2) for a real record, the report lists closeable gaps; (3) only bibliographic records are used.

The wireframe shows the case that proves the method honest: the name *Dr. Laundria Profitasari*, invented for this book, is **not found**, and the report gives no score (marker 1). For a real record, the report lists gaps with steps to close them (marker 2): link an ORCID iD, unify the affiliation, add DOIs, deposit open copies. It uses only bibliographic records (marker 3).

3.9.11 One Text, Many Platforms, from the inside

Finally, **Figure 3.10** shows the distribution tool Sari tried in step 2, now from inside her account.

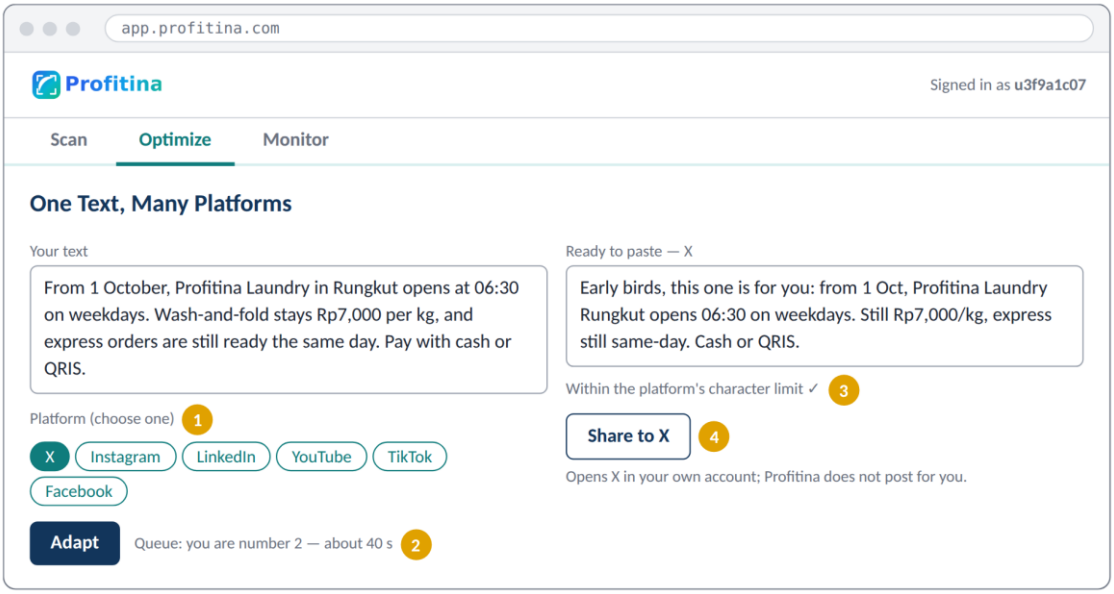


Figure 3.10 One Text, Many Platforms, as a wireframe. (1) One platform is chosen at a time; (2) the queue position and wait are shown; (3) the length is checked against the platform's limit; (4) Share to X opens X in the user's own account.

She chooses X (marker 1), waits behind one other request (marker 2), gets a version that fits within X's limit (marker 3), and presses *Share to X*, which opens X in her own account with the text ready to post (marker 4).

3.10 Lab G1: A Mini AI-Readiness Checker

3.10.1 What we will build, and whose score it is

Sari's scan in Section 3.9.4 found five gaps. In this lab we write a small program that finds gaps of the same kinds, by rules that are entirely ours and entirely visible. It is the flagship exercise of this book, because it touches nearly everything the book teaches: HTTP, the robots exclusion standard, HTML parsing, JSON, object-oriented design, a little arithmetic and a great deal of honesty about what a number means.

The program is given the address of a web page. It fetches at most four things from the page's site: `/robots.txt`, the page itself, `/llms.txt`, and, only when `robots.txt` names no sitemap, `/sitemap.xml`. It collects facts from them and runs eight checks. Each check returns three things: *points* between 0 and 1, the *evidence* it saw, and the *advice* that would close the gap. The report prints all of that and a **teaching score** from 0 to 100. **Figure 3.11** shows the parts and how data flows between them.

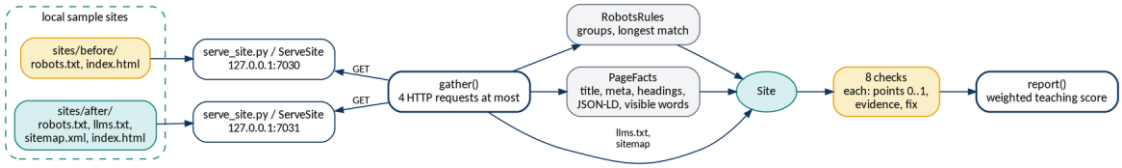


Figure 3.11 The architecture of lab G1. A tiny server publishes a sample site on 127.0.0.1; `gather()` fetches at most four resources, `RobotsRules` and `PageFacts` turn them into a `Site`, eight checks score it, and `report()` prints the result.

Table 3.4 lists the eight checks with the rule each one applies. Every threshold in it is a *teaching choice*, made to be easy to understand and easy to change; none of them is a recommendation from a standards body, and none of them is Profitina's.

Table 3.4 The eight checks of lab G1 and their rules. The thresholds are teaching choices of this book.

#	Check	Rule (points between 0 and 1)	Evidence it reports
1	`crawlers`	share of six crawlers allowed to fetch `/` by `robots.txt`	the decision and the deciding rule for each crawler
2	`sitemap`	1 if `robots.txt` names a sitemap or `/sitemap.xml` exists	the sitemap line or the status of `/sitemap.xml`
3	`llms.txt`	1 if `/llms.txt` exists and starts with `# Title`; 0.5 if it exists without	the status and the title line
4	`json-ld`	1 if every block parses and some node has `name` and `url`; 0.5 if some parse	valid and invalid blocks, the `@type`s found
5	`headings`	half for exactly one `h1`, half for no skipped level	counts of `h1`, headings and level jumps
6	`text`	half for at least 120 visible words, half for a text/HTML ratio of at least 0.20	the word count and the ratio
7	`answer-1st`	1 if the business name is among the first 60 words of the main content	the name used and where it came from
8	`meta`	share of five basics present: a title of 15+ characters, a meta description, `lang`, a canonical link, `dateModified`	the title length and the count

With the points $p_1, \ldots, p_8$ of the eight checks and a weight $w_i \geq 0$ for each, the teaching score is the weighted mean, scaled to 100 and rounded to the nearest whole number:

$$S = \left\lfloor 100 \cdot \frac{\sum_{i=1}^{8} w_i \, p_i}{\sum_{i=1}^{8} w_i} + \frac{1}{2} \right\rfloor \tag{3.2}$$

By default every $w_i = 1$, so S is simply the average of the points, times 100. The floor of "value plus one half" rounds halves upward, and it is written out rather than left to a library function for a reason that Section 3.10.7 explains: Python's `round` and Java's `Math.round` do not agree on halves, and the two versions of the lab must print the same score. The reader can change any weight on the command line, including setting it to 0 to ignore a check, and the report always prints the weights it used.

> **WARNING.** The teaching score of Equation (3.2) is a didactic device. It shows how *any* readiness score is built, from observable facts, rules, weights and aggregation, so that you can reason about one. It is not Profitina's score, it is not calibrated against real citations, and a page that scores 100 by these eight rules can still be ignored by every answer engine.

3.10.2 The sample sites

The lab never scans a third-party site. Scanning other people's servers at volume is impolite, and running a book's exercises against the live web would make its outputs unrepeatable. Instead, the chapter's code folder contains **local sample sites** that a tiny server publishes on your own machine, on the address 127.0.0.1, which never leaves your computer. The ports are 7030 to 7039, the range this book reserves for Chapter 3.

The folder `sites/before` is a copy of Profitina Laundry's website as Sari found it in step 4. Its `robots.txt` was written by a hosting control panel (**Listing 3.1**).

Listing 3.1 `sites/before/robots.txt`: a panel-managed file that blocks two AI crawlers and one other crawler by name, and allows everyone else.

```
# Managed by the hosting control panel
User-agent: GPTBot
Disallow: /

User-agent: ClaudeBot
Disallow: /

User-agent: CCBot
Disallow: /

User-agent: *
Allow: /
```

Read it with RFC 9309's rules from Chapter 2 [7]. `GPTBot` and `ClaudeBot` each have a group of their own, and in each the only rule is `Disallow: /`, which matches every path. Every crawler *not* named falls to the `*` group, whose `Allow: /` admits it. So `OAI-SearchBot`, `PerplexityBot` and `Bingbot` may read the site, while `GPTBot` and `ClaudeBot` may not. This is exactly the kind of file that surprised the team behind profitina.com (Section 3.2.2): it looks permissive at a glance, because it ends with `Allow: /`.

The home page, **Listing 3.2**, is a typical small-business page of the slideshow era.

Listing 3.2 `sites/before/index.html`: the home page before the fixes. The title is "Home", there are two `h1` headings, the JSON-LD has a trailing comma, and the business name appears only in the footer.

```
<!DOCTYPE html>
<html lang="en">
<head>
<meta charset="utf-8">
<title>Home</title>
<meta name="description" content="Clean clothes, happy life.">
<style>
  body { font-family: sans-serif; margin: 0; background: #fafafa; }
  .hero { background: #0f7c7c; color: white; padding: 48px; }
  .slider img { width: 100%; height: 420px; object-fit: cover; }
  .card { display: inline-block; width: 30%; margin: 1%; }
</style>
<script type="application/ld+json">
{
  "@context": "https://schema.org",
  "@type": "LocalBusiness",
  "name": "Profitina Laundry",
  "telephone": "+62-812-0000-1234",
}
</script>
<script>
  // image slider and visitor counter
  var slides = ["img/slide1.jpg", "img/slide2.jpg", "img/slide3.jpg"];
  var current = 0;
  function nextSlide() {
    current = (current + 1) % slides.length;
    document.getElementById("slide").src = slides[current];
  }
  setInterval(nextSlide, 4000);
  window.addEventListener("load", function () {
    fetch("/counter").then(function (r) { return r.text(); })
      .then(function (t) { document.getElementById("count").textContent = t; });
  });
</script>
</head>
<body>
<nav><a href="/">Home</a> | <a href="/services">Services</a>
 | <a href="/outlets">Outlets</a> | <a href="/status">Order status</a></nav>
<div class="hero">
  <img src="img/logo.png" alt="logo">
  <h1>Welcome!</h1>
  <p>Clean clothes, happy life.</p>
</div>
<div class="slider"><img id="slide" src="img/slide1.jpg" alt=""></div>
<h1>Our Services</h1>
<h4>Fast</h4>
```

Listing 3.2 (Continued) *sites/before/index.html: the home page before the fixes. The title is "Home", there are two h1 headings, the JSON-LD has a trailing comma, and the business name appears only in the footer.*

```
<p>We are the best and the fastest. Your satisfaction is our priority.</p>
<h4>Clean</h4>
<p>Premium quality at an affordable price. Contact us today!</p>
<h4>Friendly</h4>
<p>Our friendly team is always ready to serve you.</p>
<footer>
  <p>Visitors: <span id="count">0</span></p>
  <p>Copyright 2026 Profitina Laundry. All rights reserved.</p>
</footer>
</body>
</html>
```

Five problems hide in it. The `<title>` is “Home”, which describes every home page on the web and therefore none. The JSON-LD block ends its last property with a comma, `"telephone": "+62-812-0000-1234",` followed by `}`; JSON does not allow that, so every strict parser rejects the whole

block and the structured data might as well not exist. There are two `h1` headings, "Welcome!" and "Our Services", and the second is followed directly by `h4` headings, skipping two levels. Most of the bytes are a stylesheet and a script for the slider and the visitor counter. And the name of the business appears nowhere in the visible text except in the copyright line of the footer.

The folder `sites/after` holds the same site after Sari's step 5. Its `robots.txt` (**Listing 3.3**) admits every crawler to the public pages, keeps two private areas out, blocks one crawler by choice, and names the sitemap.

Listing 3.3 `sites/after/robots.txt`: every crawler may read the public pages; `/admin/` and `/api/` are kept out, `CCBot` is blocked by choice, and the sitemap is named.

```
# robots.txt for Profitina Laundry
User-agent: *
Allow: /
Disallow: /admin/
Disallow: /api/

User-agent: CCBot
Disallow: /

Sitemap: https://laundry.example/sitemap.xml
```

It also has an `llms.txt` (**Listing 3.4**) in the format of the proposal [13]: an H1 title, a short summary in a block quote, and a section of links with notes.

Listing 3.4 `sites/after/llms.txt`: an H1 title, a summary, and a list of the site's main pages.

```
# Profitina Laundry

> Wash-and-fold, ironing and dry cleaning in Surabaya, Indonesia.
> Rp7,000 per kilogram; normal orders take two days; QRIS accepted.

## Pages

- [Services and prices](https://laundry.example/services): prices of every service
- [Outlets](https://laundry.example/outlets): addresses and opening hours
- [Order status](https://laundry.example/status): find an order by bill number
```

And its home page (**Listing 3.5**) now opens with the answer-first paragraph, carries one `h1` with `h2` sections and `h3` subsections under them, declares `lang="en"`, a meta description and a canonical address, and describes the business, the page and its FAQ in one valid JSON-LD `@graph` using the schema.org type `DryCleaningOrLaundry`, a kind of `LocalBusiness` [11].

Listing 3.5 `sites/after/index.html`: the home page after the fixes, with an answer-first opening, a clean heading outline and one valid JSON-LD graph.

```
<!DOCTYPE html>
<html lang="en">
<head>
<meta charset="utf-8">
<title>Profitina Laundry | Wash-and-fold and dry cleaning in Surabaya</title>
<meta name="description" content="Profitina Laundry washes, irons and
  dry-cleans in Surabaya: Rp7,000 per kg, ready in two days, QRIS accepted.">
<link rel="canonical" href="https://laundry.example/">
<link rel="stylesheet" href="/style.css">
<script type="application/ld+json">
{
  "@context": "https://schema.org",
  "@graph": [
    {
      "@type": "DryCleaningOrLaundry",
      "@id": "https://laundry.example/#shop",
      "name": "Profitina Laundry",
      "url": "https://laundry.example/",
      "telephone": "+62-812-0000-1234",
      "priceRange": "Rp7,000 - Rp75,000",
      "address": {
        "@type": "PostalAddress",
        "streetAddress": "Jl. Penjaringan Sari No. 65, Rungkut",
        "addressLocality": "Surabaya",
        "addressCountry": "ID"
      }
    },
    {
      "@type": "WebPage",
      "url": "https://laundry.example/",
      "about": {"@id": "https://laundry.example/#shop"},
      "dateModified": "2026-09-20"
    },
    {
      "@type": "FAQPage",
      "mainEntity": [
        {"@type": "Question", "name": "How much does wash-and-fold cost?",
         "acceptedAnswer": {"@type": "Answer",
           "text": "Rp7,000 per kilogram, with a minimum of 3 kg per order."}},
        {"@type": "Question", "name": "How long does a normal order take?",
         "acceptedAnswer": {"@type": "Answer",
           "text": "Two days. Express orders are ready the same day."}},
        {"@type": "Question", "name": "Can I pay with QRIS?",
         "acceptedAnswer": {"@type": "Answer",
           "text": "Yes. Every outlet accepts cash and QRIS."}}
      ]
```

Listing 3.5 (Continued) *sites/after/index.html: the home page after the fixes, with an answer-first opening, a clean heading outline and one valid JSON-LD graph.*

```
    }
  ]
}
</script>
</head>
<body>
<nav><a href="/">Home</a> | <a href="/services">Services</a>
 | <a href="/outlets">Outlets</a> | <a href="/status">Order status</a></nav>
<main>
<h1>Profitina Laundry: wash-and-fold and dry cleaning in Surabaya</h1>
<p>Profitina Laundry is a laundry in Rungkut, Surabaya, that washes, irons
and dry-cleans clothes for households and small businesses. Wash-and-fold
costs Rp7,000 per kilogram with a minimum of 3 kg, a normal order is ready
in two days, and express orders are ready the same day. Every outlet
accepts cash and QRIS.</p>
<h2>Services and prices</h2>
<h3>Wash-and-fold</h3>
<p>Rp7,000 per kilogram, minimum 3 kg. Clothes are washed, dried and folded,
and sorted by color before washing.</p>
<h3>Ironing</h3>
<p>Rp5,000 per kilogram, or Rp10,000 per kilogram together with washing.</p>
<h3>Dry cleaning</h3>
<p>From Rp25,000 for a shirt to Rp75,000 for a wedding dress or a suit.</p>
<h2>Frequently asked questions</h2>
<h3>How much does wash-and-fold cost?</h3>
<p>Rp7,000 per kilogram, with a minimum of 3 kg per order.</p>
<h3>How long does a normal order take?</h3>
<p>Two days. Express orders are ready the same day.</p>
<h3>Can I pay with QRIS?</h3>
<p>Yes. Every outlet accepts cash and QRIS.</p>
<h2>Opening hours</h2>
<p>Monday to Saturday 07:00 to 21:00, Sunday 08:00 to 17:00 (WIB).
Last updated on 20 September 2026.</p>
</main>
<footer><p>Profitina Laundry, Jl. Penjaringan Sari No. 65, Rungkut,
Surabaya. Telephone +62-812-0000-1234.</p></footer>
</body>
</html>
```

Note the address inside the files: `https://laundry.example/`. The domain `.example` is reserved for documentation and examples and can never belong to a real business, which makes it safe to print in a book. The local server stands in for that public address: it serves the same files from 127.0.0.1.

3.10.3 A tiny local server

Both standard libraries contain a small HTTP server, enough to publish a folder of files. **Listing 3.6** uses Python's `http.server` [18]. The class `SimpleHTTPRequestHandler` already knows how to answer `GET /robots.txt` with the file `robots.txt`, and `GET /` with `index.html`; `functools.partial` fixes its `directory` argument to our folder. We add three small things: explicit content types with `charset=utf-8`, a one-line log per request so that you can *see* every request the checker makes, and a refusal to use any port outside 7030 to 7039.

Listing 3.6 `serve_site.py`: a static file server for one sample site, with one log line per request.

```
# serve_site.py - serve one sample site folder on 127.0.0.1 (Chapter 3 lab)
import functools
import sys
from http.server import SimpleHTTPRequestHandler, ThreadingHTTPServer


class QuietHandler(SimpleHTTPRequestHandler):
    """Serves files from one folder and logs one short line per request."""
    extensions_map = {".html": "text/html; charset=utf-8",
                      ".txt": "text/plain; charset=utf-8",
                      ".xml": "application/xml", "": "application/octet-stream"}
    verbose = True

    def log_request(self, code="-", size="-"):
        if self.verbose:
            status = getattr(code, "value", code)     # HTTPStatus or text
            print(f"  {self.command} {self.path} -> {status}", flush=True)

    def log_message(self, fmt, *args):
        pass                                    # errors are shown by log_request


def make_server(folder, port, verbose=True):
    handler = functools.partial(QuietHandler, directory=folder)
    QuietHandler.verbose = verbose
    return ThreadingHTTPServer(("127.0.0.1", port), handler)


def main():
    if len(sys.argv) != 3:
        print("usage: python serve_site.py <folder> <port 7030-7039>")
        sys.exit(1)
    folder, port = sys.argv[1], int(sys.argv[2])
    if not 7030 <= port <= 7039:
        print("Chapter 3 uses ports 7030 to 7039 only.")
        sys.exit(1)
    server = make_server(folder, port)
    print(f"Serving {folder} at http://127.0.0.1:{port}/ (Ctrl+C stops)")
    try:
        server.serve_forever()
    except KeyboardInterrupt:
        print("Stopped.")
    finally:
        server.server_close()
```

Listing 3.6 (Continued) *serve_site.py: a static file server for one sample site, with one log line per request.*

```
if __name__ == "__main__":
    main()
```

Java's standard library has had a small server since Java 6, in the package `com.sun.net.httpserver` [19]. It is lower-level than Python's: we map the request path to a file ourselves (**Listing 3.7**). Two lines deserve attention. `root.resolve(path.substring(1)).normalize()` turns the requested path into a file path, and `file.startsWith(root)` then refuses anything that escaped the folder, such as a request for `/../../secret.txt`. A server that publishes files must *always* make that check; forgetting it is one of the oldest security bugs on the web, called *path traversal*. Python's `SimpleHTTPRequestHandler` makes the same check internally.

Listing 3.7 `ServeSite.java`: the same server with `com.sun.net.httpserver`, including the path-traversal guard.

```java
// ServeSite.java - serve one sample site folder on 127.0.0.1 (Chapter 3 lab)
import com.sun.net.httpserver.HttpServer;
import java.io.IOException;
import java.io.OutputStream;
import java.net.InetSocketAddress;
import java.nio.charset.StandardCharsets;
import java.nio.file.Files;
import java.nio.file.Path;

public class ServeSite {
    static String type(Path file) {
        String name = file.getFileName().toString();
        if (name.endsWith(".html")) {
            return "text/html; charset=utf-8";
        } else if (name.endsWith(".txt")) {
            return "text/plain; charset=utf-8";
        } else if (name.endsWith(".xml")) {
            return "application/xml";
        }
        return "application/octet-stream";
    }

    static HttpServer makeServer(String folder, int port, boolean verbose)
            throws IOException {
        Path root = Path.of(folder).toAbsolutePath().normalize();
        HttpServer server =
                HttpServer.create(new InetSocketAddress("127.0.0.1", port), 0);
        server.createContext("/", exchange -> {
            String asked = exchange.getRequestURI().getPath();
            String path = asked.endsWith("/") ? asked + "index.html" : asked;
            Path file = root.resolve(path.substring(1)).normalize();
            int status = 404;
            byte[] body = "404 Not Found\n".getBytes(StandardCharsets.UTF_8);
            String type = "text/plain; charset=utf-8";
            if (file.startsWith(root) && Files.isRegularFile(file)) {
                status = 200;                   // never serve outside the folder
                body = Files.readAllBytes(file);
                type = type(file);
            }
            exchange.getResponseHeaders().set("Content-Type", type);
            exchange.sendResponseHeaders(status, body.length);
            try (OutputStream out = exchange.getResponseBody()) {
                out.write(body);
            }
            if (verbose) {
                System.out.println("  " + exchange.getRequestMethod() + " "
```

Listing 3.7 (Continued) *ServeSite.java: the same server with com.sun.net.httpserver, including the path-traversal guard.*

```
                        + asked + " -> " + status);
            }
        });
        return server;
    }

    public static void main(String[] args) throws IOException {
        if (args.length != 2) {
            System.out.println("usage: java ServeSite <folder> <port 7030-7039>");
            System.exit(1);
        }
        int port = Integer.parseInt(args[1]);
        if (port < 7030 || port > 7039) {
            System.out.println("Chapter 3 uses ports 7030 to 7039 only.");
            System.exit(1);
        }
        HttpServer server = makeServer(args[0], port, true);
        server.start();
        System.out.println("Serving " + args[0] + " at http://127.0.0.1:"
                + port + "/ (Ctrl+C stops)");
    }
}
```

3.10.4 Reading robots.txt by the rules

The checker needs `robots.txt` twice: once to decide whether *it* may fetch the page, and once to report what six AI and search crawlers may do. Chapter 2 described the procedure of RFC 9309; **Pseudocode 3.1** writes it precisely, extended with what a crawler must do when the file cannot be read.

Pseudocode 3.1 Deciding whether a crawler may fetch a path under RFC 9309: status codes first, then group choice, then the longest matching rule, with ties going to `Allow`.

```
DECIDE(robots, agent, path)
 1  if robots.status = 0 or robots.status ≥ 500     // no answer, or server error
 2      return (disallow, "unreachable: assume disallow")
 3  if robots.status is not a 2xx code               // e.g. 404: no file
 4      return (allow, "no rules apply")
 5  G ← the groups whose User-agent equals agent, ignoring case
 6  if G is empty
 7      G ← the groups whose User-agent is "*"
 8  best ← NIL
 9  for each rule r in the groups of G
10      if r.path is a prefix of path
11          if best = NIL or |r.path| > |best.path|
12                 or (|r.path| = |best.path| and r is Allow)
13              best ← r
14  if best = NIL
15      return (allow, "no rule matches")
16  return (best.kind, best)
```

Lines 1 to 4 handle the file's HTTP status, and they encode a rule of RFC 9309 that is easy to get backward [7]. If the server says the file *does not exist* (a 4xx status such as 404), the crawler may fetch anything: silence means consent. If the server *fails* (a 5xx status) or does not answer at all, the crawler must assume complete disallow: a site in trouble might have a `robots.txt` that forbids exactly what the crawler wants, and the safe assumption is that it does. Lines 5 to 7 choose the groups: the groups that name the crawler, all of them merged, or, only when there are none, the `*` groups. Lines 8 to 13 find the most specific matching rule. Written as a formula, if M is

the set of rules in the chosen groups whose path is a prefix of the requested path q, the deciding rule is

$$\hat{r} = \operatorname*{argmax}_{r \in M} \ell(r) \tag{3.3}$$

where $\ell(r)$ is the length of the rule's path in characters, and an `Allow` rule wins a tie. If M is empty, the path is allowed.

A word on cost. If the chosen groups hold R rules and the longest rule path has L characters, one decision tests R prefixes of at most L characters each, so it takes time proportional to $R \cdot L$ at worst. A real `robots.txt` has tens of rules, so a decision is instantaneous; RFC 9309 asks crawlers to parse at least the first 500 kibibytes of a file, which bounds R in practice [7].

Listing 3.8 implements the procedure in Python. The class keeps the groups as pairs of lists, so that a group with two `User-agent` lines followed by rules is one group with two agents. The flag `last_was_agent` is how the parser knows that a new `User-agent` line starts a new group: only when the line before it was *not* also a `User-agent` line. `decide` returns a pair, the decision and a *reason*, because a checker that cannot explain itself is of little use.

Listing 3.8 `robots_rules.py`: parsing `robots.txt` into groups and deciding by status code, group choice and longest match.

```
# robots_rules.py - decide what robots.txt allows, as RFC 9309 describes it
# (plain path prefixes only; the * and $ patterns are left as an exercise)


class RobotsRules:
    def __init__(self, text, status):
        self.status = status        # HTTP status of /robots.txt, 0 = no answer
        self.groups = []            # list of (agents, rules)
        self.sitemaps = []          # values of Sitemap: lines
        if 200 <= status < 300:
            self._parse(text)

    def _parse(self, text):
        agents, rules, last_was_agent = None, None, False
        for raw in text.splitlines():
            line = raw.split("#", 1)[0].strip()          # drop comments
            key, colon, value = line.partition(":")
            if not colon:
                continue
            key, value = key.strip().lower(), value.strip()
            if key == "user-agent":
                if agents is None or not last_was_agent:  # a new group begins
                    agents, rules = [], []
                    self.groups.append((agents, rules))
                agents.append(value)
                last_was_agent = True
            elif key in ("allow", "disallow"):
                if rules is not None and value:          # empty Disallow: none
                    rules.append((key == "allow", value))
                last_was_agent = False
            elif key == "sitemap":
                self.sitemaps.append(value)
            else:
                last_was_agent = False

    def _groups_for(self, agent):
        named = [(a, r) for (a, r) in self.groups
                 if agent.lower() in (x.lower() for x in a)]
        if named:
            return agent, named
        return "*", [(a, r) for (a, r) in self.groups if "*" in a]
```

Listing 3.8 (Continued) *robots_rules.py: parsing robots.txt into groups and deciding by status code, group choice and longest match.*

```
    def decide(self, agent, path):
        """Return (allowed, reason) for one crawler and one path."""
        if self.status == 0 or self.status >= 500:
            return False, "robots.txt unreachable: assume disallow"
        if self.status >= 300:
            return True, f"robots.txt {self.status}: no rules apply"
        name, groups = self._groups_for(agent)
        best = None                                        # (allow, path)
        for _, rules in groups:
            for allow, prefix in rules:
                if not path.startswith(prefix):
                    continue
                if (best is None or len(prefix) > len(best[1])
                        or (len(prefix) == len(best[1]) and allow)):
                    best = (allow, prefix)
        if not groups:
            return True, "no group applies"
        if best is None:
            return True, f'no rule matches in group "{name}"'
        word = "Allow" if best[0] else "Disallow"
        return best[0], f'{word}: {best[1]} in group "{name}"'
```

Why not use Python's own `urllib.robotparser`, which Chapter 2 used? Because, as Chapter 2 showed, it applies the first matching rule in file order rather than the longest one, and it does not merge several groups that name the same crawler; a request to bring it in line with RFC 9309 is open in the CPython project [20]. For a checker whose whole purpose is to report what a standards-following crawler will do, we want the standard's rules exactly, and we want the Python and Java versions to agree. Exercise 3.3 compares the two on a file where they disagree.

The Java class, **Listing 3.9**, is a line-by-line translation. Records, available since Java 16, give us `Rule`, `Group` and `Decision` in one line each.

Listing 3.9 `RobotsRules.java`: the same parser and decision procedure in Java, with records for rules, groups and decisions.

```java
// RobotsRules.java - decide what robots.txt allows, as RFC 9309 describes it
// (plain path prefixes only; the * and $ patterns are left as an exercise)
import java.util.ArrayList;
import java.util.List;

public class RobotsRules {
    record Rule(boolean allow, String path) { }

    record Group(List<String> agents, List<Rule> rules) { }

    record Decision(boolean allowed, String reason) { }

    final int status;               // HTTP status of /robots.txt, 0 = no answer
    final List<Group> groups = new ArrayList<>();
    final List<String> sitemaps = new ArrayList<>();

    RobotsRules(String text, int status) {
        this.status = status;
        if (status >= 200 && status < 300) {
            parse(text);
        }
    }

    private void parse(String text) {
        Group current = null;
        boolean lastWasAgent = false;
        for (String raw : text.split("\r?\n")) {
            String line = raw.split("#", 2)[0].strip();   // drop comments
            int colon = line.indexOf(':');
            if (colon < 0) {
                continue;
            }
            String key = line.substring(0, colon).strip().toLowerCase();
            String value = line.substring(colon + 1).strip();
            if (key.equals("user-agent")) {
                if (current == null || !lastWasAgent) {     // a new group begins
                    current = new Group(new ArrayList<>(), new ArrayList<>());
                    groups.add(current);
                }
                current.agents().add(value);
                lastWasAgent = true;
            } else if (key.equals("allow") || key.equals("disallow")) {
                if (current != null && !value.isEmpty()) { // empty Disallow: none
                    current.rules().add(new Rule(key.equals("allow"), value));
                }
                lastWasAgent = false;
```

Listing 3.9 (Continued) *RobotsRules.java: the same parser and decision procedure in Java, with records for rules, groups and decisions.*

```
            } else if (key.equals("sitemap")) {
                sitemaps.add(value);
            } else {
                lastWasAgent = false;
            }
        }
    }

    private List<Group> groupsFor(String agent) {
        List<Group> named = new ArrayList<>();
        List<Group> star = new ArrayList<>();
        for (Group g : groups) {
            if (g.agents().stream().anyMatch(a -> a.equalsIgnoreCase(agent))) {
                named.add(g);
            } else if (g.agents().contains("*")) {
                star.add(g);
            }
        }
        return named.isEmpty() ? star : named;
    }

    Decision decide(String agent, String path) {
        if (status == 0 || status >= 500) {
            return new Decision(false, "robots.txt unreachable: assume disallow");
        }
        if (status >= 300) {
            return new Decision(true,
                    "robots.txt " + status + ": no rules apply");
        }
        List<Group> chosen = groupsFor(agent);
        String name = chosen.stream().anyMatch(g -> g.agents().stream()
                .anyMatch(a -> a.equalsIgnoreCase(agent))) ? agent : "*";
        Rule best = null;
        for (Group g : chosen) {
            for (Rule r : g.rules()) {
                if (!path.startsWith(r.path())) {
                    continue;
                }
                int len = r.path().length();
                if (best == null || len > best.path().length()
                        || (len == best.path().length() && r.allow())) {
                    best = r;
                }
            }
        }
        if (chosen.isEmpty()) {
```

Listing 3.9 (Continued) *RobotsRules.java: the same parser and decision procedure in Java, with records for rules, groups and decisions.*

```
            return new Decision(true, "no group applies");
        }
        if (best == null) {
            return new Decision(true,
                    "no rule matches in group \"" + name + "\"");
        }
        String word = best.allow() ? "Allow" : "Disallow";
        return new Decision(best.allow(), word + ": " + best.path()
                + " in group \"" + name + "\"");
    }
}
```

3.10.5 Collecting the facts of a page

The second input is the page itself. We need, from its HTML: the title, the meta description, the `lang` attribute, the canonical link, the list of headings with their levels, the text of every JSON-LD block, and the *visible words*, the words a person reading the page would see as content. A full HTML parser builds a tree of the whole document; we need much less. We read the document once, from start to end, and react to three kinds of event: a start tag, an end tag, and a run of text between tags. That style is called *event-based* or *streaming* parsing, and it takes time proportional to the length of the document, because every character is examined a constant number of times.

Which words are "visible"? Obviously not those inside `<script>` and `<style>`, and not those in `<head>`. We also skip `<nav>` and `<footer>`, because navigation links and copyright lines appear on every page of a site and say nothing about this page; skipping them is a simple version of what extraction tools call *boilerplate removal*. With the visible text T, the words joined by single spaces, and the HTML H as bytes, the **text-to-HTML ratio** is

$$\rho = \frac{n_T}{n_H} \tag{3.4}$$

where n_T is the number of characters of T and n_H the number of bytes of H in the UTF-8 encoding. A page that is mostly text has a high ρ; a page that is mostly scripts, styles and markup has a low one. For the sample pages, which contain only characters from the ASCII range, characters and bytes coincide. The `text` check of **Table 3.4** asks for $\rho \geq 0.20$ and at least 120 words, two teaching thresholds.

Python's standard library provides the event-based parser in `html.parser.HTMLParser` [21]. We subclass it and override its three event methods (**Listing 3.10**). Setting `convert_charrefs=True` makes the parser decode entities such as `&` and deliver the text between two tags in one piece.

Listing 3.10 `page_facts.py`: an event-based collector of titles, meta tags, headings, JSON-LD blocks and visible words, built on `html.parser`.

```
# page_facts.py - collect the facts about one HTML page that the checks need
from html.parser import HTMLParser

SKIP = {"head", "script", "style", "noscript", "template", "nav", "footer"}
HEADINGS = {"h1": 1, "h2": 2, "h3": 3, "h4": 4, "h5": 5, "h6": 6}


class PageFacts(HTMLParser):
    def __init__(self, html_text):
        super().__init__(convert_charrefs=True)
        self.html_bytes = len(html_text.encode("utf-8"))
        self.title = ""
        self.description = None
        self.lang = None
        self.canonical = None
        self.headings = []          # (level, text) in document order
        self.jsonld = []            # the text of every JSON-LD block
        self.words = []             # visible words, boilerplate skipped
        self.main_words = []        # visible words inside <main>
        self.has_main = False
        self._skip = 0              # > 0 while inside a SKIP element
        self._in_main = 0
        self._in_title = False
        self._in_jsonld = False
        self._heading = None        # [level, words] while inside h1..h6
        self.feed(html_text)
        self.close()

    def handle_starttag(self, tag, attrs):
        a = {k: (v or "") for k, v in attrs}
        if tag == "html":
            self.lang = a.get("lang") or None
        elif tag == "title":
            self._in_title = True
        elif tag == "meta" and a.get("name", "").lower() == "description":
            self.description = " ".join(a.get("content", "").split())
        elif tag == "link" and "canonical" in a.get("rel", "").lower().split():
            self.canonical = a.get("href")
        elif tag == "script" and a.get("type", "") == "application/ld+json":
            self._in_jsonld = True
            self.jsonld.append("")
        elif tag == "main":
            self.has_main = True
            self._in_main += 1
        elif tag in HEADINGS:
            self._heading = [HEADINGS[tag], []]
```

Listing 3.10 (Continued) *page_facts.py: an event-based collector of titles, meta tags, headings, JSON-LD blocks and visible words, built on html.parser.*

```
        if tag in SKIP:
            self._skip += 1

    def handle_endtag(self, tag):
        if tag in SKIP and self._skip > 0:
            self._skip -= 1
        if tag == "title":
            self._in_title = False
        elif tag == "script":
            self._in_jsonld = False
        elif tag == "main" and self._in_main > 0:
            self._in_main -= 1
        elif tag in HEADINGS and self._heading is not None:
            self.headings.append((self._heading[0], " ".join(self._heading[1])))
            self._heading = None

    def handle_data(self, data):
        if self._in_title:
            self.title += " ".join(data.split())
        if self._in_jsonld:
            self.jsonld[-1] += data
        if self._skip > 0:
            return
        words = data.split()
        self.words.extend(words)
        if self._in_main > 0:
            self.main_words.extend(words)
        if self._heading is not None:
            self._heading[1].extend(words)

    def visible_text(self):
        return " ".join(self.words)

    def lead_words(self, n):
        """The first n words of the main content (the whole body if no <main>)."""
        return (self.main_words if self.has_main else self.words)[:n]
```

The counter `_skip` rather than a simple flag handles nesting: if a `<nav>` sat inside a `<footer>`, leaving the `<nav>` must not make the footer's text visible. The same idea, counting instead of flagging, appears for `<main>`. The method `lead_words(n)` returns the first n visible words of the `<main>` element if the page has one, and of the whole body otherwise; the `answer-1st` check uses it.

Java's standard library has no HTML parser, so **Listing 3.11** contains a small one: a regular expression `TAG` finds the tags, the text between two matches is a run of text, and `ATTR` splits a tag's attributes. Two details keep it faithful to the Python version. Comments and the `<!DOCTYPE>` declaration are removed first. And the content of `<script>` and `<style>` is read raw, up to the matching closing tag, because a script may contain a `<` character that is not a tag; HTML calls these *raw text* elements, and Python's parser treats them the same way.

Listing 3.11 `PageFacts.java`: the same collector in Java, with a small regular-expression tokenizer and raw-text handling for `script` and `style`.

```java
// PageFacts.java - collect the facts about one HTML page that the checks need
import java.nio.charset.StandardCharsets;
import java.util.ArrayList;
import java.util.HashMap;
import java.util.List;
import java.util.Map;
import java.util.Set;
import java.util.regex.Matcher;
import java.util.regex.Pattern;

public class PageFacts {
    static final Set<String> SKIP = Set.of("head", "script", "style",
            "noscript", "template", "nav", "footer");
    static final Pattern TAG = Pattern.compile(
            "<(/?)([a-zA-Z][a-zA-Z0-9]*)((?:[^>\"']|\"[^\"]*\"|'[^']*')*)>");
    static final Pattern ATTR = Pattern.compile("([a-zA-Z_:][-a-zA-Z0-9_:.]*)"
            + "(?:\\s*=\\s*(\"[^\"]*\"|'[^']*'|[^\\s\"'>]+))?");

    record Heading(int level, String text) { }

    final int htmlBytes;
    String title = "";
    String description = null;
    String lang = null;
    String canonical = null;
    final List<Heading> headings = new ArrayList<>();
    final List<String> jsonld = new ArrayList<>();
    final List<String> words = new ArrayList<>();      // boilerplate skipped
    final List<String> mainWords = new ArrayList<>();  // inside <main>
    boolean hasMain = false;
    private int skip = 0;
    private int inMain = 0;
    private boolean inTitle = false;
    private boolean inJsonld = false;
    private int headingLevel = 0;                      // 0 = not in a heading
    private final List<String> headingWords = new ArrayList<>();
```

Listing 3.11 (Continued) *PageFacts.java: the same collector in Java, with a small regular-expression tokenizer and raw-text handling for script and style.*

```
    PageFacts(String html) {
        htmlBytes = html.getBytes(StandardCharsets.UTF_8).length;
        String s = html.replaceAll("(?s)<!--.*?-->", "")   // comments
                       .replaceAll("<![^>]*>", "");        // <!DOCTYPE html>
        String lower = s.toLowerCase();
        Matcher m = TAG.matcher(s);
        int pos = 0;
        while (m.find(pos)) {
            data(s.substring(pos, m.start()));
            String name = m.group(2).toLowerCase();
            pos = m.end();
            if (!m.group(1).isEmpty()) {
                end(name);
                continue;
            }
            start(name, attributes(m.group(3)));
            if (name.equals("script") || name.equals("style")) {
                int close = lower.indexOf("</" + name, pos);  // raw text inside
                close = close < 0 ? s.length() : close;
                data(s.substring(pos, close));
                pos = close;
            }
        }
        data(s.substring(pos));
    }

    static Map<String, String> attributes(String text) {
        Map<String, String> map = new HashMap<>();
        Matcher m = ATTR.matcher(text);
        while (m.find()) {
            String v = m.group(2) == null ? "" : m.group(2);
            if (v.startsWith("\"") || v.startsWith("'")) {
                v = v.substring(1, v.length() - 1);
            }
            map.putIfAbsent(m.group(1).toLowerCase(), decode(v));
        }
        return map;
    }
```

Listing 3.11 (Continued) *PageFacts.java: the same collector in Java, with a small regular-expression tokenizer and raw-text handling for script and style.*

```
    static String decode(String t) {                    // the common entities
        Matcher m = Pattern.compile("&#(\\d+);").matcher(t);
        t = m.replaceAll(r -> Character.toString(Integer.parseInt(r.group(1))));
        return t.replace("&lt;", "<").replace("&gt;", ">").replace("&quot;", "\"")
                .replace("&#39;", "'").replace(" ", "\u00a0")
                .replace("&amp;", "&");
    }

    static List<String> split(String text) {
        List<String> out = new ArrayList<>();
        for (String w : text.strip().split("\\s+")) {
            if (!w.isEmpty()) {
                out.add(w);
            }
        }
        return out;
    }

    private void start(String tag, Map<String, String> a) {
        if (tag.equals("html")) {
            lang = a.getOrDefault("lang", "").isEmpty() ? null : a.get("lang");
        } else if (tag.equals("title")) {
            inTitle = true;
        } else if (tag.equals("meta")
                && a.getOrDefault("name", "").equalsIgnoreCase("description")) {
            description = String.join(" ", split(a.getOrDefault("content", "")));
        } else if (tag.equals("link") && List.of(a.getOrDefault("rel", "")
                .toLowerCase().split("\\s+")).contains("canonical")) {
            canonical = a.get("href");
        } else if (tag.equals("script")
                && a.getOrDefault("type", "").equals("application/ld+json")) {
            inJsonld = true;
            jsonld.add("");
        } else if (tag.equals("main")) {
            hasMain = true;
            inMain++;
        } else if (tag.matches("h[1-6]")) {
            headingLevel = tag.charAt(1) - '0';
            headingWords.clear();
        }
        if (SKIP.contains(tag)) {
            skip++;
        }
    }
```

Listing 3.11 (Continued) *PageFacts.java: the same collector in Java, with a small regular-expression tokenizer and raw-text handling for script and style.*

```
    private void end(String tag) {
        if (SKIP.contains(tag) && skip > 0) {
            skip--;
        }
        if (tag.equals("title")) {
            inTitle = false;
        } else if (tag.equals("script")) {
            inJsonld = false;
        } else if (tag.equals("main") && inMain > 0) {
            inMain--;
        } else if (tag.matches("h[1-6]") && headingLevel > 0) {
            String text = String.join(" ", headingWords);
            headings.add(new Heading(headingLevel, text));
            headingLevel = 0;
        }
    }

    private void data(String raw) {
        String text = inJsonld ? raw : decode(raw);
        if (inTitle) {
            title += String.join(" ", split(text));
        }
        if (inJsonld) {
            jsonld.set(jsonld.size() - 1, jsonld.get(jsonld.size() - 1) + text);
        }
        if (skip > 0) {
            return;
        }
        List<String> w = split(text);
        words.addAll(w);
        if (inMain > 0) {
            mainWords.addAll(w);
        }
        if (headingLevel > 0) {
            headingWords.addAll(w);
        }
    }

    String visibleText() {
        return String.join(" ", words);
    }
```

Listing 3.11 (Continued) *PageFacts.java: the same collector in Java, with a small regular-expression tokenizer and raw-text handling for script and style.*

```
    List<String> leadWords(int n) {
        List<String> source = hasMain ? mainWords : words;
        return source.subList(0, Math.min(n, source.size()));
    }
}
```

This tokenizer is enough for well-formed pages such as our samples; it is not a replacement for a real parser on the wild web, where tags are left open, attributes are unquoted and entities are exotic. Exercise 3.11 asks where it breaks.

For JSON, Python has `json.loads`. Java's standard library has no JSON parser either, so the Java version reuses `MiniJson.java`, the small parser written in Chapter 2's lab, copied unchanged into this chapter's folder. Both reject the sample's trailing comma, as the JSON standard requires.

3.10.6 Eight checks

With the facts collected, each check is a small function from a `Site`, the record of everything fetched, to a `CheckResult`. **Listing 3.12** contains all eight, in the order of **Table 3.4**.

Listing 3.12 `checks.py`: the eight checks. Each takes a `Site` and returns points, evidence and advice.

```
# checks.py - the eight checks of the lab; each returns a CheckResult
import json
from dataclasses import dataclass

CRAWLERS = ["GPTBot", "OAI-SearchBot", "ClaudeBot", "PerplexityBot",
            "Google-Extended", "Bingbot"]
MIN_WORDS, MIN_RATIO, LEAD = 120, 0.20, 60      # our own teaching choices


@dataclass
class Site:                     # everything fetched for one page
    url: str
    path: str
    facts: object               # PageFacts
    robots: object              # RobotsRules
    llms_status: int
    llms_text: str
    sitemap_status: int         # 0 when not fetched


@dataclass
class CheckResult:
    name: str
    points: float               # 0.0 .. 1.0
    evidence: str
    advice: str                 # empty when there is nothing to fix


def check_crawlers(site):
    blocked = [c for c in CRAWLERS if not site.robots.decide(c, "/")[0]]
    ok = len(CRAWLERS) - len(blocked)
    return CheckResult("crawlers", ok / len(CRAWLERS),
                       f"{ok} of {len(CRAWLERS)} crawlers may fetch /",
                       "unblock " + ", ".join(blocked) if blocked else "")


def check_sitemap(site):
    if site.robots.sitemaps:
        return CheckResult("sitemap", 1.0, "robots.txt names "
                           + site.robots.sitemaps[0], "")
    if site.sitemap_status == 200:
        return CheckResult("sitemap", 1.0, "/sitemap.xml exists", "")
    return CheckResult("sitemap", 0.0, "no Sitemap line, /sitemap.xml "
                       f"{site.sitemap_status}",
                       "publish /sitemap.xml and name it in robots.txt")
```

Listing 3.12 (Continued) *checks.py: the eight checks. Each takes a Site and returns points, evidence and advice.*

```
def check_llms(site):
    if site.llms_status != 200:
        return CheckResult("llms.txt", 0.0, f"/llms.txt {site.llms_status}",
                           "optional: publish /llms.txt (few AI bots read it)")
    lines = [s for s in site.llms_text.splitlines() if s.strip()]
    if lines and lines[0].startswith("# "):
        return CheckResult("llms.txt", 1.0, "present, title " + lines[0], "")
    return CheckResult("llms.txt", 0.5, "present, but no '# Title' line first",
                       "start /llms.txt with a '# Name' line")


def jsonld_nodes(doc):
    """The top-level nodes of one JSON-LD document, @graph members included."""
    nodes = doc if isinstance(doc, list) else [doc]
    out = []
    for n in nodes:
        if isinstance(n, dict):
            if "@type" in n:
                out.append(n)
            out.extend(g for g in n.get("@graph", []) if isinstance(g, dict))
    return out


def valid_nodes(site):
    """All top-level nodes of the JSON-LD blocks that parse."""
    nodes = []
    for block in site.facts.jsonld:
        try:
            nodes += jsonld_nodes(json.loads(block))
        except ValueError:
            pass
    return nodes


def check_jsonld(site):
    valid, invalid, nodes = 0, 0, []
    for block in site.facts.jsonld:
        try:
            nodes += jsonld_nodes(json.loads(block))
            valid += 1
        except ValueError:
            invalid += 1
    if valid + invalid == 0:
        return CheckResult("json-ld", 0.0, "no JSON-LD block",
                           "describe the business with schema.org JSON-LD")
```

Listing 3.12 (Continued) *checks.py: the eight checks. Each takes a Site and returns points, evidence and advice.*

```
    types = []
    for n in nodes:
        t = n.get("@type", [])
        for x in t if isinstance(t, list) else [t]:
            if x not in types:
                types.append(x)
    named = any("name" in n and "url" in n for n in nodes)
    found = ", ".join(types) or "no types"
    evidence = f"{valid} valid, {invalid} invalid: {found}"
    if valid and not invalid and named:
        return CheckResult("json-ld", 1.0, evidence, "")
    advice = "repair the invalid JSON" if invalid else "add name and url"
    return CheckResult("json-ld", 0.5 if valid else 0.0, evidence, advice)


def check_headings(site):
    levels = [lvl for lvl, _ in site.facts.headings]
    h1 = levels.count(1)
    jumps = [(a, b) for a, b in zip([1] + levels, levels) if b > a + 1]
    points = ((h1 == 1) + (not jumps)) / 2
    evidence = f"{h1} h1, {len(levels)} headings, level jumps: {len(jumps)}"
    advice = []
    if h1 != 1:
        advice.append("use exactly one h1")
    if jumps:
        advice.append(f"h{jumps[0][0]} is followed by h{jumps[0][1]}")
    return CheckResult("headings", points, evidence, "; ".join(advice))


def check_text(site):
    f = site.facts
    ratio = len(f.visible_text()) / f.html_bytes
    points = ((len(f.words) >= MIN_WORDS) + (ratio >= MIN_RATIO)) / 2
    return CheckResult("text", points,
                       f"{len(f.words)} words, text/HTML ratio {ratio:.2f}",
                       "" if points == 1 else
                       f"aim for {MIN_WORDS}+ words and ratio {MIN_RATIO:.2f}+")


def entity_name(site):
    for n in valid_nodes(site):
        if isinstance(n.get("name"), str):
            return n["name"], "JSON-LD"
    return site.facts.title.split(" | ")[0].strip(), "<title>"
```

Listing 3.12 (Continued) *checks.py: the eight checks. Each takes a Site and returns points, evidence and advice.*

```
def check_answer_first(site):
    name, source = entity_name(site)
    lead = " ".join(site.facts.lead_words(LEAD)).lower()
    found = bool(name) and name.lower() in lead
    return CheckResult("answer-1st", 1.0 if found else 0.0,
                       f'"{name}" (from {source}) '
                       + ("is" if found else "is not")
                       + f" in the first {LEAD} words",
                       "" if found else "open with who you are and what you do")


def check_meta(site):
    f = site.facts
    dated = [n["dateModified"] for n in valid_nodes(site) if "dateModified" in n]
    have = {"title": len(f.title) >= 15, "description": bool(f.description),
            "lang": bool(f.lang), "canonical": bool(f.canonical),
            "dateModified": bool(dated)}
    missing = [k for k, ok in have.items() if not ok]
    found = len(have) - len(missing)
    return CheckResult("meta", found / len(have),
                       f"title {len(f.title)} chars; {found} of {len(have)} "
                       "present"
                       + (f" (dateModified {dated[0]})" if dated else ""),
                       "add " + ", ".join(missing) if missing else "")


NAMES = ["crawlers", "sitemap", "llms.txt", "json-ld", "headings", "text",
         "answer-1st", "meta"]
CHECKS = [check_crawlers, check_sitemap, check_llms, check_jsonld,
          check_headings, check_text, check_answer_first, check_meta]
```

Three of the rules are worth writing as formulas, because the formulas make their behavior on the boundaries unambiguous. If a of the six crawlers may fetch `/`, the crawler check gives

$$p_{\text{crawl}} = \frac{a}{6} \tag{3.5}$$

so each blocked crawler costs one sixth of a point. The headings check uses the Iverson bracket $[P]$, which is 1 when the statement P is true and 0 when it is false. With h the number of `h1` headings and J the number of level jumps, a jump being a heading more than one level deeper than the heading before it,

$$p_{\text{head}} = \frac{[h = 1] + [J = 0]}{2} \tag{3.6}$$

For the "before" page, $h = 2$ and $J = 1$ (the `h1` "Our Services" is followed by `h4`), so $p_{\text{head}} = (0 + 0)/2 = 0$. For the "after" page, $h = 1$, and the outline `h1`, `h2`, `h3`, `h3`, `h3`, `h2`, … never goes down more than one level at a time, so $J = 0$ and $p_{\text{head}} = 1$. Note that going *up*, from `h3` back to `h2`, is never a jump. Finally, if k of the five basics are present,

$$p_{\text{meta}} = \frac{k}{5} \tag{3.7}$$

The functions `jsonld_nodes` and `valid_nodes` deserve a note. A JSON-LD block can be one object, a list of objects, or an object with an `@graph` list, the form our "after" page uses and the form Chapter 2's generator writes; `jsonld_nodes` flattens all three into one list of top-level nodes. `entity_name` then takes the business name from the first node that has a `name`, and falls back to

the page title, cut at the first `" | "`, when there is no valid JSON-LD. That fallback is how the "before" page ends up being tested for the name "Home".

The Java version (**Listing 3.13**) defines a functional interface, `Check`, with one method `run`. The list `CHECKS` holds eight method references, `Checks::crawlers` and so on, each of which *is* a `Check`. This is the object-oriented shape of the lab: one interface, many implementations, each scored and explained independently, and a report that does not need to know which checks exist. Adding a ninth check means writing one method and adding one entry to the list.

Listing 3.13 `Checks.java`: the `Site` and `CheckResult` records, the `Check` interface and the eight checks.

```
// Checks.java - the eight checks of the lab; each returns a CheckResult
import java.util.ArrayList;
import java.util.LinkedHashMap;
import java.util.List;
import java.util.Locale;
import java.util.Map;

public class Checks {
    static final List<String> CRAWLERS = List.of("GPTBot", "OAI-SearchBot",
            "ClaudeBot", "PerplexityBot", "Google-Extended", "Bingbot");
    static final int MIN_WORDS = 120;                  // our own teaching choices
    static final double MIN_RATIO = 0.20;
    static final int LEAD = 60;

    record Site(String url, String path, PageFacts facts, RobotsRules robots,
                int llmsStatus, String llmsText, int sitemapStatus) { }

    record CheckResult(String name, double points, String evidence,
                       String advice) { }

    interface Check {
        CheckResult run(Site site);
    }

    static CheckResult crawlers(Site site) {
        List<String> blocked = new ArrayList<>();
        for (String c : CRAWLERS) {
            if (!site.robots().decide(c, "/").allowed()) {
                blocked.add(c);
            }
        }
        int ok = CRAWLERS.size() - blocked.size();
        return new CheckResult("crawlers", (double) ok / CRAWLERS.size(),
                ok + " of " + CRAWLERS.size() + " crawlers may fetch /",
                blocked.isEmpty() ? "" : "unblock " + String.join(", ", blocked));
    }
```

Listing 3.13 (Continued) *Checks.java: the Site and CheckResult records, the Check interface and the eight checks.*

```
    static CheckResult sitemap(Site site) {
        if (!site.robots().sitemaps.isEmpty()) {
            return new CheckResult("sitemap", 1.0, "robots.txt names "
                    + site.robots().sitemaps.get(0), "");
        }
        if (site.sitemapStatus() == 200) {
            return new CheckResult("sitemap", 1.0, "/sitemap.xml exists", "");
        }
        return new CheckResult("sitemap", 0.0, "no Sitemap line, /sitemap.xml "
                + site.sitemapStatus(),
                "publish /sitemap.xml and name it in robots.txt");
    }

    static CheckResult llms(Site site) {
        if (site.llmsStatus() != 200) {
            return new CheckResult("llms.txt", 0.0,
                    "/llms.txt " + site.llmsStatus(),
                    "optional: publish /llms.txt (few AI bots read it)");
        }
        List<String> lines = site.llmsText().lines()
                .filter(s -> !s.isBlank()).toList();
        if (!lines.isEmpty() && lines.get(0).startsWith("# ")) {
            return new CheckResult("llms.txt", 1.0, "present, title "
                    + lines.get(0), "");
        }
        return new CheckResult("llms.txt", 0.5,
                "present, but no '# Title' line first",
                "start /llms.txt with a '# Name' line");
    }

    /** The top-level nodes of one JSON-LD document, @graph members included. */
    static List<Map<?, ?>> jsonldNodes(Object doc) {
        List<?> nodes = doc instanceof List<?> list ? list : List.of(doc);
        List<Map<?, ?>> out = new ArrayList<>();
        for (Object n : nodes) {
            if (n instanceof Map<?, ?> map) {
                if (map.containsKey("@type")) {
                    out.add(map);
                }
                if (map.get("@graph") instanceof List<?> graph) {
                    for (Object g : graph) {
                        if (g instanceof Map<?, ?> gm) {
                            out.add(gm);
                        }
                    }
                }
```

Listing 3.13 (Continued) *Checks.java: the Site and CheckResult records, the Check interface and the eight checks.*

```
            }
        }
        return out;
    }

    /** All top-level nodes of the JSON-LD blocks that parse. */
    static List<Map<?, ?>> validNodes(Site site) {
        List<Map<?, ?>> nodes = new ArrayList<>();
        for (String block : site.facts().jsonld) {
            try {
                nodes.addAll(jsonldNodes(MiniJson.parse(block)));
            } catch (RuntimeException e) {
                // not JSON: check jsonld counts it as invalid
            }
        }
        return nodes;
    }

    static CheckResult jsonld(Site site) {
        int valid = 0;
        int invalid = 0;
        List<Map<?, ?>> nodes = new ArrayList<>();
        for (String block : site.facts().jsonld) {
            try {
                nodes.addAll(jsonldNodes(MiniJson.parse(block)));
                valid++;
            } catch (RuntimeException e) {
                invalid++;
            }
        }
        if (valid + invalid == 0) {
            return new CheckResult("json-ld", 0.0, "no JSON-LD block",
                    "describe the business with schema.org JSON-LD");
        }
        List<String> types = new ArrayList<>();
        for (Map<?, ?> n : nodes) {
            Object t = n.get("@type");
            List<?> all = t instanceof List<?> list ? list
                    : (t == null ? List.of() : List.of(t));
            for (Object x : all) {
                if (!types.contains(x.toString())) {
                    types.add(x.toString());
                }
            }
        }
        boolean named = nodes.stream()
```

Listing 3.13 (Continued) *Checks.java: the Site and CheckResult records, the Check interface and the eight checks.*

```
                .anyMatch(n -> n.containsKey("name") && n.containsKey("url"));
        String evidence = valid + " valid, " + invalid + " invalid: "
                + (types.isEmpty() ? "no types" : String.join(", ", types));
        if (valid > 0 && invalid == 0 && named) {
            return new CheckResult("json-ld", 1.0, evidence, "");
        }
        String advice = invalid > 0 ? "repair the invalid JSON"
                : "add name and url";
        return new CheckResult("json-ld", valid > 0 ? 0.5 : 0.0, evidence,
                advice);
    }

    static CheckResult headings(Site site) {
        List<Integer> levels = site.facts().headings.stream()
                .map(PageFacts.Heading::level).toList();
        long h1 = levels.stream().filter(l -> l == 1).count();
        List<int[]> jumps = new ArrayList<>();
        int previous = 1;
        for (int level : levels) {
            if (level > previous + 1) {
                jumps.add(new int[] {previous, level});
            }
            previous = level;
        }
        double points = ((h1 == 1 ? 1 : 0) + (jumps.isEmpty() ? 1 : 0)) / 2.0;
        String evidence = h1 + " h1, " + levels.size()
                + " headings, level jumps: " + jumps.size();
        List<String> advice = new ArrayList<>();
        if (h1 != 1) {
            advice.add("use exactly one h1");
        }
        if (!jumps.isEmpty()) {
            int[] j = jumps.get(0);
            advice.add("h" + j[0] + " is followed by h" + j[1]);
        }
        return new CheckResult("headings", points, evidence,
                String.join("; ", advice));
    }
```

Listing 3.13 (Continued) *Checks.java: the Site and CheckResult records, the Check interface and the eight checks.*

```
    static CheckResult text(Site site) {
        PageFacts f = site.facts();
        double ratio = (double) f.visibleText().length() / f.htmlBytes;
        double points = ((f.words.size() >= MIN_WORDS ? 1 : 0)
                + (ratio >= MIN_RATIO ? 1 : 0)) / 2.0;
        return new CheckResult("text", points, String.format(Locale.ROOT,
                "%d words, text/HTML ratio %.2f", f.words.size(), ratio),
                points == 1 ? "" : String.format(Locale.ROOT,
                "aim for %d+ words and ratio %.2f+", MIN_WORDS, MIN_RATIO));
    }

    static String[] entityName(Site site) {
        for (Map<?, ?> n : validNodes(site)) {
            if (n.get("name") instanceof String name) {
                return new String[] {name, "JSON-LD"};
            }
        }
        return new String[] {site.facts().title.split(" \\| ")[0].strip(),
            "<title>"};
    }

    static CheckResult answerFirst(Site site) {
        String[] entity = entityName(site);
        String lead = String.join(" ", site.facts().leadWords(LEAD))
                .toLowerCase();
        boolean found = !entity[0].isEmpty()
                && lead.contains(entity[0].toLowerCase());
        return new CheckResult("answer-1st", found ? 1.0 : 0.0,
                "\"" + entity[0] + "\" (from " + entity[1] + ") "
                + (found ? "is" : "is not") + " in the first " + LEAD + " words",
                found ? "" : "open with who you are and what you do");
    }

    static CheckResult meta(Site site) {
        PageFacts f = site.facts();
        List<Object> dated = validNodes(site).stream()
                .filter(n -> n.containsKey("dateModified"))
                .<Object>map(n -> n.get("dateModified")).toList();
        Map<String, Boolean> have = new LinkedHashMap<>();
        have.put("title", f.title.length() >= 15);
        have.put("description",
                f.description != null && !f.description.isEmpty());
        have.put("lang", f.lang != null);
        have.put("canonical", f.canonical != null && !f.canonical.isEmpty());
        have.put("dateModified", !dated.isEmpty());
        List<String> missing = have.entrySet().stream()
```

Listing 3.13 (Continued) *Checks.java: the Site and CheckResult records, the Check interface and the eight checks.*

```
                .filter(e -> !e.getValue()).map(Map.Entry::getKey).toList();
        int found = have.size() - missing.size();
        return new CheckResult("meta", (double) found / have.size(),
                "title " + f.title.length() + " chars; " + found + " of "
                + have.size() + " present"
                + (dated.isEmpty() ? "" : " (dateModified " + dated.get(0) + ")"),
                missing.isEmpty() ? "" : "add " + String.join(", ", missing));
    }

    static final List<String> NAMES = List.of("crawlers", "sitemap", "llms.txt",
            "json-ld", "headings", "text", "answer-1st", "meta");
    static final List<Check> CHECKS = List.of(Checks::crawlers, Checks::sitemap,
            Checks::llms, Checks::jsonld, Checks::headings, Checks::text,
            Checks::answerFirst, Checks::meta);
}
```

3.10.7 The report and the command line

Listing 3.14 ties everything together. `fetch` wraps one HTTP request with a custom `User-Agent`, `ReadinessLab/1.0 (book exercise)`, and a five-second timeout, and turns every outcome into a status code: the real one for an HTTP answer, including errors such as 404, and 0 when there is no answer at all. `gather` is where politeness lives. It fetches `robots.txt` **first**, and then asks `decide` whether *our own* crawler, `ReadinessLab`, may fetch the page. A tool that audits other people's `robots.txt` must obey it too. Only then does it fetch the page and `llms.txt`, and `/sitemap.xml` only if `robots.txt` did not already name a sitemap: never more than four requests, one per resource.

Listing 3.14 `readiness.py`: fetching politely, running the checks, printing the report and computing the teaching score.

```
# readiness.py - a mini AI-readiness checker for one web page (Chapter 3 lab)
# usage: python readiness.py <url> [--weight name=number ...]
import sys
import urllib.error
import urllib.parse
import urllib.request

from checks import CHECKS, CRAWLERS, NAMES, Site
from page_facts import PageFacts
from robots_rules import RobotsRules

AGENT = "ReadinessLab"                          # our own crawler's name
HEADERS = {"User-Agent": AGENT + "/1.0 (book exercise)"}


def fetch(url):
    """Return (status, body bytes); status 0 means no answer at all."""
    request = urllib.request.Request(url, headers=HEADERS)
    try:
        with urllib.request.urlopen(request, timeout=5) as response:
            return response.status, response.read()
    except urllib.error.HTTPError as e:
        return e.code, b""
    except (urllib.error.URLError, OSError):
        return 0, b""


def gather(url):
    parts = urllib.parse.urlsplit(url)
    base = f"{parts.scheme}://{parts.netloc}"
    path = parts.path or "/"
    status, body = fetch(base + "/robots.txt")
    robots = RobotsRules(body.decode("utf-8", "replace"), status)
    allowed, reason = robots.decide(AGENT, path)
    if not allowed:                             # robots.txt binds us too
        return None, f"{AGENT} may not fetch {path}: {reason}"
    status, body = fetch(url)
    if status != 200:
        return None, f"the page answered {status}"
    facts = PageFacts(body.decode("utf-8", "replace"))
    llms_status, llms_body = fetch(base + "/llms.txt")
    sitemap_status = 0
    if not robots.sitemaps:                     # only when robots.txt names none
        sitemap_status, _ = fetch(base + "/sitemap.xml")
    return Site(url, path, facts, robots, llms_status,
                llms_body.decode("utf-8", "replace"), sitemap_status), ""
```

Listing 3.14 (Continued) *readiness.py: fetching politely, running the checks, printing the report and computing the teaching score.*

```
def label(points):
    return "PASS" if points == 1 else ("FAIL" if points == 0 else "PART")


def report(site, weights):
    print("AI-readiness check (a teaching score, not Profitina's)")
    print("URL:", site.url)
    extra = f", sitemap.xml {site.sitemap_status}" if site.sitemap_status else ""
    print(f"Fetched: page 200 ({site.facts.html_bytes} bytes), robots.txt "
          f"{site.robots.status}, llms.txt {site.llms_status}{extra}")
    print("Crawler access to / according to robots.txt:")
    for crawler in CRAWLERS:
        ok, reason = site.robots.decide(crawler, "/")
        print(f"  {crawler:<16} {'allowed' if ok else 'blocked':<8} {reason}")
    total = weight_sum = 0.0
    for i, check in enumerate(CHECKS, 1):
        r = check(site)
        w = weights.get(r.name, 1.0)
        total += w * r.points
        weight_sum += w
        print(f"{i}. [{label(r.points)}] {r.name:<11} points {r.points:.2f},"
              f" weight {w:g}")
        print(f"{'':10}{r.evidence}")
        if r.advice:
            print(f"{'':10}fix: {r.advice}")
    score = int(100 * total / weight_sum + 0.5) if weight_sum else 0
    shown = ", ".join(f"{k}={v:g}" for k, v in weights.items())
    shown = shown + ", others 1" if shown else "all 1"
    print(f"Teaching score: {score} / 100 (weights: {shown})")
    return score


def parse_weights(args):
    """Turn ["--weight", "text=2", ...] into {"text": 2.0, ...}."""
    weights = {}
    for i in range(0, len(args), 2):
        if args[i] != "--weight" or i + 1 == len(args):
            raise ValueError(f"unexpected argument {args[i]}")
        name, _, value = args[i + 1].partition("=")
        if name not in NAMES:
            raise ValueError(f"unknown check {name!r}")
        weights[name] = float(value)
        if weights[name] < 0:
            raise ValueError("a weight cannot be negative")
```

Listing 3.14 (Continued) *readiness.py: fetching politely, running the checks, printing the report and computing the teaching score.*

```
    return weights


def main(argv):
    if not argv or argv[0].startswith("-"):
        print("usage: python readiness.py <url> [--weight name=number ...]")
        return 1
    try:
        weights = parse_weights(argv[1:])
    except ValueError as e:
        print("Bad option:", e)
        print("Checks:", ", ".join(NAMES))
        return 1
    site, problem = gather(argv[0])
    if site is None:
        print("Stopped:", problem)
        return 2
    report(site, weights)
    return 0


if __name__ == "__main__":
    sys.exit(main(sys.argv[1:]))
```

`report` prints the crawler table, then each check with its label, `PASS` for 1 point, `FAIL` for 0 and `PART` for anything in between, its points and weight, its evidence and its advice, and finally the score of Equation (3.2). The line `int(100 * total / weight_sum + 0.5)` is the floor of "value plus one half" for non-negative values. It is there instead of `round` because Python's `round` rounds halves to the nearest *even* number, `round(12.5)` being 12, while Java's `Math.round(12.5)` is 13; writing the rounding out makes both languages agree. For the same reason the Java version formats numbers with `Locale.ROOT`: in an Indonesian locale, `String.format("%.2f", 0.5)` would print `0,50`, with a decimal comma.

`parse_weights` reads options of the form `--weight name=number`, refuses unknown check names and negative weights, and reports mistakes in one line instead of a stack trace. The Java version, **Listing 3.15**, uses `java.net.http.HttpClient` [22], set to HTTP/1.1 and to follow redirects, and reads every body as bytes so that the byte count of Equation (3.4) is exact.

Listing 3.15 `Readiness.java`: the same fetching, reporting and scoring with `java.net.http.HttpClient`.

```java
// Readiness.java - a mini AI-readiness checker for one web page (Chapter 3 lab)
// usage: java Readiness <url> [--weight name=number ...]
import java.io.IOException;
import java.net.URI;
import java.net.http.HttpClient;
import java.net.http.HttpRequest;
import java.net.http.HttpResponse;
import java.nio.charset.StandardCharsets;
import java.time.Duration;
import java.util.LinkedHashMap;
import java.util.List;
import java.util.Locale;
import java.util.Map;

public class Readiness {
    static final String AGENT = "ReadinessLab";     // our own crawler's name
    static final HttpClient CLIENT = HttpClient.newBuilder()
            .version(HttpClient.Version.HTTP_1_1)
            .followRedirects(HttpClient.Redirect.NORMAL)
            .connectTimeout(Duration.ofSeconds(5)).build();

    record Fetched(int status, byte[] body) {
        String text() {
            return new String(body, StandardCharsets.UTF_8);
        }
    }

    /** Status 0 means no answer at all. */
    static Fetched fetch(String url) {
        HttpRequest request = HttpRequest.newBuilder(URI.create(url))
                .header("User-Agent", AGENT + "/1.0 (book exercise)")
                .timeout(Duration.ofSeconds(5)).build();
        try {
            HttpResponse<byte[]> r =
                    CLIENT.send(request, HttpResponse.BodyHandlers.ofByteArray());
            return new Fetched(r.statusCode(),
                    r.statusCode() == 200 ? r.body() : new byte[0]);
        } catch (IOException | InterruptedException
                | IllegalArgumentException e) {
            return new Fetched(0, new byte[0]);
        }
    }
```

Listing 3.15 (Continued) *Readiness.java: the same fetching, reporting and scoring with java.net.http.HttpClient.*

```
    /** Returns the Site, or null after printing why the check stopped. */
    static Checks.Site gather(String url) {
        URI uri = URI.create(url);
        String base = uri.getScheme() + "://" + uri.getRawAuthority();
        String path = uri.getRawPath() == null || uri.getRawPath().isEmpty()
                ? "/" : uri.getRawPath();
        Fetched robotsFile = fetch(base + "/robots.txt");
        RobotsRules robots =
                new RobotsRules(robotsFile.text(), robotsFile.status());
        RobotsRules.Decision d = robots.decide(AGENT, path);
        if (!d.allowed()) {                          // robots.txt binds us too
            System.out.println("Stopped: " + AGENT + " may not fetch " + path
                    + ": " + d.reason());
            return null;
        }
        Fetched page = fetch(url);
        if (page.status() != 200) {
            System.out.println("Stopped: the page answered " + page.status());
            return null;
        }
        Fetched llms = fetch(base + "/llms.txt");
        int sitemapStatus = 0;
        if (robots.sitemaps.isEmpty()) {    // only when robots.txt names none
            sitemapStatus = fetch(base + "/sitemap.xml").status();
        }
        return new Checks.Site(url, path, new PageFacts(page.text()), robots,
                llms.status(), llms.text(), sitemapStatus);
    }

    static String label(double points) {
        return points == 1 ? "PASS" : (points == 0 ? "FAIL" : "PART");
    }

    static String number(double w) {                      // like Python's {w:g}
        return w == Math.rint(w) ? Long.toString((long) w) : Double.toString(w);
    }
```

Listing 3.15 (Continued) *Readiness.java: the same fetching, reporting and scoring with java.net.http.HttpClient.*

```
    static int report(Checks.Site site, Map<String, Double> weights) {
        System.out.println(
                "AI-readiness check (a teaching score, not Profitina's)");
        System.out.println("URL: " + site.url());
        String extra = site.sitemapStatus() != 0
                ? ", sitemap.xml " + site.sitemapStatus() : "";
        System.out.println("Fetched: page 200 (" + site.facts().htmlBytes
                + " bytes), robots.txt " + site.robots().status + ", llms.txt "
                + site.llmsStatus() + extra);
        System.out.println("Crawler access to / according to robots.txt:");
        for (String crawler : Checks.CRAWLERS) {
            RobotsRules.Decision d = site.robots().decide(crawler, "/");
            System.out.printf("  %-16s %-8s %s%n", crawler,
                    d.allowed() ? "allowed" : "blocked", d.reason());
        }
        double total = 0;
        double weightSum = 0;
        int i = 0;
        for (Checks.Check check : Checks.CHECKS) {
            Checks.CheckResult r = check.run(site);
            double w = weights.getOrDefault(r.name(), 1.0);
            total += w * r.points();
            weightSum += w;
            System.out.printf(Locale.ROOT,
                    "%d. [%s] %-11s points %.2f, weight %s%n", ++i,
                    label(r.points()), r.name(), r.points(), number(w));
            System.out.println(" ".repeat(10) + r.evidence());
            if (!r.advice().isEmpty()) {
                System.out.println(" ".repeat(10) + "fix: " + r.advice());
            }
        }
        int score = weightSum == 0 ? 0 : (int) (100 * total / weightSum + 0.5);
        StringBuilder shown = new StringBuilder();
        for (Map.Entry<String, Double> e : weights.entrySet()) {
            shown.append(shown.length() == 0 ? "" : ", ")
                 .append(e.getKey()).append('=').append(number(e.getValue()));
        }
        System.out.println("Teaching score: " + score + " / 100 (weights: "
                + (shown.length() == 0 ? "all 1" : shown + ", others 1") + ")");
        return score;
    }
```

Listing 3.15 (Continued) *Readiness.java: the same fetching, reporting and scoring with java.net.http.HttpClient.*

```
    /** Turn ["--weight", "text=2", ...] into {"text": 2.0, ...}. */
    static Map<String, Double> parseWeights(List<String> args) {
        Map<String, Double> weights = new LinkedHashMap<>();
        for (int i = 0; i < args.size(); i += 2) {
            if (!args.get(i).equals("--weight") || i + 1 == args.size()) {
                throw new IllegalArgumentException("unexpected argument "
                        + args.get(i));
            }
            String[] kv = args.get(i + 1).split("=", 2);
            if (!Checks.NAMES.contains(kv[0])) {
                throw new IllegalArgumentException(
                        "unknown check '" + kv[0] + "'");
            }
            double w = Double.parseDouble(kv.length == 2 ? kv[1] : "");
            if (w < 0) {
                throw new IllegalArgumentException("a weight cannot be negative");
            }
            weights.put(kv[0], w);
        }
        return weights;
    }

    public static void main(String[] args) {
        if (args.length == 0 || args[0].startsWith("-")) {
            System.out.println(
                    "usage: java Readiness <url> [--weight name=number ...]");
            System.exit(1);
        }
        Map<String, Double> weights;
        try {
            weights = parseWeights(List.of(args).subList(1, args.length));
        } catch (IllegalArgumentException e) {        // includes a bad number
            System.out.println("Bad option: " + e.getMessage());
            System.out.println("Checks: " + String.join(", ", Checks.NAMES));
            System.exit(1);
            return;
        }
        Checks.Site site = gather(args[0]);
        if (site == null) {
            System.exit(2);
        }
        report(site, weights);
    }
}
```

3.10.8 Running the lab: before and after

The quickest way to see everything is the demonstration program, which starts two servers in the background, one for each sample site, checks both, and stops both servers before it ends, even if something fails on the way, because the stopping happens in a `finally` block. **Listing 3.16** is the Python version.

Listing 3.16 `demo.py`: serve both sample sites on ports 7030 and 7031, check them, and always stop the servers.

```
# demo.py - serve both sample sites, check them, and stop the servers
import threading

from readiness import gather, report
from serve_site import make_server

SITES = [("../sites/before", 7030), ("../sites/after", 7031)]


def main():
    servers = [make_server(folder, port, verbose=False) for folder, port in SITES]
    for s in servers:
        threading.Thread(target=s.serve_forever, daemon=True).start()
    try:
        for _, port in SITES:
            site, problem = gather(f"http://127.0.0.1:{port}/")
            if site is None:
                print("Stopped:", problem)
            else:
                report(site, {})
            print()
    finally:
        for s in servers:                  # always stop what we started
            s.shutdown()
            s.server_close()
    print("Both servers stopped.")


if __name__ == "__main__":
    main()
```

Listing 3.17 is the Java version. `HttpServer.stop(0)` stops a Java server at once.

Listing 3.17 `Demo.java`: the same demonstration in Java.

```java
// Demo.java - serve both sample sites, check them, and stop the servers
import com.sun.net.httpserver.HttpServer;
import java.io.IOException;
import java.util.List;
import java.util.Map;

public class Demo {
    public static void main(String[] args) throws IOException {
        List<String> folders = List.of("../sites/before", "../sites/after");
        List<Integer> ports = List.of(7030, 7031);
        HttpServer[] servers = new HttpServer[2];
        for (int i = 0; i < 2; i++) {
            servers[i] =
                    ServeSite.makeServer(folders.get(i), ports.get(i), false);
            servers[i].start();
        }
        try {
            for (int port : ports) {
                String url = "http://127.0.0.1:" + port + "/";
                Checks.Site site = Readiness.gather(url);
                if (site != null) {
                    Readiness.report(site, Map.of());
                }
                System.out.println();
            }
        } finally {
            for (HttpServer s : servers) {        // always stop what we started
                s.stop(0);
            }
        }
        System.out.println("Both servers stopped.");
    }
}
```

Open a terminal in the chapter's `python` folder and run `python demo.py` (on some systems `python3 demo.py`); or, in the `java` folder, compile once with `javac *.java` and run `java Demo`. The chapter's `README.md` gives the exact commands for Windows, macOS and Linux, in a terminal and in VS Code. Both versions print exactly this:

```
AI-readiness check (a teaching score, not Profitina's)
URL: http://127.0.0.1:7030/
Fetched: page 200 (1860 bytes), robots.txt 200, llms.txt 404, sitemap.xml 404
Crawler access to / according to robots.txt:
  GPTBot           blocked  Disallow: / in group "GPTBot"
  OAI-SearchBot    allowed  Allow: / in group "*"
  ClaudeBot        blocked  Disallow: / in group "ClaudeBot"
  PerplexityBot    allowed  Allow: / in group "*"
  Google-Extended  allowed  Allow: / in group "*"
  Bingbot          allowed  Allow: / in group "*"
1. [PART] crawlers    points 0.67, weight 1
          4 of 6 crawlers may fetch /
          fix: unblock GPTBot, ClaudeBot
2. [FAIL] sitemap     points 0.00, weight 1
          no Sitemap line, /sitemap.xml 404
          fix: publish /sitemap.xml and name it in robots.txt
3. [FAIL] llms.txt    points 0.00, weight 1
          /llms.txt 404
          fix: optional: publish /llms.txt (few AI bots read it)
4. [FAIL] json-ld     points 0.00, weight 1
          0 valid, 1 invalid: no types
          fix: repair the invalid JSON
5. [FAIL] headings    points 0.00, weight 1
          2 h1, 5 headings, level jumps: 1
          fix: use exactly one h1; h1 is followed by h4
6. [FAIL] text        points 0.00, weight 1
          40 words, text/HTML ratio 0.13
          fix: aim for 120+ words and ratio 0.20+
7. [FAIL] answer-1st  points 0.00, weight 1
          "Home" (from <title>) is not in the first 60 words
          fix: open with who you are and what you do
8. [PART] meta        points 0.40, weight 1
          title 4 chars; 2 of 5 present
          fix: add title, canonical, dateModified
Teaching score: 13 / 100 (weights: all 1)

AI-readiness check (a teaching score, not Profitina's)
URL: http://127.0.0.1:7031/
Fetched: page 200 (3258 bytes), robots.txt 200, llms.txt 200
Crawler access to / according to robots.txt:
  GPTBot           allowed  Allow: / in group "*"
  OAI-SearchBot    allowed  Allow: / in group "*"
  ClaudeBot        allowed  Allow: / in group "*"
  PerplexityBot    allowed  Allow: / in group "*"
  Google-Extended  allowed  Allow: / in group "*"
  Bingbot          allowed  Allow: / in group "*"
1. [PASS] crawlers    points 1.00, weight 1
          6 of 6 crawlers may fetch /
2. [PASS] sitemap     points 1.00, weight 1
          robots.txt names https://laundry.example/sitemap.xml
3. [PASS] llms.txt    points 1.00, weight 1
          present, title # Profitina Laundry
4. [PASS] json-ld     points 1.00, weight 1
          1 valid, 0 invalid: DryCleaningOrLaundry, WebPage, FAQPage
5. [PASS] headings    points 1.00, weight 1
          1 h1, 10 headings, level jumps: 0
6. [PASS] text        points 1.00, weight 1
          175 words, text/HTML ratio 0.32
7. [PASS] answer-1st  points 1.00, weight 1
          "Profitina Laundry" (from JSON-LD) is in the first 60 words
8. [PASS] meta        points 1.00, weight 1
          title 62 chars; 5 of 5 present (dateModified 2026-09-20)
Teaching score: 100 / 100 (weights: all 1)

Both servers stopped.
```

Read the first report as Sari would. Crawler access is partial: four of six crawlers may read the page, and the table names the rule that shuts out each of the other two. There is no sitemap and no `llms.txt`. The JSON-LD exists but is invalid, so no types are found, and the advice says to repair it rather than to add it. The headings check finds two `h1` headings and one jump, and its advice names the jump, `h1` followed by `h4`. The page has 40 visible words and a text-to-HTML ratio of 0.13. The answer-first check has no valid JSON-LD to take a name from, so it uses the title, "Home", which of course does not appear in the opening words. Two of the five basics, the description and the `lang` attribute, are present.

It is worth computing the score of the first report by hand with Equation (3.2), as an exercise in reading one's own tool. All weights are 1, so the denominator is 8. The points are $4/6$ for the crawlers, by Equation (3.5), and $2/5 = 0.4$ for the meta tags, by Equation (3.7); all others are 0. Their sum is

$$\frac{4}{6} + 0.4 = 0.6667 + 0.4 = 1.0667,$$

and $100 \cdot 1.0667/8 = 13.33$; adding one half gives 13.83, whose floor is **13**, the score printed.

The second report passes every check, and **Figure 3.12** shows the two side by side, drawn from the output above.

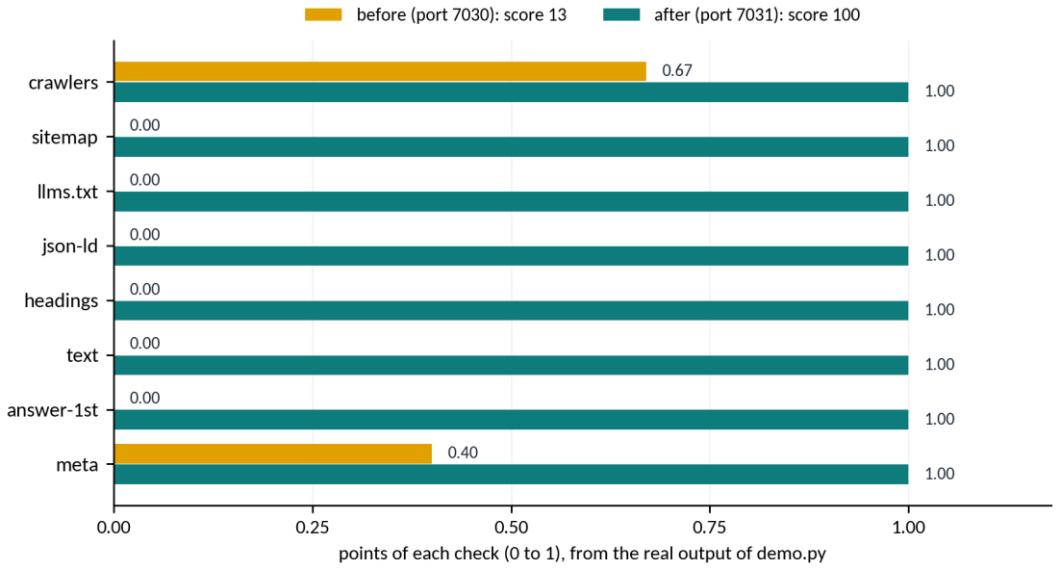


Figure 3.12 Points of each check for the two sample sites, taken from the real output of `demo.py`: before the fixes (score 13) and after them (score 100).

A score of 100 is a good moment to repeat the warning of Section 3.10.1. It means that the "after" page satisfies eight rules chosen for teaching. It does not mean that any answer engine will cite Profitina Laundry, and it does not mean that nothing else could be improved: the page has no photos with descriptive `alt` text, no opening hours in the JSON-LD, no reviews. A checklist that reaches its maximum has reached the end of *its* knowledge, not the end of the subject.

3.10.9 Choosing your own weights

The teaching score becomes a tool for thinking when you change the weights. Suppose you decide that crawler access matters three times as much as anything else, and that `llms.txt`, which almost no AI system requests yet [23], should not count at all. Start the server for the "before" site in one terminal:

```
python serve_site.py ../sites/before 7032
```

and run the checker with two weights in a second terminal:

```
python readiness.py http://127.0.0.1:7032/ --weight llms.txt=0 --weight crawlers=3
```

The server's terminal shows every request the checker made, and there are exactly four, one per resource, with `robots.txt` first:

```
Serving ../sites/before at http://127.0.0.1:7032/ (Ctrl+C stops)
  GET /robots.txt -> 200
  GET / -> 200
  GET /llms.txt -> 404
  GET /sitemap.xml -> 404
```

The checker prints the same eight checks with the new weights, and a new score:

```
AI-readiness check (a teaching score, not Profitina's)
URL: http://127.0.0.1:7032/
Fetched: page 200 (1860 bytes), robots.txt 200, llms.txt 404, sitemap.xml 404
Crawler access to / according to robots.txt:
  GPTBot           blocked  Disallow: / in group "GPTBot"
  OAI-SearchBot    allowed  Allow: / in group "*"
  ClaudeBot        blocked  Disallow: / in group "ClaudeBot"
  PerplexityBot    allowed  Allow: / in group "*"
  Google-Extended  allowed  Allow: / in group "*"
  Bingbot          allowed  Allow: / in group "*"
1. [PART] crawlers    points 0.67, weight 3
          4 of 6 crawlers may fetch /
          fix: unblock GPTBot, ClaudeBot
2. [FAIL] sitemap     points 0.00, weight 1
          no Sitemap line, /sitemap.xml 404
          fix: publish /sitemap.xml and name it in robots.txt
3. [FAIL] llms.txt    points 0.00, weight 0
          /llms.txt 404
          fix: optional: publish /llms.txt (few AI bots read it)
4. [FAIL] json-ld     points 0.00, weight 1
          0 valid, 1 invalid: no types
          fix: repair the invalid JSON
5. [FAIL] headings    points 0.00, weight 1
          2 h1, 5 headings, level jumps: 1
          fix: use exactly one h1; h1 is followed by h4
6. [FAIL] text        points 0.00, weight 1
          40 words, text/HTML ratio 0.13
          fix: aim for 120+ words and ratio 0.20+
7. [FAIL] answer-1st  points 0.00, weight 1
          "Home" (from <title>) is not in the first 60 words
          fix: open with who you are and what you do
8. [PART] meta        points 0.40, weight 1
          title 4 chars; 2 of 5 present
          fix: add title, canonical, dateModified
Teaching score: 27 / 100 (weights: llms.txt=0, crawlers=3, others 1)
```

By Equation (3.2), the denominator is now $3 + 1 + 0 + 1 + 1 + 1 + 1 + 1 = 9$, and the numerator is $3 \cdot \frac{4}{6} + 1 \cdot 0.4 = 2 + 0.4 = 2.4$. So $100 \cdot 2.4/9 = 26.67$, plus one half is 27.17, and the score is **27**. The same page scores 13 or 27 depending on what you decided matters, which is the whole point: a score is a *choice* of what to count and how much, applied consistently. When you read any readiness score, including a commercial one, the first question to ask is what it counts and how, and the second is whether it is applied the same way every time, so that differences over time mean something. Press Ctrl+C in the server's terminal to stop it. The Java commands are the same with `java ServeSite` and `java Readiness`, and they print the same lines.

3.10.10 When the checker stops

Two situations make the checker stop instead of reporting, and both are deliberate. If nothing is listening at the address, the request for `robots.txt` gets no answer, status 0, and by line 2 of **Pseudocode 3.1** the checker must assume that it is not allowed in:

```
python readiness.py http://127.0.0.1:7039/
```

```
Stopped: ReadinessLab may not fetch /: robots.txt unreachable: assume disallow
exit 2
```

The exit status 2 lets a script tell "stopped" apart from "reported" (status 0) and "used wrongly" (status 1). A misspelled check name is the second case:

```
python readiness.py http://127.0.0.1:7031/ --weight speed=2
```

```
Bad option: unknown check 'speed'
Checks: crawlers, sitemap, llms.txt, json-ld, headings, text, answer-1st, meta
exit 1
```

A mistake in the options is reported before any request is made, in words, with the list of valid names. Exercise 3.4 adds a third case, a site whose `robots.txt` closes its door to `ReadinessLab` in particular.

3.10.11 What the checker cannot see

Being clear about the limits of a tool is part of building it. Our checker reads the HTML that the server sends; it does not run JavaScript, so a page that builds its text in the browser looks empty to it, as it does to some crawlers. It checks one page, not a site. It verifies that JSON-LD is valid JSON with a name and an address, not that its facts are true, nor that they agree with the visible page. It knows nothing about links from other sites, about reviews, or about how often the business is mentioned elsewhere. And, like Profitina and by the same reasoning (Section 3.2.3), it never asks an answer engine anything. Each of these limits could be the start of an extension; several appear in the exercises.

3.11 Profitina in Practice

Lab G1 is a teaching model, and it is worth saying precisely how it relates to the product it models.

The same shape, different contents. Profitina's AI Visibility Engine, like the lab, turns observable facts into signals, signals into points, points into an estimate, and every gap into a sentence with a fix. It also, like the lab, reads `robots.txt` before anything else, because access comes first. What it contains is its own: which signals, how they are weighted, how they are combined and calibrated. The lab's eight checks and its equal weights were chosen to be read, not to predict.

Explanations as a product feature. The lab's `CheckResult` has an `evidence` field and an `advice` field for the same reason the product's result page spends most of its space on the fix list (Section 3.4): an estimate that cannot explain itself does not help anyone improve.

Honest failure. The lab stops with a clear message when `robots.txt` cannot be read, instead of guessing, and it refuses a misspelled option instead of ignoring it. The product behaves in the same spirit at every level: the category detector says "not sure yet" (Section 3.3.1); the Content Engine says when no model is available rather than passing off a template as AI output (Section 3.5.2); the Researcher Visibility Report returns "not found" for invented names, never a plausible

number (Section 3.7.1); a release is declared successful only after the new version answers its health check (Chapter 9).

The cache lesson, again. The panel-managed `robots.txt` of the "before" site is not an invented example. The same kind of file once blocked `GPTBot` and `ClaudeBot` on profitina.com itself, served by a content-delivery network in front of a correct origin file [2]. A checker that reads the *live* address, as `gather` does, catches that; a checker that reads the file in the site's source code does not. Chapter 10 builds the auditor that looks behind caches.

The loop around the lab. Chapter 2's lab G6 generates the fixes of the *Optimize* step; this chapter's lab G1 measures them in the *Scan* step; Chapter 9's lab G4 schedules re-scans for the *Monitor* step. Together the three labs form a miniature of the loop in **Figure 3.1**, built in the open.

3.12 Common Misunderstandings

"Profitina asks ChatGPT about my business." It does not. The answer engines are the targets whose readiness to cite a business is estimated from observable signals; Profitina neither queries them nor depends on them, for reasons of independence, privacy and principle (Section 3.2.3).

"A score of 100 means I will be cited." A readiness estimate measures how well a site makes itself readable, understandable and quotable by machines. Whether an engine quotes it for a particular question depends also on the question, the competing sources and the engine. The terms of service say that results are "methodological estimates, not guarantees" [3]. Lab G1's "after" page scores 100 on eight teaching rules and still has much it could improve.

"GEO is a one-time fix." Sites, crawlers and engines all change. A theme update or a hosting panel can undo a fix silently, as the profitina.com `robots.txt` story shows. That is why the method is a loop and the third tab is *Monitor*.

"If `robots.txt` ends with `Allow: /`, every crawler is allowed." A crawler with its own group ignores the `*` group entirely. In the "before" site, `GPTBot` and `ClaudeBot` are blocked by their own groups although the file ends with `Allow: /`.

"A missing `robots.txt` blocks crawlers." The opposite: under RFC 9309, a 4xx answer means the crawler may fetch anything. It is a server *error* or no answer that means "assume disallow" [7].

"Invalid JSON-LD is better than nothing." To a strict parser it *is* nothing: a single trailing comma makes the whole block unreadable, as the "before" page shows. Validate structured data every time it changes.

"`llms.txt` makes AI read my site." Publishing one has become common, 9.3% of the top 1,000 domains had one in September 2026 [24], but in May 2026, 97% of the files in one large sample received no requests at all [23]. It is cheap hygiene, which is why lab G1 counts it as one check out of eight and lets you set its weight to zero.

"The researcher report ranks lecturers." Neither the Researcher Visibility Report nor Organization Visibility is a leaderboard of people. The first names closeable gaps in one person's bibliographic record; the second reports aggregate, institutional gaps such as the 466 spellings of one affiliation [2].

"Free users do not get their data rights." Export, correction and permanent deletion are free on every plan, forever, and only the user can start a deletion [2].

"A higher weight makes a check more important in reality." A weight expresses *your* judgment of importance in *your* score. Changing weights changes the number, as Section 3.10.9 showed with 13 and 27 for the same page, and says nothing about how any engine behaves.

3.13 Chapter Summary

- Profitina's method is a **loop of four verbs**, measure, explain, fix, monitor (**Figure 3.1**), shown to users as three tabs: **Scan · Optimize · Monitor** (SOM) and **Pindai · Optimasi · Pantau** (POP) (**Table 3.1**). It is a loop because sites, crawlers and engines change.
- A scan starts from a name and a domain; **auto-fill** proposes category, location and customer questions and says **"not sure yet"** when unsure. Its evidence comes from three families of observable signals, **access, structure and extractability**, each rooted in open standards (**Table 3.2**). Profitina **never queries** the answer engines.
- The result is a **0–100 readiness estimate**, not a guarantee, with its method shown and a **prioritized fix list** in plain language; each line has evidence and a next step, and feedback on a scan is always possible.
- **Optimize** offers the **Content Engine** (answer-first drafts, entities, FAQ, JSON-LD) behind a **quality gate** that serves AI drafts only when they beat a template (5 of 6 benchmark cases), and says so when no model is available; the **Technical Assistant** handles JSON-LD, `robots.txt` and Bing sitemap submission. The owner publishes.
- **Monitor** keeps a private history and a **weekly trend**, re-scans on schedule (Free weekly, Pro daily) and alerts on significant changes; a simple alert rule compares the change Δ_t of Equation (3.1) with a threshold δ larger than the ordinary noise.
- The **Researcher Visibility Report** (OpenAlex: DOI, ORCID, affiliation, open access, citations) names closeable gaps and has a **negative control** (8 of 8 fictional names at 0.0); **Organization Visibility** reports aggregate gaps (466 spellings in 500 works; 244 of 607 without ORCID at ITS) and is not a leaderboard of people.
- **One Text, Many Platforms** adapts one text to one of six platforms within the platform's character limit, with a visible queue, and shares to X through the user's own account, with no X API.
- The **user journey** has nine steps (**Figure 3.2**): arrive, try, register (email and password, unticked consent, email code), scan, optimize, monitor, participate, upgrade (Pro, Rp50,000 for 30 days by QRIS or PROFIT; **Table 3.3**), and exercise one's rights in the **Akun & Data Pribadi** panel, free forever.
- **Lab G1** (**Figure 3.11**, **Table 3.4**) is a polite checker in Python and Java that reads `robots.txt` first (**Pseudocode 3.1**, Equation (3.3)), collects page facts in one streaming pass (Equation (3.4)), runs eight explained checks (Equation (3.5), Equation (3.6), Equation (3.7)) and computes a transparent **teaching score** (Equation (3.2)) with weights you choose. On the sample sites it scores 13 before the fixes and 100 after (**Figure 3.12**); it is our own score, not Profitina's.

3.14 Exercises

Exercise 3.1. For each feature, name the verb of the loop it serves (measure, explain, fix or monitor) and the tab in which the user meets it: (a) the CSV export of a brand's history; (b) the "not sure yet" category prompt; (c) the list of crawlers in the Technical Assistant; (d) the ordering of fixes by priority; (e) an email alert after a re-scan; (f) the progress label "query 3/8".

Exercise 3.2. A brand's weekly scores are 52, 55, 54, 71, 70, 69, 48, 50. Compute every Δ_t with Equation (3.1). Which weeks raise an alert for $\delta = 10$, and which for $\delta = 3$? Which threshold would you choose, and why?

Exercise 3.3. Consider this `robots.txt`:

```
User-agent: *
Disallow: /
Allow: /services

User-agent: GPTBot
User-agent: ClaudeBot
Disallow: /status

User-agent: gptbot
Allow: /status/public
```

Decide by hand, with **Pseudocode 3.1**, whether each of the following is allowed: `GPTBot` fetching `/`, `/status/42` and `/status/public`; `ClaudeBot` fetching `/status/public`; `PerplexityBot` fetching `/services` and `/outlets`. Then write a program, in Python and in Java, that prints the decisions with `RobotsRules`; in Python, print Python's `urllib.robotparser` decision beside it and explain every disagreement.

Exercise 3.4. The folder `sites/closed` contains a page and this `robots.txt`:

```
User-agent: ReadinessLab
Disallow: /

User-agent: *
Allow: /
```

Serve it on port 7034 and run the checker against it. Predict the output first. Is the checker's behavior right? What would a well-behaved crawler named `GPTBot` be allowed to do on this site?

Exercise 3.5. Using Equation (3.2) and the points of the "before" page in Section 3.10.8, compute the teaching score with the weights `json-ld=2` and `headings=2`, all others 1. Then confirm with the program.

Exercise 3.6. A page has 4,000 bytes of HTML and 150 visible words that make 900 characters of visible text. Compute ρ with Equation (3.4) and the points of the `text` check. A second page has 40 words, 242 characters of visible text and 1,860 bytes of HTML; compute the same. Why is the ratio a weak signal on its own?

Exercise 3.7. Write an answer-first opening paragraph, of at most three sentences, for a page of Profitina Laundry about its dry-cleaning service, using only the facts in **Listing 3.5**. Then state which questions a customer might ask an answer engine that your paragraph answers.

Exercise 3.8. Give the three reasons why Profitina does not measure visibility by querying answer engines. Then state the strongest argument *for* querying them, and explain why, even so, it cannot replace a readiness estimate.

Exercise 3.9. A site's score varies by up to 3 points from week to week with no change to the site. A broken `robots.txt` usually lowers it by 15 points or more. Choose a threshold δ for the alert rule of Section 3.6.2, justify it, and describe one situation in which your choice would still fail to alert.

Exercise 3.10. RFC 9309 also allows two special characters in rule paths: `*` matches any sequence of characters, and `$` at the end of a path anchors it to the end of the URL path. Write a function, in Python and in Java, that decides whether a rule path with these characters matches a URL path,

and test it on `/*.pdf$` against `/menu.pdf` and `/menu.pdf?v=2`, `/status/*/print` against `/status/42/print`, `/status$` against `/status` and `/status/42`, and `/serv` against `/services`. How would you plug it into `RobotsRules`?

Exercise 3.11. Find two inputs on which `PageFacts.java` and `page_facts.py` collect different words, and one input on which both collect words that a reader would consider wrong. Explain each case.

Exercise 3.12. The Researcher Visibility Report returns "not found", never a score, for invented names. Explain in a short paragraph why a *negative control* makes all the other numbers of a report more trustworthy, and propose one negative control for lab G1.

3.15 Solutions to the Exercises

Solution 3.1. (a) Monitor, in the *Monitor* (*Pantau*) tab; CSV export is a Pro feature. (b) Measure, in *Scan* (*Pindai*): it makes sure the scan measures the right kind of business. (c) Fix, in *Optimize* (*Optimasi*). (d) Explain, in *Scan*: the result page is where measurement becomes a plan. (e) Monitor, sent after a scheduled re-scan. (f) Measure, in *Scan*: it reports the progress of a running scan.

Solution 3.2. The changes are $\Delta_2 = 55 - 52 = +3$, $\Delta_3 = 54 - 55 = -1$, $\Delta_4 = 71 - 54 = +17$, $\Delta_5 = 70 - 71 = -1$, $\Delta_6 = 69 - 70 = -1$, $\Delta_7 = 48 - 69 = -21$ and $\Delta_8 = 50 - 48 = +2$. With $\delta = 10$, weeks 4 ($+17$) and 7 (-21) raise alerts. With $\delta = 3$, weeks 2, 4 and 7 do, since $+3 \geq 3$. The ordinary variation here is about 1 to 3 points, so $\delta = 3$ alerts on noise (week 2) and would soon be ignored; $\delta = 10$ catches both real events, the improvement in week 4 and the drop in week 7, which looks like breakage and deserves an immediate look.

Solution 3.3. By hand. `GPTBot` is named by two groups, the second group (together with `ClaudeBot`) and the third (as `gptbot`, since names are compared ignoring case); their rules merge into `Disallow: /status` and `Allow: /status/public`, and the `*` group is ignored. So `/` matches no rule and is allowed; `/status/42` matches only `Disallow: /status` and is blocked; `/status/public` matches both, and the longer `Allow: /status/public` (14 characters against 7) wins, so it is allowed. `ClaudeBot` is named only by the second group, so for it `/status/public` matches only `Disallow: /status`: blocked. `PerplexityBot` is named by no group and uses `*`: `/services` matches `Disallow: /` (1 character) and `Allow: /services` (9), and the longer allows it; `/outlets` matches only `Disallow: /`: blocked. **Listing 3.18** prints these decisions and Python's `urllib.robotparser` beside them.

Listing 3.18 `robots_compare.py`: the RFC 9309 decisions of `RobotsRules` beside those of `urllib.robotparser`.

```
# robots_compare.py - Exercise 3.2: RFC 9309 decisions next to urllib.robotparser
import urllib.robotparser

from robots_rules import RobotsRules

TEXT = """User-agent: *
Disallow: /
Allow: /services

User-agent: GPTBot
User-agent: ClaudeBot
Disallow: /status

User-agent: gptbot
Allow: /status/public
"""
CASES = [("GPTBot", "/"), ("GPTBot", "/status/42"), ("GPTBot", "/status/public"),
         ("ClaudeBot", "/status/public"), ("PerplexityBot", "/services"),
         ("PerplexityBot", "/outlets")]

ours = RobotsRules(TEXT, 200)
theirs = urllib.robotparser.RobotFileParser()
theirs.parse(TEXT.splitlines())
print(f"{'crawler':<14}{'path':<16}{'rfc':<7}{'urllib':<8}reason")
for agent, path in CASES:
    ok, reason = ours.decide(agent, path)
    other = theirs.can_fetch(agent, path)
    print(f"{agent:<14}{path:<16}{'allow' if ok else 'block':<7}"
          f"{'allow' if other else 'block':<8}{reason}")
```

```
crawler       path            rfc    urllib  reason
GPTBot        /               allow  allow   no rule matches in group "GPTBot"
GPTBot        /status/42      block  block   Disallow: /status in group "GPTBot"
GPTBot        /status/public  allow  block   Allow: /status/public in group "GPTBot"
ClaudeBot     /status/public  block  block   Disallow: /status in group "ClaudeBot"
PerplexityBot /services       allow  block   Allow: /services in group "*"
PerplexityBot /outlets        block  block   Disallow: / in group "*"
```

The columns disagree twice, and both disagreements are the known differences of `urllib.robotparser` in the Python 3.11 used here [20]. For `GPTBot` on `/status/public`, it uses only the *first* group that names the crawler and does not merge the third, so it never sees `Allow: /status/public`. For `PerplexityBot` on `/services`, it applies the *first* matching rule in file order, `Disallow: /`, instead of the longest. The Java program, **Listing 3.19**, prints the RFC 9309 column; Java has no robots parser of its own to compare with.

Listing 3.19 `RobotsCompare.java`: the RFC 9309 decisions for the same six cases.

```java
// RobotsCompare.java - Exercise 3.2: the RFC 9309 decisions for six cases
public class RobotsCompare {
    static final String TEXT = """
            User-agent: *
            Disallow: /
            Allow: /services

            User-agent: GPTBot
            User-agent: ClaudeBot
            Disallow: /status

            User-agent: gptbot
            Allow: /status/public
            """;
    static final String[][] CASES = {{"GPTBot", "/"}, {"GPTBot", "/status/42"},
        {"GPTBot", "/status/public"}, {"ClaudeBot", "/status/public"},
        {"PerplexityBot", "/services"}, {"PerplexityBot", "/outlets"}};

    public static void main(String[] args) {
        RobotsRules ours = new RobotsRules(TEXT, 200);
        System.out.printf("%-14s%-16s%-7sreason%n", "crawler", "path", "rfc");
        for (String[] c : CASES) {
            RobotsRules.Decision d = ours.decide(c[0], c[1]);
            System.out.printf("%-14s%-16s%-7s%s%n", c[0], c[1],
                    d.allowed() ? "allow" : "block", d.reason());
        }
    }
}
```

```
crawler       path            rfc    reason
GPTBot        /               allow  no rule matches in group "GPTBot"
GPTBot        /status/42      block  Disallow: /status in group "GPTBot"
GPTBot        /status/public  allow  Allow: /status/public in group "GPTBot"
ClaudeBot     /status/public  block  Disallow: /status in group "ClaudeBot"
PerplexityBot /services       allow  Allow: /services in group "*"
PerplexityBot /outlets        block  Disallow: / in group "*"
```

Solution 3.4. The checker reads `robots.txt` first, finds a group naming `ReadinessLab` with `Disallow: /`, and stops before fetching the page. Python and Java print the same:

```
Stopped: ReadinessLab may not fetch /: Disallow: / in group "ReadinessLab"
exit 2
```

That is right: the site's owner asked this particular crawler to stay out, and a checker that audits other people's `robots.txt` must obey it. Any other crawler, `GPTBot` included, is named by no group, uses `*`, and may fetch everything because of `Allow: /`. (Remember that `robots.txt` is a request, not a lock: the staff notes on this page would need a password to be private.)

Solution 3.5. The denominator is $1+1+1+2+2+1+1+1=10$. The weighted points are $1\cdot\frac{4}{6}=0.6667$ for the crawlers and $1\cdot 0.4$ for the meta tags; `json-ld` and `headings` have weight 2 but 0 points, so they add nothing to the numerator. So $S=\lfloor 100\cdot 1.0667/10+0.5\rfloor=\lfloor 10.67+0.5\rfloor=11$. Raising the weight of checks the page *fails* lowers its score from 13 to 11. The program's last line confirms it:

```
Teaching score: 11 / 100 (weights: json-ld=2, headings=2, others 1)
```

(only the last line of the report is shown; the eight checks are as in Section 3.10.9, with the new weights).

Solution 3.6. First page: $\rho=900/4000=0.225\geq 0.20$ and $150\geq 120$ words, so the check gives $(1+1)/2=1$ point. Second page (the "before" page): $\rho=242/1860=0.130$ and 40 words, so

$(0 + 0)/2 = 0$ points. On its own the ratio is weak because it can be raised without improving anything a reader or an engine cares about, for example by moving a stylesheet into a separate file, which removes bytes from n_H, and it can be lowered by harmless things such as a large JSON-LD block, which is *good* for readiness. That is why the check pairs it with a minimum amount of text, and why neither threshold is sacred.

Solution 3.7. One possible answer: "Profitina Laundry in Rungkut, Surabaya, dry-cleans clothes from Rp25,000 for a shirt to Rp75,000 for a wedding dress or a suit. A normal order is ready in two days, and every outlet accepts cash and QRIS." It answers: who dry-cleans in Rungkut or Surabaya; how much dry cleaning a shirt, a suit or a wedding dress costs there; how long it takes; and whether one can pay with QRIS. It names the business in its first words, gives only checkable facts from the page, and uses no superlatives.

Solution 3.8. Independence (a third party's service and answers change, so a measurement that depends on them is unstable and can break), privacy (queries would send the business's name, competitors and customers' questions to an outside provider), and principle and cost (a house rule forbids depending on paid third-party APIs) (Section 3.2.3). The strongest argument *for* querying is that it observes the outcome itself, whether the business is actually mentioned. Even so, it cannot replace a readiness estimate: an engine's answer varies between runs, users and days, so a single query is an anecdote; it says *that* a business is absent but not *why*, and therefore gives no fix; and it measures one engine at one moment, while readiness is a property of the site that all engines share.

Solution 3.9. A reasonable choice is $\delta = 8$: more than twice the ordinary variation of 3 points, so noise alone almost never triggers an alert, and about half the typical drop from a broken `robots.txt`, so breakage almost always does. It would fail to alert when a problem arrives gradually, for example when the answer-first paragraph is edited away over several weeks and the score falls by 4 points a week; each Δ_t stays below δ although the total fall is large. A second rule that compares the score with its average over the last four weeks would catch that.

Solution 3.10. Translate the rule path into a regular expression: escape every ordinary character, replace each `*` by "any sequence of characters", `.*`, and turn a final `$` into an end anchor; then match from the start of the URL path. **Listing 3.20** and **Listing 3.21** do exactly that.

Listing 3.20 `robots_patterns.py`: matching rule paths with `*` and a final `$`.

```python
# robots_patterns.py - Exercise 3.10: the * and $ patterns of RFC 9309
import re


def pattern_regex(rule_path):
    """'*' matches any run of characters; a final '$' anchors the end."""
    anchored = rule_path.endswith("$")
    body = rule_path[:-1] if anchored else rule_path
    regex = ".*".join(re.escape(part) for part in body.split("*"))
    return re.compile(regex + ("$" if anchored else ""))


def matches(rule_path, path):
    return pattern_regex(rule_path).match(path) is not None   # from the start


CASES = [("/*.pdf$", "/menu.pdf"), ("/*.pdf$", "/menu.pdf?v=2"),
         ("/status/*/print", "/status/42/print"), ("/status$", "/status"),
         ("/status$", "/status/42"), ("/serv", "/services")]

if __name__ == "__main__":
    for rule, path in CASES:
        print(f"{rule:<17} {path:<18} {'match' if matches(rule, path) else '-'}")
```

Listing 3.21 `RobotsPatterns.java`: the same matching in Java, with `Pattern.quote` for the ordinary parts.

```java
// RobotsPatterns.java - Exercise 3.10: the * and $ patterns of RFC 9309
import java.util.regex.Pattern;

public class RobotsPatterns {
    /** '*' matches any run of characters; a final '$' anchors the end. */
    static Pattern patternRegex(String rulePath) {
        boolean anchored = rulePath.endsWith("$");
        String body = anchored ? rulePath.substring(0, rulePath.length() - 1)
                : rulePath;
        StringBuilder regex = new StringBuilder();
        String[] parts = body.split("\\*", -1);
        for (int i = 0; i < parts.length; i++) {
            regex.append(i == 0 ? "" : ".*").append(Pattern.quote(parts[i]));
        }
        return Pattern.compile(regex + (anchored ? "$" : ""));
    }

    static boolean matches(String rulePath, String path) {
        return patternRegex(rulePath).matcher(path).lookingAt(); // from the start
    }

    public static void main(String[] args) {
        String[][] cases = {{"/*.pdf$", "/menu.pdf"},
            {"/*.pdf$", "/menu.pdf?v=2"},
            {"/status/*/print", "/status/42/print"}, {"/status$", "/status"},
            {"/status$", "/status/42"}, {"/serv", "/services"}};
        for (String[] c : cases) {
            System.out.printf("%-17s %-18s %s%n", c[0], c[1],
                    matches(c[0], c[1]) ? "match" : "-");
        }
    }
}
```

Both print:

```
/*.pdf$           /menu.pdf          match
/*.pdf$           /menu.pdf?v=2      -
/status/*/print   /status/42/print   match
/status$          /status            match
/status$          /status/42         -
/serv             /services          match
```

`/menu.pdf?v=2` does not match `/*.pdf$` because the `$` requires the path to end right after `.pdf`. To plug the function into `RobotsRules`, replace the test `path.startswith(prefix)` (`path.startsWith(r.path())` in Java) by `matches(prefix, path)`. The length used for "most specific" stays the length of the rule's path as written, special characters included, because RFC 9309 defines the most specific match as the one with the most octets in the rule [7].

Solution 3.11. Two differences. First, named entities: for `<p>Café</p>` Python collects `Café`, because `html.parser` knows every HTML entity, while the Java tokenizer decodes only a few common ones and collects `Café`. Second, badly formed markup: for `<p class="a>text</p><p>next words</p>`, with a missing closing quote, Python's parser reads everything after the quote as part of an attribute value and collects no words at all, while the Java regular expression cannot match the broken tag and collects `<p`, `class="a>text`, `next` and `words`. Neither is what the person who wrote the page meant. A case where both are wrong: in `<p>A<b>B</b>C</p>`, which a reader sees as the single word "ABC" with a bold middle letter, both collect three words, `A`, `B` and `C`, because they treat every tag as a word boundary. A real extraction tool knows which elements are *inline* and joins their text. None of these occur in the sample pages, which is why the two versions agree on them; on the open web, a full parser is the right tool.

Solution 3.12. A system that returns a number for every input cannot show that its numbers come from the data rather than from its own habits. If invented names reliably receive "not found" and never a score, then every score that *is* returned is known to rest on a real record; if an invented name ever receives a plausible score, the method is broken, and the control reveals it before a real person is misjudged. For lab G1, a good negative control is a page with no content at all, an empty `<html></html>` served by a site with no `robots.txt`: every check that depends on content must give 0 points, and the report must say why. (A site with no `robots.txt` still earns the crawler points, which is correct by RFC 9309 and a good test that the checker reads the standard, not our expectations.)

References

[1] P. Aggarwal, V. Murahari, T. Rajpurohit, A. Kalyan, K. Narasimhan, and A. Deshpande, "GEO: Generative Engine Optimization," in *Proc. 30th ACM SIGKDD Conf. Knowledge Discovery and Data Mining (KDD '24)*, 2024. arXiv:2311.09735. [Online]. Available: https://arxiv.org/abs/2311.09735

[2] Profitina, *Profitina Whitepaper — Public Version v24*, Sep. 2026. [Online]. Available: https://app.profitina.com/static/whitepaper/

[3] Profitina, "Syarat dan Ketentuan" (Terms and Conditions), v1.1, effective Aug. 6, 2026, §7. [Online]. Available: https://app.profitina.com/legal/

[4] OpenAI, "Overview of OpenAI crawlers." [Online]. Available: https://developers.openai.com/api/docs/bots (accessed Sep. 24, 2026).

[5] Anthropic, "Does Anthropic crawl data from the web, and how can site owners block the crawler?" *Claude Help Center*. [Online]. Available: https://support.claude.com/en/articles/8896518 (accessed Sep. 24, 2026).

[6] Perplexity, "Perplexity crawlers." [Online]. Available: https://docs.perplexity.ai/guides/bots (accessed Sep. 24, 2026).

[7] M. Koster, G. Illyes, H. Zeller, and L. Sassman, "Robots Exclusion Protocol," IETF RFC 9309, Sep. 2022.

[8] Sitemaps.org, "Sitemaps XML format," protocol version 0.9. [Online]. Available: https://www.sitemaps.org/protocol.html

[9] Google, "Google's common crawlers," *Google Search Central*. [Online]. Available: https://developers.google.com/search/docs/crawling-indexing/google-common-crawlers (accessed Sep. 24, 2026).

[10] Microsoft, "Which crawlers does Bing use?" *Bing Webmaster Tools Help*. [Online]. Available: https://www.bing.com/webmasters/help/which-crawlers-does-bing-use-8c184ec0 (accessed Sep. 2026).

[11] Schema.org, "DryCleaningOrLaundry," Schema.org v30.1, Sep. 16, 2026. [Online]. Available: https://schema.org/DryCleaningOrLaundry

[12] W3C, "JSON-LD 1.1: A JSON-based Serialization for Linked Data," W3C Recommendation, Jul. 16, 2020. [Online]. Available: https://www.w3.org/TR/json-ld11/

[13] J. Howard, "The /llms.txt file," llmstxt.org, proposed Sep. 3, 2024, updated Aug. 10, 2026. [Online]. Available: https://llmstxt.org/

[14] WHATWG, *HTML Living Standard*, sections "The title element," "The meta element" and "Headings and outlines." [Online]. Available: https://html.spec.whatwg.org/ (accessed Sep. 2026).

[15] OurResearch, "OpenAlex: The open catalog to the global research system." [Online]. Available: https://openalex.org/ (accessed Sep. 2026).

[16] ORCID, "What is ORCID?" [Online]. Available: https://info.orcid.org/what-is-orcid/ (accessed Sep. 2026).

[17] Republik Indonesia, *Undang-Undang Nomor 27 Tahun 2022 tentang Pelindungan Data Pribadi*, Oct. 17, 2022.

[18] Python Software Foundation, "http.server — HTTP servers" and "urllib.request — Extensible library for opening URLs," *The Python Standard Library*. [Online]. Available: https://docs.python.org/3/library/http.server.html

[19] Oracle, "Package com.sun.net.httpserver," *Java SE 25 & JDK 25 API Specification*. [Online]. Available: https://docs.oracle.com/en/java/javase/25/docs/api/jdk.httpserver/com/sun/net/httpserver/package-summary.html

[20] Python Software Foundation, "Support RFC 9309 in robotparser," CPython issue #138907, Sep. 15, 2025. [Online]. Available: https://github.com/python/cpython/issues/138907

[21] Python Software Foundation, "html.parser — Simple HTML and XHTML parser," *The Python Standard Library*. [Online]. Available: https://docs.python.org/3/library/html.parser.html

[22] Oracle, "Package java.net.http," *Java SE 25 & JDK 25 API Specification*. [Online]. Available: https://docs.oracle.com/en/java/javase/25/docs/api/java.net.http/java/net/http/package-summary.html

[23] PPC Land, "llms.txt adoption rises 8.8x but 97% of files get zero AI requests," Jul. 2, 2026. [Online]. Available: https://ppc.land/llms-txt-adoption-rises-8-8x-but-97-of-files-get-zero-ai-requests/

[24] Rankability, "llms.txt adoption," scan of Sep. 18, 2026 (Tranco list of Sep. 17, 2026). [Online]. Available: https://www.rankability.com/data/llms-txt-adoption/

Chapter 4

Community by Design

By the end of this chapter you will know what Profitina's community is made of and why each part has the shape it has: a live feed that updates by itself, threads and quote-reposts, channels that anyone with enough PROFIT in the balance may open without paying for them, "N new" counters that count only what matters, one-to-one messages that stay private, profiles where every item is shared only by consent, and a dashboard that behaves like a cockpit. You will understand the three ways a web page can learn about new data, polling, WebSockets and Server-Sent Events, well enough to choose between them and to compute what each one costs. You will have seen, with real measurements from the September 2026 release, three ways in which a real-time feed can go wrong and how each was put right. And you will have built a live feed of your own for Profitina Laundry, with a server and a console client in Python and in Java and a browser page that posts instantly, rolls back on failure, counts unread posts per channel and ignores late replies, all checked automatically in a headless Chromium browser.

4.1 Where We Left Off

Chapter 3 followed one user through the visibility loop: scan a site, fix what the scan finds, and keep watching. That loop is something a business owner does alone, at a desk, with their own website. It is useful, and for many users it is the whole reason they come. But the tagline on every Profitina page has three words, *Gather · Grow · Give* (in Indonesian, *Bersama · Berkembang · Berbagi*), and the loop of Chapter 3 only really serves the middle one. This chapter is about the first and the last: the place where people *gather*, and where what they know is *given* to others [1].

That place is the community. It is not an add-on bolted to the side of the scanner. It is embedded in the application and, as a widget, across the marketing site; it is where a course of about 250 students has worked since 31 August 2026; and it is where a large share of the platform's engineering effort went in August and September 2026. It is also, as we will see, the part of Profitina that most resembles the social media everyone already knows, and the part where Profitina most deliberately behaves differently.

The chapter moves from the outside in. Section 4.2 asks why a visibility tool needs a community at all. Section 4.3 draws a map of its parts. Section 4.4 explains the technology that makes it live, Server-Sent Events, by comparing it with its two alternatives and computing what each costs. Sections 4.5 to 4.10 take the parts one at a time: the feed and its threads, channels, the unread counter, direct messages and profiles, the dashboard cockpit, and instant posting. Section 4.11 tells three true stories of things that went wrong, with the numbers. Section 4.12 is the lab, in which you build the whole mechanism in miniature.

One promise before we start. Everything in this chapter is described at the level of *boxes and arrows*: what each part does and why. The lab code is our own, written for this book, and it is deliberately simple enough to read in an afternoon. It is not Profitina's code, and it does not need to be: the ideas are what travel.

4.2 Why Build a Community into a Visibility Tool

4.2.1 People already live in feeds

Indonesia is one of the most connected societies in the world. The national internet association APJII counted 235,261,078 internet users in its 2026 survey, 81.72 % of a population of 287,303,234, up from 80.66 % a year earlier [2]. DataReportal's *Digital 2026* report for Indonesia counted 180 million active social media user identities in October 2025, equal to 62.9 % of the population [3]. For most of those people, the daily experience of the internet *is* a feed: a column of short items, newest on top, that grows while you watch.

A product that wants people to return has to meet them where they already are. A scan report, however good, is read once and filed. A conversation about the report ("my score went up after I added FAQ markup, here is what I changed"; "why does my score say *not sure yet* for my category?") is read by many, answered by some, and remembered. The community turns individual work into shared knowledge. That is the *Give* in the tagline.

4.2.2 What Profitina does differently

Profitina's community borrows the familiar shapes, a feed that updates in real time like a chat application, a thread view like X, quote-reposts with an optional comment and photo, but it changes the rules underneath in four ways that recur throughout this chapter [1].

1. **Reading is free; acting needs one account.** An anonymous visitor to profitina.com can read the community widget. Posting, replying, liking or following requires signing in. There is no second-class "guest poster".
2. **Contribution, not attention, is rewarded.** Small PROFIT rewards go to useful contributions by email-verified humans, with daily caps; Chapter 5 explains the fair-play machinery and Chapter 12 the coin itself. There are no ads, permanently, so there is no business reason to maximize time spent staring at the feed.
3. **Private things stay private by default.** Direct messages are one-to-one, profiles start private, handles are neutral, and email addresses are never publishable. Section 4.8 goes through each rule.
4. **Numbers mean one thing.** A counter that says "3 new" must mean exactly the same thing everywhere it appears. Section 4.7 shows how easily that goes wrong.

> **KEY INSIGHT.** A community is not a feature you add once; it is a set of promises you keep every time the page updates. "This is new", "this is private", "this was reviewed", "this number is right": each promise is cheap to state and expensive to keep under real traffic. Most of this chapter is about the cost of keeping them.

4.3 A Map of the Community

Figure 4.1 shows the parts of the community and how they connect. At the center sits the *live event bus*: a single stream of small notifications, served at `/api/events`, that tells every open page when something has changed. Around it are the parts a member actually sees, grouped into the public square (feed and threads, channels, follows and counters, and the widget on profitina.com) and the private rooms (direct messages, profiles and the dashboard). To the left is the safety layer, which screens what members write before it reaches anyone else; it is the subject of Chapter 5.

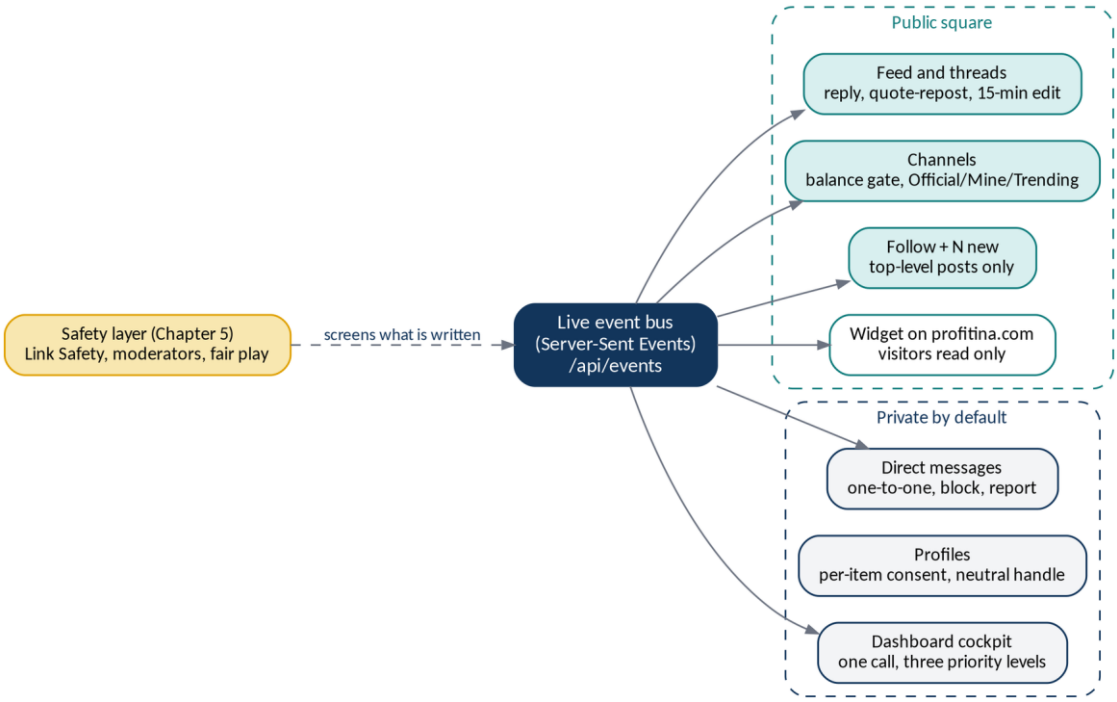


Figure 4.1 The community at box level. One live event bus feeds every part of the public square and the private rooms; the safety layer of Chapter 5 screens what is written before it is shown.

Table 4.1 lists the parts with the date on which each reached production, taken from the project's public whitepaper and release history [1]. Two things stand out. First, the whole community was built in about ten weeks, between the July 2026 launch and late September 2026. Second, several parts arrived *because of* the course of about 250 students that began using the platform on 31 August 2026: channel moderators and neutral handles were built in mid-August specifically so that a university class could work in the open without exposing anyone's student number.

Table 4.1 The parts of Profitina's community and when each reached production (dates from the public whitepaper and the release history).

Part	What it does	In production since
Live feed over SSE	New posts appear without reloading	July 2026
Threads and quote-repost	Replies nest; reposts may carry a comment and a photo	July 2026
User-created channels	Topic or course rooms, gated by a PROFIT balance	July 2026
15-minute edit window	Fix a typo; the reward is recalculated	12 August 2026
Channel moderators	Hide, pin and mute inside one channel only	13 August 2026
Neutral handles	Random u + 8 hex digits instead of real identifiers	14 August 2026
Direct messages	One-to-one, with block and report	20 August 2026
Profiles with per-item consent	Photo and four optional links, each shared by choice	20 August 2026
Dashboard cockpit	One call gathers everything that needs attention	15–16 September 2026
Instant (optimistic) posting	The post appears before the server answers	15–16 September 2026
Instant channel switching	A channel opens in milliseconds; reposts keep their channel	23–24 September 2026

4.4 Real Time Without Magic: Polling, WebSockets and Server-Sent Events

4.4.1 The problem: the web was built for asking

The web's basic conversation is a request followed by a response. A browser *asks* ("give me the page", "give me the feed") and a server *answers*. Nothing in that pattern lets the server speak first. Yet a live feed is exactly a situation in which the server knows something the browser does not: someone has just posted. How does the news get from the server to a page that did not ask?

There are three classic answers, drawn side by side in **Figure 4.2**. The amber stars mark moments when a new post appears on the server; teal arrows mark the moments when the news reaches the browser.

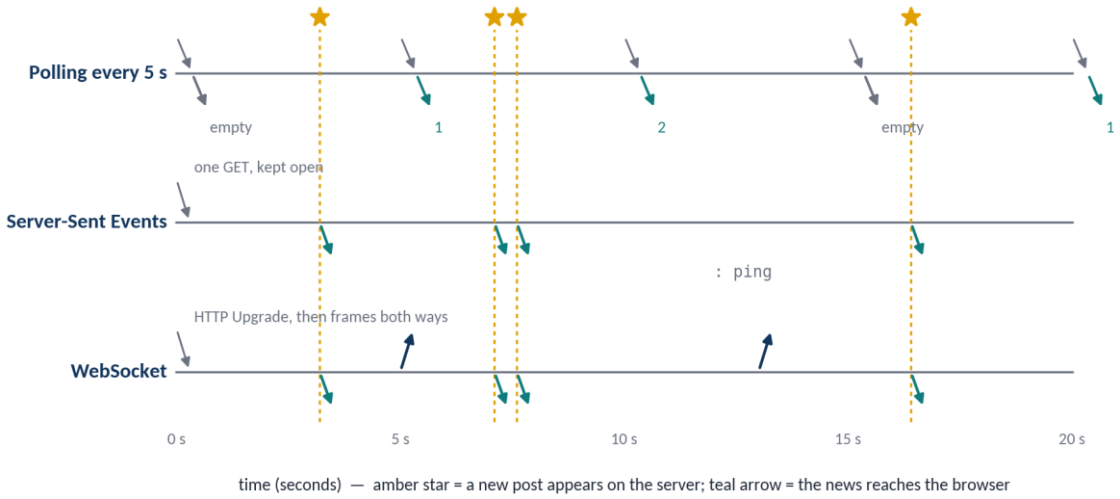


Figure 4.2 Three ways for news to reach a browser. Polling asks every five seconds and often gets nothing; Server-Sent Events keep one response open and push each post as it happens; WebSockets upgrade the connection so that both sides can send at any time.

4.4.2 Polling: asking again and again

The simplest answer is *polling*: the page asks "anything new?" every T seconds. It works with any server and any browser, and it is how many early chat pages were built. Its costs are easy to compute, and computing them is the best way to see why real-time systems moved on.

Cost 1: requests. If N pages are open and each polls every T seconds, the server receives

$$R = \frac{3600\,N}{T} \tag{4.1}$$

requests per hour. For Profitina's first class alone, $N = 250$ open pages polling every $T = 5$ seconds gives, by Equation (4.1), $R = 3600 \times 250/5 = 180{,}000$ requests per hour, or 50 per second, whether or not anyone posts anything.

Cost 2: delay. A post that appears at a random moment waits until the next poll. If the moment is equally likely to fall anywhere in the interval of length T between two polls, the waiting time W is uniformly distributed on $[0, T]$, and its average is

$$\mathbb{E}[W] = \frac{1}{T}\int_0^T w\; dw = \frac{1}{T}\cdot\frac{T^2}{2} = \frac{T}{2}. \tag{4.2}$$

With $T = 5$ seconds a new post takes 2.5 seconds on average to appear, and up to 5 seconds in the worst case. Halving T halves the delay but, by Equation (4.1), doubles the requests. Polling forces a trade between freshness and load, and there is no setting that is good at both.

Cost 3: waste. Most polls come back empty. Suppose posts arrive at random at an average rate of λ per second, independently of each other (a *Poisson process*, the standard model for arrivals such as phone calls or customers). The probability that an interval of length T contains no post at all is

$$P(\text{empty poll}) = e^{-\lambda T}. \tag{4.3}$$

Take a lively channel with one post per minute, so $\lambda = 1/60$, polled every $T = 5$ seconds. Then $\lambda T = 5/60 \approx 0.083$ and Equation (4.3) gives $e^{-0.083} \approx 0.920$. About 92 % of all polls carry nothing but the answer “no”. The polling lane of **Figure 4.2** shows two such empty answers among five.

4.4.3 WebSockets: a two-way pipe

The second answer is the WebSocket protocol, standardized in 2011 as RFC 6455 [4]. The browser opens an ordinary HTTP request that asks to *upgrade* the connection; if the server agrees, the connection stops being HTTP and becomes a two-way pipe through which either side can send small *frames* at any time. Online games, collaborative editors and trading screens use WebSockets because they need traffic in both directions, many times a second.

The price is that the pipe is no longer HTTP. Caches, proxies and load balancers that understand HTTP must be taught about the upgrade; the application must invent its own message format, its own reconnection logic and its own way of catching up after a disconnection. For a feed, where the server does almost all the talking and the browser only occasionally posts something, most of that power is unused.

4.4.4 Server-Sent Events: one long answer

The third answer, and the one Profitina uses, is *Server-Sent Events* (SSE), defined in the HTML Living Standard [5] and available in every current browser through the `EventSource` interface [6]. The idea is almost embarrassingly simple: the browser makes one ordinary GET request, and the server *never finishes answering it*. It sends the headers, then keeps the response open and writes a few lines of text each time something happens.

> **DEFINITION.** A *Server-Sent Events stream* is an HTTP response with content type `text/event-stream` whose body is a sequence of *events*. Each event is a group of lines of the form `field: value`, ended by a blank line. The fields are `data` (the payload), `event` (a name for the kind of event), `id` (a label the browser remembers) and `retry` (how many milliseconds to wait before reconnecting). A line that starts with a colon is a *comment* and is ignored.

Here is what the lab server of Section 4.12 actually writes, byte for byte, when a listener connects and two posts follow:

```
retry: 3000

id: 1
event: post
data: {"id":1,"channel":"general","top":true}

: ping

id: 2
event: post
data: {"id":2,"channel":"tips","top":true}
```

Read it line by line. The first event carries only `retry: 3000`: "if you lose me, wait three seconds and come back". The next event says that post number 1 has appeared in the channel *general* and that it is a top-level post, not a reply. Then comes `: ping`, a comment line with no meaning at all, and then post number 2.

Four properties make SSE a good fit for a feed:

- **It is plain HTTP.** The request is a GET, the response is text, and every proxy, tunnel and firewall that passes web pages passes it. Profitina's production host sits behind a tunnel that only makes outbound connections; SSE needs nothing special from it.
- **Reconnection is built in.** If the connection drops, `EventSource` waits for the `retry` interval and reconnects by itself. On reconnection it sends the header `Last-Event-ID` with the `id` of the last event it received, so a server that keeps a short history can *replay* exactly what was missed. The lab implements that replay.
- **Heartbeats are free.** A comment line such as `: ping` keeps idle connections alive through proxies that close silent ones, and it lets the server discover that a listener has gone: writing to a closed connection fails. The lab's heartbeat is fifteen seconds by default.
- **It is one-way, which is enough.** The browser still sends posts with an ordinary POST request. The stream is only for news.

SSE has two limitations worth knowing. It carries text only, so binary data such as photos travels through ordinary requests, which is how a feed wants it anyway. And under HTTP/1.1 a browser opens at most six connections to the same site, shared by all its tabs; each open stream holds one of them, so a user with seven tabs open may see the seventh stall. Under HTTP/2 the streams share one connection and the limit is effectively gone [6].

Table 4.2 summarizes the comparison.

Table 4.2 Polling, WebSockets and Server-Sent Events compared.

Property	Polling every T s	WebSocket	Server-Sent Events
Direction	Browser asks	Both ways	Server to browser
Protocol	HTTP	HTTP upgrade, then own frames	HTTP (`text/event-stream`)
Average delay	$T/2$, by Equation (4.2)	Near zero	Near zero
Requests per hour, N pages	$3600N/T$, by Equation (4.1)	N connections, held	N connections, held
Reconnect and catch-up	Next poll	Application's job	Built in (`retry`, `Last-Event-ID`)
Data	Anything	Text or binary	Text (UTF-8)
Good for	Rare updates, simplest servers	Games, editors, two-way traffic	Feeds, notifications, dashboards

> **TRAP.** “Real time” does not mean “send everything at once”. The stream should carry *small notifications*, not whole posts with photos and reply trees. Section 4.11.2 shows what happens when every notification makes every page fetch everything again.

4.4.5 Thin events

The events in the example above carry only three things: the post’s number, its channel, and whether it is top-level. They say *that* something happened, not *what* it says. A page that cares fetches the one post it needs; a page that does not care only updates a counter. We call such notifications *thin events*, and Section 4.11.2 explains why the distinction between *invalidating* and *refetching* turned out to matter a great deal in production.

4.5 The Feed, Threads and Quote-Repost

4.5.1 A feed is a list; a thread is a tree

The *feed* of a channel is the list of its posts, newest first. A *reply* is a post that names another post as its parent. Follow the parents and the posts form a *tree*: the original post is the root, its replies are its children, replies to replies are grandchildren, and so on. The thread view of a post shows that tree.

Figure 4.3 shows a small thread in a course channel. Post 1 announces an assignment. Post 2 asks about the deadline and post 3 answers it; post 4 simply says thanks. Posts 5 and 6 are not replies at all: they are *reposts*, which the next subsection explains.

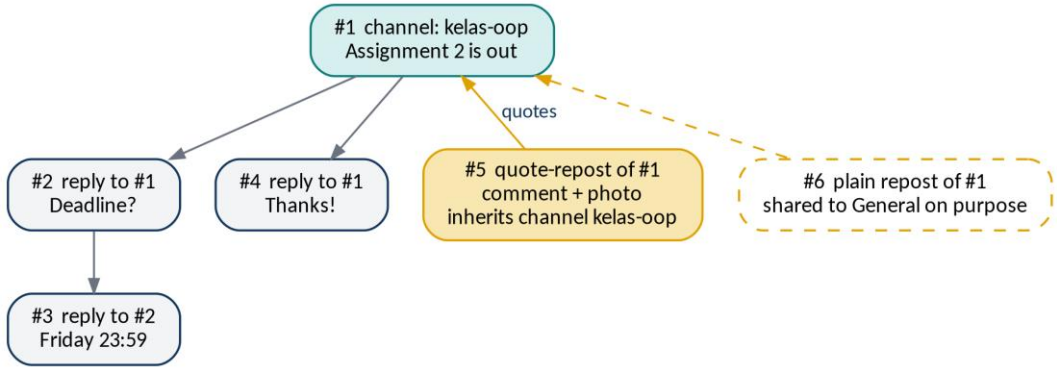


Figure 4.3 A thread is a tree of replies. A quote-repost (#5) points back to the post it quotes and, unless its writer chooses otherwise, inherits the original’s channel; a plain repost shared deliberately to General (#6) goes there instead.

Because every reply stores the number of its parent, the tree needs no extra structure: the thread of post r is post r together with every post whose chain of parents reaches r. The lab keeps exactly one field for this, `reply_to`, which is empty (`null`) for a top-level post.

4.5.2 Reposts and where they belong

A *repost* shares an existing post with more people. A *plain* repost shares it as it is; a *quote-repost* adds the sharer’s own comment and, optionally, a photo, and shows the original underneath. Profitina has both [1].

The interesting question is *where* a repost lives. Suppose a student reposts the lecturer’s assignment announcement. Should the repost appear in the course channel, where the announcement is, or in the general channel, where the student happened to be looking? In the

September 2026 release the rule became: **a repost inherits the channel of the original post, unless the person reposting deliberately picks General**. The repost dialog offers exactly those two choices, with the original's channel selected by default. The reasoning is the one used by the team chat tools most people know: a repost is a continuation of a conversation, and conversations belong to the room where they started. A repost that silently lands in a different room scatters the discussion and confuses the room's counter, because a new item appears in a channel that nobody in that channel posted to.

4.5.3 The fifteen-minute edit window

Everyone who has ever posted anything has noticed a typo one second after pressing *Send*. Since 12 August 2026 a Profitina post can be edited within fifteen minutes of posting, in the style of WhatsApp [1]. Written as a rule, a post created at time t_0 may be edited at time t if and only if

$$t - t_0 \leq 15 \text{ minutes}. \tag{4.4}$$

Two details make the rule honest. First, when a post is edited, the small PROFIT reward it earned is *recalculated in full* from the new text, and it can go down as well as up; otherwise the window would be a loophole in which a trivial post is written first and quietly upgraded later, or a useful post rewarded and then emptied. Second, after the window closes the text is final, so a reply written to a post cannot later find itself answering a different post. Equation (4.4) is short, but it protects every reply in the thread below.

4.6 Channels and the PROFIT-Balance Gate

4.6.1 Rooms with a purpose

A *channel* is a named room with its own feed. Profitina has official channels, run by the platform, and channels created by members: a course, a city's small-business owners, a topic such as structured data. The channel directory has three tabs, *Official*, *Mine* and *Trending*, so that a newcomer can find the rooms that matter without scrolling through every experiment anyone ever started [1].

Every channel that people actually use needs someone to look after it. Course channels are the clearest case: the lecturer pins announcements and keeps the room on topic. Channel-scoped moderators do exactly that, only inside their own channel, and Chapter 5 describes what they may and may not do.

4.6.2 The gate

Anyone can create a channel, but not without limits. A member may create a channel only when three conditions all hold:

$$\text{allowed} \Leftrightarrow (b \geq 50 \text{ PROFIT}) \wedge (d < 2) \wedge (a < 5), \tag{4.5}$$

where b is the member's PROFIT balance, d the number of channels they have created today (days reset at 00:00 WIB, Western Indonesian Time, UTC+7) and a the number of their channels that are currently active [1]. **Figure 4.4** draws the same rule as a flowchart.

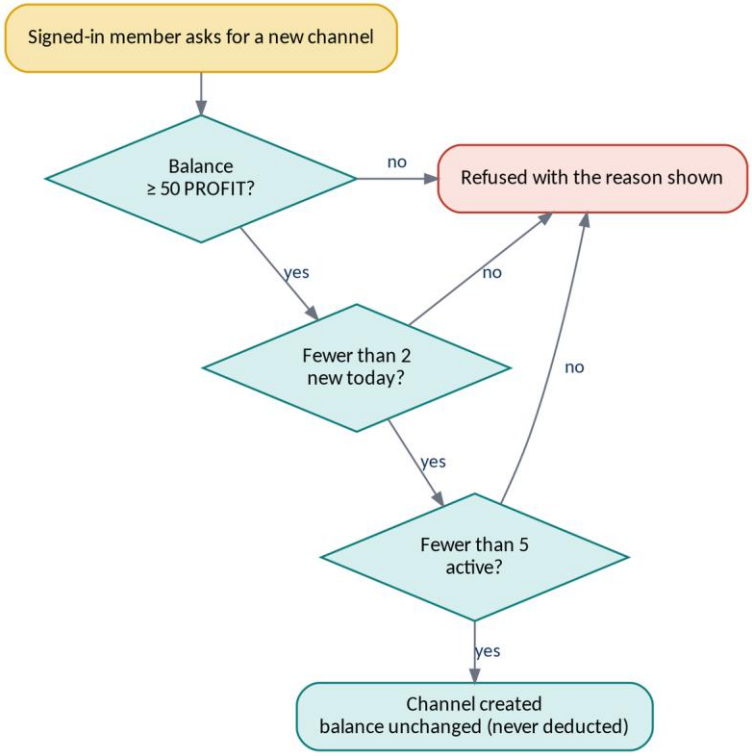


Figure 4.4 Creating a channel. A balance of at least 50 PROFIT is required but never deducted; at most two new channels per day and five active channels per member.

The most important words in the figure are *never deducted*. The balance is a *gate*, not a *fee*. A member with 50 PROFIT who creates a channel still has 50 PROFIT afterwards. Why design it this way rather than charging for channels?

- **A fee would sell visibility.** If a channel cost PROFIT, then people who bought PROFIT could buy rooms, and the directory would drift towards whoever spent most. A balance gate says only "you have taken part enough to have earned or acquired this much"; it does not transfer anything.
- **A gate slows spam without punishing anyone.** Spammers create many rooms quickly; the daily and active limits stop that, and the balance requirement means a freshly made throwaway account cannot create rooms at all.
- **It is easy to explain.** One sentence ("hold at least 50 PROFIT; nothing is taken") is a rule members can predict, which matters when the rule is enforced by software rather than by a person who can explain themselves.

> **EXAMPLE.** Sari holds 62 PROFIT, created one channel this morning and has three active channels. By Equation (4.5): $62 \geq 50$ holds, $1 < 2$ holds and $3 < 5$ holds, so she may create a second channel today. Afterwards she still holds 62 PROFIT, has created $d = 2$ channels today and has $a = 4$ active. A third attempt the same day fails at $d < 2$; the refusal message says so, and the counter resets at 00:00 WIB.

PROFIT itself, its devnet status and what may and may not be done with it are the subject of Chapter 12. Nothing in this chapter is a statement about its value; the gate uses it only as a count.

4.6.3 Per-channel leaderboards

Each channel also has its own leaderboard, free for every member, next to the platform's thirteen global leaderboards and the referral board [1]. For a course channel this is the natural place to

see who has contributed most useful material to that particular room, without comparing a first-year class with the whole platform.

4.7 Following Channels and the “N new” Counter

4.7.1 What a follower wants to know

A member who follows several channels wants to know, at a glance, *where something new has happened*. Profitina shows a small red badge next to each followed channel: “3 new”. Deciding what that number should count took more thought than one might expect.

4.7.2 The definition

For each member and each followed channel c, the platform remembers a single number s_c, the *last-seen marker*: the number of the newest post in c that the member has seen. Post numbers only ever grow, so every post with a larger number is newer. Let P_c be the set of posts in c. The badge shows

$$N_{\text{new}}(c) = \left|\{\, p \in P_c : p.\text{id} > s_c \ \wedge\ p.\text{reply\_to} = \text{null} \,\}\right|, \tag{4.6}$$

the number of *top-level* posts newer than the marker [1]. Opening the channel sets s_c to the newest post number, and the badge disappears.

Figure 4.5 works one example. The marker stands after post 5. Six items follow it, but three of them are replies, so the badge says “3 new”, not “6 new”.

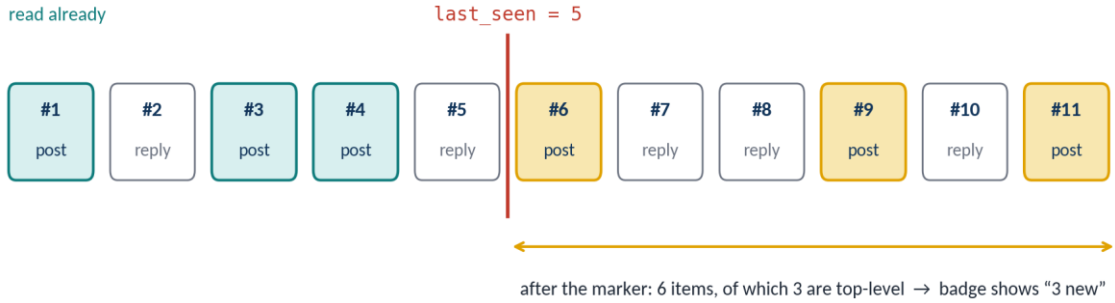


Figure 4.5 The “N new” counter. Only top-level posts after the last-seen marker are counted, so six new items of which three are replies give the badge “3 new”.

Why leave out replies? Because a badge is a promise about what the member will find. “6 new” suggests six new conversations; opening the channel would show three new posts at the top and three replies buried inside threads the member may not care about. Counting only top-level posts keeps the promise: every counted item is visible at the top of the feed. The replies are not lost; a member who wrote the parent post gets a notification for them on the dashboard (Section 4.9).

4.7.3 One word, one number

The September 2026 release also taught a smaller lesson about counters. The channel picker showed “6 posts” for a channel while a card elsewhere on the page said “this channel: 5 posts”; one count included reposts and the other did not. Both were defensible definitions, but a user who sees the same word next to two different numbers concludes that one of them is broken. The fix was to make every place that says *posts* for a channel use the same definition, and to give the different count used for rewards and rankings (original posts only) a different name. The rule generalizes: **the same word on the same screen must mean the same number.**

4.7.4 Keeping the counter live

The badge must update while the page is open. This is where the thin events of Section 4.4.5 earn their keep. Each event carries the channel and a flag `top`. When an event arrives for a channel that is *not* on screen, the page does not fetch anything; it adds one to that channel's badge if `top` is true, and nothing otherwise. That is Equation (4.6) maintained incrementally, one event at a time, at zero network cost. The lab's browser page does exactly this, and the automated browser check of Section 4.12.6 verifies that a reply does not raise the badge.

4.8 Direct Messages and Profiles: Private by Default

4.8.1 Direct messages

Since 20 August 2026 members can send each other *direct messages* (DMs) [1]. The design is deliberately narrow, and each restriction closes a known problem of larger platforms. **Table 4.3** lists them with their reasons.

Table 4.3 The rules for direct messages and the reason for each.

Rule	Why
One-to-one only; no groups, no @mentions	Groups and mentions are how spam and pile-ons spread; public discussion belongs in channels
No PROFIT for DMs	Rewards for private messages cannot be checked by anyone and would invite farming
In-app notifications only, unread badges	No email or phone channel to leak; the dashboard shows new DMs as urgent
Instant block and report	The person being bothered acts at once, without waiting for anyone
Administrators cannot read DMs	Private means private, including from the operator
Exception: a frozen copy of a *reported* message	Reporting must be possible; only the reported message, as it was when reported, becomes visible to reviewers

The last two rows deserve a closer look, because together they solve a real dilemma. If nobody can ever read a DM, then harassment by DM can never be proven. If the operator can read every DM, then DMs are not private. Profitina's answer is to make the act of reporting the *only* thing that exposes a message, and to expose a *frozen copy*: the message as it was at the moment of the report, which the sender can no longer edit or delete to hide what was said. Nothing else in the conversation becomes visible.

4.8.2 Profiles with per-item consent

A member's *profile* page can show a photo and up to four optional links to their accounts on other social platforms. Each of those five items has its own consent switch, every switch starts *off*, and each can be turned off again at any time [1]. An email address is never publishable, and there are no phone number fields at all. These rules follow directly from Indonesia's personal data protection law, UU PDP No. 27/2022, whose principles of purpose limitation and minimization Chapter 11 treats in depth: data that a feature does not need should not be collected, and data that is collected should be shown only with specific consent [7], [8].

4.8.3 Neutral handles

Every member is identified in the community by a *handle*. When a university class joined, a problem appeared immediately: the easiest handle for a student to choose is their student

number, and a student number is a personal identifier that links a post to an official record. Since 14 August 2026, new members receive a random *neutral handle* of the form `u` followed by eight hexadecimal digits, such as `u3f9a1c20`; members may change it, and the platform checks that a chosen handle does not look like a student number [1]. Anonymous visitors see names masked. The lab of Section 4.12 enforces the same handle pattern, and one of its test posts is rejected for breaking it.

> **KEY INSIGHT.** Privacy by default is mostly a matter of *starting values*. The same profile page, with the same five switches, is private or public depending only on whether the switches start off or on. Most people never change a default, so the default *is* the policy.

4.9 The Dashboard as a Cockpit

A pilot does not walk around the aircraft to check each instrument; the cockpit brings every instrument in front of one seat, arranged by urgency. Since mid-September 2026, Profitina's dashboard follows the same idea [1].

A single request, `GET /api/me/dashboard`, gathers everything that might need the member's attention: notifications (replies, likes and reposts of their posts, new DMs, pinned announcements in their channels, the status of feedback they submitted, and completed PROFIT purchases), their followed channels with each one's latest post and "N new" badge, and their rank and recent activity. At the top sits a *priority strip* with three levels: **urgent** (for example a DM waiting), **important** (for example replies to your post), and **next step** (guidance such as "re-scan your site this week"). Every item is a link to its own section of the application. **Figure 4.6** is a wireframe of the arrangement.

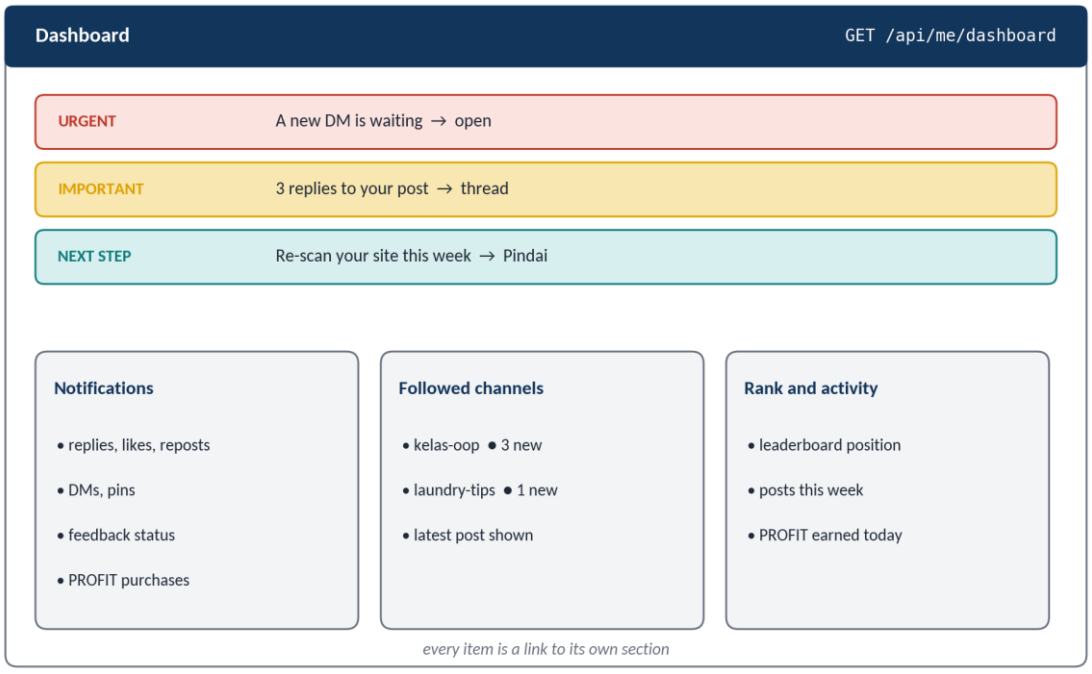


Figure 4.6 Wireframe of the dashboard cockpit. One request fills a three-level priority strip and three cards; every item links to its own section.

Why one request rather than one per card? Because each request has a fixed cost (a connection, authentication, a round trip across the internet), and a dashboard that makes ten requests pays it ten times and flickers into place piece by piece. One request that returns everything lets the page draw itself once. It also makes a rule easy to enforce on the server: the dashboard is assembled

from prefetched data in a small, fixed number of database queries. In the September 2026 release, after the changes described in Section 4.11.3, the dashboard answer took about 15 ms instead of 145 ms.

A small but telling detail concerns the followed-channels card. It first showed a channel's *pinned* announcement; members asked for the *latest* post instead, whoever wrote it, including their own. The card now shows the latest post and marks it "pinned" only if the latest post happens to be the pinned one. A dashboard is only useful if it answers the question its user is actually asking.

4.10 Instant (Optimistic) Posting

4.10.1 Waiting feels longer than it is

Human attention has well-known thresholds. About 0.1 second is the limit for an action to feel instantaneous; about 1 second is the limit for the flow of thought to stay uninterrupted; about 10 seconds is the limit for keeping attention on the task at all [9]. A post has to travel to the server, be checked (including by the safety layer of Chapter 5), be stored and be acknowledged. On a good day that is a few hundred milliseconds; on a slow mobile connection it is seconds. If the composer simply waits, the member stares at a frozen box and wonders whether anything happened.

4.10.2 Show it first, confirm it later

Optimistic posting reverses the order. The moment the member presses *Post*, the page clears the composer and draws the post at the top of the feed, marked as pending. Only then does it send the request. **Figure 4.7** shows the three states.

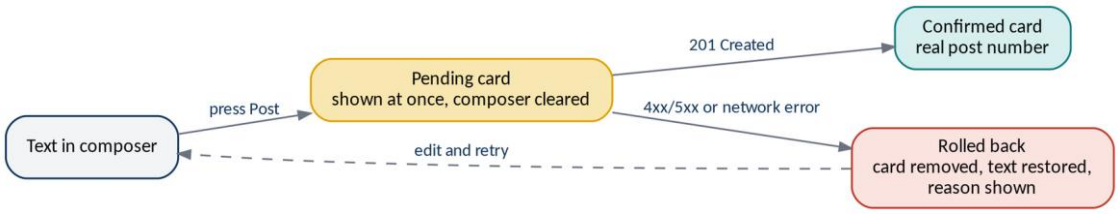


Figure 4.7 The life of an optimistic post. It is shown at once as pending; a success confirms it with its real number, and a failure removes it and gives the text back with the reason.

- If the server answers *201 Created*, the pending card becomes a normal card with the post's real number.
- If the server refuses (for example because the post is too long, or a link in it was blocked) or the network fails, the page **removes the card and puts the text back into the composer**, with the reason shown. Nothing the member typed is lost.

The optimism is safe because the server alone decides: the pending card is a prediction, and the prediction is corrected either way within one round trip. Profitina has posted this way since mid-September 2026; photos can be pasted from the clipboard or dragged into the composer and follow the same path [1].

One subtlety is easy to miss. The live stream of Section 4.4 will also announce the member's own post to their own page. If the page does not recognize it, the post appears twice: once as the optimistic card and once as "news". The lab solves this by remembering the numbers of posts it has confirmed and ignoring events for them, and by removing the pending card if the event happens to arrive before the confirmation.

4.11 When Real Time Goes Wrong: Three Engineering Lessons

Real-time systems fail in ways that simple request-and-response pages never do. The September 2026 release fixed three such failures after members reported that switching channels in the community was “very slow”. All the numbers below were measured on the platform’s own code with a PostgreSQL 16 database during that release, for a feed page of 30 posts.

4.11.1 Lesson 1: the read path must never call a model

Profitina’s link screening (Chapter 5) has an optional second layer in which a small local AI model gives its opinion about a suspicious link. Before the fix, the function that *rendered* each post, which runs every time anyone opens a feed, could ask that model about suspicious links in the post if no verdict had been stored for them. On a busy feed that meant model calls while a member was waiting for the page, six of them on one test page.

The fix was a rule rather than a patch: **a function that runs once per item while rendering must never call a model and never open its own database connection.** The model’s opinion is asked for once, on the *write* path, when the post is saved, and the verdict is stored with the post. Reading only reads.

4.11.2 Lesson 2: invalidate, do not refetch everything

The second failure was a storm. Each event on the live stream made the page reload the feed it was showing, even when the event concerned a different channel. With many members online, one post in one channel caused a full feed reload on every open page, and each reload generated more database work.

The cure is the distinction introduced in Section 4.4.5, drawn as a decision in **Figure 4.8**. An event for a channel that is *not* on screen only *invalidates*: the page marks its cached copy of that channel as stale and, if the event is top-level, adds one to the badge, with no network request at all. An event for the channel that *is* on screen causes a *refetch of exactly one item*, the new post, which is inserted at the top.

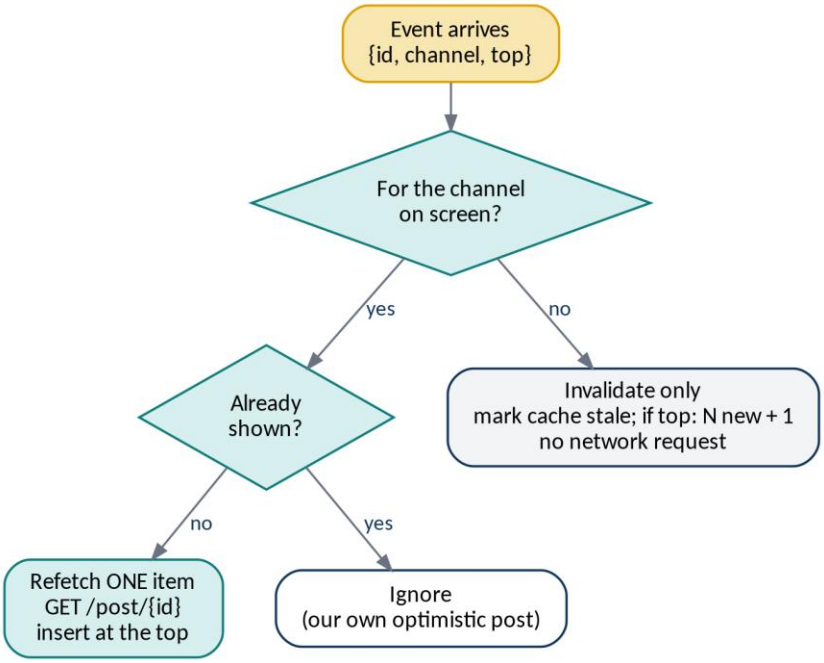


Figure 4.8 What a page does with a thin event. Events for other channels only invalidate a cache and update a counter; an event for the channel on screen fetches one post; an event for the member's own optimistic post is ignored.

4.11.3 Lesson 3: count your queries

The third failure was the classic *N+1 query problem*. To show a page of n posts, the code ran one query to list the posts and then, for each post, about eleven more: the writer's display name, the like count, whether the viewer liked it, the parent post, the channel name, and so on. The number of queries grew linearly with the page:

$$Q_{\text{before}}(n) \approx 1 + 11n, \qquad Q_{\text{after}}(n) = \text{constant}. \tag{4.7}$$

For $n = 30$, Equation (4.7) predicts $1 + 330 = 331$ queries before the fix; the measurement found 335. Worse, each query opened a *new* database connection, 122 of them for one page. The fix had two parts: an in-process *connection pool*, which keeps a few connections open and lends them out, and *prefetching*, which loads the names, counts and parents for all 30 posts in a handful of queries before rendering any of them. **Figure 4.9** shows the result.

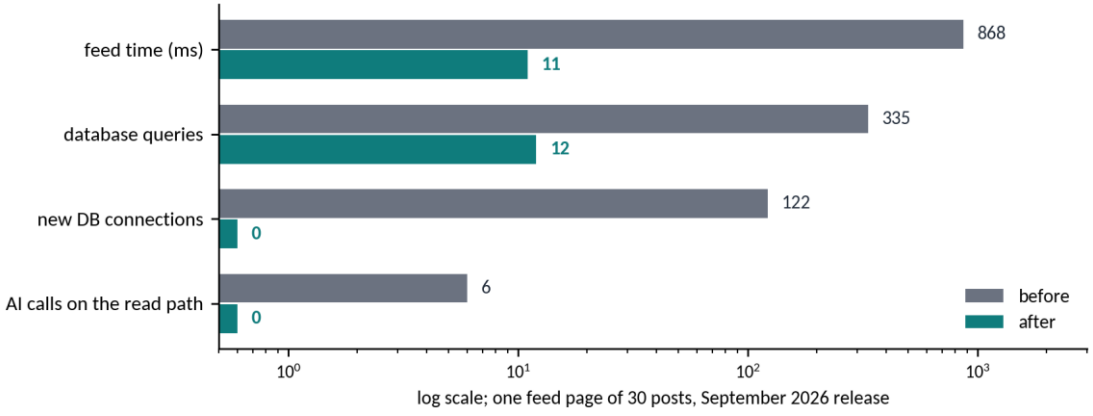


Figure 4.9 One feed page of 30 posts before and after the September 2026 fixes (log scale; a value of zero is drawn as a short stub). Time fell from 868 ms to 11 ms and queries from 335 to 12.

The same approach brought the embedded widget on profitina.com from 264 queries to 8 (about 100 ms to about 10 ms), and in a real Chromium browser 30 feed cards appeared 143 ms after a channel switch.

4.11.4 Lesson 4 (a bonus): late answers

When the channel switch became fast, a subtler bug became visible. A member clicks channel A and, before A's answer arrives, clicks channel B. B answers quickly and the screen shows B. Then A's slow answer arrives and the page, which does not know it is late, draws A's posts under B's title. **Figure 4.10** shows the race and its cure.

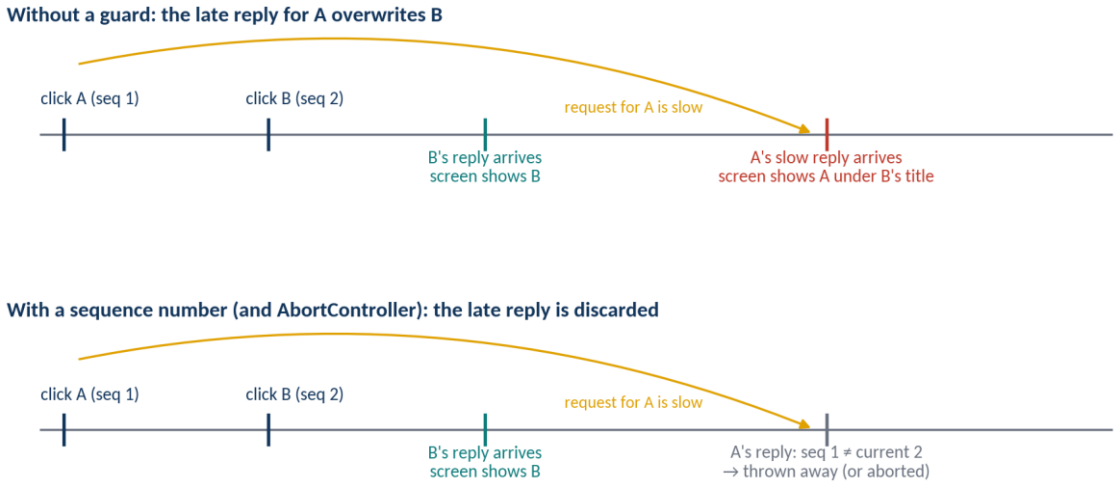


Figure 4.10 A stale-response race. Without a guard, a slow reply for channel A overwrites channel B; with a request sequence number, the late reply is recognized and thrown away.

The cure has two halves. Each request carries a *sequence number*; when an answer arrives, the page compares its number with the latest one and throws the answer away if it is older. And the older request is *aborted* with the browser's `AbortController` [10], so that it stops using the network. The release that fixed the feed initially guarded only the feed itself; the pinned-announcement banner and the channel leaderboard, loaded by separate requests, still suffered the same race until they were guarded too. The lesson: **every panel that loads per-channel data needs its own "is this still the active channel?" check.**

4.12 Lab G2: A Live Feed with Server-Sent Events

4.12.1 What we will build

The lab builds, for Profitina Laundry, a community feed with three channels (*general*, *tips* and *promo*) that updates live. It has five pieces, drawn in **Figure 4.11**:

- a **server**, once in Python (`feed_server.py`, port 7040) and once in Java (`FeedServer.java`, port 7041), that keeps posts in memory, accepts new ones, and streams thin events;
- a **browser page**, `feed.html`, shared by both servers, with channel tabs, "N new" badges, optimistic posting and a stale-reply guard;
- a **console client**, again in Python and Java, that listens to the stream or sends a post;
- a small **JSON helper** for Java, because the Java standard library has no JSON parser;
- an **automated check** that drives the page in headless Chromium.

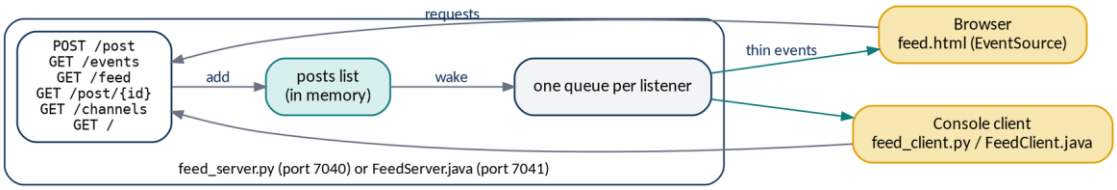


Figure 4.11 The lab. Either server keeps a list of posts and one queue per listener; browsers and console clients send ordinary requests and receive thin events over one open stream.

Everything uses the standard library only: `http.server`, `urllib` and `json` in Python [11], and `com.sun.net.httpserver` and `java.net.http` in Java [12], [13]. The code is in the book's code folder under `Chapter04/`, with a `README.md` that gives the commands for Windows, macOS and Linux. **Table 4.4** lists the server's interface.

Table 4.4 The lab server's interface (identical in Python and Java).

Request	Answer
`GET /`	The browser page `feed.html`
`GET /channels`	`["general","tips","promo"]`
`GET /events`	The event stream; honors `Last-Event-ID`
`GET /feed?channel=c&delay=ms`	Posts of channel `c`, newest first; `delay` simulates a slow answer
`GET /post/n`	Post number `n`
`POST /post`	Body `{"channel","handle","text","reply_to"}`; answers 201 with the new number, or 422 with the reason

4.12.2 The Python server

Listing 4.1 is the whole Python server. Read it in four parts.

Listing 4.1 `feed_server.py`: a live-feed server with Server-Sent Events, standard library only.

```
"""A tiny live-feed server with Server-Sent Events (standard library only).

Run:  python feed_server.py [--port 7040] [--heartbeat 15]
Then open http://127.0.0.1:7040/ in a browser.
"""
import argparse
import json
import queue
import re
import threading
import time
from http.server import BaseHTTPRequestHandler, ThreadingHTTPServer
from pathlib import Path
from urllib.parse import parse_qs, urlparse

CHANNELS = ["general", "tips", "promo"]
HANDLE = re.compile(r"^u[0-9a-f]{8}$")     # neutral handles such as u3f9a1c20
MAX_TEXT = 280
PAGE = Path(__file__).resolve().parent.parent / "web" / "feed.html"


class Hub:
    """Keeps every post in memory and one queue per connected listener."""

    def __init__(self):
        self.lock = threading.Lock()
        self.posts = []                      # post number n is posts[n - 1]
        self.listeners = []

    def add_post(self, channel, handle, text, reply_to):
        with self.lock:
            post = {"id": len(self.posts) + 1, "channel": channel,
                    "handle": handle, "text": text, "reply_to": reply_to}
            self.posts.append(post)
            for q in self.listeners:
                q.put(post)                  # wake every listener
        return post

    def get(self, post_id):
        with self.lock:
            if 1 <= post_id <= len(self.posts):
                return self.posts[post_id - 1]
        return None
```

Listing 4.1 (Continued) *feed_server.py: a live-feed server with Server-Sent Events, standard library only.*

```
    def channel_posts(self, channel, limit=50):
        with self.lock:
            found = [p for p in self.posts if p["channel"] == channel]
        return list(reversed(found[-limit:]))   # newest first

    def subscribe(self, last_id):
        """Register a listener; return its queue and the posts it missed."""
        q = queue.Queue()
        with self.lock:
            self.listeners.append(q)
            missed = self.posts[last_id:]
            count = len(self.listeners)
        return q, missed, count

    def unsubscribe(self, q):
        with self.lock:
            self.listeners.remove(q)
            return len(self.listeners)


def thin_event(post):
    """A small notification: which post, which channel, top-level or not."""
    data = {"id": post["id"], "channel": post["channel"],
            "top": post["reply_to"] is None}
    return "id: %d\nevent: post\ndata: %s\n\n" % (
        post["id"], json.dumps(data, separators=(",", ":")))


def validate(body):
    """Return (post fields, None) or (None, error message)."""
    channel = body.get("channel")
    handle = body.get("handle")
    text = body.get("text")
    reply_to = body.get("reply_to")
    if channel not in CHANNELS:
        return None, "unknown channel"
    if not isinstance(handle, str) or not HANDLE.match(handle):
        return None, "handle must be neutral, like u3f9a1c20"
    if not isinstance(text, str) or not text.strip():
        return None, "empty post"
    if len(text) > MAX_TEXT:
        return None, "post longer than %d characters" % MAX_TEXT
    if reply_to is not None and not isinstance(reply_to, int):
        return None, "reply_to must be a post number"
    return (channel, handle, text.strip(), reply_to), None
```

Listing 4.1 (Continued) *feed_server.py: a live-feed server with Server-Sent Events, standard library only.*

```
class Handler(BaseHTTPRequestHandler):
    hub = None                                  # set in main()
    heartbeat = 15.0

    def log_message(self, fmt, *args):          # keep the console quiet
        pass

    def send_json(self, status, obj):
        raw = json.dumps(obj, separators=(",", ":")).encode("utf-8")
        self.send_response(status)
        self.send_header("Content-Type", "application/json")
        self.send_header("Content-Length", str(len(raw)))
        self.end_headers()
        self.wfile.write(raw)

    def do_GET(self):
        url = urlparse(self.path)
        query = parse_qs(url.query)
        if url.path == "/":
            raw = PAGE.read_bytes()
            self.send_response(200)
            self.send_header("Content-Type", "text/html; charset=utf-8")
            self.send_header("Content-Length", str(len(raw)))
            self.end_headers()
            self.wfile.write(raw)
        elif url.path == "/events":
            self.stream_events()
        elif url.path == "/channels":
            self.send_json(200, CHANNELS)
        elif url.path == "/feed":
            channel = query.get("channel", ["general"])[0]
            delay = min(int(query.get("delay", ["0"])[0]), 5000)
            time.sleep(delay / 1000)            # lets you simulate a slow reply
            self.send_json(200, {"channel": channel,
                                 "posts": self.hub.channel_posts(channel)})
        elif url.path.startswith("/post/"):
            post = self.hub.get(int(url.path[6:] or 0))
            if post:
                self.send_json(200, post)
            else:
                self.send_json(404, {"error": "no such post"})
        else:
            self.send_json(404, {"error": "not found"})
```

Listing 4.1 (Continued) *feed_server.py: a live-feed server with Server-Sent Events, standard library only.*

```
    def do_POST(self):
        if self.path != "/post":
            self.send_json(404, {"error": "not found"})
            return
        length = int(self.headers.get("Content-Length", 0))
        try:
            body = json.loads(self.rfile.read(length) or b"{}")
        except json.JSONDecodeError:
            self.send_json(400, {"error": "body is not JSON"})
            return
        fields, error = validate(body)
        if not error and fields[3] is not None and not self.hub.get(fields[3]):
            error = "no such post to reply to"
        if error:
            print("rejected:", error, flush=True)
            self.send_json(422, {"error": error})
            return
        post = self.hub.add_post(*fields)
        print("post #%d [%s] by %s" % (post["id"], post["channel"],
                                       post["handle"]), flush=True)
        self.send_json(201, {"id": post["id"]})

    def stream_events(self):
        last = self.headers.get("Last-Event-ID") or "0"
        last_id = int(last) if last.isdigit() else 0
        q, missed, count = self.hub.subscribe(last_id)
        print("listener connected, last id %d (%d open)" % (last_id, count),
              flush=True)
        self.send_response(200)
        self.send_header("Content-Type", "text/event-stream")
        self.send_header("Cache-Control", "no-cache")
        self.end_headers()
        try:
            self.wfile.write(b"retry: 3000\n\n")      # reconnect after 3 s
            for post in missed:                       # replay what was missed
                self.wfile.write(thin_event(post).encode("utf-8"))
            self.wfile.flush()
            while True:
                try:
                    post = q.get(timeout=self.heartbeat)
                    chunk = thin_event(post)
                except queue.Empty:
                    chunk = ": ping\n\n"              # comment line = heartbeat
                self.wfile.write(chunk.encode("utf-8"))
                self.wfile.flush()
        except (BrokenPipeError, ConnectionResetError):
```

Listing 4.1 (Continued) *feed_server.py: a live-feed server with Server-Sent Events, standard library only.*

```
            pass
        finally:
            left = self.hub.unsubscribe(q)
            print("listener gone (%d open)" % left, flush=True)


def main():
    parser = argparse.ArgumentParser()
    parser.add_argument("--port", type=int, default=7040)
    parser.add_argument("--heartbeat", type=float, default=15.0)
    args = parser.parse_args()
    Handler.hub = Hub()
    Handler.heartbeat = args.heartbeat
    server = ThreadingHTTPServer(("127.0.0.1", args.port), Handler)
    server.daemon_threads = True
    print("feed server on http://127.0.0.1:%d/" % args.port, flush=True)
    try:
        server.serve_forever()
    except KeyboardInterrupt:
        print("stopped")


if __name__ == "__main__":
    main()
```

The hub. Class `Hub` holds the list of posts and a list of *listeners*, one `queue.Queue` per open stream. `add_post` appends the new post and puts it into every listener's queue while holding the lock, so a listener can never miss a post that was added between its subscription and its first read. `subscribe` registers a queue and, in the same locked step, returns the posts the listener missed: everything after `last_id`. That single lock is what makes replay exact.

Validation. `validate` refuses unknown channels, empty posts, posts longer than 280 characters, and handles that do not match the neutral pattern `u` plus eight hexadecimal digits of Section 4.8.3. The server alone decides; the browser's optimistic card is only a prediction.

Ordinary requests. `do_GET` and `do_POST` answer the requests of **Table 4.4**. The `delay` parameter of `/feed` exists only so that the stale-reply guard can be demonstrated.

The stream. `stream_events` is the heart of the lab. It reads `Last-Event-ID`, subscribes, sends the headers with content type `text/event-stream`, writes `retry: 3000`, replays what was missed, and then loops for ever: wait on the queue for up to `heartbeat` seconds; if a post arrives, write its thin event; if not, write the comment `: ping`. When the listener disappears, the next write raises `BrokenPipeError` or `ConnectionResetError`, the loop ends, and `finally` unsubscribes the queue. Because `ThreadingHTTPServer` runs each request in its own thread, a stream that waits for ever blocks only its own thread.

4.12.3 The Java server

Listing 4.2 does the same with the JDK's built-in HTTP server. A `record Post` holds one post and knows how to write itself as a map and as a thin event; the listeners are a `CopyOnWriteArrayList` of `BlockingQueue` objects, a list that is safe to iterate while other threads add or remove queues. `sendResponseHeaders(200, 0)` with a length of 0 tells the server that the length is unknown, so it streams the body using HTTP/1.1's *chunked* transfer coding [14], which is exactly what an event stream needs. `q.poll` with a timeout plays the role of Python's `q.get(timeout=...)`.

Listing 4.2 `FeedServer.java`: the same server with `com.sun.net.httpserver`, using one blocking queue per listener.

```
import com.sun.net.httpserver.HttpExchange;
import com.sun.net.httpserver.HttpServer;
import java.io.IOException;
import java.io.OutputStream;
import java.net.InetSocketAddress;
import java.nio.charset.StandardCharsets;
import java.nio.file.Files;
import java.nio.file.Path;
import java.util.ArrayList;
import java.util.HashMap;
import java.util.LinkedHashMap;
import java.util.List;
import java.util.Map;
import java.util.concurrent.BlockingQueue;
import java.util.concurrent.CopyOnWriteArrayList;
import java.util.concurrent.Executors;
import java.util.concurrent.LinkedBlockingQueue;
import java.util.concurrent.TimeUnit;
import java.util.regex.Pattern;

/** A tiny live-feed server with Server-Sent Events (JDK only).
 *  Run: java -cp out FeedServer [--port 7041] [--heartbeat 15] */
public class FeedServer {
    static final List<String> CHANNELS = List.of("general", "tips", "promo");
    static final Pattern HANDLE = Pattern.compile("^u[0-9a-f]{8}$");
    static final int MAX_TEXT = 280;

    /** One post; replyTo is null for a top-level post. */
    record Post(int id, String channel, String handle, String text,
                Long replyTo) {
        Map<String, Object> toMap() {
            Map<String, Object> m = new LinkedHashMap<>();
            m.put("id", id);
            m.put("channel", channel);
            m.put("handle", handle);
            m.put("text", text);
            m.put("reply_to", replyTo);
            return m;
        }
```

Listing 4.2 (Continued) *FeedServer.java: the same server with com.sun.net.httpserver, using one blocking queue per listener.*

```
        /** The small notification that travels over the stream. */
        String thinEvent() {
            Map<String, Object> m = new LinkedHashMap<>();
            m.put("id", id);
            m.put("channel", channel);
            m.put("top", replyTo == null);
            return "id: " + id + "\nevent: post\ndata: " + MiniJson.write(m)
                    + "\n\n";
        }
    }

    static final List<Post> posts = new ArrayList<>();       // guarded by posts
    static final List<BlockingQueue<Post>> listeners =
            new CopyOnWriteArrayList<>();
    static double heartbeat = 15.0;
    static Path page;

    static Post addPost(String channel, String handle, String text,
                        Long replyTo) {
        Post post;
        synchronized (posts) {
            post = new Post(posts.size() + 1, channel, handle, text, replyTo);
            posts.add(post);
        }
        for (BlockingQueue<Post> q : listeners) q.add(post);   // wake everyone
        return post;
    }

    static Post get(long id) {
        synchronized (posts) {
            if (id < 1 || id > posts.size()) return null;
            return posts.get((int) id - 1);
        }
    }

    static List<Object> channelPosts(String channel) {
        List<Object> out = new ArrayList<>();
        synchronized (posts) {
            for (int k = posts.size() - 1; k >= 0 && out.size() < 50; k--) {
                Post p = posts.get(k);
                if (p.channel().equals(channel)) out.add(p.toMap());
            }
        }
        return out;                                            // newest first
    }
```

Listing 4.2 (Continued) *FeedServer.java: the same server with com.sun.net.httpserver, using one blocking queue per listener.*

```
    static void sendJson(HttpExchange ex, int status, Object obj)
            throws IOException {
        byte[] raw = MiniJson.write(obj).getBytes(StandardCharsets.UTF_8);
        ex.getResponseHeaders().set("Content-Type", "application/json");
        ex.sendResponseHeaders(status, raw.length);
        try (OutputStream out = ex.getResponseBody()) { out.write(raw); }
    }

    static Map<String, String> query(HttpExchange ex) {
        Map<String, String> q = new HashMap<>();
        String raw = ex.getRequestURI().getRawQuery();
        if (raw == null) return q;
        for (String pair : raw.split("&")) {
            int eq = pair.indexOf('=');
            if (eq > 0) q.put(pair.substring(0, eq), pair.substring(eq + 1));
        }
        return q;
    }

    static String validate(Map<String, Object> body) {
        Object channel = body.get("channel"), handle = body.get("handle");
        Object text = body.get("text"), replyTo = body.get("reply_to");
        if (!CHANNELS.contains(channel)) return "unknown channel";
        if (!(handle instanceof String a) || !HANDLE.matcher(a).matches())
            return "handle must be neutral, like u3f9a1c20";
        if (!(text instanceof String t) || t.isBlank()) return "empty post";
        if (t.length() > MAX_TEXT)
            return "post longer than " + MAX_TEXT + " characters";
        if (replyTo != null && !(replyTo instanceof Long))
            return "reply_to must be a post number";
        if (replyTo != null && get((Long) replyTo) == null)
            return "no such post to reply to";
        return null;
    }

    static void handle(HttpExchange ex) throws IOException {
        String path = ex.getRequestURI().getPath();
        if (ex.getRequestMethod().equals("POST")) {
            if (!path.equals("/post")) {
                sendJson(ex, 404, Map.of("error", "not found"));
                return;
            }
            Map<String, Object> body;
            try {
                byte[] raw = ex.getRequestBody().readAllBytes();
                body = MiniJson.parseObject(
```

Listing 4.2 (Continued) *FeedServer.java: the same server with com.sun.net.httpserver, using one blocking queue per listener.*

```
                    new String(raw, StandardCharsets.UTF_8));
            } catch (RuntimeException e) {
                sendJson(ex, 400, Map.of("error", "body is not JSON"));
                return;
            }
            String error = validate(body);
            if (error != null) {
                System.out.println("rejected: " + error);
                sendJson(ex, 422, Map.of("error", error));
                return;
            }
            Post p = addPost((String) body.get("channel"),
                    (String) body.get("handle"),
                    ((String) body.get("text")).strip(),
                    (Long) body.get("reply_to"));
            System.out.printf("post #%d [%s] by %s%n",
                    p.id(), p.channel(), p.handle());
            sendJson(ex, 201, Map.of("id", p.id()));
            return;
        }
        Map<String, String> q = query(ex);
        switch (path) {
            case "/" -> {
                byte[] raw = Files.readAllBytes(page);
                ex.getResponseHeaders().set("Content-Type",
                        "text/html; charset=utf-8");
                ex.sendResponseHeaders(200, raw.length);
                try (OutputStream out = ex.getResponseBody()) { out.write(raw); }
            }
            case "/events" -> streamEvents(ex);
            case "/channels" -> sendJson(ex, 200, CHANNELS);
            case "/feed" -> {
                String channel = q.getOrDefault("channel", "general");
                int delay = Math.min(
                        Integer.parseInt(q.getOrDefault("delay", "0")), 5000);
                try {
                    Thread.sleep(delay);        // simulate a slow reply
                } catch (InterruptedException e) {
                    return;
                }
                Map<String, Object> m = new LinkedHashMap<>();
                m.put("channel", channel);
                m.put("posts", channelPosts(channel));
                sendJson(ex, 200, m);
            }
            default -> {
```

Listing 4.2 (Continued) *FeedServer.java: the same server with com.sun.net.httpserver, using one blocking queue per listener.*

```
                Post p = null;
                if (path.startsWith("/post/") && path.length() > 6
                        && path.substring(6).chars().allMatch(Character::isDigit)) {
                    p = get(Long.parseLong(path.substring(6)));
                }
                if (p != null) sendJson(ex, 200, p.toMap());
                else sendJson(ex, 404, Map.of("error", path.startsWith("/post/")
                        ? "no such post" : "not found"));
            }
        }
    }

    static void streamEvents(HttpExchange ex) throws IOException {
        String last = ex.getRequestHeaders().getFirst("Last-Event-ID");
        int lastId = last != null && last.matches("\\d+")
                ? Integer.parseInt(last) : 0;
        BlockingQueue<Post> q = new LinkedBlockingQueue<>();
        List<Post> missed;
        synchronized (posts) {                  // subscribe and snapshot together
            listeners.add(q);
            int from = Math.min(lastId, posts.size());
            missed = new ArrayList<>(posts.subList(from, posts.size()));
        }
        System.out.printf("listener connected, last id %d (%d open)%n",
                lastId, listeners.size());
        ex.getResponseHeaders().set("Content-Type", "text/event-stream");
        ex.getResponseHeaders().set("Cache-Control", "no-cache");
        ex.sendResponseHeaders(200, 0);          // 0 = length unknown, streamed
        OutputStream out = ex.getResponseBody();
        try {
            out.write("retry: 3000\n\n".getBytes(StandardCharsets.UTF_8));
            for (Post p : missed) {                 // replay what was missed
                out.write(p.thinEvent().getBytes(StandardCharsets.UTF_8));
            }
            out.flush();
            while (true) {
                Post p = q.poll((long) (heartbeat * 1000),
                                TimeUnit.MILLISECONDS);
                String chunk = p != null ? p.thinEvent() : ": ping\n\n";
                out.write(chunk.getBytes(StandardCharsets.UTF_8));
                out.flush();
            }
        } catch (IOException | InterruptedException gone) {
            // the listener closed the connection
        } finally {
            listeners.remove(q);
```

Listing 4.2 (Continued) *FeedServer.java: the same server with com.sun.net.httpserver, using one blocking queue per listener.*

```
            System.out.printf("listener gone (%d open)%n", listeners.size());
            ex.close();
        }
    }

    public static void main(String[] args) throws IOException {
        int port = 7041;
        for (int k = 0; k + 1 < args.length; k += 2) {
            if (args[k].equals("--port")) port = Integer.parseInt(args[k + 1]);
            if (args[k].equals("--heartbeat"))
                heartbeat = Double.parseDouble(args[k + 1]);
        }
        page = Files.exists(Path.of("web/feed.html"))
                ? Path.of("web/feed.html") : Path.of("../web/feed.html");
        HttpServer server =
                HttpServer.create(new InetSocketAddress("127.0.0.1", port), 0);
        server.createContext("/", ex -> {
            try { handle(ex); } catch (IOException e) { ex.close(); }
        });
        server.setExecutor(Executors.newCachedThreadPool()); // thread per stream
        server.start();
        System.out.println("feed server on http://127.0.0.1:" + port + "/");
    }
}
```

Java needs a JSON reader and writer for the request bodies and the answers. Rather than add a library, **Listing 4.3** is a small helper that handles exactly what the lab uses (flat objects of strings, numbers, booleans and `null`, and lists of strings) and writes JSON character for character as Python's `json.dumps` with compact separators does, so that the two servers give byte-identical answers.

Listing 4.3 `MiniJson.java`: just enough JSON for the lab, compatible with Python's output.

```java
import java.util.LinkedHashMap;
import java.util.List;
import java.util.Map;

/** Just enough JSON for this lab: flat objects of strings, numbers,
 *  booleans and null, plus lists of strings. Output matches Python's
 *  json.dumps(obj, separators=(",", ":")) character for character. */
public final class MiniJson {
    private final String s;
    private int i;

    private MiniJson(String s) { this.s = s; }

    /** Parse one flat JSON object such as {"id":3,"text":"hi"}. */
    public static Map<String, Object> parseObject(String text) {
        MiniJson p = new MiniJson(text.trim());
        Map<String, Object> map = new LinkedHashMap<>();
        p.expect('{');
        p.skip();
        if (p.peek() == '}') { p.i++; return map; }
        while (true) {
            p.skip();
            String key = p.string();
            p.skip();
            p.expect(':');
            p.skip();
            map.put(key, p.value());
            p.skip();
            char c = p.s.charAt(p.i++);
            if (c == '}') return map;
            if (c != ',') throw new IllegalArgumentException("bad JSON");
        }
    }

    private Object value() {
        char c = peek();
        if (c == '"') return string();
        if (s.startsWith("null", i)) { i += 4; return null; }
        if (s.startsWith("true", i)) { i += 4; return Boolean.TRUE; }
        if (s.startsWith("false", i)) { i += 5; return Boolean.FALSE; }
        int start = i;
        while (i < s.length() && "-0123456789".indexOf(s.charAt(i)) >= 0) i++;
        if (start == i) throw new IllegalArgumentException("bad JSON value");
        return Long.parseLong(s.substring(start, i));
    }
```

Listing 4.3 (Continued) *MiniJson.java: just enough JSON for the lab, compatible with Python's output.*

```
    private String string() {
        expect('"');
        StringBuilder out = new StringBuilder();
        while (true) {
            char c = s.charAt(i++);
            if (c == '"') return out.toString();
            if (c != '\\') { out.append(c); continue; }
            char e = s.charAt(i++);
            switch (e) {
                case 'n' -> out.append('\n');
                case 't' -> out.append('\t');
                case 'r' -> out.append('\r');
                case 'b' -> out.append('\b');
                case 'f' -> out.append('\f');
                case 'u' -> {
                    out.append((char) Integer.parseInt(s.substring(i, i + 4), 16));
                    i += 4;
                }
                default -> out.append(e);          // \" \\ \/
            }
        }
    }

    private char peek() {
        if (i >= s.length()) throw new IllegalArgumentException("bad JSON");
        return s.charAt(i);
    }

    private void skip() {
        while (i < s.length() && Character.isWhitespace(s.charAt(i))) i++;
    }

    private void expect(char c) {
        if (peek() != c) throw new IllegalArgumentException("expected " + c);
        i++;
    }
```

Listing 4.3 (Continued) *MiniJson.java: just enough JSON for the lab, compatible with Python's output.*

```
    /** Write a value the way Python's json.dumps does (ASCII only). */
    public static String write(Object v) {
        if (v == null) return "null";
        if (v instanceof Boolean || v instanceof Number) return v.toString();
        if (v instanceof List<?> list) {
            StringBuilder out = new StringBuilder("[");
            for (Object item : list) {
                if (out.length() > 1) out.append(',');
                out.append(write(item));
            }
            return out.append(']').toString();
        }
        if (v instanceof Map<?, ?> map) {
            StringBuilder out = new StringBuilder("{");
            for (Map.Entry<?, ?> e : map.entrySet()) {
                if (out.length() > 1) out.append(',');
                out.append(write(e.getKey().toString())).append(':')
                   .append(write(e.getValue()));
            }
            return out.append('}').toString();
        }
        StringBuilder out = new StringBuilder("\"");
        for (char c : v.toString().toCharArray()) {
            switch (c) {
                case '"' -> out.append("\\\"");
                case '\\' -> out.append("\\\\");
                case '\n' -> out.append("\\n");
                case '\r' -> out.append("\\r");
                case '\t' -> out.append("\\t");
                case '\b' -> out.append("\\b");
                case '\f' -> out.append("\\f");
                default -> {
                    if (c < 0x20 || c > 0x7f) {
                        out.append(String.format("\\u%04x", (int) c));
                    } else {
                        out.append(c);
                    }
                }
            }
        }
        return out.append('"').toString();
    }
}
```

4.12.4 The browser page

Listing 4.4 is the page both servers serve at `/`. It is plain HTML with a small script, no framework, and it implements four ideas of this chapter.

Listing 4.4 `feed.html`: the live page with channel badges, optimistic posting, one-item refetch and a stale-reply guard.

```
<!DOCTYPE html>
<html lang="en">
<head>
<meta charset="utf-8">
<meta name="viewport" content="width=device-width, initial-scale=1">
<title>Laundry Live Feed</title>
<style>
  body { font-family: system-ui, sans-serif; margin: 0; background: #f3f4f6;
         color: #12355b; }
  header { background: #12355b; color: #fff; padding: 12px 16px;
           display: flex; justify-content: space-between; }
  main { max-width: 640px; margin: 0 auto; padding: 16px; }
  nav button { border: 1px solid #0f7c7c; background: #fff; color: #0f7c7c;
               padding: 6px 12px; border-radius: 16px; margin-right: 6px; }
  nav button.active { background: #0f7c7c; color: #fff; }
  .badge { background: #c0392b; color: #fff; border-radius: 9px;
           padding: 0 6px; margin-left: 4px; font-size: 12px; }
  textarea { width: 100%; box-sizing: border-box; height: 64px; }
  .card { background: #fff; border-radius: 8px; padding: 10px 12px;
          margin: 8px 0; border-left: 4px solid #0f7c7c; }
  .card.pending { opacity: 0.6; border-left-color: #e0a100; }
  .card.reply { margin-left: 24px; border-left-color: #6b7280; }
  .meta { font-size: 12px; color: #6b7280; }
  #error { color: #c0392b; min-height: 1.2em; }
</style>
</head>
<body>
<header><strong id="title"></strong><span id="status"></span></header>
<main>
  <nav id="tabs"></nav>
  <h2 id="heading"></h2>
  <textarea id="text"></textarea>
  <button id="send"></button> <span id="error"></span>
  <div id="feed"></div>
</main>
<script>
const WORDS = {
  en: { title: "Profitina Laundry community (lab)", live: "live",
        wait: "reconnecting...", send: "Post", sending: "sending...",
        loading: "Loading...", fresh: "new", reply: "reply to #" },
  id: { title: "Komunitas Profitina Laundry (lab)", live: "langsung",
        wait: "menyambung ulang...", send: "Kirim", sending: "mengirim...",
        loading: "Memuat...", fresh: "baru", reply: "balasan untuk #" } };
const params = new URLSearchParams(location.search);
const T = WORDS[params.get("lang")] || WORDS.en;
const me = params.get("me") ||
```

Listing 4.4 (Continued) *feed.html: the live page with channel badges, optimistic posting, one-item refetch and a stale-reply guard.*

```
  "u" + Math.floor(Math.random() * 2 ** 32).toString(16).padStart(8, "0");
const slow = params.get("slow") || "";      // e.g. "tips:1500" for a demo
const unread = {}, known = new Set();
let active = "general", seq = 0, aborter = null, tempId = 0;
const $ = (id) => document.getElementById(id);

function card(post, pending) {
  const div = document.createElement("div");
  div.className = "card" + (pending ? " pending" : "") +
                  (post.reply_to ? " reply" : "");
  div.dataset.id = post.id;
  const meta = document.createElement("div");
  meta.className = "meta";
  meta.textContent = post.handle + (post.reply_to ?
    " - " + T.reply + post.reply_to : "") + (pending ? " - " + T.sending : "");
  const body = document.createElement("div");
  body.textContent = post.text;             // never innerHTML for user text
  div.append(meta, body);
  return div;
}

function drawTabs(names) {
  $("tabs").replaceChildren(...names.map((name) => {
    const b = document.createElement("button");
    b.textContent = name;
    b.className = name === active ? "active" : "";
    if (unread[name]) {
      const s = document.createElement("span");
      s.className = "badge";
      s.textContent = unread[name] + " " + T.fresh;
      b.append(s);
    }
    b.onclick = () => openChannel(name);
    return b;
  }));
}
```

Listing 4.4 (Continued) *feed.html: the live page with channel badges, optimistic posting, one-item refetch and a stale-reply guard.*

```
async function openChannel(name) {
  active = name;
  unread[name] = 0;
  drawTabs(channels);
  $("heading").textContent = "# " + name;
  $("feed").textContent = T.loading;          // skeleton at once
  const mine = ++seq;                         // sequence number of this request
  if (aborter) aborter.abort();               // cancel the previous request
  aborter = new AbortController();
  const [slowName, ms] = slow.split(":");
  const delay = slowName === name ? ms : 0;
  try {
    const r = await fetch("/feed?channel=" + name + "&delay=" + delay,
                          { signal: aborter.signal });
    const data = await r.json();
    if (mine !== seq || data.channel !== active) return;   // stale reply
    $("feed").replaceChildren(...data.posts.map((p) => card(p, false)));
    data.posts.forEach((p) => known.add(p.id));
  } catch (e) {
    if (e.name !== "AbortError") $("feed").textContent = String(e);
  }
}

async function onEvent(note) {
  if (note.channel !== active) {              // invalidate: just count it
    if (note.top) unread[note.channel] = (unread[note.channel] || 0) + 1;
    drawTabs(channels);
    return;
  }
  if (known.has(note.id)) return;           // our own post, already shown
  const r = await fetch("/post/" + note.id); // refetch exactly one item
  const post = await r.json();
  if (post.channel !== active || known.has(post.id)) return;
  known.add(post.id);
  $("feed").prepend(card(post, false));
}
```

Listing 4.4 (Continued) *feed.html: the live page with channel badges, optimistic posting, one-item refetch and a stale-reply guard.*

```
async function send() {
  const text = $("text").value;
  if (!text.trim()) return;
  $("error").textContent = "";
  const draft = { id: "tmp" + (++tempId), channel: active, handle: me,
                  text: text, reply_to: null };
  const node = card(draft, true);
  $("feed").prepend(node);                  // optimistic: show it now
  $("text").value = "";                     // and clear the composer
  try {
    const r = await fetch("/post", { method: "POST",
      headers: { "Content-Type": "application/json" },
      body: JSON.stringify({ channel: draft.channel, handle: me,
                             text: text }) });
    const answer = await r.json();
    if (!r.ok) throw new Error(answer.error);
    known.add(answer.id);
    if ($("feed").querySelector('[data-id="' + answer.id + '"]')) {
      node.remove();                        // the live event was faster
    } else {
      node.replaceWith(card({ ...draft, id: answer.id }, false));
    }
  } catch (e) {
    node.remove();                          // roll back the optimistic card
    $("text").value = text;                 // give the text back
    $("error").textContent = e.message;
  }
}

let channels = [];
async function start() {
  document.title = T.title;
  $("title").textContent = T.title;
  $("send").textContent = T.send;
  $("send").onclick = send;
  $("text").oninput = () => { $("error").textContent = ""; };
  channels = await (await fetch("/channels")).json();
  openChannel("general");
  const source = new EventSource("/events");  // reconnects by itself
  source.onopen = () => { $("status").textContent = T.live; };
  source.onerror = () => { $("status").textContent = T.wait; };
  source.addEventListener("post", (e) => onEvent(JSON.parse(e.data)));
}
start();
</script>
</body>
```

Listing 4.4 (Continued) *feed.html: the live page with channel badges, optimistic posting, one-item refetch and a stale-reply guard.*

```
</html>
```

- **The stream.** `new EventSource("/events")` opens the stream; its `onopen` and `onerror` handlers show *live* or *reconnecting…* in the header. Reconnection needs no code at all.
- **Invalidate or refetch.** `onEvent` follows **Figure 4.8**: an event for another channel only raises that channel's badge (if `top`); an event for the channel on screen fetches exactly one post with `/post/n`.
- **Optimistic posting.** `send` draws a pending card and clears the composer *before* the request; on success it replaces the card with the confirmed post, and on failure it removes the card and restores the text, as in **Figure 4.7**.

- **Stale-reply guard.** `openChannel` increments `seq` and aborts the previous request; when an answer arrives it is used only if its sequence number is still the latest and its channel is still active.

Notice also `body.textContent = post.text`: user text is always inserted as text, never as HTML, so that a post containing `<script>` shows those characters instead of running them. Chapter 5 returns to this idea from the safety side.

4.12.5 The console clients

Listing 4.5 and **Listing 4.6** are the console clients. `listen` reads the stream one line at a time, collects `field: value` lines into an event, and on the blank line that ends an event fetches the one post it announces; `post` sends a post and prints the server's status code and answer.

Listing 4.5 `feed_client.py`: listening to the stream and posting, in Python.

```
"""Console client for the live feed: listen to the stream, or send a post.

  python feed_client.py listen [--max 5] [--last-id 0]
  python feed_client.py post --text "Hello" [--channel general] [--reply-to 1]
Add --url http://127.0.0.1:7041 to talk to the Java server instead.
"""
import argparse
import json
import urllib.error
import urllib.request


def fetch_post(base, post_id):
    with urllib.request.urlopen("%s/post/%d" % (base, post_id)) as reply:
        return json.load(reply)


def show(post):
    where = " (reply to #%d)" % post["reply_to"] if post["reply_to"] else ""
    print("#%d [%s] %s%s: %s" % (post["id"], post["channel"], post["handle"],
                                 where, post["text"]), flush=True)


def listen(base, max_events, last_id):
    request = urllib.request.Request(base + "/events")
    request.add_header("Accept", "text/event-stream")
    if last_id:
        request.add_header("Last-Event-ID", str(last_id))   # ask for a replay
    seen = 0
    event = {}
    with urllib.request.urlopen(request) as stream:
        for raw in stream:                      # one line at a time
            line = raw.decode("utf-8").rstrip("\r\n")
            if line.startswith(":"):
                print("(heartbeat)", flush=True)
            elif line.startswith("retry:"):
                print("server asks us to reconnect after",
                      line[6:].strip(), "ms", flush=True)
            elif line and ":" in line:
                field, _, value = line.partition(":")
                event[field] = value.lstrip(" ")
            elif not line and "data" in event:   # blank line ends an event
                note = json.loads(event["data"])
                show(fetch_post(base, note["id"]))   # refetch one item
                event = {}
                seen += 1
```

Listing 4.5 (Continued) *feed_client.py: listening to the stream and posting, in Python.*

```
                if seen == max_events:
                    return


def send(base, channel, handle, text, reply_to):
    body = {"channel": channel, "handle": handle, "text": text,
            "reply_to": reply_to}
    request = urllib.request.Request(
        base + "/post", data=json.dumps(body).encode("utf-8"),
        headers={"Content-Type": "application/json"}, method="POST")
    try:
        with urllib.request.urlopen(request) as reply:
            print(reply.status, reply.read().decode("utf-8"))
    except urllib.error.HTTPError as err:
        print(err.code, err.read().decode("utf-8"))


def main():
    parser = argparse.ArgumentParser()
    parser.add_argument("command", choices=["listen", "post"])
    parser.add_argument("--url", default="http://127.0.0.1:7040")
    parser.add_argument("--max", type=int, default=5)
    parser.add_argument("--last-id", type=int, default=0)
    parser.add_argument("--channel", default="general")
    parser.add_argument("--handle", default="u3f9a1c20")
    parser.add_argument("--text", default="")
    parser.add_argument("--reply-to", type=int, default=None)
    args = parser.parse_args()
    if args.command == "listen":
        listen(args.url, args.max, args.last_id)
    else:
        send(args.url, args.channel, args.handle, args.text, args.reply_to)


if __name__ == "__main__":
    main()
```

Listing 4.6 `FeedClient.java`: the same client with `java.net.http.HttpClient` and `BodyHandlers.ofLines()`.

```
import java.io.IOException;
import java.net.URI;
import java.net.http.HttpClient;
import java.net.http.HttpRequest;
import java.net.http.HttpResponse;
import java.util.HashMap;
import java.util.Iterator;
import java.util.LinkedHashMap;
import java.util.Map;
import java.util.stream.Stream;

/** Console client for the live feed: listen to the stream, or send a post.
 *    java -cp out FeedClient listen [--max 5] [--last-id 0]
 *    java -cp out FeedClient post --text "Hello" [--channel general]
 *  Add --url http://127.0.0.1:7040 to talk to the Python server instead. */
public class FeedClient {
    static final HttpClient HTTP = HttpClient.newBuilder()
            .version(HttpClient.Version.HTTP_1_1).build();

    static Map<String, Object> fetchPost(String base, long id)
            throws IOException, InterruptedException {
        HttpRequest req = HttpRequest.newBuilder(
                URI.create(base + "/post/" + id)).build();
        return MiniJson.parseObject(
                HTTP.send(req, HttpResponse.BodyHandlers.ofString()).body());
    }

    static void show(Map<String, Object> post) {
        Object reply = post.get("reply_to");
        String where = reply != null ? " (reply to #" + reply + ")" : "";
        System.out.println("#" + post.get("id") + " [" + post.get("channel") + "] "
                + post.get("handle") + where + ": " + post.get("text"));
    }

    static void listen(String base, int maxEvents, int lastId)
            throws IOException, InterruptedException {
        HttpRequest.Builder b = HttpRequest
                .newBuilder(URI.create(base + "/events"))
                .header("Accept", "text/event-stream");
        if (lastId > 0) b.header("Last-Event-ID", String.valueOf(lastId));
        HttpResponse<Stream<String>> resp =
                HTTP.send(b.build(), HttpResponse.BodyHandlers.ofLines());
        int seen = 0;
        Map<String, String> event = new HashMap<>();
        try (Stream<String> lines = resp.body()) {
            Iterator<String> it = lines.iterator();       // one line at a time
```

Listing 4.6 (Continued) *FeedClient.java: the same client with java.net.http.HttpClient and BodyHandlers.ofLines().*

```
            while (it.hasNext()) {
                String line = it.next();
                if (line.startsWith(":")) {
                    System.out.println("(heartbeat)");
                } else if (line.startsWith("retry:")) {
                    System.out.println("server asks us to reconnect after "
                            + line.substring(6).strip() + " ms");
                } else if (!line.isEmpty() && line.contains(":")) {
                    int colon = line.indexOf(':');
                    String value = line.substring(colon + 1);
                    event.put(line.substring(0, colon),
                              value.startsWith(" ") ? value.substring(1) : value);
                } else if (line.isEmpty() && event.containsKey("data")) {
                    Map<String, Object> note =
                            MiniJson.parseObject(event.get("data"));
                    show(fetchPost(base, (Long) note.get("id")));  // refetch one
                    event.clear();
                    if (++seen == maxEvents) return;
                }
            }
        }
    }

    static void send(String base, String channel, String handle, String text,
                     Long replyTo) throws IOException, InterruptedException {
        Map<String, Object> body = new LinkedHashMap<>();
        body.put("channel", channel);
        body.put("handle", handle);
        body.put("text", text);
        body.put("reply_to", replyTo);
        HttpRequest req = HttpRequest.newBuilder(URI.create(base + "/post"))
                .header("Content-Type", "application/json")
                .POST(HttpRequest.BodyPublishers.ofString(MiniJson.write(body)))
                .build();
        HttpResponse<String> resp =
                HTTP.send(req, HttpResponse.BodyHandlers.ofString());
        System.out.println(resp.statusCode() + " " + resp.body());
    }
```

Listing 4.6 (Continued) *FeedClient.java: the same client with java.net.http.HttpClient and BodyHandlers.ofLines().*

```
    public static void main(String[] args) throws Exception {
        Map<String, String> opt = new HashMap<>(Map.of(
                "--url", "http://127.0.0.1:7041", "--max", "5",
                "--last-id", "0", "--channel", "general",
                "--handle", "u3f9a1c20", "--text", ""));
        for (int k = 1; k + 1 < args.length; k += 2) {
            opt.put(args[k], args[k + 1]);          // --name value pairs
        }
        String command = args.length > 0 ? args[0] : "";
        if (command.equals("listen")) {
            listen(opt.get("--url"), Integer.parseInt(opt.get("--max")),
                   Integer.parseInt(opt.get("--last-id")));
        } else if (command.equals("post")) {
            String reply = opt.get("--reply-to");
            send(opt.get("--url"), opt.get("--channel"), opt.get("--handle"),
                 opt.get("--text"), reply == null ? null : Long.parseLong(reply));
        } else {
            System.out.println("usage: FeedClient listen|post [options]");
        }
    }
}
```

4.12.6 Running it

Open three terminals in the `Chapter04` folder. In the first, start the Python server with a short two-second heartbeat so that we can see one:

```
cd python
python feed_server.py --heartbeat 2
```

In the second, start a listener that stops after four events:

```
python python/feed_client.py listen --max 4
```

In the third, send five posts, one of which uses an invalid handle:

```
python python/feed_client.py post --channel general --text "Machine 3 is fixed; express
orders are back."
python python/feed_client.py post --channel tips --handle u7d20be41 --text "Tip: wash
batik inside out in cold water."
python python/feed_client.py post --channel general --handle budi --text "Hello!"
python python/feed_client.py post --channel general --handle u7d20be41 --reply-to 1 --
text "Great news, thanks!"
python python/feed_client.py post --channel promo --text "Weekend promo: 10% off bed
covers."
```

The third terminal prints the server's answers:

```
201 {"id":1}
201 {"id":2}
422 {"error":"handle must be neutral, like u3f9a1c20"}
201 {"id":3}
201 {"id":4}
```

The listener prints:

```
server asks us to reconnect after 3000 ms
(heartbeat)
#1 [general] u3f9a1c20: Machine 3 is fixed; express orders are back.
#2 [tips] u7d20be41: Tip: wash batik inside out in cold water.
#3 [general] u7d20be41 (reply to #1): Great news, thanks!
#4 [promo] u3f9a1c20: Weekend promo: 10% off bed covers.
```

Every line deserves a comment. The first line is the `retry` field. `(heartbeat)` is the `: ping` comment, sent because two seconds passed with no post. Then come the four accepted posts, each fetched individually after its thin event; the refused post never reached the stream. After the fourth event the listener exits.

Now pretend that a listener was disconnected after event 2 and reconnects, as a browser would, with `Last-Event-ID: 2`:

```
python python/feed_client.py listen --max 2 --last-id 2
```

```
server asks us to reconnect after 3000 ms
#3 [general] u7d20be41 (reply to #1): Great news, thanks!
#4 [promo] u3f9a1c20: Weekend promo: 10% off bed covers.
```

The server replayed exactly the two posts the listener had missed. The first terminal's log tells the whole story from the server's side:

```
feed server on http://127.0.0.1:7040/
listener connected, last id 0 (1 open)
post #1 [general] by u3f9a1c20
post #2 [tips] by u7d20be41
rejected: handle must be neutral, like u3f9a1c20
post #3 [general] by u7d20be41
post #4 [promo] by u3f9a1c20
listener gone (0 open)
listener connected, last id 2 (1 open)
listener gone (0 open)
```

Note the line `listener gone (0 open)`: the server noticed that the first listener had left only when a heartbeat failed to be written. Without heartbeats, a server can hold connections to readers that vanished long ago. Stop the server with **Ctrl+C**.

The Java server and client take the same commands (`java -cp out FeedServer --heartbeat 2` in the `java` folder, and `java -cp java/out FeedClient ...` in place of `python python/feed_client.py ...`; the Java client's default address is the Java server's port 7041) and produce the same output. The only difference you may see is an extra `(heartbeat)` line, because starting a new Java virtual machine for each `post` command takes longer than two seconds on some machines. Every client works with either server: the Python client against the Java server gives exactly the output above.

4.12.7 Checking the page in a headless browser

Open `http://127.0.0.1:7040/` in a browser to try the page by hand. Open it in two windows side by side and post in one: the post appears in the other without reloading. To check the page automatically, the code folder includes `tools/check_page.py`, which uses Playwright [15] to drive a real Chromium browser without a window. It needs `pip install playwright` and `python -m playwright install chromium` once, and a freshly started server. It holds every POST for half a second to imitate a slow network, then checks eight things:

```
PASS optimistic card shown at once as pending
PASS composer cleared at once
PASS card confirmed by the server
PASS failed post rolled back, text restored
PASS event for the open channel fetched one post
PASS counter counts top-level posts only (1 new)
PASS late reply for 'tips' discarded
PASS no JavaScript errors
8 of 8 checks passed
```

The checks are the chapter's ideas made testable: the optimistic card appears at once and the composer clears; the server confirms it; a 300-character post is rolled back with its text restored; an event for the open channel fetches one post; a top-level post in another channel raises its badge to "1 new" while a reply does not; a slow reply for *tips* that arrives after the member has moved on to *promo* is thrown away; and no JavaScript error occurs. The same eight checks pass against the Java server. **Figure 4.12** is the page as the check leaves it.

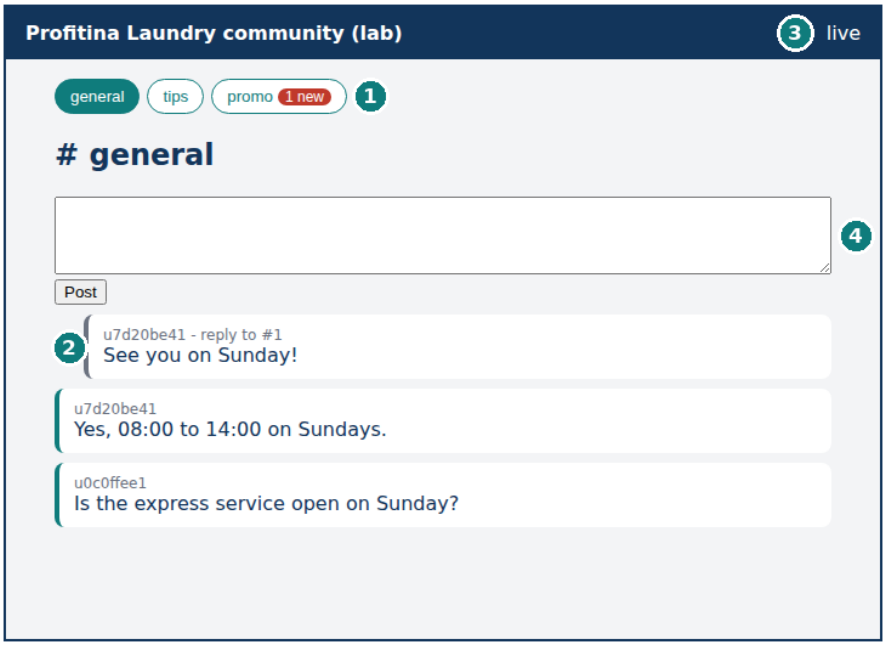


Figure 4.12 The lab page in headless Chromium after the automated check. Marker 1: the "1 new" badge of a channel not on screen; 2: a reply, indented; 3: the connection status; 4: the composer, cleared at once when a post is sent.

4.12.8 What the lab leaves out

The lab is honest about its limits, and naming them is part of the lesson. Posts live in memory and vanish when the server stops; a real system stores them in a database. There is no sign-in, so anyone may post under any valid handle. Each stream occupies one thread, which is fine for a classroom and wasteful for thousands of listeners. The replay keeps the whole history; a real server would keep a bounded window. And nothing screens the links in posts; that is the next chapter's lab.

4.13 Profitina in Practice

The ideas of this chapter can be seen at work in the live application at `app.profitina.com`.

Open the community and watch. The community tab updates while you read; nothing needs reloading. Open a thread with the hash route `#/komunitas/post/{id}` and you are looking at the tree of Section 4.5.1. On profitina.com, the same feed appears in the embedded widget, read-only for visitors.

Follow two channels. The red badges count top-level posts only, by Equation (4.6). Reposts land in the original's channel unless you choose General in the repost dialog.

Check the gate. The *Create channel* dialog explains the balance gate of Equation (4.5); a balance below 50 PROFIT is enough to see the refusal, and nothing is deducted when you succeed.

Look at your dashboard. The priority strip of **Figure 4.6** is the first thing you see after signing in; every item takes you to its section.

Live and roadmap. Everything in **Table 4.1** is live in the September 2026 release (version `0.14.5-testnet`). Group messages, @mentions and rewards for DMs are not planned; their absence is a design decision, not a missing feature. Questions about the community go to `info@profitina.com`.

4.14 Common Misunderstandings

"Real time needs WebSockets." Only when data must flow both ways many times a second. A feed is server-to-browser news plus occasional posts, and SSE over plain HTTP does that with built-in reconnection and replay (Section 4.4.4).

"Polling every second is almost as good." By Equation (4.1) it multiplies requests by five compared with five-second polling, and by Equation (4.3) almost every one of them returns nothing. It is not almost as good; it is much more expensive.

"An event should carry the whole post, so the page need not ask." A fat event is sent to every open page, including all those not showing that channel. Thin events plus one-item refetch send the full post only to the pages that display it (Section 4.11.2).

"The unread counter counts everything new." Profitina's counts new *top-level* posts, so that the number matches what is visible at the top of the feed (Section 4.7.2).

"The channel gate charges 50 PROFIT." It requires a balance of 50 PROFIT and deducts nothing (Section 4.6.2).

"Optimistic posting pretends that the post succeeded." It shows a pending prediction and corrects it within one round trip; on failure the text comes back with the reason (Section 4.10.2).

"If the server is fast, races cannot happen." Speed makes races rarer, not impossible; the late-reply race of **Figure 4.10** became *visible* only after the feed became fast. Guards are needed on every panel.

"Administrators can read DMs if they need to." They cannot. The only exception is a frozen copy of a message that its recipient reported (Section 4.8.1).

"A heartbeat is only for keeping proxies awake." It is also how the server discovers that a listener has gone, as the lab's log showed.

4.15 Chapter Summary

- Profitina's community serves the *Gather* and *Give* of its tagline: a live feed, threads and quote-reposts, channels, follows with "N new" badges, DMs, consent-based profiles and a dashboard cockpit, all built between July and September 2026 (**Table 4.1**).
- The web was built for asking. **Polling** costs $3600N/T$ requests per hour (Equation (4.1)), an average delay of $T/2$ (Equation (4.2)) and mostly empty answers (Equation (4.3)). **WebSockets** give a two-way pipe outside HTTP. **Server-Sent Events** keep one HTTP response open and push text events, with `retry`, `Last-Event-ID` replay and comment heartbeats built in (**Table 4.2**).
- A thread is a tree of replies. Reposts inherit the original's channel unless the member picks General. Posts can be edited for 15 minutes (Equation (4.4)), and the reward is recalculated.

- Channel creation is gated by holding at least 50 PROFIT, never deducted, with at most two new channels per day and five active (Equation (4.5)).
- The “N new” badge counts top-level posts after the last-seen marker (Equation (4.6)); it is kept live by thin events at no network cost. The same word must mean the same number everywhere.
- DMs are one-to-one, unrewarded and unreadable by administrators except for a frozen copy of a reported message. Profiles share each item only by its own consent, starting off; handles are neutral.
- The dashboard is one request that fills a three-level priority strip and three cards.
- Optimistic posting shows a pending post at once and rolls back with the text restored on failure.
- Three lessons from the September 2026 release: the read path never calls a model; events invalidate or fetch one item rather than reloading everything; queries must not grow with the page (Equation (4.7)), which took a feed page from 868 ms to 11 ms. A fourth: guard every panel against late replies.
- Lab G2 built all of this in miniature in Python and Java, with a browser page checked automatically in headless Chromium.

4.16 Exercises

Exercise 4.1. A course channel has 250 open pages. (a) Using Equation (4.1), how many requests per hour does polling every 10 seconds generate? (b) What is the average delay by Equation (4.2)? (c) If the channel receives 30 posts per hour at random, what fraction of polls is empty by Equation (4.3)?

Exercise 4.2. With SSE and the lab’s 15-second heartbeat, how many heartbeat comments per hour does the server write to the same 250 pages if nobody posts? Compare with your answer to Exercise 4.1(a) and explain why the two numbers do not measure the same kind of cost.

Exercise 4.3. Write down, byte for byte, the event that the lab server sends for post number 7, a reply in channel *tips*.

Exercise 4.4. A member’s last-seen marker for channel *kelas-oop* is 40. Posts 41 to 48 then arrive: 41, 43, 44 and 47 are top-level, the rest are replies. What does the badge show by Equation (4.6)? What would a badge that counted everything show, and why is that misleading?

Exercise 4.5. Budi holds 55 PROFIT, has created no channel today and has 5 active channels. Can he create a channel? What is the smallest change to his situation that would let him, and what happens to his balance if he then does?

Exercise 4.6. A post was created at 09:52:30 WIB. Using Equation (4.4), until what time may it be edited? The member edits it at 10:05 and removes the only useful sentence. What happens to its reward, and why is that the right rule?

Exercise 4.7. In the lab page, remove in your head both the line `if (aborter) aborter.abort();` and the line `if (mine !== seq || data.channel !== active) return;`. Describe exactly what a member would see in the scenario of **Figure 4.10**, and which of the eight automated checks would fail. Then put back only the `abort` line: is the page correct again, and why is the sequence check still worth keeping?

Exercise 4.8. Suppose rendering one post costs 11 database queries and 2 ms of database time per query, and a page shows n posts. (a) Write the page’s database time before prefetching as a

function of n. (b) If prefetching uses 12 queries regardless of n, from which n onwards is prefetching faster? (c) Why does a new connection per query make things much worse than (a) suggests?

Exercise 4.9. The lab replays the whole history after `Last-Event-ID`. Propose a change that keeps only the last 100 posts in memory for replay, and say what the server should do when a listener asks for a replay older than that window.

Exercise 4.10. Explain why DMs are one-to-one, unrewarded and unreadable by administrators, and why the one exception (a frozen copy of a reported message) does not break the promise of privacy.

Exercise 4.11. Run the lab with the Java server on port 7041 and the Python client (`--url http://127.0.0.1:7041`). Then kill the server while a listener is connected. What does the Python client do? What would a browser's `EventSource` do instead, and which field of the stream controls it?

Exercise 4.12. Design question. A member asks for "@mentions in DMs so I can add a third person". Using the principles of Sections 4.2.2 and 4.8.1, write a short, polite answer explaining why the platform does not offer it and where such a conversation belongs instead.

4.17 Solutions to the Exercises

Solution 4.1. (a) $R = 3600 \times 250/10 = 90{,}000$ requests per hour, 25 per second. (b) $T/2 = 5$ seconds on average, up to 10 seconds. (c) $\lambda = 30/3600 = 1/120$ per second, so $\lambda T = 10/120 \approx 0.0833$ and $P(\text{empty}) = e^{-0.0833} \approx 0.920$: about 92 % of polls return nothing.

Solution 4.2. One heartbeat per page every 15 seconds is $3600/15 = 240$ per hour per page, so $240 \times 250 = 60{,}000$ comments per hour. The number is of the same order as the 90,000 polls, but the cost is very different: a heartbeat is about eight bytes written to an already open connection, with no request to parse, no authentication and no database query, whereas each poll is a full HTTP request and response. Counting messages is not the same as counting work.

Solution 4.3. The `thin_event` function writes

```
id: 7
event: post
data: {"id":7,"channel":"tips","top":false}
```

with a blank line at the end; `top` is `false` because the post is a reply.

Solution 4.4. Posts with numbers greater than 40 that are top-level: 41, 43, 44 and 47, so the badge shows "4 new". Counting everything would show "8 new", but opening the channel would show only four new posts at the top; the other four are replies inside threads, so the badge would promise more than the member can see.

Solution 4.5. No: $55 \geq 50$ and $0 < 2$ hold, but $5 < 5$ fails. The smallest change is to make one of his channels inactive, so that $a = 4$. Then all three conditions of Equation (4.5) hold, he can create the channel, and his balance stays 55 PROFIT because the gate never deducts.

Solution 4.6. Until 10:07:30 WIB, fifteen minutes after 09:52:30. The edit at 10:05 is allowed. Its reward is recalculated in full from the new text and will go down, possibly to zero. Otherwise a member could earn a reward for a useful post and then replace it with anything; recalculation keeps the reward tied to what readers actually see.

Solution 4.7. The member clicks *tips* (slow), then *promo*. Promo's posts appear under the heading "# promo". About 1.5 seconds later the tips answer arrives; with neither guard it replaces the feed, so the screen shows tips posts under the heading "# promo", and the check "late reply for 'tips' discarded" fails. With the `abort` line restored, the tips request is cancelled when *promo* is clicked, its `fetch` rejects with an `AbortError` that the page ignores, and the check passes again. The sequence check is still worth keeping because aborting depends on timing: an answer that has already arrived and is being parsed when the second click happens cannot be aborted any more, and other code paths (a retry button, a cached answer) may start requests that no `AbortController` knows about. The sequence number is the guard that holds whatever the network does.

Solution 4.8. (a) $(1 + 11n) \times 2 = 2 + 22n$ ms. (b) Prefetching costs $12 \times 2 = 24$ ms; $2 + 22n > 24$ when $n > 1$, so from $n = 2$ onwards prefetching is faster, and for $n = 30$ it is 662 ms against 24 ms. (c) Opening a connection involves a network handshake, authentication and process setup on the database side, which usually costs far more than a simple query; paying it for every query multiplies the total, which is why the measured time (868 ms) was far above what query time alone suggests and why the pool alone removed 122 connections per page.

Solution 4.9. Keep the posts in a bounded structure, for example a `collections.deque(maxlen=100)` in Python or an `ArrayDeque` trimmed to 100 in Java, together with the number of the oldest post kept. If a listener asks for `Last-Event-ID` older than that number, the server cannot replay exactly; it should send an event such as `event: reset` telling the client to reload the channel with an ordinary `GET /feed`, rather than silently replaying an incomplete history.

Solution 4.10. One-to-one keeps DMs from becoming unmoderated group channels, where spam and pile-ons spread; public discussion belongs in channels that have moderators. No rewards, because nobody can check whether a private message was useful, so rewards would invite farming. Unreadable by administrators, because a private message that the operator can read is not private. The frozen copy of a *reported* message is created only by the recipient's own act of reporting, exposes only that message, and freezes it so that it cannot be altered afterwards; the rest of the conversation stays private.

Solution 4.11. The Python client's `urlopen` stream ends (or raises a connection error) and the program exits; it has no reconnection logic. A browser's `EventSource` would wait for the `retry` interval, 3000 ms here, and reconnect automatically, sending `Last-Event-ID` with the last `id` it received; when the server comes back it replays what was missed (in the lab, only posts still in memory, which after a restart means none).

Solution 4.12. One possible answer: "Thank you for the suggestion. DMs on Profitina are deliberately one-to-one: they are private conversations that nobody else, including the platform, can read, and that is only manageable between two people. A conversation among three or more is a small community, and communities work best in a channel, where a moderator can pin, hide and mute and where the rules are visible to everyone. You are welcome to create a channel for your group once your balance meets the gate; nothing is deducted."

References

[1] Profitina, *Profitina Whitepaper — Public Version v24*, Sep. 2026. [Online]. Available: https://app.profitina.com/static/whitepaper/

[2] ANTARA, “Survei APJII: Penetrasi internet di Indonesia 2026 capai 81,7 persen,” May 21, 2026. [Online]. Available: https://www.antaranews.com/berita/5576225/survei-apjii-penetrasi-internet-di-indonesia-2026-capai-817-persen

[3] S. Kemp, “Digital 2026: Indonesia,” DataReportal, Nov. 5, 2025. [Online]. Available: https://datareportal.com/reports/digital-2026-indonesia

[4] I. Fette and A. Melnikov, “The WebSocket Protocol,” IETF RFC 6455, Dec. 2011.

[5] WHATWG, “Server-sent events,” in *HTML Living Standard*, sec. 9.2. [Online]. Available: https://html.spec.whatwg.org/multipage/server-sent-events.html

[6] MDN Web Docs, “Using server-sent events,” Mozilla. [Online]. Available: https://developer.mozilla.org/en-US/docs/Web/API/Server-sent_events/Using_server-sent_events

[7] Profitina, “Kebijakan Privasi” and “Syarat dan Ketentuan” (Privacy Policy and Terms and Conditions, v1.1, effective Aug. 6, 2026). [Online]. Available: https://app.profitina.com/legal/privacy

[8] Republik Indonesia, *Undang-Undang Nomor 27 Tahun 2022 tentang Pelindungan Data Pribadi*, Oct. 17, 2022.

[9] J. Nielsen, “Response Times: The 3 Important Limits,” Nielsen Norman Group, 1993 (updated 2014). [Online]. Available: https://www.nngroup.com/articles/response-times-3-important-limits/

[10] MDN Web Docs, “AbortController,” Mozilla. [Online]. Available: https://developer.mozilla.org/en-US/docs/Web/API/AbortController

[11] Python Software Foundation, “http.server — HTTP servers,” *The Python Standard Library*. [Online]. Available: https://docs.python.org/3/library/http.server.html

[12] Oracle, “Package com.sun.net.httpserver,” *Java SE 25 & JDK 25 API Specification*, 2025. [Online]. Available: https://docs.oracle.com/en/java/javase/25/docs/api/jdk.httpserver/com/sun/net/httpserver/package-summary.html

[13] Oracle, “Class HttpClient,” *Java SE 25 & JDK 25 API Specification*, 2025. [Online]. Available: https://docs.oracle.com/en/java/javase/25/docs/api/java.net.http/java/net/http/HttpClient.html

[14] R. Fielding, M. Nottingham, and J. Reschke, “HTTP/1.1,” IETF RFC 9112, Jun. 2022.

[15] Microsoft, “Playwright for Python.” [Online]. Available: https://playwright.dev/python/

Chapter 5

Keeping the Community Safe: Link Safety, Moderation, Fair Play

By the end of this chapter you will be able to take a link apart and point to the places where scams hide in it; explain why Profitina screens links in layers, with a deterministic rule floor at the bottom and an optional local-AI opinion on top, and why neither layer ever runs while someone is waiting to read the feed; measure any screener honestly with a confusion matrix, precision, recall, F1 and the false positive rate, and say how much a small benchmark can and cannot prove; describe what a channel moderator may and may not do, and why those limits are the point; and explain, at the level of boxes and arrows, how a reward system keeps its rewards for humans. In the lab you will write your own rule-based link screener in Python and Java, with rules of your own choosing, and evaluate it on a labeled sample set that you can read line by line, ending with a confusion matrix, a precision–recall trade-off and an analysis of every mistake.

5.1 Where We Left Off

Chapter 4 built a community that updates by itself: posts appear the moment they are written, counters climb while the page is open, and a thread grows under the eyes of everyone reading it. Every one of those properties is a gift to a scammer. A link posted in a live feed reaches every open page within a second. A message that looks like an announcement from staff is read by hundreds of people before anyone questions it. A reward for posting is, to a program, simply a function to call as often as possible.

This chapter is about the three defenses that make the community of Chapter 4 safe to use. The first is **link safety**: every link in a post is screened before anyone can click it. The second is **moderation**: people with narrow, visible powers look after the rooms they are responsible for. The third is **fair play**: the reward system is designed so that it pays humans for useful contributions and pays bots nothing. The chapter takes them in that order, and spends longest on the first, because it is the one you can build and measure yourself in an afternoon.

As in Chapter 4, Profitina is described at box level: which layers exist, in what order, and why. Its actual rules, weights, thresholds and prompts are its own. The lab screener is ours, built from textbook rules that anyone can find in the security literature, and it is evaluated on a sample set written for this book.

5.2 Why a Community Needs Protecting

5.2.1 Three numbers

Figure 5.1 puts three recent numbers side by side. Each measures something different, and together they describe the environment any Indonesian community platform lives in.



Figure 5.1 Three measures of the threat environment in 2026: phishing attacks observed worldwide in one quarter, scam reports received by Indonesia's Anti-Scam Centre since it opened, and the share of web traffic that is automated.

- **Phishing keeps growing.** The Anti-Phishing Working Group (APWG) observed 971,181 phishing attacks in the first quarter of 2026, 13.8 % more than the 853,244 of the fourth quarter of 2025. Its report, published on 27 May 2026, also found threat volume rising on every social media platform it tracked, with scams making up 27.1 % and impersonation 43.8 % of the threats recorded there, and voice and SMS phishing together up 15 % in one quarter [1].
- **Indonesians are being targeted at scale.** The Indonesia Anti-Scam Centre (IASC), run by the financial services regulator OJK with banks and other institutions, received 636,014 scam reports between its launch on 22 November 2024 and 30 July 2026. In that period it blocked 604,923 accounts holding Rp723.7 billion and returned Rp204.3 billion to victims [2].
- **Much of the web is not human.** Imperva's *Bad Bot Report 2026*, published on 29 April 2026, found that automated traffic made up 53 % of web traffic in 2025, overtaking human traffic (47 %) [3].

A community platform sits exactly where these three meet. Its members are ordinary Indonesians, many of them small-business owners who handle payments every day; its feed is a place where a link can reach hundreds of people in seconds; and its rewards are an invitation to automation.

5.2.2 Three threats, three defenses

It helps to name the threats precisely, because each calls for a different defense.

1. **Malicious links.** A post carries a link to a page that steals a password, an OTP code, a wallet or a card number, or that installs something harmful. The defense is *link safety*: screen the link before it becomes clickable.
2. **Abuse by people.** Harassment, impersonation of staff, off-topic flooding, scams without any link ("send me the OTP code we just texted you"). Software catches some of this; people catch the rest. The defense is *moderation* with narrow, logged powers, and *reporting* that any member can use.
3. **Abuse by automation.** Scripts that post, like and refer in bulk to farm rewards. The defense is *fair play*: make sure that only verified humans earn, that trivial or repeated contributions earn nothing, and that abuse found later is taken back.

No single defense covers all three. A perfect link screener does nothing against the no-link OTP scam; the best moderator cannot read a thousand posts a minute; and fair play protects the rewards, not the readers. The rest of the chapter takes them one at a time.

5.3 Anatomy of a Link, and Where the Tricks Hide

5.3.1 The parts of a URL

A web address, or URL, has a structure defined in RFC 3986 [4]. For our purposes five parts matter:

- the **scheme**, such as `https`, which says how to fetch the resource;
- the **site part**, everything between `//` and the next `/`, `?` or `#`, which may contain *user information* ending in `@`, then the **host**, then an optional `:port`;
- the **host**, the name of the machine, made of *labels* separated by dots, the last of which is the **top-level domain** (TLD) such as `com` or `id`;
- the **path**, which names a resource on that machine;
- optionally a *query* after `?` and a *fragment* after `#`.

The single most important fact for link safety is that **only the host decides which organization's server you reach**. Everything else, including words that look like brand names, can be chosen freely by whoever sets up the page. **Figure 5.2** shows a link built to exploit exactly that.

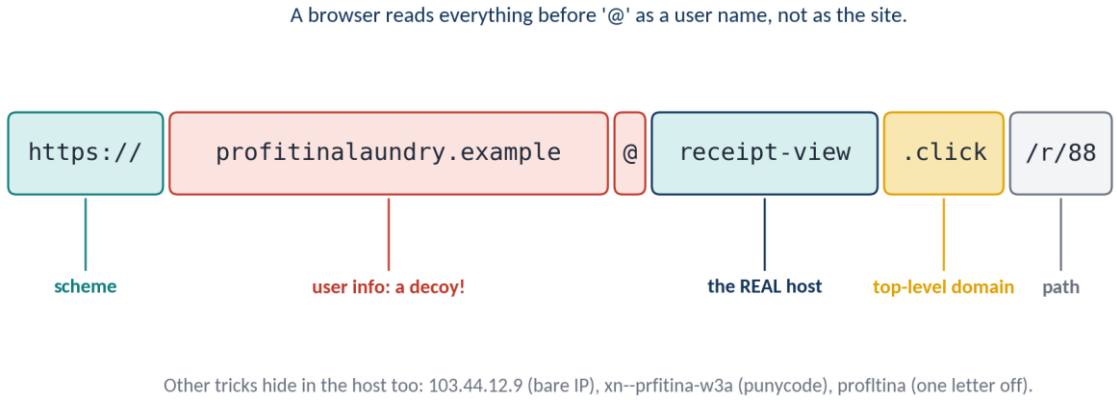


Figure 5.2 A link that hides its real host. Everything before `@` is user information, a decoy; the host is `receipt-view.click`. The bottom line lists three other places where tricks hide in a host.

A reader who sees `https://profitinalaundry.example@...` reads "Profitina Laundry". A browser reads a user name, `profitinalaundry.example`, followed by the real host, `receipt-view.click`. The laundry's name is decoration.

5.3.2 The usual tricks

Table 5.1 lists the tricks that phishing links use most often, with an example of each written for this book. None of them is new; they are the classic material of every security course, and knowing them is the first step to building a screener.

Table 5.1 Common tricks in malicious links, with made-up examples.

Trick	Example	Why it works
Bare IP address as host	`http://103.44.12.9/pay`	No name to check; nobody remembers numbers
User information before `@`	`https://brand.example@evil.click/`	The brand appears first; the real host comes after `@`
Punycode (internationalized) host	`https://xn--prfitina-w3a.com/`	Displays as a look-alike with an accented letter
One letter off a brand	`https://profltina.com/`	`l` for `i` is hard to see in many fonts
Brand name on someone else's domain	`https://profitina-support.xyz/`	The brand is a label; the domain is not the brand's
Stacked subdomains	`https://profitina.com.account.secure-login.buzz/`	The start looks right; the real domain is at the end
URL shortener	`bit.ly/pl-prize2026`	Hides the destination entirely
No HTTPS	`http://voucher-laundry.icu`	Traffic can be read or altered on the way
Cheap, often-abused TLD	`.xyz`, `.top`, `.click`	Low cost makes throwaway domains easy
Urgency in the text	"verify within 24 hours", "claim your prize"	Pressure stops people from checking

Two rows need a word of explanation.

Punycode. Domain names were originally limited to the letters a–z, digits and hyphens. To allow names in other scripts, internationalized domain names are converted into that limited alphabet by an encoding called Punycode, and the result starts with `xn--` [5]. The name `prófitina.com`, with an accented *o*, becomes `xn--prfitina-w3a.com`. A browser may show the accented version, which at a glance is indistinguishable from the real name. The Unicode Consortium maintains a whole standard on such *confusable* characters, UTS #39, whose version 18.0.0 was published on 27 August 2026 [6]. A community of Indonesian businesses has little need for accented host names, so treating `xn--` as a strong warning sign costs almost nothing.

Cheap TLDs. No top-level domain is malicious in itself, and legitimate businesses use every one of them. But some endings are sold so cheaply that they are favored for throwaway domains. A rule that looks at the TLD must therefore be *soft*: a hint to be combined with other hints, never a verdict on its own. The lab will show exactly what happens when a legitimate local printer uses such an ending.

5.3.3 Hard signals and soft signals

The tricks fall into two groups, and the difference drives the whole design of a screener.

> **DEFINITION.** A *hard signal* is a property of a link that legitimate links in the community almost never have, so that one occurrence is enough to block. A *soft signal* is a property that is

> suspicious but also occurs in legitimate links, so that it should count only in combination with other soft signals.

A bare IP address, user information before `@`, a punycode host, or a host one letter away from a protected brand are hard signals in a community like Profitina's: an honest member almost never posts them. A shortener, plain HTTP, a cheap TLD, a deep subdomain or urgent wording are soft: they appear in honest posts every day ("last chance to join the study group tonight!"). A screener that blocks on any soft signal blocks honest members; a screener that ignores soft signals misses scams that are careful to avoid hard ones. The lab of Section 5.8 makes that tension measurable.

5.3.4 Five seconds of checking

Software is one line of defense; the reader is another. Every rule in this section can also be applied by a person in a few seconds, and a community is safer when its members know how. Before clicking a link in any feed, not only Profitina's, it is worth asking five questions in order.

1. *Where does the host end?* Find the first `/` after `//`, look just before it, and read the last two labels. That, not the beginning, is whose server you will reach.
2. *Is there an `@` before that point?* If so, everything before it is decoration.
3. *Is it a number or does it start with `xn--`?* Honest businesses rarely send either.
4. *Is it a shortened link?* Then you cannot see the destination at all; ask the sender, or open it only if you trust the sender completely.
5. *Is the message rushing me?* "Within 24 hours", "last chance", "your account is blocked": pressure is the scammer's most reliable tool, because it stops exactly this kind of checking.

These questions do not replace a screener, and a screener does not replace them. A screener sees every link but understands no context; a reader understands context but gets tired. Together they catch more than either does alone, which is the same argument, at human scale, that Section 5.4 makes for layers of software.

5.4 Link Safety as Layers

5.4.1 The two layers

Profitina screens every link posted in the community. The screening has two layers [7], [8]:

1. **A rule floor.** A set of rules that combines several signals of the kinds described above into a verdict. It is deterministic (the same link always gets the same verdict), fast (it needs no network and no model), and explainable (every verdict comes with the rules that fired).
2. **An optional local-AI opinion.** On top of the floor, a small language model running locally on the platform's fast GPU lane can give its opinion about a link and the text around it. It sees the kind of intent that rules find hard to capture, such as a friendly message that is really a request for a code. It runs on Profitina's own machine, like all of its AI, so the message never leaves the server.

Safe links are shown as clickable links; malicious links are blocked [7]. **Figure 5.3** draws the two layers and, just as importantly, *when* they run.

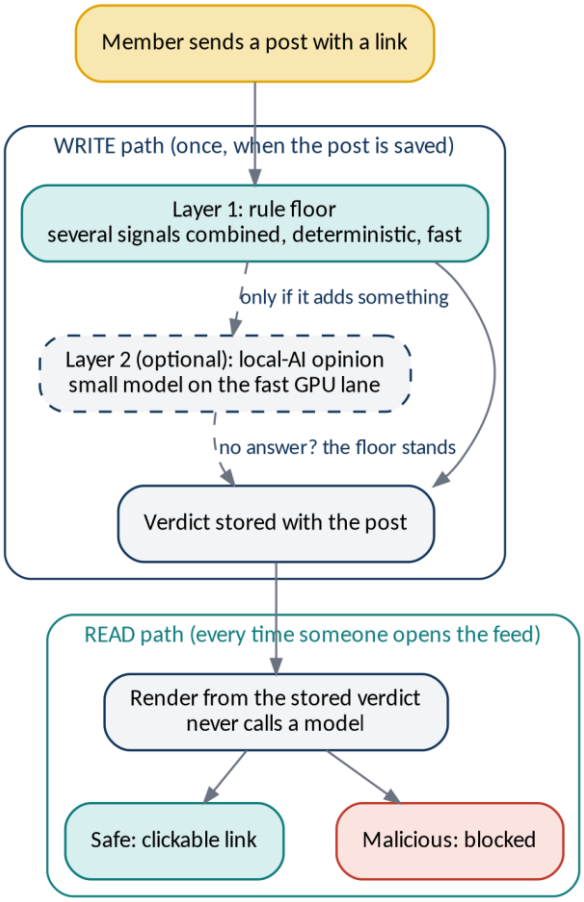


Figure 5.3 Link safety as layers. Both layers run once, on the write path, and the verdict is stored with the post; rendering the feed only reads the stored verdict and never calls a model.

5.4.2 Why the floor comes first

It would be tempting to hand the whole job to the model. Four reasons argue against it.

- **Availability.** A model can be busy, restarting, or slower than usual. A safety check that depends on it either blocks posting or silently lets links through. The floor always answers. If the model does not answer in time, the floor's verdict stands.
- **Explainability.** When a post is refused, the member deserves a reason. "The link's host is an IP address" is a reason; "the model said so" is not.
- **Testability.** A deterministic floor can be tested with a fixed set of examples, and a change to a rule can be measured before release. Profitina's public benchmark in Section 5.5.3 is exactly such a measurement.
- **Cost.** Rules cost microseconds. A model call, even on a small model and a fast GPU, costs orders of magnitude more.

The model is therefore an *addition*: it may notice what the rules miss, but the platform does not depend on it being there.

5.4.3 Never on the read path

Chapter 4 told the story of the September 2026 release, in which a feed page had been calling the model while rendering, six calls on one test page, and took 868 ms to appear. The lesson applies with special force to link safety. A link is written once and read perhaps thousands of times. If screening runs when the post is *written*, it runs once, while one member waits for their own post;

its verdict is stored with the post. If screening runs when the post is *read*, it runs thousands of times, while everybody waits. Since that release, the model's opinion is asked for only on the write path, and the read path only reads the stored verdict [8].

> **KEY INSIGHT.** Decide once, when data enters the system; read the decision many times. This is the same rule that makes databases store computed columns and web servers cache pages, and it is the reason a safety check can be thorough without making the feed slow.

5.4.4 A second, quieter layer: showing text as text

Screening links is one half of showing user content safely. The other half is how the page inserts that content. The lab page of Chapter 4 inserts every post with `textContent`, never with `innerHTML`, so that a post containing `<script>` shows those characters instead of running them. This is not a filter that can be fooled; it is a rule about how the browser is asked to treat the text, and it holds for every post whether or not anything was screened.

5.5 Measuring a Screener: The Confusion Matrix

5.5.1 Four kinds of outcome

A screener looks at a message and says either *flag* (block it) or *pass*. The message, in truth, is either a *scam* or *normal*. That gives four combinations, and every evaluation of every classifier starts by counting them [9], [10].

> **DEFINITION.** Call a scam a *positive*. Then a **true positive** (TP) is a scam that was flagged; a **false negative** (FN) is a scam that was passed; a **false positive** (FP) is a normal message that was flagged; and a **true negative** (TN) is a normal message that was passed. Arranged in a 2 × 2 table with the truth as rows and the verdict as columns, these four counts form the *confusion matrix*.

A tiny example makes the four counts concrete. Suppose a screener sees ten messages: four scams and six normal ones. It flags three of the scams and one of the normal messages. Then $TP = 3$ (scams flagged), $FN = 1$ (the scam it missed), $FP = 1$ (the normal message it wrongly flagged) and $TN = 5$ (normal messages passed). The four counts always add up to the number of messages, $3 + 1 + 1 + 5 = 10$, and the two rows add up to the number of scams and of normal messages, $3 + 1 = 4$ and $1 + 5 = 6$. Checking those sums is the quickest way to catch a counting mistake in any evaluation, including your own.

The two kinds of mistake have very different costs. A false negative lets a scam reach members. A false positive blocks an honest member, who is then told, in effect, that the platform thinks they are a scammer. A good screener keeps both low, and a good evaluation reports both separately.

5.5.2 Five measures from four counts

From the four counts we compute a handful of standard measures. **Precision** asks: of everything flagged, how much really was a scam?

$$\text{precision} = \frac{TP}{TP + FP} \tag{5.1}$$

Recall, also called sensitivity or the true positive rate, asks: of all the scams, how many were flagged?

$$\text{recall} = \frac{TP}{TP + FN} \tag{5.2}$$

Precision and recall pull in opposite directions: flag everything and recall is perfect while precision collapses; flag almost nothing and precision may be perfect while recall collapses. The **F1 score** combines them into one number, their harmonic mean:

$$F_1 = \frac{2 \cdot \text{precision} \cdot \text{recall}}{\text{precision} + \text{recall}} = \frac{2\,TP}{2\,TP + FP + FN} \tag{5.3}$$

The second form follows from the first by substituting Equation (5.1) and Equation (5.2) and multiplying numerator and denominator by $(TP + FP)(TP + FN)/TP$. The harmonic mean punishes imbalance: a screener with precision 1.0 and recall 0.1 has an arithmetic mean of 0.55 but an F_1 of only $2 \times 0.1/1.1 \approx 0.18$.

Accuracy is the share of all verdicts that were right:

$$\text{accuracy} = \frac{TP + TN}{TP + FN + FP + TN} \tag{5.4}$$

and the **false positive rate** (FPR) is the share of normal messages that were wrongly flagged:

$$\text{FPR} = \frac{FP}{FP + TN} \tag{5.5}$$

Accuracy is the measure people quote first and should trust least. If 99 of every 100 messages are normal, a "screener" that passes everything has an accuracy of 99 % and catches nothing. For a community, the two numbers that matter are recall (how many scams get through) and FPR (how many honest members get blocked).

5.5.3 Profitina's public benchmark

Profitina publishes one benchmark result for its rule floor: on a set of 28 realistic scam texts and 30 normal texts, it catches 27 of the 28 scams with 0 false positives on the 30 normal texts [7], [8]. The left half of **Figure 5.4** shows that result as a confusion matrix: $TP = 27$, $FN = 1$, $FP = 0$, $TN = 30$.

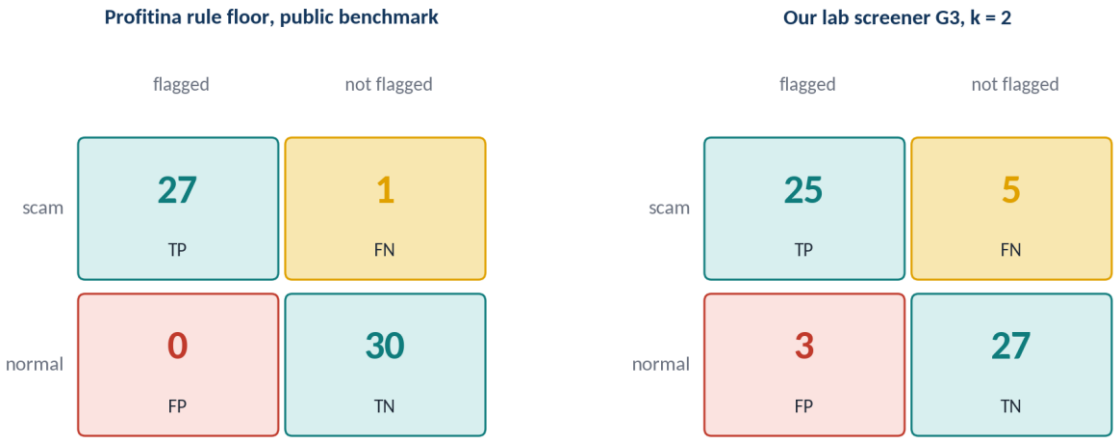


Figure 5.4 Two confusion matrices. Left: Profitina's published benchmark for its rule floor (27 of 28 scams caught, 0 false positives on 30 normal texts). Right: the lab screener of Section 5.8 on this book's own 60 samples with $k = 2$.

Working the formulas through, one step at a time:

- precision by Equation (5.1): $27/(27+0) = 1.000$;
- recall by Equation (5.2): $27/(27+1) = 27/28 \approx 0.964$;
- F_1 by Equation (5.3): $2 \times 27/(54+0+1) = 54/55 \approx 0.982$;
- accuracy by Equation (5.4): $(27+30)/58 = 57/58 \approx 0.983$;
- FPR by Equation (5.5): $0/30 = 0$.

These are good numbers. The next two subsections are about reading them correctly.

5.5.4 What zero out of thirty proves

Zero false positives on 30 normal texts does *not* prove that the false positive rate is zero. It proves that it is probably small. How small? Suppose the true rate is p. The chance of seeing no false positive in n independent normal texts is $(1-p)^n$. The largest p for which that chance is still at least 5 % is the 95 % upper confidence bound, found by solving $(1-p)^n = 0.05$:

$$p_{\text{upper}} = 1 - 0.05^{1/n} \approx \frac{3}{n}. \tag{5.6}$$

The approximation on the right is the classical *rule of three* [11]: it follows from $\ln 0.05 \approx -3$ and $\ln(1-p) \approx -p$ for small p. For $n = 30$ the exact bound is $1 - 0.05^{1/30} = 1 - e^{-0.0999} \approx 0.095$, and the rule of three gives $3/30 = 0.10$. In words: after 0 false positives in 30 normal texts, we can say with 95 % confidence that the false positive rate is below about 10 %. To push that bound below 1 %, Equation (5.6) says we would need about 300 normal texts with no false positive. Small benchmarks are honest about direction, not about precision.

5.5.5 Why the false positive rate matters more than it seems

Benchmarks are usually *balanced*: roughly as many scams as normal texts. Real feeds are not. Suppose that a fraction π of all messages containing links are scams, the *prevalence*. Then among all flagged messages, the fraction that really are scams, the precision the members actually experience, is

$$\text{PPV} = \frac{\text{recall} \cdot \pi}{\text{recall} \cdot \pi + \text{FPR} \cdot (1-\pi)}, \tag{5.7}$$

which follows from Bayes' rule: of M messages, $\text{recall} \cdot \pi M$ are flagged scams and $\text{FPR} \cdot (1-\pi) M$ are flagged normal messages. PPV stands for *positive predictive value*.

Take a hypothetical screener with recall 0.833 and FPR 0.100 (these happen to be the lab screener's numbers in Section 5.8) and a feed where 2 % of link-carrying messages are scams. By Equation (5.7),

$$\text{PPV} = \frac{0.833 \times 0.02}{0.833 \times 0.02 + 0.100 \times 0.98} = \frac{0.01666}{0.01666 + 0.098} \approx 0.145.$$

Only about one flagged message in seven would be a scam; the other six would be honest members being blocked. On a balanced benchmark the same screener has a precision of 0.893. The difference is entirely due to prevalence, and it is why a platform must drive its false positive rate very low before it can block automatically, and why measuring on more normal texts is worth the effort.

TRAP. Never compare two screeners' numbers when they were measured on different sample sets. A screener that scores 0.96 on easy samples may be worse than one that scores 0.83 on hard ones. The lab's numbers and Profitina's benchmark in **Figure 5.4** are side by side to show two confusion matrices, not to rank two screeners.

5.6 Moderators: Human Judgment, Narrow Powers

5.6.1 What software cannot do

A link screener catches links. It does not notice that a member is repeatedly mocking a classmate, that a thread has drifted into selling something, or that a "staff member" asking for OTP codes has no link in the message at all. These are judgments about people and context, and the best judge is a person who knows the room.

5.6.2 Channel-scoped moderators

Since 13 August 2026 Profitina has had *channel-scoped moderators*, built so that a university course could run its own channel [7]. A moderator is appointed for one channel, and inside that channel only may do three things: **hide** a post, **pin** an announcement, and **mute** a member for a limited time. **Figure 5.5** shows those powers and, just as clearly, what lies outside them.

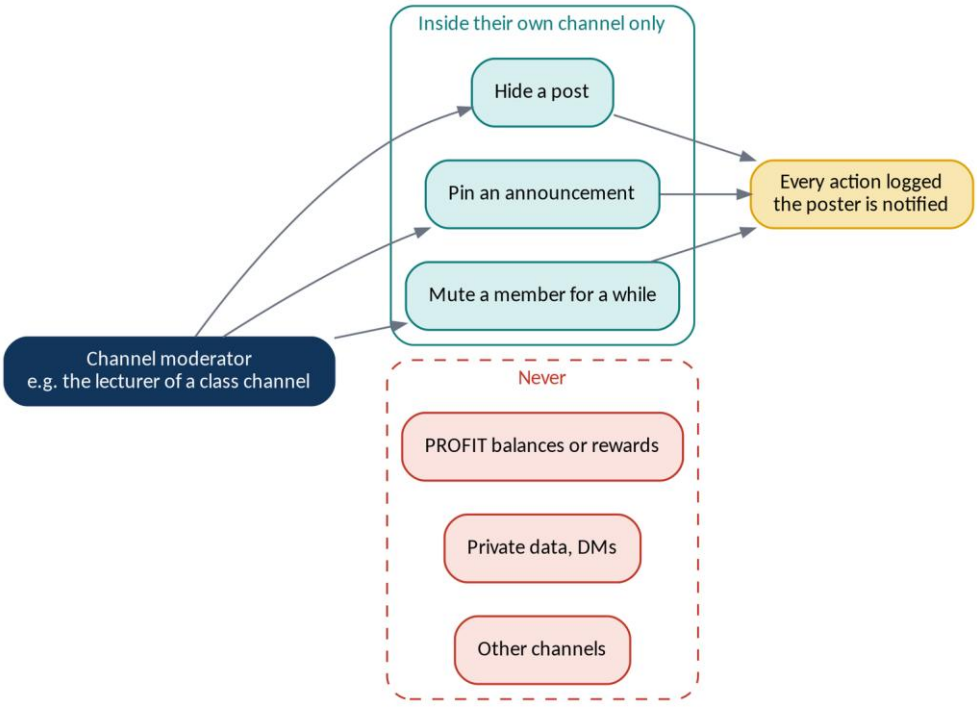


Figure 5.5 What a channel moderator can and cannot do. Three powers inside one channel, every action logged and the poster notified; no access to PROFIT, private data, direct messages or other channels.

The limits are deliberate, and each one answers a risk.

- **One channel only.** A lecturer who moderates a course channel has no power anywhere else. A moderator who could act across the platform would be a second administrator without an administrator's accountability.
- **No PROFIT, no private data.** A moderator never touches balances or rewards and never sees private information such as email addresses or DMs. Moderation is about what is said in public, not about who someone is or what they own.

- **Hide, not delete.** The power a moderator has is to take a post out of view, a milder step than destroying it.
- **Mute for a while, do not ban.** A temporary mute cools a situation; it does not end someone's membership.
- **Everything is logged, and the poster is told.** Every moderation action is recorded, and the person whose post was hidden or who was muted is notified.

These rules echo the *Santa Clara Principles on Transparency and Accountability in Content Moderation*, drawn up by civil-society organizations and researchers, whose second version calls for clear rules, notice to the affected person, and the possibility of appeal [12]. A small platform cannot run a large appeals department, but it can make every action visible to the person affected and reversible by design.

5.6.3 Reporting, and the frozen copy

Every member can report a post, and every DM recipient can report a message. As Chapter 4 explained, administrators cannot read DMs; a report creates a *frozen copy* of the reported message, the only part of a private conversation that ever becomes visible to reviewers, preserved exactly as it was so that it cannot be edited away afterwards. Blocking is instant and needs no one's approval.

> **EXAMPLE.** In a course channel, a member posts "Answers for assignment 2, message me for the file" with no link. The link screener has nothing to screen. Two classmates report the post. The lecturer, who moderates the channel, hides it and pins a reminder about the course's rules. The poster is notified that the post was hidden, and the action appears in the moderation log. Nobody's PROFIT balance is touched, and nothing outside the channel changes.

5.6.4 Rules that can be moderated

Moderators can only be fair if the rules they enforce are written so that a reasonable person can apply them. Course channels taught a simple lesson here: the pinned rules of a channel work best when each rule names a *behavior*, not an attitude, and says what happens when it is broken. "Be respectful" is a good value and a poor rule, because two people will disagree about whether a joke crossed the line. "No posting of assignment answers before the deadline; such posts are hidden" can be applied the same way by any moderator, and a member can predict the outcome before posting.

A good channel rule has three parts. The *behavior* it forbids or asks for, stated so that it can be recognized in a single post. The *action* a moderator will take, chosen from the three powers of Section 5.6.2. And the *reason*, in one short sentence, because members follow rules they understand far more willingly than rules they merely obey. A short list of such rules, pinned at the top of the channel, does more for a room than any amount of software, and it makes every moderation action easy to explain in the notification the poster receives.

The same discipline applies to the platform itself. Profitina's own rules for moderators, one channel, three powers, no PROFIT, no private data, every action logged, are written in exactly this style: each one names what may or may not be done, and each one has a reason that can be stated in a sentence.

5.7 Fair Play: Keeping Rewards for Humans

5.7.1 Rewards attract automation

Any measure that is rewarded becomes a target. The economist Charles Goodhart's observation, in the general form popularized by Marilyn Strathern, is that "when a measure becomes a target, it ceases to be a good measure" [13]. Pay for posts, and you get posts, whether or not anyone wanted them. With more than half of web traffic already automated [3], a platform that pays for contributions must assume that some of its "members" are programs.

Profitina's answer is a set of fair-play layers, described in its public whitepaper [7] and drawn at box level in **Figure 5.6**. The order in the figure is for teaching; what matters is that every contribution passes all of them.

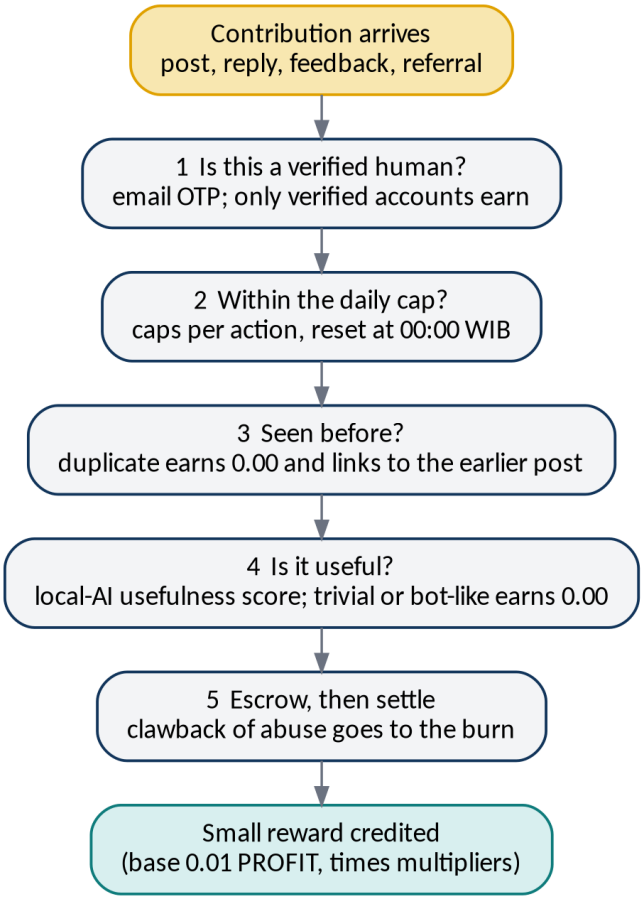


Figure 5.6 Fair play at box level. A contribution earns a small reward only if it comes from a verified human, stays within the daily cap, is not a duplicate and is judged useful; abuse found later is clawed back and burned.

5.7.2 The layers

Verified humans only. Registration needs an email address and a password, and the email is verified with a one-time code (OTP), the first anti-bot layer. Only email-verified accounts earn anything; a referral reward, 0.25 PROFIT to each side, is paid only after the referred person has verified their email, and only once per pair, for life. The referral program has exactly one tier, because multi-level referral schemes are, in the whitepaper's words, "structurally similar to pyramid schemes" [7].

Small rewards, daily caps. Rewards are small (the base unit is 0.01 PROFIT) and each kind of action has a daily cap that resets at 00:00 WIB. A script that posts a thousand times earns no more than a person who posts a few times.

Duplicates earn nothing. A post that repeats earlier content earns 0.00 and links to the earlier post, so copying is visible as well as unrewarded.

Usefulness is judged. A local-AI usefulness score multiplies each reward; trivial or bot-like contributions receive 0.00. The model runs locally, like all of Profitina's AI.

Escrow, clawback and burn. Rewards are held before they settle; if abuse is found, they are clawed back, and the clawed-back PROFIT is burned rather than redistributed, which is one of the coin's four burn mechanisms described in Chapter 12.

Edits are recalculated. As Chapter 4 explained, editing a post within its fifteen-minute window recalculates its reward in full, so the edit window cannot be used to upgrade a post after it has been scored.

Rank cannot be bought. PROFIT spent on services never raises a member's rank; PROFIT that was bought or granted cannot be transferred to others, only PROFIT earned organically; and administrators' discretionary grants are published and excluded from every leaderboard [7].

> **KEY INSIGHT.** None of these layers needs to be perfect, because they multiply. A bot that passes email verification still hits the daily cap; a bot that stays under the cap still produces duplicates or trivial text; a bot that writes varied text still has its rewards held in escrow where abuse can be clawed back. Each layer turns a large gain from automation into a smaller one, and together they make farming not worth the effort.

5.7.3 Why layers multiply: a worked illustration

The claim that layers *multiply* can be made precise with a small, entirely made-up example; none of its numbers are Profitina's. Suppose a farming script would, with no defenses at all, earn some amount G per day. Suppose now that each layer, acting alone, lets through only a fraction of what reaches it: verification lets through a fraction v of the accounts the script tries to create, the daily cap lets each surviving account keep a fraction c of what it would otherwise earn, duplicate detection and the usefulness score together let a fraction u of the remaining contributions earn anything, and clawback leaves a fraction r of that in the script's hands. If the layers act independently, the script's expected gain is

$$G_{\text{kept}} = G \cdot v \cdot c \cdot u \cdot r. \tag{5.8}$$

With the illustrative values $v = 0.5$, $c = 0.1$, $u = 0.2$ and $r = 0.5$, each of which on its own looks like a weak filter (half the accounts survive, half the rewards stay), Equation (5.8) gives $0.5 \times 0.1 \times 0.2 \times 0.5 = 0.005$: the script keeps half of one percent of what it would have earned. Meanwhile its costs, mailboxes to verify, varied text to generate and time to wait out escrow, are all higher than before. Automation stops being worth it long before any single layer is perfect. The same arithmetic explains why a platform can afford to keep each layer *gentle* towards honest members: no one layer has to be harsh, because the protection comes from their product.

5.7.4 What is deliberately not published

The layers are public; their settings are not. The exact caps per action, the usefulness threshold, the similarity at which two posts count as duplicates and the timing of escrow are tuned from

observation and changed when abuse changes. Publishing them would turn each one into a target in exactly Goodhart's sense. What *is* published is the result that matters to members: small rewards, for verified humans, for useful contributions, with daily caps that reset at 00:00 WIB.

5.8 Lab G3: Our Own Rule-Based Link Screener

5.8.1 What we will build

The lab builds a link screener for the Profitina Laundry community in Python and Java, with identical behavior and output, and an evaluation harness that measures it on a labeled sample set. The design, drawn in **Figure 5.7**, has four steps: extract the links from a message, split each link into its parts, apply every rule, and decide.

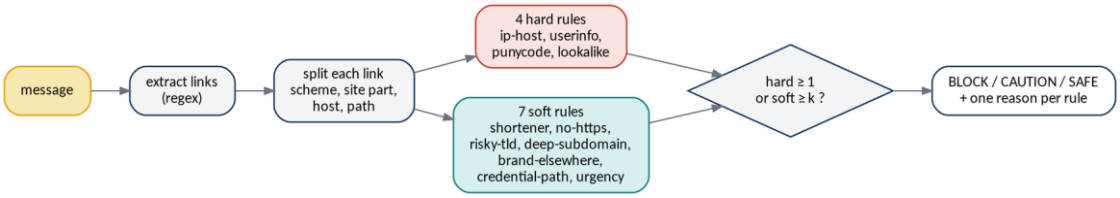


Figure 5.7 The lab screener. Links are extracted and split, eleven textbook rules are applied, and the verdict is BLOCK if any hard rule fires or at least k soft rules fire.

The rules are the textbook tricks of **Table 5.1**, turned into code with lists that you are meant to change: the brands we protect, their official domains, the shorteners, the TLDs we consider often abused, the urgency phrases and the words that suggest a request for credentials. **Table 5.2** lists all eleven.

Table 5.2 The lab screener's eleven rules. The lists behind them are this book's own examples and are meant to be edited.

Rule	Kind	Fires when
`ip-host`	hard	The host is an IPv4 address
`userinfo`	hard	The site part contains `@`
`punycode`	hard	A host label starts with `xn--`
`lookalike`	hard	A host label or hyphen-separated part is exactly one edit from a protected brand, on a non-official host
`shortener`	soft	The host is on the shortener list
`no-https`	soft	The scheme is `http`
`risky-tld`	soft	The TLD is on the often-abused list
`deep-subdomain`	soft	The host has four or more labels
`brand-elsewhere`	soft	A protected brand appears as a label or part on a non-official host
`credential-path`	soft	The path contains a word such as `login`, `verify` or `otp`, on a non-official host
`urgency`	soft	The message contains an urgency phrase

5.8.2 The decision rule

Let h be the number of hard rules and s the number of soft rules that fire for a link, and let k be a threshold that you choose. The verdict for the link is

$$\text{verdict} = \begin{cases} \text{BLOCK} & \text{if } h \geq 1 \text{ or } s \geq k, \\ \text{CAUTION} & \text{if } h = 0 \text{ and } 1 \leq s < k, \\ \text{SAFE} & \text{if } h = 0 \text{ and } s = 0. \end{cases} \tag{5.9}$$

A message with several links gets the worst verdict of its links, and a message with no link at all is SAFE, because a link screener has nothing to say about it. For the evaluation, BLOCK counts as *flagged* and both CAUTION and SAFE count as *passed*: a CAUTION link would still be shown, perhaps with a warning.

The threshold k is the lab's one tuning knob. With $k = 1$ any soft signal blocks; with $k = 5$ almost nothing soft ever does. Section 5.8.7 measures what each choice costs.

5.8.3 How far is "one letter away"?

The `lookalike` rule needs a precise notion of "one letter away". The standard one is the *edit distance* introduced by Levenshtein in 1966 [14]: the smallest number of single-character insertions, deletions and substitutions that turn one string into another. Write $D(i, j)$ for the distance between the first i characters of a string a and the first j characters of a string b. Then $D(i, 0) = i$ and $D(0, j) = j$ (building from or down to nothing), and for $i, j \geq 1$

$$D(i,j) = \min(D(i-1,j) + 1,\ D(i,j-1) + 1,\ D(i-1,j-1) + [a_i \neq b_j]), \tag{5.10}$$

where $[a_i \neq b_j]$ is 1 if the i-th character of a differs from the j-th character of b and 0 otherwise. The three terms are, in order, deleting a_i, inserting b_j, and keeping or substituting. Filling the table row by row is the classic dynamic-programming algorithm of Wagner and Fischer [15]; it takes $O(|a|\,|b|)$ time and, keeping only two rows, $O(|b|)$ memory.

Table 5.3 fills the table for $a = $ `pay` and $b = $ `pya`, two strings that differ by swapping two neighboring letters.

Table 5.3 The edit-distance table for `pay` and `pya`, filled row by row with Equation (5.10); the answer is in the bottom-right cell.

	""	p	y	a
""	0	1	2	3
p	1	0	1	2
a	2	1	1	1
y	3	2	1	2

Take one cell to see the recurrence at work: row `a`, column `a`. The characters are equal, so the diagonal term is $D(1,2) + 0 = 1 + 0 = 1$, while deleting gives $D(1,3) + 1 = 3$ and inserting gives $D(2,2) + 1 = 2$; the minimum is 1. The bottom-right cell says the distance is 2: a swap of neighbors costs two edits in Levenshtein's definition. (A variant by Damerau counts a swap as one edit; it is a good extension exercise.) For `profltina` and `profitina` the distance is 1, a single substitution, and for `profitina-laundry` and `profitinalaundry` it is also 1, a single insertion of the hyphen.

5.8.4 The Python screener

Listing 5.1 is the whole screener. The lists at the top are the rule data; change them and the behavior changes.

Listing 5.1 `link_screener.py`: a rule-based link screener with eleven textbook rules and the decision rule of Equation (5.9).

```
"""A small rule-based link screener, written for teaching.

The rules and lists below are this book's own textbook examples. Change them,
measure again with evaluate.py, and see what happens.

  python link_screener.py "message text with a link"  [--k 2]
"""
import re
import sys

BRANDS = ["profitina", "profitinalaundry", "dompetkita"]       # names we protect
OFFICIAL = ["profitina.com", "profitinalaundry.example", "dompetkita.example"]
SHORTENERS = ["bit.ly", "s.id", "tinyurl.com", "t.ly", "cutt.ly"]
RISKY_TLDS = ["xyz", "top", "click", "icu", "buzz", "cfd", "rest"]
URGENCY = ["urgent", "immediately", "24 hours", "account suspended",
           "account is blocked", "verify your account", "claim", "prize",
           "winner", "last chance", "limited time", "act now",
           "segera", "hadiah", "verifikasi"]
CREDENTIAL_WORDS = ["login", "verify", "secure", "wallet", "otp", "password"]

URL = re.compile(r"https?://[^\s<>\"']+"
                 r"|\b(?:www\.)?[a-z0-9-]+(?:\.[a-z0-9-]+)*\.[a-z]{2,}"
                 r"(?:/[^\s<>\"']*)?", re.IGNORECASE)
IPV4 = re.compile(r"^\d{1,3}(?:\.\d{1,3}){3}$")


def levenshtein(a, b):
    """Fewest one-character insertions, deletions or substitutions a -> b."""
    prev = list(range(len(b) + 1))
    for i in range(1, len(a) + 1):
        cur = [i] + [0] * len(b)
        for j in range(1, len(b) + 1):
            cost = 0 if a[i - 1] == b[j - 1] else 1
            cur[j] = min(prev[j] + 1, cur[j - 1] + 1, prev[j - 1] + cost)
        prev = cur
    return prev[len(b)]


def extract_links(text):
    return [m.group(0).rstrip(".,;:!?)'\"") for m in URL.finditer(text)]
```

Listing 5.1 (Continued) *link_screener.py: a rule-based link screener with eleven textbook rules and the decision rule of Equation (5.9).*

```
def split_link(link):
    """Return (scheme, authority, host, path) using plain string steps."""
    lower = link.lower()
    scheme = ""
    rest = link
    for s in ("https", "http"):
        if lower.startswith(s + "://"):
            scheme, rest = s, link[len(s) + 3:]
            break
    cut = len(rest)
    for ch in "/?#":
        pos = rest.find(ch)
        if pos != -1 and pos < cut:
            cut = pos
    authority, path = rest[:cut], rest[cut:]
    host = authority[authority.rfind("@") + 1:].split(":")[0].lower()
    return scheme, authority, host, path.lower()


def is_official(host):
    return any(host == d or host.endswith("." + d) for d in OFFICIAL)


def check_link(link, text):
    """Apply every rule to one link; return a list of (kind, name, why)."""
    scheme, authority, host, path = split_link(link)
    labels = host.split(".")
    tokens = set(labels[:-1])
    for label in labels[:-1]:
        tokens.update(label.split("-"))
    official = is_official(host)
    fired = []
    # Hard rules: one is enough to block.
    if IPV4.match(host):
        fired.append(("hard", "ip-host", "the host is a bare IP address"))
    if "@" in authority:
        fired.append(("hard", "userinfo",
                      "text before '@' hides the real host " + host))
    if any(label.startswith("xn--") for label in labels):
        fired.append(("hard", "punycode",
                      "the host uses punycode (look-alike letters)"))
    if not official:
        for tok in sorted(tokens):
            near = [b for b in BRANDS if levenshtein(tok, b) == 1]
            if near:
                fired.append(("hard", "lookalike",
```

Listing 5.1 (Continued) *link_screener.py: a rule-based link screener with eleven textbook rules and the decision rule of Equation (5.9).*

```
                            "'%s' is one letter away from '%s'"
                            % (tok, near[0])))
                break
    # Soft rules: block when k or more of them fire.
    if host in SHORTENERS:
        fired.append(("soft", "shortener", "a short link hides the target"))
    if scheme == "http":
        fired.append(("soft", "no-https", "the link does not use HTTPS"))
    if labels[-1] in RISKY_TLDS:
        fired.append(("soft", "risky-tld",
                      "." + labels[-1] + " is on our list of often-abused endings"))
    if len(labels) >= 4:
        fired.append(("soft", "deep-subdomain",
                      "%d labels in the host name" % len(labels)))
    if not official and any(b in tokens for b in BRANDS):
        fired.append(("soft", "brand-elsewhere",
                      "a brand name on a domain that is not the brand's own"))
    if not official and any(w in path for w in CREDENTIAL_WORDS):
        fired.append(("soft", "credential-path",
                      "the path asks for a login, a code or a wallet"))
    lowered = text.lower()
    if any(w in lowered for w in URGENCY):
        fired.append(("soft", "urgency", "the message pushes you to act fast"))
    return host, fired


def decide(fired, k):
    """BLOCK if any hard rule, or at least k soft rules; else CAUTION/SAFE."""
    hard = sum(1 for kind, _, _ in fired if kind == "hard")
    soft = len(fired) - hard
    if hard >= 1 or soft >= k:
        return "BLOCK", hard, soft
    return ("CAUTION" if soft else "SAFE"), hard, soft


def screen(text, k=2):
    """Screen a whole message; the worst link decides. No link means SAFE."""
    order = {"SAFE": 0, "CAUTION": 1, "BLOCK": 2}
    worst, report = "SAFE", []
    for link in extract_links(text):
        host, fired = check_link(link, text)
        verdict, hard, soft = decide(fired, k)
        report.append((link, host, fired, verdict, hard, soft))
        if order[verdict] > order[worst]:
            worst = verdict
    return worst, report
```

Listing 5.1 (Continued) *link_screener.py: a rule-based link screener with eleven textbook rules and the decision rule of Equation (5.9).*

```
def main():
    args = sys.argv[1:]
    k = 2
    if "--k" in args:
        k = int(args[args.index("--k") + 1])
        del args[args.index("--k"):args.index("--k") + 2]
    text = " ".join(args)
    verdict, report = screen(text, k)
    if not report:
        print("no link found")
    for link, host, fired, v, hard, soft in report:
        print("link:", link)
        print("  host:", host)
        for kind, name, why in fired:
            print("  [%s] %s: %s" % (kind, name, why))
        print("  hard %d, soft %d (k = %d)  ->  %s" % (hard, soft, k, v))
    print("verdict:", verdict)


if __name__ == "__main__":
    main()
```

A few points deserve attention.

- **Extracting links.** The regular expression `URL` has two alternatives: a full link starting with `http://` or `https://`, or a bare domain such as `www.profitina-bonus.top` or `s.id/laundry-kirim`, because scammers often omit the scheme. Trailing punctuation is stripped, so that a link at the end of a sentence does not include the period.
- **Splitting without a library.** `split_link` finds the scheme, cuts the site part at the first `/`, `?` or `#`, and takes the host as the part after the *last* `@` and before any `:`. It does this with plain string operations rather than `urllib.parse`, so that the Java version can do exactly the same steps and produce exactly the same answers.
- **Official hosts.** `is_official` accepts a host that equals an official domain or ends with a dot followed by it, so `app.profitina.com` is official and `profitina.com.account.secure-login.buzz` is not. The three rules about brands and credentials are skipped for official hosts.
- **Tokens.** For the brand rules, a host is broken into its labels (without the TLD) and the hyphen-separated parts of each label, so that `secure-dompetkita.co` yields the token `dompetkita`. Sorting the tokens makes the output deterministic in both languages.
- **Explanations.** Every rule that fires returns a short reason. The screener never says only "blocked".

How much work does the screener do? For a message of length m with L links, extraction is one pass over the text, $O(m)$. For each link the rules are simple scans of the link and of the message, except `lookalike`, which computes the edit distance between each of the host's tokens and each of the B protected brands. A token and a brand are both short, a few dozen characters at most, so each distance costs at most a few thousand steps by the $O(|a|\,|b|)$ bound of Section 5.8.3. In total the work is $O(m + L \cdot t \cdot B \cdot \ell^2)$, where t is the number of tokens per host and ℓ the maximum token length: microseconds for any real message. That is the cost argument of Section 5.4.2 in numbers.

5.8.5 The Java screener

Listing 5.2 is the same screener in Java. Records hold a rule hit, the report for one link, and the result for a message; a `TreeSet` keeps the tokens sorted as Python's `sorted` does; and a labeled `break outer` stops after the first lookalike, like Python's `break`.

Listing 5.2 `LinkScreener.java`: the same screener in Java, producing byte-identical output.

```
import java.util.ArrayList;
import java.util.List;
import java.util.Locale;
import java.util.Set;
import java.util.TreeSet;
import java.util.regex.Matcher;
import java.util.regex.Pattern;

/** A small rule-based link screener, written for teaching.
 *  The rules and lists are this book's own textbook examples.
 *    java -cp out LinkScreener "message text with a link" [--k 2] */
public class LinkScreener {
    static final List<String> BRANDS =
            List.of("profitina", "profitinalaundry", "dompetkita");
    static final List<String> OFFICIAL = List.of("profitina.com",
            "profitinalaundry.example", "dompetkita.example");
    static final List<String> SHORTENERS =
            List.of("bit.ly", "s.id", "tinyurl.com", "t.ly", "cutt.ly");
    static final List<String> RISKY_TLDS =
            List.of("xyz", "top", "click", "icu", "buzz", "cfd", "rest");
    static final List<String> URGENCY = List.of("urgent", "immediately",
            "24 hours", "account suspended", "account is blocked",
            "verify your account", "claim", "prize", "winner", "last chance",
            "limited time", "act now", "segera", "hadiah", "verifikasi");
    static final List<String> CREDENTIAL_WORDS =
            List.of("login", "verify", "secure", "wallet", "otp", "password");

    static final Pattern URL = Pattern.compile("https?://[^\\s<>\"']+"
            + "|\\b(?:www\\.)?[a-z0-9-]+(?:\\.[a-z0-9-]+)*\\.[a-z]{2,}"
            + "(?:/[^\\s<>\"']*)?", Pattern.CASE_INSENSITIVE);
    static final Pattern IPV4 = Pattern.compile("^\\d{1,3}(?:\\.\\d{1,3}){3}$");

    /** One rule that fired: hard or soft, its name and a reason. */
    record Hit(String kind, String name, String why) { }

    /** The result for one link. */
    record LinkReport(String link, String host, List<Hit> fired,
                      String verdict, int hard, int soft) { }

    /** The result for one message: the worst link decides. */
    record Result(String verdict, List<LinkReport> links) { }
```

Listing 5.2 (Continued) *LinkScreener.java: the same screener in Java, producing byte-identical output.*

```
    /** Fewest one-character insertions, deletions or substitutions a -> b. */
    static int levenshtein(String a, String b) {
        int[] prev = new int[b.length() + 1];
        for (int j = 0; j <= b.length(); j++) prev[j] = j;
        for (int i = 1; i <= a.length(); i++) {
            int[] cur = new int[b.length() + 1];
            cur[0] = i;
            for (int j = 1; j <= b.length(); j++) {
                int cost = a.charAt(i - 1) == b.charAt(j - 1) ? 0 : 1;
                cur[j] = Math.min(Math.min(prev[j] + 1, cur[j - 1] + 1),
                                  prev[j - 1] + cost);
            }
            prev = cur;
        }
        return prev[b.length()];
    }

    static List<String> extractLinks(String text) {
        List<String> links = new ArrayList<>();
        Matcher m = URL.matcher(text);
        while (m.find()) {
            String link = m.group();
            int end = link.length();
            while (end > 0 && ".,;:!?)'\"".indexOf(link.charAt(end - 1)) >= 0) {
                end--;                                  // drop trailing punctuation
            }
            links.add(link.substring(0, end));
        }
        return links;
    }

    static boolean isOfficial(String host) {
        for (String d : OFFICIAL) {
            if (host.equals(d) || host.endsWith("." + d)) return true;
        }
        return false;
    }
```

Listing 5.2 (Continued) *LinkScreener.java: the same screener in Java, producing byte-identical output.*

```
    /** Apply every rule to one link. */
    static List<Hit> checkLink(String link, String text, String[] hostOut) {
        String lower = link.toLowerCase(Locale.ROOT);
        String scheme = "", rest = link;
        for (String s : new String[] {"https", "http"}) {
            if (lower.startsWith(s + "://")) {
                scheme = s;
                rest = link.substring(s.length() + 3);
                break;
            }
        }
        int cut = rest.length();
        for (char ch : "/?#".toCharArray()) {
            int pos = rest.indexOf(ch);
            if (pos != -1 && pos < cut) cut = pos;
        }
        String authority = rest.substring(0, cut);
        String path = rest.substring(cut).toLowerCase(Locale.ROOT);
        String host = authority.substring(authority.lastIndexOf('@') + 1)
                .split(":")[0].toLowerCase(Locale.ROOT);
        hostOut[0] = host;
        String[] labels = host.split("\\.", -1);
        Set<String> tokens = new TreeSet<>();          // sorted, like Python
        for (int i = 0; i < labels.length - 1; i++) {
            tokens.add(labels[i]);
            for (String part : labels[i].split("-", -1)) tokens.add(part);
        }
        boolean official = isOfficial(host);
        String tld = labels[labels.length - 1];
        List<Hit> fired = new ArrayList<>();
        // Hard rules: one is enough to block.
        if (IPV4.matcher(host).matches())
            fired.add(new Hit("hard", "ip-host", "the host is a bare IP address"));
        if (authority.contains("@"))
            fired.add(new Hit("hard", "userinfo",
                    "text before '@' hides the real host " + host));
        for (String label : labels) {
            if (label.startsWith("xn--")) {
                fired.add(new Hit("hard", "punycode",
                        "the host uses punycode (look-alike letters)"));
                break;
            }
        }
        if (!official) {
            outer:
            for (String tok : tokens) {
```

Listing 5.2 (Continued) *LinkScreener.java: the same screener in Java, producing byte-identical output.*

```
                for (String b : BRANDS) {
                    if (levenshtein(tok, b) == 1) {
                        fired.add(new Hit("hard", "lookalike", "'" + tok
                                + "' is one letter away from '" + b + "'"));
                        break outer;
                    }
                }
            }
        }
        // Soft rules: block when k or more of them fire.
        if (SHORTENERS.contains(host))
            fired.add(new Hit("soft", "shortener",
                    "a short link hides the target"));
        if (scheme.equals("http"))
            fired.add(new Hit("soft", "no-https", "the link does not use HTTPS"));
        if (RISKY_TLDS.contains(tld))
            fired.add(new Hit("soft", "risky-tld",
                    "." + tld + " is on our list of often-abused endings"));
        if (labels.length >= 4)
            fired.add(new Hit("soft", "deep-subdomain",
                    labels.length + " labels in the host name"));
        if (!official && BRANDS.stream().anyMatch(tokens::contains))
            fired.add(new Hit("soft", "brand-elsewhere",
                    "a brand name on a domain that is not the brand's own"));
        if (!official && CREDENTIAL_WORDS.stream().anyMatch(path::contains))
            fired.add(new Hit("soft", "credential-path",
                    "the path asks for a login, a code or a wallet"));
        String lowered = text.toLowerCase(Locale.ROOT);
        if (URGENCY.stream().anyMatch(lowered::contains))
            fired.add(new Hit("soft", "urgency",
                    "the message pushes you to act fast"));
        return fired;
    }

    /** Screen a whole message; the worst link decides. No link means SAFE. */
    static Result screen(String text, int k) {
        List<String> order = List.of("SAFE", "CAUTION", "BLOCK");
        String worst = "SAFE";
        List<LinkReport> report = new ArrayList<>();
        for (String link : extractLinks(text)) {
            String[] host = new String[1];
            List<Hit> fired = checkLink(link, text, host);
            int hard = (int) fired.stream()
                    .filter(h -> h.kind().equals("hard")).count();
            int soft = fired.size() - hard;
            String verdict = hard >= 1 || soft >= k ? "BLOCK"
```

Listing 5.2 (Continued) *LinkScreener.java: the same screener in Java, producing byte-identical output.*

```
                            : soft > 0 ? "CAUTION" : "SAFE";
            report.add(new LinkReport(link, host[0], fired, verdict, hard, soft));
            if (order.indexOf(verdict) > order.indexOf(worst)) worst = verdict;
        }
        return new Result(worst, report);
    }

    public static void main(String[] args) {
        int k = 2;
        StringBuilder text = new StringBuilder();
        for (int i = 0; i < args.length; i++) {
            if (args[i].equals("--k") && i + 1 < args.length) {
                k = Integer.parseInt(args[++i]);
            } else {
                if (text.length() > 0) text.append(' ');
                text.append(args[i]);
            }
        }
        Result r = screen(text.toString(), k);
        if (r.links().isEmpty()) System.out.println("no link found");
        for (LinkReport lr : r.links()) {
            System.out.println("link: " + lr.link());
            System.out.println("  host: " + lr.host());
            for (Hit h : lr.fired()) {
                System.out.println("  [" + h.kind() + "] " + h.name() + ": "
                        + h.why());
            }
            System.out.printf("  hard %d, soft %d (k = %d)  ->  %s%n",
                    lr.hard(), lr.soft(), k, lr.verdict());
        }
        System.out.println("verdict: " + r.verdict());
    }
}
```

5.8.6 Trying it on single messages

Run the screener on one message from the command line. The first example is a classic scam:

```
python link_screener.py "URGENT: your Profitina account is blocked. Verify within 24
hours at http://profitina-support.xyz/login"
```

```
link: http://profitina-support.xyz/login
  host: profitina-support.xyz
  [soft] no-https: the link does not use HTTPS
  [soft] risky-tld: .xyz is on our list of often-abused endings
  [soft] brand-elsewhere: a brand name on a domain that is not the brand's own
  [soft] credential-path: the path asks for a login, a code or a wallet
  [soft] urgency: the message pushes you to act fast
  hard 0, soft 5 (k = 2)  ->  BLOCK
verdict: BLOCK
```

No single rule here is damning, which is exactly why the example is instructive: five soft rules fire, far above $k = 2$. The link from **Figure 5.2** is caught by a hard rule:

```
python link_screener.py "Check your laundry receipt:
https://profitinalaundry.example@receipt-view.click/r/88"
```

```
link: https://profitinalaundry.example@receipt-view.click/r/88
  host: receipt-view.click
  [hard] userinfo: text before '@' hides the real host receipt-view.click
  [soft] risky-tld: .click is on our list of often-abused endings
  hard 1, soft 1 (k = 2)  ->  BLOCK
verdict: BLOCK
```

A one-letter look-alike is caught by the edit distance of Equation (5.10):

```
python link_screener.py "Free 50 PROFIT for early members! Connect your wallet:
https://profltina.com/airdrop"
```

```
link: https://profltina.com/airdrop
  host: profltina.com
  [hard] lookalike: 'profltina' is one letter away from 'profitina'
  hard 1, soft 0 (k = 2)  ->  BLOCK
verdict: BLOCK
```

An official link passes with nothing to report, and a legitimate but unusual host gets a CAUTION rather than a block:

```
python link_screener.py "Scan your site again after adding FAQ schema:
https://app.profitina.com/#/pindai"
python link_screener.py "Read the APJII survey: https://survey.apjii.or.id/"
```

```
link: https://app.profitina.com/#/pindai
  host: app.profitina.com
  hard 0, soft 0 (k = 2)  ->  SAFE
verdict: SAFE
link: https://survey.apjii.or.id/
  host: survey.apjii.or.id
  [soft] deep-subdomain: 4 labels in the host name
  hard 0, soft 1 (k = 2)  ->  CAUTION
verdict: CAUTION
```

The Java version gives the same output: in the `java` folder, compile with `javac -d out *.java` and run `java -cp out LinkScreener "..."`.

5.8.7 The sample set and the evaluation

A screener is only as trustworthy as its evaluation. The code folder contains `data/samples.tsv`, 60 messages written for this book about the Profitina Laundry community, each labeled `scam` or `normal` and separated from its label by a tab. Every name and link in it is made up, and it deliberately includes hard cases on both sides. **Table 5.4** shows a few lines.

Table 5.4 A few of the 60 labeled samples (30 scam, 30 normal) in `data/samples.tsv`.

#	Label	Message (shortened)
1	scam	URGENT: your Profitina account is blocked. Verify within 24 hours at http://profitina-support.xyz/login
9	scam	Your parcel of clean clothes is waiting. Confirm the address: s.id/laundry-kirim
10	scam	Hi, this is the Profitina admin. Please send me the OTP code we just texted you.
26	scam	Confirm your email to keep earning PROFIT: https://profitina.com-verify.email/confirm
33	normal	Scan your site again after adding FAQ schema: https://app.profitina.com/#/pindai
42	normal	Last chance to join the OOP study group tonight: http://kelas.example/oop
50	normal	Local printer shop for our flyers: http://cetak-murah.xyz
51	normal	Read the APJII survey: https://survey.apjii.or.id/

Listing 5.3 reads the samples, screens each one, counts the four outcomes, prints the measures of Section 5.5 with their arithmetic, lists every mistake with the rules that fired, and finally repeats the count for $k = 1$ to 5. **Listing 5.4** is the Java version.

Listing 5.3 `evaluate.py`: the confusion matrix, five measures, the list of mistakes and a sweep over k.

```
"""Measure the screener on our own labeled samples.

  python evaluate.py [--k 2] [--data ../data/samples.tsv]
"""
import sys
from pathlib import Path

from link_screener import screen

DATA = Path(__file__).resolve().parent.parent / "data" / "samples.tsv"


def load(path):
    samples = []
    for line in path.read_text(encoding="utf-8").splitlines():
        if line and not line.startswith("#"):
            label, text = line.split("\t", 1)
            samples.append((label, text))
    return samples


def ratio(num, den):
    return "n/a" if den == 0 else "%.3f" % (num / den)


def confusion(samples, k):
    """Count TP, FN, FP, TN; 'positive' means the screener says BLOCK."""
    tp = fn = fp = tn = 0
    mistakes = []
    for number, (label, text) in enumerate(samples, start=1):
        verdict, report = screen(text, k)
        blocked = verdict == "BLOCK"
        if label == "scam" and blocked:
            tp += 1
        elif label == "scam":
            fn += 1
            mistakes.append(("FN", number, verdict, report, text))
        elif blocked:
            fp += 1
            mistakes.append(("FP", number, verdict, report, text))
        else:
            tn += 1
    return tp, fn, fp, tn, mistakes
```

Listing 5.3 (Continued) *evaluate.py: the confusion matrix, five measures, the list of mistakes and a sweep over k.*

```
def main():
    args = sys.argv[1:]
    k = int(args[args.index("--k") + 1]) if "--k" in args else 2
    path = Path(args[args.index("--data") + 1]) if "--data" in args else DATA
    samples = load(path)
    scams = sum(1 for label, _ in samples if label == "scam")
    tp, fn, fp, tn, mistakes = confusion(samples, k)
    print("samples: %d (%d scam, %d normal), k = %d"
          % (len(samples), scams, len(samples) - scams, k))
    print()
    print("                  predicted BLOCK   predicted not BLOCK")
    print("actual scam       TP = %-12d  FN = %d" % (tp, fn))
    print("actual normal     FP = %-12d  TN = %d" % (fp, tn))
    print()
    print("precision = TP / (TP + FP) = %d / %d = %s"
          % (tp, tp + fp, ratio(tp, tp + fp)))
    print("recall    = TP / (TP + FN) = %d / %d = %s"
          % (tp, tp + fn, ratio(tp, tp + fn)))
    print("F1        = 2TP / (2TP + FP + FN) = %d / %d = %s"
          % (2 * tp, 2 * tp + fp + fn, ratio(2 * tp, 2 * tp + fp + fn)))
    print("accuracy  = (TP + TN) / all = %d / %d = %s"
          % (tp + tn, len(samples), ratio(tp + tn, len(samples))))
    print("FPR       = FP / (FP + TN) = %d / %d = %s"
          % (fp, fp + tn, ratio(fp, fp + tn)))
    print()
    print("mistakes:")
    for kind, number, verdict, report, text in mistakes:
        rules = sorted({name for _, _, fired, _, _, _ in report
                        for _, name, _ in fired})
        print("  %s #%02d %-7s rules: %s" % (kind, number, verdict,
                                             ", ".join(rules) or "(no link)"))
        print("         " + (text if len(text) <= 70 else text[:67] + "..."))
    print()
    print("k   TP  FN  FP  TN  precision  recall  F1")
    for kk in range(1, 6):
        a, b, c, d, _ = confusion(samples, kk)
        print("%d  %3d %3d %3d %3d  %9s  %6s  %s"
              % (kk, a, b, c, d, ratio(a, a + c), ratio(a, a + b),
                 ratio(2 * a, 2 * a + c + b)))


if __name__ == "__main__":
    main()
```

Listing 5.4 `Evaluate.java`: the same evaluation in Java.

```
import java.io.IOException;
import java.nio.charset.StandardCharsets;
import java.nio.file.Files;
import java.nio.file.Path;
import java.util.ArrayList;
import java.util.List;
import java.util.Locale;
import java.util.TreeSet;

/** Measure the screener on our own labeled samples.
 *    java -cp out Evaluate [--k 2] [--data ../data/samples.tsv] */
public class Evaluate {
    record Sample(String label, String text) { }

    record Mistake(String kind, int number, LinkScreener.Result result,
                   String text) { }

    record Counts(int tp, int fn, int fp, int tn, List<Mistake> mistakes) { }

    static List<Sample> load(Path path) throws IOException {
        List<Sample> samples = new ArrayList<>();
        for (String line : Files.readAllLines(path, StandardCharsets.UTF_8)) {
            if (!line.isEmpty() && !line.startsWith("#")) {
                int tab = line.indexOf('\t');
                samples.add(new Sample(line.substring(0, tab),
                                       line.substring(tab + 1)));
            }
        }
        return samples;
    }

    static String ratio(int num, int den) {
        if (den == 0) return "n/a";
        return String.format(Locale.ROOT, "%.3f", (double) num / den);
    }
```

Listing 5.4 (Continued) *Evaluate.java: the same evaluation in Java.*

```
    /** Count TP, FN, FP, TN; "positive" means the screener says BLOCK. */
    static Counts confusion(List<Sample> samples, int k) {
        int tp = 0, fn = 0, fp = 0, tn = 0;
        List<Mistake> mistakes = new ArrayList<>();
        for (int i = 0; i < samples.size(); i++) {
            Sample s = samples.get(i);
            LinkScreener.Result r = LinkScreener.screen(s.text(), k);
            boolean blocked = r.verdict().equals("BLOCK");
            if (s.label().equals("scam") && blocked) {
                tp++;
            } else if (s.label().equals("scam")) {
                fn++;
                mistakes.add(new Mistake("FN", i + 1, r, s.text()));
            } else if (blocked) {
                fp++;
                mistakes.add(new Mistake("FP", i + 1, r, s.text()));
            } else {
                tn++;
            }
        }
        return new Counts(tp, fn, fp, tn, mistakes);
    }

    public static void main(String[] args) throws IOException {
        int k = 2;
        Path path = Files.exists(Path.of("data/samples.tsv"))
                ? Path.of("data/samples.tsv") : Path.of("../data/samples.tsv");
        for (int i = 0; i + 1 < args.length; i += 2) {
            if (args[i].equals("--k")) k = Integer.parseInt(args[i + 1]);
            if (args[i].equals("--data")) path = Path.of(args[i + 1]);
        }
        List<Sample> samples = load(path);
        int scams = (int) samples.stream()
                .filter(s -> s.label().equals("scam")).count();
        Counts c = confusion(samples, k);
        int tp = c.tp(), fn = c.fn(), fp = c.fp(), tn = c.tn(), n = samples.size();
        System.out.printf("samples: %d (%d scam, %d normal), k = %d%n",
                n, scams, n - scams, k);
        System.out.println();
        System.out.println("                 predicted BLOCK"
                + "   predicted not BLOCK");
        System.out.printf("actual scam      TP = %-12d  FN = %d%n", tp, fn);
        System.out.printf("actual normal    FP = %-12d  TN = %d%n", fp, tn);
        System.out.println();
        System.out.printf("precision = TP / (TP + FP) = %d / %d = %s%n",
                tp, tp + fp, ratio(tp, tp + fp));
```

Listing 5.4 (Continued) *Evaluate.java: the same evaluation in Java.*

```
        System.out.printf("recall    = TP / (TP + FN) = %d / %d = %s%n",
                tp, tp + fn, ratio(tp, tp + fn));
        System.out.printf("F1        = 2TP / (2TP + FP + FN) = %d / %d = %s%n",
                2 * tp, 2 * tp + fp + fn, ratio(2 * tp, 2 * tp + fp + fn));
        System.out.printf("accuracy  = (TP + TN) / all = %d / %d = %s%n",
                tp + tn, n, ratio(tp + tn, n));
        System.out.printf("FPR       = FP / (FP + TN) = %d / %d = %s%n",
                fp, fp + tn, ratio(fp, fp + tn));
        System.out.println();
        System.out.println("mistakes:");
        for (Mistake m : c.mistakes()) {
            TreeSet<String> rules = new TreeSet<>();
            for (LinkScreener.LinkReport lr : m.result().links()) {
                for (LinkScreener.Hit h : lr.fired()) rules.add(h.name());
            }
            String names = rules.isEmpty() ? "(no link)" : String.join(", ", rules);
            System.out.printf("  %s #%02d %-7s rules: %s%n",
                    m.kind(), m.number(), m.result().verdict(), names);
            String t = m.text();
            if (t.length() > 70) t = t.substring(0, 67) + "...";
            System.out.println("         " + t);
        }
        System.out.println();
        System.out.println("k   TP  FN  FP  TN  precision  recall  F1");
        for (int kk = 1; kk <= 5; kk++) {
            Counts d = confusion(samples, kk);
            System.out.printf("%d  %3d %3d %3d %3d  %9s  %6s  %s%n", kk,
                    d.tp(), d.fn(), d.fp(), d.tn(), ratio(d.tp(), d.tp() + d.fp()),
                    ratio(d.tp(), d.tp() + d.fn()),
                    ratio(2 * d.tp(), 2 * d.tp() + d.fp() + d.fn()));
        }
    }
}
```

Run it (`python evaluate.py` in the `python` folder, or `java -cp out Evaluate` in the `java` folder; both print the same):

```
samples: 60 (30 scam, 30 normal), k = 2

                 predicted BLOCK   predicted not BLOCK
actual scam      TP = 25            FN = 5
actual normal    FP = 3             TN = 27

precision = TP / (TP + FP) = 25 / 28 = 0.893
recall    = TP / (TP + FN) = 25 / 30 = 0.833
F1        = 2TP / (2TP + FP + FN) = 50 / 58 = 0.862
accuracy  = (TP + TN) / all = 52 / 60 = 0.867
FPR       = FP / (FP + TN) = 3 / 30 = 0.100

mistakes:
  FN #09 CAUTION rules: shortener
         Your parcel of clean clothes is waiting. Confirm the address: s.id/...
  FN #10 SAFE    rules: (no link)
         Hi, this is the Profitina admin. Please send me the OTP code we jus...
  FN #11 SAFE    rules: (no link)
         Transfer 20 PROFIT to u9a8b7c6d and I will send back 200 tomorrow, ...
  FN #23 SAFE    rules: (no link)
         I am staff. Send me your password and I will move your channel to O...
  FN #26 CAUTION rules: brand-elsewhere
         Confirm your email to keep earning PROFIT: https://profitina.com-ve...
  FP #42 BLOCK   rules: no-https, urgency
         Last chance to join the OOP study group tonight: http://kelas.examp...
  FP #43 BLOCK   rules: shortener, urgency
         Claim your seat for the GEO workshop at s.id/workshop-geo
  FP #50 BLOCK   rules: no-https, risky-tld
         Local printer shop for our flyers: http://cetak-murah.xyz

k   TP  FN  FP  TN  precision  recall  F1
1   27   3   6  24      0.818   0.900  0.857
2   25   5   3  27      0.893   0.833  0.862
3   14  16   0  30      1.000   0.467  0.636
4    9  21   0  30      1.000   0.300  0.462
5    8  22   0  30      1.000   0.267  0.421
```

5.8.8 Reading the results

The confusion matrix. With $k = 2$ the screener flags 25 of the 30 scams and 3 of the 30 normal messages: $TP = 25$, $FN = 5$, $FP = 3$, $TN = 27$, the right half of **Figure 5.4**. The program prints the arithmetic of Equation (5.1) to Equation (5.5) itself: precision $25/28 = 0.893$, recall $25/30 = 0.833$, $F_1 = 50/58 = 0.862$, accuracy $52/60 = 0.867$, FPR $3/30 = 0.100$.

The false negatives. Three of the five missed scams (#10, #11 and #23) contain no link at all: an OTP request, a "send me 20 PROFIT and I will send back 200" promise, and a fake staff member asking for a password. No link screener can catch them; they are the moderators' and the reporting system's job (Section 5.6), which is exactly why a community needs more than one defense. The other two are more interesting. Scam #9 uses only a shortener, one soft signal, and gets CAUTION. Scam #26 uses the host `profitina.com-verify.email`, which begins with the real domain and then continues with `-verify.email`; our tokenizer sees the label `profitina`, so `brand-elsewhere` fires, but nothing else does. A rule that noticed an official domain followed by a hyphen, or `com` inside a label, would catch it; that is Exercise 5.9.

The false positives. All three are honest messages that happen to trigger two soft rules: a study-group invitation with "last chance" and a plain-HTTP link (#42), a workshop invitation with "claim your seat" and a shortener (#43), and a local printer on a cheap TLD without HTTPS (#50). Each is a reminder that soft signals are soft. Notice also what did *not* happen: #51, the survey site of the national internet association with four labels in its host, got only CAUTION.

The sweep over k. The last table, drawn in **Figure 5.8**, shows the trade-off that every screener faces.

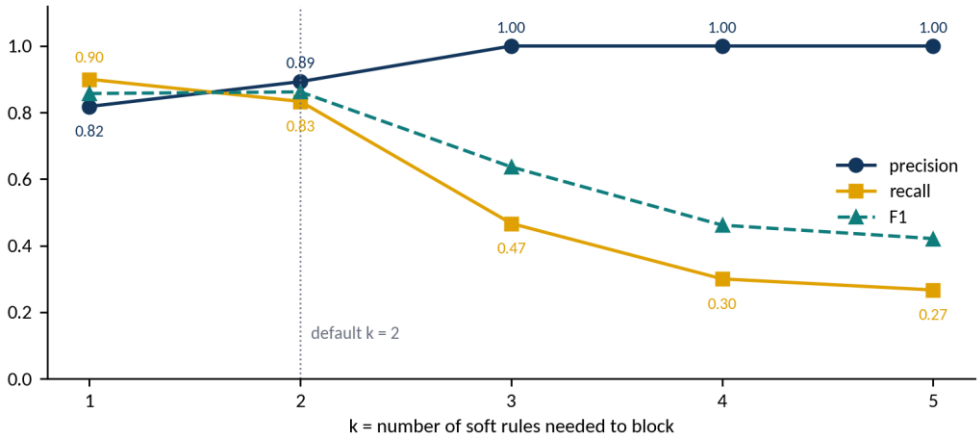


Figure 5.8 Precision, recall and F_1 of the lab screener as the soft threshold k varies from 1 to 5. Lower k catches more scams and blocks more honest messages; from $k = 3$ upwards there are no false positives, but most scams get through.

At $k = 1$ recall rises to 0.900 (27 of 30: every scam that has a link is caught) but the false positive rate doubles to $6/30 = 0.200$. At $k = 3$ there are no false positives at all, but recall falls to 0.467: more than half of the scams pass. The best F_1, 0.862, is at $k = 2$, which is why it is the default. But the sweep also shows why F_1 is not the whole story: on a real feed, where scams are rare, Equation (5.7) says that an FPR of 0.100 would make most blocks wrong. A production screener would need far better rules, more normal samples, and a second opinion, which is exactly what Profitina's layered design provides.

Comparing with Profitina. Profitina's published benchmark (27 of 28 scams caught, 0 of 30 normal texts flagged) is better on every measure than our lab screener. That should not surprise anyone: a production rule floor refined on real traffic will beat eleven textbook rules written for a chapter. But, as the warning in Section 5.5.5 says, the two results come from different sample sets, and neither set is large. The point of putting them side by side is the method: the same four counts, the same formulas, and the same honesty about what 30 normal samples can prove.

5.8.9 Another view: points on an ROC plot

The sweep over k can also be read the way the classification literature usually reads it, as points on a *receiver operating characteristic* (ROC) plot [10]. Each value of k gives one pair: the false positive rate on the horizontal axis and the recall, which in this setting is called the true positive rate, on the vertical axis. From the evaluation output, $k = 1$ gives the point (0.200,0.900), $k = 2$ gives (0.100,0.833), $k = 3$ gives (0.000,0.467), $k = 4$ gives (0.000,0.300) and $k = 5$ gives (0.000,0.267). A perfect screener would sit at (0,1), the top-left corner; a screener that flags at random sits on the diagonal from (0,0) to (1,1).

Two things are visible in these five points. First, no setting of k reaches the corner: lowering the threshold buys recall only by paying in false positives, which is the trade-off of **Figure 5.8** in another form. Second, the points for $k = 3$, 4 and 5 all lie on the vertical axis: once k is at least 3, our screener never flags a normal message on this sample set, and raising k further only loses scams. Profitina's published benchmark, drawn as a point on the same axes, would sit at (0.000,0.964), close to the corner. The ROC view makes one more lesson clear: to move a screener towards the corner, adjusting a threshold is not enough; the rules themselves must get better at

separating the two classes, which is exactly what the exercises at the end of this chapter ask you to do.

5.8.10 What the lab leaves out

Our screener never visits the links it screens, so it cannot see where a shortener leads, what a page contains or whether a domain was registered yesterday. It knows nothing about reputation lists maintained by browser makers and security companies. Its brand list has three entries, and its sample set has 60 lines written by one hand. Each of these is a direction in which a real system grows. The layered idea, a fast, explainable floor with optional opinions on top, all decided once on the write path, stays the same.

5.9 Profitina in Practice

Post a link. In the live application, a link in a community post is screened when you post it. A safe link appears as a clickable link; a malicious one is blocked [7].

Look at a course channel. Course channels have moderators, often the lecturer, whose pinned announcements appear at the top. Hidden posts disappear from view; the person who wrote them is notified.

Report something. Every post and every DM has a report action. A DM report freezes a copy of that one message for review; nothing else in the conversation becomes visible.

Watch the rewards. A useful post earns a small amount of PROFIT after it settles; a duplicate earns 0.00 and points to the earlier post; nothing earns anything until your email is verified. The caps reset at 00:00 WIB.

Live and roadmap. The rule floor, the optional local-AI opinion on the fast lane, channel moderators, reporting with frozen copies, and the fair-play layers are all live in the September 2026 release [7]. Questions about safety go to `info@profitina.com`.

5.10 Common Misunderstandings

"An AI model should decide whether a link is safe." A model is a useful second opinion, but a safety check must answer even when the model is busy, must explain its verdicts, and must be testable. That is the job of the rule floor (Section 5.4.2).

"Safety checks belong where content is displayed." Content is written once and read many times; checking on the read path multiplies the cost by the number of readers and puts it in front of every reader. Decide on the write path and store the verdict (Section 5.4.3).

"A link that starts with the brand's name is the brand's link." Only the host decides. In `https://profitina.com.account.secure-login.buzz/verify` the registered domain is `secure-login.buzz` (Section 5.3.1).

"Accuracy tells you how good a screener is." On a feed where 99 % of messages are normal, a screener that passes everything is 99 % accurate. Report recall and the false positive rate (Section 5.5.2).

"Zero false positives means the screener never blocks honest people." Zero in 30 bounds the rate below about 10 % with 95 % confidence, by Equation (5.6); it does not make it zero (Section 5.5.4).

"A precision of 0.89 on a benchmark means 89 % of real blocks are right." Only if real traffic has the benchmark's mix. With rare scams, Equation (5.7) can turn the same screener's real precision into 0.15 (Section 5.5.5).

"Moderators are small administrators." Their powers are three, inside one channel, logged and visible to the person affected; they never touch PROFIT or private data (Section 5.6.2).

"Publishing the fair-play thresholds would be more transparent." It would make each threshold a target. The layers and their purpose are public; their settings are tuned and kept private (Section 5.7.4).

5.11 Chapter Summary

- A community faces three threats: malicious links, abuse by people, and abuse by automation. In 2026 phishing reached 971,181 attacks in one quarter, Indonesia's Anti-Scam Centre had received 636,014 reports, and 53 % of web traffic was automated (**Figure 5.1**).
- Only the host of a URL decides whose server you reach. The classic tricks, IP hosts, user information before `@`, punycode, one-letter look-alikes, brands on other domains, stacked subdomains, shorteners, plain HTTP, cheap TLDs and urgency, split into hard and soft signals (**Table 5.1**).
- Profitina screens links in layers: a deterministic rule floor that combines several signals, with an optional local-AI opinion on the fast GPU lane on top. Both run once on the write path; the read path only reads the stored verdict (**Figure 5.3**).
- A screener is measured with a confusion matrix and precision, recall, F_1, accuracy and the false positive rate (Equation (5.1)–Equation (5.5)). Profitina's public benchmark, 27 of 28 scams caught with 0 of 30 false positives, gives precision 1.000 and recall 0.964; by the rule of three (Equation (5.6)) its false positive rate is below about 10 % with 95 % confidence, and with rare scams the precision members experience depends heavily on it (Equation (5.7)).
- Channel moderators hide, pin and mute inside one channel only, with every action logged and the poster notified, and never touch PROFIT or private data. Reports freeze a copy of the reported message.
- Fair play keeps rewards for humans through layers that multiply (Equation (5.8)): email OTP, earning only when verified, small rewards with daily caps, zero for duplicates, a local-AI usefulness score, escrow with clawback and burn, recalculation on edit, and ranks that cannot be bought.
- Lab G3 built an eleven-rule screener in Python and Java with the decision rule Equation (5.9) and the edit distance Equation (5.10), and measured it on 60 labeled samples: at $k = 2$, precision 0.893, recall 0.833, F_1 0.862, FPR 0.100; the sweep over k showed the trade-off, and every mistake was explained.

5.12 Exercises

Exercise 5.1. For each link, name the host and the registered domain, and say which rules of **Table 5.2** fire: (a) `https://login.dompetkita.example/secure`; (b) `http://dompetkita.example.secure-pay.top/login`; (c) `https://www.profitina.com/blog`.

Exercise 5.2. Compute by hand, with the table method of **Table 5.3**, the edit distance between `dana` and `danna`, and between `kilat` and `klilat`. Which of them would the `lookalike` rule consider one letter away?

Exercise 5.3. A screener is tested on 40 scams and 60 normal messages and flags 34 scams and 3 normal messages. Write its confusion matrix and compute precision, recall, F_1, accuracy and FPR with Equation (5.1) to Equation (5.5).

Exercise 5.4. A benchmark contains 200 normal messages and none of them is flagged. Using Equation (5.6), give the exact and approximate 95 % upper bound on the false positive rate. How many normal messages with zero false positives would be needed to push the bound below 0.5 %?

Exercise 5.5. For the screener of Exercise 5.3, compute the positive predictive value with Equation (5.7) when 1 % of link-carrying messages are scams. What false positive rate would be needed, with the same recall, to make at least half of all blocks correct?

Exercise 5.6. Explain in your own words why Profitina's optional AI layer runs on the write path, and what the feed would pay if it ran on the read path. Use the numbers of Section 4.11.

Exercise 5.7. Run `python evaluate.py --k 1` and list the six false positives. For each, say which soft rules fired and whether you think the message should be shown with a warning or shown normally.

Exercise 5.8. Add a twelfth, soft rule `at-in-path` that fires when the *path* contains `@`. Predict its effect on the 60 samples before running the evaluation, then check. Why is this rule soft, when `userinfo` is hard?

Exercise 5.9. Scam #26 (`profitina.com-verify.email`) is missed at $k = 2$. Design one new rule that catches it without creating a false positive on the 30 normal samples. State it precisely, say whether it is hard or soft, and name one legitimate link it might still block.

Exercise 5.10. Write five new normal messages about Profitina Laundry that the current screener would wrongly block at $k = 2$, and five new scams that it would miss. What does this exercise teach about sample sets written by the same person who wrote the rules?

Exercise 5.11. A lecturer who moderates a course channel asks for the power to "see which account is behind a handle" when a post breaks the rules. Using Section 5.6.2, write a short answer explaining why moderators do not have that power and what they can do instead.

Exercise 5.12. A script creates 100 accounts and posts from each of them. Walk through the layers of **Figure 5.6** and say where each layer slows the script down. Which layer stops it earliest, and which catches it latest?

5.13 Solutions to the Exercises

Solution 5.1. (a) Host `login.dompetkita.example`, registered domain `dompetkita.example`, which is official; the host ends with `.dompetkita.example`, so brand and credential rules are skipped, and with HTTPS, a normal TLD and three labels no rule fires: SAFE. (b) Host `dompetkita.example.secure-pay.top`, registered domain `secure-pay.top`. Rules: `no-https`, `risky-tld` (`.top`), `deep-subdomain` (four labels), `brand-elsewhere` (`dompetkita` on a non-official host) and `credential-path` (`login`): five soft rules, BLOCK. (c) Host `www.profitina.com`, which ends with `.profitina.com` and is official: SAFE.

Solution 5.2. `dana` to `danna`: one insertion, distance 1. `kilat` to `klilat`: insert `l` after `k` gives `klilat`, distance 1. Both are one letter away; the `lookalike` rule would fire only if the string were a host token and the other were on the protected brand list.

Solution 5.3. $TP = 34$, $FN = 6$, $FP = 3$, $TN = 57$. Precision $34/37 \approx 0.919$; recall $34/40 = 0.850$; $F_1 = 68/(68 + 3 + 6) = 68/77 \approx 0.883$; accuracy $91/100 = 0.910$; FPR $3/60 = 0.050$.

Solution 5.4. Exact: $1 - 0.05^{1/200} = 1 - e^{-2.996/200} = 1 - e^{-0.01498} \approx 0.0149$. Approximate: $3/200 = 0.015$. To get below 0.005 we need $3/n < 0.005$, so $n > 600$; with the exact formula, $n \geq 598$. In practice, about 600 normal messages with no false positive.

Solution 5.5. With recall 0.850, FPR 0.050 and $\pi = 0.01$: PPV $= 0.0085/(0.0085 + 0.0495) = 0.0085/0.058 \approx 0.147$. For PPV at least 0.5 we need recall $\cdot\, \pi \geq$ FPR $\cdot\, (1 - \pi)$, so FPR $\leq 0.0085/0.99 \approx 0.0086$: below about 0.9 %, roughly six times lower than now.

Solution 5.6. On the write path the model is asked once per post, while only the poster waits, and the verdict is stored. On the read path it would be asked every time any member opens a feed containing the post, while that member waits. Section 4.11 measured a feed page of 30 posts at 868 ms with six model calls on the read path (plus the other problems described there), against 11 ms after the read path stopped calling the model and the queries were fixed. A thousand readers would multiply the model calls by a thousand; on the write path the number of calls is the number of posts.

Solution 5.7. At $k = 1$ the output lists six false positives: #39 (shortener), #40 (no-https), #42 (no-https, urgency), #43 (shortener, urgency), #50 (no-https, risky-tld) and #51 (deep-subdomain). A reasonable judgment: #39, #40 and #51 should be shown normally or with a mild warning, since a single soft signal is common in honest posts; #42, #43 and #50 are honest but carry two signals, so a warning is defensible. This is exactly why the default is $k = 2$ and why CAUTION exists.

Solution 5.8. In Python, after the `credential-path` check add `if "@" in path: fired.append(("soft", "at-in-path", "the path contains '@'"))`, and the equivalent in Java. None of the 60 samples has `@` in a path, so the evaluation does not change: a useful reminder that a rule can only be judged on samples that exercise it. It is soft because `@` appears legitimately in paths, for example in profile pages such as `/@username` on some platforms, whereas `@` in the *site part* has no honest use in a shared link.

Solution 5.9. One possibility: a soft rule `official-prefix` that fires when the host *starts with* an official domain followed by a character other than `.` or the end of the host, for example `profitina.com-`; or, more generally, when a label other than the last one is exactly `com` or begins with `com-`. On the 30 normal samples it never fires, and on #26 it adds a second soft rule, so #26 then has two soft rules (`brand-elsewhere`, `official-prefix`) and is blocked at $k = 2$, with no new false positive. A legitimate link it might still block: a partner whose domain happens to begin with the brand's domain name, such as a hypothetical `profitina.community-events.example`; that is why it is soft.

Solution 5.10. Answers vary. Typical wrongly blocked normals: a local shop on a cheap TLD without HTTPS; a shortened link to a registration form with "last chance"; a message quoting a scam as a warning ("Beware of http://profitina-support.xyz/login!"). Typical missed scams: a scam on an ordinary `.com` domain with HTTPS and no brand; a scam that asks the reader to type a code into a reply; a scam split across two messages. The lesson: whoever writes the rules also knows, consciously or not, which samples they will catch. Independent samples, ideally real reports, are needed for an honest evaluation.

Solution 5.11. One possible answer: "Moderators look after what is said in the channel, not who says it. Linking a handle to an account's private details would give every moderator access to

personal data that UU PDP requires us to protect, and would make a lecturer's channel less safe for students to speak in. What you can do is hide the post, mute the member for a while, and pin a reminder of the rules; the member is notified, the action is logged, and if the problem is serious you can report it for review by the platform."

Solution 5.12. Email OTP: each of the 100 accounts needs a working mailbox and a code, which slows mass creation and stops unverified accounts from earning anything. Daily caps: each account earns at most the capped amount per action per day, however much it posts. Duplicates: identical or near-identical posts across accounts earn 0.00 and point to the earlier post. Usefulness score: trivial generated text earns 0.00. Escrow and clawback: rewards found later to be abuse are taken back and burned. The earliest stop is email verification, before any reward; the latest is clawback, after rewards were provisionally credited.

References

[1] Anti-Phishing Working Group, *Phishing Activity Trends Report, 1st Quarter 2026*, May 27, 2026. [Online]. Available: https://docs.apwg.org/reports/apwg_trends_report_q1_2026.pdf

[2] ANTARA, "OJK: IASC memulihkan dana korban penipuan Rp204,3 miliar hingga Juli," Aug. 4, 2026. [Online]. Available: https://www.antaranews.com/berita/5679972/ojk-iasc-memulihkan-dana-korban-penipuan-rp2043-miliar-hingga-juli

[3] Imperva (Thales), "Bad Bot Report 2026: Bots in the Agentic Age," Apr. 29, 2026. [Online]. Available: https://www.imperva.com/blog/bad-bot-report-2026-bots-agentic-age/

[4] T. Berners-Lee, R. Fielding, and L. Masinter, "Uniform Resource Identifier (URI): Generic Syntax," IETF RFC 3986, Jan. 2005.

[5] A. Costello, "Punycode: A Bootstring encoding of Unicode for Internationalized Domain Names in Applications (IDNA)," IETF RFC 3492, Mar. 2003.

[6] Unicode Consortium, "Unicode Security Mechanisms," Unicode Technical Standard #39, version 18.0.0, Aug. 27, 2026. [Online]. Available: https://www.unicode.org/reports/tr39/

[7] Profitina, *Profitina Whitepaper — Public Version v24*, Sep. 2026. [Online]. Available: https://app.profitina.com/static/whitepaper/

[8] Profitina, "Current facts for course material," Profitina Course Library, Sep. 23, 2026.

[9] C. D. Manning, P. Raghavan, and H. Schütze, *Introduction to Information Retrieval*. Cambridge, U.K.: Cambridge University Press, 2008, ch. 8.

[10] T. Fawcett, "An introduction to ROC analysis," *Pattern Recognition Letters*, vol. 27, no. 8, pp. 861–874, 2006.

[11] J. A. Hanley and A. Lippman-Hand, "If nothing goes wrong, is everything all right? Interpreting zero numerators," *JAMA*, vol. 249, no. 13, pp. 1743–1745, 1983.

[12] The Santa Clara Principles on Transparency and Accountability in Content Moderation, 2nd ed., Dec. 2021. [Online]. Available: https://santaclaraprinciples.org/

[13] M. Strathern, "'Improving ratings': audit in the British University system," *European Review*, vol. 5, no. 3, pp. 305–321, 1997.

[14] V. I. Levenshtein, "Binary codes capable of correcting deletions, insertions, and reversals," *Soviet Physics Doklady*, vol. 10, no. 8, pp. 707–710, 1966.

[15] R. A. Wagner and M. J. Fischer, "The string-to-string correction problem," *Journal of the ACM*, vol. 21, no. 1, pp. 168–173, 1974.

Chapter 6

Local AI: The Content Engine and the Two-Lane Model

By the end of this chapter you will be able to explain, in plain words, what a large language model does when it "writes"; say why Profitina runs every piece of its artificial intelligence on its own machine and never sends a user's data to a hosted AI service, and what that choice costs; describe Ollama and the Qwen family of models as they stand in September 2026; draw the two-lane model, with a quality lane for long drafts and a fast lane for short labels, and say which work belongs in which lane and why some work belongs in neither; explain the quality gate, the rule that an AI draft is served only when it beats a built-in template, and read the public result of 5 wins in 6 benchmark cases without overreading it; and design software that says "no model available" instead of pretending. In the lab you will write a client for Ollama's HTTP interface in Python and in Java, a small stub server that imitates Ollama so that everything runs even on a computer with no model at all, a two-lane router, and a quality gate of your own.

6.1 Where We Left Off

Chapter 5 ended with a small but important sentence about Link Safety: every link posted in the community is screened first by a *rule floor*, a set of plain, deterministic checks, and an optional local-AI opinion may sit on top of that floor, never underneath it. The rule floor answers every time, in microseconds, and gives the same answer for the same input. The AI opinion is slower, occasionally unavailable, and sometimes wrong, so the product is built to work without it.

That sentence contains the whole attitude of this chapter in miniature. Profitina uses artificial intelligence seriously: its Content Engine writes answer-first drafts, its scanner uses a model to suggest a business category, its Fair-Play system uses a model to judge whether a post is useful, and Link Safety may ask a model for a second opinion. Yet the product never treats a model as a magic box that is always there and always right. A model is a component with a speed, a failure rate and a quality level, all of which can be measured, and the software around it is designed from those measurements.

We start from the ground up. Section 6.2 explains what a language model is for a reader who has never looked inside one. Section 6.3 explains why Profitina insists on running its models locally. Section 6.4 introduces Ollama, the program that serves the models, and the Qwen family of models it serves. Sections 6.5 to 6.7 describe the three ideas that make local AI dependable in a product: two lanes, a quality gate, and honesty when no model is available. Section 6.8 is the lab, and Section 6.9 returns to the real product.

6.2 What a Language Model Does, in Plain Words

6.2.1 Predicting the next piece of text

Imagine a very well-read shop assistant at Profitina Laundry who has read millions of pages of text: receipts, recipes, newspaper articles, product pages, conversations. You say to the assistant,

"Profitina Laundry is a laundry service in", and stop. Almost without thinking, the assistant continues: "Surabaya." It has not looked anything up. It has simply learned, from all that reading, which words tend to follow which other words, and it has noticed from the earlier part of your sentence that you are talking about a place.

A *large language model* (LLM) is a computer program that does exactly this, and nothing more mysterious: given a piece of text, it predicts a likely continuation, one small piece at a time. The small pieces are called *tokens*. A token is often a whole common word ("laundry"), sometimes part of a rarer word ("Profit" + "ina"), sometimes a single punctuation mark. When the model has predicted one token, the token is added to the text, and the model predicts the next one from the longer text. Repeating this step a few hundred times produces a paragraph.

> **DEFINITION.** A *language model* assigns probabilities to the possible next *tokens* of a text. *Generation* is the loop that repeatedly picks one next token, appends it, and asks again. A *large* language model is one with billions of adjustable numbers, called *parameters*, learned from a very large collection of text.

The modern design behind nearly all such models is the *transformer*, introduced in 2017 [1]. You do not need its mathematics to follow this book; you need three consequences of it.

First, the model's knowledge is frozen in its parameters at the moment training ended. A model does not know the opening hours of your laundry unless you tell it in the request, and it may confidently state wrong hours if you do not. This is why every serious use of a model in Profitina hands the model the facts it needs, in the request, instead of trusting its memory.

Second, generation is a loop, so time grows with the length of the answer. A one-word label is fast; a four-hundred-word article is slow. Section 6.5 builds a whole architecture on this single fact.

Third, generation involves chance. At each step the model has a *distribution* of likely next tokens, and the software picks one; a setting called *temperature* controls how adventurous that pick is, and a *seed* makes the choice repeatable. Two runs with the same request can produce different text. A product therefore cannot test a model once and assume the answer forever; it has to check each answer, which is what the quality gate of Section 6.6 does. Practitioners know this well: in the 2025 Stack Overflow Developer Survey, 84% of respondents used or planned to use AI tools, yet more of them distrusted the accuracy of those tools (46%) than trusted it (33%) [2].

6.2.2 Parameters, memory and quantization

A model's size is quoted as a number of parameters: "7B" means about seven billion. Every parameter is a number, and every number takes memory. If a model has N parameters and each is stored with b bits, the weights need about

$$M \approx \frac{N \cdot b}{8} \text{ bytes} \tag{6.1}$$

of memory, because eight bits make one byte. Training normally uses 16 bits per parameter. Running a model at home or on a single server usually uses *quantization*: each parameter is rounded to a coarser number with only 4 or 5 bits, which shrinks the model to about a quarter or a third of its size and costs surprisingly little quality [3]. Ollama distributes its models in a quantized file format called GGUF, which comes from the llama.cpp project [4].

> **EXAMPLE.** Qwen2.5 7B has about $N = 7.6 \times 10^9$ parameters. At 16 bits, Equation (6.1) gives $7.6 \times 10^9 \times 16/8 = 15.2 \times 10^9$ bytes, about 15 GB, which does not fit on most laptop graphics cards. With 4-bit quantization plus a little overhead, say $b \approx 4.9$ bits on average, it gives

> $7.6 \times 10^9 \times 4.9/8 \approx 4.7 \times 10^9$ bytes. That matches the 4.7 GB download that Ollama's library lists for `qwen2.5:7b` [5].

The same arithmetic explains why a very small model such as Qwen3 1.7B fits comfortably on a modest graphics card, and why a 27-billion-parameter model needs a workstation. Size is not quality, but size decides where a model can run, and where it runs decides how fast it answers.

In plain language: a model is a big table of learned numbers plus a loop that predicts the next token. The table's size decides the hardware; the loop's length decides the waiting time; and the chance in each step decides that every answer must be checked.

6.3 Why Local-Only Inference: Privacy, Cost, Control

Inference means running a trained model to get an answer, as opposed to *training* it. There are two ways to do inference in a product. You can send the text over the internet to a company that runs a model for you and charges per request (a *hosted* or *cloud* model), or you can run the model yourself on a machine you control (a *local* or *self-hosted* model). Profitina chose the second way for every feature, and removed the code paths for hosted AI entirely: user data never leaves the server for AI processing [6]. This section explains the three reasons, and then the honest costs.

6.3.1 Privacy

A brand owner who scans a website with Profitina is telling the platform which business they run, which competitors worry them, which questions their customers ask, and where their weaknesses are. A community member who posts in a course channel is writing under a handle but still as a person. Sending that material to a third-party AI service would copy it to another company's servers, under another company's retention rules, sometimes in another country. The whitepaper names this directly as part of the problem Profitina exists to solve: "sending brand and competitive data to third-party APIs creates leakage and compliance risk" [6].

This is not an abstract worry. In Cisco's 2025 privacy benchmark, 64% of 2,600 privacy and security professionals said they worry about inadvertently sharing sensitive information publicly or with competitors through generative AI, and nearly half admitted entering personal or non-public data into such tools [7]. Cisco's 2026 study, published on January 26, 2026 with 5,200 respondents in 12 markets, found that 86% of organizations believe locally stored data is inherently more secure and 81% face heightened demand for data localization [8].

Indonesia's Personal Data Protection Law, UU PDP No. 27 of 2022, gives the same intuition legal force. Its data-minimization principle says personal data should be processed only as far as the purpose requires, cross-border transfers need equivalent protection, and Article 10 gives every data subject the right to object to decisions based solely on automated processing, including profiling, that have legal or similarly significant effects [9]. Chapter 11 treats the law in detail. For this chapter the point is simple: the easiest personal data to protect is the data you never send anywhere.

6.3.2 Cost

A hosted model charges per request, or per token. A local model costs money once, for the hardware, and then a small amount per request for electricity. Write n for the number of requests in a period, p for the hosted price per request, F for the fixed cost of the local hardware over the same period and e for the local running cost per request. Then

$$C_{\text{hosted}}(n) = n\,p, \qquad C_{\text{local}}(n) = F + n\,e \tag{6.2}$$

and the local option becomes cheaper once $F + ne < np$, that is, once

$$n > n^{*} = \frac{F}{p - e} \quad (p > e). \tag{6.3}$$

> **EXAMPLE.** The numbers here are made up and round, to practice Equation (6.3), not to describe any real price. Suppose a machine costs $F = \text{Rp}15{,}000{,}000$ for a year of service, electricity costs $e = \text{Rp}5$ per request, and a hosted service would charge $p = \text{Rp}50$ per request. Then $n^{*} = 15{,}000{,}000/(50 - 5) \approx 333{,}334$ requests a year, a little over 900 a day. Below that volume the hosted service is cheaper; above it the local machine is. Now change one assumption: if the machine is already paid for because it runs the web application anyway, the extra F for AI is close to zero and local inference wins almost at once.

Profitina's house rule adds a second, less obvious cost argument: the product uses no paid third-party APIs at all. A startup whose every feature calls a per-token service has a cost that grows with its own success, can be repriced by a supplier overnight, and cannot offer a generous free tier without fear. A startup that owns its inference can offer 15 free scans a day (Chapter 3) without a meter running in someone else's data center.

6.3.3 Control

The third reason is control. A hosted model can be renamed, changed, rate-limited or retired by its owner, and your product changes with it whether you like it or not. A local model is a file. The version that passed your tests on Tuesday is the version that answers on Friday. You decide when to try a newer model, you measure it against the old one on your own examples, and you switch only if it wins, which is exactly the discipline of the quality gate. Control also means working when the internet does not: a local model on the same machine as the application has no network hop to fail.

There is a fourth, quieter benefit: independence of measurement. Profitina's job is to estimate how ready a website is to be cited by ChatGPT, Gemini and Perplexity. Those engines are the *targets* of the measurement, not components of it. A tool that asked the engines themselves would depend on them, pay them, and see only what they choose to show at that moment. Keeping the AI local keeps the measurement independent (Chapter 2 and Chapter 3 explain the readiness signals themselves).

6.3.4 The honest costs of going local

Local inference is not free of trouble, and a textbook that only listed advantages would be selling, not teaching.

- **A quality ceiling.** The largest hosted models are larger than anything a small team can run on one server. A local 7B model writes good plain paragraphs but will not match a frontier model on hard reasoning. The quality gate exists partly because of this ceiling.
- **Hardware and operations.** Someone must install the model server, watch its memory, restart it, and keep enough disk for model files. Chapter 9 shows how Profitina's health check reports the state of the local AI next to the database and the disk.
- **Speed.** A model on a CPU produces a handful of tokens per second. That is why there are two lanes (Section 6.5).

- **Waiting in line.** One machine can run only so many generations at once, so long jobs must queue, and the product must tell users honestly where they are in the queue.

The decision, then, is not "local is better". It is: for a platform that handles brand and personal data in Indonesia, that must stay affordable, and that must keep its measurements independent, the benefits of local inference outweigh these costs, and each cost is handled by a design idea in the rest of this chapter.

6.4 Ollama and the Qwen Family

6.4.1 Ollama: a model server on your own machine

A model file on disk does nothing by itself. Something must load it into memory, feed it text, run the generation loop, and hand back the tokens. *Ollama* is a free, open-source program that does this job. You install it once, download models with a command such as `ollama pull qwen2.5:7b`, and from then on Ollama listens on your own computer for requests over HTTP, by default on port 11434. Any program that can send an HTTP request, in any language, can then use the model. As of September 24, 2026 the latest stable release is v0.34.3, published on September 19, 2026, with a v0.34.4 release candidate out on September 23 [10]. Ollama changes quickly; the part this chapter relies on, the HTTP interface, has stayed stable across many releases.

The heart of that interface is one endpoint, `POST /api/generate` [11]. The request is a JSON object with a `model` name and a `prompt`; optional fields include `stream` (true by default, meaning the answer arrives piece by piece), `format` (set it to `"json"` or give a JSON schema to ask for structured output), `system` (instructions that frame the request), and `options`, which holds generation settings such as `temperature`, `seed`, `num_predict` (the maximum number of tokens to generate) and `num_ctx` (the size of the context window). The response carries the generated text in `response`, a `done` flag, and timing fields measured in nanoseconds: `total_duration`, `load_duration`, `prompt_eval_duration` and `eval_duration`, together with token counts `prompt_eval_count` and `eval_count` [11]. Two other endpoints matter for this chapter: `GET /api/tags` lists the models installed on the server, and `GET /api/version` reports the server's version. Section 6.8 uses all three.

> **KEY INSIGHT:** Call Ollama at `127.0.0.1`, not at `localhost`. The two names usually mean the same machine, but on Windows `localhost` is resolved to the IPv6 address first, and the first attempt waits before falling back to IPv4, where Ollama listens. On Profitina's production Windows host this single change cut the time to the first token from 2.25 seconds to 0.17 seconds in the September 2026 release. The lab code in this chapter uses `127.0.0.1` everywhere for the same reason.

6.4.2 The Qwen family, as of September 2026

Qwen is a family of language models published by the Qwen team at Alibaba Cloud. Many of its members are released with *open weights*: anyone may download the parameter files and run them locally, under a license stated with each model, most often Apache 2.0. That property, not fashion, is why Profitina uses Qwen: a model you cannot download cannot run on your own server. **Table 6.1** summarizes the generations that matter for this book, checked on September 24, 2026.

Table 6.1 Qwen generations relevant to local inference, as of September 24, 2026. "MoE" (mixture of experts) models such as 30B-A3B have 30 billion parameters in total but use only about 3 billion for each token.

Generation	Released	Open-weight sizes (examples)	Notes
Qwen2.5	Sep 2024	0.5B, 1.5B, 3B, 7B, 14B, 32B, 72B	18 trillion training tokens; Apache 2.0 except 3B and 72B [12], [5]
Qwen3	Apr 2025	0.6B, 1.7B, 4B, 8B, 14B, 32B; MoE 30B-A3B, 235B-A22B	"thinking" and "non-thinking" modes; 119 languages; Apache 2.0 [13], [14]
Qwen3.5	Feb–Mar 2026	0.8B, 2B, 4B, 9B, 27B, 35B, 122B; 397B-A17B	text and images; 256K-token context on Ollama [15], [16]
Qwen3.6	Apr 2026	27B; 35B-A3B	Apache 2.0 [16]
Qwen3.8	Aug 2026	27B; 2.4T-A95B	27B released August 14, 2026 [17]

Two remarks keep the table honest. First, not every Qwen release is open: Qwen3.7-Plus, for example, was offered only as a hosted service [16]. Second, newer is not automatically better for a given job. A newer family may be multimodal, may reason more, or may support longer inputs, and all of that can cost speed or memory that a single server cannot spare.

Figure 6.1 shows the download sizes that Ollama's library lists for several Qwen models, with the two that Profitina's lanes use highlighted. Notice how Equation (6.1) predicts the pattern: sizes grow roughly in proportion to the number of parameters.

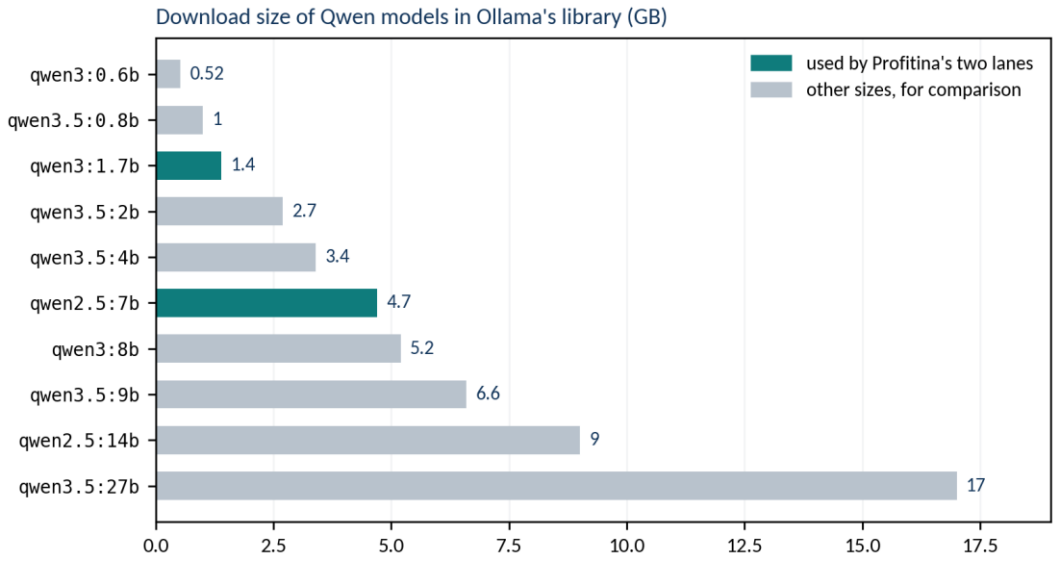


Figure 6.1 Download sizes of Qwen models in Ollama's library, in gigabytes, as listed on September 24, 2026 [15], [5], [14]. The two teal bars are the models behind Profitina's quality lane and fast lane.

6.4.3 Why Profitina still runs Qwen2.5 7B

A reader who has just seen **Table 6.1** may ask why a product in September 2026 runs a model family from 2024 in its quality lane, next to a 2025 model in its fast lane. The answer is the discipline this chapter teaches: a model is replaced when a measurement says the replacement is better at *this product's* jobs, on *this product's* hardware, and not before. Qwen2.5 7B writes plain, factual Indonesian and English paragraphs at a speed the server can sustain, and the Content Engine's gate (Section 6.6) measures what it writes. Qwen3 1.7B is small enough to answer short classification questions in a fraction of a second on a modest graphics card. Either choice can change; the change would come through the same gate, as a candidate that must win on the product's own benchmark, never because a newer name appeared.

Profitina also offers users no choice of model. That is deliberate. A user who picks a model is being asked to judge something they cannot see; the platform, which can measure, takes that responsibility.

6.5 The Two-Lane Model

6.5.1 Two kinds of work

Look at what Profitina asks its models to do, and the jobs fall into two groups.

- **Long jobs** produce text that people will read and publish: the Content Engine's answer-first draft with its FAQ and structured data, or a version of one text adapted to one of six social platforms in *One Text, Many Platforms*. These answers are hundreds of tokens long, their quality matters a great deal, and a user can wait for them if the wait is shown honestly.
- **Short jobs** produce a label that software uses immediately: which category a business belongs to when the scanner is unsure, whether a community post is useful or trivial for Fair-Play, whether a link looks like a scam for Link Safety's optional opinion. These answers are a few tokens long, and they sit in the middle of something the user is already doing, so they must be fast or absent.

Putting both kinds of work on one model in one queue gives the worst of both worlds: the short job waits behind the long one, and the user who posted a message waits for someone else's article to finish. Profitina therefore runs two *lanes*, shown in **Figure 6.2**.

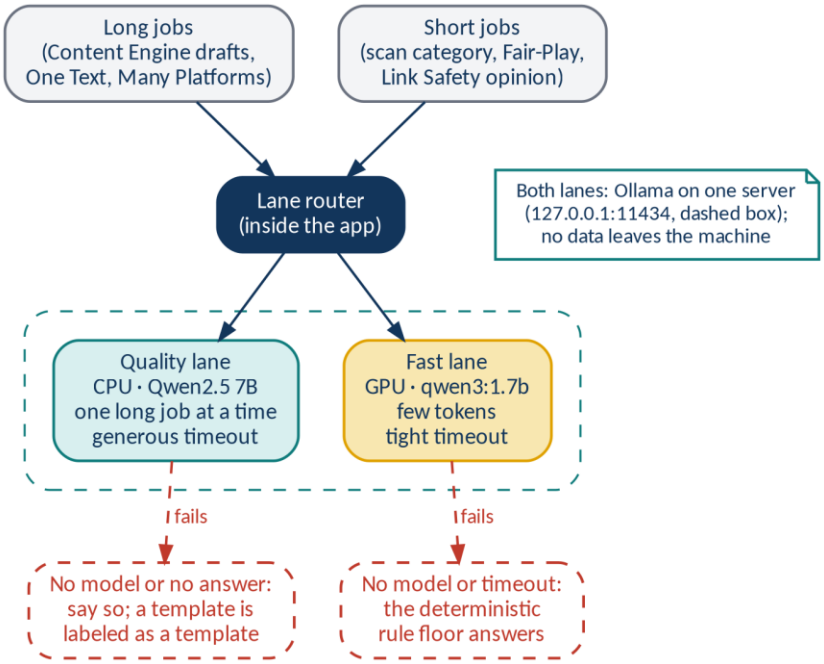


Figure 6.2 The two-lane model. Long jobs go to the quality lane, short jobs to the fast lane, both served by Ollama on the same machine. Each lane has its own honest failure path.

> **DEFINITION.** A *lane* is a route for AI work with its own model, hardware, output length, timeout and failure behavior. The *quality lane* runs a larger model for long, carefully checked text; the *fast lane* runs a very small model for short labels under a tight time limit.

Table 6.2 sets the two lanes side by side as they run in the September 2026 release.

Table 6.2 The quality lane and the fast lane in the September 2026 release.

	Quality lane	Fast lane
Model	Qwen2.5 7B	qwen3:1.7b
Hardware	CPU	GPU
Typical output	hundreds of tokens	a few tokens
Example jobs	Content Engine drafts, One Text, Many Platforms	scan category, Fair-Play usefulness, Link Safety opinion
Concurrency	one long job at a time, visible queue	short jobs, answered quickly
When it fails	say so; a template is labeled as a template	the deterministic rule floor answers

6.5.2 Why the lengths matter: a little arithmetic

The time a generation takes is roughly the time to load the model (zero if it is already in memory), plus the time to read the prompt, plus the time to produce the answer. If the answer has n tokens and the model produces r tokens per second, then

$$T \approx T_{\text{load}} + T_{\text{prompt}} + \frac{n}{r}. \tag{6.4}$$

Ollama reports r directly: the number of generated tokens is `eval_count` and the time spent generating them is `eval_duration` in nanoseconds, so

$$r = \frac{\text{eval\_count}}{\text{eval\_duration}/10^9} \text{ tokens per second.} \tag{6.5}$$

EXAMPLE. Suppose, for illustration, that a 7B model on a CPU produces $r = 8$ tokens per second and a draft has $n = 400$ tokens. By Equation (6.4), the generation alone takes $400/8 = 50$ seconds. Suppose a 1.7B model on a GPU produces $r = 60$ tokens per second and a label has $n = 3$ tokens: $3/60 = 0.05$ seconds. The ratio is a thousand to one. If the label had to wait in line behind the draft, the user posting a message would wait 50 seconds for a job that needs a twentieth of a second.

6.5.3 One long job at a time, and an honest queue

A CPU running a 7B model is fully busy with one generation. Starting a second one does not double the output; it roughly halves the speed of both, and memory pressure can make things worse. The quality lane therefore runs one long job at a time and queues the rest. If k jobs are ahead of you and each takes about T_{avg} seconds, your expected wait before starting is

$$W_k \approx k\, T_{\text{avg}}. \tag{6.6}$$

Profitina shows exactly this to users of *One Text, Many Platforms*: requests are processed one at a time across the whole system, and the page shows an honest queue position and a wait estimate. Long operations also show granular progress, with sub-step labels such as “drafting — 466 tokens” and a time-remaining estimate, so a user can see that work is happening instead of staring at a spinner [6]. Equation (6.6) is simple, but it turns an anxious wait into an informed one.

6.5.4 Work that belongs in neither lane

The most important lane rule is about where *not* to call a model at all. In September 2026 Profitina investigated why switching channels in the community feed felt very slow. Among three root causes, one was that rendering the feed made a synchronous local-model call for each suspicious link on every render. The fix removed every model call from the read path. Together with the other fixes, the feed for 30 posts went from 868 milliseconds to 11 milliseconds, and the AI calls per page went from 6 to 0, in the September 2026 release. The rule the team wrote down is worth memorizing:

> **WARNING:** A function called per item while rendering must never touch the model or open its own connection. Judge a link when it is posted, store the verdict, and let the read path read the stored verdict.

This is also why the rule floor exists under the fast lane. When a short job cannot be answered by the model in time, because the model is busy, missing, or slow, the deterministic rules answer instead, and the product keeps working. Chapter 5 described the rule floor for Link Safety; the same shape appears wherever the fast lane is used.

6.6 The Content Engine and the Quality Gate

6.6.1 What the Content Engine produces

The Content Engine is the "Optimize" step of Profitina's Scan · Optimize · Monitor loop (Chapter 3). After a scan has found where a business is hard for answer engines to quote, the Content Engine turns the gaps into publish-ready material: an answer-first paragraph that states who the business is and what it offers in the first sentence, the key entities, a set of frequently asked questions with direct answers, and schema.org JSON-LD that describes the business in machine-readable form. A Technical Assistant sits beside it with robots.txt help for AI crawlers and guidance on submitting a sitemap to Bing (Chapter 2 explains why Bing matters to ChatGPT and Copilot). The drafting runs on the quality lane.

6.6.2 The gate: serve AI only when it is better

A generated draft can be excellent, mediocre, or wrong in a way that is embarrassing to publish: it can invent a year of founding, promise "the best service in town", or forget the one fact the user gave it. Profitina's answer is a *quality gate*. For every request the system also has a built-in template, a plain, correct, slightly stiff text built directly from the facts. The AI draft is served only when it beats that template; otherwise the template is served. **Figure 6.3** shows the flow as this chapter's lab implements it.

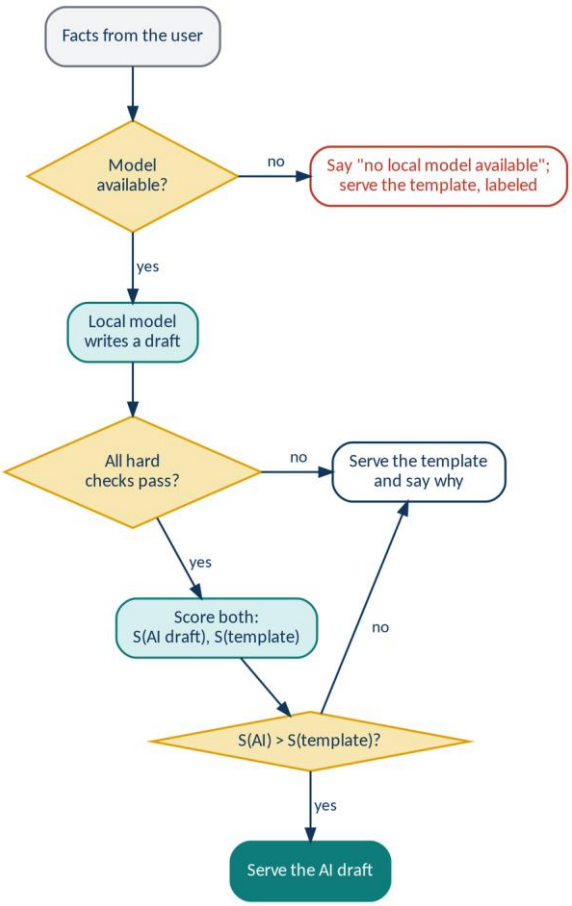


Figure 6.3 A quality gate as built in this chapter's lab. A draft must pass every hard check and then score strictly higher than the template; when no model is available the user is told so.

In symbols, write d for the AI draft and t for the template. Let $H(d)$ be true when the draft passes every *hard check* (the rules that must never be broken), and let $S(\cdot)$ be a *soft score* that measures how good a text is. Then

$$\text{serve}(d, t) = \begin{cases} d & \text{if } H(d) \text{ and } S(d) > S(t), \\ t & \text{otherwise.} \end{cases} \tag{6.7}$$

Three design choices hide in Equation (6.7). The hard checks come first, so no score, however high, can buy a draft past an invented fact. The comparison is strict, $S(d) > S(t)$, so a tie goes to the template: when the AI adds nothing, the predictable text wins. And the template is not a straw man. It is correct by construction, which makes it a meaningful baseline: an AI draft that cannot beat a correct, complete template has no business being served.

What goes into H and S is a product decision. The lab in Section 6.8 uses its own checklist, chosen for Profitina Laundry: the draft must name the business in its first sentence, must not use superlatives such as "best" or "terbaik", must not contain any number that is not in the facts, and must be of reasonable length; the soft score counts how many facts it covers and whether it opens in answer-first form. Your own gate may check different things. The principle is what matters.

6.6.3 Reading the public result: 5 of 6

Profitina publishes one number about its gate: after the September 2026 revision of the Content Engine, AI drafts beat the template in 5 of 6 benchmark cases [6]. The win rate is

$$\widehat{w} = \frac{5}{6} \approx 0.833. \tag{6.8}$$

Read Equation (6.8) carefully. It says the revised engine is usually better than the template on the cases tried, which is exactly what one wants to know before serving AI drafts at all. It does not say that 83.3% of all future drafts will be served, and it does not say that the sixth case was a disaster; it says that in that case the template was at least as good. With only six cases, each case moves the rate by $1/6 \approx 16.7$ percentage points, so one different outcome would give 4 of 6 or 6 of 6. The number is honest precisely because it is small and specific; the gate, which runs on every request, is what protects each user.

6.7 Honest "No Model Available" Behavior

A local model can be missing. It may not be downloaded yet, the model server may be restarting, the machine may be short of memory, or a timeout may pass before the first token arrives. What should the product do?

The tempting answer is to serve the template quietly, so the user never notices. Profitina rejects that answer. When no model is available, the app *says so* instead of silently serving the template [6]. The user sees that the text in front of them is the template, not an AI draft, and can decide whether to wait and try again.

Why insist on something that looks, at first, like admitting weakness? Because a silent fallback is a small lie, and small lies in software grow. A user who believes they are reading an AI draft will judge the AI by the template's stiffness, will not report the outage, and will not know to retry. An operator who never sees the fallback in the logs will not notice that the model has been down for a day. Profitina learned the same lesson elsewhere: a `try`/`except` around a failed database count once would have shown a convincing "0" on a dashboard; the rule since then is that errors must be loud. The Profitina group's sibling products share the stance, summed up in the motto "never stop silently".

The design that makes honesty easy has three parts, and the lab builds all three.

1. **A distinct, typed failure.** The client raises a specific exception, `NoModelAvailable`, for every way a model can be absent: the server cannot be reached, the model is not installed, or no answer arrived in time. Callers cannot confuse it with an ordinary result.
2. **A reason that a person can read.** The exception carries a short explanation such as "cannot reach http://127.0.0.1:11434 (connection refused)" or "HTTP 404: model 'qwen2.5:7b' not found", which can go to the log and, in plainer words, to the user.
3. **A labeled fallback.** Where a fallback exists, such as the template or the rule floor, the output says which one answered. Where none exists, the product says that the feature is unavailable right now.

> **KEY INSIGHT:** Degrading gracefully and degrading silently are different things. Graceful degradation keeps the product useful when a component fails; honesty tells the user which version of the product they are getting. A dependable system needs both.

6.8 Lab G5: Talking to a Local Model over HTTP

6.8.1 What we will build, and where the output comes from

This lab builds five small programs, each in Python and in Java, using only the standard library of each language:

1. `stub_ollama.py` / `StubOllama.java`: a tiny server that imitates the three Ollama endpoints of Section 6.4.1, so that everything runs on any computer, even one with no model installed;
2. `ollama_client.py` / `OllamaClient.java`: a client for Ollama's HTTP interface that works unchanged against the stub and against a real Ollama, and raises `NoModelAvailable` honestly;
3. `two_lanes.py` / `TwoLanes.java`: a router with a quality lane, a fast lane and a rule floor;
4. `quality_gate.py` / `QualityGate.java`: a quality gate that serves an AI draft only when it beats a template;
5. `json_label.py` / `JsonLabel.java`: structured output with validation and one retry (the solution to Exercise 6.6).

The Java programs also need `MiniJson.java`, a small JSON reader and writer, because the JDK has no JSON library of its own. The code is in the `Chapter06` folder of the book's code collection, with a README that gives the commands for Windows, macOS and Linux, in a terminal and in VS Code.

A plain statement about the output printed in this section: every output shown here came from the stub server, not from a real language model. We tried to install the Ollama program from its GitHub releases page and to pull a tiny Qwen model on the machine used to prepare this chapter, but that machine's network policy refused the downloads (github.com, ollama.com, the Ollama model registry and Hugging Face all answered with HTTP 403 from the proxy). Rather than print invented model output, we show exactly what the stub produced, and Section 6.8.8 explains what changes when you point the same programs at a real Ollama on your own computer. That is, fittingly, the honest "no model available" behavior of Section 6.7 applied to the book itself.

The servers in this lab listen on port 7060 of `127.0.0.1`, so they never collide with a real Ollama on port 11434. Stop each server with Ctrl+C when you have finished with it.

6.8.2 A stub that imitates `/api/generate`

A *stub* is a stand-in for a real component that behaves the same way at its interface and does something much simpler inside. Our stub answers `GET /api/version`, `GET /api/tags` and `POST /api/generate` with the same JSON field names as Ollama [11], and it offers two "models", `stub-quality` and `stub-fast`. They are not intelligent at all. If the prompt starts with `CLASSIFY:`, the stub returns one label, chosen by a few keyword rules. Otherwise it looks for a line `Business: …` and for lines that start with `-`, and it rearranges those facts into a paragraph. The `seed` option chooses among three behaviors on purpose: seed 0 gives a well-behaved draft, seed 1 a boastful draft that invents a founding year, and seed 2 a draft that says too little. The gate of Section 6.8.7 needs all three to show what it does.

Listing 6.1 is the Python version, built on `http.server`.

Listing 6.1 `stub_ollama.py`: a stand-in for Ollama's `/api/version`, `/api/tags` and `/api/generate`, with streaming as newline-delimited JSON.

```
"""A tiny stand-in for Ollama's HTTP API, so the lab runs without a model.

It answers GET /api/version, GET /api/tags and POST /api/generate with the
same JSON shapes as Ollama. The "models" are not intelligent: they only
rearrange the facts found in the prompt, deterministically.

    python stub_ollama.py [port] [--no-models] [--sleep-ms N]
"""
import json
import re
import sys
import time
from http.server import BaseHTTPRequestHandler, ThreadingHTTPServer

MODELS = {  # name -> simulated nanoseconds per generated token
    "stub-quality": 50_000_000,
    "stub-fast": 8_000_000,
}
NO_MODELS = "--no-models" in sys.argv
SLEEP_S = 0.005  # real pause per streamed token, so streaming is visible
if "--sleep-ms" in sys.argv:
    SLEEP_S = int(sys.argv[sys.argv.index("--sleep-ms") + 1]) / 1000


def classify(text):
    """Fast-lane job: one label for a short customer message."""
    t = text.lower()
    if re.search(r"\b(late|lost|broken|torn|refund|rusak|hilang)\b", t):
        return "complaint"
    if re.search(r"\b(thank|thanks|great|mantap)\b", t):
        return "praise"
    if "?" in t:
        return "question"
    return "other"
```

Listing 6.1 (Continued) *stub_ollama.py: a stand-in for Ollama's /api/version, /api/tags and /api/generate, with streaming as newline-delimited JSON.*

```
def draft(prompt, seed):
    """Quality-lane job: an answer-first paragraph built from '- ' facts."""
    name = re.search(r"^Business: (.+)$", prompt, re.M)
    name = name.group(1).strip() if name else "The business"
    facts = [f.strip().rstrip(".") for f in re.findall(r"^- (.+)$", prompt, re.M)]
    if not facts:
        return name + " did not give me any facts to write about."
    variant = seed % 3
    if variant == 0:  # a well-behaved draft
        rest = " ".join(f[0].upper() + f[1:] + "." for f in facts[1:])
        return f"{name} is {facts[0]}. {rest}".strip()
    if variant == 1:  # a boastful draft that invents a fact
        return (f"{name} is the best laundry in town, trusted since 1990. "
                f"It is {facts[0]}.")
    return f"{name}: {facts[0]}."  # a draft that says too little


def answer(model, prompt, options, fmt):
    if prompt.startswith("CLASSIFY:"):
        label = classify(prompt[len("CLASSIFY:"):])
        return json.dumps({"label": label}) if fmt == "json" else label
    return draft(prompt, int(options.get("seed", 0)))


class Handler(BaseHTTPRequestHandler):
    protocol_version = "HTTP/1.1"

    def log_message(self, *args):  # keep the terminal quiet
        pass

    def send_json(self, status, obj):
        body = json.dumps(obj).encode("utf-8")
        self.send_response(status)
        self.send_header("Content-Type", "application/json; charset=utf-8")
        self.send_header("Content-Length", str(len(body)))
        self.end_headers()
        self.wfile.write(body)
```

Listing 6.1 (Continued) *stub_ollama.py: a stand-in for Ollama's /api/version, /api/tags and /api/generate, with streaming as newline-delimited JSON.*

```
    def do_GET(self):
        if self.path == "/api/version":
            self.send_json(200, {"version": "0.0.0-stub"})
        elif self.path == "/api/tags":
            names = [] if NO_MODELS else sorted(MODELS)
            self.send_json(200, {"models": [{"name": n, "model": n}
                                            for n in names]})
        else:
            self.send_json(404, {"error": "not found"})

    def do_POST(self):
        if self.path != "/api/generate":
            self.send_json(404, {"error": "not found"})
            return
        length = int(self.headers.get("Content-Length", "0"))
        try:
            req = json.loads(self.rfile.read(length) or b"{}")
        except json.JSONDecodeError:
            self.send_json(400, {"error": "invalid JSON"})
            return
        model = req.get("model", "")
        if NO_MODELS or model not in MODELS:
            self.send_json(404, {"error": f"model '{model}' not found"})
            return
        text = answer(model, req.get("prompt", ""), req.get("options") or {},
                      req.get("format"))
        tokens = re.findall(r"\S+\s*", text)
        stats = {
            "done": True, "done_reason": "stop",
            "total_duration": 20_000_000 + len(tokens) * MODELS[model],
            "load_duration": 20_000_000,
            "prompt_eval_count": len(req.get("prompt", "").split()),
            "eval_count": len(tokens),
            "eval_duration": len(tokens) * MODELS[model],
        }
        head = {"model": model, "created_at": "2026-09-24T00:00:00Z"}
        if req.get("stream", True) is False:
            self.send_json(200, {**head, "response": text, **stats})
            return
        self.send_response(200)  # newline-delimited JSON, chunked
        self.send_header("Content-Type", "application/x-ndjson")
        self.send_header("Transfer-Encoding", "chunked")
        self.end_headers()
        for tok in tokens:
            self.write_chunk({**head, "response": tok, "done": False})
            time.sleep(SLEEP_S)
```

Listing 6.1 (Continued) *stub_ollama.py: a stand-in for Ollama's /api/version, /api/tags and /api/generate, with streaming as newline-delimited JSON.*

```
        self.write_chunk({**head, "response": "", **stats})
        self.wfile.write(b"0\r\n\r\n")

    def write_chunk(self, obj):
        line = (json.dumps(obj) + "\n").encode("utf-8")
        self.wfile.write(b"%x\r\n%s\r\n" % (len(line), line))
        self.wfile.flush()


def main():
    port = int(sys.argv[1]) if len(sys.argv) > 1 and sys.argv[1].isdigit() \
        else 7060
    server = ThreadingHTTPServer(("127.0.0.1", port), Handler)
    state = ("with no models" if NO_MODELS
             else "models: " + ", ".join(sorted(MODELS)))
    print(f"stub Ollama on http://127.0.0.1:{port} ({state})", flush=True)
    try:
        server.serve_forever()
    except KeyboardInterrupt:
        pass


if __name__ == "__main__":
    main()
```

Read it from the bottom up. `main` binds a `ThreadingHTTPServer` to `127.0.0.1`, never to all network interfaces: a development stub has no business being reachable from other machines. The optional port and the flags `--no-models` and `--sleep-ms` are read from the command line. `do_GET` answers the two small endpoints. `do_POST` reads exactly `Content-Length` bytes of the request body, parses the JSON, and returns `404` with Ollama's error shape, `{"error": "model '…' not found"}`, when the model is unknown or the server was started with `--no-models`. Then `answer` produces the text, and the text is cut into word-sized "tokens" with a regular expression. The statistics block imitates Ollama's fields; its durations are *simulated*, a fixed 50 milliseconds per token for `stub-quality` and 8 milliseconds for `stub-fast`, so that the rates printed later are repeatable. When the request says `"stream": false`, one JSON object is returned. Otherwise the stub sends one JSON object per line, using HTTP *chunked* transfer encoding: each chunk is its length in hexadecimal, a line break, the bytes, and another line break, and a chunk of length zero ends the response. That is exactly how a real Ollama streams.

Listing 6.2 is the same server in Java, using the JDK's built-in `com.sun.net.httpserver.HttpServer`.

Listing 6.2 `StubOllama.java`: the same stub with `com.sun.net.httpserver`; `sendResponseHeaders(200, 0)` selects chunked streaming.

```
import com.sun.net.httpserver.HttpExchange;
import com.sun.net.httpserver.HttpServer;
import java.io.IOException;
import java.io.OutputStream;
import java.net.InetSocketAddress;
import java.nio.charset.StandardCharsets;
import java.util.ArrayList;
import java.util.Arrays;
import java.util.LinkedHashMap;
import java.util.List;
import java.util.Map;
import java.util.TreeMap;
import java.util.concurrent.Executors;
import java.util.regex.Matcher;
import java.util.regex.Pattern;

/** A tiny stand-in for Ollama's HTTP API, so the lab runs without a model.
 *  java StubOllama [port] [--no-models] [--sleep-ms N] */
public class StubOllama {
    static final Map<String, Long> MODELS = new TreeMap<>(Map.of(
            "stub-quality", 50_000_000L,           // simulated ns per token
            "stub-fast", 8_000_000L));
    static boolean noModels = false;
    static long sleepMs = 5;                       // real pause per streamed token

    static String classify(String text) {
        String t = text.toLowerCase();
        if (Pattern.compile("\\b(late|lost|broken|torn|refund|rusak|hilang)\\b")
                .matcher(t).find()) return "complaint";
        if (Pattern.compile("\\b(thank|thanks|great|mantap)\\b")
                .matcher(t).find()) return "praise";
        return t.contains("?") ? "question" : "other";
    }

    static String draft(String prompt, long seed) {
        Matcher mn = Pattern.compile("(?m)^Business: (.+)$").matcher(prompt);
        String name = mn.find() ? mn.group(1).strip() : "The business";
        List<String> facts = new ArrayList<>();
        Matcher mf = Pattern.compile("(?m)^- (.+)$").matcher(prompt);
        while (mf.find()) facts.add(mf.group(1).strip().replaceAll("\\.+$", ""));
        if (facts.isEmpty()) {
            return name + " did not give me any facts to write about.";
        }
        long variant = seed % 3;
        if (variant == 0) {                        // a well-behaved draft
            StringBuilder rest = new StringBuilder();
```

Listing 6.2 (Continued) *StubOllama.java: the same stub with com.sun.net.httpserver; sendResponseHeaders(200, 0) selects chunked streaming.*

```
            for (String f : facts.subList(1, facts.size())) {
                rest.append(Character.toUpperCase(f.charAt(0)))
                    .append(f.substring(1)).append(". ");
            }
            return (name + " is " + facts.get(0) + ". " + rest).strip();
        }
        if (variant == 1) {                          // boastful, invents a fact
            return name + " is the best laundry in town, trusted since 1990. "
                    + "It is " + facts.get(0) + ".";
        }
        return name + ": " + facts.get(0) + ".";   // says too little
    }

    static String answer(String prompt, Map<String, Object> options, Object fmt) {
        if (prompt.startsWith("CLASSIFY:")) {
            String label = classify(prompt.substring("CLASSIFY:".length()));
            return "json".equals(fmt) ? MiniJson.write(Map.of("label", label))
                                      : label;
        }
        Object seed = options.getOrDefault("seed", 0L);
        return draft(prompt, ((Number) seed).longValue());
    }

    static void sendJson(HttpExchange ex, int status, Object obj)
            throws IOException {
        byte[] body = MiniJson.write(obj).getBytes(StandardCharsets.UTF_8);
        ex.getResponseHeaders().set("Content-Type",
                                    "application/json; charset=utf-8");
        ex.sendResponseHeaders(status, body.length);
        try (OutputStream out = ex.getResponseBody()) { out.write(body); }
    }

    static void handle(HttpExchange ex) throws IOException {
        String path = ex.getRequestURI().getPath();
        String method = ex.getRequestMethod();
        if (method.equals("GET") && path.equals("/api/version")) {
            sendJson(ex, 200, Map.of("version", "0.0.0-stub"));
        } else if (method.equals("GET") && path.equals("/api/tags")) {
            List<Object> list = new ArrayList<>();
            if (!noModels) {
                for (String n : MODELS.keySet()) {
                    list.add(MiniJson.obj("name", n, "model", n));
                }
            }
            sendJson(ex, 200, Map.of("models", list));
        } else if (method.equals("POST") && path.equals("/api/generate")) {
```

Listing 6.2 (Continued) *StubOllama.java: the same stub with com.sun.net.httpserver; sendResponseHeaders(200, 0) selects chunked streaming.*

```
            generate(ex);
        } else {
            sendJson(ex, 404, Map.of("error", "not found"));
        }
    }

    @SuppressWarnings("unchecked")
    static void generate(HttpExchange ex) throws IOException {
        Map<String, Object> req;
        try {
            String raw = new String(ex.getRequestBody().readAllBytes(),
                                    StandardCharsets.UTF_8);
            req = MiniJson.parseObject(raw.isBlank() ? "{}" : raw);
        } catch (RuntimeException e) {
            sendJson(ex, 400, Map.of("error", "invalid JSON"));
            return;
        }
        String model = String.valueOf(req.getOrDefault("model", ""));
        if (noModels || !MODELS.containsKey(model)) {
            sendJson(ex, 404, Map.of("error", "model '" + model + "' not found"));
            return;
        }
        String prompt = String.valueOf(req.getOrDefault("prompt", ""));
        Object opts = req.get("options");
        Map<String, Object> options =
                opts instanceof Map ? (Map<String, Object>) opts : Map.of();
        String text = answer(prompt, options, req.get("format"));
        List<String> tokens = new ArrayList<>();
        Matcher m = Pattern.compile("\\S+\\s*").matcher(text);
        while (m.find()) tokens.add(m.group());
        long perToken = MODELS.get(model), n = tokens.size();
        Map<String, Object> head = MiniJson.obj("model", model,
                "created_at", "2026-09-24T00:00:00Z");
        Map<String, Object> stats = MiniJson.obj(
                "done", true, "done_reason", "stop",
                "total_duration", 20_000_000L + n * perToken,
                "load_duration", 20_000_000L,
                "prompt_eval_count", (long) prompt.split("\\s+").length,
                "eval_count", n, "eval_duration", n * perToken);
        if (Boolean.FALSE.equals(req.get("stream"))) {
            Map<String, Object> all = new LinkedHashMap<>(head);
            all.put("response", text);
            all.putAll(stats);
            sendJson(ex, 200, all);
            return;
        }
```

Listing 6.2 (Continued) *StubOllama.java: the same stub with com.sun.net.httpserver; sendResponseHeaders(200, 0) selects chunked streaming.*

```
        ex.getResponseHeaders().set("Content-Type", "application/x-ndjson");
        ex.sendResponseHeaders(200, 0);            // 0 = chunked, length unknown
        try (OutputStream out = ex.getResponseBody()) {
            for (String tok : tokens) {
                Map<String, Object> chunk = new LinkedHashMap<>(head);
                chunk.put("response", tok);
                chunk.put("done", false);
                writeLine(out, chunk);
                try {
                    Thread.sleep(sleepMs);
                } catch (InterruptedException e) {
                    return;
                }
            }
            Map<String, Object> last = new LinkedHashMap<>(head);
            last.put("response", "");
            last.putAll(stats);
            writeLine(out, last);
        }
    }

    static void writeLine(OutputStream out, Object obj) throws IOException {
        out.write((MiniJson.write(obj) + "\n").getBytes(StandardCharsets.UTF_8));
        out.flush();
    }

    public static void main(String[] args) throws IOException {
        List<String> a = Arrays.asList(args);
        int port = !a.isEmpty() && a.get(0).matches("\\d+")
                ? Integer.parseInt(a.get(0)) : 7060;
        noModels = a.contains("--no-models");
        if (a.contains("--sleep-ms")) {
            sleepMs = Long.parseLong(a.get(a.indexOf("--sleep-ms") + 1));
        }
        HttpServer server =
                HttpServer.create(new InetSocketAddress("127.0.0.1", port), 0);
        server.createContext("/", ex -> {
            try { handle(ex); } finally { ex.close(); }
        });
        server.setExecutor(Executors.newCachedThreadPool());
        server.start();
        String state = noModels ? "with no models"
                                : "models: " + String.join(", ", MODELS.keySet());
        System.out.println("stub Ollama on http://127.0.0.1:" + port
                           + " (" + state + ")");
    }
```

Listing 6.2 (Continued) *StubOllama.java: the same stub with com.sun.net.httpserver; sendResponseHeaders(200, 0) selects chunked streaming.*

```
}
```

The structure matches the Python version line for line. Two Java details deserve a comment. `ex.sendResponseHeaders(200, 0)` means "length unknown", which makes the server use chunked encoding for the streamed answer, while a positive length sends an ordinary response. And a `TreeMap` keeps the model names sorted, so the two stubs list them in the same order.

Both Java programs read and write JSON with **Listing 6.3**, a deliberately small parser and writer. It handles objects, arrays, strings with escapes, numbers, `true`, `false` and `null`, which is everything Ollama sends in the JSON format defined by RFC 8259 [18], and its writer uses the same spacing as Python's `json.dumps`, so both stubs produce byte-identical responses.

Listing 6.3 `MiniJson.java`: a minimal recursive-descent JSON reader and a writer, enough for this lab.

```
import java.util.ArrayList;
import java.util.LinkedHashMap;
import java.util.List;
import java.util.Map;

/** Just enough JSON for this lab: objects, arrays, strings, numbers,
 *  true, false and null. Parsed objects become LinkedHashMap. */
public final class MiniJson {
    private final String s;
    private int i;

    private MiniJson(String s) { this.s = s; }

    public static Object parse(String text) {
        MiniJson p = new MiniJson(text);
        Object v = p.value();
        p.ws();
        if (p.i != p.s.length()) throw p.err("trailing characters");
        return v;
    }

    @SuppressWarnings("unchecked")
    public static Map<String, Object> parseObject(String text) {
        Object v = parse(text);
        if (!(v instanceof Map)) {
            throw new IllegalArgumentException("not an object");
        }
        return (Map<String, Object>) v;
    }

    private IllegalArgumentException err(String what) {
        return new IllegalArgumentException("bad JSON at " + i + ": " + what);
    }

    private void ws() {
        while (i < s.length() && Character.isWhitespace(s.charAt(i))) i++;
    }
```

Listing 6.3 (Continued) *MiniJson.java: a minimal recursive-descent JSON reader and a writer, enough for this lab.*

```
    private Object value() {
        ws();
        if (i >= s.length()) throw err("unexpected end");
        char c = s.charAt(i);
        if (c == '{') return object();
        if (c == '[') return array();
        if (c == '"') return string();
        if (s.startsWith("true", i)) { i += 4; return Boolean.TRUE; }
        if (s.startsWith("false", i)) { i += 5; return Boolean.FALSE; }
        if (s.startsWith("null", i)) { i += 4; return null; }
        return number();
    }

    private Map<String, Object> object() {
        Map<String, Object> m = new LinkedHashMap<>();
        i++;                                          // skip {
        ws();
        if (s.charAt(i) == '}') { i++; return m; }
        while (true) {
            ws();
            String key = string();
            ws();
            if (s.charAt(i++) != ':') throw err("':' expected");
            m.put(key, value());
            ws();
            char c = s.charAt(i++);
            if (c == '}') return m;
            if (c != ',') throw err("',' or '}' expected");
        }
    }

    private List<Object> array() {
        List<Object> a = new ArrayList<>();
        i++;                                          // skip [
        ws();
        if (s.charAt(i) == ']') { i++; return a; }
        while (true) {
            a.add(value());
            ws();
            char c = s.charAt(i++);
            if (c == ']') return a;
            if (c != ',') throw err("',' or ']' expected");
        }
    }
```

Listing 6.3 (Continued) *MiniJson.java: a minimal recursive-descent JSON reader and a writer, enough for this lab.*

```
    private String string() {
        if (s.charAt(i) != '"') throw err("string expected");
        StringBuilder b = new StringBuilder();
        i++;
        while (true) {
            char c = s.charAt(i++);
            if (c == '"') return b.toString();
            if (c != '\\') { b.append(c); continue; }
            char e = s.charAt(i++);
            switch (e) {
                case 'n' -> b.append('\n');
                case 't' -> b.append('\t');
                case 'r' -> b.append('\r');
                case 'b' -> b.append('\b');
                case 'f' -> b.append('\f');
                case 'u' -> {
                    b.append((char) Integer.parseInt(s.substring(i, i + 4), 16));
                    i += 4;
                }
                default -> b.append(e);            // \" \\ \/
            }
        }
    }

    private Object number() {
        int start = i;
        while (i < s.length() && "+-0123456789.eE".indexOf(s.charAt(i)) >= 0) i++;
        String t = s.substring(start, i);
        if (t.isEmpty()) throw err("value expected");
        if (t.contains(".") || t.contains("e") || t.contains("E")) {
            return Double.parseDouble(t);
        }
        return Long.parseLong(t);
    }

    /** Write a value as JSON, with the same spacing as Python's json.dumps. */
    public static String write(Object v) {
        StringBuilder b = new StringBuilder();
        write(v, b);
        return b.toString();
    }
```

Listing 6.3 (Continued) *MiniJson.java: a minimal recursive-descent JSON reader and a writer, enough for this lab.*

```
    private static void write(Object v, StringBuilder b) {
        if (v == null) b.append("null");
        else if (v instanceof String str) quote(str, b);
        else if (v instanceof Map<?, ?> m) {
            b.append('{');
            boolean first = true;
            for (Map.Entry<?, ?> e : m.entrySet()) {
                if (!first) b.append(", ");
                first = false;
                quote(String.valueOf(e.getKey()), b);
                b.append(": ");
                write(e.getValue(), b);
            }
            b.append('}');
        } else if (v instanceof List<?> list) {
            b.append('[');
            for (int k = 0; k < list.size(); k++) {
                if (k > 0) b.append(", ");
                write(list.get(k), b);
            }
            b.append(']');
        } else b.append(v);                          // numbers and booleans
    }

    private static void quote(String str, StringBuilder b) {
        b.append('"');
        for (char c : str.toCharArray()) {
            switch (c) {
                case '"' -> b.append("\\\"");
                case '\\' -> b.append("\\\\");
                case '\n' -> b.append("\\n");
                case '\r' -> b.append("\\r");
                case '\t' -> b.append("\\t");
                default -> {
                    if (c >= 0x20 && c <= 0x7e) b.append(c);
                    else b.append(String.format("\\u%04x", (int) c));
                }
            }
        }
        b.append('"');
    }
```

Listing 6.3 (Continued) *MiniJson.java: a minimal recursive-descent JSON reader and a writer, enough for this lab.*

```
    /** Small helper for building objects: obj("a", 1, "b", "x"). */
    public static Map<String, Object> obj(Object... kv) {
        Map<String, Object> m = new LinkedHashMap<>();
        for (int k = 0; k < kv.length; k += 2) m.put((String) kv[k], kv[k + 1]);
        return m;
    }
}
```

`MiniJson` is a *recursive-descent* parser: `value` looks at the next character and hands over to `object`, `array`, `string` or `number`, and `object` and `array` call `value` again for each element, so nested JSON is handled by nested calls. In a larger project you would use a library such as Jackson or Gson; here, avoiding dependencies keeps the lab runnable with nothing but a JDK.

6.8.3 Starting the stub

Open a terminal in `Chapter06/python` and start the Python stub, or in `Chapter06/java` compile and start the Java one. Use one or the other: both listen on port 7060.

```
python stub_ollama.py
javac *.java
java StubOllama
```

Either prints one line and then waits for requests:

```
stub Ollama on http://127.0.0.1:7060 (models: stub-fast, stub-quality)
```

On macOS and Linux the Python command may be `python3`. Leave this terminal open and use a second terminal for the programs that follow.

6.8.4 Looking at the wire

Before writing a client, it pays to look at the raw exchange. With the stub running, send one streaming request with `curl` and pretty-print each line of the answer with Python's `json.tool` module (on Windows, type `curl.exe` in PowerShell, because `curl` there is an alias for a different command, and put the JSON in a file with `-d @request.json` if the quoting gives trouble):

```
curl -s http://127.0.0.1:7060/api/generate -d '{"model": "stub-fast", "prompt":
"CLASSIFY: thanks, great ironing"}' | python -m json.tool --json-lines
```

```
{
    "model": "stub-fast",
    "created_at": "2026-09-24T00:00:00Z",
    "response": "praise",
    "done": false
}
{
    "model": "stub-fast",
    "created_at": "2026-09-24T00:00:00Z",
    "response": "",
    "done": true,
    "done_reason": "stop",
    "total_duration": 28000000,
    "load_duration": 20000000,
    "prompt_eval_count": 4,
    "eval_count": 1,
    "eval_duration": 8000000
}
```

Two JSON objects arrived, one per line. The first carries the only token of the answer, `praise`, with `"done": false`. The second carries an empty `response`, `"done": true`, and the statistics. By Equation (6.5), the stub's simulated generation rate is $1/(8{,}000{,}000/10^9) = 125$ tokens per second. **Figure 6.4** draws the same exchange for a longer answer.

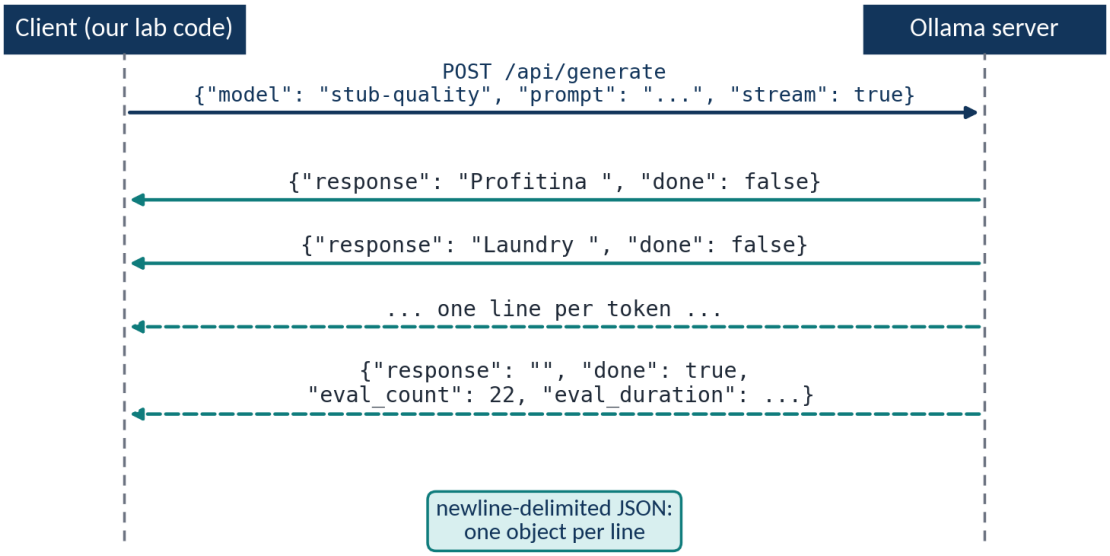


Figure 6.4 A streaming call to `/api/generate`. The client sends one JSON request; the server answers with one JSON object per line, the last one marked `"done": true` and carrying the token counts and durations.

For comparison, the Ollama documentation gives this example of a complete, non-streaming answer from a real server (it is quoted from the documentation [11], not produced by us):

```
{
  "model": "gemma4",
  "created_at": "2025-10-17T23:14:07.414671Z",
  "response": "Hello! How can I help you today?",
  "done": true,
  "done_reason": "stop",
  "total_duration": 174560334,
  "load_duration": 101397084,
  "prompt_eval_count": 11,
  "prompt_eval_duration": 13074791,
  "eval_count": 18,
  "eval_duration": 52479709
}
```

The field names and meanings are the ones our stub uses. Applying Equation (6.5) to the documentation example gives $18/0.0525 \approx 343$ tokens per second on whatever hardware produced it.

6.8.5 The client

Listing 6.4 is the client in Python.

Listing 6.4 `ollama_client.py`: a standard-library client for Ollama's HTTP interface, with one honest exception for every way a model can be missing.

```
"""A small client for Ollama's HTTP API, standard library only.

    python ollama_client.py [--host URL] [--model NAME] [--stream] PROMPT

Works the same against a real Ollama (default http://127.0.0.1:11434) and
against stub_ollama.py (http://127.0.0.1:7060).
"""
import json
import sys
import urllib.error
import urllib.request

DEFAULT_HOST = "http://127.0.0.1:11434"  # 127.0.0.1, not localhost


class NoModelAvailable(Exception):
    """No local model can answer. The caller must say so, not pretend."""


class OllamaClient:
    def __init__(self, host=DEFAULT_HOST, timeout=60.0):
        self.host = host.rstrip("/")
        self.timeout = timeout
        # Local traffic must never be sent through a proxy.
        self.opener = urllib.request.build_opener(
            urllib.request.ProxyHandler({}))

    def _open(self, path, payload=None):
        data = None if payload is None else json.dumps(payload).encode()
        req = urllib.request.Request(
            self.host + path, data=data,
            headers={"Content-Type": "application/json"})
        try:
            return self.opener.open(req, timeout=self.timeout)
        except urllib.error.HTTPError as e:
            detail = e.read().decode("utf-8", "replace")
            try:
                detail = json.loads(detail).get("error", detail)
            except json.JSONDecodeError:
                pass
            raise NoModelAvailable(f"HTTP {e.code}: {detail}") from None
        except OSError as e:  # URLError, refused, timed out, ...
            reason = getattr(e, "reason", e)
            if isinstance(reason, TimeoutError):  # socket.timeout too
                raise NoModelAvailable(
                    f"no answer within {self.timeout} s") from None
```

Listing 6.4 (Continued) *ollama_client.py: a standard-library client for Ollama's HTTP interface, with one honest exception for every way a model can be missing.*

```
            if isinstance(reason, ConnectionRefusedError):
                reason = "connection refused"
            raise NoModelAvailable(
                f"cannot reach {self.host} ({reason})") from None

    def models(self):
        with self._open("/api/tags") as resp:
            return [m["name"] for m in json.load(resp).get("models", [])]

    def generate(self, model, prompt, options=None, fmt=None):
        """One request, one complete answer (stream: false)."""
        payload = {"model": model, "prompt": prompt, "stream": False}
        if options:
            payload["options"] = options
        if fmt:
            payload["format"] = fmt
        try:
            with self._open("/api/generate", payload) as resp:
                return json.load(resp)
        except TimeoutError:  # the answer started but did not finish
            raise NoModelAvailable(
                f"no answer within {self.timeout} s") from None

    def stream(self, model, prompt, options=None):
        """Yield the partial answers as they arrive (stream: true)."""
        payload = {"model": model, "prompt": prompt, "stream": True}
        if options:
            payload["options"] = options
        with self._open("/api/generate", payload) as resp:
            for line in resp:  # one JSON object per line
                if line.strip():
                    yield json.loads(line)


def main(argv):
    host, model, stream, words = DEFAULT_HOST, "qwen2.5:7b", False, []
    i = 0
    while i < len(argv):
        if argv[i] == "--host":
            host, i = argv[i + 1], i + 2
        elif argv[i] == "--model":
            model, i = argv[i + 1], i + 2
        elif argv[i] == "--stream":
            stream, i = True, i + 1
        else:
            words, i = words + [argv[i]], i + 1
```

Listing 6.4 (Continued) *ollama_client.py: a standard-library client for Ollama's HTTP interface, with one honest exception for every way a model can be missing.*

```
    prompt = " ".join(words) or "Say hello in one short sentence."
    client = OllamaClient(host)
    try:
        print("models on server:", ", ".join(client.models()) or "(none)")
        if stream:
            for chunk in client.stream(model, prompt):
                print(chunk["response"], end="", flush=True)
                if chunk["done"]:
                    final = chunk
            print()
        else:
            final = client.generate(model, prompt)
            print(final["response"])
    except NoModelAvailable as e:
        print("No local model available:", e)
        return 2
    rate = final["eval_count"] / (final["eval_duration"] / 1e9)
    print(f"[{final['model']}: eval_count={final['eval_count']}, "
          f"{rate:.1f} tokens/s as reported by the server]")
    return 0


if __name__ == "__main__":
    sys.exit(main(sys.argv[1:]))
```

Three decisions shape this client.

- **No proxy for local traffic.** `urllib` normally honors the proxy settings of the environment. Many company and campus networks set a proxy, and sending a request for `127.0.0.1` through it fails in confusing ways. `ProxyHandler({})` builds an opener that never uses a proxy. The Java client makes the same choice with `HttpClient.Builder.NO_PROXY`.
- **One exception, many causes.** `_open` turns every failure into `NoModelAvailable` with a readable reason: an HTTP error such as 404 (the server's `error` field is extracted from the JSON body), a refused connection (nothing is listening), or a timeout (the server is listening but did not answer in time). Callers only need one `except` clause, and they cannot forget that the model may be missing, because the exception is the only way the client reports that.
- **Two ways to receive an answer.** `generate` asks for `"stream": false` and returns one dictionary. `stream` asks for streaming and is a *generator*: it `yield`s each JSON line as soon as it arrives, so the caller can print tokens while the model is still producing them, which is what makes a long draft feel alive.

Listing 6.5 is the Java version.

Listing 6.5 `OllamaClient.java`: the same client with `java.net.http.HttpClient`; streaming uses `BodyHandlers.ofLines()`.

```
import java.io.IOException;
import java.net.ConnectException;
import java.net.URI;
import java.net.http.HttpClient;
import java.net.http.HttpRequest;
import java.net.http.HttpResponse;
import java.net.http.HttpTimeoutException;
import java.time.Duration;
import java.util.ArrayList;
import java.util.LinkedHashMap;
import java.util.List;
import java.util.Locale;
import java.util.Map;
import java.util.function.Consumer;
import java.util.stream.Stream;

/** A small client for Ollama's HTTP API, JDK standard library only.
 *  java OllamaClient [--host URL] [--model NAME] [--stream] PROMPT */
public class OllamaClient {
    public static final String DEFAULT_HOST = "http://127.0.0.1:11434";

    /** No local model can answer. The caller must say so, not pretend. */
    public static class NoModelAvailable extends Exception {
        public NoModelAvailable(String message) { super(message); }
    }

    private final String host;
    private final double timeoutS;
    private final HttpClient http;

    public OllamaClient(String host, double timeoutS) {
        this.host = host.replaceAll("/+$", "");
        this.timeoutS = timeoutS;
        this.http = HttpClient.newBuilder()
                .proxy(HttpClient.Builder.NO_PROXY)  // local traffic, no proxy
                .connectTimeout(Duration.ofMillis((long) (timeoutS * 1000)))
                .build();
    }
```

Listing 6.5 (Continued) *OllamaClient.java: the same client with java.net.http.HttpClient; streaming uses BodyHandlers.ofLines().*

```
    private <T> HttpResponse<T> send(String path, Object payload,
                                     HttpResponse.BodyHandler<T> handler)
            throws NoModelAvailable {
        HttpRequest.Builder b = HttpRequest.newBuilder(URI.create(host + path))
                .timeout(Duration.ofMillis((long) (timeoutS * 1000)))
                .header("Content-Type", "application/json");
        if (payload == null) b.GET();
        else b.POST(HttpRequest.BodyPublishers.ofString(MiniJson.write(payload)));
        try {
            return http.send(b.build(), handler);
        } catch (HttpTimeoutException e) {
            throw new NoModelAvailable("no answer within " + timeoutS + " s");
        } catch (ConnectException e) {
            throw new NoModelAvailable("cannot reach " + host
                                       + " (connection refused)");
        } catch (IOException e) {
            throw new NoModelAvailable("cannot reach " + host + " (" + e + ")");
        } catch (InterruptedException e) {
            Thread.currentThread().interrupt();
            throw new NoModelAvailable("interrupted");
        }
    }

    private static void check(int status, String body) throws NoModelAvailable {
        if (status == 200) return;
        String detail = body;
        try {
            Object err = MiniJson.parseObject(body).get("error");
            if (err != null) detail = err.toString();
        } catch (RuntimeException ignored) {
            // the body was not JSON; keep it as it is
        }
        throw new NoModelAvailable("HTTP " + status + ": " + detail);
    }

    @SuppressWarnings("unchecked")
    public List<String> models() throws NoModelAvailable {
        HttpResponse<String> r = send("/api/tags", null,
                                      HttpResponse.BodyHandlers.ofString());
        check(r.statusCode(), r.body());
        List<String> names = new ArrayList<>();
        Object models = MiniJson.parseObject(r.body()).get("models");
        for (Object m : (List<Object>) models) {
            names.add((String) ((Map<String, Object>) m).get("name"));
        }
        return names;
```

Listing 6.5 (Continued) *OllamaClient.java: the same client with java.net.http.HttpClient; streaming uses BodyHandlers.ofLines().*

```
    }

    /** One request, one complete answer (stream: false). */
    public Map<String, Object> generate(String model, String prompt,
            Map<String, Object> options, String format) throws NoModelAvailable {
        Map<String, Object> payload = MiniJson.obj(
                "model", model, "prompt", prompt, "stream", false);
        if (options != null && !options.isEmpty()) payload.put("options", options);
        if (format != null) payload.put("format", format);
        HttpResponse<String> r = send("/api/generate", payload,
                                      HttpResponse.BodyHandlers.ofString());
        check(r.statusCode(), r.body());
        return MiniJson.parseObject(r.body());
    }

    /** Hand each partial answer to onChunk as it arrives (stream: true). */
    public void stream(String model, String prompt, Map<String, Object> options,
                       Consumer<Map<String, Object>> onChunk)
            throws NoModelAvailable {
        Map<String, Object> payload = MiniJson.obj(
                "model", model, "prompt", prompt, "stream", true);
        if (options != null && !options.isEmpty()) payload.put("options", options);
        HttpResponse<Stream<String>> r = send("/api/generate", payload,
                                              HttpResponse.BodyHandlers.ofLines());
        if (r.statusCode() != 200) {
            check(r.statusCode(), String.join("\n", r.body().toList()));
        }
        try (Stream<String> lines = r.body()) {    // one JSON object per line
            lines.filter(l -> !l.isBlank())
                 .forEach(l -> onChunk.accept(MiniJson.parseObject(l)));
        } catch (java.io.UncheckedIOException e) {
            throw new NoModelAvailable("stream broke off (" + e.getCause() + ")");
        }
    }

    public static void main(String[] args) {
        String host = DEFAULT_HOST, model = "qwen2.5:7b";
        boolean stream = false;
        List<String> words = new ArrayList<>();
        for (int i = 0; i < args.length; i++) {
            switch (args[i]) {
                case "--host" -> host = args[++i];
                case "--model" -> model = args[++i];
                case "--stream" -> stream = true;
                default -> words.add(args[i]);
            }
```

Listing 6.5 (Continued) *OllamaClient.java: the same client with java.net.http.HttpClient; streaming uses BodyHandlers.ofLines().*

```
        }
        String prompt = words.isEmpty() ? "Say hello in one short sentence."
                                        : String.join(" ", words);
        OllamaClient client = new OllamaClient(host, 60.0);
        Map<String, Object> fin = new LinkedHashMap<>();
        try {
            List<String> names = client.models();
            System.out.println("models on server: "
                    + (names.isEmpty() ? "(none)" : String.join(", ", names)));
            if (stream) {
                client.stream(model, prompt, null, chunk -> {
                    System.out.print(chunk.get("response"));
                    System.out.flush();
                    if (Boolean.TRUE.equals(chunk.get("done"))) fin.putAll(chunk);
                });
                System.out.println();
            } else {
                fin.putAll(client.generate(model, prompt, null, null));
                System.out.println(fin.get("response"));
            }
        } catch (NoModelAvailable e) {
            System.out.println("No local model available: " + e.getMessage());
            System.exit(2);
        }
        long n = ((Number) fin.get("eval_count")).longValue();
        double secs = ((Number) fin.get("eval_duration")).doubleValue() / 1e9;
        System.out.printf(Locale.ROOT, "[%s: eval_count=%d, %.1f tokens/s as"
                + " reported by the server]%n", fin.get("model"), n, n / secs);
    }
}
```

In Java, `NoModelAvailable` extends `Exception`, so it is a *checked* exception: the compiler refuses to compile a caller that neither catches it nor declares it. That is the language enforcing Section 6.7 for us. `HttpResponse.BodyHandlers.ofLines()` delivers the streamed body as a `Stream<String>` of lines, which the client parses one by one and hands to a `Consumer`, the Java counterpart of Python's generator. `Locale.ROOT` in the last `printf` keeps the decimal point a point on computers whose language setting would print `20,0`.

Now run the client against the stub. From `Chapter06/python` (or with `java OllamaClient` and the same arguments from `Chapter06/java`):

```
python ollama_client.py --host http://127.0.0.1:7060 --model stub-fast "CLASSIFY: my
shirt came back torn"
python ollama_client.py --host http://127.0.0.1:7060 --model stub-quality --stream "Tell
me about the laundry"
```

```
models on server: stub-fast, stub-quality
complaint
[stub-fast: eval_count=1, 125.0 tokens/s as reported by the server]
models on server: stub-fast, stub-quality
The business did not give me any facts to write about.
[stub-quality: eval_count=11, 20.0 tokens/s as reported by the server]
```

The fast "model" labeled the message a complaint; the quality "model", given no facts, said so rather than inventing some. The rates, 125 and 20 tokens per second, are the stub's simulated rates, computed with Equation (6.5) from the fields it returned.

Now the three ways to have no model. First, ask for a model the server does not have (the client's default is `qwen2.5:7b`); second, point the client at a port where nothing listens; third, restart the stub with `python stub_ollama.py --no-models` and ask again:

```
python ollama_client.py --host http://127.0.0.1:7060 "Say hello"
python ollama_client.py --host http://127.0.0.1:7069 "Say hello"
python ollama_client.py --host http://127.0.0.1:7060 --model stub-quality "Say hello"
```

```
models on server: stub-fast, stub-quality
No local model available: HTTP 404: model 'qwen2.5:7b' not found
No local model available: cannot reach http://127.0.0.1:7069 (connection refused)
models on server: (none)
No local model available: HTTP 404: model 'stub-quality' not found
```

Each failure produced a clear sentence and exit code 2, never an invented answer. The Java client prints the same lines; we compared the outputs of both languages with `diff`, against both stubs, and they are identical.

6.8.6 Two lanes in code

Listing 6.6 turns **Figure 6.2** into a program. A `Lane` records a name, a model, a timeout and a maximum number of tokens. The quality lane gets 120 seconds and up to 400 tokens; the fast lane gets 3 seconds and 8 tokens. These are this lab's own values, chosen for a laptop; tune them to your hardware.

Listing 6.6 `two_lanes.py`: a quality lane that runs one long job at a time and reports queue positions, a fast lane for labels, and a rule floor underneath.

```
"""Two lanes for local AI: a quality lane for long drafts, a fast lane for
short labels. Start stub_ollama.py with --sleep-ms 100 first.

    python two_lanes.py [--host URL]
"""
import re
import sys
import threading
from dataclasses import dataclass

from ollama_client import NoModelAvailable, OllamaClient


@dataclass
class Lane:
    name: str
    model: str
    timeout_s: float
    max_tokens: int


QUALITY = Lane("quality", "stub-quality", 120.0, 400)
FAST = Lane("fast", "stub-fast", 3.0, 8)


def rule_floor(message):
    """Deterministic labels that need no model at all."""
    t = message.lower()
    if re.search(r"\b(late|lost|broken|torn|refund)\b", t):
        return "complaint"
    return "question" if "?" in t else "other"


class Router:
    def __init__(self, host):
        self.host = host
        self.quality_lock = threading.Lock()  # one long job at a time
        self.waiting = 0
        self.log_lock = threading.Lock()

    def log(self, text):
        with self.log_lock:
            print(text, flush=True)
```

Listing 6.6 (Continued) *two_lanes.py: a quality lane that runs one long job at a time and reports queue positions, a fast lane for labels, and a rule floor underneath.*

```
    def classify(self, message, lane=FAST):
        client = OllamaClient(self.host, lane.timeout_s)
        try:
            out = client.generate(lane.model, "CLASSIFY: " + message,
                                  {"num_predict": lane.max_tokens})
            return out["response"].strip(), f"{lane.name} lane", ""
        except NoModelAvailable as e:
            return rule_floor(message), "rule floor", str(e)

    def draft(self, job, prompt):
        with self.log_lock:
            self.waiting += 1
            position = self.waiting - 1
        if position:
            self.log(f"  {job}: queued, position {position} in quality lane")
        with self.quality_lock:
            self.log(f"  {job}: drafting on {QUALITY.model} ...")
            client = OllamaClient(self.host, QUALITY.timeout_s)
            try:
                text = "".join(c["response"] for c in client.stream(
                    QUALITY.model, prompt, {"num_predict": QUALITY.max_tokens}))
                self.log(f"  {job}: done, {len(text.split())} words: "
                         f"{text[:34]}...")
            except NoModelAvailable as e:
                self.log(f"  {job}: no local model available ({e})")
            finally:
                with self.log_lock:
                    self.waiting -= 1


PROMPT = """Business: Profitina Laundry
- a laundry service in Surabaya
- open every day from 07:00 to 21:00
- express service is ready in 6 hours"""

MESSAGES = ["Is express really ready in 6 hours?",
            "My order came back late and one shirt was torn",
            "Thanks, the ironing was great"]
```

Listing 6.6 (Continued) *two_lanes.py: a quality lane that runs one long job at a time and reports queue positions, a fast lane for labels, and a rule floor underneath.*

```
def main(argv):
    host = argv[argv.index("--host") + 1] if "--host" in argv \
        else "http://127.0.0.1:7060"
    router = Router(host)
    print("1. Two long drafts go to the quality lane:")
    jobs = [threading.Thread(target=router.draft, args=(j, PROMPT))
            for j in ("draft A", "draft B")]
    for t in jobs:
        t.start()
        t.join(0.2)  # let A take the lane before B arrives
    print("2. Meanwhile, short labels go to the fast lane:")
    for m in MESSAGES:
        label, source, _ = router.classify(m)
        router.log(f"  {label:<9} <- {m!r} [{source}]")
    for t in jobs:
        t.join()
    print("3. The fast lane loses its model; the rule floor answers:")
    gone = Lane("fast", "stub-missing", 3.0, 8)
    for m in MESSAGES[:2]:
        label, source, why = router.classify(m, gone)
        print(f"  {label:<9} <- {m!r} [{source}]")
    print(f"  (the fast lane said: {why})")


if __name__ == "__main__":
    main(sys.argv[1:])
```

The `Router` enforces the lane rules. `draft` counts waiting jobs to report an honest queue position, which is Equation (6.6) with a real k, and then takes `quality_lock`, a lock that lets only one long job use the quality lane at a time. `classify` never waits for that lock: it goes straight to the fast lane with its own short timeout, and if the fast lane raises `NoModelAvailable`, the deterministic `rule_floor` answers and the result says so. `log` prints under a second lock so that lines from different threads never interleave.

`main` stages a small play in three acts. Two drafts are started on background threads, 0.2 seconds apart, so draft A takes the lane and draft B has to queue. While the drafts are being written, three customer messages are labeled on the main thread. Finally the fast lane is pointed at a model that does not exist, to show the rule floor.

Listing 6.7 is the same router in Java. A `record` replaces the Python dataclass, a fair `ReentrantLock` guards the quality lane, and `synchronized` methods protect the counter and the printing.

Listing 6.7 `TwoLanes.java`: the two-lane router in Java, with `record`, `ReentrantLock` and threads.

```
import java.util.List;
import java.util.Map;
import java.util.concurrent.locks.ReentrantLock;
import java.util.regex.Pattern;

/** Two lanes for local AI: a quality lane for long drafts, a fast lane for
 *  short labels. Start StubOllama with --sleep-ms 100 first.
 *  java TwoLanes [--host URL] */
public class TwoLanes {
    record Lane(String name, String model, double timeoutS, int maxTokens) {}

    static final Lane QUALITY = new Lane("quality", "stub-quality", 120.0, 400);
    static final Lane FAST = new Lane("fast", "stub-fast", 3.0, 8);

    record Label(String label, String source, String why) {}

    /** Deterministic labels that need no model at all. */
    static String ruleFloor(String message) {
        String t = message.toLowerCase();
        if (Pattern.compile("\\b(late|lost|broken|torn|refund)\\b")
                .matcher(t).find()) return "complaint";
        return t.contains("?") ? "question" : "other";
    }

    private final String host;
    private final ReentrantLock qualityLock = new ReentrantLock(true);
    private int waiting = 0;

    TwoLanes(String host) { this.host = host; }

    synchronized void log(String text) { System.out.println(text); }

    Label classify(String message, Lane lane) {
        OllamaClient client = new OllamaClient(host, lane.timeoutS());
        try {
            Map<String, Object> out = client.generate(lane.model(),
                    "CLASSIFY: " + message, Map.of("num_predict", lane.maxTokens()),
                    null);
            return new Label(out.get("response").toString().strip(),
                             lane.name() + " lane", "");
        } catch (OllamaClient.NoModelAvailable e) {
            return new Label(ruleFloor(message), "rule floor", e.getMessage());
        }
    }
```

Listing 6.7 (Continued) *TwoLanes.java: the two-lane router in Java, with record, ReentrantLock and threads.*

```
    void draft(String job, String prompt) {
        int position;
        synchronized (this) { position = waiting++; }
        if (position > 0) {
            log("  " + job + ": queued, position " + position + " in quality lane");
        }
        qualityLock.lock();                          // one long job at a time
        try {
            log("  " + job + ": drafting on " + QUALITY.model() + " ...");
            OllamaClient client = new OllamaClient(host, QUALITY.timeoutS());
            StringBuilder text = new StringBuilder();
            client.stream(QUALITY.model(), prompt,
                          Map.of("num_predict", QUALITY.maxTokens()),
                          c -> text.append(c.get("response")));
            String t = text.toString();
            log("  " + job + ": done, " + t.split("\\s+").length + " words: "
                + t.substring(0, Math.min(34, t.length())) + "...");
        } catch (OllamaClient.NoModelAvailable e) {
            log("  " + job + ": no local model available (" + e.getMessage() + ")");
        } finally {
            synchronized (this) { waiting--; }
            qualityLock.unlock();
        }
    }

    static final String PROMPT = """
            Business: Profitina Laundry
            - a laundry service in Surabaya
            - open every day from 07:00 to 21:00
            - express service is ready in 6 hours""";

    static final List<String> MESSAGES = List.of(
            "Is express really ready in 6 hours?",
            "My order came back late and one shirt was torn",
            "Thanks, the ironing was great");

    static String show(String label, String message, String source) {
        return String.format("  %-9s <- '%s' [%s]", label, message, source);
    }
```

Listing 6.7 (Continued) *TwoLanes.java: the two-lane router in Java, with record, ReentrantLock and threads.*

```
    public static void main(String[] args) throws InterruptedException {
        List<String> a = List.of(args);
        String host = a.contains("--host") ? a.get(a.indexOf("--host") + 1)
                                           : "http://127.0.0.1:7060";
        TwoLanes router = new TwoLanes(host);
        System.out.println("1. Two long drafts go to the quality lane:");
        Thread[] jobs = new Thread[2];
        String[] names = {"draft A", "draft B"};
        for (int k = 0; k < 2; k++) {
            String job = names[k];
            jobs[k] = new Thread(() -> router.draft(job, PROMPT));
            jobs[k].start();
            jobs[k].join(200);                    // let A take the lane first
        }
        System.out.println("2. Meanwhile, short labels go to the fast lane:");
        for (String m : MESSAGES) {
            Label r = router.classify(m, FAST);
            router.log(show(r.label(), m, r.source()));
        }
        for (Thread t : jobs) t.join();
        System.out.println(
                "3. The fast lane loses its model; the rule floor answers:");
        Lane gone = new Lane("fast", "stub-missing", 3.0, 8);
        String why = "";
        for (String m : MESSAGES.subList(0, 2)) {
            Label r = router.classify(m, gone);
            System.out.println(show(r.label(), m, r.source()));
            why = r.why();
        }
        System.out.println("  (the fast lane said: " + why + ")");
    }
}
```

To make the queue visible, restart the stub so that it pauses 100 milliseconds per streamed token, and run the router:

```
python stub_ollama.py --sleep-ms 100
python two_lanes.py
```

```
1. Two long drafts go to the quality lane:
  draft A: drafting on stub-quality ...
  draft B: queued, position 1 in quality lane
2. Meanwhile, short labels go to the fast lane:
  question  <- 'Is express really ready in 6 hours?' [fast lane]
  complaint <- 'My order came back late and one shirt was torn' [fast lane]
  praise    <- 'Thanks, the ironing was great' [fast lane]
  draft A: done, 22 words: Profitina Laundry is a laundry ser...
  draft B: drafting on stub-quality ...
  draft B: done, 22 words: Profitina Laundry is a laundry ser...
3. The fast lane loses its model; the rule floor answers:
  question  <- 'Is express really ready in 6 hours?' [rule floor]
  complaint <- 'My order came back late and one shirt was torn' [rule floor]
  (the fast lane said: HTTP 404: model 'stub-missing' not found)
```

Read the output as a timeline. Draft A started drafting and draft B was told, truthfully, that it was first in line. While draft A was still streaming its 22 words, all three labels came back from the fast lane, because they did not have to wait behind the draft; that is the whole point of **Figure 6.2**. Only when draft A finished did draft B start. In act 3 the fast lane's model was missing, the rule floor labeled both messages correctly, and the program said which source answered and why the fast lane failed. The Java version printed exactly the same lines, against either stub.

6.8.7 A quality gate in code

Listing 6.8 implements Equation (6.7) for Profitina Laundry.

Listing 6.8 `quality_gate.py`: hard checks, a soft score, and a strict comparison against a correct template.

```
"""A quality gate: serve an AI draft only when it beats a template.

The checklist below is this lab's own, chosen by the reader. Start
stub_ollama.py first (any --sleep-ms).

    python quality_gate.py [--host URL]
"""
import re
import sys
import textwrap

from ollama_client import NoModelAvailable, OllamaClient

NAME = "Profitina Laundry"
FACTS = ["a laundry service in Surabaya",
         "open every day from 07:00 to 21:00",
         "express service is ready in 6 hours"]
BANNED = ["best", "terbaik", "number one", "#1", "guaranteed", "cheapest"]


def template():
    """The safe fallback: correct, complete, and a little stiff."""
    return f"{NAME} - " + "; ".join(FACTS) + "."


def prompt():
    lines = [f"Business: {NAME}"] + [f"- {f}" for f in FACTS]
    return "\n".join(lines)


def hard_checks(text):
    """Every check must pass, or the draft is never served."""
    fails = []
    words = text.split()
    if not 5 <= len(words) <= 120:
        fails.append(f"length {len(words)} words is outside 5..120")
    first = re.split(r"(?<=\.)\s", text, maxsplit=1)[0]
    if NAME not in first:
        fails.append("the first sentence does not name the business")
    low = text.lower()
    for b in BANNED:
        if re.search(r"(?<!\w)" + re.escape(b) + r"(?!\w)", low):
            fails.append(f"banned phrase {b!r}")
    known = set(re.findall(r"\d+", " ".join(FACTS)))
    for n in re.findall(r"\d+", text):
        if n not in known:
```

Listing 6.8 (Continued) *quality_gate.py: hard checks, a soft score, and a strict comparison against a correct template.*

```
            fails.append(f"number {n} is not in the facts")
    return fails


def score(text):
    """Soft score: facts covered (0..3) + 1 if it opens with 'NAME is'."""
    low = text.lower()
    covered = sum(all(w in low for w in f.lower().split()[-3:])
                  for f in FACTS)
    return covered + (1 if text.startswith(NAME + " is ") else 0)


def show(label, text):
    print(textwrap.fill(text, 78, initial_indent=f"  {label:<9}: ",
                        subsequent_indent=" " * 13, break_on_hyphens=False))


def gate(client, model, seed):
    base = template()
    try:
        ai = client.generate(model, prompt(), {"seed": seed})["response"]
    except NoModelAvailable as e:
        return base, f"template (no local model available: {e})", False
    fails = hard_checks(ai)
    show("AI draft", ai)
    if fails:
        return base, "template (AI draft failed: " + "; ".join(fails) + ")", \
            False
    s_ai, s_t = score(ai), score(base)
    show("scores", f"AI {s_ai}, template {s_t}")
    if s_ai > s_t:
        return ai, "AI draft (it beats the template)", True
    return base, "template (AI draft did not beat it)", False


def main(argv):
    host = argv[argv.index("--host") + 1] if "--host" in argv \
        else "http://127.0.0.1:7060"
    client = OllamaClient(host, timeout=60)
    wins = 0
    cases = [("stub-quality", 0), ("stub-quality", 1), ("stub-quality", 2),
             ("stub-missing", 0)]
    for i, (model, seed) in enumerate(cases, 1):
        print(f"Case {i}: model={model}, seed={seed}")
        text, why, ai_won = gate(client, model, seed)
        wins += ai_won
```

Listing 6.8 (Continued) *quality_gate.py: hard checks, a soft score, and a strict comparison against a correct template.*

```
        show("decision", why)
        show("SERVED", text)
    print(f"AI drafts served in {wins} of {len(cases)} cases.")


if __name__ == "__main__":
    main(sys.argv[1:])
```

`FACTS` holds three true statements about the laundry; they are the only source of truth. `template` builds a correct, complete and stiff text from them, and `prompt` hands the same facts to the model. `hard_checks` implements $H(d)$ of Equation (6.7) as a list of reasons, so an empty list means "passed" and a non-empty list explains the failure. Its most interesting rule is the last: every

number in the draft must appear somewhere in the facts, which catches invented prices, years and phone numbers with one regular expression. `score` implements *S*: one point for each fact whose last three words all appear in the text, plus one point for opening with "Profitina Laundry is", the answer-first form. `gate` combines them exactly as **Figure 6.3** draws it, including the honest branch when no model is available. **Listing 6.9** is the Java version; its `show` method reproduces the line wrapping of Python's `textwrap.fill`, so that both programs print identical lines.

Listing 6.9 `QualityGate.java`: the same gate in Java; `show` reproduces Python's `textwrap.fill` so both print identical lines.

```
import java.util.ArrayList;
import java.util.HashSet;
import java.util.List;
import java.util.Map;
import java.util.Set;
import java.util.regex.Matcher;
import java.util.regex.Pattern;

/** A quality gate: serve an AI draft only when it beats a template.
 *  The checklist is this lab's own. Start StubOllama first.
 *  java QualityGate [--host URL] */
public class QualityGate {
    static final String NAME = "Profitina Laundry";
    static final List<String> FACTS = List.of(
            "a laundry service in Surabaya",
            "open every day from 07:00 to 21:00",
            "express service is ready in 6 hours");
    static final List<String> BANNED = List.of(
            "best", "terbaik", "number one", "#1", "guaranteed", "cheapest");

    record Outcome(String text, String why, boolean aiWon) {}

    /** The safe fallback: correct, complete, and a little stiff. */
    static String template() {
        return NAME + " - " + String.join("; ", FACTS) + ".";
    }

    static String prompt() {
        StringBuilder b = new StringBuilder("Business: " + NAME);
        for (String f : FACTS) b.append("\n- ").append(f);
        return b.toString();
    }

    static List<String> numbers(String text) {
        List<String> out = new ArrayList<>();
        Matcher m = Pattern.compile("\\d+").matcher(text);
        while (m.find()) out.add(m.group());
        return out;
    }
```

Listing 6.9 (Continued) *QualityGate.java: the same gate in Java; show reproduces Python's textwrap.fill so both print identical lines.*

```
    /** Every check must pass, or the draft is never served. */
    static List<String> hardChecks(String text) {
        List<String> fails = new ArrayList<>();
        int words = text.strip().split("\\s+").length;
        if (words < 5 || words > 120) {
            fails.add("length " + words + " words is outside 5..120");
        }
        String first = text.split("(?<=\\.)\\s", 2)[0];
        if (!first.contains(NAME)) {
            fails.add("the first sentence does not name the business");
        }
        String low = text.toLowerCase();
        for (String b : BANNED) {
            if (Pattern.compile("(?<!\\w)" + Pattern.quote(b) + "(?!\\w)")
                       .matcher(low).find()) {
                fails.add("banned phrase '" + b + "'");
            }
        }
        Set<String> known = new HashSet<>(numbers(String.join(" ", FACTS)));
        for (String n : numbers(text)) {
            if (!known.contains(n)) {
                fails.add("number " + n + " is not in the facts");
            }
        }
        return fails;
    }

    /** Soft score: facts covered (0..3) + 1 if it opens with 'NAME is'. */
    static int score(String text) {
        String low = text.toLowerCase();
        int covered = 0;
        for (String f : FACTS) {
            String[] w = f.toLowerCase().split(" ");
            boolean all = true;
            for (int k = Math.max(0, w.length - 3); k < w.length; k++) {
                all &= low.contains(w[k]);
            }
            if (all) covered++;
        }
        return covered + (text.startsWith(NAME + " is ") ? 1 : 0);
    }
```

Listing 6.9 (Continued) *QualityGate.java: the same gate in Java; show reproduces Python's textwrap.fill so both print identical lines.*

```
    /** Greedy word wrap, the same as Python's textwrap.fill here. */
    static void show(String label, String text) {
        String indent = String.format("  %-9s: ", label);
        StringBuilder line = new StringBuilder(indent);
        boolean empty = true;
        for (String w : text.split(" ")) {
            if (!empty && line.length() + 1 + w.length() > 78) {
                System.out.println(line);
                line = new StringBuilder(" ".repeat(13));
                empty = true;
            }
            if (!empty) line.append(' ');
            line.append(w);
            empty = false;
        }
        System.out.println(line);
    }

    static Outcome gate(OllamaClient client, String model, long seed) {
        String base = template();
        String ai;
        try {
            ai = client.generate(model, prompt(), Map.of("seed", seed), null)
                       .get("response").toString();
        } catch (OllamaClient.NoModelAvailable e) {
            return new Outcome(base, "template (no local model available: "
                               + e.getMessage() + ")", false);
        }
        List<String> fails = hardChecks(ai);
        show("AI draft", ai);
        if (!fails.isEmpty()) {
            return new Outcome(base, "template (AI draft failed: "
                               + String.join("; ", fails) + ")", false);
        }
        int sAi = score(ai), sT = score(base);
        show("scores", "AI " + sAi + ", template " + sT);
        if (sAi > sT) {
            return new Outcome(ai, "AI draft (it beats the template)", true);
        }
        return new Outcome(base, "template (AI draft did not beat it)", false);
    }
```

Listing 6.9 (Continued) *QualityGate.java: the same gate in Java; show reproduces Python's textwrap.fill so both print identical lines.*

```
    public static void main(String[] args) {
        List<String> a = List.of(args);
        String host = a.contains("--host") ? a.get(a.indexOf("--host") + 1)
                                           : "http://127.0.0.1:7060";
        OllamaClient client = new OllamaClient(host, 60.0);
        String[] models = {"stub-quality", "stub-quality", "stub-quality",
                           "stub-missing"};
        long[] seeds = {0, 1, 2, 0};
        int wins = 0;
        for (int i = 0; i < models.length; i++) {
            System.out.println("Case " + (i + 1) + ": model=" + models[i]
                               + ", seed=" + seeds[i]);
            Outcome o = gate(client, models[i], seeds[i]);
            if (o.aiWon()) wins++;
            show("decision", o.why());
            show("SERVED", o.text());
        }
        System.out.println("AI drafts served in " + wins + " of " + models.length
                           + " cases.");
    }
}
```

With the stub running (at any speed), run the gate:

```
python quality_gate.py
```

```
Case 1: model=stub-quality, seed=0
  AI draft : Profitina Laundry is a laundry service in Surabaya. Open every
             day from 07:00 to 21:00. Express service is ready in 6 hours.
  scores   : AI 4, template 3
  decision : AI draft (it beats the template)
  SERVED   : Profitina Laundry is a laundry service in Surabaya. Open every
             day from 07:00 to 21:00. Express service is ready in 6 hours.
Case 2: model=stub-quality, seed=1
  AI draft : Profitina Laundry is the best laundry in town, trusted since
             1990. It is a laundry service in Surabaya.
  decision : template (AI draft failed: banned phrase 'best'; number 1990 is
             not in the facts)
  SERVED   : Profitina Laundry - a laundry service in Surabaya; open every day
             from 07:00 to 21:00; express service is ready in 6 hours.
Case 3: model=stub-quality, seed=2
  AI draft : Profitina Laundry: a laundry service in Surabaya.
  scores   : AI 1, template 3
  decision : template (AI draft did not beat it)
  SERVED   : Profitina Laundry - a laundry service in Surabaya; open every day
             from 07:00 to 21:00; express service is ready in 6 hours.
Case 4: model=stub-missing, seed=0
  decision : template (no local model available: HTTP 404: model
             'stub-missing' not found)
  SERVED   : Profitina Laundry - a laundry service in Surabaya; open every day
             from 07:00 to 21:00; express service is ready in 6 hours.
AI drafts served in 1 of 4 cases.
```

Case 1 is the draft we hope for: it passes every hard check, covers all three facts and opens in answer-first form, so it scores 4 against the template's 3 and is served. Case 2 would have been embarrassing to publish; the gate names both of its faults, the superlative and the invented year, and serves the template instead. Case 3 breaks no rule but covers only one fact, so it loses on the soft score. Case 4 has no model at all, and the decision line says so in plain words while the template is served. One AI draft in four was served, which is the right answer for these four drafts. It is also a reminder that the public figure of 5 of 6 in Section 6.6.3 describes Profitina's revised engine on its own benchmark, not what any model will do anywhere.

Finally, restart the stub with `--no-models` and run the gate again. Every case now reports that no local model is available and serves the labeled template:

```
Case 1: model=stub-quality, seed=0
  decision : template (no local model available: HTTP 404: model
             'stub-quality' not found)
  SERVED   : Profitina Laundry - a laundry service in Surabaya; open every day
             from 07:00 to 21:00; express service is ready in 6 hours.
Case 2: model=stub-quality, seed=1
  decision : template (no local model available: HTTP 404: model
             'stub-quality' not found)
  SERVED   : Profitina Laundry - a laundry service in Surabaya; open every day
             from 07:00 to 21:00; express service is ready in 6 hours.
Case 3: model=stub-quality, seed=2
  decision : template (no local model available: HTTP 404: model
             'stub-quality' not found)
  SERVED   : Profitina Laundry - a laundry service in Surabaya; open every day
             from 07:00 to 21:00; express service is ready in 6 hours.
Case 4: model=stub-missing, seed=0
  decision : template (no local model available: HTTP 404: model
             'stub-missing' not found)
  SERVED   : Profitina Laundry - a laundry service in Surabaya; open every day
             from 07:00 to 21:00; express service is ready in 6 hours.
AI drafts served in 0 of 4 cases.
```

6.8.8 The same code against a real Ollama

To see real model output on your own computer, install Ollama from its official website or its GitHub releases page [10], then download one quality model and one fast model and check that the server answers:

```
ollama pull qwen2.5:7b
ollama pull qwen3:1.7b
ollama list
python ollama_client.py --model qwen3:1.7b "Say hello in one short sentence."
```

With no `--host`, the client talks to `http://127.0.0.1:11434`, the real Ollama. To run the lane and gate programs against it, change the model names in `QUALITY`, `FAST` and `cases` from the `stub-…` names to the models you pulled, remove the `CLASSIFY:` convention in favor of a clear instruction such as “Answer with one word: question, complaint, praise or other”, and pass `--host http://127.0.0.1:11434`. Expect three differences from the stub. The text will be written by a model, so it will vary with the seed and temperature and will sometimes fail your hard checks, which is exactly what the gate is for. The durations will be real, so the rate from Equation (6.5) will tell you about your hardware. And the first request after a pause will include a noticeable `load_duration`, because Ollama loads a model into memory on first use and unloads it after a period of inactivity. Compare `127.0.0.1` with `localhost` while you are there: Exercise 6.4 asks you to.

6.9 Profitina in Practice

Everything in this chapter has a counterpart in the running product, described here at the level of boxes and arrows.

Local only. Every AI feature of app.profitina.com runs on Ollama with Qwen models on the platform’s own server. The hosted-AI code paths were removed entirely; user data never leaves the server for AI processing, and users are not asked to choose a model [6]. ChatGPT, Gemini and Perplexity appear in Profitina only as the answer engines whose readiness to cite a website is estimated; they are never called to power a feature. (Profitina AI, a separate local-first assistant

app in the same group, can route between local and cloud models for its own users; it is not part of app.profitina.com, and the GEO platform uses no cloud AI.)

Two lanes. The quality lane runs Qwen2.5 7B on the CPU for long-form work such as Content Engine drafts; the fast lane runs qwen3:1.7b on the GPU for short classification: scan category, Fair-Play usefulness and the optional Link Safety opinion [6]. In the September 2026 release, the Link Safety model opinion runs on the fast lane only, and the read path of the community feed calls no model at all (Section 6.5.4).

The Content Engine and its gate. Drafts, FAQs and JSON-LD come from the quality lane, and a quality gate serves an AI draft only when it beats a built-in template; after the September 2026 revision, AI drafts win in 5 of 6 benchmark cases. When no model is available, the app says so [6].

Honest progress and queues. Long operations show 0–100% progress with sub-step labels such as "drafting — 466 tokens" and a time-remaining estimate. *One Text, Many Platforms* processes one request at a time across the whole system and shows the queue position and the expected wait. What it does not promise is stated too: a "more engaging" or "viral" result is not promised [6].

Operations. The platform's health endpoint, `/api/health`, reports the state of the local AI next to the database, disk and uptime, and the admin Command Center shows the same health at a glance (Chapter 9). The `127.0.0.1` rule of Section 6.4.1 is written into the project's working rules because of the measured 2.25-second penalty.

What this chapter's lab is not. The stub, the lanes' timeouts and token limits, the hard checks and the soft score in Section 6.8 are this book's own teaching choices for Profitina Laundry. They show the ideas; the product has its own prompts, templates and gate, tuned on its own benchmark.

6.10 Common Misunderstandings

"Local AI means offline AI of lower quality, so it is only for hobbyists." Local means *where* the model runs, not how good it is. A 7B model writes clear factual paragraphs; the quality ceiling is real but it is managed by a gate that serves the model's work only when it is better than a correct template.

"If the data never leaves the server, privacy is solved." Local inference removes one large risk, copying data to a third party. It does not replace access control, retention limits, consent, or the right to object to automated decisions (Chapter 11). It is one privacy measure among several.

"A bigger or newer model is always an upgrade." A bigger model is slower and needs more memory by Equation (6.1); a newer family may have different strengths. A model is an upgrade when it wins on your benchmark, on your hardware, at your acceptable speed.

"One model and one queue are simpler, so they are better." They are simpler to draw and worse to use: by Equation (6.4), a label that needs a twentieth of a second can wait for a draft that needs a minute. Two lanes separate work by its length and urgency.

"If the AI is down, quietly use the template; users will not notice." Users will notice the quality and blame the wrong thing, and operators will not notice the outage. Serve the template if you must, but say that it is the template.

"5 of 6 means 83.3% of drafts are AI drafts." It means the revised engine beat the template on five of six benchmark cases. Each request still goes through the gate, and six cases are too few for a precise percentage; one case moves the rate by 16.7 points.

"`localhost` and `127.0.0.1` are identical, so it does not matter which I write." They reach the same machine, but name resolution can try IPv6 first. On Windows that cost Profitina more than two seconds per call until the address was written explicitly.

"Calling the model while rendering is fine if the model is fast." Rendering happens for every item on every view; a small cost per item becomes a large cost per page, and a model that is occasionally slow makes the whole page occasionally slow. Judge when content is created; read stored verdicts when it is shown.

6.11 Chapter Summary

- A language model predicts the next token of a text, again and again. Its knowledge is frozen at training time, its running time grows with the length of the answer, and its output involves chance, so every answer must be checked.
- A model with N parameters stored at b bits needs about $Nb/8$ bytes (Equation (6.1)); quantization to 4–5 bits lets a 7B model fit in about 4.7 GB.
- Profitina runs all AI locally for privacy (no brand or personal data copied to third parties, in line with UU PDP), cost (a fixed cost instead of a per-request meter, Equation (6.2) and Equation (6.3)) and control (a pinned model that changes only when a measurement says so). The price is a quality ceiling, hardware work, lower speed and queues.
- Ollama serves local models over HTTP on port 11434; `POST /api/generate` takes a model and a prompt and returns text plus token counts and nanosecond durations. The latest stable release on September 24, 2026 is v0.34.3. Call it at `127.0.0.1`.
- The Qwen family offers open-weight models from Qwen2.5 (2024) through Qwen3 (2025) and Qwen3.5, 3.6 and 3.8 (2026). Profitina's quality lane runs Qwen2.5 7B and its fast lane qwen3:1.7b.
- The two-lane model separates long, carefully checked drafts from short, urgent labels; a rule floor answers when the fast lane cannot, and the read path never calls a model.
- The quality gate serves an AI draft only if it passes every hard check and scores strictly higher than a correct template (Equation (6.7)). Profitina's public result is 5 wins in 6 benchmark cases.
- When no model is available, the software says so: a typed failure, a readable reason, and a labeled fallback.
- The lab built a stub of Ollama's interface, a client, a two-lane router and a quality gate in Python and Java; all printed output in this chapter came from the stub.

6.12 Exercises

Exercise 6.1 (memory). Use Equation (6.1) to estimate the memory needed for the weights of a 27-billion-parameter model (a) at 16 bits per parameter and (b) at 4.5 bits per parameter. Ollama lists `qwen3.5:27b` as a 17 GB download [15]. Give two reasons why the listed size can be larger than your estimate (b).

Exercise 6.2 (break-even). A small company pays $F = \text{Rp}24{,}000{,}000$ a year for a machine that will run a local model, spends $e = \text{Rp}4$ of electricity per request, and is offered a hosted model at $p = \text{Rp}40$ per request. These numbers are invented for practice. (a) Find the break-even volume

n^* per year and per day. (b) The hosted price falls to Rp20. What is the new n^*? (c) Name one cost that Equation (6.2) leaves out on each side.

Exercise 6.3 (latency and an honest queue). A quality lane generates $r = 6$ tokens per second. Drafts are about 300 tokens long, the model is already loaded, and reading the prompt takes about 2 seconds. (a) Estimate the time for one draft with Equation (6.4). (b) Two drafts are ahead of a user in the queue. Estimate the wait before the user's draft starts, with Equation (6.6), and the total time until it is finished. (c) Write the one-sentence message the user should see.

Exercise 6.4 (127.0.0.1 versus localhost). Install Ollama on your own computer and pull a small model. Measure the *time to first token* (the time from sending a streaming request until the first chunk arrives) ten times with the host `http://127.0.0.1:11434` and ten times with `http://localhost:11434`, and report the medians. Explain your result.

Exercise 6.5 (run the gate by hand). Using the hard checks and the soft score of **Listing 6.8**, decide what the gate serves for each draft, and why. The template scores 3.

(a) "Profitina Laundry is the cheapest laundry service in Surabaya, open every day from 07:00 to 21:00."

(b) "Open every day from 07:00 to 21:00, Profitina Laundry is a laundry service in Surabaya."

(c) "Profitina Laundry is a laundry service in Surabaya. Express service is ready in 6 hours, from Rp12,000 per kilogram."

(d) "Profitina Laundry is a laundry service in Surabaya, open every day from 07:00 to 21:00."

Exercise 6.6 (structured output). Ollama's `format` field can ask for JSON. Write a program, in Python and in Java, that asks the fast lane for `{"label": …}` for a customer message, accepts the answer only if it is valid JSON whose label is one of `question`, `complaint` or `praise`, retries once with a different seed, and otherwise reports honestly that no valid label was obtained and a human should decide. Run it against the stub for "Where is the shop?", "My jacket was lost" and "Good morning".

Exercise 6.7 (which lane?). For each job, say whether it belongs in the quality lane, the fast lane, or no lane at all, and why: (a) writing six FAQ answers for a bakery from its facts; (b) deciding whether a new post is written in Indonesian or English; (c) deciding, while the feed is being displayed, whether each link in 30 posts is safe; (d) suggesting a business category when the scanner's rules are unsure.

Exercise 6.8 (honest wording). Rewrite each message so that it is honest and useful to a non-technical user: (a) a Content Engine page that shows the template with the heading "Your AI draft"; (b) "Error 500"; (c) a spinner with no text that has been turning for 90 seconds while the user is fourth in the queue.

Exercise 6.9 (why strictly greater?). Equation (6.7) serves the AI draft only if $S(d) > S(t)$. Give one argument for this strict comparison and describe what would change in Case 1 and in draft (d) of Exercise 6.5 if it were replaced by $S(d) \geq S(t)$.

Exercise 6.10 (small samples). A gate result is reported as "w wins in n cases". (a) With $n = 6$, list every possible win rate. (b) How large must n be so that one case changes the rate by at most 2 percentage points? (c) Why is it still useful to publish a result based on six cases?

6.13 Solutions to the Exercises

Solution 6.1. (a) $27 \times 10^9 \times 16/8 = 54 \times 10^9$ bytes, about 54 GB. (b) $27 \times 10^9 \times 4.5/8 \approx 15.2 \times 10^9$ bytes, about 15.2 GB. The listed 17 GB is larger because (1) quantization formats keep some tensors, such as embeddings or normalization weights, at higher precision, so the average number of bits per parameter is above the nominal 4 or 4.5; and (2) Qwen3.5 models are multimodal, and the download includes the image encoder and metadata in addition to the language model's weights. At run time you also need memory for the context (the *KV cache*), which grows with the number of tokens in the conversation, so the machine needs more than the file size.

Solution 6.2. (a) By Equation (6.3), $n^* = 24{,}000{,}000/(40-4) = 24{,}000{,}000/36 \approx 666{,}667$ requests a year, or $666{,}667/365 \approx 1{,}827$ a day. (b) $n^* = 24{,}000{,}000/(20-4) = 1{,}500{,}000$ a year, about 4,110 a day: when hosted prices fall, local inference needs more volume to pay off. (c) Hosted side: the cost of the privacy and compliance risk of sending data out, and the risk of price changes or model retirement. Local side: the people-hours for installing, monitoring and updating the model server, and the quality ceiling, which may cost sales if drafts are weaker.

Solution 6.3. (a) $T \approx 0 + 2 + 300/6 = 52$ seconds. (b) With $k = 2$, Equation (6.6) gives $W \approx 2 \times 52 = 104$ seconds before the draft starts, and $104 + 52 = 156$ seconds, about 2.6 minutes, until it is finished. (c) For example: "Two drafts are ahead of yours. Estimated start in about 2 minutes, ready in about 3 minutes. You can leave this page open or come back later."

Solution 6.4. Measure with a small change to `stream` in **Listing 6.4**: record `time.perf_counter()` before `_open` and again when the first line arrives. On Linux and macOS the two hosts usually give the same median, a few tens of milliseconds after the model is loaded, because the system either tries IPv6 successfully or falls back instantly. On Windows, `localhost` may be resolved to the IPv6 address `::1` first; if Ollama listens only on IPv4, the first connection attempt must fail or time out before the client tries `127.0.0.1`, which adds a delay to every new connection. Profitina measured 2.25 seconds against 0.17 seconds on its Windows host. Whatever your numbers, the conclusion is to write the address you mean.

Solution 6.5. (a) Template: the hard check fails on the banned word "cheapest", whatever the score. (b) Template: the draft passes the hard checks (the business is named in the only sentence, and 07, 00, 21 and 00 all appear in the facts), but its score is 2: it covers two facts and does not open with "Profitina Laundry is", and $2 > 3$ is false. (c) Template: the numbers 12 and 000 in "Rp12,000" do not appear in the facts, so the draft invents a price, which is exactly the fault the number rule exists to catch. (d) Template, by a tie: the draft passes the hard checks and scores 3 (two facts plus the answer-first opening), equal to the template's 3, and a tie goes to the template. Running the checks from **Listing 6.8** in Python confirms all four results.

Solution 6.6. **Listing 6.10** and **Listing 6.11** give a solution.

Listing 6.10 `json_label.py`: ask for JSON, validate the shape and the allowed values, retry once, then hand over to a human.

```
"""Structured output: ask for JSON, validate it, retry once, then give up
honestly. Start stub_ollama.py first.

    python json_label.py [--host URL]
"""
import json
import sys

from ollama_client import NoModelAvailable, OllamaClient

ALLOWED = {"question", "complaint", "praise"}


def valid(text):
    """Return the label if text is {"label": <allowed>}, else None."""
    try:
        obj = json.loads(text)
    except json.JSONDecodeError:
        return None
    if isinstance(obj, dict) and obj.get("label") in ALLOWED:
        return obj["label"]
    return None


def label(client, message, attempts=2):
    for attempt in range(1, attempts + 1):
        try:
            out = client.generate("stub-fast", "CLASSIFY: " + message,
                                  {"seed": attempt}, fmt="json")
        except NoModelAvailable as e:
            return None, f"no local model available ({e})"
        got = valid(out["response"])
        if got:
            return got, f"valid JSON on attempt {attempt}"
        print(f"    attempt {attempt}: rejected {out['response']!r}")
    return None, f"no valid label after {attempts} attempts; a human decides"
```

Listing 6.10 (Continued) *json_label.py: ask for JSON, validate the shape and the allowed values, retry once, then hand over to a human.*

```
def main(argv):
    host = argv[argv.index("--host") + 1] if "--host" in argv \
        else "http://127.0.0.1:7060"
    client = OllamaClient(host, timeout=10)
    for m in ["Where is the shop?", "My jacket was lost", "Good morning"]:
        print(f"{m!r}")
        got, why = label(client, m)
        print(f"    -> {got or 'UNLABELED'} ({why})")


if __name__ == "__main__":
    main(sys.argv[1:])
```

Listing 6.11 `JsonLabel.java`: the same validation and retry in Java, using `MiniJson`.

```
import java.util.List;
import java.util.Map;
import java.util.Set;

/** Structured output: ask for JSON, validate it, retry once, then give up
 *  honestly. Start StubOllama first.  java JsonLabel [--host URL] */
public class JsonLabel {
    static final Set<String> ALLOWED = Set.of("question", "complaint", "praise");

    record Result(String label, String why) {}

    /** Return the label if text is {"label": <allowed>}, else null. */
    static String valid(String text) {
        try {
            Object label = MiniJson.parseObject(text).get("label");
            return ALLOWED.contains(label) ? (String) label : null;
        } catch (RuntimeException e) {
            return null;
        }
    }

    static Result label(OllamaClient client, String message, int attempts) {
        for (int attempt = 1; attempt <= attempts; attempt++) {
            Map<String, Object> out;
            try {
                out = client.generate("stub-fast", "CLASSIFY: " + message,
                                      Map.of("seed", attempt), "json");
            } catch (OllamaClient.NoModelAvailable e) {
                return new Result(null, "no local model available ("
                                  + e.getMessage() + ")");
            }
            String text = out.get("response").toString();
            String got = valid(text);
            if (got != null) {
                return new Result(got, "valid JSON on attempt " + attempt);
            }
            System.out.println("    attempt " + attempt + ": rejected '"
                               + text + "'");
        }
        return new Result(null, "no valid label after " + attempts
                          + " attempts; a human decides");
    }
```

Listing 6.11 (Continued) *JsonLabel.java: the same validation and retry in Java, using MiniJson.*

```
    public static void main(String[] args) {
        List<String> a = List.of(args);
        String host = a.contains("--host") ? a.get(a.indexOf("--host") + 1)
                                           : "http://127.0.0.1:7060";
        OllamaClient client = new OllamaClient(host, 10.0);
        for (String m : List.of("Where is the shop?", "My jacket was lost",
                                "Good morning")) {
            System.out.println("'" + m + "'");
            Result r = label(client, m, 2);
            String shown = r.label() == null ? "UNLABELED" : r.label();
            System.out.println("    -> " + shown + " (" + r.why() + ")");
        }
    }
}
```

Against the stub, both programs print:

```
'Where is the shop?'
    -> question (valid JSON on attempt 1)
'My jacket was lost'
    -> complaint (valid JSON on attempt 1)
'Good morning'
    attempt 1: rejected '{"label": "other"}'
    attempt 2: rejected '{"label": "other"}'
    -> UNLABELED (no valid label after 2 attempts; a human decides)
```

The first two messages receive valid labels at once. For “Good morning” the model twice returns well-formed JSON with the label `other`, which is not in the allowed set, so validation rejects it, the retry changes nothing, and the program says so instead of accepting a label that the rest of the system does not expect. Two points generalize: validate the *content* of structured output, not just its syntax; and bound the retries, because a model that fails twice in the same way usually fails a third time too.

Solution 6.7. (a) Quality lane: long, published text, worth checking with a gate. (b) No lane is needed in most cases: a character and word-list rule identifies Indonesian or English reliably, and the fast lane can be a fallback for mixed text. (c) No lane at display time: judge each link when it is posted (rule floor plus an optional fast-lane opinion) and store the verdict; the feed only reads stored verdicts (Section 6.5.4). (d) Fast lane with a rule floor: a short label needed while the user waits, and when the model is unsure or unavailable, the honest behavior is to say “not sure yet” and ask the user to pick.

Solution 6.8. (a) “Template draft (the AI writer is unavailable right now). This text was built directly from your facts. Try again in a few minutes for an AI draft.” (b) “Something went wrong on our side while creating your draft. Nothing you entered was lost. Please try again; if it keeps happening, tell us at info@profitina.com.” (c) “You are number 4 in line. Estimated wait: about 3 minutes. Drafting — 120 of about 400 tokens.”

Solution 6.9. The strict comparison makes the predictable, correct template the default whenever the AI adds nothing measurable; serving generated text carries risks that the score does not capture, so the generated text must earn its place. With $S(d) \geq S(t)$, Case 1 would not change (4 is greater than 3), but draft (d) of Exercise 6.5, which ties at 3, would be served instead of the template.

Solution 6.10. (a) $0, \frac{1}{6}, \frac{2}{6}, \frac{3}{6}, \frac{4}{6}, \frac{5}{6}, 1$, that is 0%, 16.7%, 33.3%, 50%, 66.7%, 83.3% and 100%. (b) One case changes the rate by $1/n$, and $1/n \leq 0.02$ requires $n \geq 50$. (c) Because it is honest and checkable: it says what was measured and on how many cases, it shows that the revision changed the outcome in the right direction, and it is backed by a gate that decides every single request on its merits, so no user depends on the percentage being precise.

References

[1] A. Vaswani *et al.*, “Attention is all you need,” in *Advances in Neural Information Processing Systems 30 (NIPS 2017)*, 2017, pp. 5998–6008.

[2] Stack Overflow, “AI,” *2025 Stack Overflow Developer Survey*, Jul. 2025. [Online]. Available: https://survey.stackoverflow.co/2025/ai

[3] T. Dettmers and L. Zettlemoyer, “The case for 4-bit precision: k-bit inference scaling laws,” in *Proc. 40th Int. Conf. Machine Learning (ICML)*, 2023, pp. 7750–7774.

[4] ggml-org, “GGUF file format,” *ggml documentation*. [Online]. Available: https://github.com/ggml-org/ggml/blob/master/docs/gguf.md (accessed Sep. 24, 2026).

[5] Ollama, "qwen2.5," *Ollama model library*. [Online]. Available: https://ollama.com/library/qwen2.5 (accessed Sep. 24, 2026).

[6] Profitina, *Profitina Whitepaper — Public Version v24*, Sep. 2026. [Online]. Available: https://app.profitina.com/static/whitepaper/

[7] Cisco Systems, "Cisco's 2025 Data Privacy Benchmark Study: Privacy landscape grows increasingly complex in the age of AI," press release, Apr. 2, 2025. [Online]. Available: https://newsroom.cisco.com/c/r/newsroom/en/us/a/y2025/m04/cisco-2025-data-privacy-benchmark-study-privacy-landscape-grows-increasingly-complex-in-the-age-of-ai.html

[8] Cisco Systems, "AI fuels surge in data privacy investments and redefines governance, Cisco reports (Cisco 2026 Data and Privacy Benchmark Study)," press release, Jan. 26, 2026. [Online]. Available: https://newsroom.cisco.com/c/r/newsroom/en/us/a/y2026/m01/ai-data-privacy-investments-governance-cisco-report.html

[9] Republik Indonesia, *Undang-Undang Nomor 27 Tahun 2022 tentang Pelindungan Data Pribadi* (UU PDP), Oct. 17, 2022, esp. Art. 10(1). [Online]. Available: https://jdih.komdigi.go.id/produk_hukum/view/id/832/t/undangundang+nomor+27+tahun+202

[10] Ollama, "Releases: v0.34.3 (Sep. 19, 2026) and v0.34.4-rc0 (Sep. 23, 2026)," *GitHub*. [Online]. Available: https://github.com/ollama/ollama/releases (accessed Sep. 24, 2026).

[11] Ollama, "Generate a response (POST /api/generate)," *Ollama API documentation*. [Online]. Available: https://docs.ollama.com/api/generate (accessed Sep. 24, 2026).

[12] Qwen Team, A. Yang *et al.*, "Qwen2.5 technical report," arXiv:2412.15115, Dec. 2024 (rev. Jan. 2025).

[13] A. Yang *et al.*, "Qwen3 technical report," arXiv:2505.09388, May 2025.

[14] Ollama, "qwen3," *Ollama model library*. [Online]. Available: https://ollama.com/library/qwen3 (accessed Sep. 24, 2026).

[15] Ollama, "qwen3.5," *Ollama model library*. [Online]. Available: https://ollama.com/library/qwen3.5 (accessed Sep. 24, 2026).

[16] Wikipedia contributors, "Qwen," *Wikipedia* (secondary source for the 2026 release list). [Online]. Available: https://en.wikipedia.org/wiki/Qwen (accessed Sep. 24, 2026).

[17] Qwen Team, "Qwen3.8," *GitHub repository QwenLM/Qwen3.8*, news of Aug. 12 and Aug. 14, 2026. [Online]. Available: https://github.com/QwenLM/Qwen3.8 (accessed Sep. 24, 2026).

[18] T. Bray, Ed., "The JavaScript Object Notation (JSON) data interchange format," IETF RFC 8259, Dec. 2017.

Chapter 7

The Feedback Loop: Humans Decide, Models Learn

By the end of this chapter you will be able to explain why an AI product that cannot learn from its users slowly falls behind them, and why one that learns from them without a human in charge is dangerous; walk through Profitina's five-step feedback pipeline, submit → curate → dataset → train and measure → release, and say at each step what the machine does, what a person decides, and what the person who gave the feedback can see; turn curated feedback into a dataset of preference pairs in the JSON Lines format; measure a model before and after with a held-out set, compute a pass rate and find regressions; explain how Profitina rewards people with small amounts of PROFIT for improving AI responses, why the rewards have daily caps that reset at midnight Western Indonesia Time, and why only verified humans earn; and state the principle that runs through the whole chapter, *the app advises, the user decides*. The lab builds a complete miniature pipeline in Python and Java: it takes eleven feedback items about Profitina Laundry's help answers, sorts them, applies a human's decisions, writes a JSONL dataset, awards lab rewards under a daily cap, measures eight held-out questions before and after, and stops to wait for a human release decision.

7.1 Where We Left Off

Chapter 6 built the machinery of local AI: two lanes, a quality gate that serves an AI draft only when it beats a template, and honest behavior when no model is available. All of that makes a model *safe to use* today. None of it makes the model *better* tomorrow.

A model left alone does not improve. Its knowledge is frozen at the end of training, it repeats the same small mistakes, and the world it describes moves on: a laundry changes its prices, a new payment method appears, a regulation takes effect. The people who notice these mistakes first are the users. A brand owner reads a Content Engine draft and sees that the FAQ answer about opening hours is vague. A student in a course channel notices that a scan explanation uses a term incorrectly. Each of them holds a small piece of knowledge the model lacks.

This chapter is about collecting those pieces, carefully and fairly, and turning them into improvements without ever letting the machine decide on its own what "better" means. It answers four questions. How does feedback travel from a person to a model (Sections 7.2 to 7.4)? How is it cleaned, stored and measured (Sections 7.5 to 7.7)? Who decides whether an improvement ships (Section 7.8)? And how does Profitina thank the people who help, without creating a machine that pays bots to generate noise (Section 7.9)?

7.2 Why Feedback, and Why Humans Decide

7.2.1 Models learn from people

Every useful language model today was shaped by human judgment after its initial training. The best-known method, *reinforcement learning from human feedback* (RLHF), asks people to

compare two answers from the same model and say which is better; those comparisons train the model to prefer answers people prefer [1], [2]. The InstructGPT study that popularized the method found that people preferred the answers of a 1.3-billion-parameter model tuned this way over those of a 175-billion-parameter model without it [1]. The lesson for a small team is encouraging: a modest model plus careful human judgment can beat a much larger model without it.

Comparisons are also easier for people than writing perfect answers from scratch. It is hard to write the ideal FAQ answer about express service; it is easy to say "this answer is wrong, it is six hours, not two days" and suggest a better sentence. Profitina's feedback pipeline is built on exactly that kind of small, concrete correction.

7.2.2 Why a human must decide

If feedback is so valuable, why not let the system take every suggestion and retrain itself automatically? Three reasons, each of which appears in the lab.

1. **Feedback can be wrong.** A user may suggest a cheaper price because they wish it were cheaper, or copy an answer from a competitor. The lab's item F08 does exactly that.
2. **Feedback can be hostile or automated.** Any system that rewards input will attract people, and programs, that produce input only to collect the reward. Chapter 5 described how Fair-Play defends the community; the same defenses protect the feedback loop.
3. **An improvement in one place can break another.** A correction that makes one answer precise can remove a detail another question depended on. Only measurement can reveal this, and only a person can weigh a fix against a regression.

Surveys of practitioners point the same way. In the 2025 Stack Overflow Developer Survey, the most common frustration with AI tools, named by 66% of respondents, was "AI solutions that are almost right, but not quite", and 75.3% said they would still ask a person "when I don't trust AI's answers" [3]. Research on human–AI interaction recommends designs that make clear what the system can do, support efficient correction, and let people dismiss or override it [4].

7.2.3 The app advises, the user decides

Profitina writes the principle down as a house rule: *the app advises; the user decides; never present output as a guarantee* [5]. It applies in two directions.

Towards users of the product, it means a visibility score is a methodological estimate, a Content Engine draft is a suggestion, and the platform says so: "Visibility results are methodological estimates, not guarantees of actual third-party AI engine output" [5]. When the category auto-fill is unsure, it says "not sure yet" and asks the user to pick, instead of guessing silently.

Towards the product itself, it means every decision that changes what the product tells people, whether a suggestion enters the dataset or whether a new model version ships, is made by a person, with the pipeline's measurements as advice. Indonesian law points in the same direction: Article 10 of UU PDP No. 27 of 2022 gives people the right to object to decisions based solely on automated processing that significantly affect them [6]. A feedback loop in which people decide is not only better engineering; it is the kind of system the law expects.

> **DEFINITION.** A *human-in-the-loop* process is one in which the machine proposes, sorts or measures, and a named person makes each decision that changes the product's behavior, with a recorded reason. In this chapter, the pipeline *advises* and a human *decides*.

7.2.4 A story in five steps

Before the details, here is the whole loop as a story. Ibu Sari runs a small bakery in Sidoarjo and uses Profitina to make her website easier for answer engines to quote. The Content Engine drafts an FAQ for her, and one answer says the bakery "delivers across East Java". Her bakery delivers only within 5 km. She taps the feedback button on that answer, writes "Wrong: we deliver only within 5 km of the shop", and suggests a corrected sentence. A thank-you appears, with a small PROFIT reward.

That evening her item is one of forty in the review queue. The sorting call has labeled it *wrong fact*, severity *medium*, and grouped it with two similar reports from other bakeries about delivery areas. A reviewer reads the group, sees that the drafts invent delivery areas when a business gives none, and accepts Ibu Sari's item with the reason "delivery areas must come from the user's facts". Her dashboard now shows "In dataset #212".

A week later the next candidate is measured. On held-out questions about delivery, it now asks for or uses the business's own area instead of inventing one; two unrelated answers got slightly worse. The reviewer reads the report, decides the regressions are minor wording issues, records that decision, and approves. Ibu Sari's dashboard reads "Shipped in v1.4", and, because she opted in, her pseudonym appears on the "You changed this" page. She did not need to know anything about models to improve one. That is the promise of a transparent, human-governed feedback loop; the rest of the chapter explains each step. (Ibu Sari, her bakery and the numbers in this story are invented for illustration.)

7.3 The Five-Step Pipeline

Profitina's feedback pipeline has five steps, all live as of September 2026 [5]. **Figure 7.1** shows them with what the person who gave the feedback sees at each step, and where a human decides.

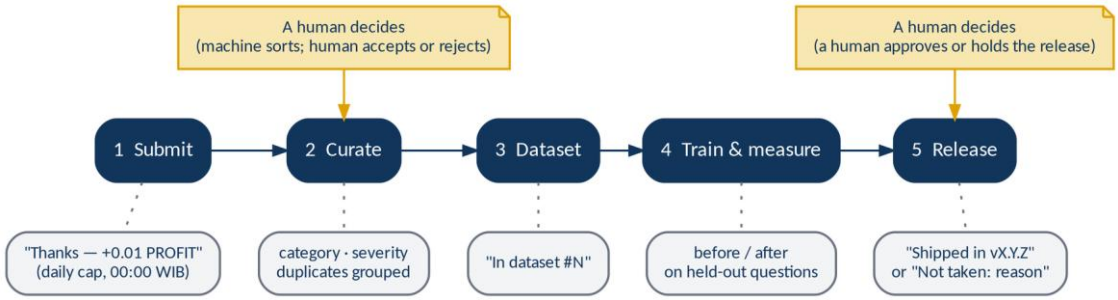


Figure 7.1 The five-step feedback pipeline. Dark boxes are the steps; grey boxes show what the giver sees; the amber notes mark the two places where a human decides. After a release, the loop starts again with new feedback.

Step 1: Submit. A user sends feedback about something the AI produced: a scan explanation, a Content Engine draft, a category suggestion. Feedback on scans can earn a small amount of PROFIT, within a daily cap (Section 7.9). The giver sees a thank-you and, where applicable, the reward.

Step 2: Curate. One local-AI call on the fast lane sorts each item: it proposes a category, a severity and a one-line summary, and it groups items that say the same thing. Then a human reviews the sorted queue and accepts or rejects each item, with a reason. The giver sees the status change.

Step 3: Dataset. Accepted items are added to a numbered training dataset. The giver sees "In dataset #N", a visible, verifiable sign that their feedback was used.

Step 4: Train and measure. The dataset is used to improve the local model's behavior, and every candidate is measured on held-out examples before and after, so that improvements and regressions are both visible. Whether and when weights are fine-tuned is a technical choice inside this step; a full fine-tune of the model's weights with a method such as LoRA [7] is a roadmap item as of September 2026, and this book does not describe it as live.

Step 5: Release. A human decides whether the candidate ships. The giver sees "Shipped in vX.Y.Z" or "Not taken", always with a reason. Givers who opt in are credited by pseudonym on a public "You changed this" page [5].

Two properties make the pipeline trustworthy. It is *transparent*: at every step the giver can see where their item is, which turns "we value your feedback" from a slogan into a status line. And it is *human-governed*: product decisions are always made by a person [5]. The machine does the tiring parts, sorting and grouping and measuring, and a person does the part that requires judgment.

7.4 The Life of One Feedback Item

It helps to follow a single item through the pipeline as a *state machine*: a small set of named states and the allowed moves between them. **Figure 7.2** draws the states used in this chapter's lab, which mirror the statuses the product shows.

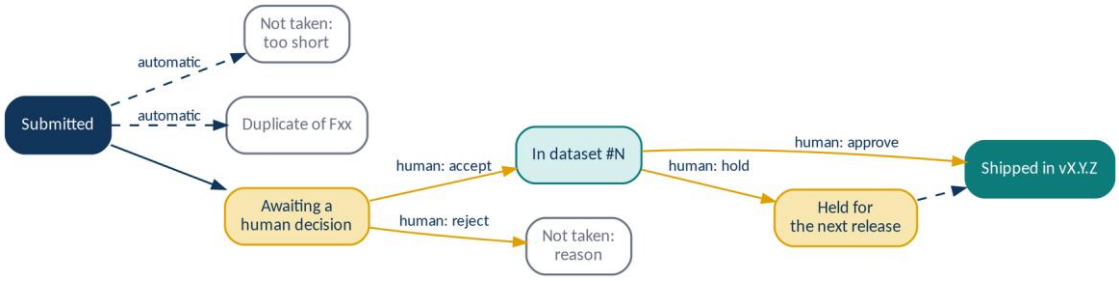


Figure 7.2 The life of one feedback item. Dashed moves are automatic and explainable; amber moves are human decisions. Only accepted items can ever be shipped.

Table 7.1 lists each state, who puts an item there, and what the giver reads.

Table 7.1 The states of a feedback item in this chapter's lab, who moves an item into each state, and what the giver sees.

State	Who decides	What the giver reads
Submitted	the giver	"Thanks, your feedback was received"
Not taken: too short	automatic rule	"Not taken: too short"
Duplicate of F*xx*	automatic rule	"Duplicate of F*xx*", with a link to the earlier item
Awaiting a human decision	nobody yet	"Waiting for review"
In dataset *#N*	a human (accept)	"In dataset *#N*"
Not taken: *reason*	a human (reject)	"Not taken", with the reason
Shipped in v*X.Y.Z*	a human (approve release)	"Shipped in v*X.Y.Z*"
Held for the next release	a human (hold release)	"In dataset *#N*; not shipped yet"

Three rules are visible in the table. Automatic moves only ever *remove* noise (too short, duplicate), and each says why. Every move that changes what the product will do is human. And there is no state in which an item silently disappears: even a rejection comes with a reason.

Follow one item from the lab to see the states at work. F06 arrives from user u5 at 12:00 WIB: "Yes, free pickup and delivery within 3 km." It is long enough and it is not a near-duplicate of anything earlier, so no automatic rule touches it, and it enters *Awaiting a human decision*. The reviewer checks the distance with the shop owner and accepts it, so it moves to *In dataset #5*. When the candidate that contains it is measured and a person approves the release, it would move to *Shipped in v1.1*; in the lab's first run the release is held, so it stays *In dataset #5* and is *Held for the next release*. Its reward follows a separate path that the state machine does not show: u5 has not verified an email address, so the reward is held at 0.00 until verification. Keeping the content decision and the reward decision apart is deliberate: a good correction is used even when its giver is not yet eligible for a reward, and a reward is never a reason to accept a correction.

7.5 Curating: The Machine Sorts, a Human Decides

7.5.1 Why sort before a person looks

A person's attention is the scarcest resource in the pipeline. If a reviewer receives a hundred items in arrival order, the urgent safety problem may be item 87, and items 12, 40 and 71 may say the same thing in different words. Sorting fixes both problems: put the severe items first, and group the duplicates so that the reviewer reads each idea once.

Profitina uses one fast-lane AI call per item for this sorting, proposing a category, a severity and a one-line summary and grouping duplicates [5]. The lab does the same job with transparent rules instead of a model, so that every step can be traced by hand: a comment that mentions "unsafe" or "damage" becomes *unsafe advice* with *high* severity, one that says "wrong" becomes *wrong fact* with *medium* severity, one that says "missing" becomes *missing info*, and anything else is *other*. Exercise 7.6 asks you to replace the rules with a call to the fast lane from Chapter 6, keeping the rules as the floor underneath.

7.5.2 Finding duplicates with word overlap

Two suggestions are near-duplicates if they share most of their words. A simple, classic measure is the *Jaccard similarity* of their word sets. If A and B are the sets of distinct words in two texts, then

$$J(A, B) = \frac{|A \cap B|}{|A \cup B|}, \tag{7.1}$$

the number of words they share divided by the number of distinct words in either. $J = 1$ means the same words, $J = 0$ means none in common. The lab first lowercases the text, removes commas inside numbers so that "Rp12,000" becomes the single word "rp12000", and drops very common words such as "the", "is" and "we", which carry little meaning.

> **EXAMPLE.** The lab's items F01 and F02 both correct the express-service answer. After cleaning, F01 has the words {express, service, ready, 6, hours, costs, rp12000, per, kilogram} (9 words) and F02 has {express, ready, 6, hours, costing, rp12000, per, kilogram} (8 words). They share 7 words (express, ready, 6, hours, rp12000, per, kilogram), and together they have $9 + 8 - 7 = 10$ distinct words. By Equation (7.1), $J = 7/10 = 0.7$. The lab treats $J \geq 0.7$ between two items about the same answer as a duplicate, so F02, which arrived later, is marked "duplicate of F01". The threshold 0.7 is this lab's own choice.

Automatic duplicate detection saves the reviewer time, but it is also a place where the machine can be wrong in a costly way, as the lab's item F11 will show: an item that looks like an earlier one may add the one detail that matters. That is why the lab keeps its threshold high, reports every automatic decision with its reason, and why Exercise 7.4 asks you to explore what happens when the threshold is lowered.

7.5.3 The human decision

After sorting, a reviewer reads each remaining item and writes a decision and a reason: *accept* ("matches the price list"), or *reject* ("contradicts the price list (Rp7,000)"). In the lab these decisions live in a small file, `decisions.tsv`, which stands for the reviewer's work in an admin screen. An item with no decision stays "awaiting a human decision"; it does not enter the dataset by default. This is the conservative choice: silence is not consent.

7.5.4 Ordering the queue

With a category and a severity on every item, the review queue can be ordered so that a reviewer with only ten minutes sees the most important items first. A simple and transparent order sorts by severity first (high, then medium, then low), then by the size of the duplicate group (an idea reported five times matters more than one reported once), and finally by arrival time (older first, so nothing waits forever). Writing the order as a *sort key*, the tuple

$$\kappa(i) = (-\mathrm{sev}(i),\ -\mathrm{group}(i),\ t_i), \tag{7.2}$$

where sev is 3 for high, 2 for medium and 1 for low, $\mathrm{group}(i)$ is the number of items grouped with i, and t_i is its arrival time, sorting items by increasing κ in Equation (7.2) produces that order, because tuples are compared element by element and the minus signs turn "larger first" into "smaller first". In the lab, F04 (unsafe advice, high) would come first, then the two *wrong fact* items that remain after duplicates are grouped, with F01 ahead of F05 because its group includes the duplicate F02. Exercise 7.11 applies this ordering to the lab's queue. The same pattern, a priority queue ordered by a tuple, drives the re-scan scheduler of Chapter 9.

7.6 The Dataset: Preference Pairs in JSON Lines

7.6.1 What a training example looks like

A useful shape for feedback about an answer is a *preference pair*: the prompt, the answer the model gave (the *rejected* answer), and the answer a human preferred (the *chosen* answer). Preference pairs are exactly what RLHF collects [1], and they are the input of *direct preference optimization* (DPO), a simpler method that tunes a model directly from such pairs without training a separate reward model [8]. Even without any training, a set of preference pairs is valuable: it is a precise, reviewable record of what the product used to say and what people agreed it should say.

7.6.2 JSON Lines

Datasets of this kind are usually stored in the *JSON Lines* format, also called JSONL: a text file with one complete JSON object per line [9]. It is the format Ollama uses for streaming (Chapter 6), and for the same reasons: it can be written one record at a time, read one record at a time, appended to without rewriting the file, and split at any line boundary. Each line is an ordinary JSON value as defined by RFC 8259 [10], encoded in UTF-8, with no line breaks inside it.

DEFINITION. A *JSON Lines* (JSONL) file is a UTF-8 text file in which every line is one complete JSON value, usually an object, and lines are separated by `\n`. A dataset of preference pairs in JSONL has one pair per line.

7.6.3 What must not go into a dataset

A training dataset is long-lived and widely copied, so it must be clean. It should hold the content of the correction, not the person: no email address, no real name, no phone number, and no free text that identifies someone. The lab's dataset rows contain only the item identifier, the prompt, the rejected and chosen answers, and the category. Credit to givers is a separate matter handled with pseudonyms and the giver's opt-in (Section 7.3). Chapter 11 treats the privacy rules in full; here the rule of thumb is that the dataset should be publishable without harming anyone.

7.6.4 Versioning and provenance

A dataset changes every week, so every use of it must say *which* version was used. Three habits make that possible. First, the dataset is *append-only*: accepted items get the next number and are never renumbered, so "In dataset #212" means the same thing forever. Second, each release records the range of item numbers it was trained or measured with, for example "v1.4 used items #1–#240". Third, each item keeps its *provenance*: the feedback item it came from (the lab's `id` field), the decision and its reason, and the date. With these three habits, a question such as "why does the assistant now say 6 hours?" has a precise answer: item #1, from feedback F01, accepted because it matches the price list, first shipped in the release that included it.

Provenance is also what makes the public "You changed this" page honest. The page lists pseudonyms next to the changes their feedback caused, and only for givers who opted in [5]. Because every dataset row points back to a feedback item, and every feedback item to a giver, the credit is computed from the record, not remembered by hand; and because the dataset itself holds no personal data, the credit can be withdrawn without touching the training data.

7.7 Train and Measure: Before and After

7.7.1 A held-out set

How do we know that the dataset made the model better? We ask it questions and check the answers, before and after. The questions must not be the same as the training prompts; otherwise we are only checking that the model memorized them. A set of questions kept apart for measurement is called a *held-out* set. In the lab, the "model" is a small answer bank that finds the closest known prompt by word overlap, and the held-out set is eight paraphrased customer questions such as "What hours are you open?" or "Is it safe to send a silk dress?". Each question has words its answer *must* contain (for example "07:00" and "21:00") and words it *must not* contain (for example "2 days" for express service).

7.7.2 Pass rate and regressions

Let E be the held-out set, and let $\text{pass}(e) = 1$ if the answer to question e satisfies its checks and 0 otherwise. The *pass rate* is

$$p = \frac{1}{|E|} \sum_{e \in E} \text{pass}\,(e). \tag{7.3}$$

Comparing p_{before} and p_{after} tells us whether the candidate is better *on average*. That is not enough. We also want to know which questions got *worse*. The set of *regressions* is

$$R = \{\, e \in E : \text{pass}_{\text{before}}(e) = 1 \text{ and } \text{pass}_{\text{after}}(e) = 0 \,\}. \tag{7.4}$$

A candidate can improve the pass rate and still have regressions: it fixes six things and breaks one. Whether that trade is acceptable is a judgment, which is why the pipeline reports R explicitly and leaves the decision to a person.

Figure 7.3 shows the lab's result, taken from the real run in Section 7.10: the pass rate rises from 2 of 8 to 7 of 8, and one question regresses.

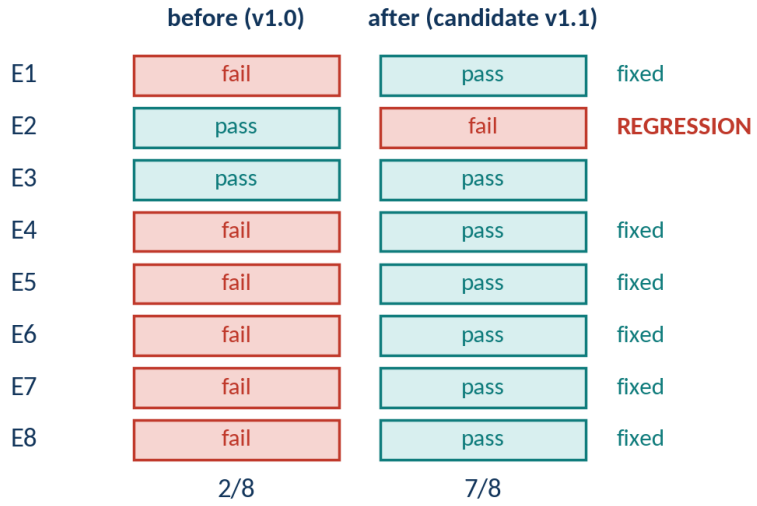


Figure 7.3 The lab's held-out questions before and after the dataset. Six questions are fixed, one keeps passing, and E2 regresses: the pass rate rises from 2/8 to 7/8, and the pipeline still advises a hold.

7.7.3 Measuring honestly

Three cautions keep measurement honest.

- **Small sets are coarse.** With eight questions, each one moves p by $1/8 = 12.5$ percentage points, just as Chapter 6's six benchmark cases moved the gate's win rate by 16.7 points. A real evaluation set should grow with every accepted item.
- **A measure that becomes a target stops measuring.** This observation, usually called Goodhart's law in Strathern's phrasing, "when a measure becomes a target, it ceases to be a good measure" [11], warns against tuning the model until it passes the held-out questions: at that point they are no longer held out. Keep adding fresh questions, and never train on the evaluation set.
- **Feedback loops can feed themselves.** A model trained on data shaped by its own earlier outputs can drift in ways no single change reveals; this is one of the classic forms of hidden technical debt in machine-learning systems [12]. Recording where every dataset item came from, as the lab's `id` field does, is the first defense.

7.7.4 How precise is a pass rate?

A pass rate measured on n questions is an estimate of how the candidate would do on all the questions users might ask. A standard rough measure of its uncertainty is the *standard error* of a proportion,

$$\mathrm{SE}(p) = \sqrt{\frac{p\,(1-p)}{n}}. \tag{7.5}$$

> **EXAMPLE.** For the lab's candidate, $p = 7/8 = 0.875$ and $n = 8$, so $\mathrm{SE} = \sqrt{0.875 \times 0.125/8} = \sqrt{0.01367} \approx 0.117$, about 12 percentage points. A rough interval of two standard errors, 0.875 ± 0.23, runs from about 64% to beyond 100%: eight questions say "clearly better than 25%" but cannot distinguish 70% from 95%. With $n = 100$ questions and the same p, $\mathrm{SE} = \sqrt{0.875 \times 0.125/100} \approx 0.033$, about 3 points, which is precise enough to compare two candidates that differ by 10 points.

Equation (7.5) is only a rough guide (for very small n or for p near 0 or 1, better intervals exist), but it gives the right instinct: grow the held-out set as the dataset grows, and distrust differences smaller than about two standard errors. It also explains why the pipeline reports each question's result and not only the percentage: with few questions, the individual results carry more information than the average.

7.8 Release: The Pipeline Advises, a Human Decides

The last step turns measurements into a decision. The lab's pipeline gives advice by a simple rule:

$$\text{advice} = \begin{cases} \text{release} & \text{if } p_{\text{after}} > p_{\text{before}} \text{ and } R = \emptyset, \\ \text{hold} & \text{otherwise.} \end{cases} \tag{7.6}$$

Equation (7.6) is deliberately cautious: any regression means "hold". But it is only advice. A person may approve a release despite a regression, for example when the regressed answer is minor and the fixed one is a safety problem, or hold a release despite a clean result, for example because a holiday week is a bad time to change customer-facing answers. Either way, the decision and its reason are recorded, and the givers read the outcome: "Shipped in v1.1" or "Not taken in v1.1", with the reason.

This is the same principle that governs Profitina's own software releases (Chapter 9): the release gate runs every automated test and reports success only when the new version answers the health check, but a person starts the release. Machines are excellent at checking; people are responsible for deciding.

What should the release decision look like to the people who gave feedback? A good release note is short, concrete and in their language. For the lab's second run it could read: "v1.1 was not released. It fixed six answers (opening hours, express time and price, pickup distance, silk and wool care, QRIS) but made one answer worse: it no longer says that the shop is open every day. We are waiting for a correction that keeps both (F11). Your accepted feedback stays in the dataset and will be part of the next attempt." Nothing in that note requires the reader to know what a pass rate is, yet it states the measurement, the regression, the decision and the next step. Writing such notes is part of the human decision, not an afterthought, because it is how the people who helped learn that their help was taken seriously.

7.9 PROFIT for Improving AI Responses

7.9.1 Why reward feedback at all

Feedback takes effort, and most platforms reward it with nothing but a thank-you. Profitina's founding idea is that useful contributions, including feedback that improves the local model, deserve a reward with some substance. That reward is PROFIT, the platform's utility coin: an SPL token on the Solana network, with 6 decimal places, whose contract is live on Solana *devnet* as of September 2026 [5], [13]. Chapter 12 explains PROFIT in full. For this chapter, four facts matter, and they must be stated plainly:

- PROFIT is described by Profitina as a utility coin, "not a security, not electronic money, not a promise of returns", and on devnet it has no real monetary value [5].
- Selling or redeeming PROFIT for money is disabled until mainnet and regulatory clarity; mainnet is a roadmap item, gated by licensing and audit [5].
- What is live: earning small amounts for useful activity, spending PROFIT on services inside the platform, and transfers between users [5].
- Nothing in this chapter is investment advice, and no price or value is promised.

7.9.2 The public earning rules

Table 7.2 reproduces the public earning table from the September 2026 whitepaper [5]. Amounts are in PROFIT; the daily cap is the number of rewarded actions per day.

Table 7.2 Public PROFIT earning rules (whitepaper v24, September 2026). Amounts are further multiplied by the decaying emission multiplier and by an AI usefulness judgment.

Action	Reward (PROFIT)	Daily cap
Feedback on a scan	0.01–0.05	10
Post	0.02	1
Reply received	0.01	2
Repost received	0.01	1
Like received	0.01	5
Create a channel	0.05	2
Referral (both sides, once per pair)	0.25	50

Around the table sit the rules that keep it fair [5]:

- **Small amounts.** The base unit is 0.01 PROFIT. Rewards are a thank-you, not a wage.
- **Daily caps that reset at 00:00 WIB.** WIB is *Waktu Indonesia Barat*, Western Indonesia Time, UTC+7, the time zone of Jakarta and Surabaya. Caps stop any one account from farming rewards by repetition.
- **Only verified humans earn.** An account earns only after its email address is verified, the first anti-bot layer.
- **Usefulness, not volume.** A local-AI usefulness judgment gives trivial or bot-like contributions 0.00; rewards on suspicious activity are held in escrow and can be clawed back and burned.
- **Duplicates earn nothing.** Duplicate content earns 0.00 and links to the earlier item.
- **Decaying emission.** Every amount is multiplied by an emission multiplier that halves each time the number of human users grows tenfold, so rewards shrink as the community grows (Chapter 12).

The whitepaper publishes these rules and the results, not the internal thresholds that implement them, and this book follows the same line.

7.9.3 The arithmetic of a capped reward

The lab implements a reward rule of the same shape with its own values. For item i, let B be the base reward, A the bonus for an item a human accepted into the dataset, $\text{acc}_i = 1$ if item i was accepted and 0 otherwise, and $\text{elig}_i = 1$ if the item is eligible (verified giver, not too short, not a duplicate, and under the daily cap). Then

$$r_i = (B + A \cdot \text{acc}_i) \cdot \text{elig}_i. \tag{7.7}$$

The lab uses $B = 0.01$ and $A = 0.04$, so an accepted item earns 0.05, matching the public range of 0.01–0.05 for scan feedback, and it uses a lab cap of $K = 3$ rewarded items per user per day, smaller than the public cap of 10, so that the cap is visible in a short example. For every user u and every WIB day d,

$$\sum_{i \in \text{items}(u,d)} \text{rewarded}_i \le K. \tag{7.8}$$

Two engineering details deserve their own attention.

Integers, not floating-point numbers. PROFIT has 6 decimals, so the smallest unit is one millionth of a PROFIT, a *micro-PROFIT* (µP). The lab stores every amount as an integer number of micro-units,

$$\text{amount}_{\mu\text{P}} = \text{amount}_{\text{PROFIT}} \times 10^6, \tag{7.9}$$

so 0.01 PROFIT is stored as 10,000 and 0.05 as 50,000. Integer arithmetic is exact; floating-point arithmetic is not ($0.1 + 0.2$ is not exactly 0.3 in binary floating point), and a ledger must add up exactly. The same idea underlies the token's own `decimals` field on Solana [13].

Which day is "today"? A daily cap needs a definition of the day. Servers usually keep time in UTC, but Profitina's users live in WIB, and the cap resets at midnight WIB. The WIB date of an instant t is the UTC date of t shifted by seven hours:

$$\text{day}_{\text{WIB}}(t) = \text{date}(t_{\text{UTC}} + 7\,\text{h}). \tag{7.10}$$

A program that keys the cap on the UTC date instead resets it at 07:00 WIB, in the middle of the morning, which is a real bug that Profitina's team found and fixed in its own quota code. Because Indonesia does not use daylight saving time, a fixed offset of seven hours is exact all year. Indonesia has three time zones, WIB (UTC+7), WITA (UTC+8) and WIT (UTC+9), so a user in Makassar or Jayapura lives one or two hours ahead of Jakarta. A platform must still pick one definition of "today" for a shared allowance, and Profitina publishes its choice: all daily quotas and caps reset at 00:00 WIB. Publishing the rule matters as much as choosing it, because a user in Jayapura who knows that the reset happens at 02:00 local time is not surprised by it. **Figure 7.4** shows the lab's user u1 on 24–25 September 2026: three rewarded items, a fourth at 23:30 WIB that hits the cap, and a fifth at 00:10 WIB that is rewarded again because a new WIB day has begun.

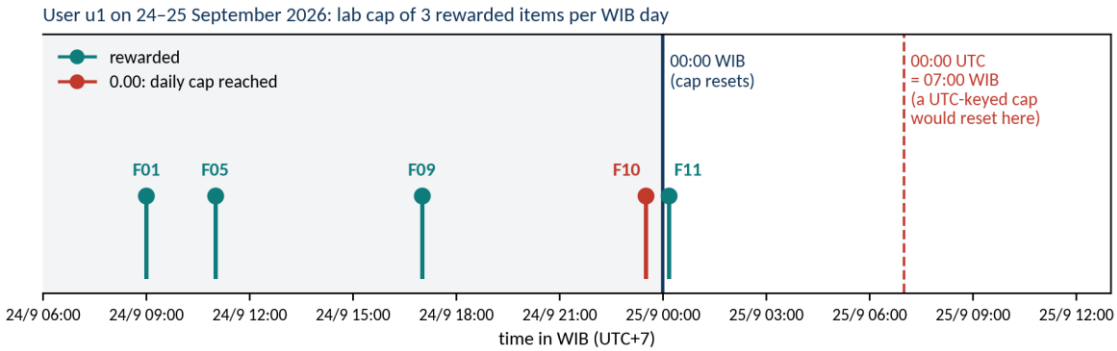


Figure 7.4 A lab daily cap of 3 rewarded items per WIB day for user u1. F10 at 23:30 WIB hits the cap; F11 at 00:10 WIB falls on a new WIB day. A cap keyed on the UTC date would reset only at 07:00 WIB.

7.9.4 Bounding what one account can earn

Caps do more than keep rewards small; they make the worst case *computable*. If an action j pays at most v_j and may be rewarded at most c_j times a day, then no single account can earn more in a day than

$$M_{\text{day}} = \sum_j v_j \; c_j, \tag{7.11}$$

before the emission multiplier and the usefulness judgment, both of which can only reduce the amount.

> **EXAMPLE.** Apply Equation (7.11) to the daily actions of **Table 7.2**, leaving out the one-time referral: scan feedback $0.05 \times 10 = 0.50$, a post $0.02 \times 1 = 0.02$, replies received $0.01 \times 2 = 0.02$, a repost received $0.01 \times 1 = 0.01$, likes received $0.01 \times 5 = 0.05$, and channels created $0.05 \times 2 = 0.10$. The sum is $M_{\text{day}} = 0.70$ PROFIT, and in practice far less, because the maximum amounts require accepted, useful contributions, and the emission multiplier falls as the community grows.

A bound per account is not yet a bound per person, since one person could create many accounts; this is called a *Sybil attack*. That is why the per-account caps are combined with email verification, duplicate detection across accounts, usefulness judgments, and escrow with clawback for suspicious patterns. None of these defenses is perfect alone; together, and with Equation (7.11) keeping each account's worst case small, they make farming unprofitable. Chapter 5 describes the Fair-Play side of the same defenses.

7.10 Lab: A Mini Feedback Pipeline

7.10.1 The scenario and the data

Profitina Laundry has a small help assistant that answers customer questions. For this lab its "model" is deliberately simple: an *answer bank* of six known prompts, B1 to B6, each with an answer, and a rule that answers any question with the answer of the closest known prompt by word overlap (Equation (7.1)). Several of the answers are wrong or incomplete: B1 says only "We are open every day." without hours, B3 claims express service takes 2 days (it takes 6 hours), B4 says only "Yes." to a question about pickup, B5 claims silk and wool go in the machines (they must be dry-cleaned), and B6 says the laundry accepts cash only (it also accepts QRIS).

Customers have sent eleven feedback items. They are stored in `submissions.tsv`, a tab-separated file with one item per line; **Table 7.3** shows them. All users, times and texts are invented for this lab.

Table 7.3 The eleven lab feedback items in `submissions.tsv`. Times are in UTC; add seven hours for WIB.

Item	User	Verified	Time (UTC)	Answer	Suggested answer	Comment
F01	u1	yes	24 Sep 02:00	B3	Express service is ready in 6 hours and costs Rp12,000 per kilogram.	The express answer is wrong, it is 6 hours.
F02	u2	yes	24 Sep 02:40	B3	Express is ready in 6 hours, costing Rp12,000 per kilogram.	Wrong: express is not 2 days.
F03	u3	yes	24 Sep 03:10	B1	We are open from 07:00 to 21:00.	The hours are missing.
F04	u4	yes	24 Sep 03:30	B5	Silk and wool are dry-clean only; a partner cleans them in 4 days.	Unsafe advice: silk would be damaged in a machine.
F05	u1	yes	24 Sep 04:00	B6	We accept cash and QRIS (DANA, GoPay, OVO, ShopeePay, mobile banking).	Wrong, QRIS works at the counter.
F06	u5	no	24 Sep 05:00	B4	Yes, free pickup and delivery within 3 km.	Missing the distance.
F07	u6	yes	24 Sep 05:30	B4	ok	ok
F08	u2	yes	24 Sep 06:00	B2	Regular washing is Rp5,000 per kilogram.	Should be cheaper.
F09	u1	yes	24 Sep 10:00	B2	Regular washing costs Rp7,000 per kilogram and takes 2 days.	Just checking this one.
F10	u1	yes	24 Sep 16:30	B4	We offer free pickup and delivery within 3 km of the shop.	Missing the distance.
F11	u1	yes	24 Sep 17:10	B1	We are open every day from 07:00 to 21:00.	Missing hours, keep every day.

The reviewer's decisions are in `decisions.tsv`: accept F01, F03, F04, F05, F06 and F10; reject F08 ("contradicts the price list (Rp7,000)") and F09 ("same as the current answer"). The reviewer has not yet looked at F11. Notice a few traps planted in the data: F02 repeats F01, F07 says only "ok", F06 comes from an unverified account, F08 is wrong, user u1 sends five items in one day, and F11, which arrives last, quietly contains the detail that F03 forgot.

The held-out set, E1 to E8, is written into the program; each question names the words its answer must and must not contain.

7.10.2 The program

Listing 7.1 is the whole pipeline in Python, standard library only.

Listing 7.1 `feedback_pipeline.py`: submit, curate with a human decision, export a JSONL dataset, reward under a WIB daily cap, measure before and after, and wait for a human release decision.

```
"""A mini feedback pipeline: submit -> curate -> dataset -> measure -> release.

    python feedback_pipeline.py [--decisions FILE] [--release approve|hold REASON]

Reads submissions.tsv and decisions.tsv (or FILE), writes dataset.jsonl.
"""
import json
import re
import sys
from datetime import datetime, timedelta, timezone

WIB = timezone(timedelta(hours=7))  # Indonesia has no daylight saving
MICRO = 1_000_000                   # amounts are kept in integer micro-units
BASE, BONUS, DAILY_CAP = 10_000, 40_000, 3  # this lab's own values
STOP = set("a an and are can do does for from how i in is it my of on the to "
           "we what when which will with you your".split())

MODEL_V1 = {  # the "model" before feedback: prompt -> answer
    "B1": ("what are your opening hours", "We are open every day."),
    "B2": ("how much does regular washing cost per kilogram",
           "Regular washing costs Rp7,000 per kilogram and takes 2 days."),
    "B3": ("how long does express service take", "Express service takes 2 days."),
    "B4": ("do you pick up and deliver laundry", "Yes."),
    "B5": ("can you wash silk and wool clothes",
           "Yes, we wash everything in our machines."),
    "B6": ("which payment methods do you accept", "We accept cash only."),
}
EVAL = [  # held-out questions, never used for "training"
    ("E1", "What hours are you open?", ["07:00", "21:00"], []),
    ("E2", "Are your opening hours the same every day?", ["every day"], []),
    ("E3", "Price per kilo for regular washing?", ["Rp7,000"], []),
    ("E4", "How fast is the express service?", ["6 hours"], ["2 days"]),
    ("E5", "What does the express service cost?", ["Rp12,000"], []),
    ("E6", "Will you pick up laundry from my house?", ["3 km"], []),
    ("E7", "Is it safe to send a silk dress?", ["dry-clean"], ["everything"]),
    ("E8", "Can I pay with QRIS?", ["QRIS"], []),
]


def read_tsv(path):
    with open(path, encoding="utf-8") as f:
        header, *rows = [ln.rstrip("\n").split("\t") for ln in f if ln.strip()]
    return [dict(zip(header, r)) for r in rows]
```

Listing 7.1 (Continued) *feedback_pipeline.py: submit, curate with a human decision, export a JSONL dataset, reward under a WIB daily cap, measure before and after, and wait for a human release decision.*

```
def words(text):
    return set(re.findall(r"[a-z0-9:]+", text.lower().replace(",", ""))) - STOP


def jaccard(a, b):
    a, b = words(a), words(b)
    return len(a & b) / len(a | b) if a | b else 0.0


def answer(model, question):
    """Toy model: the answer of the closest known prompt (word overlap)."""
    best = max(model, key=lambda k: (jaccard(question, model[k][0]), k))
    return model[best][1]


def curate(items, decisions):
    """Sort each item, group duplicates, then apply the human decisions."""
    dataset = []
    for it in items:
        c = it["comment"].lower()
        it["category"] = ("unsafe advice" if re.search(r"unsafe|damage", c) else
                          "wrong fact" if "wrong" in c else
                          "missing info" if "missing" in c else "other")
        it["severity"] = {"unsafe advice": "high", "wrong fact": "medium"}.get(
            it["category"], "low")
        earlier = [e for e in items if e is not it and e["target"] == it["target"]
                   and e["utc_time"] < it["utc_time"]
                   and jaccard(e["suggestion"], it["suggestion"]) >= 0.7]
        if len(it["suggestion"].split()) < 4:
            it["status"] = "not taken: too short (automatic)"
        elif earlier:
            it["status"] = f"duplicate of {earlier[0]['id']} (automatic)"
        elif it["id"] not in decisions:
            it["status"] = "awaiting a human decision"
        elif decisions[it["id"]]["decision"] == "accept":
            dataset.append(it)
            it["status"] = f"in dataset #{len(dataset)}"
        else:
            it["status"] = "not taken: " + decisions[it["id"]]["reason"]
    return dataset
```

Listing 7.1 (Continued) *feedback_pipeline.py: submit, curate with a human decision, export a JSONL dataset, reward under a WIB daily cap, measure before and after, and wait for a human release decision.*

```
def rewards(items):
    """Base reward for a useful submission, bonus when a human accepts it."""
    per_day = {}
    for it in items:
        t = datetime.strptime(it["utc_time"], "%Y-%m-%dT%H:%MZ")
        local = t.replace(tzinfo=timezone.utc).astimezone(WIB)
        key = (it["user"], local.date())
        it["wib"] = local.strftime("%d %b %H:%M")
        if it["verified"] != "yes":
            it["reward"], it["note"] = 0, "held: email not verified"
        elif it["status"].startswith(("not taken: too", "duplicate")):
            it["reward"], it["note"] = 0, it["status"].split(" (")[0]
        elif per_day.get(key, 0) >= DAILY_CAP:
            it["reward"], it["note"] = 0, "daily cap reached (resets 00:00 WIB)"
        else:
            per_day[key] = per_day.get(key, 0) + 1
            bonus = BONUS if it["status"].startswith("in dataset") else 0
            it["reward"] = BASE + bonus
            it["note"] = "base + accepted bonus" if bonus else "base"


def measure(model):
    results = {}
    for eid, q, must, must_not in EVAL:
        a = answer(model, q)
        results[eid] = (all(m in a for m in must)
                        and not any(m in a for m in must_not))
    return results


def main(argv):
    items = sorted(read_tsv("submissions.tsv"), key=lambda r: r["utc_time"])
    path = argv[argv.index("--decisions") + 1] if "--decisions" in argv \
        else "decisions.tsv"
    decisions = {d["id"]: d for d in read_tsv(path)}

    print("1-2. SUBMIT and CURATE")
    dataset = curate(items, decisions)
    rewards(items)
    for it in items:
        print(f"  {it['id']} {it['user']} {it['target']} {it['category']:<13} "
              f"{it['severity']:<6} {it['status']}")
```

Listing 7.1 (Continued) *feedback_pipeline.py: submit, curate with a human decision, export a JSONL dataset, reward under a WIB daily cap, measure before and after, and wait for a human release decision.*

```
    print("PROFIT earned (lab values, integer micro-units shown as PROFIT)")
    for it in items:
        print(f"  {it['id']} {it['user']} {it['wib']} WIB  "
              f"{it['reward'] / MICRO:.2f}  {it['note']}")
    total = sum(it["reward"] for it in items)
    print(f"  total emitted: {total / MICRO:.2f} PROFIT")

    print("3. DATASET -> dataset.jsonl")
    with open("dataset.jsonl", "w", encoding="utf-8", newline="\n") as f:
        for n, it in enumerate(dataset, 1):
            row = {"n": n, "id": it["id"], "prompt": MODEL_V1[it["target"]][0],
                   "rejected": MODEL_V1[it["target"]][1],
                   "chosen": it["suggestion"], "category": it["category"]}
            f.write(json.dumps(row) + "\n")
    print(f"  {len(dataset)} preference pairs written")

    print("4. TRAIN and MEASURE (held-out questions)")
    model_v2 = dict(MODEL_V1)
    for it in dataset:  # "training" here = updating the answer bank
        model_v2[it["target"]] = (MODEL_V1[it["target"]][0], it["suggestion"])
    before, after = measure(MODEL_V1), measure(model_v2)
    for eid, *_ in EVAL:
        mark = {(False, True): "fixed", (True, False): "REGRESSION"}.get(
            (before[eid], after[eid]), "")
        print(f"  {eid}  before {'pass' if before[eid] else 'fail'}  "
              f"after {'pass' if after[eid] else 'fail'}  {mark}".rstrip())
    b, a = sum(before.values()), sum(after.values())
    regressions = [e for e in before if before[e] and not after[e]]
    print(f"  pass rate: {b}/{len(EVAL)} -> {a}/{len(EVAL)}"
          f" ({100 * b / len(EVAL):.1f}% -> {100 * a / len(EVAL):.1f}%)")

    print("5. RELEASE")
    advice = ("release" if a > b and not regressions else
              "hold (regression on " + ", ".join(regressions) + ")"
              if regressions else "hold (no improvement)")
    print(f"  the pipeline advises: {advice}")
    if "--release" not in argv:
        print("  waiting for a human: run again with --release approve"
              " or --release hold REASON")
    elif argv[argv.index("--release") + 1] == "approve":
        print("  human decision: approved -> Shipped in v1.1")
    else:
        reason = " ".join(argv[argv.index("--release") + 2:]) or "no reason"
        print(f"  human decision: Not taken in v1.1 ({reason})")


if __name__ == "__main__":
    main(sys.argv[1:])
```

Walk through it in the order of the five steps.

- **Constants.** `WIB` is a fixed UTC+7 time zone; Python's `zoneinfo.ZoneInfo("Asia/Jakarta")` would also work, but on Windows it needs the extra `tzdata` package, and the fixed offset is exact for Indonesia. `MICRO`, `BASE`, `BONUS` and `DAILY_CAP` implement Equation (7.9), Equation (7.7) and Equation (7.8) with this lab's values. `STOP` holds the very common words that `words` removes.

- **The toy model.** `MODEL_V1` maps B1–B6 to a prompt and an answer. `answer` picks the known prompt with the highest Jaccard similarity to the question; ties go to the larger key so that the result never depends on dictionary order.
- **Submit.** `read_tsv` reads a tab-separated file into a list of dictionaries, one per line, using the first line as the header. `main` sorts the items by time, because the order of arrival matters for duplicates and for the cap.
- **Curate.** `curate` gives each item a category and a severity from its comment, then checks, in order: too short (fewer than 4 words, automatic), a near-duplicate of an earlier item about the same answer ($J \geq 0.7$, automatic), no human decision yet, accepted (the item joins the dataset and gets its number), or rejected (with the reviewer's reason).
- **Rewards.** `rewards` converts each UTC time to WIB with `astimezone`, counts rewarded items per user and WIB date, and applies Equation (7.7) and Equation (7.8). Unverified accounts are held at 0.00, and so are duplicates and too-short items, each with its reason.
- **Dataset.** Each accepted item becomes one JSON object on one line of `dataset.jsonl`, with `newline="\n"` so that the file is identical on Windows, macOS and Linux.
- **Train and measure.** "Training" here is honest about being a toy: it replaces the answer of each corrected prompt with the chosen answer. `measure` applies Equation (7.3) to both versions, and the loop prints each question's result, marking fixes and regressions as in Equation (7.4).
- **Release.** The advice follows Equation (7.6). Without `--release` the program stops and waits for a person; with `--release approve` or `--release hold REASON` it records the human decision.

Listing 7.2 is the same pipeline in Java.

Listing 7.2 `FeedbackPipeline.java`: the same pipeline in Java, with records, `java.time` for WIB, integer micro-units, and a hand-written JSON Lines writer.

```java
import java.io.IOException;
import java.io.PrintWriter;
import java.nio.charset.StandardCharsets;
import java.nio.file.Files;
import java.nio.file.Path;
import java.time.LocalDateTime;
import java.time.ZoneOffset;
import java.time.format.DateTimeFormatter;
import java.util.ArrayList;
import java.util.Arrays;
import java.util.Comparator;
import java.util.HashMap;
import java.util.HashSet;
import java.util.LinkedHashMap;
import java.util.List;
import java.util.Locale;
import java.util.Map;
import java.util.Set;
import java.util.regex.Matcher;
import java.util.regex.Pattern;

/** A mini feedback pipeline: submit -> curate -> dataset -> measure -> release.
 *  java FeedbackPipeline [--decisions FILE] [--release approve|hold REASON] */
public class FeedbackPipeline {
    static final ZoneOffset WIB = ZoneOffset.ofHours(7);   // no daylight saving
    static final long MICRO = 1_000_000;                    // integer micro-units
    static final long BASE = 10_000, BONUS = 40_000;        // this lab's own values
    static final int DAILY_CAP = 3;
    static final Set<String> STOP = Set.of(("a an and are can do does for from"
            + " how i in is it my of on the to we what when which will with you"
            + " your").split(" "));

    record Q(String prompt, String answer) {}
    record Eval(String id, String question, List<String> must,
                List<String> mustNot) {}
```

Listing 7.2 (Continued) *FeedbackPipeline.java: the same pipeline in Java, with records, java.time for WIB, integer micro-units, and a hand-written JSON Lines writer.*

```
    static final Map<String, Q> MODEL_V1 = new LinkedHashMap<>();
    static {                                  // the "model" before feedback
        MODEL_V1.put("B1", new Q("what are your opening hours",
                "We are open every day."));
        MODEL_V1.put("B2", new Q("how much does regular washing cost per kilogram",
                "Regular washing costs Rp7,000 per kilogram and takes 2 days."));
        MODEL_V1.put("B3", new Q("how long does express service take",
                "Express service takes 2 days."));
        MODEL_V1.put("B4", new Q("do you pick up and deliver laundry", "Yes."));
        MODEL_V1.put("B5", new Q("can you wash silk and wool clothes",
                "Yes, we wash everything in our machines."));
        MODEL_V1.put("B6", new Q("which payment methods do you accept",
                "We accept cash only."));
    }

    static final List<Eval> EVAL = List.of(    // held-out, never "trained" on
        new Eval("E1", "What hours are you open?", List.of("07:00", "21:00"),
                 List.of()),
        new Eval("E2", "Are your opening hours the same every day?",
                 List.of("every day"), List.of()),
        new Eval("E3", "Price per kilo for regular washing?", List.of("Rp7,000"),
                 List.of()),
        new Eval("E4", "How fast is the express service?", List.of("6 hours"),
                 List.of("2 days")),
        new Eval("E5", "What does the express service cost?", List.of("Rp12,000"),
                 List.of()),
        new Eval("E6", "Will you pick up laundry from my house?", List.of("3 km"),
                 List.of()),
        new Eval("E7", "Is it safe to send a silk dress?", List.of("dry-clean"),
                 List.of("everything")),
        new Eval("E8", "Can I pay with QRIS?", List.of("QRIS"), List.of()));

    static List<Map<String, String>> readTsv(String path) throws IOException {
        List<String> lines = new ArrayList<>();
        for (String l : Files.readAllLines(Path.of(path), StandardCharsets.UTF_8)) {
            if (!l.isBlank()) lines.add(l);
        }
        String[] header = lines.get(0).split("\t");
        List<Map<String, String>> rows = new ArrayList<>();
        for (String l : lines.subList(1, lines.size())) {
            String[] cells = l.split("\t", -1);
            Map<String, String> row = new HashMap<>();
            for (int k = 0; k < header.length; k++) row.put(header[k], cells[k]);
            rows.add(row);
        }
        return rows;
```

Listing 7.2 (Continued) *FeedbackPipeline.java: the same pipeline in Java, with records, java.time for WIB, integer micro-units, and a hand-written JSON Lines writer.*

```
    }

    static Set<String> words(String text) {
        Set<String> out = new HashSet<>();
        Matcher m = Pattern.compile("[a-z0-9:]+")
                .matcher(text.toLowerCase().replace(",", ""));
        while (m.find()) out.add(m.group());
        out.removeAll(STOP);
        return out;
    }

    static double jaccard(String a, String b) {
        Set<String> x = words(a), y = words(b);
        Set<String> union = new HashSet<>(x);
        union.addAll(y);
        if (union.isEmpty()) return 0.0;
        x.retainAll(y);
        return (double) x.size() / union.size();
    }

    /** Toy model: the answer of the closest known prompt (word overlap). */
    static String answer(Map<String, Q> model, String question) {
        String best = null;
        double bestScore = -1;
        for (String k : model.keySet()) {      // ties go to the larger key
            double s = jaccard(question, model.get(k).prompt());
            if (s > bestScore || (s == bestScore && k.compareTo(best) > 0)) {
                best = k;
                bestScore = s;
            }
        }
        return model.get(best).answer();
    }

    static List<Map<String, String>> curate(List<Map<String, String>> items,
            Map<String, Map<String, String>> decisions) {
        List<Map<String, String>> dataset = new ArrayList<>();
        for (Map<String, String> it : items) {
            String c = it.get("comment").toLowerCase();
            String cat = Pattern.compile("unsafe|damage").matcher(c).find()
                    ? "unsafe advice" : c.contains("wrong") ? "wrong fact"
                    : c.contains("missing") ? "missing info" : "other";
            it.put("category", cat);
            it.put("severity", cat.equals("unsafe advice") ? "high"
                    : cat.equals("wrong fact") ? "medium" : "low");
            Map<String, String> earlier = null;
```

Listing 7.2 (Continued) *FeedbackPipeline.java: the same pipeline in Java, with records, java.time for WIB, integer micro-units, and a hand-written JSON Lines writer.*

```
            for (Map<String, String> e : items) {
                if (e != it && e.get("target").equals(it.get("target"))
                        && e.get("utc_time").compareTo(it.get("utc_time")) < 0
                        && jaccard(e.get("suggestion"),
                                   it.get("suggestion")) >= 0.7) {
                    earlier = e;
                    break;
                }
            }
            Map<String, String> d = decisions.get(it.get("id"));
            if (it.get("suggestion").strip().split("\\s+").length < 4) {
                it.put("status", "not taken: too short (automatic)");
            } else if (earlier != null) {
                it.put("status",
                       "duplicate of " + earlier.get("id") + " (automatic)");
            } else if (d == null) {
                it.put("status", "awaiting a human decision");
            } else if (d.get("decision").equals("accept")) {
                dataset.add(it);
                it.put("status", "in dataset #" + dataset.size());
            } else {
                it.put("status", "not taken: " + d.get("reason"));
            }
        }
        return dataset;
    }

    static void rewards(List<Map<String, String>> items) {
        Map<String, Integer> perDay = new HashMap<>();
        DateTimeFormatter in = DateTimeFormatter.ofPattern("yyyy-MM-dd'T'HH:mm'Z'");
        DateTimeFormatter out =
                DateTimeFormatter.ofPattern("dd MMM HH:mm", Locale.ENGLISH);
        for (Map<String, String> it : items) {
            var local = LocalDateTime.parse(it.get("utc_time"), in)
                    .atOffset(ZoneOffset.UTC).withOffsetSameInstant(WIB);
            String key = it.get("user") + "|" + local.toLocalDate();
            it.put("wib", local.format(out));
            String st = it.get("status");
            long reward = 0;
            String note;
            if (!it.get("verified").equals("yes")) {
                note = "held: email not verified";
            } else if (st.startsWith("not taken: too")
                       || st.startsWith("duplicate")) {
                note = st.split(" \\(")[0];
            } else if (perDay.getOrDefault(key, 0) >= DAILY_CAP) {
```

Listing 7.2 (Continued) *FeedbackPipeline.java: the same pipeline in Java, with records, java.time for WIB, integer micro-units, and a hand-written JSON Lines writer.*

```
                note = "daily cap reached (resets 00:00 WIB)";
            } else {
                perDay.merge(key, 1, Integer::sum);
                long bonus = st.startsWith("in dataset") ? BONUS : 0;
                reward = BASE + bonus;
                note = bonus > 0 ? "base + accepted bonus" : "base";
            }
            it.put("reward", Long.toString(reward));
            it.put("note", note);
        }
    }

    static Map<String, Boolean> measure(Map<String, Q> model) {
        Map<String, Boolean> results = new LinkedHashMap<>();
        for (Eval e : EVAL) {
            String a = answer(model, e.question());
            boolean ok = e.must().stream().allMatch(a::contains)
                    && e.mustNot().stream().noneMatch(a::contains);
            results.put(e.id(), ok);
        }
        return results;
    }

    static String quote(String s) {            // JSON string, ASCII text only
        return "\"" + s.replace("\\", "\\\\").replace("\"", "\\\"") + "\"";
    }

    static String pct(long part, long whole) {
        return String.format(Locale.ROOT, "%.1f%%", 100.0 * part / whole);
    }

    public static void main(String[] args) throws IOException {
        List<String> argv = Arrays.asList(args);
        List<Map<String, String>> items = readTsv("submissions.tsv");
        items.sort(Comparator.comparing(r -> r.get("utc_time")));
        String path = argv.contains("--decisions")
                ? argv.get(argv.indexOf("--decisions") + 1) : "decisions.tsv";
        Map<String, Map<String, String>> decisions = new HashMap<>();
        for (Map<String, String> d : readTsv(path)) decisions.put(d.get("id"), d);
```

Listing 7.2 (Continued) *FeedbackPipeline.java: the same pipeline in Java, with records, java.time for WIB, integer micro-units, and a hand-written JSON Lines writer.*

```
        System.out.println("1-2. SUBMIT and CURATE");
        List<Map<String, String>> dataset = curate(items, decisions);
        rewards(items);
        for (Map<String, String> it : items) {
            System.out.printf("  %s %s %s %-13s %-6s %s%n", it.get("id"),
                    it.get("user"), it.get("target"), it.get("category"),
                    it.get("severity"), it.get("status"));
        }

        System.out.println("PROFIT earned (lab values, integer micro-units shown"
                           + " as PROFIT)");
        long total = 0;
        for (Map<String, String> it : items) {
            long r = Long.parseLong(it.get("reward"));
            total += r;
            System.out.printf(Locale.ROOT, "  %s %s %s WIB  %.2f  %s%n",
                    it.get("id"), it.get("user"), it.get("wib"),
                    (double) r / MICRO, it.get("note"));
        }
        System.out.printf(Locale.ROOT, "  total emitted: %.2f PROFIT%n",
                          (double) total / MICRO);

        System.out.println("3. DATASET -> dataset.jsonl");
        try (PrintWriter w = new PrintWriter(Files.newBufferedWriter(
                Path.of("dataset.jsonl"), StandardCharsets.UTF_8))) {
            int n = 0;
            for (Map<String, String> it : dataset) {
                Q q = MODEL_V1.get(it.get("target"));
                w.print("{\"n\": " + (++n) + ", \"id\": " + quote(it.get("id"))
                        + ", \"prompt\": " + quote(q.prompt())
                        + ", \"rejected\": " + quote(q.answer())
                        + ", \"chosen\": " + quote(it.get("suggestion"))
                        + ", \"category\": " + quote(it.get("category")) + "}\n");
            }
        }
        System.out.println("  " + dataset.size() + " preference pairs written");
```

Listing 7.2 (Continued) *FeedbackPipeline.java: the same pipeline in Java, with records, java.time for WIB, integer micro-units, and a hand-written JSON Lines writer.*

```
        System.out.println("4. TRAIN and MEASURE (held-out questions)");
        Map<String, Q> modelV2 = new LinkedHashMap<>(MODEL_V1);
        for (Map<String, String> it : dataset) {  // "training" = update the bank
            String t = it.get("target");
            modelV2.put(t, new Q(MODEL_V1.get(t).prompt(), it.get("suggestion")));
        }
        Map<String, Boolean> before = measure(MODEL_V1), after = measure(modelV2);
        List<String> regressions = new ArrayList<>();
        int b = 0, a = 0;
        for (Eval e : EVAL) {
            boolean x = before.get(e.id()), y = after.get(e.id());
            if (x) b++;
            if (y) a++;
            if (x && !y) regressions.add(e.id());
            String mark = !x && y ? "fixed" : x && !y ? "REGRESSION" : "";
            String line = "  " + e.id() + "  before " + (x ? "pass" : "fail")
                    + "  after " + (y ? "pass" : "fail") + "  " + mark;
            System.out.println(line.stripTrailing());
        }
        int n = EVAL.size();
        System.out.println("  pass rate: " + b + "/" + n + " -> " + a + "/" + n
                           + " (" + pct(b, n) + " -> " + pct(a, n) + ")");

        System.out.println("5. RELEASE");
        String advice = a > b && regressions.isEmpty() ? "release"
                : !regressions.isEmpty()
                        ? "hold (regression on " + String.join(", ", regressions)
                          + ")"
                        : "hold (no improvement)";
        System.out.println("  the pipeline advises: " + advice);
        int r = argv.indexOf("--release");
        if (r < 0) {
            System.out.println("  waiting for a human: run again with --release"
                               + " approve or --release hold REASON");
        } else if (argv.get(r + 1).equals("approve")) {
            System.out.println("  human decision: approved -> Shipped in v1.1");
        } else {
            String reason = String.join(" ", argv.subList(r + 2, argv.size()));
            System.out.println("  human decision: Not taken in v1.1 ("
                               + (reason.isEmpty() ? "no reason" : reason) + ")");
        }
    }
}
```

The Java version mirrors the Python one step for step. `ZoneOffset.ofHours(7)` plays the part of the fixed WIB zone, `LocalDateTime.parse(...).atOffset(ZoneOffset.UTC).withOffsetSameInstant(WIB)` converts a UTC time to WIB, and `Locale.ROOT` and `Locale.ENGLISH` keep numbers and month names identical on every computer. JSON is written by hand with a small `quote` method, which is enough because the lab's text is plain ASCII; a production system would use a JSON library.

7.10.3 Running the pipeline

Open a terminal in `Chapter07/python` and run the program, or in `Chapter07/java` compile and run the Java version:

```
python feedback_pipeline.py
javac FeedbackPipeline.java
java FeedbackPipeline
```

Both print exactly the following; we compared the two outputs, and the two `dataset.jsonl` files, with `diff`.

```
1-2. SUBMIT and CURATE
  F01 u1 B3 wrong fact    medium in dataset #1
  F02 u2 B3 wrong fact    medium duplicate of F01 (automatic)
  F03 u3 B1 missing info  low    in dataset #2
  F04 u4 B5 unsafe advice high   in dataset #3
  F05 u1 B6 wrong fact    medium in dataset #4
  F06 u5 B4 missing info  low    in dataset #5
  F07 u6 B4 other         low    not taken: too short (automatic)
  F08 u2 B2 other         low    not taken: contradicts the price list (Rp7,000)
  F09 u1 B2 other         low    not taken: same as the current answer
  F10 u1 B4 missing info  low    in dataset #6
  F11 u1 B1 missing info  low    awaiting a human decision
PROFIT earned (lab values, integer micro-units shown as PROFIT)
  F01 u1 24 Sep 09:00 WIB  0.05  base + accepted bonus
  F02 u2 24 Sep 09:40 WIB  0.00  duplicate of F01
  F03 u3 24 Sep 10:10 WIB  0.05  base + accepted bonus
  F04 u4 24 Sep 10:30 WIB  0.05  base + accepted bonus
  F05 u1 24 Sep 11:00 WIB  0.05  base + accepted bonus
  F06 u5 24 Sep 12:00 WIB  0.00  held: email not verified
  F07 u6 24 Sep 12:30 WIB  0.00  not taken: too short
  F08 u2 24 Sep 13:00 WIB  0.01  base
  F09 u1 24 Sep 17:00 WIB  0.01  base
  F10 u1 24 Sep 23:30 WIB  0.00  daily cap reached (resets 00:00 WIB)
  F11 u1 25 Sep 00:10 WIB  0.01  base
  total emitted: 0.23 PROFIT
3. DATASET -> dataset.jsonl
  6 preference pairs written
4. TRAIN and MEASURE (held-out questions)
  E1  before fail  after pass  fixed
  E2  before pass  after fail  REGRESSION
  E3  before pass  after pass
  E4  before fail  after pass  fixed
  E5  before fail  after pass  fixed
  E6  before fail  after pass  fixed
  E7  before fail  after pass  fixed
  E8  before fail  after pass  fixed
  pass rate: 2/8 -> 7/8 (25.0% -> 87.5%)
5. RELEASE
  the pipeline advises: hold (regression on E2)
  waiting for a human: run again with --release approve or --release hold REASON
```

Read the report from the top.

Curation. Six items were accepted and numbered #1 to #6 in arrival order (F01, F03, F04, F05, F06, F10). F02 was recognized as a duplicate of F01 by Equation (7.1), exactly as in the worked example of Section 7.5.2. F07 was too short. F08 and F09 were rejected by the reviewer, with reasons. F11 is awaiting a human decision, because nobody has looked at it yet. F10 and F06 both improve B4 but are not duplicates ($J = 6/9 \approx 0.67 < 0.7$), and F11 is not a duplicate of F03 ($J = 3/5 = 0.6$).

Rewards. The times now appear in WIB. F01, F03, F04 and F05 earned 0.05 each (base plus acceptance bonus). F06 earned nothing *yet*: its content was good and was accepted, but the account is unverified, so the reward is held. F02 and F07 earned nothing, with the reason shown. F08 and F09 earned the base 0.01 each: they were honest, non-trivial submissions that a person reviewed, even though they were not taken. User u1 had three rewarded items on 24 September by 17:00 WIB (F01, F05, F09), so F10 at 23:30 WIB hit the lab's daily cap, although its content was accepted into the dataset. F11 at 00:10 WIB fell on 25 September and earned the base again. The total emitted by this round is 0.23 PROFIT, computed exactly in micro-units.

Dataset. Six preference pairs were written. The first line of `dataset.jsonl`, pretty-printed with `python -m json.tool --json-lines`, is:


```
{
    "n": 1,
    "id": "F01",
    "prompt": "how long does express service take",
    "rejected": "Express service takes 2 days.",
    "chosen": "Express service is ready in 6 hours and costs Rp12,000 per kilogram.",
    "category": "wrong fact"
}
```


Measurement. The pass rate rose from 2 of 8 (25.0%) to 7 of 8 (87.5%). E1 (hours), E4 and E5 (express), E6 (pickup), E7 (silk) and E8 (QRIS) were fixed. But E2, “Are your opening hours the same every day?”, passed before and fails after. The reason is instructive: F03 replaced “We are open every day.” with “We are open from 07:00 to 21:00.”, which adds the hours and silently drops “every day”. The correction fixed one question and broke another, which is exactly what Equation (7.4) exists to catch.

Release. Because $R = \{\text{E2}\}$ is not empty, Equation (7.6) advises a hold, and the program waits for a person. Suppose the reviewer reads the report and decides to hold the release until F11, which keeps “every day” and adds the hours, has been reviewed:

```
python feedback_pipeline.py --release hold E2 regressed, wait for F11
```

The last three lines of the report now record the human decision, which the givers would see as “Not taken in v1.1”, with the reason:

```
5. RELEASE
  the pipeline advises: hold (regression on E2)
  human decision: Not taken in v1.1 (E2 regressed, wait for F11)
```

Exercise 7.2 continues the story: accept F11 and run the pipeline again.

7.10.4 Which day is it? UTC versus WIB

Listing 7.3 and **Listing 7.4** isolate the time-zone lesson of Equation (7.10) in a dozen lines each.

Listing 7.3 `day_key.py`: the UTC date and the WIB date of four instants.

```
"""Which "day" does a submission belong to? UTC date versus WIB date.

    python day_key.py
"""
from datetime import datetime, timedelta, timezone

WIB = timezone(timedelta(hours=7))
TIMES = ["2026-09-24T16:30Z", "2026-09-24T17:10Z",
         "2026-09-24T23:50Z", "2026-09-25T00:10Z"]

print("UTC time           WIB time           UTC day  WIB day")
for s in TIMES:
    t = datetime.strptime(s, "%Y-%m-%dT%H:%MZ").replace(tzinfo=timezone.utc)
    w = t.astimezone(WIB)
    print(f"{t:%d %b %H:%M} UTC   {w:%d %b %H:%M} WIB   "
          f"{t:%d %b}   {w:%d %b}")
print("A cap keyed on the UTC day resets at 07:00 WIB, not at midnight.")
```

Listing 7.4 `DayKey.java`: the same comparison with `OffsetDateTime` and `ZoneOffset.ofHours(7)`.

```
import java.time.OffsetDateTime;
import java.time.ZoneOffset;
import java.time.format.DateTimeFormatter;
import java.util.List;
import java.util.Locale;

/** Which "day" does a submission belong to? UTC date versus WIB date.
 *  java DayKey */
public class DayKey {
    public static void main(String[] args) {
        ZoneOffset wib = ZoneOffset.ofHours(7);
        DateTimeFormatter full = DateTimeFormatter.ofPattern("dd MMM HH:mm",
                                                             Locale.ENGLISH);
        DateTimeFormatter day =
                DateTimeFormatter.ofPattern("dd MMM", Locale.ENGLISH);
        List<String> times = List.of("2026-09-24T16:30Z", "2026-09-24T17:10Z",
                                     "2026-09-24T23:50Z", "2026-09-25T00:10Z");
        System.out.println("UTC time           WIB time"
                           + "           UTC day  WIB day");
        for (String s : times) {
            OffsetDateTime t = OffsetDateTime.parse(s.replace("Z", ":00Z"));
            OffsetDateTime w = t.withOffsetSameInstant(wib);
            System.out.println(t.format(full) + " UTC   " + w.format(full)
                               + " WIB   " + t.format(day) + "    " + w.format(day));
        }
        System.out.println("A cap keyed on the UTC day resets at 07:00 WIB,"
                           + " not at midnight.");
    }
}
```

```
UTC time           WIB time           UTC day  WIB day
24 Sep 16:30 UTC   24 Sep 23:30 WIB   24 Sep   24 Sep
24 Sep 17:10 UTC   25 Sep 00:10 WIB   24 Sep   25 Sep
24 Sep 23:50 UTC   25 Sep 06:50 WIB   24 Sep   25 Sep
25 Sep 00:10 UTC   25 Sep 07:10 WIB   25 Sep   25 Sep
A cap keyed on the UTC day resets at 07:00 WIB, not at midnight.
```

The second and third rows are the dangerous ones: for users in Indonesia they are already 25 September, but a program that asks “what is today’s date?” in UTC still says 24 September. Only at 00:10 UTC, which is 07:10 WIB, do the two agree again. A daily cap keyed on the UTC date would therefore let a user earn a full second allowance between midnight and 07:00 WIB, and would reset in the middle of the morning, when users least expect it.

7.11 Profitina in Practice

The lab is a miniature; here is the real pipeline, at box level.

Where feedback comes from. Users can give feedback on a scan and on the AI’s outputs, and the feedback status appears in their dashboard, the “cockpit” that gathers notifications about replies, likes, reposts, direct messages, pins, feedback status and PROFIT purchases in one place, each linked to its section [5]. A giver never has to wonder what happened to an item.

The five steps. Submit (earn PROFIT) → Curate (one local-AI call sorts category, severity and summary, then groups duplicates) → Dataset (“In dataset #N”, visible to the giver) → Train and measure the local model → Release (“Shipped in vX.Y.Z”, or “Not taken” with a reason). Product decisions are always made by a human, and givers who opt in are credited by pseudonym on a public “You changed this” page [5]. A weight-level fine-tune of the model, for example with LoRA, remains a roadmap item as of September 2026, and the release decision stays human either way.

Rewards with guardrails. Earning starts at 0.01 PROFIT, daily caps reset at 00:00 WIB, only email-verified humans earn, a local-AI usefulness judgment gives trivial or bot-like contributions 0.00 (with escrow and clawback to a burn), and duplicates earn 0.00 while linking to the earlier item [5]. The amounts are multiplied by the decaying emission multiplier (Chapter 12). PROFIT runs on Solana devnet, selling is disabled, and nothing about it is a promise of value.

The same principle elsewhere. The idea that the app advises and the user decides appears across the product: visibility results are estimates, not guarantees; the category auto-fill says "not sure yet" instead of guessing; *One Text, Many Platforms* promises a platform's character limit but not a "viral" result; and self-service privacy rights (export, correction, deletion) are always in the user's hands (Chapter 11) [5].

What the lab is not. The lab's categories, duplicate threshold, reward values, cap of 3 and release rule are this book's own teaching choices, and its "training" is an answer-bank update. The product's usefulness scoring, duplicate thresholds and escrow timing are its own and are not described here.

7.12 Common Misunderstandings

"More feedback is always better." Unfiltered feedback contains mistakes, duplicates, spam and wishful thinking (F08 wanted a lower price). Curation, human decisions and measurement turn a pile of feedback into a small amount of trustworthy data.

"If the model can sort feedback, it can also decide on it." Sorting is a suggestion that saves attention; deciding changes what the product tells people. The lab's rules sorted F11 correctly as *missing info*, but only a person could judge that it was the fix for the E2 regression.

"A higher pass rate means the release is safe." The lab's candidate rose from 25.0% to 87.5% and still broke E2. Always look at the regression set R, not only at the average.

"Rewarding feedback will bring in bots, so do not reward it." Unguarded rewards do attract bots. Caps, verification, usefulness judgments, duplicate detection and escrow make rewards possible without turning them into a farm. The alternative, rewarding nothing, tells contributors their effort is worth nothing.

"Rejected feedback should earn nothing." Honest, non-trivial feedback that a person reviewed has value even when it is not taken; it shows where users are confused. The lab pays the base reward for such items and the bonus only for accepted ones. What earns nothing is noise: duplicates, trivial text and unverified accounts.

"A daily cap is a daily cap; the time zone does not matter." A cap keyed on the UTC date resets at 07:00 WIB. For users in Indonesia, "today" starts at 00:00 WIB, and the code must say so explicitly.

"PROFIT rewards are a way to earn money." PROFIT is a utility coin on Solana devnet with no real monetary value today; selling is disabled; amounts are small and capped; and nothing in the product or this book promises a value or a return.

"The feedback pipeline is a customer-service inbox with extra steps." An inbox answers one person; the pipeline changes what the product tells everyone. That is why it needs what an inbox does not: duplicate grouping, a numbered and versioned dataset, measurement on held-out questions, and a release decision recorded with its reason. Treating feedback only as support

tickets loses the improvement; treating it only as training data loses the person. The pipeline keeps both: the giver sees their item's status, and the product learns from it.

"Devnet tokens are just a demo, so the rules around them do not matter." The rules matter precisely because they are being tested before any real value is involved. Caps, verification, duplicate handling and burns that are designed and measured on devnet are the ones that would carry over if the regulatory and audit gates for mainnet are ever passed; rules added in a hurry later are how reward systems are abused.

"Once a dataset is built, the evaluation set can be used for training too." Then it stops being a held-out set, and every future measurement overstates quality. Keep the evaluation set apart, and keep adding fresh questions to it.

7.13 Chapter Summary

- A model does not improve by itself; users notice its mistakes first. Human feedback, especially comparisons and concrete corrections, is how models learn what people prefer [1], [2].
- Profitina's pipeline has five steps: submit, curate, dataset, train and measure, release. The machine sorts and measures; a human accepts or rejects items and approves or holds releases; the giver sees each status ("In dataset #N", "Shipped in vX.Y.Z", "Not taken" with a reason).
- *The app advises, the user decides*: outputs are presented as estimates and suggestions, and decisions that change the product are made by people, in line with UU PDP Article 10 on automated decisions.
- Near-duplicates can be found with the Jaccard similarity of word sets (Equation (7.1)); automatic decisions should only remove noise and should always state their reason.
- Curated corrections become preference pairs (prompt, rejected, chosen) stored as JSON Lines, one object per line, without personal data.
- Measure on a held-out set: compute the pass rate (Equation (7.3)) before and after, and the set of regressions (Equation (7.4)). The lab went from 2/8 to 7/8 with one regression, so the pipeline advised a hold (Equation (7.6)).
- PROFIT rewards for feedback are small (0.01–0.05 per scan feedback), capped per day with the day defined in WIB (Equation (7.8), Equation (7.10)), paid only to verified humans, and zero for duplicates and trivial content. Amounts are best kept as integer micro-units (Equation (7.9)). PROFIT is a utility coin on devnet; selling is disabled; no value is promised.
- The lab implemented the whole loop in Python and Java with identical output: curation with human decisions, a JSONL dataset, capped rewards, before-and-after measurement, and a release that waits for a person.

7.14 Exercises

Exercise 7.1 (Jaccard by hand). Compute the Jaccard similarity, after the lab's cleaning (lowercase, commas removed inside numbers, stop words removed), of F06 "Yes, free pickup and delivery within 3 km." and F10 "We offer free pickup and delivery within 3 km of the shop." Then compute it for F03 and F11. With the lab's threshold of 0.7, which pairs are duplicates?

Exercise 7.2 (finish the story). The reviewer reads F11 and accepts it. Add the line `F11\taccept\tkeeps "every day" and adds the hours` to a copy of the decisions (the book's code folder ships it as `decisions_v2.tsv`) and run `python feedback_pipeline.py --decisions`

`decisions_v2.tsv --release approve`. Predict before running: the number of preference pairs, the pass rate after, the regressions, the advice, and F11's reward.

Exercise 7.3 (the reward ledger). Using Equation (7.7) with $B = 0.01$ and $A = 0.04$, and the lab cap $K = 3$, recompute by hand the rewards of user u1's five items in the run of Section 7.10.3, in micro-units and in PROFIT. What would change if the cap were keyed on the UTC date?

Exercise 7.4 (a lower threshold). Change the duplicate threshold in the lab from 0.7 to 0.6. Which items change status? Explain why this is worse for the product, even though "more duplicates caught" sounds better.

Exercise 7.5 (pass rates and regressions). A candidate model is measured on a held-out set of 20 questions. Before: questions 1–12 pass. After: questions 1–10 and 13–18 pass. Compute p_{before}, p_{after} and R, give the advice of Equation (7.6), and describe two different human decisions that could be reasonable, with reasons.

Exercise 7.6 (a model in the curation step). Using the `OllamaClient` from Chapter 6, describe how you would replace the lab's keyword rules for category and severity with one fast-lane call per item, while keeping the pipeline honest when no model is available. Which parts of `curate` must stay rule-based?

Exercise 7.7 (floating point). In Python, adding the eleven rewards of Section 7.10.3 in order as floating-point numbers, `0.05 + 0.00 + 0.05 + 0.05 + 0.05 + 0.00 + 0.00 + 0.01 + 0.01 + 0.00 + 0.01`, gives `0.23000000000000004`, and adding `0.01` ten times gives `0.09999999999999999`. Explain why, and show how the lab's integer micro-units avoid the problem. What is 0.05 PROFIT in micro-units, and how many micro-units does the whole round emit?

Exercise 7.8 (what the giver sees). For each of F02, F06, F08, F10 and F11 in the first run, write the one-line status a giver should see in their dashboard, in plain language.

Exercise 7.9 (a dataset check). Write down three checks that should run on every line of `dataset.jsonl` before it is used, and say what each protects against.

Exercise 7.10 (advise or decide?). For each situation, say whether the software should decide on its own or only advise, and why: (a) rejecting a two-letter feedback item; (b) shipping a new model version with a higher pass rate and no regressions; (c) giving 0.00 PROFIT to a duplicate; (d) blocking a user's account because their feedback looked automated.

Exercise 7.11 (ordering the queue). Using the sort key of Equation (7.2) and the categories and severities printed in Section 7.10.3, list the order in which a reviewer should see the nine items that were not removed automatically. Treat F01's group as size 2 (it includes F02) and every other group as size 1. Where does F11 end up, and what does that suggest about ordering by severity alone?

7.15 Solutions to the Exercises

Solution 7.1. F06 gives {yes, free, pickup, delivery, within, 3, km} (7 words; "and" is a stop word) and F10 gives {offer, free, pickup, delivery, within, 3, km, shop} (8 words; "we", "and", "of" and "the" are stop words). They share {free, pickup, delivery, within, 3, km}, 6 words, and together have $7 + 8 - 6 = 9$ distinct words, so $J = 6/9 \approx 0.667$. F03 gives {open, 07:00, 21:00} and F11 gives {open, every, day, 07:00, 21:00}; they share 3 words out of 5 distinct ones, so $J = 3/5 = 0.6$. Neither pair reaches 0.7, so neither is a duplicate; only F01 and F02 ($J = 0.7$) are.

Solution 7.2. The run prints:

```
3. DATASET -> dataset.jsonl
  7 preference pairs written
4. TRAIN and MEASURE (held-out questions)
  E1  before fail  after pass  fixed
  E2  before pass  after pass
  E3  before pass  after pass
  E4  before fail  after pass  fixed
  E5  before fail  after pass  fixed
  E6  before fail  after pass  fixed
  E7  before fail  after pass  fixed
  E8  before fail  after pass  fixed
  pass rate: 2/8 -> 8/8 (25.0% -> 100.0%)
5. RELEASE
  the pipeline advises: release
  human decision: approved -> Shipped in v1.1
```

Seven preference pairs are written, because F11 joins the dataset as #7. The answer to B1 is now F11's "We are open every day from 07:00 to 21:00.", which satisfies both E1 (the hours) and E2 ("every day"), so the pass rate becomes 8/8 and R is empty. Equation (7.6) advises a release, and the human approves: the givers now read "Shipped in v1.1". F11's reward rises from 0.01 to 0.05, because it was accepted; it is still inside u1's allowance for 25 September. Eight of eight on eight questions is encouraging but not proof; the next round should add new held-out questions.

Solution 7.3. In WIB, u1's items fall at 09:00 (F01), 11:00 (F05), 17:00 (F09) and 23:30 (F10) on 24 September, and 00:10 (F11) on 25 September. F01 and F05 are accepted: $10{,}000 + 40{,}000 = 50{,}000$ µP each, that is 0.05 PROFIT. F09 is rejected but eligible: 10,000 µP, 0.01 PROFIT, and it is the third rewarded item of the day. F10 is the fourth, so Equation (7.8) sets $\text{elig} = 0$ and it earns 0. F11 is the first item of 25 September: 10,000 µP, 0.01 PROFIT, because it is not yet accepted. Total for u1: 120,000 µP, 0.12 PROFIT. With a UTC-keyed cap, all five items fall on 24 September UTC (F11 is at 17:10 UTC); F11 would then be the fifth item of the day and earn 0, while the cap would reset only at 07:00 WIB on 25 September.

Solution 7.4. With 0.6, F11 becomes a "duplicate of F03" ($J = 0.6$), while F10 and F06 ($J \approx 0.667$) become duplicates as well, so F10 is removed automatically. The dataset loses the very item (F11) that fixes the E2 regression, and it loses F10's clearer wording, and no human ever sees either of them. Catching more "duplicates" means the machine silently discards more information; a lower threshold should come only with a human check of the grouped items.

Solution 7.5. $p_{\text{before}} = 12/20 = 60\%$ and $p_{\text{after}} = 16/20 = 80\%$. The regressions are $R = \{11,12\}$, which passed before and fail after. Because R is not empty, the advice is *hold*. Reasonable human decisions: (1) hold, if questions 11 and 12 concern safety or money (for example, a price or a care instruction), and fix them first; (2) approve, if 11 and 12 are minor wording questions and the fixed questions 13–18 include a safety problem that users are hitting now, recording the regressions as known issues for the next round. Either way, the reason is written down.

Solution 7.6. For each item, send a short prompt to the fast lane containing only the comment and suggestion, ask for JSON such as `{"category": ..., "severity": ...}` with `format: "json"`, validate the values against the allowed lists, and retry once, as in Solution 6.6. If the client raises `NoModelAvailable` or the answer is invalid, fall back to the keyword rules and record "sorted by rules". The automatic *rejections*, too short and duplicate, should stay rule-based and explainable, and the accept/reject decision must stay human. The model's role is to save the reviewer time, never to decide.

Solution 7.7. Binary floating point stores numbers as sums of powers of two, and 0.01 and 0.05 have no exact binary representation, just as 1/3 has no exact decimal one. Each stored value is off by a tiny amount, and the errors accumulate over a sum, so the total differs from 0.23 in the seventeenth significant digit. A ledger that compares such totals, or sums millions of them, drifts. Integers have no such problem: $10{,}000 + 40{,}000 = 50{,}000$ exactly, and the lab converts to PROFIT only when printing. So 0.05 PROFIT is 50,000 µP, and the round of Section 7.10.3 emits $0.23 \times 10^6 = 230{,}000$ µP: four accepted items at 50,000 plus three base rewards at 10,000.

Solution 7.8. F02: "Thanks! This matches an earlier report (F01), so it was grouped with it; duplicates do not earn PROFIT." F06: "Accepted into the dataset. Verify your email address to receive the reward." F08: "Reviewed, not taken: it contradicts the current price list (Rp7,000 per kilogram)." F10: "Accepted into the dataset (#6). You reached today's reward limit; limits reset at 00:00 WIB." F11: "Received; waiting for review."

Solution 7.9. (1) The line parses as JSON and has exactly the expected fields with non-empty string values, which protects training tools from malformed data. (2) No field contains an email address, phone number or other personal identifier (a pattern check), which protects privacy. (3) The `id` refers to an item whose status is "in dataset", and no two lines have the same `id` or the same prompt–chosen pair, which protects against injected or duplicated rows. A fourth useful check: no `prompt` equals a held-out question, which protects the evaluation set.

Solution 7.10. (a) Decide automatically: the rule is simple, explainable and only removes noise; the status says why. (b) Advise only: a release changes what the product tells everyone, so a person approves it, even when the numbers are clean. (c) Decide automatically, with the reason and a link to the earlier item; the giver can object if the grouping is wrong. (d) Advise only: blocking an account has a significant effect on a person, which is exactly the kind of solely automated decision UU PDP Article 10 lets people object to [6]; a human must review it.

Solution 7.11. The nine items are F01, F03, F04, F05, F06, F08, F09, F10 and F11. F04 is the only *high* item, so it comes first. The *medium* items are F01 (group of 2) and F05 (group of 1), so F01 precedes F05 even though both are *wrong fact*. All other items are *low* with groups of size 1, so they follow in arrival order: F03, F06, F08, F09, F10, F11. The order is F04, F01, F05, F03, F06, F08, F09, F10, F11, and F11, the item that repairs the E2 regression, comes last. A reviewer with little time might never reach it. Severity describes how bad the reported problem is, not how valuable the fix is; a practical queue therefore also surfaces items that touch an answer with a known regression, or items that *add to* an earlier accepted correction, which is a good reason to keep a person, not only a formula, in charge of the queue.

References

[1] L. Ouyang *et al.*, "Training language models to follow instructions with human feedback," in *Advances in Neural Information Processing Systems 35 (NeurIPS 2022)*, 2022, pp. 27730–27744.

[2] P. F. Christiano, J. Leike, T. B. Brown, M. Martic, S. Legg, and D. Amodei, "Deep reinforcement learning from human preferences," in *Advances in Neural Information Processing Systems 30 (NIPS 2017)*, 2017, pp. 4299–4307.

[3] Stack Overflow, "AI," *2025 Stack Overflow Developer Survey*, Jul. 2025. [Online]. Available: https://survey.stackoverflow.co/2025/ai

[4] S. Amershi *et al.*, "Guidelines for human-AI interaction," in *Proc. CHI Conf. Human Factors in Computing Systems (CHI '19)*, Glasgow, UK, 2019, pp. 1–13.

[5] Profitina, *Profitina Whitepaper — Public Version v24*, Sep. 2026. [Online]. Available: https://app.profitina.com/static/whitepaper/

[6] Republik Indonesia, *Undang-Undang Nomor 27 Tahun 2022 tentang Pelindungan Data Pribadi* (UU PDP), Oct. 17, 2022, esp. Art. 10(1). [Online]. Available: https://jdih.komdigi.go.id/produk_hukum/view/id/832/t/undangundang+nomor+27+tahun+202

[7] E. J. Hu *et al.*, "LoRA: Low-rank adaptation of large language models," in *Proc. Int. Conf. Learning Representations (ICLR)*, 2022.

[8] R. Rafailov, A. Sharma, E. Mitchell, S. Ermon, C. D. Manning, and C. Finn, "Direct preference optimization: Your language model is secretly a reward model," in *Advances in Neural Information Processing Systems 36 (NeurIPS 2023)*, 2023.

[9] "JSON Lines," *jsonlines.org*. [Online]. Available: https://jsonlines.org/ (accessed Sep. 24, 2026).

[10] T. Bray, Ed., "The JavaScript Object Notation (JSON) data interchange format," IETF RFC 8259, Dec. 2017.

[11] M. Strathern, "'Improving ratings': Audit in the British university system," *European Review*, vol. 5, no. 3, pp. 305–321, 1997.

[12] D. Sculley *et al.*, "Hidden technical debt in machine learning systems," in *Advances in Neural Information Processing Systems 28 (NIPS 2015)*, 2015, pp. 2503–2511.

[13] Solana Foundation, "Token Program," *Solana Program Library documentation*. [Online]. Available: https://spl.solana.com/token (accessed Sep. 24, 2026).

Chapter 8
The Business Model

By the end of this chapter you will be able to explain, in one sentence and then in detail, how Profitina intends to pay for itself: two tiers, Free and Pro, where Pro costs Rp50,000 for 30 days and can be paid through QRIS or redeemed with PROFIT, renews itself from a PROFIT balance if the user wishes, and never comes bundled with advertising. You will understand why a third tier, Agency, was removed in August 2026 even though everything in it worked, and why the team adopted the rule "measure interest before building payment" after an honest look at the numbers showed zero PROFIT purchases in the whole production history. You will be able to say precisely what "a daily quota" means in a country whose day starts seven hours before Greenwich's, and why the reset belongs at 00:00 WIB. You will read the market in which all of this happens, from the 2026 APJII internet survey and Bank Indonesia's QRIS figures. Finally, you will build two small but complete programs, in Python and in Java: a quota and tier calculator that counts days the Indonesian way, and a subscription modeled as a *state machine* that accepts QRIS payments, PROFIT redemptions, reminders and renewals, with a set of tests that refuse to let the rules drift.

8.1 Where We Left Off

Chapter 7 closed the feedback loop: people submit feedback, the platform curates it, a dataset grows, a local model is trained and measured, and a human decides what ships. That loop is about *quality*. It makes the product better. This chapter asks a blunter question that every product must eventually answer: who pays for all of this, and why would they?

The question deserves care because the easy answers are traps. A platform with a community and a coin could, in principle, earn money in a dozen ways: advertising, selling data, selling tokens, charging per scan, charging agencies per client, taking a cut of every transfer. Each of these has been tried by someone somewhere. Most of them would contradict something that earlier chapters established as a principle: that brand data stays on Profitina's own server (Chapter 6), that the community is designed to reward useful contributions rather than attention (Chapter 4), that the feedback loop credits people rather than harvests them (Chapter 7).

So the business model is not a separate subject glued onto the engineering. It is the same set of values, expressed as prices and limits. We will see that the model is deliberately small: two tiers, one price, one payment rail, two ways to pay, no ads. We will also see that "small" was reached by removing things, not by never adding them, and that the removals were driven by measurement.

This chapter is written for three readers at once. The first is a small business owner deciding whether Pro is worth Rp50,000. The second is a student who wants to understand a real monetization case, with its successes and its disappointments told plainly. The third is a programmer who will one day have to implement quotas, subscriptions and renewals, and who would like to get the clock right the first time. The first half of the chapter speaks mostly to the first two readers; the labs in Sections 8.10 to 8.12 speak to the third, but they are written so that the first two can follow them as well.

8.2 What a Business Model Is

The phrase *business model* is used so loosely that it is worth pinning down before we use it.

> **DEFINITION.** A *business model* answers four questions about a product: *what value* it creates, *for whom*, *who pays* for that value and *how*, and *what it costs* to keep creating it. A business model is sustainable when, over time, what is paid covers what it costs.

The definition is short, and each of its four parts matters.

What value. For Profitina, Chapter 3 described the value in detail: a readiness score from 0 to 100 that estimates how likely a brand's pages are to be understood and cited by AI answer engines, a prioritized list of fixes, content drafts that close the gaps, and monitoring that shows whether things improve.

For whom. The platform serves businesses of every size, creators, researchers and organizations, and a community of people who help improve it. Not all of these people will ever pay, and a sound model does not pretend they will.

Who pays, and how. This is the heart of the chapter. In Profitina's case: some users pay for Pro, in Rupiah via QRIS or in PROFIT that they earned or bought.

What it costs. A platform that runs its own AI models on its own hardware (Chapter 6) has a cost structure that is unusual for a young product: no per-request bill from an AI provider, but a real machine, real electricity, real bandwidth and real hours of engineering and moderation.

8.2.1 Freemium, in one picture

Profitina's model belongs to a family called *freemium*: a free tier that anyone can use indefinitely, and a paid tier with more capacity or more features. The word is a blend of "free" and "premium". It is the model of many tools that readers already use every day: cloud storage with a free allowance, music services with a free plan, office suites with a free edition.

Freemium works when three things are true at the same time. The free tier must be good enough that people come, stay and tell others. The paid tier must be clearly better for the people who need more. And the cost of serving free users must stay small enough that the paying minority can carry them.

A convenient way to see the arithmetic is a single formula. Let N be the number of active accounts in a 30-day period, let c be the *conversion rate* (the fraction of active accounts that pay), and let P be the price of one paid term. Then the revenue R for that period is

$$R = N \cdot c \cdot P \tag{8.1}$$

Equation (8.1) looks trivial, and that is its value: it shows that there are only three levers. You can have more users (N), convince a larger share of them to pay (c), or charge more (P). Every pricing debate is a debate about which lever to pull, and at what cost to the other two. Raising P usually lowers c. Making the free tier richer usually raises N but can lower c, because fewer people need to upgrade.

How large is c in practice? A widely cited *Harvard Business Review* analysis of freemium businesses reports that most companies' conversion rates "range from 2% to 5%", and warns that a rate of 1% is probably too low [1]. The same analysis adds a second observation that matters for a community platform: a free user is typically worth 15% to 25% as much as a paying subscriber,

much of it through referrals [1]. Free users are not a cost to be minimized; they are part of how the product grows.

> **EXAMPLE.** Suppose, purely for illustration, that a GEO tool has $N = 2{,}000$ active accounts in a month and converts $c = 3\%$ of them at $P = \text{Rp}50{,}000$. Equation (8.1) gives $R = 2{,}000 \times 0.03 \times 50{,}000 = \text{Rp}3{,}000{,}000$ per 30 days. If the conversion rate were only 1%, the same user base would bring Rp1,000,000. These are invented numbers for a worked example, not Profitina's; the point is how quickly c dominates the result when N is modest.

Keep the example in mind. It explains why a young platform cannot hope to live on subscriptions alone in its first months, and why it must be careful about spending engineering time on payment machinery before anyone pays.

8.2.2 What Profitina deliberately does not sell

A business model is defined as much by what it refuses as by what it offers. Three refusals shape everything else in this chapter.

1. **No advertising, permanently.** Section 8.6 explains why this is a structural decision rather than a matter of taste.
2. **No sale of user data.** Scan history is private to its owner (Chapter 3), and brand data never leaves the server for AI processing (Chapter 6). A model that sold data would contradict both.
3. **No promise of returns.** PROFIT, the utility coin of Chapter 12, is "not a security, not electronic money, not a promise of returns" in the words of the public whitepaper [2]. Nothing in the business model depends on anyone expecting PROFIT to rise in value, and this chapter never suggests that it will.

8.3 The Market Profitina Serves

A price is only meaningful against the market it is set in. Profitina is Indonesia-first by design [2], so the market we need to understand is Indonesia's: how many people are online, how they pay, and what a small business owner is used to.

8.3.1 Who is online

The Indonesian Internet Service Providers Association (APJII) publishes an annual survey of internet use. Its 2026 survey, released in May 2026, counts **235,261,078** internet users in a population of **287,303,234**, a penetration of **81.72%**, up from 80.66% in 2025 [3]. That is about six million more users than a year earlier [4].

Averages hide geography, and Indonesia's geography is dramatic. **Figure 8.1** (a) shows penetration by region. Java leads at 85.95% and holds 58.24% of all users; Maluku and Papua trail at 69.74% [4]. Urban penetration is 84.75% and rural 78.18% [3]. Among the younger generations the figure is higher still: 90.34% for millennials and 89.02% for Generation Z, and 84.31% of respondents go online mainly through a smartphone [4].

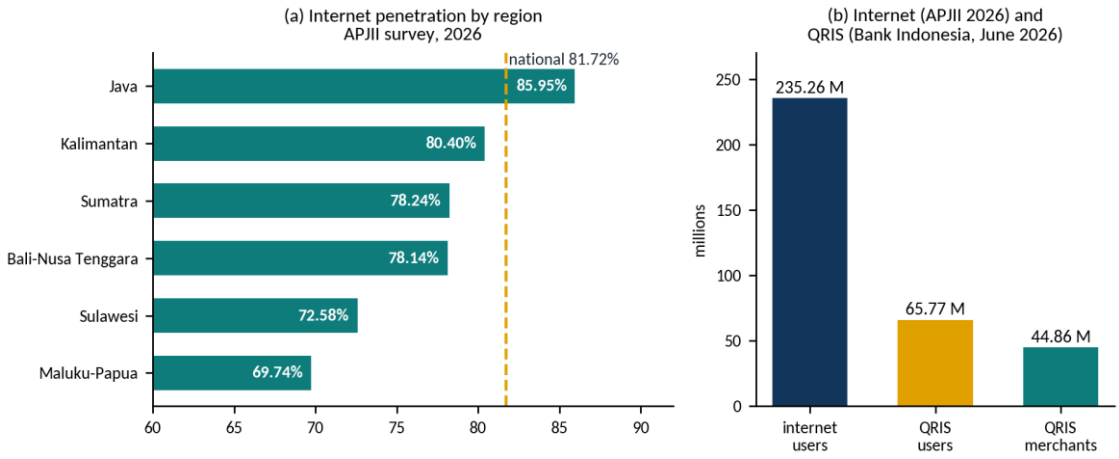


Figure 8.1 The market in 2026. (a) Internet penetration by region from the APJII survey, with the national figure of 81.72% dashed; the axis starts at 60%. (b) Internet users (APJII, 2026) beside QRIS users and QRIS merchants (Bank Indonesia, June 2026).

What does this mean for a business model? Three things. First, the potential audience is large: a quarter of a billion people online. Second, it is mobile: a product whose upgrade path requires a desktop, a credit card or a foreign account will lose most of that audience at the payment step. Third, the audience is unevenly distributed, so a price that feels trivial in a Jakarta office may be a real decision for a laundry owner in a smaller town. A price must be judged against the reader, not against the developer.

8.3.2 How people pay: QRIS

QRIS (Quick Response Code Indonesian Standard) is Bank Indonesia's national standard for QR-code payments. Its central idea is interoperability: a single QR code printed at a warung can be paid from any participating bank's mobile app or e-wallet, whichever one the customer happens to use. Before QRIS, a shop often displayed a row of different QR codes, one per wallet; after QRIS, one code serves them all.

The scale of QRIS in 2026 is remarkable. Bank Indonesia reported **65.77 million** QRIS users and **44.86 million** merchants accepting it as of June 2026, of which 96.68% are micro, small and medium enterprises [5]. In the first half of 2026 alone, QRIS carried **12.55 billion** transactions worth **Rp600.7 trillion**, roughly double the volume of the first half of 2025 [6]. **Figure 8.1** (b) sets the user and merchant counts beside the number of internet users.

Two facts follow for Profitina. First, QRIS is already the way small businesses in Indonesia expect to pay and be paid; Bank Indonesia's own deputy governor called it "the main gateway, the entry point, for MSMEs into the digital financial ecosystem" [5]. A laundry owner who pays suppliers by scanning a code will find paying for Pro the same way natural. Second, QRIS works in Rupiah, so the price can be stated in the currency people think in, with no exchange rate and no card network in between.

> **KEY INSIGHT.** Choosing a payment rail is choosing an audience. A card-only checkout quietly excludes people who do not have a card or do not want to use it online. QRIS includes the tens of millions who already pay this way every day.

8.3.3 What a small business compares the price with

A small business owner rarely compares a software subscription with other software. They compare it with things they already buy. Rp50,000 is in the range of a few plates of rice with a

drink, a small top-up of mobile data, or a modest printing job. That is not an accident of the number; it is what makes the price a decision that one person can take on the spot, without a meeting or a budget line. Section 8.4.3 turns the price into a daily figure, which is how many customers actually think about it.

8.4 Two Tiers: Free and Pro

In the September 2026 release, Profitina offers exactly two tiers. **Table 8.1** reproduces what the app's plan cards say [2].

Table 8.1 The two tiers as the plan cards describe them in the September 2026 release.

	Free	Pro
Scans and auto-fill	15 per day	Unlimited scans and content
Content Engine, schema, Technical Assistant	Yes	Yes
Scheduled monitoring	1 monitor, weekly	50 daily monitors, with alerts
History	30 days	Full history and trends, CSV export
Term and price	No charge	30 days, Rp50,000 via QRIS, or redeemed with PROFIT
Renewal	Not applicable	Auto-renew from PROFIT balance, reminder at most 24 hours before expiry

Read **Table 8.1** slowly, because its shape carries the philosophy.

8.4.1 What Free includes, and why so much

The Free tier is not a demo. It includes the full AI Visibility Engine, the Content Engine, schema.org help and the Technical Assistant. A Free user can scan, read the fix list, generate an answer-first draft with FAQ and JSON-LD, and apply it. Nothing essential to *improving* a site is behind the paywall.

Why give so much away? Because of the "for whom" in the definition of Section 8.2. Many of the businesses that most need help with AI visibility are the ones least able to pay for it. A Free tier that actually fixes things spreads good practice, produces satisfied users who tell others, and generates the feedback that Chapter 7's loop feeds on. It also fits the platform's stance that the value is the improvement itself, not access to a number.

What Free limits is *volume* and *continuity*: 15 scans a day, one monitor that re-scans weekly, and 30 days of history. Those limits bite exactly the users whose work has become regular: an agency-minded freelancer checking many sites, a business that wants daily tracking of several pages, a team that must report trends over a quarter.

Two further things are free for everyone, forever, and appear in no plan card at all: the rights of Chapter 11 (export your data, correct it, delete your account) and safety features such as Link Safety and account recovery [2]. A business model must never charge for a person's rights.

8.4.2 What Pro adds

Pro removes the scan limit, raises monitoring from one weekly monitor to fifty daily ones with alerts, and keeps the full history with trend charts and CSV export. Each of these is a *capacity* feature: it lets a user do more of what Free already does, more often and over a longer period.

None of them unlocks a secret better score. A Pro user's site is measured by exactly the same engine as a Free user's site.

This is a design choice worth stating plainly: the tier changes *how much* and *how often*, never *how honestly*. If Pro users received a kinder score, the score would stop meaning anything for anyone.

8.4.3 The price, per day

The price is Rp50,000 for a 30-day term. Dividing by the length of the term gives the daily price p:

$$p = \frac{\text{Rp50,000}}{30 \text{ days}} \approx \text{Rp1,666.67 per day} \tag{8.2}$$

Equation (8.2) is a useful sentence to be able to say to a customer: Pro costs less than Rp1,700 a day. The lab in Section 8.10 computes this number from the plan table rather than typing it in, so that if the price ever changes, the explanation changes with it.

A second way to read the price is per scan. If a Pro user runs n scans a day on average, the cost per scan is p/n. At $n = 30$, just double the Free allowance, each scan costs about Rp56. The more a user relies on the product, the cheaper each use becomes, which is the right shape for a tool meant to be used regularly.

8.4.4 A rule for plan cards

The team works under a rule that is easy to state and surprisingly easy to break: **never write a promise of an unbuilt feature next to a paid price**. A plan card that says "coming soon: API access" beside "Rp50,000" invites people to pay for something that does not yet exist. If the feature is late, or never arrives, the product has in effect taken money under a false description.

The rule has a practical consequence for this book as well. Everything in **Table 8.1** is live in the September 2026 release. When later chapters describe roadmap items, such as the PROFIT mainnet in Chapter 12, they are labeled as roadmap and never attached to a price.

8.5 Why the Agency Tier Was Removed

Until early August 2026 there was a third tier, Agency, above Pro. It added three things: **multi-client** management (one account handling many clients' brands), **API access** (programs rather than people calling the platform), and **white label** (reports carrying the agency's own brand rather than Profitina's). On 6–7 August 2026 the tier was removed, leaving two [2]. **Figure 8.2** shows the before and after.

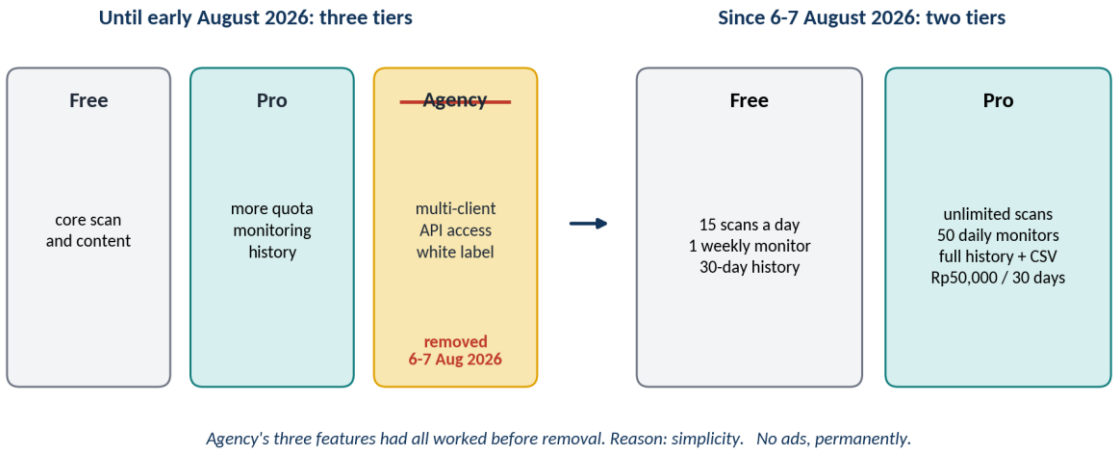


Figure 8.2 The tier structure before and after 6–7 August 2026. Agency's three features had been shown to work; the tier was removed for simplicity, and the permanent no-ads stance was kept.

The most interesting fact about the removal is the reason that was *not* given. Agency was not removed because its features were broken. The whitepaper records that all three "had proven to work before removal" [2]. It was removed "for simplicity". That deserves an explanation, because removing working features for simplicity sounds, at first, like waste.

8.5.1 The hidden cost of a tier

A tier is not only a price on a card. It is a set of rules that every part of the system must respect. Each feature has to ask, at the moment it runs, "which tier is this user in, and what am I allowed to do for them?" Each such question is a place where the answer can be wrong.

We can make the cost concrete. Suppose the product has F features whose behavior depends on the tier, and T tiers. The number of feature-and-tier combinations that must be designed, documented, tested and explained is

$$C = T \times F \tag{8.3}$$

With $F = 12$ tier-dependent features, two tiers give $C = 24$ combinations and three tiers give $C = 36$, half as many again, by Equation (8.3). And the real cost is worse than linear, because tiers interact with time: what happens when an Agency user downgrades to Pro while holding twenty client brands and a live API key? Which clients survive? Does the key stop at once or at the end of the term? Each transition between tiers is another set of questions, and with T tiers there are $T(T-1)$ possible directed transitions: two with two tiers, six with three.

For a small team, every one of those combinations and transitions competes for the same hours as the core product. Removing Agency did not remove three features; it removed twelve combinations, four transitions, a paragraph of legal terms, a column of plan card, and a whole category of support questions.

8.5.2 Simplicity for the customer

The second reason is on the customer's side. A choice between two options is quick: "Do I need more than Free gives me?" A choice among three invites comparison shopping, second-guessing and the worry of picking the wrong one. For a product whose users are often small businesses meeting AI visibility for the first time, one clear question is kinder than a matrix.

8.5.3 What happens to agencies now

Agencies are still welcome. An agency that manages several clients can use one account per client, or a Pro account with up to fifty daily monitors. What no longer exists is a dedicated workflow for them. If a real, measured demand for multi-client features appears, it would be a new decision about a new product, not a restoration of the old tier. Section 8.8 explains how such a decision should be made.

> **WARNING.** Old marketing copy that mentions an Agency tier, or a PROFIT price for Agency, describes the product as it was before August 2026. It is not a preview of the future. When sources disagree, the app's current plan cards are the reference to trust.

8.6 No Ads, Permanently

Many free services pay for themselves with advertising. Profitina has committed to **no ads, permanently** [2]. This is often read as a stylistic preference, a wish for a clean screen. It is more than that.

8.6.1 The conflict of interest

An advertising business earns money in proportion to attention: the longer people look, the more ads they see. Its incentives therefore point toward keeping people on the platform, and toward the kinds of content and design that do so.

Profitina's value to a user points the other way. A good scan tells you what to fix and lets you go and fix it on your own site. A good fix list is short. A good community post answers a question and ends the thread. If the platform earned money from attention, every one of these good outcomes would be, from the revenue's point of view, a loss. That is a conflict of interest built into the model, and a conflict of interest built into a model does not stay theoretical for long.

A subscription reverses the incentive. A Pro user renews because the product made their visibility better last month. The platform earns money by being useful, not by being sticky.

8.6.2 The privacy argument

Advertising at scale depends on targeting, and targeting depends on data about people: what they look at, what they buy, who they are. Chapter 11 describes Profitina's approach to personal data under UU PDP No. 27/2022: collect as little as possible, keep it on its own server, give people self-service rights. An ad business would pull in the opposite direction on every one of those points. Refusing ads is a precondition for keeping the privacy promises, not a separate decision.

8.6.3 The credibility argument

Finally, a platform that tells businesses how trustworthy their content looks to AI engines must itself look trustworthy. A GEO dashboard surrounded by banner ads for "guaranteed page-one results" would undermine the very message it exists to deliver.

> **KEY INSIGHT.** "No ads" is a statement about incentives. It aligns what the platform earns with what the user gains, protects the privacy stance of Chapter 11, and keeps the product's advice credible.

8.7 Paying for Pro: QRIS and PROFIT

There are two ways to pay for a Pro term: in Rupiah through QRIS, or by redeeming PROFIT. **Table 8.2** separates what is live from what is not, because in anything to do with money the difference must be impossible to miss.

Table 8.2 What is live and what is not, for everything in the business model that involves payment.

Capability	Status in the September 2026 release
Pro via QRIS, Rp50,000 per 30 days	Live
Pro by redeeming PROFIT	Live
Auto-renew from PROFIT balance, reminder at most 24 hours before expiry	Live
Buying PROFIT via QRIS (Rp50,000 to Rp1,000,000)	Live; credited when an administrator verifies the payment
Spending PROFIT on scan top-ups and the Academic Citation Scan	Live
Selling or redeeming PROFIT back to Rupiah	Disabled until mainnet and regulatory clarity
PROFIT mainnet and a non-custodial exchange	Roadmap (Phase 3, gated by audit and licensing)

8.7.1 QRIS, step by step

From the user's point of view, paying with QRIS is the same as paying at any shop. The user chooses Pro, the app shows a QRIS code and the amount, the user scans it with the banking or wallet app they already use (DANA, GoPay, OVO, ShopeePay or a mobile-banking app), and confirms. Once the payment is confirmed, Pro begins and runs for 30 days.

Behind that simple experience there is a small but important piece of logic: an *order* that moves through states. It is created, it waits for payment, it is checked, and it either becomes an active term or is closed. The order can also simply expire if nobody pays. Section 8.11 builds this logic as a program, and **Figure 8.5** previews it.

In the September 2026 release, PROFIT bought through QRIS is credited when an administrator verifies the payment [2]. A human check at the moment money turns into something of value is slower than a fully automatic gateway, but it is cheap to build, easy to audit, and appropriate for a platform whose purchase volume is still small. Section 8.8 explains why this is a deliberate stage rather than a shortcut.

8.7.2 Redeeming PROFIT

The second way to pay is to redeem PROFIT, the utility coin that users earn by contributing useful feedback and community activity (Chapter 7) or buy through QRIS. The app shows the PROFIT price before the user confirms. When a user redeems, the PROFIT spent is **burned**: it leaves circulation rather than going into a company wallet [2]. Chapter 12 explains why burning matters for the coin's supply; here it is enough to note that redeeming is a use of a utility, not a trade.

Redeeming has a pleasant property for the business model: a user who earned PROFIT by helping the platform can turn that contribution into Pro time. The product rewards its most useful community members with the same capacity others pay Rupiah for.

8.7.3 Auto-renew and the reminder

Pro can renew itself at the end of a term by redeeming PROFIT from the user's balance, if the user has switched auto-renew on and the balance is sufficient. A reminder arrives **at most 24 hours before expiry** [2]. Both rules exist for the user's benefit: auto-renew avoids an accidental gap in daily monitoring, and the reminder ensures nothing renews silently.

Getting renewal right is subtler than it looks. If the renewal is processed a few hours late, because the machine was busy or restarted, from which moment does the new term run? From the old expiry, or from the moment the renewal ran? The fair answer is from the old expiry: the user paid for thirty days following the previous thirty days, not for thirty days following an accident of the server's schedule. Writing e_k for the expiry of the k-th term,

$$e_{k+1} = e_k + 30 \text{ days} \tag{8.4}$$

regardless of when the renewal code actually ran. Equation (8.4) looks too obvious to need writing down, and yet the first version of the lab code in Section 8.11 got it wrong, and a test caught it. Section 8.12 tells that story.

> **WARNING.** Nothing in this section is financial advice. PROFIT is a utility coin on Solana devnet with no real monetary value in the September 2026 release; it is not a security, not electronic money and not a promise of returns [2]. Redeeming it for Pro is a use of a product feature.

8.8 Measure Interest Before Building Payment

This section tells the least glamorous story in the chapter, and perhaps the most useful one.

8.8.1 The zero-purchase finding

In August 2026 the team analyzed how PROFIT was actually being acquired and used. The finding, recorded in the public whitepaper, was blunt: **zero PROFIT purchases across the whole production history** [2]. People were earning PROFIT, spending it, and using the community, but nobody had bought any with Rupiah.

A less honest team would have buried that number, or explained it away with the launch date, or kept building more payment features in the hope that the next one would unlock demand. Profitina did something better: it wrote the number down in a public document and drew a rule from it.

> **DEFINITION.** *Measure interest before building payment* means: before investing engineering effort in a way to take money for something, obtain evidence that people want to pay for it, using the cheapest mechanism that produces real evidence.

8.8.2 Why the rule is right

Payment machinery is expensive in ways that are not obvious from the outside. A fully automatic checkout needs integration with a payment provider, reconciliation of what was paid with what was credited, handling of partial payments, duplicate notifications, refunds, receipts, tax records, and a customer-support path for every one of those going wrong. Each of those is code that must be tested and maintained, and each is a place where a mistake costs someone real money.

If nobody is buying, all of that effort earns nothing. Worse, it can hide the real problem. When a feature does not sell, the question is usually "is this worth paying for?" rather than "is the checkout smooth enough?". A beautiful checkout for something nobody wants is still unsold.

The idea is not new. Eric Ries's *The Lean Startup* built a method around it: treat each product decision as an experiment, build the smallest thing that can test the riskiest assumption, measure what real people do, and learn before building more [7]. The riskiest assumption in a payment feature is not "can we take money?". It is "will anyone pay?".

8.8.3 A ladder of evidence

Figure 8.3 shows the rule as a ladder. Each step costs more to build than the one below it, and one climbs only when the step below shows real demand.

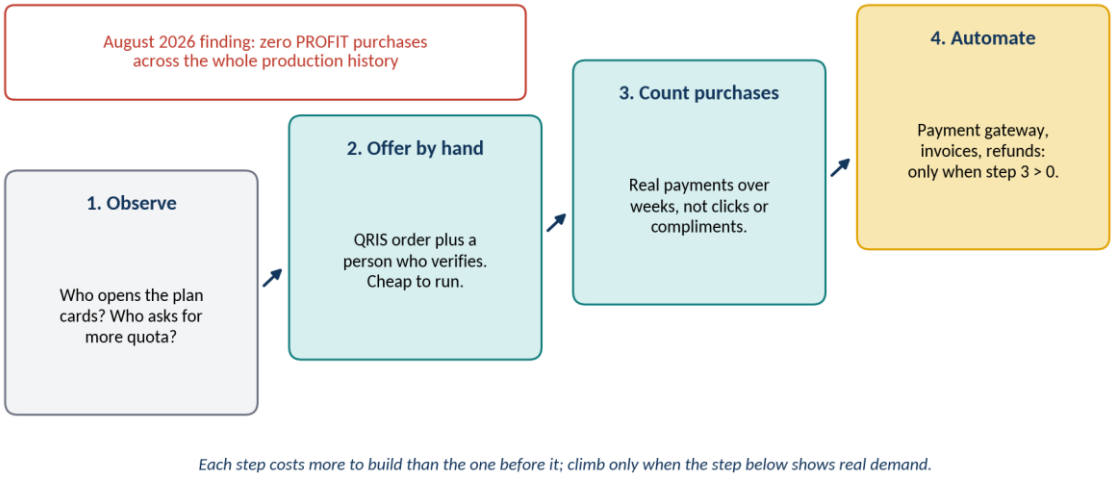


Figure 8.3 Measure interest before building payment, as a ladder. Each step is built only when the one below it shows real demand; the August 2026 finding of zero PROFIT purchases kept the platform on the lower steps.

1. **Observe.** Who opens the plan cards? Who hits the daily limit? Who asks for more quota? This costs almost nothing and is honest evidence of *need*, though not of willingness to pay.
2. **Offer by hand.** Make it possible to pay with the simplest reliable mechanism: a QRIS order and a person who verifies. This is the stage the September 2026 release is at for PROFIT purchases. It is cheap to build, easy to audit, and produces the only evidence that matters: real payments.
3. **Count purchases.** Watch real payments over weeks, not clicks or compliments. People say many things in surveys; what they pay for is what they value.
4. **Automate.** Only when purchases exist, and the manual step has become a bottleneck, invest in a payment gateway, automatic invoices and refunds.

Notice what the zero-purchase finding means on this ladder. It does not mean the product failed. It means step 3 returned zero, and therefore step 4 should not be built yet. The money and time that would have gone into automating payment went into the product itself instead.

8.8.4 The same rule elsewhere

The rule is not special to PROFIT. In mid-September 2026 the team evaluated a proposal to offer free AI assistance to the roughly 250 students of the first university cohort using the platform, measured what it would cost and what it would deliver, and **froze** it after the measurement. A feature that sounds generous is still a feature that must be justified by evidence, and the same discipline that stops an unused checkout also stops an unaffordable gift.

> **KEY INSIGHT.** An honest zero is information. The zero-purchase finding did not end the business model; it told the team where not to spend its next month.

8.9 Daily Quotas and the 00:00 WIB Rule

The Free tier allows 15 scans *per day*, and PROFIT earning has daily caps as well. “Per day” sounds like the simplest phrase in the chapter. It is, in fact, the phrase most likely to be implemented wrongly, and Profitina learned this the way most teams do: by getting it wrong once.

8.9.1 Whose day?

A day is not a universal thing. It is a stretch of 24 hours between two midnights, and midnight arrives at different instants in different places. Computers usually keep time as *Coordinated Universal Time* (UTC), the time at longitude zero with no daylight saving. Western Indonesia Time (WIB, *Waktu Indonesia Barat*), used in Java, Sumatra and western and central Kalimantan, is seven hours ahead: WIB = UTC + 7 hours, all year, with no daylight saving. The time-zone database that computers use names this zone `Asia/Jakarta` [8].

Profitina’s daily quotas reset at **00:00 WIB** [2]. That is the midnight its users live by. A laundry owner in Surabaya who used up their 15 scans on Wednesday evening expects to have 15 again when they open the app on Thursday morning at six.

8.9.2 The UTC-date trap

Here is the trap. The easiest code to write for “how many scans today?” takes the current time as the computer holds it, in UTC, takes its calendar date, and counts scans with the same date. That code is short, looks correct, and passes every test that someone writes during office hours in Indonesia.

It is wrong for seven hours of every day. **Figure 8.4** shows why. At 00:05 WIB on 25 September, the UTC clock reads 17:05 on *24 September*. A quota counted by UTC date therefore still sees all of Wednesday’s scans until 07:00 WIB on Thursday, when the UTC date finally turns over. Between midnight and seven in the morning, the user is told they have no scans left, even though, by every calendar they use, it is a new day.

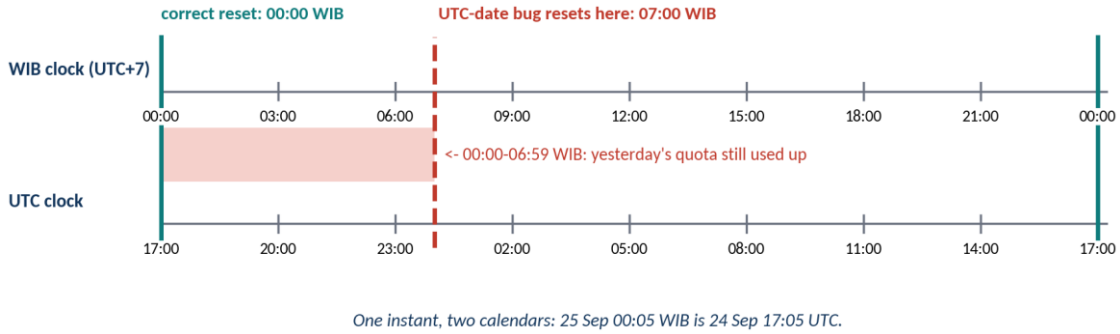


Figure 8.4 One day on two clocks. The correct reset is at 00:00 WIB; code that uses the UTC date resets at 07:00 WIB instead, leaving users with yesterday’s exhausted quota for the first seven hours of their day.

This exact bug, a daily reset that happened at 07:00 WIB because the code used the UTC date, is one that Profitina found and fixed in its own history, and the rule it produced is part of how the team works: every daily limit is counted in WIB.

8.9.3 The day as a formula

We can say precisely what “the WIB day” is. Let t be a moment in time, written as the number of seconds since the Unix epoch (1 January 1970, 00:00 UTC), which is how most systems store

instants. A day has 86,400 seconds and WIB is $7 \times 3{,}600 = 25{,}200$ seconds ahead of UTC. The day number of t in UTC and in WIB are

$$d_{\text{UTC}}(t) = \left\lfloor \frac{t}{86{,}400} \right\rfloor \tag{8.5}$$

$$d_{\text{WIB}}(t) = \left\lfloor \frac{t + 25{,}200}{86{,}400} \right\rfloor \tag{8.6}$$

where $\lfloor x \rfloor$ means "the largest whole number not greater than x". Two scans belong to the same quota day exactly when they have the same d_{WIB}. Equation (8.5) and Equation (8.6) differ only by the shift of 25,200 seconds inside the floor, and that shift is the entire difference between the correct behavior and the bug of **Figure 8.4**.

> **EXAMPLE.** The instant 25 September 2026, 00:05 WIB is $t = 1{,}790{,}269{,}500$ seconds. Then $t/86{,}400 = 20{,}720.71\ldots$, so $d_{\text{UTC}} = 20{,}720$, which is 24 September. But $(t + 25{,}200)/86{,}400 = 20{,}721.00\ldots$, so $d_{\text{WIB}} = 20{,}721$, which is 25 September. Same instant, different day numbers: the UTC rule still counts Wednesday's scans, the WIB rule starts afresh.

In real code we do not compute floors by hand. Python's `zoneinfo` module and Java's `java.time` package know the rules for `Asia/Jakarta` [8], [9], and converting an instant to a WIB date is one method call. The formulas are there so that we understand what the method call does, and can recognize the bug when we see it.

8.9.4 How many scans are left

With the day defined, the remaining quota is simple. Let q be the plan's daily allowance (15 for Free), let u be any extra scans the user bought as a top-up *for this WIB day*, and let s be the scans already used this WIB day. Then the remaining scans r are

$$r = \max(0,\ q + u - s) \tag{8.7}$$

The max with zero keeps the answer from going negative if, for example, a top-up expired. For Pro, q is unlimited and Equation (8.7) does not apply; the program represents "unlimited" explicitly rather than with a very large number, so that no one ever sees "999,999 scans left".

Top-ups deserve a sentence of their own. In the September 2026 release, PROFIT can be spent on scan top-ups that are valid **for the purchase day only** [2]. "The purchase day" is, again, the WIB day. A top-up bought at 23:00 WIB is valid for one hour; that is precisely why the app shows the price and the validity before the user pays.

8.10 Lab: Plans as Data and a Quota Calculator

We now build the first program of the chapter. It does three things: it stores the two plans as *data* rather than scattering numbers through the code, it counts scans per WIB day with same-day top-ups, and it demonstrates the UTC-date bug side by side with the correct behavior. The program is a teaching model built for this book. The only product facts in it are the numbers from the public plan cards in **Table 8.1**.

8.10.1 Design

Three decisions shape the code.

Plans are data. A `Plan` record holds the name, scans per day, number of monitors, monitor cadence, history length, CSV export flag and price. The two plans live in a dictionary (Python) or map (Java) called `PLANS`. Code that needs a limit looks it up; no function contains the number 15. When a price or limit changes, one line changes.

"Unlimited" is a value, not a big number. Pro's scans per day and history length are `None` in Python and `null` in Java. Every function that reads them checks for that case explicitly.

The day is a function you can swap. `QuotaBook` receives a function `day_of` that maps an instant to a calendar date. The default is `wib_day`. Passing `utc_day` instead reproduces the bug, so that both behaviors can be printed next to each other and tested.

8.10.2 The Python program

Listing 8.1 is the complete program.

Listing 8.1 `tiers.py`: the plans as data, a quota book that counts scans per WIB day, same-day top-ups, and the UTC-date bug for comparison.

```
"""Free and Pro plans as data, and a quota calculator whose day
starts at 00:00 WIB (Western Indonesia Time, UTC+7)."""
from dataclasses import dataclass
from datetime import date, datetime, time, timedelta, timezone
from zoneinfo import ZoneInfo

WIB = ZoneInfo("Asia/Jakarta")


@dataclass(frozen=True)
class Plan:
    name: str
    scans_per_day: int | None      # None means unlimited
    monitors: int
    monitor_every_days: int
    history_days: int | None       # None means full history
    csv_export: bool
    price_idr: int                 # per 30 days; 0 for Free


PLANS = {
    "free": Plan("Free", 15, 1, 7, 30, False, 0),
    "pro": Plan("Pro", None, 50, 1, None, True, 50_000),
}
TERM_DAYS = 30


def wib_day(instant: datetime) -> date:
    """The calendar day in Indonesia on which this instant falls."""
    return instant.astimezone(WIB).date()


def utc_day(instant: datetime) -> date:
    """The WRONG day for a quota in Indonesia: the UTC calendar day."""
    return instant.astimezone(timezone.utc).date()


def next_reset(instant: datetime) -> datetime:
    """The next 00:00 WIB after this instant, as a UTC instant."""
    tomorrow = wib_day(instant) + timedelta(days=1)
    return datetime.combine(tomorrow, time(0), WIB).astimezone(timezone.utc)


class QuotaBook:
    """Counts one account's scans per WIB day, plus same-day top-ups."""
```

Listing 8.1 (Continued) *tiers.py: the plans as data, a quota book that counts scans per WIB day, same-day top-ups, and the UTC-date bug for comparison.*

```
    def __init__(self, tier: str, day_of=wib_day):
        self.plan = PLANS[tier]
        self.day_of = day_of            # wib_day, or utc_day to show the bug
        self.scans: list[datetime] = []
        self.topups: dict[date, int] = {}

    def allowance(self, now: datetime) -> int | None:
        if self.plan.scans_per_day is None:
            return None
        return self.plan.scans_per_day + self.topups.get(self.day_of(now), 0)

    def used(self, now: datetime) -> int:
        today = self.day_of(now)
        return sum(1 for t in self.scans if self.day_of(t) == today)

    def remaining(self, now: datetime) -> int | None:
        limit = self.allowance(now)
        return None if limit is None else max(0, limit - self.used(now))

    def try_scan(self, now: datetime) -> bool:
        left = self.remaining(now)
        if left is not None and left == 0:
            return False
        self.scans.append(now)
        return True

    def add_topup(self, now: datetime, extra: int) -> None:
        """Extra scans bought today are valid for today (WIB) only."""
        day = self.day_of(now)
        self.topups[day] = self.topups.get(day, 0) + extra

    def can_add_monitor(self, active_monitors: int) -> bool:
        return active_monitors < self.plan.monitors

    def history_from(self, now: datetime) -> date | None:
        """Oldest visible day of scan history; None means everything."""
        if self.plan.history_days is None:
            return None
        return self.day_of(now) - timedelta(days=self.plan.history_days)


def price_per_day(plan: Plan) -> float:
    return plan.price_idr / TERM_DAYS
```

Listing 8.1 (Continued) *tiers.py: the plans as data, a quota book that counts scans per WIB day, same-day top-ups, and the UTC-date bug for comparison.*

```
def at(text: str) -> datetime:
    """Parse 'YYYY-MM-DD HH:MM' as a WIB wall-clock time."""
    return datetime.strptime(text, "%Y-%m-%d %H:%M").replace(tzinfo=WIB)


def show(n) -> str:
    return "unlimited" if n is None else str(n)


def main() -> None:
    print("Plan    scans/day  monitors  every  history  CSV  price/30d")
    for p in PLANS.values():
        hist = "full" if p.history_days is None else f"{p.history_days} d"
        print(f"{p.name:<7} {show(p.scans_per_day):>9}  {p.monitors:>8}  "
              f"{p.monitor_every_days:>3} d  {hist:>7}  "
              f"{'yes' if p.csv_export else 'no':>3}  Rp{p.price_idr:,}")
    print(f"Pro costs Rp{price_per_day(PLANS['pro']):,.2f} per day")

    good = QuotaBook("free")                 # counts WIB days
    bad = QuotaBook("free", day_of=utc_day)  # counts UTC days
    start = at("2026-09-24 20:00")
    for i in range(15):
        t = start + timedelta(minutes=14 * i)
        good.try_scan(t)
        bad.try_scan(t)
    late = at("2026-09-24 23:45")
    print("\n15 scans on 24 Sep between 20:00 and 23:16 WIB")
    print("16th scan at 23:45 WIB allowed?", good.try_scan(late))
    print("next reset:", next_reset(late).astimezone(WIB),
          "=", next_reset(late))

    for clock in ("2026-09-25 00:05", "2026-09-25 06:59",
                  "2026-09-25 07:00"):
        now = at(clock)
        print(f"{clock} WIB  remaining (WIB day) = {good.remaining(now):>2}"
              f"   remaining (UTC day) = {bad.remaining(now):>2}")

    now = at("2026-09-25 21:00")
    for _ in range(15):
        good.try_scan(now)
    good.add_topup(now, 5)
    print("\nAfter 15 scans on 25 Sep and a top-up of 5 at 21:00 WIB:")
    print("remaining on 25 Sep 21:00:", good.remaining(now))
    print("remaining on 26 Sep 00:01:", good.remaining(at("2026-09-26 00:01")))
```

Listing 8.1 (Continued) *tiers.py: the plans as data, a quota book that counts scans per WIB day, same-day top-ups, and the UTC-date bug for comparison.*

```
    pro = QuotaBook("pro")
    for _ in range(100):
        pro.try_scan(now)
    print("\nPro after 100 scans today, remaining:", show(pro.remaining(now)))
    print("Free may add a 2nd monitor?", good.can_add_monitor(1))
    print("Pro may add a 2nd monitor?", pro.can_add_monitor(1))
    print("Free history starts:", good.history_from(now))
    print("Pro history starts:", pro.history_from(now) or "the beginning")


if __name__ == "__main__":
    main()
```

A few lines deserve comment.

`WIB = ZoneInfo("Asia/Jakarta")` loads the time-zone rules for Western Indonesia from the system's time-zone database [8]. Every datetime in the program is *aware*, meaning it carries its zone, so there is never any doubt about which clock a time was read from.

`wib_day` converts an instant to Jakarta time and takes its date. `utc_day` does the same with UTC, and its docstring calls it wrong on purpose. `next_reset` finds tomorrow's date in WIB, builds midnight on that date in WIB, and converts it back to UTC; the result, 17:00 UTC, is the instant the quota resets as a server's log would record it.

In `QuotaBook`, `allowance` returns the plan's daily scans plus any top-up for the current day, or `None` for unlimited. `used` counts scans whose day equals today's day, by whichever `day_of` function the book was given. `remaining` is Equation (8.7) in code. `try_scan` refuses the scan when nothing remains and records it otherwise. `add_topup` credits extra scans to the current day and to no other day, which is how "valid for the purchase day only" is implemented: tomorrow's `allowance` simply does not look at today's top-up.

`main` runs a small story. It prints the plan table, then makes 15 scans between 20:00 and 23:16 WIB on 24 September in two quota books, one counting WIB days and one counting UTC days. It tries a sixteenth scan, prints the next reset, and then asks both books how many scans remain at three moments on 25 September: just after midnight, one minute before seven, and at seven. It ends with a top-up, the Pro tier's unlimited scans, the monitor limits and the history cut-offs. The output is:

```
Plan    scans/day  monitors  every  history  CSV  price/30d
Free           15         1    7 d     30 d   no  Rp0
Pro     unlimited        50    1 d     full  yes  Rp50,000
Pro costs Rp1,666.67 per day

15 scans on 24 Sep between 20:00 and 23:16 WIB
16th scan at 23:45 WIB allowed? False
next reset: 2026-09-25 00:00:00+07:00 = 2026-09-24 17:00:00+00:00
2026-09-25 00:05 WIB  remaining (WIB day) = 15   remaining (UTC day) =  0
2026-09-25 06:59 WIB  remaining (WIB day) = 15   remaining (UTC day) =  0
2026-09-25 07:00 WIB  remaining (WIB day) = 15   remaining (UTC day) = 15

After 15 scans on 25 Sep and a top-up of 5 at 21:00 WIB:
remaining on 25 Sep 21:00: 5
remaining on 26 Sep 00:01: 15

Pro after 100 scans today, remaining: unlimited
Free may add a 2nd monitor? False
Pro may add a 2nd monitor? True
Free history starts: 2026-08-26
Pro history starts: the beginning
```

Read the three lines that begin with `2026-09-25`. At 00:05 and 06:59 WIB the correct book says 15 and the UTC book says 0; at 07:00 both say 15. That is **Figure 8.4**, reproduced by running code. The top-up lines show the other half of the day rule: five extra scans at 21:00 on the 25th bring the remaining count from 0 to 5, and at 00:01 on the 26th the count is back to the plain 15, the top-up gone with its day.

8.10.3 The Java program

Listing 8.2 is the same program in Java. It prints exactly the same lines as the Python version.

Listing 8.2 `Tiers.java`: the same plans, quota book and demonstration in Java, using `java.time`.

```
import java.time.LocalDate;
import java.time.LocalDateTime;
import java.time.LocalTime;
import java.time.ZoneId;
import java.time.ZoneOffset;
import java.time.ZonedDateTime;
import java.time.format.DateTimeFormatter;
import java.util.ArrayList;
import java.util.HashMap;
import java.util.LinkedHashMap;
import java.util.List;
import java.util.Locale;
import java.util.Map;
import java.util.function.Function;

/** Free and Pro plans as data, and a quota calculator whose day
 *  starts at 00:00 WIB (Western Indonesia Time, UTC+7). */
public class Tiers {
    static final ZoneId WIB = ZoneId.of("Asia/Jakarta");
    static final int TERM_DAYS = 30;

    /** scansPerDay and historyDays are null when unlimited. */
    record Plan(String name, Integer scansPerDay, int monitors,
                int monitorEveryDays, Integer historyDays,
                boolean csvExport, int priceIdr) { }

    static final Map<String, Plan> PLANS = new LinkedHashMap<>();
    static {
        PLANS.put("free", new Plan("Free", 15, 1, 7, 30, false, 0));
        PLANS.put("pro", new Plan("Pro", null, 50, 1, null, true, 50_000));
    }

    /** The calendar day in Indonesia on which this instant falls. */
    static LocalDate wibDay(ZonedDateTime t) {
        return t.withZoneSameInstant(WIB).toLocalDate();
    }

    /** The WRONG day for a quota in Indonesia: the UTC calendar day. */
    static LocalDate utcDay(ZonedDateTime t) {
        return t.withZoneSameInstant(ZoneOffset.UTC).toLocalDate();
    }
```

Listing 8.2 (Continued) *Tiers.java: the same plans, quota book and demonstration in Java, using java.time.*

```
    /** The next 00:00 WIB after this instant, as a UTC instant. */
    static ZonedDateTime nextReset(ZonedDateTime t) {
        LocalDate tomorrow = wibDay(t).plusDays(1);
        return ZonedDateTime.of(tomorrow, LocalTime.MIDNIGHT, WIB)
                .withZoneSameInstant(ZoneOffset.UTC);
    }

    /** Counts one account's scans per WIB day, plus same-day top-ups. */
    static class QuotaBook {
        final Plan plan;
        final Function<ZonedDateTime, LocalDate> dayOf;
        final List<ZonedDateTime> scans = new ArrayList<>();
        final Map<LocalDate, Integer> topups = new HashMap<>();

        QuotaBook(String tier) { this(tier, Tiers::wibDay); }

        QuotaBook(String tier, Function<ZonedDateTime, LocalDate> dayOf) {
            this.plan = PLANS.get(tier);
            this.dayOf = dayOf;          // wibDay, or utcDay to show the bug
        }

        Integer allowance(ZonedDateTime now) {
            if (plan.scansPerDay() == null) return null;
            return plan.scansPerDay() + topups.getOrDefault(dayOf.apply(now), 0);
        }

        int used(ZonedDateTime now) {
            LocalDate today = dayOf.apply(now);
            int n = 0;
            for (ZonedDateTime t : scans) if (dayOf.apply(t).equals(today)) n++;
            return n;
        }

        Integer remaining(ZonedDateTime now) {
            Integer limit = allowance(now);
            return limit == null ? null : Math.max(0, limit - used(now));
        }

        boolean tryScan(ZonedDateTime now) {
            Integer left = remaining(now);
            if (left != null && left == 0) return false;
            scans.add(now);
            return true;
        }
```

Listing 8.2 (Continued) *Tiers.java: the same plans, quota book and demonstration in Java, using java.time.*

```
        /** Extra scans bought today are valid for today (WIB) only. */
        void addTopup(ZonedDateTime now, int extra) {
            topups.merge(dayOf.apply(now), extra, Integer::sum);
        }

        boolean canAddMonitor(int activeMonitors) {
            return activeMonitors < plan.monitors();
        }

        /** Oldest visible day of scan history; null means everything. */
        LocalDate historyFrom(ZonedDateTime now) {
            if (plan.historyDays() == null) return null;
            return dayOf.apply(now).minusDays(plan.historyDays());
        }
    }

    static double pricePerDay(Plan p) { return p.priceIdr() / (double) TERM_DAYS; }

    /** Parse "yyyy-MM-dd HH:mm" as a WIB wall-clock time. */
    static ZonedDateTime at(String text) {
        DateTimeFormatter f = DateTimeFormatter.ofPattern("yyyy-MM-dd HH:mm");
        return LocalDateTime.parse(text, f).atZone(WIB);
    }

    static String show(Integer n) { return n == null ? "unlimited" : n.toString(); }

    static String stamp(ZonedDateTime t) {
        return t.format(DateTimeFormatter.ofPattern("yyyy-MM-dd HH:mm:ssxxx"));
    }

    public static void main(String[] args) {
        Locale.setDefault(Locale.US);
        System.out.println(
                "Plan    scans/day  monitors  every  history  CSV  price/30d");
        for (Plan p : PLANS.values()) {
            String hist = p.historyDays() == null ? "full"
                                                  : p.historyDays() + " d";
            System.out.printf("%-7s %9s  %8d  %3d d  %7s  %3s  Rp%,d%n", p.name(),
                    show(p.scansPerDay()), p.monitors(), p.monitorEveryDays(),
                    hist, p.csvExport() ? "yes" : "no", p.priceIdr());
        }
        System.out.printf("Pro costs Rp%,.2f per day%n",
                pricePerDay(PLANS.get("pro")));
```

Listing 8.2 (Continued) *Tiers.java: the same plans, quota book and demonstration in Java, using java.time.*

```
        QuotaBook good = new QuotaBook("free");                 // WIB days
        QuotaBook bad = new QuotaBook("free", Tiers::utcDay);  // UTC days
        ZonedDateTime start = at("2026-09-24 20:00");
        for (int i = 0; i < 15; i++) {
            ZonedDateTime t = start.plusMinutes(14 * i);
            good.tryScan(t);
            bad.tryScan(t);
        }
        ZonedDateTime late = at("2026-09-24 23:45");
        System.out.println("\n15 scans on 24 Sep between 20:00 and 23:16 WIB");
        System.out.println("16th scan at 23:45 WIB allowed? "
                + (good.tryScan(late) ? "True" : "False"));
        ZonedDateTime r = nextReset(late);
        System.out.println("next reset: " + stamp(r.withZoneSameInstant(WIB))
                + " = " + stamp(r));

        for (String clock : new String[] {"2026-09-25 00:05", "2026-09-25 06:59",
                                          "2026-09-25 07:00"}) {
            ZonedDateTime now = at(clock);
            System.out.printf("%s WIB  remaining (WIB day) = %2d"
                    + "   remaining (UTC day) = %2d%n",
                    clock, good.remaining(now), bad.remaining(now));
        }

        ZonedDateTime now = at("2026-09-25 21:00");
        for (int i = 0; i < 15; i++) good.tryScan(now);
        good.addTopup(now, 5);
        System.out.println(
                "\nAfter 15 scans on 25 Sep and a top-up of 5 at 21:00 WIB:");
        System.out.println("remaining on 25 Sep 21:00: " + good.remaining(now));
        System.out.println("remaining on 26 Sep 00:01: "
                + good.remaining(at("2026-09-26 00:01")));

        QuotaBook pro = new QuotaBook("pro");
        for (int i = 0; i < 100; i++) pro.tryScan(now);
        System.out.println("\nPro after 100 scans today, remaining: "
                + show(pro.remaining(now)));
        System.out.println("Free may add a 2nd monitor? "
                + (good.canAddMonitor(1) ? "True" : "False"));
        System.out.println("Pro may add a 2nd monitor? "
                + (pro.canAddMonitor(1) ? "True" : "False"));
        System.out.println("Free history starts: " + good.historyFrom(now));
        LocalDate ph = pro.historyFrom(now);
        System.out.println("Pro history starts: "
                + (ph == null ? "the beginning" : ph.toString()));
    }
```

Listing 8.2 (Continued) *Tiers.java: the same plans, quota book and demonstration in Java, using java.time.*

```
}
```

The Java version uses a `record` for `Plan`, which gives an immutable class with a constructor and accessors in one line. `Integer` rather than `int` is used for the two "maybe unlimited" fields, because only an object type can be `null`. The `dayOf` field has type `Function<ZonedDateTime, LocalDate>`, so the two date functions can be passed as method references, `Tiers::wibDay` and `Tiers::utcDay`. `ZonedDateTime.withZoneSameInstant` is Java's equivalent of Python's `astimezone`: it keeps the instant and changes the clock it is read on [9].

`main` calls `Locale.setDefault(Locale.US)` before formatting numbers. Without it, a computer set to an Indonesian locale would print `Rp50.000` and `Rp1.666,67`, which is correct Indonesian but would no longer match the Python output. The printed booleans are spelled `True` and `False` for the same reason. To compile and run it:

```
javac -d out Tiers.java
java -cp out Tiers
```

The output is identical to the output of **Listing 8.1**, line for line, which is itself a small test: two independent implementations of the same rules agree.

8.11 Lab: The Subscription as a State Machine

The second program models a Pro subscription. We could write it as a handful of Boolean flags (`is_pro`, `is_waiting`, `is_verifying`) and a tangle of `if` statements. Programs written that way tend to reach combinations that should be impossible, such as "Pro and waiting for payment at the same time", and nobody notices until a customer complains. A *state machine* rules those combinations out by construction.

> **DEFINITION.** A *finite state machine* is a model in which a system is always in exactly one of a finite set of *states*, and moves between states only in response to *events*, along *transitions* that are listed in advance. An event that has no transition from the current state is refused.

8.11.1 States, events and transitions

Our subscription has four states:

- `FREE`: no active Pro term and no order in progress.
- `AWAITING_PAYMENT`: a QRIS order for Rp50,000 exists and has not yet been paid.
- `VERIFYING`: the user has reported paying, and the payment is being checked.
- `PRO`: an active 30-day term is running.

And it responds to seven events. `choose_qris` creates an order. `report_paid` says the user has paid. `verify_ok` and `verify_reject` are the outcome of the check. `redeem_profit` pays with PROFIT. `tick` is the passage of time, delivered by a clock: it expires unpaid orders, sends the reminder, and renews or ends the term. **Figure 8.5** draws the machine.

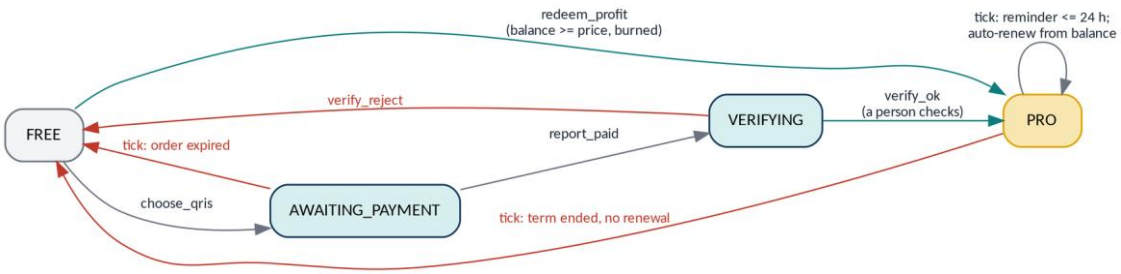


Figure 8.5 The subscription state machine of the lab. Teal arrows lead to Pro, red arrows return to Free, and the loop on Pro is the reminder and the auto-renewal; every state also accepts the clock's tick.

The same information as a table is often more useful to a programmer, because it can be checked mechanically. **Table 8.3** lists, for each state, the events it accepts.

Table 8.3 The transition table of the subscription machine. Any event not listed for the current state is refused with an error.

State	Events accepted	Where they lead
`FREE`	`choose_qris`, `redeem_profit`, `tick`	`AWAITING_PAYMENT`; `PRO` if the balance suffices; no change
`AWAITING_PAYMENT`	`report_paid`, `tick`	`VERIFYING`; `FREE` when the order expires
`VERIFYING`	`verify_ok`, `verify_reject`, `tick`	`PRO`; `FREE`; no change
`PRO`	`tick`	reminder, renewal (stays `PRO`), or `FREE` at the end of the term

Notice what the table forbids. `verify_ok` is accepted only in `VERIFYING`. Once it has moved the subscription to `PRO`, a second `verify_ok` for the same order finds the machine in `PRO`, where the event is not listed, and is refused. A double click by a tired administrator, or a duplicated notification from a payment system, cannot credit two terms. The protection is not an extra `if` added after an incident; it is a consequence of the table.

8.11.2 The Python program

Listing 8.3 implements the machine. `ALLOWED` is **Table 8.3** as a dictionary, and every event method begins by calling `_require`, which raises `IllegalTransition` if the event is not allowed in the current state.

Listing 8.3 `subscription.py`: the subscription state machine, with QRIS orders, verification, PROFIT redemption, reminders and auto-renewal.

```
"""A Pro subscription as a state machine: pay by QRIS (verified by a
person) or redeem PROFIT, 30-day terms, reminder and auto-renew."""
from datetime import datetime, timedelta
from enum import Enum
from zoneinfo import ZoneInfo

WIB = ZoneInfo("Asia/Jakarta")
TERM = timedelta(days=30)
ORDER_TTL = timedelta(hours=1)      # our lab's choice, not Profitina's
REMIND_BEFORE = timedelta(hours=24)
PRICE_IDR = 50_000
MICRO = 1_000_000                   # PROFIT has 6 decimals
PRICE_PROFIT = 50 * MICRO           # lab price: Rp50,000 at Rp1,000/PROFIT


class State(Enum):
    FREE = "FREE"
    AWAITING_PAYMENT = "AWAITING_PAYMENT"
    VERIFYING = "VERIFYING"
    PRO = "PRO"


ALLOWED = {
    State.FREE: {"choose_qris", "redeem_profit", "tick"},
    State.AWAITING_PAYMENT: {"report_paid", "tick"},
    State.VERIFYING: {"verify_ok", "verify_reject", "tick"},
    State.PRO: {"tick"},
}


class IllegalTransition(Exception):
    pass


def profit(micro: int) -> str:
    return f"{micro // MICRO}.{micro % MICRO:06d} PROFIT"
```

Listing 8.3 (Continued) *subscription.py: the subscription state machine, with QRIS orders, verification, PROFIT redemption, reminders and auto-renewal.*

```
class Subscription:
    def __init__(self, balance_micro: int = 0, auto_renew: bool = True):
        self.state = State.FREE
        self.balance = balance_micro     # PROFIT, in micro-units
        self.burned = 0
        self.auto_renew = auto_renew
        self.order_deadline: datetime | None = None
        self.expires: datetime | None = None
        self.reminded = False
        self.log: list[str] = []

    def _require(self, event: str) -> None:
        if event not in ALLOWED[self.state]:
            raise IllegalTransition(f"{event} not allowed in {self.state.value}")

    def _note(self, now: datetime, text: str) -> None:
        stamp = now.astimezone(WIB).strftime("%d %b %H:%M")
        self.log.append(f"{stamp}  {self.state.value:<16} {text}")

    def _start_term(self, base: datetime) -> None:
        self.expires = base + TERM
        self.reminded = False
        self.state = State.PRO

    def choose_qris(self, now: datetime) -> None:
        self._require("choose_qris")
        self.state = State.AWAITING_PAYMENT
        self.order_deadline = now + ORDER_TTL
        self._note(now, f"QRIS order for Rp{PRICE_IDR:,} created")

    def report_paid(self, now: datetime) -> None:
        self._require("report_paid")
        self.state = State.VERIFYING
        self._note(now, "user reports payment; waiting for verification")

    def verify_ok(self, now: datetime) -> None:
        self._require("verify_ok")
        self._start_term(now)            # a new term starts now
        self._note(now, f"payment verified; Pro until {self._exp()}")

    def verify_reject(self, now: datetime, reason: str) -> None:
        self._require("verify_reject")
        self.state = State.FREE
        self._note(now, f"payment rejected: {reason}")
```

Listing 8.3 (Continued) *subscription.py: the subscription state machine, with QRIS orders, verification, PROFIT redemption, reminders and auto-renewal.*

```
    def redeem_profit(self, now: datetime) -> None:
        self._require("redeem_profit")
        if self.balance < PRICE_PROFIT:
            self._note(now, f"redeem refused: balance {profit(self.balance)}")
            return
        self._spend()
        self._start_term(now)
        self._note(now, f"redeemed {profit(PRICE_PROFIT)}; Pro until "
                        f"{self._exp()}")

    def tick(self, now: datetime) -> None:
        """Called by a clock; the only event that makes time pass."""
        self._require("tick")
        if (self.state is State.AWAITING_PAYMENT
                and now >= self.order_deadline):
            self.state = State.FREE
            self._note(now, "QRIS order expired unpaid")
        elif self.state is State.PRO:
            if now >= self.expires:
                self._renew_or_lapse(now)
            elif now >= self.expires - REMIND_BEFORE and not self.reminded:
                self.reminded = True
                will = (self.auto_renew and self.balance >= PRICE_PROFIT)
                self._note(now, f"reminder: ends {self._exp()}, "
                                f"{'will renew' if will else 'will NOT renew'}")

    def _renew_or_lapse(self, now: datetime) -> None:
        if self.auto_renew and self.balance >= PRICE_PROFIT:
            self._spend()
            self._start_term(self.expires)  # extend: no hour lost or gained
            self._note(now, f"auto-renewed from balance; Pro until "
                            f"{self._exp()}")
        else:
            self.state = State.FREE
            self._note(now, "term ended; back to Free")

    def _spend(self) -> None:
        self.balance -= PRICE_PROFIT
        self.burned += PRICE_PROFIT      # spent PROFIT is burned, not kept

    def _exp(self) -> str:
        return self.expires.astimezone(WIB).strftime("%d %b %H:%M")
```

Listing 8.3 (Continued) *subscription.py: the subscription state machine, with QRIS orders, verification, PROFIT redemption, reminders and auto-renewal.*

```
def at(text: str) -> datetime:
    return datetime.strptime(text, "%Y-%m-%d %H:%M").replace(tzinfo=WIB)


def main() -> None:
    a = Subscription()                              # pays with QRIS
    a.choose_qris(at("2026-09-24 10:00"))
    a.report_paid(at("2026-09-24 10:07"))
    a.verify_ok(at("2026-09-24 11:30"))
    for day in ("2026-10-23 09:00", "2026-10-23 12:00", "2026-10-24 11:30"):
        a.tick(at(day))
    print("Account A (QRIS, no PROFIT balance)          times in WIB")
    print("\n".join(a.log))

    b = Subscription(balance_micro=120 * MICRO)     # redeems PROFIT
    b.redeem_profit(at("2026-09-24 08:00"))
    for day in ("2026-10-23 09:00", "2026-10-24 08:00", "2026-11-22 09:00",
                "2026-11-23 08:00"):
        b.tick(at(day))
    print("\nAccount B (120 PROFIT, auto-renew on)")
    print("\n".join(b.log))
    print("balance", profit(b.balance), "| burned", profit(b.burned))

    c = Subscription()
    c.choose_qris(at("2026-09-24 20:00"))
    c.tick(at("2026-09-24 21:01"))
    try:
        c.report_paid(at("2026-09-24 21:05"))
    except IllegalTransition as e:
        c.log.append(f"{'':14}{'':17}refused: {e}")
    print("\nAccount C (order left unpaid)")
    print("\n".join(c.log))


if __name__ == "__main__":
    main()
```

The constants at the top state the lab's assumptions in one place. `TERM` is 30 days and `REMIND_BEFORE` is 24 hours, as on the plan cards. `ORDER_TTL`, the time a QRIS order stays open, is one hour; this is our lab's choice, and the comment says so. PROFIT amounts are kept as whole numbers of micro-PROFIT, because PROFIT has six decimals (Chapter 12): `MICRO` is one million, and `PRICE_PROFIT` is 50 PROFIT. Fifty is what Rp50,000 comes to at the initial reference of Rp1,000 per PROFIT; in the real app, the PROFIT price is shown before the user confirms. Keeping money as integers rather than floating-point numbers is a habit worth forming early: 0.1 + 0.2 is not exactly 0.3 in binary floating point, but 100,000 + 200,000 micro-units is exactly 300,000.

The event methods are short because the table does the heavy lifting. `choose_qris` records a deadline. `report_paid` only changes state. `verify_ok` starts a term *from now*, because a new purchase begins when it is confirmed. `redeem_profit` checks the balance, and if it is enough, spends (and burns) the price and starts a term.

`tick` is the only method that makes time pass. In `AWAITING_PAYMENT` it expires an order whose deadline has come. In `PRO` it either ends the term (renewing it if possible) or, within 24 hours of the end, sends one reminder that says honestly whether the term *will* renew. That honesty is small but real: a reminder that says "your Pro ends tomorrow" to someone whose auto-renew will fail for lack of balance is less useful than one that says "will NOT renew".

`_renew_or_lapse` renews from the balance when auto-renew is on and the balance is enough, and otherwise returns the account to `FREE`. Look at the line `self._start_term(self.expires)`: a renewal starts the new term *at the old expiry*, which is Equation (8.4). Section 8.12 shows what happened when an earlier draft of this method started it at `now` instead.

`main` runs three accounts through the machine. Account A pays with QRIS and has no PROFIT, so its reminder says it will not renew, and it returns to Free after 30 days. Account B starts with 120 PROFIT and auto-renew on; it redeems once, renews once, is warned the second time that it will not renew, and ends with 20 PROFIT left and 100 burned. Account C opens an order, lets it expire, and then tries to report a payment, which the machine refuses. The output:

```
Account A (QRIS, no PROFIT balance)          times in WIB
24 Sep 10:00  AWAITING_PAYMENT QRIS order for Rp50,000 created
24 Sep 10:07  VERIFYING        user reports payment; waiting for verification
24 Sep 11:30  PRO              payment verified; Pro until 24 Oct 11:30
23 Oct 12:00  PRO              reminder: ends 24 Oct 11:30, will NOT renew
24 Oct 11:30  FREE             term ended; back to Free

Account B (120 PROFIT, auto-renew on)
24 Sep 08:00  PRO              redeemed 50.000000 PROFIT; Pro until 24 Oct 08:00
23 Oct 09:00  PRO              reminder: ends 24 Oct 08:00, will renew
24 Oct 08:00  PRO              auto-renewed from balance; Pro until 23 Nov 08:00
22 Nov 09:00  PRO              reminder: ends 23 Nov 08:00, will NOT renew
23 Nov 08:00  FREE             term ended; back to Free
balance 20.000000 PROFIT | burned 100.000000 PROFIT

Account C (order left unpaid)
24 Sep 20:00  AWAITING_PAYMENT QRIS order for Rp50,000 created
24 Sep 21:01  FREE             QRIS order expired unpaid
                               refused: report_paid not allowed in FREE
```

The last line is the machine doing its job. The user reported a payment for an order that no longer exists; the program neither crashes nor quietly accepts it, but refuses with a message that says exactly which rule was broken.

8.11.3 The Java program

Listing 8.4 is the Java version. The state is an `enum`, the transition table is an `EnumMap` from states to sets of event names, and `IllegalTransition` is an unchecked exception. Its output is identical to the Python output above.

Listing 8.4 `Subscription.java`: the same state machine in Java, with an `EnumMap` as the transition table.

```
import java.time.Duration;
import java.time.LocalDateTime;
import java.time.ZoneId;
import java.time.ZonedDateTime;
import java.time.format.DateTimeFormatter;
import java.util.ArrayList;
import java.util.EnumMap;
import java.util.List;
import java.util.Locale;
import java.util.Map;
import java.util.Set;

/** A Pro subscription as a state machine: pay by QRIS (verified by a
 *  person) or redeem PROFIT, 30-day terms, reminder and auto-renew. */
public class Subscription {
    static final ZoneId WIB = ZoneId.of("Asia/Jakarta");
    static final Duration TERM = Duration.ofDays(30);
    static final Duration ORDER_TTL = Duration.ofHours(1); // our lab's choice
    static final Duration REMIND_BEFORE = Duration.ofHours(24);
    static final int PRICE_IDR = 50_000;
    static final long MICRO = 1_000_000L;                  // 6 decimals
    static final long PRICE_PROFIT = 50 * MICRO;  // Rp50,000 at Rp1,000/PROFIT

    enum State { FREE, AWAITING_PAYMENT, VERIFYING, PRO }

    static final Map<State, Set<String>> ALLOWED = new EnumMap<>(State.class);
    static {
        ALLOWED.put(State.FREE, Set.of("choose_qris", "redeem_profit", "tick"));
        ALLOWED.put(State.AWAITING_PAYMENT, Set.of("report_paid", "tick"));
        ALLOWED.put(State.VERIFYING, Set.of("verify_ok", "verify_reject", "tick"));
        ALLOWED.put(State.PRO, Set.of("tick"));
    }

    static class IllegalTransition extends RuntimeException {
        private static final long serialVersionUID = 1L;
        IllegalTransition(String msg) { super(msg); }
    }

    static String profit(long micro) {
        return String.format("%d.%06d PROFIT", micro / MICRO, micro % MICRO);
    }
```

Listing 8.4 (Continued) *Subscription.java: the same state machine in Java, with an EnumMap as the transition table.*

```
    State state = State.FREE;
    long balance;                   // PROFIT, in micro-units
    long burned = 0;
    boolean autoRenew;
    ZonedDateTime orderDeadline;
    ZonedDateTime expires;
    boolean reminded = false;
    final List<String> log = new ArrayList<>();

    Subscription() { this(0, true); }

    Subscription(long balanceMicro, boolean autoRenew) {
        this.balance = balanceMicro;
        this.autoRenew = autoRenew;
    }

    private void require(String event) {
        if (!ALLOWED.get(state).contains(event))
            throw new IllegalTransition(event + " not allowed in " + state);
    }

    private void note(ZonedDateTime now, String text) {
        String stamp = now.withZoneSameInstant(WIB)
                .format(DateTimeFormatter.ofPattern("dd MMM HH:mm", Locale.US));
        log.add(String.format("%s  %-16s %s", stamp, state, text));
    }

    private void startTerm(ZonedDateTime base) {
        expires = base.plus(TERM);
        reminded = false;
        state = State.PRO;
    }

    void chooseQris(ZonedDateTime now) {
        require("choose_qris");
        state = State.AWAITING_PAYMENT;
        orderDeadline = now.plus(ORDER_TTL);
        note(now, String.format(Locale.US, "QRIS order for Rp%,d created",
                PRICE_IDR));
    }
```

Listing 8.4 (Continued) *Subscription.java: the same state machine in Java, with an EnumMap as the transition table.*

```
    void reportPaid(ZonedDateTime now) {
        require("report_paid");
        state = State.VERIFYING;
        note(now, "user reports payment; waiting for verification");
    }

    void verifyOk(ZonedDateTime now) {
        require("verify_ok");
        startTerm(now);                          // a new term starts now
        note(now, "payment verified; Pro until " + exp());
    }

    void verifyReject(ZonedDateTime now, String reason) {
        require("verify_reject");
        state = State.FREE;
        note(now, "payment rejected: " + reason);
    }

    void redeemProfit(ZonedDateTime now) {
        require("redeem_profit");
        if (balance < PRICE_PROFIT) {
            note(now, "redeem refused: balance " + profit(balance));
            return;
        }
        spend();
        startTerm(now);
        note(now, "redeemed " + profit(PRICE_PROFIT) + "; Pro until " + exp());
    }

    /** Called by a clock; the only event that makes time pass. */
    void tick(ZonedDateTime now) {
        require("tick");
        if (state == State.AWAITING_PAYMENT && !now.isBefore(orderDeadline)) {
            state = State.FREE;
            note(now, "QRIS order expired unpaid");
        } else if (state == State.PRO) {
            if (!now.isBefore(expires)) {
                renewOrLapse(now);
            } else if (!now.isBefore(expires.minus(REMIND_BEFORE)) && !reminded) {
                reminded = true;
                boolean will = autoRenew && balance >= PRICE_PROFIT;
                note(now, "reminder: ends " + exp() + ", "
                        + (will ? "will renew" : "will NOT renew"));
            }
        }
    }
```

Listing 8.4 (Continued) *Subscription.java: the same state machine in Java, with an EnumMap as the transition table.*

```
    private void renewOrLapse(ZonedDateTime now) {
        if (autoRenew && balance >= PRICE_PROFIT) {
            spend();
            startTerm(expires);                // extend: no hour lost or gained
            note(now, "auto-renewed from balance; Pro until " + exp());
        } else {
            state = State.FREE;
            note(now, "term ended; back to Free");
        }
    }

    private void spend() {
        balance -= PRICE_PROFIT;
        burned += PRICE_PROFIT;                // spent PROFIT is burned, not kept
    }

    private String exp() {
        return expires.withZoneSameInstant(WIB)
                .format(DateTimeFormatter.ofPattern("dd MMM HH:mm", Locale.US));
    }

    static ZonedDateTime at(String text) {
        DateTimeFormatter f = DateTimeFormatter.ofPattern("yyyy-MM-dd HH:mm");
        return LocalDateTime.parse(text, f).atZone(WIB);
    }

    public static void main(String[] args) {
        Subscription a = new Subscription();              // pays with QRIS
        a.chooseQris(at("2026-09-24 10:00"));
        a.reportPaid(at("2026-09-24 10:07"));
        a.verifyOk(at("2026-09-24 11:30"));
        for (String d : new String[] {"2026-10-23 09:00", "2026-10-23 12:00",
                                      "2026-10-24 11:30"}) a.tick(at(d));
        System.out.println(
                "Account A (QRIS, no PROFIT balance)          times in WIB");
        a.log.forEach(System.out::println);
```

Listing 8.4 (Continued) *Subscription.java: the same state machine in Java, with an EnumMap as the transition table.*

```
        Subscription b = new Subscription(120 * MICRO, true); // redeems PROFIT
        b.redeemProfit(at("2026-09-24 08:00"));
        for (String d : new String[] {"2026-10-23 09:00", "2026-10-24 08:00",
                "2026-11-22 09:00", "2026-11-23 08:00"}) b.tick(at(d));
        System.out.println("\nAccount B (120 PROFIT, auto-renew on)");
        b.log.forEach(System.out::println);
        System.out.println("balance " + profit(b.balance) + " | burned "
                + profit(b.burned));

        Subscription c = new Subscription();
        c.chooseQris(at("2026-09-24 20:00"));
        c.tick(at("2026-09-24 21:01"));
        try {
            c.reportPaid(at("2026-09-24 21:05"));
        } catch (IllegalTransition e) {
            c.log.add(" ".repeat(31) + "refused: " + e.getMessage());
        }
        System.out.println("\nAccount C (order left unpaid)");
        c.log.forEach(System.out::println);
    }
}
```

Two Java details are worth pointing out. `ZonedDateTime` comparisons use `isBefore` and `isAfter`; the condition "now is at or after the deadline" is written `!now.isBefore(orderDeadline)`, which reads awkwardly but is exact, including the case where the two are equal. And `Duration.ofDays(30)` means exactly 30 × 86,400 seconds, which is what we want for a term: WIB has no daylight saving, so a day is always 24 hours there, but using a `Duration` makes the program correct even if it is moved to a zone that does.

To run it:

```
javac -d out Subscription.java
java -cp out Subscription
```

8.12 Lab: Tests That Pin the Rules Down

A business rule that exists only in a program is one careless edit away from changing. Somebody "tidies" `_start_term`, or changes a date function while fixing something else, and a rule that the plan cards promise to every customer silently stops being true. The defense is a set of *tests*: small programs that run the real code on chosen inputs and check that the answers are the ones the rules require. Chapter 9 shows how Profitina's much larger suite gates every release; here we write the tests for our two labs.

8.12.1 What to test

A good way to choose tests is to go through the rules one sentence at a time and ask, for each, "what input would show that this sentence is false?". For the quota calculator:

- The plan table says Free has 15 scans and 1 monitor, Pro unlimited scans, 50 monitors and Rp50,000. A test reads those values from `PLANS`.
- The day starts at 00:00 WIB. A test takes 06:59 WIB on 25 September and checks that its WIB date is the 25th and its UTC date the 24th.
- The sixteenth Free scan of a day is refused, and a scan is allowed again at midnight WIB.
- The UTC-counting book reopens only at 07:00 WIB. This test pins down the *bug*, which may seem odd; it guarantees that the demonstration in **Listing 8.1** keeps demonstrating what the text says it does.
- A top-up is valid for its purchase day only.
- Pro is unlimited, and its 51st monitor is refused.
- Free history starts 30 days back; Pro history is complete.

For the subscription machine the list is longer, because a state machine has more ways to go wrong: the QRIS happy path; an unpaid order expiring at exactly its deadline and not a minute before; a rejected payment; the double `verify_ok`; a balance one micro-PROFIT short of the price; the burn on redemption; the reminder sent once and only within 24 hours; renewal extending from the old expiry; no renewal when auto-renew is off; and a structural check that every state accepts `tick`, so that the clock can never be refused.

8.12.2 The Python tests

Python's standard library includes `unittest` [10]. A test is a method whose name begins with `test_`, inside a class that extends `unittest.TestCase`; the `assert...` methods compare values and report precisely what differed. **Listing 8.5** and **Listing 8.6** are the two test files.

Listing 8.5 `test_tiers.py`: eight tests for the plan table, the WIB day, the daily limit, top-ups, monitors and history.

```
"""Tests for the plan table and the WIB quota day."""
import unittest
from datetime import timedelta, timezone

from tiers import PLANS, QuotaBook, at, next_reset, utc_day, wib_day


class TierTests(unittest.TestCase):
    def test_plan_table_matches_the_plan_cards(self):
        free, pro = PLANS["free"], PLANS["pro"]
        self.assertEqual((free.scans_per_day, free.monitors), (15, 1))
        self.assertEqual((pro.scans_per_day, pro.monitors), (None, 50))
        self.assertEqual(pro.price_idr, 50_000)

    def test_day_starts_at_midnight_wib_not_utc(self):
        t = at("2026-09-25 06:59")               # 24 Sep 23:59 UTC
        self.assertEqual(str(wib_day(t)), "2026-09-25")
        self.assertEqual(str(utc_day(t)), "2026-09-24")

    def test_next_reset_is_17_00_utc(self):
        r = next_reset(at("2026-09-24 23:45"))
        self.assertEqual((r.hour, r.minute, r.tzinfo), (17, 0, timezone.utc))

    def test_sixteenth_free_scan_is_refused(self):
        q = QuotaBook("free")
        t = at("2026-09-24 09:00")
        self.assertTrue(all(q.try_scan(t) for _ in range(15)))
        self.assertFalse(q.try_scan(t))
        self.assertTrue(q.try_scan(t + timedelta(hours=15)))   # 00:00 WIB

    def test_utc_counting_reopens_at_seven_wib(self):
        q = QuotaBook("free", day_of=utc_day)
        for _ in range(15):
            q.try_scan(at("2026-09-24 20:00"))
        self.assertEqual(q.remaining(at("2026-09-25 06:59")), 0)
        self.assertEqual(q.remaining(at("2026-09-25 07:00")), 15)

    def test_topup_is_valid_for_the_purchase_day_only(self):
        q = QuotaBook("free")
        q.add_topup(at("2026-09-24 23:00"), 5)
        self.assertEqual(q.remaining(at("2026-09-24 23:59")), 20)
        self.assertEqual(q.remaining(at("2026-09-25 00:00")), 15)
```

Listing 8.5 (Continued) *test_tiers.py: eight tests for the plan table, the WIB day, the daily limit, top-ups, monitors and history.*

```
    def test_pro_is_unlimited_and_monitors_are_capped(self):
        q = QuotaBook("pro")
        t = at("2026-09-24 12:00")
        self.assertTrue(all(q.try_scan(t) for _ in range(500)))
        self.assertTrue(q.can_add_monitor(49))
        self.assertFalse(q.can_add_monitor(50))

    def test_free_history_is_thirty_days(self):
        q = QuotaBook("free")
        self.assertEqual(str(q.history_from(at("2026-09-24 12:00"))),
                         "2026-08-25")
        self.assertIsNone(QuotaBook("pro").history_from(at("2026-09-24 12:00")))


if __name__ == "__main__":
    unittest.main()
```

Listing 8.6 `test_subscription.py`: ten tests for the subscription state machine.

```
"""Tests for the subscription state machine."""
import unittest
from datetime import timedelta

from subscription import (MICRO, IllegalTransition, State, Subscription,
                          TERM, at)

T0 = at("2026-09-24 10:00")


class SubscriptionTests(unittest.TestCase):
    def test_qris_happy_path(self):
        s = Subscription()
        s.choose_qris(T0)
        s.report_paid(T0 + timedelta(minutes=5))
        s.verify_ok(T0 + timedelta(hours=2))
        self.assertIs(s.state, State.PRO)
        self.assertEqual(s.expires, T0 + timedelta(hours=2) + TERM)

    def test_unpaid_order_expires(self):
        s = Subscription()
        s.choose_qris(T0)
        s.tick(T0 + timedelta(minutes=59))
        self.assertIs(s.state, State.AWAITING_PAYMENT)
        s.tick(T0 + timedelta(hours=1))
        self.assertIs(s.state, State.FREE)

    def test_rejected_payment_returns_to_free(self):
        s = Subscription()
        s.choose_qris(T0)
        s.report_paid(T0)
        s.verify_reject(T0, "amount does not match")
        self.assertIs(s.state, State.FREE)
        self.assertIsNone(s.expires)

    def test_verifying_twice_is_illegal(self):
        s = Subscription()
        s.choose_qris(T0)
        s.report_paid(T0)
        s.verify_ok(T0)
        with self.assertRaises(IllegalTransition):
            s.verify_ok(T0)                    # no double credit
```

Listing 8.6 (Continued) *test_subscription.py: ten tests for the subscription state machine.*

```
    def test_redeem_needs_enough_balance(self):
        s = Subscription(balance_micro=49 * MICRO + 999_999)
        s.redeem_profit(T0)
        self.assertIs(s.state, State.FREE)
        self.assertEqual(s.burned, 0)

    def test_redeem_burns_the_price(self):
        s = Subscription(balance_micro=60 * MICRO)
        s.redeem_profit(T0)
        self.assertIs(s.state, State.PRO)
        self.assertEqual((s.balance, s.burned), (10 * MICRO, 50 * MICRO))

    def test_reminder_once_within_24_hours(self):
        s = Subscription(balance_micro=50 * MICRO)
        s.redeem_profit(T0)
        s.tick(s.expires - timedelta(hours=25))
        self.assertFalse(s.reminded)
        s.tick(s.expires - timedelta(hours=24))
        s.tick(s.expires - timedelta(hours=1))
        self.assertEqual(sum("reminder" in line for line in s.log), 1)

    def test_auto_renew_extends_from_old_expiry(self):
        s = Subscription(balance_micro=100 * MICRO)
        s.redeem_profit(T0)
        first = s.expires
        s.tick(first + timedelta(hours=3))     # the clock ran late
        self.assertIs(s.state, State.PRO)
        self.assertEqual(s.expires, first + TERM)

    def test_no_renew_when_switched_off(self):
        s = Subscription(balance_micro=100 * MICRO, auto_renew=False)
        s.redeem_profit(T0)
        s.tick(s.expires)
        self.assertIs(s.state, State.FREE)
        self.assertEqual(s.balance, 50 * MICRO)

    def test_every_state_accepts_tick(self):
        for state, events in __import__("subscription").ALLOWED.items():
            self.assertIn("tick", events, state)


if __name__ == "__main__":
    unittest.main()
```

Running `python3 -m unittest` in the `python` folder discovers every file whose name begins with `test`, runs all 18 tests, prints one dot per passing test, and ends with `OK`:

```
..................
----------------------------------------------------------------------
Ran 18 tests in 0.005s

OK
```

8.12.3 The test that caught a real bug

The first draft of `subscription.py` had a `_start_term` that looked like this:

```
    def _start_term(self, now):
        # a renewal extends the old expiry, so no day is lost or gained
        base = self.expires if self.expires and self.expires > now else now
        self.expires = base + TERM
```

The comment states the right intention, and for the common case the code does what it says: a renewal processed *before* the old expiry extends from the old expiry. But renewals are processed

by `tick`, and `tick` runs *at or after* the expiry, never before. In that case `self.expires > now` is false, `base` becomes `now`, and every renewal that runs late quietly grants the user the lateness for free. A tick three hours late gives a term three hours too long.

The test `test_auto_renew_extends_from_old_expiry` makes exactly that happen: it redeems, lets the clock run three hours past the expiry, ticks, and compares the new expiry with the old one plus 30 days. The first run failed, with `unittest` reporting an expiry of 23 November 13:00 where 10:00 was expected. The fix was to stop guessing inside `_start_term` and let each caller say where the term starts: `now` for a new purchase, `self.expires` for a renewal. That is the version in **Listing 8.3**.

Three lessons are hidden in this small story. First, the comment was right and the code was wrong, which is common; tests check the code, not the comment. Second, the bug was *in the user's favor*, so no customer would ever have reported it; it would have shown up only as terms that did not add up. Third, the test came from reading a rule, Equation (8.4), and asking what input would break it. That habit found the bug, not cleverness.

8.12.4 The Java tests, with a tiny runner

Java's most popular test framework, JUnit, is a separate library. To keep the labs free of downloads, we write a tiny runner of our own, `MiniTest`, in **Listing 8.7**. It runs a named block of code, prints `ok` or `FAIL`, and ends with a summary; its exit code is 1 when anything failed, which is what Chapter 9's release gate will look at.

Listing 8.7 `MiniTest.java`: a test runner in fifty lines, so the Java tests need nothing but the JDK.

```
import java.util.ArrayList;
import java.util.List;

/** A tiny test runner, so the tests need nothing but the JDK. */
public class MiniTest {
    interface Body { void run() throws Exception; }

    private final List<String> failures = new ArrayList<>();
    private int ran = 0;

    void test(String name, Body body) {
        ran++;
        try {
            body.run();
            System.out.println("ok    " + name);
        } catch (Throwable t) {
            failures.add(name);
            System.out.println("FAIL  " + name + ": " + t.getMessage());
        }
    }

    static void check(boolean condition, String what) {
        if (!condition) throw new AssertionError(what);
    }

    static void equal(Object expected, Object actual) {
        if (expected == null ? actual != null : !expected.equals(actual))
            throw new AssertionError("expected " + expected + " but got " + actual);
    }

    static void raises(Class<? extends Throwable> kind, Body body) {
        try {
            body.run();
        } catch (Throwable t) {
            if (kind.isInstance(t)) return;
            throw new AssertionError("expected " + kind.getSimpleName()
                    + " but got " + t);
        }
        throw new AssertionError("expected " + kind.getSimpleName());
    }
```

Listing 8.7 (Continued) *MiniTest.java: a test runner in fifty lines, so the Java tests need nothing but the JDK.*

```
    /** Prints the summary; the exit code is 1 when anything failed. */
    void finish() {
        System.out.printf("Ran %d tests: %d passed, %d failed%n",
                ran, ran - failures.size(), failures.size());
        if (!failures.isEmpty()) System.exit(1);
    }
}
```

The tests themselves, **Listing 8.8** and **Listing 8.9**, are the same tests as the Python ones, one for one.

Listing 8.8 `TestTiers.java`: the quota tests in Java.

```
import java.time.ZoneOffset;
import java.time.ZonedDateTime;

/** Tests for the plan table and the WIB quota day. */
public class TestTiers {
    public static void main(String[] args) {
        MiniTest t = new MiniTest();
        t.test("plan table matches the plan cards", () -> {
            Tiers.Plan free = Tiers.PLANS.get("free");
            Tiers.Plan pro = Tiers.PLANS.get("pro");
            MiniTest.equal(15, free.scansPerDay());
            MiniTest.equal(1, free.monitors());
            MiniTest.equal(null, pro.scansPerDay());
            MiniTest.equal(50, pro.monitors());
            MiniTest.equal(50_000, pro.priceIdr());
        });
        t.test("day starts at midnight WIB, not UTC", () -> {
            ZonedDateTime x = Tiers.at("2026-09-25 06:59");   // 23:59 UTC
            MiniTest.equal("2026-09-25", Tiers.wibDay(x).toString());
            MiniTest.equal("2026-09-24", Tiers.utcDay(x).toString());
        });
        t.test("next reset is 17:00 UTC", () -> {
            ZonedDateTime r = Tiers.nextReset(Tiers.at("2026-09-24 23:45"));
            MiniTest.equal(17, r.getHour());
            MiniTest.equal(0, r.getMinute());
            MiniTest.equal(ZoneOffset.UTC, r.getZone());
        });
        t.test("sixteenth Free scan is refused", () -> {
            Tiers.QuotaBook q = new Tiers.QuotaBook("free");
            ZonedDateTime x = Tiers.at("2026-09-24 09:00");
            for (int i = 0; i < 15; i++) MiniTest.check(q.tryScan(x), "scan " + i);
            MiniTest.check(!q.tryScan(x), "16th refused");
            MiniTest.check(q.tryScan(x.plusHours(15)), "new day at 00:00 WIB");
        });
        t.test("UTC counting reopens at 07:00 WIB", () -> {
            Tiers.QuotaBook q = new Tiers.QuotaBook("free", Tiers::utcDay);
            for (int i = 0; i < 15; i++) q.tryScan(Tiers.at("2026-09-24 20:00"));
            MiniTest.equal(0, q.remaining(Tiers.at("2026-09-25 06:59")));
            MiniTest.equal(15, q.remaining(Tiers.at("2026-09-25 07:00")));
        });
        t.test("top-up is valid for the purchase day only", () -> {
            Tiers.QuotaBook q = new Tiers.QuotaBook("free");
            q.addTopup(Tiers.at("2026-09-24 23:00"), 5);
            MiniTest.equal(20, q.remaining(Tiers.at("2026-09-24 23:59")));
            MiniTest.equal(15, q.remaining(Tiers.at("2026-09-25 00:00")));
        });
```

Listing 8.8 (Continued) *TestTiers.java: the quota tests in Java.*

```
        t.test("Pro is unlimited and monitors are capped", () -> {
            Tiers.QuotaBook q = new Tiers.QuotaBook("pro");
            ZonedDateTime x = Tiers.at("2026-09-24 12:00");
            for (int i = 0; i < 500; i++) MiniTest.check(q.tryScan(x), "scan");
            MiniTest.check(q.canAddMonitor(49), "49 -> 50 allowed");
            MiniTest.check(!q.canAddMonitor(50), "50 -> 51 refused");
        });
        t.test("Free history is thirty days", () -> {
            ZonedDateTime x = Tiers.at("2026-09-24 12:00");
            MiniTest.equal("2026-08-25",
                    new Tiers.QuotaBook("free").historyFrom(x).toString());
            MiniTest.equal(null, new Tiers.QuotaBook("pro").historyFrom(x));
        });
        t.finish();
    }
}
```

Listing 8.9 `TestSubscription.java`: the subscription tests in Java.

```
import java.time.ZonedDateTime;
import java.util.Set;

/** Tests for the subscription state machine. */
public class TestSubscription {
    static final ZonedDateTime T0 = Subscription.at("2026-09-24 10:00");
    static final long M = Subscription.MICRO;

    public static void main(String[] args) {
        MiniTest t = new MiniTest();
        t.test("QRIS happy path", () -> {
            Subscription s = new Subscription();
            s.chooseQris(T0);
            s.reportPaid(T0.plusMinutes(5));
            s.verifyOk(T0.plusHours(2));
            MiniTest.equal(Subscription.State.PRO, s.state);
            MiniTest.equal(T0.plusHours(2).plus(Subscription.TERM), s.expires);
        });
        t.test("unpaid order expires", () -> {
            Subscription s = new Subscription();
            s.chooseQris(T0);
            s.tick(T0.plusMinutes(59));
            MiniTest.equal(Subscription.State.AWAITING_PAYMENT, s.state);
            s.tick(T0.plusHours(1));
            MiniTest.equal(Subscription.State.FREE, s.state);
        });
        t.test("rejected payment returns to Free", () -> {
            Subscription s = new Subscription();
            s.chooseQris(T0);
            s.reportPaid(T0);
            s.verifyReject(T0, "amount does not match");
            MiniTest.equal(Subscription.State.FREE, s.state);
            MiniTest.equal(null, s.expires);
        });
        t.test("verifying twice is illegal", () -> {
            Subscription s = new Subscription();
            s.chooseQris(T0);
            s.reportPaid(T0);
            s.verifyOk(T0);
            MiniTest.raises(Subscription.IllegalTransition.class,
                    () -> s.verifyOk(T0));             // no double credit
        });
        t.test("redeem needs enough balance", () -> {
            Subscription s = new Subscription(49 * M + 999_999, true);
            s.redeemProfit(T0);
            MiniTest.equal(Subscription.State.FREE, s.state);
```

Listing 8.9 (Continued) *TestSubscription.java: the subscription tests in Java.*

```
            MiniTest.equal(0L, s.burned);
        });
        t.test("redeem burns the price", () -> {
            Subscription s = new Subscription(60 * M, true);
            s.redeemProfit(T0);
            MiniTest.equal(Subscription.State.PRO, s.state);
            MiniTest.equal(10 * M, s.balance);
            MiniTest.equal(50 * M, s.burned);
        });
        t.test("reminder once within 24 hours", () -> {
            Subscription s = new Subscription(50 * M, true);
            s.redeemProfit(T0);
            s.tick(s.expires.minusHours(25));
            MiniTest.check(!s.reminded, "too early for a reminder");
            s.tick(s.expires.minusHours(24));
            s.tick(s.expires.minusHours(1));
            long n = s.log.stream().filter(l -> l.contains("reminder")).count();
            MiniTest.equal(1L, n);
        });
        t.test("auto-renew extends from old expiry", () -> {
            Subscription s = new Subscription(100 * M, true);
            s.redeemProfit(T0);
            ZonedDateTime first = s.expires;
            s.tick(first.plusHours(3));                // the clock ran late
            MiniTest.equal(Subscription.State.PRO, s.state);
            MiniTest.equal(first.plus(Subscription.TERM), s.expires);
        });
        t.test("no renew when switched off", () -> {
            Subscription s = new Subscription(100 * M, false);
            s.redeemProfit(T0);
            s.tick(s.expires);
            MiniTest.equal(Subscription.State.FREE, s.state);
            MiniTest.equal(50 * M, s.balance);
        });
        t.test("every state accepts tick", () -> {
            for (Set<String> events : Subscription.ALLOWED.values())
                MiniTest.check(events.contains("tick"), "tick missing");
        });
        t.finish();
    }
}
```

Compile everything and run both test classes:

```
javac -d out *.java
java -cp out TestTiers
java -cp out TestSubscription
```

```
ok    plan table matches the plan cards
ok    day starts at midnight WIB, not UTC
ok    next reset is 17:00 UTC
ok    sixteenth Free scan is refused
ok    UTC counting reopens at 07:00 WIB
ok    top-up is valid for the purchase day only
ok    Pro is unlimited and monitors are capped
ok    Free history is thirty days
Ran 8 tests: 8 passed, 0 failed
```

```
ok    QRIS happy path
ok    unpaid order expires
ok    rejected payment returns to Free
ok    verifying twice is illegal
ok    redeem needs enough balance
ok    redeem burns the price
ok    reminder once within 24 hours
ok    auto-renew extends from old expiry
ok    no renew when switched off
ok    every state accepts tick
Ran 10 tests: 10 passed, 0 failed
```

Eighteen tests in each language, all passing. Try changing `REMIND_BEFORE` to 12 hours in either language and running the tests again: the reminder test fails, and it names the rule you broke.

8.13 Profitina in Practice

Everything in this chapter maps onto the live product as follows, in the September 2026 release.

- **Two tiers.** Free and Pro, with exactly the limits of **Table 8.1**. The plan cards are the single place where these limits are stated; the book copies them from there.
- **Price and payment.** Pro is Rp50,000 for 30 days, paid via QRIS from any participating bank or e-wallet, or by redeeming PROFIT, whose price the app shows before confirmation. Redeemed PROFIT is burned.
- **Renewal.** Auto-renew from the PROFIT balance, with a reminder at most 24 hours before expiry.
- **Agency.** Removed on 6–7 August 2026 for simplicity, after its three features had been shown to work.
- **No ads.** A permanent commitment, not a phase.
- **Measurement.** The August 2026 analysis found zero PROFIT purchases across production history, and the team drew from it the rule “measure interest before building payment”. PROFIT purchases are verified by a person; payment automation waits for evidence of demand.
- **Daily quotas.** Every daily limit, from the 15 Free scans to the PROFIT earning caps, resets at 00:00 WIB. Scan top-ups bought with PROFIT are valid for the purchase day only.
- **Free forever.** Personal-data rights, safety features and account recovery are never part of any tier.

The labs of Sections 8.10 to 8.12 reproduce none of Profitina’s code or configuration. They teach the ideas with the public numbers, on examples that could belong to any small product, including Profitina Laundry, the teaching case study of this book series. A laundry that sold a monthly “express washing” plan would face precisely the same questions: what is included, when does the day start, what happens when a renewal is late, and how do we stop a payment being credited twice.

8.14 Common Misunderstandings

> **MISUNDERSTANDING 1: “Free is a trial.”** It is not. Free has no end date and includes the full scan, content and technical tools. It limits volume and continuity, not quality.
>
> **MISUNDERSTANDING 2: “Pro users get better scores.”** They do not. Both tiers are measured by the same engine. Pro buys capacity: more scans, more and more frequent monitors, longer history.

MISUNDERSTANDING 3: "Agency was removed because it did not work." All three Agency features had been shown to work. The tier was removed for simplicity: fewer combinations to build, test, document and explain, and an easier decision for customers.

MISUNDERSTANDING 4: "PROFIT is the main source of revenue." It is not. PROFIT is a utility coin that users mainly earn by contributing. The subscription is the business model; and the August 2026 finding of zero PROFIT purchases shows that nobody should plan a budget around PROFIT sales.

MISUNDERSTANDING 5: "A daily quota resets at midnight, so the computer's date is fine." The computer's date is usually UTC, whose midnight is 07:00 WIB. A daily limit must be counted on the users' calendar, which for Profitina is `Asia/Jakarta`.

MISUNDERSTANDING 6: "Manual payment verification is unprofessional." At low volume it is the professional choice: cheap to build, easy to audit, and it produces real evidence of demand. Automation is the right investment once there are purchases to automate.

MISUNDERSTANDING 7: "No ads just means a cleaner screen." It removes a conflict of interest between the platform's revenue and the user's success, protects the privacy stance of Chapter 11, and keeps the platform's advice credible.

8.15 Chapter Summary

- A business model states what value is created, for whom, who pays and how, and what it costs. Freemium revenue follows Equation (8.1), $R = N \cdot c \cdot P$; typical conversion rates are 2% to 5% [1].
- Indonesia in 2026 has 235.26 million internet users, a penetration of 81.72% [3], and QRIS has 65.77 million users and 44.86 million merchants [5], with 12.55 billion transactions in the first half of 2026 [6]. A price in Rupiah paid by QRIS meets people where they already pay.
- Profitina has two tiers. Free: 15 scans a day, one weekly monitor, 30 days of history, full tools. Pro: unlimited scans, 50 daily monitors with alerts, full history and CSV, Rp50,000 per 30 days (about Rp1,667 a day by Equation (8.2)), paid by QRIS or by redeeming PROFIT, with auto-renew from balance and a reminder at most 24 hours before expiry.
- The Agency tier was removed on 6–7 August 2026 for simplicity, though its features worked; Equation (8.3) shows how each tier multiplies what must be built and tested.
- No ads, permanently: a structural choice that aligns incentives, protects privacy and keeps advice credible.
- The August 2026 finding of zero PROFIT purchases produced the rule "measure interest before building payment", a ladder from observation to manual offer to counted purchases to automation.
- Daily quotas reset at 00:00 WIB. The WIB day of an instant is Equation (8.6); using the UTC date (Equation (8.5)) resets at 07:00 WIB instead. Remaining scans are Equation (8.7).
- The labs model plans as data, quotas per WIB day, and a subscription as a finite state machine whose transition table makes double crediting impossible. A test caught a renewal bug that violated Equation (8.4).

8.16 Exercises

Exercise 8.1. Using Equation (8.1), compute the 30-day revenue of a tool with $N = 5{,}000$ active accounts, $c = 2\%$ and $P = \text{Rp}50{,}000$. What conversion rate would be needed to reach

Rp10,000,000 with the same N and P? Is that rate plausible according to the range in Section 8.2.1?

Exercise 8.2. (a) Recompute Equation (8.2). (b) A Pro user runs n scans a day. For which values of n does one scan cost less than Rp100? (c) Why is "cost per scan" a misleading way to judge Pro for a user whose main need is daily monitoring?

Exercise 8.3. Using Equation (8.5) and Equation (8.6), compute d_{UTC} and d_{WIB} for the instants 25 September 2026 06:59 WIB ($t = 1{,}790{,}294{,}340$) and 25 September 2026 07:00 WIB ($t = 1{,}790{,}294{,}400$). Explain how your answers match the output of **Listing 8.1**.

Exercise 8.4. Write a function `seconds_until_reset(now)` (Java: `secondsUntilReset`) that returns how many seconds remain until the next 00:00 WIB, using `next_reset` from **Listing 8.1**. Print it for 23:45 WIB on 24 September, 00:00 WIB on 25 September, and 06:59 WIB on 25 September. What should the answer be at exactly midnight, and why?

Exercise 8.5. Add an event `cancel_order` that lets a user abandon a QRIS order before paying. (a) In which state or states should it be allowed, and where should it lead? (b) Write the change to `ALLOWED` and the new method. (c) Write one test that shows it works and one that shows it is refused in `VERIFYING`. Explain why it must be refused there.

Exercise 8.6. Suppose the server was off for 40 days and the renewal `tick` runs 40 days after the expiry of a term, with enough balance to renew. (a) What end date does Equation (8.4) give for the new term, relative to the moment the renewal runs? (b) Propose a fairer rule for renewals that run more than a whole term late, and implement it as a function `fair_base(old_expiry, now)` (Java: `fairBase`) that returns where the renewed term should start. Print both rules for renewals 0, 3, 29 and 40 days late.

Exercise 8.7. Using Equation (8.3) and the count $T(T-1)$ of directed tier transitions, tabulate combinations and transitions for $T = 1,2,3,4$ tiers with $F = 12$ tier-dependent features. By how much did removing Agency reduce each?

Exercise 8.8. A user asks for "white-label PDF reports with my agency's logo". Following **Figure 8.3**, describe concretely what you would do at each of the four steps before building the feature, and what evidence would make you climb to the next step.

Exercise 8.9. In at most 150 words, explain to a small business owner why a GEO platform that promised "no ads, ever" is more trustworthy than one supported by advertising. Use the three arguments of Section 8.6.

Exercise 8.10. Write a test (in Python or Java) showing that a top-up bought at 23:59 WIB is no longer available at 00:00 WIB the next day, and another showing that the same top-up bought in a `QuotaBook` that counts UTC days *would* still be available at 00:00 WIB. Which test describes the rule, and which describes the bug?

8.17 Solutions to the Exercises

Solution 8.1. $R = 5{,}000 \times 0.02 \times 50{,}000 = \text{Rp}5{,}000{,}000$. To reach Rp10,000,000 we need $c = R/(N \cdot P) = 10{,}000{,}000/(5{,}000 \times 50{,}000) = 0.04$, that is, 4%. This lies inside the 2% to 5% range that most freemium companies reach [1], but toward its upper end; a young product should not assume it without evidence, which is exactly the point of Section 8.8.

Solution 8.2. (a) $50{,}000/30 = 1{,}666.\overline{6}$, so about Rp1,666.67 a day, as the first lines of **Listing 8.1**'s output print. (b) The cost per scan is $p/n < 100$ exactly when $n > p/100 = 16.67$, so from $n = 17$ scans a day upward. (c) A monitoring user may run few manual scans but depend on daily re-scans of up to fifty pages, alerts and full history. None of that appears in "cost per scan", so the measure understates Pro's value for exactly the users whose need is continuity.

Solution 8.3. For $t = 1{,}790{,}294{,}340$: $t/86{,}400 = 20{,}720.998\ldots$, so $d_{\text{UTC}} = 20{,}720$ (24 September); $(t + 25{,}200)/86{,}400 = 20{,}721.29\ldots$, so $d_{\text{WIB}} = 20{,}721$ (25 September). For $t = 1{,}790{,}294{,}400$: $t/86{,}400 = 20{,}721.000$, so $d_{\text{UTC}} = 20{,}721$, and $d_{\text{WIB}} = \lfloor 20{,}721.29\ldots \rfloor = 20{,}721$. At 06:59 WIB the two day numbers differ, so the UTC book still counts the 15 scans of the 24th and prints 0 remaining; at 07:00 WIB they agree, and both books print 15. That is exactly the pair of lines `06:59` and `07:00` in the output of **Listing 8.1**.

Solution 8.4. **Listing 8.10** and **Listing 8.11** subtract the current instant from `next_reset`.

Listing 8.10 `until_reset.py`: seconds until the next 00:00 WIB.

```python
"""Exercise 8.4: how long until the daily quota resets (00:00 WIB)?"""
from tiers import at, next_reset


def seconds_until_reset(now) -> int:
    return int((next_reset(now) - now).total_seconds())


for clock in ("2026-09-24 23:45", "2026-09-25 00:00", "2026-09-25 06:59"):
    s = seconds_until_reset(at(clock))
    print(f"{clock} WIB: {s:>5} s = {s // 3600:>2} h {s % 3600 // 60:>2} min")
```

Listing 8.11 `UntilReset.java`: the same computation with `Duration.between`.

```java
import java.time.Duration;
import java.time.ZonedDateTime;

/** Exercise 8.4: how long until the daily quota resets (00:00 WIB)? */
public class UntilReset {
    static long secondsUntilReset(ZonedDateTime now) {
        return Duration.between(now, Tiers.nextReset(now)).getSeconds();
    }

    public static void main(String[] args) {
        for (String clock : new String[] {"2026-09-24 23:45", "2026-09-25 00:00",
                                          "2026-09-25 06:59"}) {
            long s = secondsUntilReset(Tiers.at(clock));
            System.out.printf("%s WIB: %5d s = %2d h %2d min%n",
                    clock, s, s / 3600, s % 3600 / 60);
        }
    }
}
```

Both print:

```
2026-09-24 23:45 WIB:   900 s =  0 h 15 min
2026-09-25 00:00 WIB: 86400 s = 24 h  0 min
2026-09-25 06:59 WIB: 61260 s = 17 h  1 min
```

At exactly 00:00 WIB the answer is a full day, 86,400 seconds, because `next_reset` returns the *next* midnight after the date of `now`, and at midnight the current WIB day has just begun. Returning 0 would suggest a reset is still pending when it has in fact just happened.

Solution 8.5. (a) Only in `AWAITING_PAYMENT`, leading to `FREE`. It must be refused in `VERIFYING`, because at that point the user has said they paid: if a payment really arrived, cancelling would lose track of money that exists. A payment under review must end in `verify_ok` or `verify_reject`, decided by the person checking it. (b) In `ALLOWED`, change the entry for `AWAITING_PAYMENT` to `{"report_paid", "cancel_order", "tick"}`, and add the method

```
    def cancel_order(self, now):
        self._require("cancel_order")
        self.state = State.FREE
        self._note(now, "QRIS order cancelled by the user")
```

(c) Two tests:

```
    def test_cancel_before_paying(self):
        s = Subscription()
        s.choose_qris(T0)
        s.cancel_order(T0)
        self.assertIs(s.state, State.FREE)

    def test_cancel_refused_while_verifying(self):
        s = Subscription()
        s.choose_qris(T0)
        s.report_paid(T0)
        with self.assertRaises(IllegalTransition):
            s.cancel_order(T0)
```

The Java version adds `"cancel_order"` to the `Set.of(...)` for `AWAITING_PAYMENT` and a method `cancelOrder` with the same three lines.

Solution 8.6. (a) Equation (8.4) gives old expiry plus 30 days, which is 10 days *before* the moment the renewal runs: the user is charged for a term that is already over, and the next tick ends it at once. (b) A fair rule: if the renewal runs less than a full term late, extend from the old expiry as before (the user had service or a short gap caused by the platform); if it runs a full term or more late, start the new term from `now`, because the service was not delivered for that period and the user should not pay for it. **Listing 8.12** and **Listing 8.13** implement and compare the two rules.

Listing 8.12 `late_renewal.py`: the old rule and a fair rule for very late renewals.

```
"""Exercise 8.6: a renewal that runs more than a whole term late."""
from datetime import timedelta

from subscription import TERM, at


def fair_base(old_expiry, now):
    """Where should the renewed term start?"""
    # up to one term late: extend; longer: start from now
    return old_expiry if now - old_expiry < TERM else now


expiry = at("2026-10-01 08:00")
for days in (0, 3, 29, 40):
    now = expiry + timedelta(days=days)
    old_rule = (expiry + TERM).date()
    fair_rule = (fair_base(expiry, now) + TERM).date()
    print(f"renewal {days:>2} days late: old rule ends {old_rule}, "
          f"fair rule ends {fair_rule}")
```

Listing 8.13 `LateRenewal.java`: the same comparison in Java.

```java
import java.time.Duration;
import java.time.ZonedDateTime;

/** Exercise 8.6: a renewal that runs more than a whole term late. */
public class LateRenewal {
    /** Where should the renewed term start? */
    static ZonedDateTime fairBase(ZonedDateTime oldExpiry, ZonedDateTime now) {
        Duration late = Duration.between(oldExpiry, now);
        // up to one term late: extend; longer: start from now
        return late.compareTo(Subscription.TERM) < 0 ? oldExpiry : now;
    }

    public static void main(String[] args) {
        ZonedDateTime expiry = Subscription.at("2026-10-01 08:00");
        for (int days : new int[] {0, 3, 29, 40}) {
            ZonedDateTime now = expiry.plusDays(days);
            System.out.printf("renewal %2d days late: old rule ends %s, "
                    + "fair rule ends %s%n", days,
                    expiry.plus(Subscription.TERM).toLocalDate(),
                    fairBase(expiry, now).plus(Subscription.TERM).toLocalDate());
        }
    }
}
```

```
renewal  0 days late: old rule ends 2026-10-31, fair rule ends 2026-10-31
renewal  3 days late: old rule ends 2026-10-31, fair rule ends 2026-10-31
renewal 29 days late: old rule ends 2026-10-31, fair rule ends 2026-10-31
renewal 40 days late: old rule ends 2026-10-31, fair rule ends 2026-12-10
```

To use the rule in the machine, replace the argument of `_start_term` in `_renew_or_lapse` with `fair_base(self.expires, now)`, and add a test with a tick 40 days late. Other fair rules exist; the important thing is to choose one deliberately and pin it with a test.

Solution 8.7.

Table 8.4 Feature-and-tier combinations and directed tier transitions for $F = 12$.

T	Combinations $C=T\times12$	Transitions $T(T-1)$
1	12	0
2	24	2
3	36	6
4	48	12

Going from three tiers to two, as in August 2026, removed 12 combinations (a third) and 4 of the 6 transitions, as **Table 8.4** shows.

Solution 8.8. *Observe:* count how many users ask for branded reports, and how many currently export CSV and build their own reports; add a line under the export button, “Would you like this as a branded PDF?”, and count clicks. *Offer by hand:* for the users who asked, produce the branded PDF manually once, for a stated price via a QRIS order verified by a person. *Count purchases:* track paid requests over several weeks, and whether the same users come back. *Automate:* only if repeated paid requests appear, and the manual work becomes a bottleneck, build the feature, and even then add it to a plan card only when it is live. Evidence to climb: clicks and requests justify the manual offer; payments justify automation; compliments justify nothing.

Solution 8.9. A sample answer: “An ad-supported tool earns more the longer you stay and the more you are shown, so it is paid to keep you busy, not to solve your problem. A subscription earns only when you are satisfied enough to renew, so it is paid to make your visibility better. Advertising also needs data about you to target ads; a platform that refuses ads can keep your

brand data on its own server and ask for almost nothing. And a platform that advises you on trustworthiness should itself look trustworthy, which is hard to do while selling banner space to anyone who pays." (106 words.)

Solution 8.10. In Python, using the names of **Listing 8.5**:

```
def test_topup_ends_at_wib_midnight(self):
    q = QuotaBook("free")
    q.add_topup(at("2026-09-24 23:59"), 5)
    self.assertEqual(q.remaining(at("2026-09-25 00:00")), 15)

def test_utc_book_keeps_topup_after_wib_midnight(self):
    q = QuotaBook("free", day_of=utc_day)
    q.add_topup(at("2026-09-24 23:59"), 5)
    self.assertEqual(q.remaining(at("2026-09-25 00:00")), 20)
```

In the UTC book, 23:59 WIB on the 24th and 00:00 WIB on the 25th are both on 24 September UTC, so the top-up is still counted. The first test describes the rule (a top-up is valid for its WIB purchase day only); the second describes the bug. Keeping the second as a test is worthwhile only while the program deliberately demonstrates the bug, as **Listing 8.1** does.

References

[1] V. Kumar, "Making 'freemium' work," *Harvard Business Review*, vol. 92, no. 5, pp. 27–29, May 2014. [Online]. Available: https://hbr.org/2014/05/making-freemium-work

[2] Profitina, *Profitina Whitepaper, Public Version v24*, Sep. 2026. [Online]. Available: https://app.profitina.com/static/whitepaper/

[3] ANTARA, "Survei APJII: penetrasi internet di Indonesia 2026 capai 81,7 persen," May 21, 2026. [Online]. Available: https://www.antaranews.com/berita/5576225/survei-apjii-penetrasi-internet-di-indonesia-2026-capai-817-persen

[4] CNN Indonesia, "Survei: pengguna internet Indonesia tumbuh 6 juta di 2026," May 21, 2026. [Online]. Available: https://www.cnnindonesia.com/teknologi/20260521122201-213-1360738/survei-pengguna-internet-indonesia-tumbuh-6-juta-di-2026

[5] ANTARA News, "QRIS reaches 65.77 mln users, accepted by 44.86 mln merchants: BI," Aug. 21, 2026. [Online]. Available: https://en.antaranews.com/news/428168/qris-reaches-6577-mln-users-accepted-by-4486-mln-merchants-bi

[6] Kontan, "QRIS tembus 12,55 miliar transaksi pada semester I 2026, pengguna capai 66 juta," Sep. 18, 2026. [Online]. Available: https://keuangan.kontan.co.id/news/qris-tembus-1255-miliar-transaksi-pada-semester-i-2026-pengguna-capai-66-juta

[7] E. Ries, *The Lean Startup: How Today's Entrepreneurs Use Continuous Innovation to Create Radically Successful Businesses*. New York, NY, USA: Crown Business, 2011.

[8] Python Software Foundation, "zoneinfo: IANA time zone support," *Python 3.14 Documentation*. [Online]. Available: https://docs.python.org/3/library/zoneinfo.html

[9] Oracle, "Class ZonedDateTime," *Java SE 25 & JDK 25 API Specification*, 2025. [Online]. Available: https://docs.oracle.com/en/java/javase/25/docs/api/java.base/java/time/ZonedDateTime.html

[10] Python Software Foundation, "unittest: Unit testing framework," *Python 3.14 Documentation*. [Online]. Available: https://docs.python.org/3/library/unittest.html

Chapter 9

Architecture, Reliability, and the Release Gate

By the end of this chapter you will be able to draw Profitina's architecture at the level of boxes and arrows, and explain what each box does and why it is there: a single Windows 11 machine running a FastAPI application under Uvicorn, kept alive as a Windows service by NSSM, storing its data in PostgreSQL, pushing live updates with Server-Sent Events, thinking with local AI models in two lanes, and reaching the internet only through an outbound Cloudflare Tunnel, with a WordPress marketing site beside it. You will be able to say what "reliable" means in numbers, and why a small system on one machine can be reliable if it is careful. You will understand how a suite of more than fifteen hundred automated tests acts as a *release gate*, and why Profitina's release declares success only after the running service answers its health check with the new version number. You will follow a real performance investigation, in which switching channels in the community feed went from 868 milliseconds to 11, and learn the engineering habits that made the fix possible and keep it fixed. Finally, in lab G4, you will build three small programs in Python and in Java: a scheduler that re-scans monitored pages at a per-tier cadence using a priority queue, a health-check endpoint, and a release-gate script that refuses to "deploy" when the tests fail and rolls back when the new version does not come up.

9.1 Where We Left Off

Chapter 8 explained who pays for Profitina and why. This chapter explains what they are paying *for*, underneath the screens: the machine, the software that runs on it, and the habits that keep it running.

It is tempting to think that architecture is about big diagrams and famous technologies. For a young product it is mostly about three plainer questions. *Where does each piece of work happen? What happens when something breaks? How do we change the system without breaking it?* The first question gives the architecture in Section 9.3. The second gives the reliability of Sections 9.2 and 9.4. The third, which is the one most teams answer last and regret, gives the release gate of Sections 9.5 and 9.6.

We will describe Profitina at the level of boxes: what each component is and why it was chosen. We will not describe how the scoring inside those boxes works, and we will not print anything that would help someone attack the system, such as machine names, folder paths or the details of how the machine is reached. Those are not needed to learn the lessons, and a good engineer learns early to leave them out of any document that will be read by many people.

The chapter ends with lab G4, which rebuilds the three ideas that matter most for daily operation, a scheduler, a health check and a release gate, as small programs you can run on your own computer.

9.2 Reliability, in Plain Words

Before we look at the architecture, we need a way to talk about how well it works. Everyone agrees that a service should be "reliable"; fewer people can say what number they mean.

9.2.1 Availability

DEFINITION. The *availability* A of a service over a period is the fraction of that period during which the service did its job. It is often written as a percentage, and informally counted in "nines": 99.9% is "three nines".

A useful way to think about availability is through two averages. The *mean time between failures* (MTBF) is how long, on average, the service runs before something stops it. The *mean time to repair* (MTTR) is how long, on average, it takes to bring it back. Over a long period, the service alternates between an up-time of about MTBF and a down-time of about MTTR, so

$$A = \frac{\text{MTBF}}{\text{MTBF} + \text{MTTR}} \tag{9.1}$$

Equation (9.1) has a lesson that surprises many beginners. There are two ways to raise availability, and the second is often much cheaper than the first. You can make failures rarer (raise MTBF), which usually means expensive redundancy. Or you can make recovery faster (lower MTTR), which often means simple things: a service manager that restarts a crashed process within seconds, a backup that can be restored in minutes, a release process that can roll back without thinking. A service that fails once a week but recovers in ten seconds has an availability above 99.998%.

9.2.2 What the nines mean in minutes

The down-time D allowed in a period of length T at availability A is

$$D = (1 - A) \cdot T \tag{9.2}$$

Table 9.1 applies Equation (9.2) to a 30-day month and a 365-day year.

Table 9.1 Allowed down-time at common availability targets, from Equation (9.2).

Availability	Down-time per 30 days	Down-time per year
99%	7.2 hours	87.6 hours
99.9%	43.2 minutes	8.76 hours
99.99%	4.32 minutes	52.6 minutes
99.999%	25.9 seconds	5.26 minutes

EXAMPLE. A small laundry runs an online ordering service with a 99.9% target. By Equation (9.2), in 30 days it may be down for $(1 - 0.999) \times 30 \times 24 \times 60 = 43.2$ minutes. If one careless release takes the service down for an hour, the whole month's allowance is gone in one afternoon. If instead every release goes through a careful gate and interrupts the service for only five seconds during the restart, eight releases a month use forty seconds of that allowance. This is why the chapter pays so much attention to how releases are made: for a small system, the release is the event that most often threatens availability.

Each extra nine costs roughly ten times the effort of the one before it. Google's site reliability engineers, who wrote the best-known book on the subject, argue that the right target is not "as high as possible" but "as high as users need", and that a service which is more reliable than its

users need is wasting effort that could have gone into features [1]. For a young GEO platform whose users scan their sites and read their fix lists, a few minutes of down-time during a careful release at a quiet hour is acceptable; losing a user's scan history is not. Reliability is about protecting what matters most, not about a single number.

9.2.3 Reliability includes being right

A second, less obvious point: a service that answers quickly with a wrong answer is not reliable. Profitina's engineering rules include one that sounds almost moral: **errors must be loud**. In a mid-September 2026 review, the team found a place where a failed database count, wrapped in a `try`/`except` that returned zero, would have shown a user a convincing "0" instead of an error. Nobody would ever have reported it, because a zero looks like a normal answer. The fix was to let the error surface. A silent wrong answer is worse than a visible failure, because it cannot be noticed, and what cannot be noticed cannot be repaired.

The same rule appears in the product itself: when no local AI model is available, the app says so rather than quietly serving a template (Chapter 6). Honesty is a reliability property.

9.3 The Architecture at Box Level

Figure 9.1 is the whole of Profitina's production system on one page, as it stands in the September 2026 release.

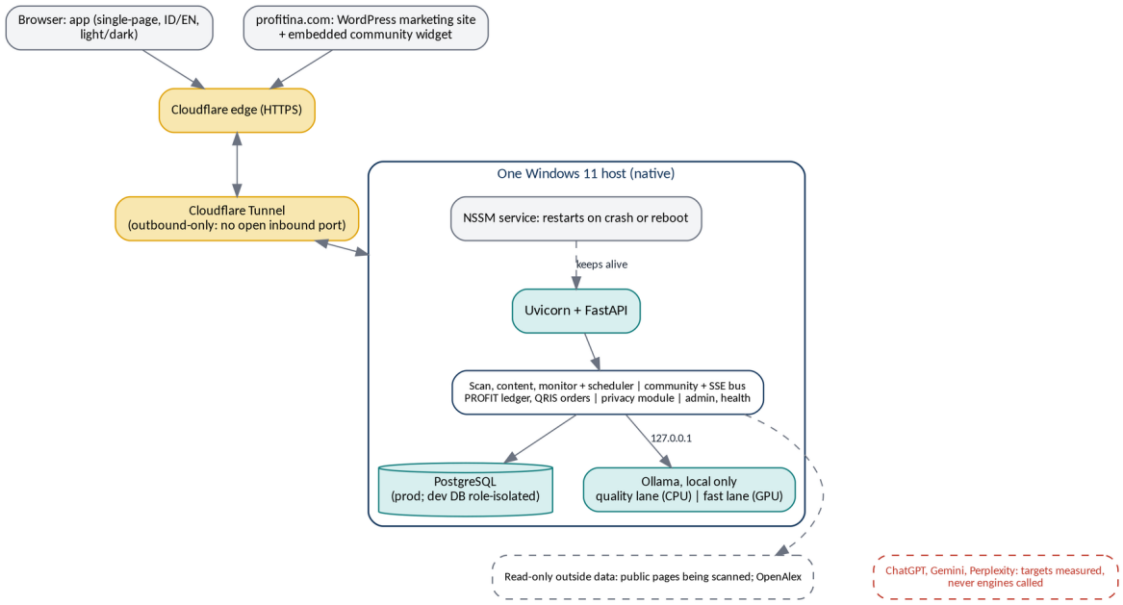


Figure 9.1 Profitina's architecture at box level. Browsers and the marketing site reach one Windows 11 host through the Cloudflare edge and an outbound-only tunnel; inside, NSSM keeps Uvicorn and FastAPI alive, with PostgreSQL and local AI beside them. Answer engines are measured targets, never called.

We will walk through it from the outside in, following a request from a person's phone to the database and back.

9.3.1 The two front doors

There are two places a person meets Profitina.

The **app** is a single-page application in plain JavaScript, served to the browser once and then updating itself without reloading pages. It uses hash routing (addresses such as

`#/komunitas/post/123`), a bilingual Indonesian and English dictionary, and light and dark themes. A single-page app is a good fit for a dashboard that people keep open, because moving between sections costs only the data, not a whole new page.

The **marketing site**, profitina.com, is a separate WordPress site with a custom bilingual theme. It explains the product in answer-first pages, publishes structured data and an `llms.txt` file for AI answer engines, and embeds the community widget read-only. Chapter 10 treats it as a GEO case study in its own right. Architecturally, the important point is separation: the marketing site can be edited, cached and even broken without touching the app.

9.3.2 The edge and the tunnel

Both front doors reach the internet through Cloudflare. Its *edge*, a network of data centers close to users, terminates HTTPS, absorbs abusive traffic, and forwards legitimate requests.

The link from the edge to the app's machine is a **Cloudflare Tunnel**. The tunnel is a small program running on the machine that opens an *outbound* connection to Cloudflare and keeps it open; requests for the app travel back down that connection. Cloudflare's documentation describes the model exactly: the connector "initiates an outbound connection through your firewall", which lets you "block all inbound access to your origin except through Cloudflare" [2].

> **KEY INSIGHT.** An outbound-only tunnel turns a hard networking problem into an easy one. The machine needs no fixed public address and no open inbound port. Nothing on the public internet can knock on its door directly; every request has passed through the edge first.

This design fits a machine that sits on an ordinary network, behind a router whose settings its owner may not control, and it removes a whole category of attacks. It also illustrates a principle this chapter returns to: prefer the arrangement in which the dangerous thing is impossible, rather than the one in which it is merely forbidden.

9.3.3 One host, kept alive by NSSM

The app runs on a single machine with **Windows 11, native**: no virtual machine, no Linux subsystem, no containers. The Python application is run as a Windows *service*, so that it starts when the machine boots and runs without anyone logged in. The service wrapper is **NSSM**, the "Non-Sucking Service Manager", a small open-source tool that turns any program into a Windows service and restarts it automatically if it exits [3].

In terms of Equation (9.1), NSSM is an MTTR tool. If the application crashes, it is back within seconds; if the machine reboots after an update, the service starts by itself. Neither requires a person.

9.3.4 The application: FastAPI under Uvicorn

The application itself is written in Python with **FastAPI**, a framework for building web APIs that uses Python type hints to validate requests and generate documentation [4]. FastAPI applications are served by an ASGI server; Profitina uses **Uvicorn** [5]. ASGI (Asynchronous Server Gateway Interface) is the Python standard that lets one process hold many connections open at once, which matters for the next box.

Inside the application, the work is divided into modules by area: scanning, content and the technical assistant; monitoring and its scheduler; the community with its channels, direct messages and moderation; the PROFIT ledger, QRIS orders and spending; the privacy module that implements consent, export and deletion (Chapter 11); administration; and observability. At the

time of writing the application exposes about two hundred HTTP routes. Describing what each module does is fine and useful; how the scoring inside them works stays inside them.

9.3.5 Real time: Server-Sent Events

When someone posts in the community, everyone watching that channel should see it at once. Profitina does this with **Server-Sent Events** (SSE): the browser opens one long-lived HTTP request to an events endpoint, and the server writes a small text message down that connection whenever something happens [6]. The browser's built-in `EventSource` object reconnects automatically if the connection drops. Chapter 4 built an SSE feed in its lab G2; here it is enough to note that SSE is one-way (server to browser), uses ordinary HTTP, and passes through the edge and the tunnel without special configuration.

Why not the older approach, in which the browser asks the server every few seconds whether anything is new? That approach, called *polling*, is simple but wasteful: most questions are answered "nothing yet", and every question still costs a connection, a query and some of a phone's battery. With a hundred readers each asking every three seconds, the server answers two thousand questions a minute only to say that nothing happened. With SSE, the server stays quiet until there is something worth saying, then says it once to everyone listening. And why not WebSocket, which is two-way? Because the feed needs only one direction: new posts are sent with ordinary HTTP requests, while notifications flow from server to browser. Choosing the smallest tool that fits the need is a habit we will meet repeatedly in this chapter.

9.3.6 The database: PostgreSQL, with separation enforced

Production data lives in **PostgreSQL**. A separate development database exists, and the separation between the two is enforced by PostgreSQL's own role permissions, not by a convention that developers are trusted to follow [7]. A development session connects with a role that simply has no rights on production tables; a mistake is refused by the database rather than merely discouraged by a document.

The test suite also runs against SQLite, which is quicker to set up. Two SQL dialects invite subtle differences, and Section 9.5.3 tells how one of them was caught.

9.3.7 The AI lanes: local only

All AI work runs on the same machine through **Ollama**, a local model server, with models from the Qwen family in two lanes (Chapter 6): a *quality lane* on the processor for long-form content, and a *fast lane* on the graphics card for short classifications such as link safety and category detection. No user data leaves the server for AI processing, and the answer engines that Profitina measures (ChatGPT, Gemini, Perplexity) are never called: they are targets, not components.

One small lesson from this box has a number attached. On Windows, the name `localhost` may resolve first to the IPv6 address `::1`, while a server listening only on IPv4 waits at `127.0.0.1`. The client tries IPv6, waits, and falls back. In the team's measurement, calling the model server at `127.0.0.1` rather than `localhost` cut the time to the first generated token from **2.25 seconds to 0.17 seconds**. The lab's health check in Section 9.10 follows the same rule, and says so in a comment.

9.3.8 What is deliberately not in the picture

Just as important as the boxes in **Figure 9.1** are the ones that are absent. There is no cloud hyperscaler, no Kubernetes, no microservices, no automatic scaling. **Table 9.2** summarizes the choices and their reasons.

Table 9.2 The components of **Figure 9.1**, the choice made for each, and the main reason.

Box	Choice	Main reason
Front end	Single-page app, vanilla JavaScript	A dashboard people keep open; no build chain to maintain
Marketing	WordPress, bilingual theme	Editors can publish without touching the app
Edge	Cloudflare	HTTPS, caching, protection in front of one machine
Ingress	Cloudflare Tunnel, outbound-only	No public address or inbound port needed
Host	One Windows 11 machine, native	Stage-appropriate; simplest thing that works
Service	NSSM	Restart on crash and on boot, without a person
Application	FastAPI on Uvicorn (ASGI)	Typed validation, many open connections
Real time	Server-Sent Events	One-way push over plain HTTP
Data	PostgreSQL, dev isolated by roles	Mature, strict; mistakes refused by the database
AI	Ollama, two local lanes	Privacy, independence, no per-request bill

9.4 Why One Machine Is Enough, for Now

A single machine is a *single point of failure*: if it is off, the service is off. Critics of small architectures usually stop there. It is worth continuing.

First, the alternative has its own failure modes. A distributed system has more moving parts, more network links, more configuration and more ways for two parts to disagree. For a team of a few people, the most likely cause of an outage is not hardware failure; it is a mistake during a change. A simpler system makes fewer such mistakes possible, and makes the remaining ones easier to see.

Second, the costs of scale arrive before its benefits. Running the same service on many machines requires shared sessions, a database reachable over the network, coordinated releases, and monitoring that understands all of it. Each is weeks of work, and none of it improves a single scan for a single user while traffic is modest.

Third, the rest of this chapter shows that most of the reliability comes from *process*, not from the number of machines: a service manager that restarts quickly, backups taken before every change, a test suite that must pass, and a release that proves the new version is running before it claims success. Those habits carry over unchanged when the system does grow.

A note on backups, since the word appears many times in this chapter. A backup that has never been restored is not a backup; it is a hope. The healthy habit is to restore a backup now and then into a separate place and check that it is actually usable: the application starts, the data reads, the row counts make sense. Such a rehearsal also measures the real MTTR, because the time a restore takes is known only once it has been done. In terms of Equation (9.1), a restore rehearsal is a cheap investment in a smaller denominator.

Profitina's public roadmap reserves regional scale for a later phase, after the domestic product has proved itself [8]. At that point the architecture will change, and it should. Building for that scale now would be paying today for a problem the product does not yet have.

WARNING. "One machine is enough" is a claim about *this* stage, supported by measurement. It is not a general law. The honest version is: know what would make it false (sustained load the machine cannot serve, or a recovery time users cannot accept) and watch for it.

9.5 The Test Suite as a Release Gate

Every change to a running system is a small risk. The purpose of automated tests is to make most of that risk visible *before* the change reaches users.

DEFINITION. A *release gate* is a check that a release must pass before it may proceed. A gate is only a gate if failing it actually stops the release: a check whose failure is reported but ignored is a warning, not a gate.

9.5.1 The size of the suite

At the time of writing (24 September 2026), Profitina's suite looks like **Table 9.3**.

Table 9.3 The automated suite that gates every release, at the time of writing (24 September 2026).

Part of the suite	Size in the September 2026 release
Full Python suite on SQLite	1,577 passed, 1 skipped
Priority subset on PostgreSQL 16	440 passed (49 test files)
Python test files	153
JavaScript UI guard files	6, with their own harness
End-to-end checks	Playwright driving a real Chromium browser

The full suite runs with `pytest`, configured so that any *warning* becomes an error (`filterwarnings = error`). That setting is small and strict: a deprecation warning today is tomorrow's failure after a library upgrade, and treating it as an error means it is fixed while it is still easy. The JavaScript guards check user-interface rules that are easy to break without noticing, such as every visible string having both an Indonesian and an English translation.

Numbers like 1,577 age within days, and they are not the point. The point is the rule: **the whole suite must be green before a release proceeds**.

9.5.2 Why a gate, and why now

It is worth pausing on why this matters more in 2026 than it did a few years ago. The 2025 DORA report, based on a survey of nearly 5,000 technology professionals, found that 90% of respondents use AI at work and that AI adoption now has a positive relationship with software delivery *throughput*, but a continuing *negative* relationship with delivery *stability* [9]. The report's summary is memorable: "AI doesn't fix a team; it amplifies what's already there." Code is being written faster than ever. A team whose tests are thin will simply ship its mistakes faster. A team with a strong gate turns speed into value.

9.5.3 Two lessons the suite taught

Tests are only as good as what they exercise. Two stories from the product's history show how a suite gets better.

The dialect trap. SQLite and PostgreSQL both speak SQL, but not identically. SQLite accepts `MAX(a, b)` as a function of two values, returning the larger; PostgreSQL does not, and expects `GREATEST(a, b)`. Separately, a hand-written test once inserted a row directly into a table, leaving

out a column that PostgreSQL declared `NOT NULL`; SQLite accepted it, PostgreSQL refused. Two rules came out of these incidents. First, a *dialect guard* test scans the code for constructs that behave differently in the two databases, and a priority subset of tests runs on real PostgreSQL. Second: **tests set up their state through the product's own functions when those exist**, not with hand-written SQL. A test that uses the real "create post" function will fail if the real function is broken, and will keep working if the table gains a column; a test with its own `INSERT` checks only itself.

The silent zero. The loud-errors rule of Section 9.2.3 was itself enforced by tests: once the team knew that a swallowed error could look like a zero, tests were added that force the failure and check that it is visible.

> **KEY INSIGHT.** A good test suite is grown from incidents. Every bug that reached a person is a test that did not yet exist; writing that test is how the same bug is prevented from ever coming back.

9.6 A Health-Verified, Staged Release

Tests show that the new code works *on the test machine*. They do not show that it is *running* in production. A release has to cross that last gap, and the last gap is where one of Profitina's most instructive incidents happened.

9.6.1 The gate that lied

Profitina's releases on Windows were once driven by a batch script: a text file of commands that the Windows command interpreter, `cmd.exe`, executes line by line. There is a little-known property of `cmd.exe`: it does not read the whole script before running it. It reads the file *by byte position* as it goes, remembering where it is. If the script file changes while it runs, the interpreter continues at the same byte position in the *new* file, which may now be the middle of a different line.

During one release, a commit that touched the release script landed while the script was running. The interpreter resumed at a shifted position, skipped part of its work, and reached the lines that printed success. The gate reported a successful release of a version that was not, in fact, the one running.

Notice what failed. Not the tests; they had passed. Not the copy; it may even have happened. What failed was the *claim*: the gate said "released" on the strength of having reached the end of a script, rather than on evidence that the new version was serving users.

9.6.2 The fix: success means the service says so

Two changes closed the gap. The first is mechanical: the release runs a *copy* of the script, so edits to the original cannot affect a release in progress. The second is the important one: **the release declares success only after the service's health endpoint answers with the new version number**. The gate no longer believes itself; it asks the running system.

9.6.3 Six steps

The release, as it stands in the September 2026 release, is a sequence of six steps, drawn in **Figure 9.2**.

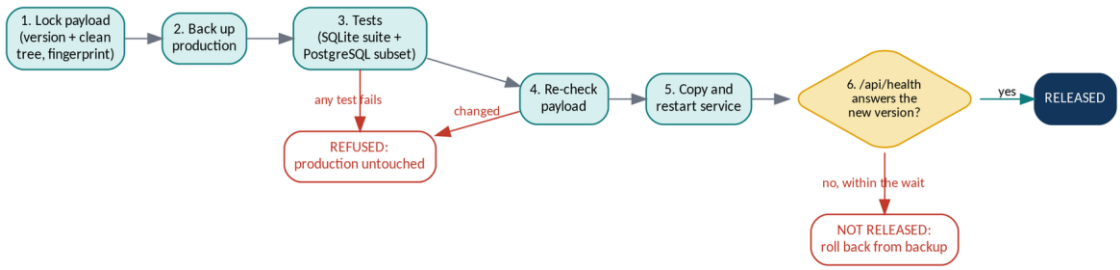


Figure 9.2 The six-step, health-verified release. Any failing test or a payload that changed during the gate refuses the release with production untouched; a new version that does not answer the health check in time is rolled back.

1. **Lock the payload.** Record exactly which version is being released (the commit, and a clean working tree with no uncommitted edits), so that everything that follows refers to one fixed thing.
2. **Back up.** Take a backup of production before anything changes. If the release goes wrong, there is a known good state to return to.
3. **Test.** Run the full SQLite suite and the PostgreSQL priority subset. Any failure stops here, and production has not been touched.
4. **Re-check the payload.** Confirm that the version is still the one locked in step 1. If someone committed during the tests, the tested code is no longer the code about to be copied, and the release stops.
5. **Copy and restart.** Copy the release into place and restart the service.
6. **Verify.** Ask `/api/health` which version is running. Only when it answers with the new version is the release a success. If it does not within a reasonable wait, the release has failed, and the backup of step 2 makes rolling back routine.

One more property of this order deserves attention: the expensive or risky steps come after the cheap and safe ones. Locking the payload and taking a backup cost almost nothing; testing takes minutes but touches nothing; only step 5 actually changes production. With this arrangement most failures happen *before* anything has changed, and when a failure happens afterwards, the backup from step 2 is already waiting. The principle reaches far beyond software: an electrician switches off the circuit breaker before opening the socket, not after.

The word *staged* in the title of this section refers to steps 1 to 4: the release is assembled and proved in a staging area before anything touches the production copy.

> **EXAMPLE.** Why is step 4 needed if step 1 already locked the version? Suppose the tests take eight minutes. During those eight minutes a developer commits a small change. Without step 4, the copy in step 5 would take the newest files, which include the untested change. The gate would release code it never tested and report that it had. Step 4 turns that silent mismatch into a refusal.

9.6.4 What the health endpoint reports

A health endpoint is a URL that monitoring tools, and release scripts, can call to ask “are you working?”. Microsoft’s architecture guidance describes the pattern as functional checks exposed through an endpoint that “external tools can access at regular intervals”, where “the response code indicates application status, with optional information about component and service health” [10].

Profitina's `/api/health` reports the running version and the state of its dependencies: the database, the local AI, the disk, the uptime, and the count of recent server errors. Two details matter.

It reports the version the process is running, not the version on disk. This is exactly what made step 6 meaningful: a process that was never restarted still reports its old version, however many new files were copied beside it.

It separates "down" from "degraded". If the database is unreachable, the service cannot serve users and should say so with an error status. If only the AI lane is unavailable, the service still works, and the honest answer is "working, without AI", not "failed". The lab endpoint in Section 9.10 makes the same distinction.

Observability goes a little further than health. Every server error (HTTP 5xx) is recorded without request bodies or headers, with the client address hashed, and spike alerts watch for sudden increases. Error logs are deleted after 90 days by a purge job that actually runs; a retention promise that is never executed is not a promise.

9.7 The Performance Story: Switching Channels

This section tells a performance investigation from start to finish, because the habits it shows are more valuable than the result.

9.7.1 The symptom

In September 2026 users reported that switching channels in the community feed was "very slow". The symptom was vague, as symptoms usually are. The first job of an engineer is to turn a feeling into a number. The team measured `/api/feed` for a page of 30 posts and counted what it did. The answers were startling: **868 milliseconds**, **335 database queries**, **122 new database connections**, and **6 calls to an AI model**, for one page of posts.

How one measures matters too. A single measurement can mislead: the first request is often slower because caches are cold, and a request that happens to coincide with other work can look slow for no reason. A trustworthy measurement therefore repeats the same request several times, discards the warm-up, and reports a typical figure such as the median. The team also counted *work*, not only *time*: numbers of queries, connections and model calls do not depend on how busy the machine happens to be, so they stay meaningful on any computer. Time tells you that there is a problem; counts of work tell you where it is.

9.7.2 Three root causes

The investigation found three separate causes, each ordinary on its own.

An AI call on the read path. When rendering a post containing a link, the code asked the local model for an opinion about any link that looked suspicious. That is a reasonable thing to do *once*, when the post is created. Done on every render, for every viewer, it put a model call inside the loop that draws the feed.

A new connection per call. Each helper function opened its own connection to PostgreSQL. Opening a connection involves a network handshake and authentication; doing it 122 times per page is pure overhead.

The N+1 query pattern. The feed loaded 30 posts with one query, then, for each post, ran further queries for its writer, its channel, its counts of replies and likes, and so on, about eleven per post. If a page has N posts and each post needs k extra queries, the total is

$$Q(N) = 1 + kN \tag{9.3}$$

With $N = 30$ and $k \approx 11$, Equation (9.3) gives about 331 queries, close to the 335 measured. The name "N+1" comes from exactly this shape: one query for the list, then N more (here kN) for its members. The fix, *prefetched hydration*, collects what all 30 posts need and fetches it in a handful of queries, one per kind of data, so the count no longer grows with N.

9.7.3 The fix and the result

The fix had three parts, one per cause: the read path never calls a model (the link opinion is computed when the post is written and stored); connections come from an in-process *pool*, opened once and reused; and hydration is prefetched. **Figure 9.3** shows the result.

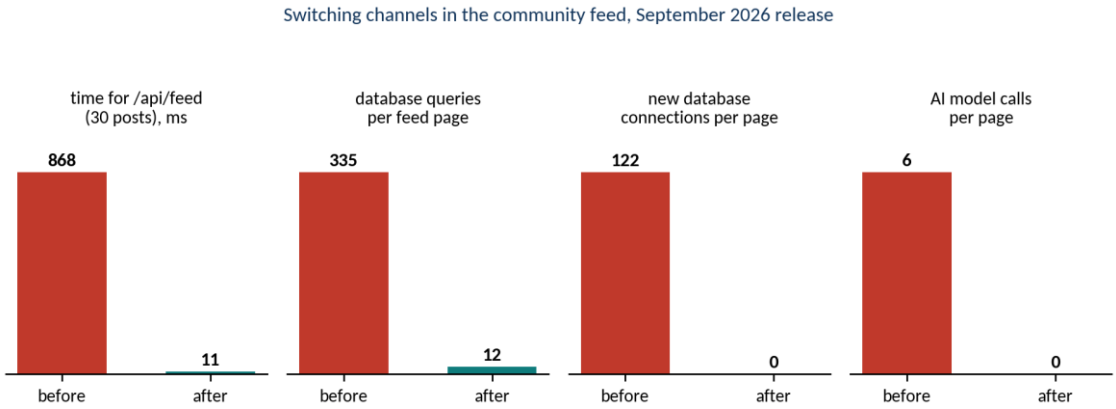


Figure 9.3 Switching channels in the community feed, before and after the September 2026 fix: time for a 30-post page, database queries, new connections and AI calls per page.

The feed now takes **11 milliseconds**, with **12 queries**, **no new connections** and **no AI calls**. The speed-up factor is

$$S = \frac{868\ \text{ms}}{11\ \text{ms}} \approx 79 \tag{9.4}$$

The same treatment brought the dashboard endpoint from 145 to 15 milliseconds and the embedded community widget from 264 queries to 8. In a real Chromium browser, the 30 feed cards now render in about 143 milliseconds after a channel switch, which people experience as instant.

9.7.4 The rule, and the race

A fix that is not written down as a rule will be undone by the next well-meaning change. The team extracted one: **a function called per item while rendering must never touch the model or open its own connection.** It is short enough to remember and specific enough to check in review, and tests now count queries on the feed path, so that a return of the N+1 pattern fails the release gate of Section 9.5.

The investigation also found a correctness bug hiding behind the slowness: a *stale-response race*, drawn in **Figure 9.4**. A user opens channel A, then quickly switches to B. Both requests are in flight. If B's reply arrives first and A's arrives late, a naive page shows B's posts and then

overwrites them with A's, under B's heading. When every request took almost a second, this happened often.

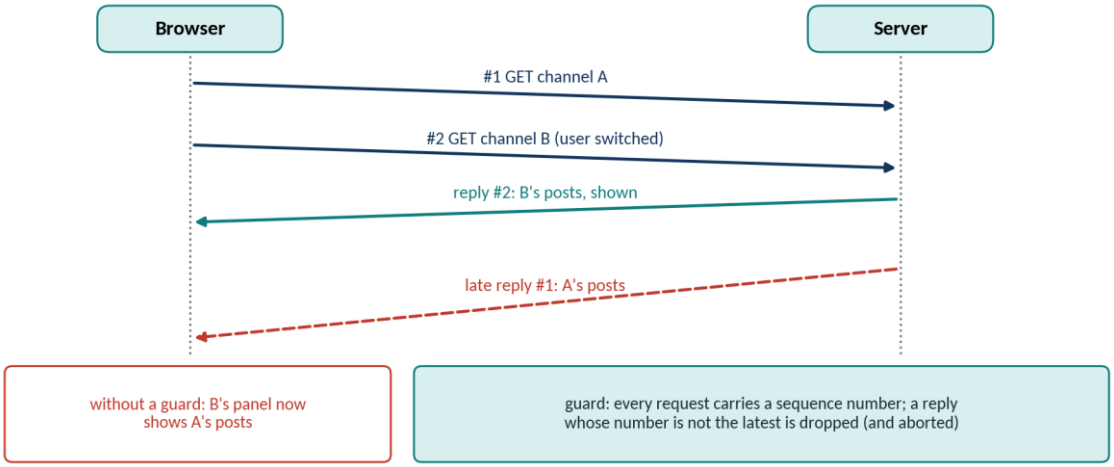


Figure 9.4 The stale-response race. A late reply to an earlier request must not overwrite the reply to the latest one; a sequence number on each request, checked when the reply arrives, discards the stale reply.

The fix gives every request a sequence number and ignores any reply whose number is not the latest; the browser's `AbortController` also cancels the obsolete request, so it stops consuming the network [11]. The guard was then applied to *every* panel that loads per channel, not only the feed, because a bug found in one place usually lives in its neighbors too. Exercise 9.7 asks you to write the guard.

> **KEY INSIGHT.** Measure first, name the causes, fix each one, write the rule, and add a test that enforces the rule. Speed that is not protected by a test is on loan.

9.8 Scheduling Re-scans: A Priority Queue

Chapter 8 listed monitoring among the differences between the tiers: Free gets one monitor re-scanned weekly, Pro up to fifty re-scanned daily. Somebody has to decide, at every moment, which monitor is due next. That somebody is a *scheduler*, and the data structure at its heart is a *priority queue*.

9.8.1 The problem

Suppose there are n monitors, each with a *due time*, the moment its next scan should run. The scheduler repeatedly needs three operations: find the monitor with the earliest due time; remove it; and insert it again with a new due time after it has run. A plain list would make one of these slow: keeping it sorted makes insertion cost up to n steps, and leaving it unsorted makes finding the earliest cost n steps.

A **binary min-heap** does all three well. It is a binary tree stored in an array, in which every node's key is no larger than its children's keys, so the smallest key is always at the root [12]. Looking at the root costs one step; removing the root or inserting a new entry costs a number of steps proportional to the height of the tree, which is about $\log_2 n$. **Figure 9.5** shows a small heap of scheduled scans.

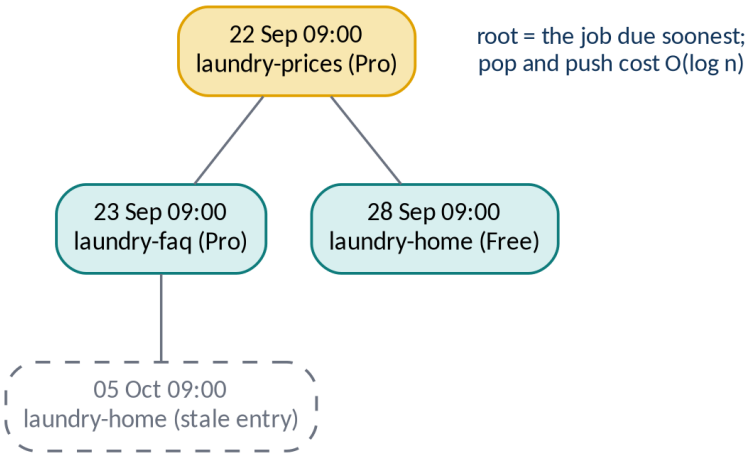


Figure 9.5 A min-heap of scheduled scans ordered by due time. The root is the next job to run; the dashed entry is a stale one left behind by a reschedule, skipped when it reaches the root.

If the scheduler performs m pops and pushes in a day over a heap of n entries, the total work is

$$W = O(m \log n) \tag{9.5}$$

With $n = 1{,}000$ monitors, $\log_2 n \approx 10$, so each operation is about ten steps. Python's `heapq` module and Java's `PriorityQueue` class both implement binary heaps [13], [14].

9.8.2 Cadence and how many runs to expect

Each tier has a *cadence* c: seven days for Free, one day for Pro. Over a period of length P starting with a scan at time zero, a monitor runs at times $0, c, 2c, \ldots$ that fall inside the period, which is

$$r = \left\lceil \frac{P}{c} \right\rceil \tag{9.6}$$

runs, where $\lceil x \rceil$ is the smallest whole number not less than x. For a 14-day period Equation (9.6) gives $\lceil 14/7 \rceil = 2$ Free runs and $\lceil 14/1 \rceil = 14$ Pro runs; the lab's tests check exactly these numbers.

9.8.3 Three details that separate a toy from a tool

Lazy deletion. When a monitor is removed or its tier changes, its old entry is still somewhere inside the heap, and removing an entry from the middle of a heap is awkward. Instead, each monitor carries a *version* number that increases every time it is rescheduled, and each heap entry records the version it was created with. When an entry reaches the root, the scheduler compares versions; a mismatch means the entry is stale, and it is simply discarded. The dashed box in **Figure 9.5** is such an entry.

Late workers skip, they do not burst. If the machine was off for three days, a Pro monitor has missed three daily slots. Running all three the moment the machine returns would produce a burst of identical scans, loading the system at the worst moment and telling the user nothing new. The scheduler instead runs the monitor once and moves its next due time forward to the first slot in the future, counting the slots it skipped. If t is the current time and d the old due time, the number of skipped slots is

$$s = \left\lfloor \frac{t - d}{c} \right\rfloor \tag{9.7}$$

For a Pro monitor due at day 1 whose worker returns at day 5 plus one hour, Equation (9.7) gives $\lfloor 4.04 \rfloor = 4$ skipped slots, and the next due time is day 6.

Idempotence. Running the scheduler twice at the same instant must not run anything twice. Because a job, once run, is pushed back with a *later* due time, a second call at the same instant finds nothing due. An operation that has the same effect whether it is performed once or several times is called *idempotent*, and it is exactly the property that makes a scheduler safe to call from a timer that might fire twice.

A simulated clock. The scheduler never reads the real clock; it is told what time it is. That makes it possible to test three weeks of scheduling in a millisecond, including a three-day outage, which no test could do with a real clock.

9.9 Lab G4, Part 1: The Scheduler

Lab G4 has three parts. This part builds the scheduler, and its tests become the test suite that the release gate of Part 3 will guard.

9.9.1 The Python program

Listing 9.1 is the scheduler, with a demonstration that runs 21 simulated days for three Profitina Laundry pages.

Listing 9.1 `scheduler.py`: a min-heap scheduler with per-tier cadence, lazy deletion, skip-not-burst catch-up and a simulated clock.

```
"""A tiny re-scan scheduler: a min-heap of (due time, monitor) with a
per-tier cadence, driven by a simulated clock."""
import heapq
import zlib
from dataclasses import dataclass
from datetime import datetime, timedelta
from zoneinfo import ZoneInfo

WIB = ZoneInfo("Asia/Jakarta")
CADENCE = {"free": timedelta(days=7), "pro": timedelta(days=1)}


@dataclass
class Monitor:
    mid: str
    tier: str
    version: int = 0              # bumped whenever the monitor is rescheduled
    next_due: datetime | None = None
    last_run: datetime | None = None


def stub_scan(mid: str, when: datetime) -> int:
    """Stands in for a real scan: a repeatable score from 50 to 89."""
    key = f"{mid}|{when.astimezone(WIB):%Y-%m-%d}".encode()
    return 50 + zlib.crc32(key) % 40


class Scheduler:
    def __init__(self, scan=stub_scan):
        self.scan = scan
        self.heap: list[tuple[datetime, int, str, int]] = []
        self.seq = 0                  # breaks ties in insertion order
        self.monitors: dict[str, Monitor] = {}
        self.results: list[tuple[str, datetime, int]] = []
        self.skipped = 0

    def _push(self, m: Monitor, due: datetime) -> None:
        m.version += 1                # older heap entries become stale
        m.next_due = due
        self.seq += 1
        heapq.heappush(self.heap, (due, self.seq, m.mid, m.version))

    def add(self, m: Monitor, now: datetime) -> None:
        self.monitors[m.mid] = m
        self._push(m, now)            # first scan as soon as possible
```

Listing 9.1 (Continued) *scheduler.py: a min-heap scheduler with per-tier cadence, lazy deletion, skip-not-burst catch-up and a simulated clock.*

```
    def remove(self, mid: str) -> None:
        self.monitors.pop(mid, None)   # its heap entry is skipped later

    def change_tier(self, mid: str, tier: str, now: datetime) -> None:
        m = self.monitors[mid]
        m.tier = tier
        base = m.last_run or now
        self._push(m, max(now, base + CADENCE[tier]))

    def run_due(self, now: datetime) -> int:
        """Run every job whose time has come; return how many ran."""
        ran = 0
        while self.heap and self.heap[0][0] <= now:
            due, _, mid, version = heapq.heappop(self.heap)
            m = self.monitors.get(mid)
            if m is None or m.version != version:
                continue               # removed or rescheduled: stale entry
            self.results.append((mid, now, self.scan(mid, now)))
            m.last_run = now
            ran += 1
            step = CADENCE[m.tier]
            nxt = due + step
            while nxt <= now:          # we were late: skip, never burst
                nxt += step
                self.skipped += 1
            self._push(m, nxt)
        return ran


def at(text: str) -> datetime:
    return datetime.strptime(text, "%Y-%m-%d %H:%M").replace(tzinfo=WIB)


def main() -> None:
    s = Scheduler()
    start = at("2026-09-21 09:00")
    s.add(Monitor("laundry-home", "free"), start)
    s.add(Monitor("laundry-prices", "pro"), start)
    down_from, down_to = at("2026-10-01 00:00"), at("2026-10-03 23:59")
    days = 21
    for step in range(days * 8):               # every 3 hours
        now = start + timedelta(hours=3 * step)
        if now == at("2026-09-23 09:00"):
            s.add(Monitor("laundry-faq", "pro"), now)
        if now == at("2026-09-30 12:00"):
            s.change_tier("laundry-home", "pro", now)
```

Listing 9.1 (Continued) *scheduler.py: a min-heap scheduler with per-tier cadence, lazy deletion, skip-not-burst catch-up and a simulated clock.*

```
        if now == at("2026-10-06 09:00"):
            s.remove("laundry-faq")
        if down_from <= now <= down_to:
            continue                            # the machine is off
        s.run_due(now)

    first = start.date()
    print("day of month   " + "".join(str((first + timedelta(d)).day % 10)
                                      for d in range(days)))
    for mid in ("laundry-home", "laundry-prices", "laundry-faq"):
        row = ["."] * days
        for d in range(days):
            day = first + timedelta(d)
            if down_from.date() <= day <= down_to.date():
                row[d] = "-"
        for rmid, when, _ in s.results:
            if rmid == mid:
                row[(when.date() - first).days] = "x"
        print(f"{mid:<15}{''.join(row)}")
    print("(last digit of the date, 21 Sep to 11 Oct 2026)")
    print("x = scan ran, . = nothing due, - = machine off")
    print("runs:", len(s.results), "| skipped slots:", s.skipped,
          "| heap entries left:", len(s.heap))
    for mid, when, score in s.results[:4]:
        print(f"  {when:%d %b %H:%M} WIB  {mid:<15} score {score}")


if __name__ == "__main__":
    main()
```

The heap stores tuples `(due, seq, mid, version)`. Python compares tuples element by element, so the heap orders entries by due time first; `seq`, a counter that only grows, breaks ties in insertion order so that two monitors due at the same instant never force Python to compare the rest of the tuple. `_push` is the only place entries are created, and it always bumps the monitor's version first.

`run_due` is the heart of the program. While the root of the heap is due, it pops the entry, discards it if it is stale, runs the scan, records the result, and computes the next due time from the *old due time*, not from now, so that a daily monitor stays on its daily grid. The inner `while` loop is Equation (9.7) done by counting.

`stub_scan` stands in for a real scan. It returns a repeatable number from 50 to 89 computed with CRC-32 from the monitor's name and the WIB date, so that the Python and Java versions print the same scores. It is not a visibility score and does not pretend to be one; lab G1 of Chapter 3 is the place to plug in a real checker.

The demonstration adds a Free monitor for the laundry's home page and a Pro monitor for its price page on Monday 21 September 2026, adds a Pro monitor for its FAQ page two days later, upgrades the home page to Pro on 30 September, switches the machine off from 1 to 3 October, and removes the FAQ monitor on 6 October. It prints a small calendar, one column per day:

```
day of month   12345678901234567890
laundry-home   x.....x.x---xxxxxxxx
laundry-prices xxxxxxxxxx---xxxxxxxx
laundry-faq    ..xxxxxxxx---xx......
(last digit of the date, 21 Sep to 11 Oct 2026)
x = scan ran, . = nothing due, - = machine off
runs: 42 | skipped slots: 6 | heap entries left: 2
  21 Sep 09:00 WIB  laundry-home    score 71
  21 Sep 09:00 WIB  laundry-prices  score 81
  22 Sep 09:00 WIB  laundry-prices  score 79
  23 Sep 09:00 WIB  laundry-prices  score 77
```

Read the rows. `laundry-home` runs on 21 September and a week later on the 28th, as a Free monitor should. On the 30th, the upgrade brings its next run forward at once, since a day had already passed since its last scan, and from then on it runs daily. All three monitors show `-` for the three days the machine was off, then resume on 4 October with a single run each: six missed slots were skipped (two for each Pro monitor), not replayed. `laundry-faq` stops after 5 October. At the end, two heap entries remain, one for each live monitor; the FAQ monitor's last entry was discarded as stale when it reached the root.

9.9.2 The Java program

Listing 9.2 is the same scheduler in Java. `PriorityQueue` takes a comparator; `Comparator.comparing(Entry::due).thenComparingLong(Entry::seq)` expresses "by due time, then by sequence" directly. The output is identical to the Python output above.

Listing 9.2 `Scheduler.java`: the same scheduler with `java.util.PriorityQueue` and records.

```
import java.nio.charset.StandardCharsets;
import java.time.Duration;
import java.time.LocalDate;
import java.time.LocalDateTime;
import java.time.ZoneId;
import java.time.ZonedDateTime;
import java.time.format.DateTimeFormatter;
import java.time.temporal.ChronoUnit;
import java.util.ArrayList;
import java.util.Comparator;
import java.util.HashMap;
import java.util.List;
import java.util.Locale;
import java.util.Map;
import java.util.PriorityQueue;
import java.util.zip.CRC32;

/** A tiny re-scan scheduler: a min-heap of (due time, monitor) with a
 *  per-tier cadence, driven by a simulated clock. */
public class Scheduler {
    static final ZoneId WIB = ZoneId.of("Asia/Jakarta");
    static final Map<String, Duration> CADENCE = Map.of(
            "free", Duration.ofDays(7), "pro", Duration.ofDays(1));

    static class Monitor {
        final String mid;
        String tier;
        int version = 0;        // bumped whenever the monitor is rescheduled
        ZonedDateTime nextDue;
        ZonedDateTime lastRun;
        Monitor(String mid, String tier) { this.mid = mid; this.tier = tier; }
    }

    record Entry(ZonedDateTime due, long seq, String mid, int version) { }
    record Result(String mid, ZonedDateTime when, int score) { }

    /** Stands in for a real scan: a repeatable score from 50 to 89. */
    static int stubScan(String mid, ZonedDateTime when) {
        String day = when.withZoneSameInstant(WIB).toLocalDate().toString();
        CRC32 crc = new CRC32();
        crc.update((mid + "|" + day).getBytes(StandardCharsets.UTF_8));
        return 50 + (int) (crc.getValue() % 40);
    }
```

Listing 9.2 (Continued) *Scheduler.java: the same scheduler with java.util.PriorityQueue and records.*

```
    final PriorityQueue<Entry> heap = new PriorityQueue<>(
            Comparator.comparing(Entry::due).thenComparingLong(Entry::seq));
    long seq = 0;                       // breaks ties in insertion order
    final Map<String, Monitor> monitors = new HashMap<>();
    final List<Result> results = new ArrayList<>();
    int skipped = 0;

    private void push(Monitor m, ZonedDateTime due) {
        m.version++;                    // older heap entries become stale
        m.nextDue = due;
        heap.add(new Entry(due, ++seq, m.mid, m.version));
    }

    void add(Monitor m, ZonedDateTime now) {
        monitors.put(m.mid, m);
        push(m, now);                   // first scan as soon as possible
    }

    void remove(String mid) {
        monitors.remove(mid);           // its heap entry is skipped later
    }

    void changeTier(String mid, String tier, ZonedDateTime now) {
        Monitor m = monitors.get(mid);
        m.tier = tier;
        ZonedDateTime base = m.lastRun != null ? m.lastRun : now;
        ZonedDateTime due = base.plus(CADENCE.get(tier));
        push(m, due.isAfter(now) ? due : now);
    }

    /** Run every job whose time has come; return how many ran. */
    int runDue(ZonedDateTime now) {
        int ran = 0;
        while (!heap.isEmpty() && !heap.peek().due().isAfter(now)) {
            Entry e = heap.poll();
            Monitor m = monitors.get(e.mid());
            if (m == null || m.version != e.version())
                continue;               // removed or rescheduled: stale entry
            results.add(new Result(m.mid, now, stubScan(m.mid, now)));
            m.lastRun = now;
            ran++;
            Duration step = CADENCE.get(m.tier);
            ZonedDateTime next = e.due().plus(step);
            while (!next.isAfter(now)) {  // we were late: skip, never burst
                next = next.plus(step);
                skipped++;
```

Listing 9.2 (Continued) *Scheduler.java: the same scheduler with java.util.PriorityQueue and records.*

```
            }
            push(m, next);
        }
        return ran;
    }

    static ZonedDateTime at(String text) {
        DateTimeFormatter f = DateTimeFormatter.ofPattern("yyyy-MM-dd HH:mm");
        return LocalDateTime.parse(text, f).atZone(WIB);
    }

    public static void main(String[] args) {
        Scheduler s = new Scheduler();
        ZonedDateTime start = at("2026-09-21 09:00");
        s.add(new Monitor("laundry-home", "free"), start);
        s.add(new Monitor("laundry-prices", "pro"), start);
        ZonedDateTime downFrom = at("2026-10-01 00:00");
        ZonedDateTime downTo = at("2026-10-03 23:59");
        int days = 21;
        for (int step = 0; step < days * 8; step++) {      // every 3 hours
            ZonedDateTime now = start.plusHours(3L * step);
            if (now.equals(at("2026-09-23 09:00")))
                s.add(new Monitor("laundry-faq", "pro"), now);
            if (now.equals(at("2026-09-30 12:00")))
                s.changeTier("laundry-home", "pro", now);
            if (now.equals(at("2026-10-06 09:00")))
                s.remove("laundry-faq");
            if (!now.isBefore(downFrom) && !now.isAfter(downTo))
                continue;                                  // the machine is off
            s.runDue(now);
        }

        LocalDate first = start.toLocalDate();
        StringBuilder head = new StringBuilder("day of month   ");
        for (int d = 0; d < days; d++)
            head.append(first.plusDays(d).getDayOfMonth() % 10);
        System.out.println(head);
        for (String mid : new String[] {"laundry-home", "laundry-prices",
                                        "laundry-faq"}) {
            char[] row = ".".repeat(days).toCharArray();
            for (int d = 0; d < days; d++) {
                LocalDate day = first.plusDays(d);
                if (!day.isBefore(downFrom.toLocalDate())
                        && !day.isAfter(downTo.toLocalDate())) row[d] = '-';
            }
            for (Result r : s.results)
```

Listing 9.2 (Continued) *Scheduler.java: the same scheduler with java.util.PriorityQueue and records.*

```
                if (r.mid().equals(mid))
                    row[(int) ChronoUnit.DAYS.between(first,
                            r.when().toLocalDate())] = 'x';
            System.out.printf("%-15s%s%n", mid, new String(row));
        }
        System.out.println("(last digit of the date, 21 Sep to 11 Oct 2026)");
        System.out.println("x = scan ran, . = nothing due, - = machine off");
        System.out.println("runs: " + s.results.size() + " | skipped slots: "
                + s.skipped + " | heap entries left: " + s.heap.size());
        DateTimeFormatter f = DateTimeFormatter.ofPattern("dd MMM HH:mm",
                Locale.US);
        for (Result r : s.results.subList(0, 4))
            System.out.printf("  %s WIB  %-15s score %d%n",
                    r.when().format(f), r.mid(), r.score());
    }
}
```

9.9.3 The tests

Listing 9.3 and **Listing 9.4** are seven tests, each pinning one rule from Section 9.8: the cadence of the plans, the expected counts of Equation (9.6), the heap order, idempotence, skip-not-burst (Equation (9.7)), an upgrade bringing the next scan forward, and a removed monitor never running again. The Java tests use the `MiniTest` runner of Section 8.12.4, copied into this chapter's folder.

Listing 9.3 `test_scheduler.py`: seven tests for the scheduler. The release gate runs exactly this file.

```
"""Tests for the re-scan scheduler; the release gate runs these."""
import unittest
from datetime import timedelta

from scheduler import CADENCE, Monitor, Scheduler, at

T0 = at("2026-09-21 09:00")


def run_hourly(s: Scheduler, hours: int) -> None:
    for h in range(hours + 1):
        s.run_due(T0 + timedelta(hours=h))


class SchedulerTests(unittest.TestCase):
    def test_cadence_matches_the_plans(self):
        self.assertEqual(CADENCE["free"], timedelta(days=7))
        self.assertEqual(CADENCE["pro"], timedelta(days=1))

    def test_two_weeks_give_2_free_and_14_pro_scans(self):
        s = Scheduler()
        s.add(Monitor("f", "free"), T0)
        s.add(Monitor("p", "pro"), T0)
        run_hourly(s, 14 * 24 - 1)
        runs = [mid for mid, _, _ in s.results]
        self.assertEqual((runs.count("f"), runs.count("p")), (2, 14))

    def test_jobs_come_out_in_due_order(self):
        s = Scheduler()
        s.add(Monitor("late", "pro"), T0 + timedelta(hours=5))
        s.add(Monitor("early", "pro"), T0 + timedelta(hours=1))
        s.run_due(T0 + timedelta(hours=6))
        self.assertEqual([r[0] for r in s.results], ["early", "late"])

    def test_running_twice_at_the_same_time_is_idempotent(self):
        s = Scheduler()
        s.add(Monitor("p", "pro"), T0)
        self.assertEqual(s.run_due(T0), 1)
        self.assertEqual(s.run_due(T0), 0)
```

Listing 9.3 (Continued) *test_scheduler.py: seven tests for the scheduler. The release gate runs exactly this file.*

```
    def test_late_worker_skips_instead_of_bursting(self):
        s = Scheduler()
        s.add(Monitor("p", "pro"), T0)
        s.run_due(T0)
        self.assertEqual(s.run_due(T0 + timedelta(days=5, hours=1)), 1)
        self.assertEqual(s.skipped, 4)
        self.assertEqual(s.monitors["p"].next_due, T0 + timedelta(days=6))

    def test_upgrade_brings_the_next_scan_forward(self):
        s = Scheduler()
        s.add(Monitor("f", "free"), T0)
        s.run_due(T0)
        s.change_tier("f", "pro", T0 + timedelta(hours=2))
        self.assertEqual(s.monitors["f"].next_due, T0 + timedelta(days=1))

    def test_removed_monitor_never_runs_again(self):
        s = Scheduler()
        s.add(Monitor("p", "pro"), T0)
        s.run_due(T0)
        s.remove("p")
        run_hourly(s, 72)
        self.assertEqual(len(s.results), 1)
        self.assertEqual(s.heap, [])


if __name__ == "__main__":
    unittest.main()
```

Listing 9.4 `TestScheduler.java`: the same seven tests in Java.

```
import java.time.Duration;
import java.time.ZonedDateTime;

/** Tests for the re-scan scheduler; the release gate runs these. */
public class TestScheduler {
    static final ZonedDateTime T0 = Scheduler.at("2026-09-21 09:00");

    static void runHourly(Scheduler s, int hours) {
        for (int h = 0; h <= hours; h++) s.runDue(T0.plusHours(h));
    }

    public static void main(String[] args) {
        MiniTest t = new MiniTest();
        t.test("cadence matches the plans", () -> {
            MiniTest.equal(Duration.ofDays(7), Scheduler.CADENCE.get("free"));
            MiniTest.equal(Duration.ofDays(1), Scheduler.CADENCE.get("pro"));
        });
        t.test("two weeks give 2 Free and 14 Pro scans", () -> {
            Scheduler s = new Scheduler();
            s.add(new Scheduler.Monitor("f", "free"), T0);
            s.add(new Scheduler.Monitor("p", "pro"), T0);
            runHourly(s, 14 * 24 - 1);
            long f = s.results.stream().filter(r -> r.mid().equals("f")).count();
            long p = s.results.stream().filter(r -> r.mid().equals("p")).count();
            MiniTest.equal(2L, f);
            MiniTest.equal(14L, p);
        });
        t.test("jobs come out in due order", () -> {
            Scheduler s = new Scheduler();
            s.add(new Scheduler.Monitor("late", "pro"), T0.plusHours(5));
            s.add(new Scheduler.Monitor("early", "pro"), T0.plusHours(1));
            s.runDue(T0.plusHours(6));
            MiniTest.equal("early", s.results.get(0).mid());
            MiniTest.equal("late", s.results.get(1).mid());
        });
        t.test("running twice at the same time is idempotent", () -> {
            Scheduler s = new Scheduler();
            s.add(new Scheduler.Monitor("p", "pro"), T0);
            MiniTest.equal(1, s.runDue(T0));
            MiniTest.equal(0, s.runDue(T0));
        });
        t.test("late worker skips instead of bursting", () -> {
            Scheduler s = new Scheduler();
            s.add(new Scheduler.Monitor("p", "pro"), T0);
            s.runDue(T0);
            MiniTest.equal(1, s.runDue(T0.plusDays(5).plusHours(1)));
```

Listing 9.4 (Continued) *TestScheduler.java: the same seven tests in Java.*

```
            MiniTest.equal(4, s.skipped);
            MiniTest.equal(T0.plusDays(6), s.monitors.get("p").nextDue);
        });
        t.test("upgrade brings the next scan forward", () -> {
            Scheduler s = new Scheduler();
            s.add(new Scheduler.Monitor("f", "free"), T0);
            s.runDue(T0);
            s.changeTier("f", "pro", T0.plusHours(2));
            MiniTest.equal(T0.plusDays(1), s.monitors.get("f").nextDue);
        });
        t.test("removed monitor never runs again", () -> {
            Scheduler s = new Scheduler();
            s.add(new Scheduler.Monitor("p", "pro"), T0);
            s.runDue(T0);
            s.remove("p");
            runHourly(s, 72);
            MiniTest.equal(1, s.results.size());
            MiniTest.equal(0, s.heap.size());
        });
        t.finish();
    }
}
```

```
.......
----------------------------------------------------------------------
Ran 7 tests in 0.001s

OK
ok    cadence matches the plans
ok    two weeks give 2 Free and 14 Pro scans
ok    jobs come out in due order
ok    running twice at the same time is idempotent
ok    late worker skips instead of bursting
ok    upgrade brings the next scan forward
ok    removed monitor never runs again
Ran 7 tests: 7 passed, 0 failed
```

9.10 Lab G4, Part 2: The Health Endpoint

The second part is a health endpoint that answers `GET /api/health` with the running version and the state of three dependencies. It follows the rules of Section 9.6.4.

9.10.1 The Python program

Listing 9.5 uses only the standard library's `http.server`.

Listing 9.5 `health.py`: a health endpoint that reports the running version and three checks, distinguishing down from degraded.

```
"""A health-check endpoint: GET /api/health answers which version is
running and whether its dependencies work.
usage: python health.py [PORT] [APP_DIR]"""
import json
import shutil
import socket
import sys
import time
from http.server import BaseHTTPRequestHandler, ThreadingHTTPServer
from pathlib import Path

STARTED = time.monotonic()
APP_DIR = Path(sys.argv[2]) if len(sys.argv) > 2 else Path(__file__).parent
MIN_FREE_BYTES = 1 * 1024**3          # our lab's threshold: 1 GiB
# read once at start: a running process reports the code it was started with
VERSION = (APP_DIR / "VERSION").read_text(encoding="utf-8").strip()


def check_store() -> str:
    """Can we write a file and read the same bytes back?"""
    try:
        data = APP_DIR / "_data"
        data.mkdir(exist_ok=True)
        probe = data / "health.tmp"
        probe.write_text("ping", encoding="utf-8")
        return "ok" if probe.read_text(encoding="utf-8") == "ping" else "fail"
    except OSError:
        return "fail"


def check_disk() -> str:
    return "ok" if shutil.disk_usage(APP_DIR).free >= MIN_FREE_BYTES else "fail"


def check_ai() -> str:
    """Is a local model server listening? 127.0.0.1, never 'localhost'."""
    try:
        with socket.create_connection(("127.0.0.1", 11434), timeout=0.3):
            return "ok"
    except OSError:
        return "unavailable"
```

Listing 9.5 (Continued) *health.py: a health endpoint that reports the running version and three checks, distinguishing down from degraded.*

```
def health() -> tuple[int, dict]:
    checks = {"store": check_store(), "disk": check_disk(), "ai": check_ai()}
    if "fail" in (checks["store"], checks["disk"]):
        status, code = "down", 503          # cannot serve users
    elif checks["ai"] != "ok":
        status, code = "degraded", 200      # serves, and says what is missing
    else:
        status, code = "ok", 200
    return code, {"status": status, "version": VERSION, "checks": checks,
                  "uptime_s": int(time.monotonic() - STARTED)}


class Handler(BaseHTTPRequestHandler):
    def do_GET(self) -> None:
        if self.path != "/api/health":
            self.send_error(404)
            return
        code, body = health()
        raw = (json.dumps(body, indent=2) + "\n").encode()
        self.send_response(code)
        self.send_header("Content-Type", "application/json")
        self.send_header("Cache-Control", "no-store")   # never cache health
        self.send_header("Content-Length", str(len(raw)))
        self.end_headers()
        self.wfile.write(raw)

    def log_message(self, *args) -> None:
        pass                                    # keep the console quiet


def main() -> None:
    port = int(sys.argv[1]) if len(sys.argv) > 1 else 7090
    server = ThreadingHTTPServer(("127.0.0.1", port), Handler)
    print(f"health on http://127.0.0.1:{port}/api/health", flush=True)
    server.serve_forever()


if __name__ == "__main__":
    main()
```

Three checks stand in for a real service's dependencies. `check_store` writes a small file and reads it back, the simplest honest test that storage works; in a real service this would be a `SELECT 1` against the database. `check_disk` compares free space with a threshold chosen for the lab. `check_ai` tries to open a connection to a local model server at `127.0.0.1` port 11434, the default address of Ollama's API [15], with a timeout of 0.3 seconds so that the health check itself stays fast.

`health` turns the checks into a status. A failing store or disk makes the service `down` with HTTP status 503 (*Service Unavailable*), because it cannot serve users. An unavailable AI makes it `degraded` with status 200, because it still serves, and says exactly what is missing. Everything is well only when all three pass.

The version is read **once, when the process starts**, into the constant `VERSION`. This one line is what makes the release gate of Part 3 honest: if new files are copied beside a running process but the process is not restarted, it keeps reporting the old version. The response carries `Cache-Control: no-store`, because a cached health answer is worse than none.

Start the server in one terminal, then ask it from another (Windows 10 and 11 include `curl`; in PowerShell, `Invoke-RestMethod` works too):

```
python health.py 7090
```

```
health on http://127.0.0.1:7090/api/health
HTTP/1.0 200 OK
Server: BaseHTTP/0.6 Python/3.11.15
Date: Thu, 24 Sep 2026 04:45:34 GMT
Content-Type: application/json
Cache-Control: no-store
Content-Length: 146

{
  "status": "degraded",
  "version": "0.1.0",
  "checks": {
    "store": "ok",
    "disk": "ok",
    "ai": "unavailable"
  },
  "uptime_s": 1
}
```

On the computer used for this book no model server was running, so `ai` is `unavailable` and the status is `degraded`, with HTTP 200. Start Ollama and ask again, and the status becomes `ok`. Stop the server with Ctrl+C.

9.10.2 The Java program

Listing 9.6 uses the JDK's built-in `com.sun.net.httpserver.HttpServer`. It builds the JSON text by hand, with the same indentation as Python's `json.dumps(..., indent=2)`, so that both servers return the same body.

Listing 9.6 `HealthServer.java`: the same endpoint with `com.sun.net.httpserver`.

```
import com.sun.net.httpserver.HttpExchange;
import com.sun.net.httpserver.HttpServer;
import java.io.IOException;
import java.io.OutputStream;
import java.net.InetSocketAddress;
import java.net.Socket;
import java.nio.charset.StandardCharsets;
import java.nio.file.Files;
import java.nio.file.Path;

/** A health-check endpoint: GET /api/health answers which version is
 *  running and whether its dependencies work.
 *  usage: java HealthServer [PORT] [APP_DIR] */
public class HealthServer {
    static final long STARTED = System.nanoTime();
    static final long MIN_FREE_BYTES = 1L << 30;     // our lab's threshold: 1 GiB
    static Path appDir = Path.of(".");
    static String version;       // read once at start, like the running code

    /** Can we write a file and read the same bytes back? */
    static String checkStore() {
        try {
            Path data = Files.createDirectories(appDir.resolve("_data"));
            Path probe = data.resolve("health.tmp");
            Files.writeString(probe, "ping");
            return Files.readString(probe).equals("ping") ? "ok" : "fail";
        } catch (IOException e) {
            return "fail";
        }
    }

    static String checkDisk() {
        return appDir.toFile().getUsableSpace() >= MIN_FREE_BYTES ? "ok" : "fail";
    }

    /** Is a local model server listening? 127.0.0.1, never "localhost". */
    static String checkAi() {
        try (Socket s = new Socket()) {
            s.connect(new InetSocketAddress("127.0.0.1", 11434), 300);
            return "ok";
        } catch (IOException e) {
            return "unavailable";
        }
    }
```

Listing 9.6 (Continued) *HealthServer.java: the same endpoint with com.sun.net.httpserver.*

```
    static void handle(HttpExchange ex) throws IOException {
        if (!ex.getRequestURI().getPath().equals("/api/health")) {
            ex.sendResponseHeaders(404, -1);
            ex.close();
            return;
        }
        String store = checkStore(), disk = checkDisk(), ai = checkAi();
        String status;
        int code;
        if (store.equals("fail") || disk.equals("fail")) {
            status = "down"; code = 503;             // cannot serve users
        } else if (!ai.equals("ok")) {
            status = "degraded"; code = 200;         // serves, says what is missing
        } else {
            status = "ok"; code = 200;
        }
        long uptime = (System.nanoTime() - STARTED) / 1_000_000_000L;
        String json = "{\n"
                + "  \"status\": \"" + status + "\",\n"
                + "  \"version\": \"" + version + "\",\n"
                + "  \"checks\": {\n"
                + "    \"store\": \"" + store + "\",\n"
                + "    \"disk\": \"" + disk + "\",\n"
                + "    \"ai\": \"" + ai + "\"\n"
                + "  },\n"
                + "  \"uptime_s\": " + uptime + "\n"
                + "}\n";
        byte[] raw = json.getBytes(StandardCharsets.UTF_8);
        ex.getResponseHeaders().set("Content-Type", "application/json");
        ex.getResponseHeaders().set("Cache-Control", "no-store"); // never cache
        ex.sendResponseHeaders(code, raw.length);
        try (OutputStream out = ex.getResponseBody()) {
            out.write(raw);
        }
    }
```

Listing 9.6 (Continued) *HealthServer.java: the same endpoint with com.sun.net.httpserver.*

```
    public static void main(String[] args) throws IOException {
        int port = args.length > 0 ? Integer.parseInt(args[0]) : 7090;
        if (args.length > 1) appDir = Path.of(args[1]);
        version = Files.readString(appDir.resolve("VERSION")).strip();
        HttpServer server = HttpServer.create(
                new InetSocketAddress("127.0.0.1", port), 0);
        server.createContext("/", HealthServer::handle);
        server.start();
        System.out.println("health on http://127.0.0.1:" + port + "/api/health");
    }
}
```

```
javac -d out HealthServer.java
java -cp out HealthServer 7090 .
```

```
HTTP/1.1 200 OK
Date: Thu, 24 Sep 2026 04:45:43 GMT
Content-type: application/json
Content-length: 146
Cache-control: no-store

{
  "status": "degraded",
  "version": "0.1.0",
  "checks": {
    "store": "ok",
    "disk": "ok",
    "ai": "unavailable"
  },
  "uptime_s": 1
}
```

The body is identical; only the headers differ, because the two HTTP servers write them differently (Java's server capitalizes only the first letter of each header name, which HTTP allows, since header names are case-insensitive).

9.11 Lab G4, Part 3: The Release Gate

The third part puts the first two together. The gate "deploys" the scheduler and the health endpoint into a folder that plays the role of production, runs the health server from there, and follows the six steps of **Figure 9.2**.

9.11.1 How the lab maps onto the real release

Table 9.4 shows how each step is imitated. The folders live inside a `_gate` directory next to the code: `staging`, `prod` and `backup`.

Table 9.4 How the lab gate imitates each step of the real release.

Step	Real release (Section 9.6.3)	Lab gate
1	Lock commit and clean tree	Copy the payload to `staging`, write `VERSION`, record a SHA-256 fingerprint of the source files
2	Back up production	Copy `prod` to `backup`
3	SQLite suite + PostgreSQL subset	Run the scheduler tests inside `staging`
4	Re-check the payload	Recompute the fingerprint; refuse if it changed
5	Copy and restart the service	Copy `staging` to `prod`; stop the old health server, start the new one
6	`/api/health` answers the new version	Poll `/api/health` for up to 5 seconds; else roll back from `backup`

Two flags let us make things go wrong on purpose. `--inject-bug` slips a regression into the staged copy of the scheduler after the payload is locked: it changes the Pro cadence from one day to seven. `--skip-restart` copies the new files but "forgets" to restart the server, which is exactly the lying-gate situation of Section 9.6.1.

If the health check allows up to w seconds and polls every Δ seconds, it makes at most

$$n_{\text{polls}} = \frac{w}{\Delta} = \frac{5\ \text{s}}{0.25\ \text{s}} = 20 \tag{9.8}$$

attempts. Equation (9.8) is the budget of patience: long enough for a normal restart, short enough that a failure is declared promptly.

9.11.2 The Python gate

Listing 9.7 is the gate. It uses `subprocess` to run the tests and to start the server, `hashlib` for the fingerprint, and `urllib` to ask the health endpoint.

Listing 9.7 `release_gate.py`: a six-step release gate that refuses on failing tests or a changed payload, and declares success only when `/api/health` answers the new version.

```
"""A release gate in six steps. It refuses to "deploy" when the tests
fail, and it reports success only when /api/health answers the new
version.
usage: python release_gate.py VERSION [--inject-bug] [--skip-restart]
       python release_gate.py --stop"""
import hashlib
import json
import os
import re
import shutil
import signal
import socket
import subprocess
import sys
import time
import urllib.request
from pathlib import Path

HERE = Path(__file__).resolve().parent
GATE = HERE / "_gate"
STAGING, PROD, BACKUP = GATE / "staging", GATE / "prod", GATE / "backup"
PIDFILE = GATE / "server.pid"
PAYLOAD = ["scheduler.py", "test_scheduler.py", "health.py"]
PORT = 7091
HEALTH = f"http://127.0.0.1:{PORT}/api/health"


def fingerprint(folder: Path) -> str:
    h = hashlib.sha256()
    for name in PAYLOAD:
        h.update(name.encode() + b"\0" + (folder / name).read_bytes())
    return h.hexdigest()[:12]


def copy_tree(src: Path, dst: Path) -> None:
    shutil.rmtree(dst, ignore_errors=True)
    shutil.copytree(src, dst, ignore=shutil.ignore_patterns("_data",
                                                            "__pycache__"))
```

Listing 9.7 (Continued) *release_gate.py: a six-step release gate that refuses on failing tests or a changed payload, and declares success only when /api/health answers the new version.*

```
def port_open() -> bool:
    try:
        with socket.create_connection(("127.0.0.1", PORT), timeout=0.2):
            return True
    except OSError:
        return False


def stop_server() -> None:
    if PIDFILE.exists():
        try:
            os.kill(int(PIDFILE.read_text()), signal.SIGTERM)
        except OSError:
            pass                                    # already gone
        PIDFILE.unlink()
    for _ in range(50):                             # wait for the port
        if not port_open():
            return
        time.sleep(0.1)


def start_server() -> None:
    proc = subprocess.Popen([sys.executable, str(PROD / "health.py"),
                             str(PORT), str(PROD)],
                            stdout=subprocess.DEVNULL,
                            stderr=subprocess.DEVNULL,
                            start_new_session=True)  # outlives the gate
    PIDFILE.write_text(str(proc.pid))


def running_version(want: str | None, wait_s: float) -> str | None:
    """Poll /api/health until it answers `want` or the time is up."""
    deadline = time.monotonic() + wait_s
    seen = None
    while time.monotonic() < deadline:
        try:
            with urllib.request.urlopen(HEALTH, timeout=1) as r:
                seen = json.load(r)["version"]
        except (OSError, ValueError, KeyError):
            pass                                    # not up yet
        if seen == want:
            return seen
        time.sleep(0.25)
    return seen
```

Listing 9.7 (Continued) *release_gate.py: a six-step release gate that refuses on failing tests or a changed payload, and declares success only when /api/health answers the new version.*

```
def run_tests(folder: Path) -> tuple[bool, str]:
    r = subprocess.run([sys.executable, "-m", "unittest", "test_scheduler"],
                       cwd=folder, capture_output=True, text=True)
    ran = re.search(r"Ran (\d+) test", r.stderr)
    failed = re.findall(r"^FAIL: (\w+)", r.stderr, re.M)
    summary = f"{ran.group(1) if ran else '?'} run, {len(failed)} failed"
    for name in failed:
        summary += f"\n        failed: {name}"
    return r.returncode == 0, summary


def gate(target: str, inject_bug: bool, skip_restart: bool) -> int:
    GATE.mkdir(exist_ok=True)
    print(f"[1/6] lock payload for {target}", end=": ")
    shutil.rmtree(STAGING, ignore_errors=True)
    STAGING.mkdir(parents=True)
    for name in PAYLOAD:
        shutil.copy2(HERE / name, STAGING / name)
    (STAGING / "VERSION").write_text(target + "\n")
    lock = fingerprint(HERE)
    print(lock)
    if inject_bug:                                  # a regression slips in
        f = STAGING / "scheduler.py"
        f.write_text(f.read_text().replace('"pro": timedelta(days=1)',
                                           '"pro": timedelta(days=7)'))

    print("[2/6] backup", end=": ")
    if (PROD / "VERSION").exists():
        old = (PROD / "VERSION").read_text().strip()
        copy_tree(PROD, BACKUP)
        print(f"saved production {old}")
    else:
        old = None
        print("nothing in production yet")

    ok, summary = run_tests(STAGING)
    print(f"[3/6] tests: {summary}")
    if not ok:
        print("REFUSED: tests failed; production was not touched")
        return 1
```

Listing 9.7 (Continued) *release_gate.py: a six-step release gate that refuses on failing tests or a changed payload, and declares success only when /api/health answers the new version.*

```
    print("[4/6] re-check payload", end=": ")
    if fingerprint(HERE) != lock:
        print("changed during the gate")
        print("REFUSED: payload changed; run the gate again")
        return 1
    print("unchanged")

    print("[5/6] copy and restart")
    copy_tree(STAGING, PROD)
    if not skip_restart:
        stop_server()
        start_server()

    seen = running_version(target, wait_s=5)
    print(f"[6/6] health answers: {seen}")
    if seen == target:
        print(f"RELEASED {target}")
        return 0
    print(f"NOT RELEASED: expected {target}, rolling back")
    stop_server()
    if old is None:
        print("no earlier version to restore; server stopped")
        return 1
    copy_tree(BACKUP, PROD)
    start_server()
    print(f"health after rollback: {running_version(old, wait_s=5)}")
    return 1


if __name__ == "__main__":
    if sys.argv[1:] == ["--stop"]:
        stop_server()
        print("server stopped")
        sys.exit(0)
    sys.exit(gate(sys.argv[1], "--inject-bug" in sys.argv,
                  "--skip-restart" in sys.argv))
```

A few decisions deserve comment. The production server is started with `start_new_session=True`, so it keeps running after the gate exits, as a real service would; its process number is kept in `server.pid` so the next release, or `--stop`, can stop it. `stop_server` waits until the port is actually free before a new server is started, because starting too early would fail to bind the port. `run_tests` reads the summary that `unittest` writes and extracts the names of failing tests, so the refusal says *which* rules broke. And `running_version` returns what it last saw, even when it is not what it wanted, so the report says what the service actually answered.

Here is a full session: a first release, a release with an injected bug, a release whose restart is skipped, a good release, a look at the health endpoint, and a stop.

```
$ python release_gate.py 0.1.0
[1/6] lock payload for 0.1.0: ec5b3806882b
[2/6] backup: nothing in production yet
[3/6] tests: 7 run, 0 failed
[4/6] re-check payload: unchanged
[5/6] copy and restart
[6/6] health answers: 0.1.0
RELEASED 0.1.0
$ python release_gate.py 0.2.0 --inject-bug
[1/6] lock payload for 0.2.0: ec5b3806882b
[2/6] backup: saved production 0.1.0
[3/6] tests: 7 run, 5 failed
        failed: test_cadence_matches_the_plans
        failed: test_late_worker_skips_instead_of_bursting
        failed: test_removed_monitor_never_runs_again
        failed: test_two_weeks_give_2_free_and_14_pro_scans
        failed: test_upgrade_brings_the_next_scan_forward
REFUSED: tests failed; production was not touched
$ python release_gate.py 0.2.0 --skip-restart
[1/6] lock payload for 0.2.0: ec5b3806882b
[2/6] backup: saved production 0.1.0
[3/6] tests: 7 run, 0 failed
[4/6] re-check payload: unchanged
[5/6] copy and restart
[6/6] health answers: 0.1.0
NOT RELEASED: expected 0.2.0, rolling back
health after rollback: 0.1.0
$ python release_gate.py 0.2.0
[1/6] lock payload for 0.2.0: ec5b3806882b
[2/6] backup: saved production 0.1.0
[3/6] tests: 7 run, 0 failed
[4/6] re-check payload: unchanged
[5/6] copy and restart
[6/6] health answers: 0.2.0
RELEASED 0.2.0
$ curl http://127.0.0.1:7091/api/health
{
  "status": "degraded",
  "version": "0.2.0",
  "checks": {
    "store": "ok",
    "disk": "ok",
    "ai": "unavailable"
  },
  "uptime_s": 0
}
$ python release_gate.py --stop
server stopped
```

Read the four runs in turn.

The first release has nothing to back up, passes its seven tests, and is declared released only when the health endpoint answers `0.1.0`.

The second run locks version 0.2.0, backs up production, and then the tests catch the injected regression. Five of seven tests fail, each naming a rule: the cadence, the two-week count, skip-not-burst, the upgrade, and the removed monitor, whose last entry now lingers in the heap for a week. The gate refuses. Production still runs 0.1.0, untouched.

The third run is the lying gate, caught. Tests pass, the files are copied, but no restart happens. The old process, which read its version when it started, still answers `0.1.0`. The gate does not claim success; it restores the backup and restarts, and the service answers `0.1.0` again. Without step 6, this run would have printed “RELEASED 0.2.0”.

The fourth run does everything right, and only then says `RELEASED 0.2.0`. The `curl` that follows confirms it independently.

The lab deliberately simplifies two things a real release faces. First, database schema changes: if the new version adds a column, a backup of the code alone is not enough to go back, so schema changes are usually designed so that the old version still works with the new schema. Second, the gap during the restart: for a few seconds between the old process stopping and the new one being ready, incoming requests fail. For a small system that gap is accepted and scheduled at a quiet hour; larger systems close it by running the new version beside the old one and then switching traffic. Both topics are worth exploring once the basic gate is mastered.

9.11.3 The Java gate

Listing 9.8 is the same gate in Java. It compiles the staged sources with `javac` inside the staging folder, so a payload that does not compile fails at step 3 like any failing test. It can be run without compiling it first, using the single-file source launcher: `java ReleaseGate.java 0.1.0`.

Listing 9.8 `ReleaseGate.java`: the same gate in Java, with `ProcessBuilder`, `MessageDigest` and `java.net.http.HttpClient`.

```
import java.io.IOException;
import java.net.InetSocketAddress;
import java.net.Socket;
import java.net.URI;
import java.net.http.HttpClient;
import java.net.http.HttpRequest;
import java.net.http.HttpResponse;
import java.nio.charset.StandardCharsets;
import java.nio.file.Files;
import java.nio.file.Path;
import java.nio.file.StandardCopyOption;
import java.security.MessageDigest;
import java.time.Duration;
import java.util.ArrayList;
import java.util.Comparator;
import java.util.HexFormat;
import java.util.List;
import java.util.regex.Matcher;
import java.util.regex.Pattern;
import java.util.stream.Stream;

/** A release gate in six steps. It refuses to "deploy" when the tests
 *  fail, and it reports success only when /api/health answers the new
 *  version.
 *  usage: java ReleaseGate.java VERSION [--inject-bug] [--skip-restart]
 *         java ReleaseGate.java --stop */
public class ReleaseGate {
    static final Path HERE = Path.of("").toAbsolutePath();
    static final Path GATE = HERE.resolve("_gate");
    static final Path STAGING = GATE.resolve("staging");
    static final Path PROD = GATE.resolve("prod");
    static final Path BACKUP = GATE.resolve("backup");
    static final Path PIDFILE = GATE.resolve("server.pid");
    static final List<String> PAYLOAD = List.of("Scheduler.java",
            "TestScheduler.java", "MiniTest.java", "HealthServer.java");
    static final int PORT = 7092;
    static final String HEALTH = "http://127.0.0.1:" + PORT + "/api/health";
    static final String JAVA = ProcessHandle.current().info().command()
            .orElse("java");
```

Listing 9.8 (Continued) *ReleaseGate.java: the same gate in Java, with ProcessBuilder, MessageDigest and java.net.http.HttpClient.*

```
    static String fingerprint(Path folder) throws Exception {
        MessageDigest h = MessageDigest.getInstance("SHA-256");
        for (String name : PAYLOAD) {
            h.update((name + "\0").getBytes(StandardCharsets.UTF_8));
            h.update(Files.readAllBytes(folder.resolve(name)));
        }
        return HexFormat.of().formatHex(h.digest()).substring(0, 12);
    }

    static void delete(Path dir) throws IOException {
        if (!Files.exists(dir)) return;
        try (Stream<Path> s = Files.walk(dir)) {
            for (Path p : s.sorted(Comparator.reverseOrder()).toList())
                Files.delete(p);
        }
    }

    static void copyTree(Path src, Path dst) throws IOException {
        delete(dst);
        try (Stream<Path> s = Files.walk(src)) {
            for (Path p : s.toList()) {
                if (p.toString().contains("_data")) continue;
                Path q = dst.resolve(src.relativize(p).toString());
                if (Files.isDirectory(p)) Files.createDirectories(q);
                else Files.copy(p, q, StandardCopyOption.REPLACE_EXISTING);
            }
        }
    }

    static boolean portOpen() {
        try (Socket s = new Socket()) {
            s.connect(new InetSocketAddress("127.0.0.1", PORT), 200);
            return true;
        } catch (IOException e) {
            return false;
        }
    }
```

Listing 9.8 (Continued) *ReleaseGate.java: the same gate in Java, with ProcessBuilder, MessageDigest and java.net.http.HttpClient.*

```
    static void stopServer() throws Exception {
        if (Files.exists(PIDFILE)) {
            long pid = Long.parseLong(Files.readString(PIDFILE).strip());
            ProcessHandle.of(pid).ifPresent(ProcessHandle::destroy);
            Files.delete(PIDFILE);
        }
        for (int i = 0; i < 50 && portOpen(); i++) Thread.sleep(100);
    }

    static void startServer() throws IOException {
        Process p = new ProcessBuilder(JAVA, "-cp",
                PROD.resolve("classes").toString(), "HealthServer",
                String.valueOf(PORT), PROD.toString())
                .redirectOutput(ProcessBuilder.Redirect.DISCARD)
                .redirectError(ProcessBuilder.Redirect.DISCARD)
                .start();                              // outlives the gate
        Files.writeString(PIDFILE, String.valueOf(p.pid()));
    }

    /** Poll /api/health until it answers want or the time is up. */
    static String runningVersion(String want, double waitS) throws Exception {
        HttpClient client = HttpClient.newBuilder()
                .connectTimeout(Duration.ofSeconds(1)).build();
        HttpRequest req = HttpRequest.newBuilder(URI.create(HEALTH))
                .timeout(Duration.ofSeconds(1)).build();
        Pattern v = Pattern.compile("\"version\": \"([^\"]+)\"");
        long deadline = System.nanoTime() + (long) (waitS * 1e9);
        String seen = null;
        while (System.nanoTime() < deadline) {
            try {
                String body = client.send(req,
                        HttpResponse.BodyHandlers.ofString()).body();
                Matcher m = v.matcher(body);
                if (m.find()) seen = m.group(1);
            } catch (IOException e) {
                // not up yet
            }
            if (seen != null && seen.equals(want)) return seen;
            Thread.sleep(250);
        }
        return seen;
    }
```

Listing 9.8 (Continued) *ReleaseGate.java: the same gate in Java, with ProcessBuilder, MessageDigest and java.net.http.HttpClient.*

```
    /** Compile and run the tests in a folder; returns {passed, summary}. */
    static Object[] runTests(Path folder) throws Exception {
        List<String> cmd = new ArrayList<>(List.of("javac", "-d", "classes"));
        cmd.addAll(PAYLOAD);
        Process javac = new ProcessBuilder(cmd).directory(folder.toFile())
                .redirectErrorStream(true).start();
        javac.getInputStream().readAllBytes();
        if (javac.waitFor() != 0) return new Object[] {false, "did not compile"};
        Process run = new ProcessBuilder(JAVA, "-cp", "classes", "TestScheduler")
                .directory(folder.toFile()).start();
        String out = new String(run.getInputStream().readAllBytes(),
                StandardCharsets.UTF_8);
        run.getErrorStream().readAllBytes();
        boolean ok = run.waitFor() == 0;
        Matcher m = Pattern.compile("Ran (\\d+) tests: \\d+ passed, (\\d+) failed")
                .matcher(out);
        StringBuilder summary = new StringBuilder(
                m.find() ? m.group(1) + " run, " + m.group(2) + " failed" : "?");
        Matcher f = Pattern.compile("(?m)^FAIL  ([^:]+):").matcher(out);
        while (f.find()) summary.append("\n        failed: ").append(f.group(1));
        return new Object[] {ok, summary.toString()};
    }

    static int gate(String target, boolean injectBug, boolean skipRestart)
            throws Exception {
        Files.createDirectories(GATE);
        System.out.print("[1/6] lock payload for " + target + ": ");
        delete(STAGING);
        Files.createDirectories(STAGING);
        for (String name : PAYLOAD)
            Files.copy(HERE.resolve(name), STAGING.resolve(name));
        Files.writeString(STAGING.resolve("VERSION"), target + "\n");
        String lock = fingerprint(HERE);
        System.out.println(lock);
        if (injectBug) {                                  // a regression slips in
            Path f = STAGING.resolve("Scheduler.java");
            Files.writeString(f, Files.readString(f).replace(
                    "\"pro\", Duration.ofDays(1)", "\"pro\", Duration.ofDays(7)"));
        }
```

Listing 9.8 (Continued) *ReleaseGate.java: the same gate in Java, with ProcessBuilder, MessageDigest and java.net.http.HttpClient.*

```
        System.out.print("[2/6] backup: ");
        String old = null;
        if (Files.exists(PROD.resolve("VERSION"))) {
            old = Files.readString(PROD.resolve("VERSION")).strip();
            copyTree(PROD, BACKUP);
            System.out.println("saved production " + old);
        } else {
            System.out.println("nothing in production yet");
        }

        Object[] tests = runTests(STAGING);
        System.out.println("[3/6] tests: " + tests[1]);
        if (!(Boolean) tests[0]) {
            System.out.println("REFUSED: tests failed; production was not touched");
            return 1;
        }

        System.out.print("[4/6] re-check payload: ");
        if (!fingerprint(HERE).equals(lock)) {
            System.out.println("changed during the gate");
            System.out.println("REFUSED: payload changed; run the gate again");
            return 1;
        }
        System.out.println("unchanged");

        System.out.println("[5/6] copy and restart");
        copyTree(STAGING, PROD);
        if (!skipRestart) {
            stopServer();
            startServer();
        }

        String seen = runningVersion(target, 5);
        System.out.println("[6/6] health answers: " + seen);
        if (target.equals(seen)) {
            System.out.println("RELEASED " + target);
            return 0;
        }
        System.out.println("NOT RELEASED: expected " + target + ", rolling back");
        stopServer();
        if (old == null) {
            System.out.println("no earlier version to restore; server stopped");
            return 1;
        }
        copyTree(BACKUP, PROD);
        startServer();
```

Listing 9.8 (Continued) *ReleaseGate.java: the same gate in Java, with ProcessBuilder, MessageDigest and java.net.http.HttpClient.*

```
        System.out.println("health after rollback: " + runningVersion(old, 5));
        return 1;
    }

    public static void main(String[] args) throws Exception {
        if (args.length == 1 && args[0].equals("--stop")) {
            stopServer();
            System.out.println("server stopped");
            return;
        }
        List<String> a = List.of(args);
        System.exit(gate(args[0], a.contains("--inject-bug"),
                         a.contains("--skip-restart")));
    }
}
```

```
$ java ReleaseGate.java 0.1.0
[1/6] lock payload for 0.1.0: d085bacb2b69
[2/6] backup: nothing in production yet
[3/6] tests: 7 run, 0 failed
[4/6] re-check payload: unchanged
[5/6] copy and restart
[6/6] health answers: 0.1.0
RELEASED 0.1.0
$ java ReleaseGate.java 0.2.0 --inject-bug
[1/6] lock payload for 0.2.0: d085bacb2b69
[2/6] backup: saved production 0.1.0
[3/6] tests: 7 run, 5 failed
        failed: cadence matches the plans
        failed: two weeks give 2 Free and 14 Pro scans
        failed: late worker skips instead of bursting
        failed: upgrade brings the next scan forward
        failed: removed monitor never runs again
REFUSED: tests failed; production was not touched
$ java ReleaseGate.java 0.2.0 --skip-restart
[1/6] lock payload for 0.2.0: d085bacb2b69
[2/6] backup: saved production 0.1.0
[3/6] tests: 7 run, 0 failed
[4/6] re-check payload: unchanged
[5/6] copy and restart
[6/6] health answers: 0.1.0
NOT RELEASED: expected 0.2.0, rolling back
health after rollback: 0.1.0
$ java ReleaseGate.java 0.2.0
[1/6] lock payload for 0.2.0: d085bacb2b69
[2/6] backup: saved production 0.1.0
[3/6] tests: 7 run, 0 failed
[4/6] re-check payload: unchanged
[5/6] copy and restart
[6/6] health answers: 0.2.0
RELEASED 0.2.0
$ java ReleaseGate.java --stop
server stopped
```

The story is the same, line for line. The fingerprints differ from Python's because they are computed over different files, and the failing tests are listed in the order the Java runner executes them. The gate uses port 7092 so that the Python and Java gates can run side by side without conflict.

> **WARNING.** Always finish with `--stop`. The production copy of the health server is designed to outlive the gate, like a real service; if you do not stop it, it keeps running in the background until you close the session or restart the computer.

9.12 Profitina in Practice

In the September 2026 release, the ideas of this chapter appear in the product as follows.

- **Architecture.** One Windows 11 machine, native; FastAPI on Uvicorn as a Windows service under NSSM; PostgreSQL with the development database isolated by role permissions; Server-Sent Events for real time; Ollama with a quality lane and a fast lane, called at `127.0.0.1`; a WordPress marketing site beside the app; everything reached through the Cloudflare edge and an outbound-only tunnel.
- **Deliberately absent.** No hyperscaler, no Kubernetes, no microservices, no automatic scaling; regional scale is a later phase of the roadmap.
- **Release gate.** The full SQLite suite (1,577 passed, 1 skipped at the time of writing) and a PostgreSQL 16 priority subset (440 passed) must be green, with warnings treated as errors, JavaScript UI guards and browser checks alongside.
- **Health-verified release.** Six steps, ending with success only when `/api/health` answers the new version; the release runs a copy of its script.
- **Observability.** Server errors recorded without bodies or headers and with addresses hashed; spike alerts; the health endpoint reporting database, AI, disk, uptime and recent errors; error logs purged after 90 days.
- **Performance.** The feed for 30 posts at 11 milliseconds after the September 2026 fix, with 12 queries, no new connections and no model calls, guarded by the rule that per-item render functions never touch the model or open connections.
- **Monitoring cadence.** Free monitors re-scanned weekly, Pro monitors daily, as in the plan table of Chapter 8.

Lab G4 reproduces none of Profitina's code. It teaches the three operational ideas with small programs on Profitina Laundry's pages, which is the right place for hands-on practice in this book series.

9.13 Common Misunderstandings

MISUNDERSTANDING 1: "A serious product needs many servers." A product needs the reliability its users require. Much of that comes from fast recovery (Equation (9.1)), backups, tests and careful releases, all of which work on one machine. More machines add capacity and new failure modes at the same time.

MISUNDERSTANDING 2: "The tests passed, so the release succeeded." Tests show the code works in the test environment. Only the running service can show that the new version is live, which is why the release asks `/api/health`.

MISUNDERSTANDING 3: "A health check should say OK if the process is alive." A useful health check reports whether the service can do its job, separates "down" from "degraded", and names the version it is running.

MISUNDERSTANDING 4: "The feed was slow because the machine is small." It was slow because of three software patterns: a model call on the read path, a connection per call, and N+1 queries. The same machine now serves the page in 11 milliseconds.

MISUNDERSTANDING 5: "`localhost` and `127.0.0.1` are the same thing." They often are, but `localhost` may resolve to IPv6 first. Against a server that listens only on IPv4, that can cost seconds per call; measured on the product, 2.25 seconds against 0.17.

> **MISUNDERSTANDING 6: "A scheduler that was down should catch up by running every missed job."** Replaying missed periodic scans produces a burst of identical work at the worst moment. Run once, skip the missed slots (Equation (9.7)), and return to the grid.
>
> **MISUNDERSTANDING 7: "Printing a zero when a query fails is a safe default."** A wrong answer that looks normal cannot be noticed. Errors must be loud.

9.14 Chapter Summary

- Availability is $A = \text{MTBF}/(\text{MTBF} + \text{MTTR})$ (Equation (9.1)); faster recovery often raises it more cheaply than rarer failure. The down-time allowed at each target follows Equation (9.2) and **Table 9.1**.
- Profitina's architecture (**Figure 9.1**, **Table 9.2**) is one native Windows 11 host with FastAPI on Uvicorn under NSSM, PostgreSQL with role-isolated development data, Server-Sent Events, two local AI lanes via Ollama, and a WordPress marketing site, all reached through the Cloudflare edge and an outbound-only tunnel.
- One machine is a stage-appropriate choice: most outages in small teams come from changes, and most reliability comes from process.
- A release gate must stop the release when it fails. Profitina's suite (**Table 9.3**) has 1,577 passing tests on SQLite and 440 on PostgreSQL 16 at the time of writing, with warnings as errors; it grew from incidents such as the SQL dialect trap and the silent zero.
- The six-step release (**Figure 9.2**) locks, backs up, tests, re-checks, copies and restarts, and declares success only when `/api/health` answers the new version; this closed the gap exposed when a batch script's success message lied.
- Switching channels went from 868 to 11 milliseconds (**Figure 9.3**, Equation (9.4)) by removing a model call from the read path, pooling connections, and replacing N+1 queries (Equation (9.3)) with prefetched hydration; a sequence-number guard fixed the stale-response race (**Figure 9.4**).
- A scheduler is a min-heap of due times (**Figure 9.5**); operations cost $O(\log n)$ (Equation (9.5)), runs follow Equation (9.6), late workers skip (Equation (9.7)), and stale entries are discarded lazily.
- Lab G4 built the scheduler, a health endpoint that separates down from degraded, and a release gate that refuses failing tests and rolls back a version that does not answer, polling at most Equation (9.8) times.

9.15 Exercises

Exercise 9.1. A service fails on average once every 14 days and takes 6 minutes to restore. (a) Compute its availability with Equation (9.1). (b) How long would the mean time to repair have to be to reach 99.99% with the same failure rate? (c) Using **Table 9.1**, which change is more realistic for a small team: halving the failure rate, or cutting the repair time to one minute?

Exercise 9.2. A page lists $N = 50$ posts, and naive code runs $k = 6$ extra queries per post. (a) How many queries does Equation (9.3) predict? (b) If each query takes 1.5 milliseconds and a new connection adds 4 milliseconds per query, estimate the time with and without a connection pool. (c) With prefetched hydration the page needs one query for the list and one per kind of data (writers, channels, counts): how many queries is that, and why does it no longer depend on N?

Exercise 9.3. Starting from an empty min-heap, insert the due times 9, 4, 7, 1, 8, 2 (in hours) in that order, drawing the array after each insertion. Then pop the minimum twice, drawing the

array after each pop. How many element swaps did each operation need, and how does that compare with $\log_2 n$?

Exercise 9.4. Extend the scheduler with a *weekly trend*: run one Pro monitor for 21 simulated days, group its scores by ISO week, print the mean for each week, and print an alert when the week-over-week change in the mean exceeds a chosen $\delta = 5$ points. Write it in Python and in Java.

Exercise 9.5. How would you test that `health.py` returns HTTP 503 with status `down` when free disk space is below the threshold, without filling your disk? Describe the change to the program and the test.

Exercise 9.6. Step 4 of the gate refuses when the payload changes during the tests, but none of the demonstration runs triggers it. Describe a way to trigger it deliberately, and predict the gate's output.

Exercise 9.7. Write a small class `LatestOnly` (Python and Java) that implements the sequence-number guard of **Figure 9.4**: `request()` returns a new sequence number, and `reply(seq, payload)` shows the payload only if `seq` is the latest. Demonstrate that a late reply for channel A is dropped after the user has switched to channel B.

Exercise 9.8. Explain, in terms of what each step *observes*, why the release in Section 9.6.1 could report success for a version that was not running, and why step 6 of the new release cannot.

Exercise 9.9. List four pieces of information that a public health endpoint may safely report and four that it must never report. Justify each item in one sentence.

Exercise 9.10. Some platforms distinguish a *liveness* check ("is the process alive?") from a *readiness* check ("can it serve requests now?"). Which of the three statuses of `health.py` corresponds to which answer, and what should a service manager like NSSM do on each?

9.16 Solutions to the Exercises

Solution 9.1. (a) MTBF is $14 \times 24 \times 60 = 20{,}160$ minutes, MTTR is 6 minutes, so $A = 20{,}160/20{,}166 \approx 0.99970$, that is, 99.970%. (b) We need $\text{MTBF}/(\text{MTBF} + \text{MTTR}) \geq 0.9999$, so $\text{MTTR} \leq \text{MTBF} \times (1/0.9999 - 1) \approx 20{,}160 \times 0.0001 \approx 2.0$ minutes. (c) Cutting repair time to one minute gives $A = 20{,}160/20{,}161 \approx 99.995\%$, already beyond four nines, and usually needs only automation (a service manager, a scripted rollback). Halving the failure rate would give $40{,}320/40{,}326 \approx 99.985\%$ and usually needs deep work on causes. For a small team, faster repair is the more realistic lever, which is the lesson of Equation (9.1).

Solution 9.2. (a) $Q(50) = 1 + 6 \times 50 = 301$ queries. (b) Without a pool, each query costs $1.5 + 4 = 5.5$ ms, so about $301 \times 5.5 \approx 1{,}656$ ms; with a pool, about $301 \times 1.5 \approx 452$ ms. The pool helps, but the page is still slow, because the query count is the real problem. (c) One query for the list plus one each for writers, channels and counts is 4 queries, about 6 ms at 1.5 ms each. Each prefetch query asks for all the needed rows at once (for example, all writers whose identifiers appear in the list), so the number of queries depends on the number of *kinds* of data, not on the number of posts.

Solution 9.3. Writing the heap as an array (index 0 is the root; the children of index i are $2i + 1$ and $2i + 2$), each insertion places the new value at the end and lets it rise while it is smaller than its parent; each pop moves the last value to the root and lets it sink toward its smaller child:

```
insert 9   [9]                0 swaps
insert 4   [4, 9]             1 swap   (4 < 9)
insert 7   [4, 9, 7]          0 swaps  (7 > 4)
insert 1   [1, 4, 7, 9]       2 swaps  (1 < 9, then 1 < 4)
insert 8   [1, 4, 7, 9, 8]    0 swaps  (8 > 4)
insert 2   [1, 4, 2, 9, 8, 7] 1 swap   (2 < 7, then 2 > 1)
pop -> 1   [2, 4, 7, 9, 8]    1 swap   (7 moved to root, 7 > 2)
pop -> 2   [4, 8, 7, 9]       1 swap   (8 moved to root, 8 > 4, 8 < 9)
```

No operation needed more than 2 swaps. The heap never held more than 6 entries, and a binary tree of 6 nodes has height $\lfloor \log_2 6 \rfloor = 2$, so no operation can need more than 2 swaps: that is the $O(\log n)$ cost behind Equation (9.5).

Solution 9.4. **Listing 9.9** and **Listing 9.10** run the scheduler for 21 days and group the results with the ISO week number (`isocalendar().week` in Python, `IsoFields.WEEK_OF_WEEK_BASED_YEAR` in Java).

Listing 9.9 `weekly_trend.py`: ISO-week means of one monitor's scores, with a week-over-week alert.

```
"""Exercise 9.4: weekly trend per monitor (ISO weeks) with an alert when
the week-over-week change is larger than a chosen delta."""
from collections import defaultdict
from datetime import timedelta

from scheduler import Monitor, Scheduler, at

DELTA = 5.0                                  # the reader's choice, in points

s = Scheduler()
start = at("2026-09-21 09:00")
s.add(Monitor("laundry-prices", "pro"), start)
for h in range(0, 21 * 24, 3):
    s.run_due(start + timedelta(hours=h))

weeks: dict[tuple[str, int], list[int]] = defaultdict(list)
for mid, when, score in s.results:
    weeks[(mid, when.isocalendar().week)].append(score)

previous = None
for (mid, week), scores in sorted(weeks.items()):
    mean = sum(scores) / len(scores)
    line = f"{mid}  ISO week {week}: {len(scores)} scans, mean {mean:.1f}"
    if previous is not None and abs(mean - previous) > DELTA:
        line += f"  ALERT: changed by {mean - previous:+.1f}"
    print(line)
    previous = mean
```

Listing 9.10 `WeeklyTrend.java`: the same trend in Java.

```java
import java.time.ZonedDateTime;
import java.time.temporal.IsoFields;
import java.util.ArrayList;
import java.util.List;
import java.util.Map;
import java.util.TreeMap;

/** Exercise 9.4: weekly trend per monitor (ISO weeks) with an alert when
 *  the week-over-week change is larger than a chosen delta. */
public class WeeklyTrend {
    static final double DELTA = 5.0;         // the reader's choice, in points

    public static void main(String[] args) {
        Scheduler s = new Scheduler();
        ZonedDateTime start = Scheduler.at("2026-09-21 09:00");
        s.add(new Scheduler.Monitor("laundry-prices", "pro"), start);
        for (int h = 0; h < 21 * 24; h += 3) s.runDue(start.plusHours(h));

        Map<Integer, List<Integer>> weeks = new TreeMap<>();
        for (Scheduler.Result r : s.results)
            weeks.computeIfAbsent(r.when().get(IsoFields.WEEK_OF_WEEK_BASED_YEAR),
                    k -> new ArrayList<>()).add(r.score());

        Double previous = null;
        for (Map.Entry<Integer, List<Integer>> e : weeks.entrySet()) {
            List<Integer> scores = e.getValue();
            double mean = scores.stream().mapToInt(Integer::intValue).average()
                    .orElse(0);
            String line = String.format("laundry-prices  ISO week %d: %d scans, "
                    + "mean %.1f", e.getKey(), scores.size(), mean);
            if (previous != null && Math.abs(mean - previous) > DELTA)
                line += String.format("  ALERT: changed by %+.1f", mean - previous);
            System.out.println(line);
            previous = mean;
        }
    }
}
```

Both print:

```
laundry-prices  ISO week 39: 7 scans, mean 76.7
laundry-prices  ISO week 40: 7 scans, mean 59.0  ALERT: changed by -17.7
laundry-prices  ISO week 41: 7 scans, mean 70.9  ALERT: changed by +11.9
```

The alerts fire because `stub_scan` produces noise, not because anything changed on the page; a real scanner would move far less from day to day. The exercise is about the mechanics: grouping by ISO week (which begins on Monday, the same day the simulation starts), averaging, and comparing with a threshold the reader chooses. Choosing δ is a product decision: too small and users learn to ignore alerts, too large and real changes go unnoticed.

Solution 9.5. Make the threshold configurable from outside, for example by reading an environment variable `HEALTH_MIN_FREE_BYTES` with the lab default as fallback. A test then starts the server with the variable set to a number larger than any disk (say 10^{18} bytes), requests `/api/health`, and checks for status 503 and `"status": "down"`. The same approach tests the store check, by pointing `APP_DIR` at a folder that cannot be written. The general lesson: design checks so that their failure can be *caused* in a test, or the failure path will never be exercised until it matters.

Solution 9.6. Step 4 compares the fingerprint of the source files before and after the tests. To trigger it, change a payload file while the tests run: for example, add a line to `test_scheduler.py` that sleeps for ten seconds, start the gate, and during the pause edit a comment in `scheduler.py`

in the source folder. The output would show steps 1 to 3 as usual, then `[4/6] re-check payload: changed during the gate` and `REFUSED: payload changed; run the gate again`, with production untouched. The tests passed, but they tested the old text; the gate refuses to copy code it did not test.

Solution 9.7. **Listing 9.11** and **Listing 9.12** keep a counter of the latest request and compare it with the number carried by each reply.

Listing 9.11 `latest_only.py`: the sequence-number guard against stale replies.

```
"""Exercise 9.7: drop stale replies with a request sequence number."""


class LatestOnly:
    def __init__(self):
        self.latest = 0
        self.shown: list[str] = []

    def request(self) -> int:
        self.latest += 1                # every new request gets a number
        return self.latest

    def reply(self, seq: int, payload: str) -> bool:
        if seq != self.latest:
            return False                # stale: a newer request exists
        self.shown.append(payload)
        return True


panel = LatestOnly()
a = panel.request()                     # user opens channel A
b = panel.request()                     # ...and quickly switches to B
print("reply for B shown:", panel.reply(b, "posts of B"))
print("late reply for A shown:", panel.reply(a, "posts of A"))
print("panel shows:", panel.shown)
```

Listing 9.12 `LatestOnly.java`: the same guard in Java.

```java
import java.util.ArrayList;
import java.util.List;

/** Exercise 9.7: drop stale replies with a request sequence number. */
public class LatestOnly {
    int latest = 0;
    final List<String> shown = new ArrayList<>();

    int request() {
        return ++latest;                // every new request gets a number
    }

    boolean reply(int seq, String payload) {
        if (seq != latest) return false; // stale: a newer request exists
        shown.add(payload);
        return true;
    }

    public static void main(String[] args) {
        LatestOnly panel = new LatestOnly();
        int a = panel.request();        // user opens channel A
        int b = panel.request();        // ...and quickly switches to B
        System.out.println("reply for B shown: "
                + (panel.reply(b, "posts of B") ? "True" : "False"));
        System.out.println("late reply for A shown: "
                + (panel.reply(a, "posts of A") ? "True" : "False"));
        System.out.println("panel shows: ['" + String.join("', '", panel.shown)
                + "']");
    }
}
```

```
reply for B shown: True
late reply for A shown: False
panel shows: ['posts of B']
```

In a browser, the same idea is combined with `AbortController`, which also cancels the obsolete network request [11]; the guard is still needed, because a reply may already be on its way when the abort is issued.

Solution 9.8. The old release observed *its own progress*: reaching the last lines of the script. When `cmd.exe` resumed at a shifted byte position, the script skipped work and still reached those lines, so the observation was true (the end was reached) but irrelevant (the release was incomplete). Step 6 observes *the running service*: the version string that the process reports, which it read when it started. A process that was not restarted, or not replaced, cannot report a version it was never started with, so the observation can only succeed if the new version is actually serving.

Solution 9.9. Safe: the running version (needed to verify releases); an overall status such as ok, degraded or down (needed by monitors); per-dependency states such as “database ok” (needed to locate failures); uptime (shows unexpected restarts). Never: credentials or connection strings (would hand an attacker access); file-system paths and machine names (help an attacker map the system); personal data or user counts by name (privacy, UU PDP); exact security thresholds such as rate limits (help an attacker stay just below them). If more detail is needed for operators, it belongs behind authentication, not on a public endpoint [10].

Solution 9.10. Liveness asks whether the process should be left running; readiness asks whether it should receive requests. In `health.py`, any answer at all means *alive*. `ok` and `degraded` both mean *ready* (degraded is ready with reduced features); `down` means *not ready*. A service manager should restart the process only when it is not alive (no answer, or a crash), which is what NSSM does. It should not restart on `down`, because if the database is unreachable a restart does not help

and may make recovery noisier; that case needs an alert to a person. On `degraded` no action is needed beyond reporting, which is exactly why degraded exists as a separate status.

References

[1] B. Beyer, C. Jones, J. Petoff, and N. R. Murphy, Eds., *Site Reliability Engineering: How Google Runs Production Systems*. Sebastopol, CA, USA: O'Reilly Media, 2016, ch. 3 "Embracing Risk" and ch. 8 "Release Engineering."

[2] Cloudflare, "Cloudflare Tunnel," *Cloudflare One Docs*. [Online]. Available: https://developers.cloudflare.com/cloudflare-one/networks/connectors/cloudflare-tunnel/

[3] NSSM, "NSSM: the Non-Sucking Service Manager." [Online]. Available: https://nssm.cc/

[4] S. Ramírez, "FastAPI," documentation. [Online]. Available: https://fastapi.tiangolo.com/

[5] Encode, "Uvicorn: an ASGI web server for Python," documentation. [Online]. Available: https://www.uvicorn.org/

[6] WHATWG, "Server-sent events," in *HTML Living Standard*, sec. 9.2. [Online]. Available: https://html.spec.whatwg.org/multipage/server-sent-events.html

[7] The PostgreSQL Global Development Group, "Database roles," *PostgreSQL Documentation*, ch. 21. [Online]. Available: https://www.postgresql.org/docs/current/user-manag.html

[8] Profitina, *Profitina Whitepaper, Public Version v24*, Sep. 2026. [Online]. Available: https://app.profitina.com/static/whitepaper/

[9] N. Harvey and D. DeBellis, "Announcing the 2025 DORA report," *Google Cloud Blog*, Sep. 23, 2025. [Online]. Available: https://cloud.google.com/blog/products/ai-machine-learning/announcing-the-2025-dora-report

[10] Microsoft, "Health Endpoint Monitoring pattern," *Azure Architecture Center*. [Online]. Available: https://learn.microsoft.com/en-us/azure/architecture/patterns/health-endpoint-monitoring

[11] Mozilla, "AbortController," *MDN Web Docs*. [Online]. Available: https://developer.mozilla.org/en-US/docs/Web/API/AbortController

[12] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, *Introduction to Algorithms*, 4th ed. Cambridge, MA, USA: MIT Press, 2022, ch. 6.

[13] Python Software Foundation, "heapq: Heap queue algorithm," *Python 3.14 Documentation*. [Online]. Available: https://docs.python.org/3/library/heapq.html

[14] Oracle, "Class PriorityQueue," *Java SE 25 & JDK 25 API Specification*, 2025. [Online]. Available: https://docs.oracle.com/en/java/javase/25/docs/api/java.base/java/util/PriorityQueue.html

[15] Ollama, "API reference," documentation. [Online]. Available: https://docs.ollama.com/api

Chapter 10

The Marketing Site as a GEO Case Study

By the end of this chapter you will be able to describe how profitina.com, Profitina's public marketing site, is built and why it is built that way: a WordPress installation with a custom bilingual theme, Indonesian and English side by side, sitting behind two cache layers, one at the web host and one at the edge of a content delivery network. You will be able to read a `robots.txt` file exactly as RFC 9309 prescribes, including the parts that surprise people: how groups for the same crawler are merged, why a crawler with its own group ignores the `*` group, and what a crawler must do when the file answers with an error. You will know the real incident that taught the Profitina team to check the *live* `robots.txt` and not the file on the server, and why that incident is easy to repeat. You will be able to say what `llms.txt` is, how many sites publish one and how few of those files are ever read, with the sources. You will understand HTTP caching well enough to compute how long a stale page can survive when two caches sit one in front of the other, and why the order in which you purge them matters. You will know which structured data the marketing site carries and what that data can and cannot do after Google's May 2026 change to FAQ results. Finally, you will have built and run an auditor, in Python and in Java, that checks `robots.txt`, `llms.txt`, cache headers and JSON-LD on four small sample sites served on your own computer, and that finds, among other things, the very fault that once hid profitina.com from two AI crawlers.

10.1 Where We Left Off

Chapter 9 took the application apart: one Windows host, a Cloudflare Tunnel that opens no inbound port, FastAPI and PostgreSQL, two lanes of local AI, and a release gate that declares success only when the health endpoint answers with the new version. That chapter was about `app.profitina.com`, the product that users log in to.

This chapter turns to the other half of the brand, `profitina.com`, the site that a newcomer usually meets first. It is tempting to treat a marketing site as a brochure, something a designer makes once and nobody thinks about again. For a company whose product is *AI visibility*, that would be a strange position to take. Profitina tells its users that an answer engine can only quote what its crawler was allowed to fetch, what it could parse, and what was still true when it arrived. Its own site is therefore the first place where that advice is tested, in public, with the whole world able to check.

That makes the marketing site an unusually honest case study. Every idea from Chapter 2, where Section 2.7 introduced structured data, `robots.txt` and `llms.txt`, and where lab G6 generated all three for Profitina Laundry, has to survive contact with real infrastructure here: a content management system with its own habits, a hosting company with its own cache, and a content delivery network with its own opinions about which crawlers should be allowed in. Some of those ideas survived unchanged. One of them failed silently, and the story of that failure is the heart of this chapter.

The chapter moves from the outside in. Section 10.2 describes the two surfaces of the brand and the stack behind the marketing site. Section 10.3 looks at how a bilingual theme is built properly and tells a small but instructive bug story. Sections 10.4 and 10.5 study `robots.txt` in depth and then tell the incident. Section 10.6 examines `llms.txt` with the latest adoption and usage numbers. Section 10.7 explains HTTP caching and the purge order of two cache layers, with a little arithmetic. Section 10.8 covers the structured data on the site. Section 10.9 builds the auditor, and Sections 10.10 to 10.14 close with Profitina in practice, common misunderstandings, a summary, exercises and full solutions.

10.2 Two Surfaces, One Brand

10.2.1 Why a separate marketing site at all

Profitina has two public addresses, and they do different jobs.

`app.profitina.com` is the application. It is the place where a user registers, runs a scan, generates a draft, watches a trend line, posts in a community channel, and exercises their data rights. It is written in Python with FastAPI, it keeps its data in PostgreSQL, and every page of it is produced by code the team writes and tests. Its title in the browser tab is “Profitina — Dasbor GEO”.

`profitina.com` is the marketing site. It explains what Profitina is, publishes articles, stories, tips and course material, hosts the whitepaper link and the frequently asked questions, and carries an embedded, read-only view of the community feed. Its headline, on 24 September 2026, reads “A self-reliant GEO platform — from us, for everyone”, and the page title is “Profitina – Gather · Grow · Give” [1].

Why not put everything in one application? Because the two jobs have opposite needs. The application changes every few days, is tested by well over a thousand automated tests, and must never show one user another user’s data. The marketing site changes when someone writes an article, must be editable by people who do not write code, and shows the same page to everyone. A content management system is the right tool for the second job, and a custom application is the right tool for the first. Keeping them apart also keeps their failures apart: a broken theme cannot take down the scanner, and a failed release of the application cannot hide the articles.

10.2.2 Why WordPress

The marketing site runs on WordPress. That choice needs little defense in 2026. According to W3Techs, WordPress was used in September 2026 by 40.2% of all websites and by 58.8% of all websites whose content management system is known [2]. For a small team, that popularity is not a matter of fashion but of supply: hosting companies tune their servers for it, cache plugins exist for it, translators know it, and almost any problem has already been met and solved by someone else.

The price of that popularity is that a WordPress site is shaped by conventions the team did not choose. A theme runs inside WordPress’s own page-building loop and shares its global variables. Plugins add their own headers, their own files and their own caches. A hosting company may add a server-level cache, and a content delivery network may add another. Each of these is useful, and each is a place where what the team *wrote* and what a visitor *receives* can drift apart. The rest of this chapter is, in a sense, a catalogue of those places.

10.2.3 The stack, from the visitor inward

Figure 10.1 shows the path a request takes on its way to profitina.com.

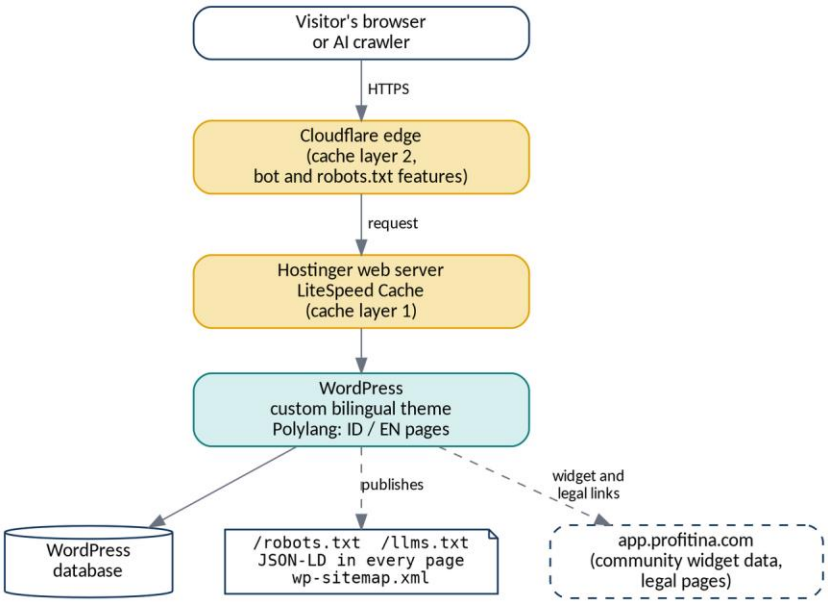


Figure 10.1 The marketing site's stack. A request passes the Cloudflare edge (cache layer 2) and the host's LiteSpeed Cache (cache layer 1) before WordPress and its bilingual theme build a page; the application is reached only through the embedded widget and the legal links.

Reading it from top to bottom:

1. **The visitor.** A person with a browser, or a program: a search crawler, an AI crawler, a link-preview bot, an SEO audit tool. They all ask for the same URLs, and they all receive whatever the layers in front of WordPress decide to give them.
2. **The Cloudflare edge.** Cloudflare terminates HTTPS close to the visitor, filters abusive traffic, and can keep copies of responses. It also offers features that *change* responses, including, as Section 10.5 will show, a managed `robots.txt`. This is cache layer 2, the outer layer.
3. **The web host.** The site is hosted at Hostinger on a LiteSpeed web server, with the LiteSpeed Cache plugin for WordPress. LiteSpeed keeps ready-made copies of pages so that WordPress does not have to rebuild every page for every visitor. This is cache layer 1, the inner layer.
4. **WordPress and the theme.** When neither cache has an answer, WordPress runs, the custom theme builds the page in the requested language, and the database supplies the content.
5. **What the site publishes.** Besides its pages, the site serves `/robots.txt`, `/llms.txt`, an XML sitemap at `/wp-sitemap.xml`, and JSON-LD structured data inside its pages.
6. **The link to the application.** The community widget on the marketing pages reads the public feed from the application, and the footer's Privacy Policy and Terms links point to `app.profitina.com/legal/`, the single home of those documents.

That last point is a deliberate design decision with a short name in the team's notes: *remove duplication, do not synchronize it* [3]. The Privacy Policy used to exist in two places. Two copies of a legal text are two chances to be wrong, and sooner or later one of them is. Since late August 2026 the policy lives only in the application, and the marketing site summarizes and links. Chapter 11 returns to that decision from the legal side.

> **KEY INSIGHT.** Every layer between the writer and the reader is a place where the published version can differ from the written version. A GEO audit must therefore look at what the *world* receives, at the public URL, not at the file on the server or in the repository.

10.2.4 What lives where

Table 10.1 summarizes the division of labor.

Table 10.1 The two public surfaces of the Profitina brand and what each one is responsible for.

Concern	Profitina.com (WordPress)	App.profitina.com (FastAPI)
Audience	Anyone, including every crawler	Registered users, plus public read-only views
Content	Articles, stories, tips, courses, FAQ, whitepaper link	Scans, drafts, trends, community, account, PROFIT
Languages	Indonesian and English pages (Polylang)	One interface with an ID/EN dictionary
Machine-readable files	`robots.txt`, `llms.txt`, `wp-sitemap.xml`, JSON-LD	`robots.txt`, `sitemap.xml`, API endpoints
Caching	LiteSpeed Cache at the host, Cloudflare at the edge	Application-level; the edge passes the tunnel through
Legal texts	Summary and links only	The single home: `/legal/privacy`, `/legal/terms`
Changes	When an editor publishes	Through the tested release gate of Chapter 9

10.3 A Bilingual Theme Done Properly

10.3.1 Two languages, two sets of pages

Profitina is Indonesia-first, and its audience reads Indonesian, English, or both. A bilingual site can be built in two very different ways. The cheap way is to write every page once and translate it on the fly, in the browser, with a machine-translation widget. The careful way is to keep two real versions of each page, one per language, each with its own address, its own title, and its own text written or at least reviewed by a person.

profitina.com takes the careful way. Its theme is a custom WordPress theme, and the language layer is the Polylang plugin, which lets editors create a translation of each post, page, category and menu and link the translations together [4]. Polylang keeps separate menus for each language, which is more work for an editor but produces menus that read naturally in each language instead of translated labels squeezed into a layout designed for the other one.

Why does this matter for GEO? Because an answer engine that reads a page needs to know which language it is reading and whether another version exists. Three small signals carry that information:

- the `lang` attribute of the `<html>` element, `id` or `en`, which tells every program the language of the page;
- the address of each version, which must be stable so that it can be cited;
- `hreflang` links, which tell search engines that the Indonesian page and the English page are translations of each other, so that a reader is sent to the right one [5].

A machine-translated page has none of these properly: there is one address, one declared language, and a text that exists only in the visitor's browser. No crawler ever sees the translation, so no engine can ever quote it.

10.3.2 The `$id` bug: a lesson in shared names

The theme's history contains a small bug that is worth telling because its pattern appears in every framework that runs your code inside its own loop.

Inside the WordPress loop, the function that prepares each post for display, `setup_postdata()` (called by `the_post()`), sets a number of global variables, among them `$id`, the numeric identifier of the current post [6]. An early version of the profitina.com theme, in the 2.20 line, also used a variable called `$id` in its template code, to hold the identifier of the language-specific label of a hero section. In PHP, a template included by WordPress shares the scope in which those globals live. The two uses of the same name therefore collided. The symptom was the worst kind: not a crash, not an error message, but an Indonesian label appearing on an English hero now and then, depending on the order in which things happened inside a given page's loop.

A first look at a symptom like "the languages are mixed up" goes to the language logic. That logic was correct. The fault was one level below, in a name. The fix was equally simple and, importantly, a root-cause fix rather than a patch: the theme's variable was renamed to `$pf_id`, with a prefix that WordPress will never use, across a short run of point releases of the theme. The general rule it teaches is one that every framework user learns sooner or later:

> **WARNING.** When your code runs inside someone else's loop, every unqualified name you choose may already belong to the framework. Prefix your own names (`pf_`, a namespace, a class), and when a bug appears intermittently and depends on the order of events, suspect shared state before you suspect your logic.

The same habit of versioning each change of the theme, with a number and a line in a changelog, is what made the bug traceable at all. "Introduced before this version, fixed across these releases" is a sentence that only a versioned project can write.

10.4 robots.txt from First Principles

10.4.1 What Chapter 2 established

Section 2.7.3 introduced `robots.txt`: a plain text file at the root of a site, standardized as the Robots Exclusion Protocol in RFC 9309 in September 2022 [7], made of groups, each group a list of `User-agent` lines followed by `Allow` and `Disallow` rules. It also gave the decision procedure: a crawler uses the groups that name it, or the `*` group if none does; the longest matching rule wins; on a tie `Allow` wins; and if no rule matches, the path is allowed. The crawler table of Section 2.7.4 listed the crawlers of OpenAI, Anthropic, Perplexity, Google and Microsoft, and lab G6 generated a file that allows the ones a small business usually wants.

This section goes further, into the parts of RFC 9309 that matter when the file you wrote is not the only file involved, or not the file that arrives. We need four more ideas: how the lines of a file become groups, what happens when two groups name the same crawler, what wildcards mean, and what a crawler does when the file cannot be read.

10.4.2 From lines to groups

A crawler reads the file line by line. Everything after a `#` is a comment. Each remaining line has the form `key: value`, and keys are case-insensitive. A new **group** starts at a `User-agent` line that follows a rule line (or at the first `User-agent` line of the file); consecutive `User-agent` lines belong to the same group. Rule lines (`Allow`, `Disallow`) attach to the group currently open. Other lines, such as `Sitemap`, do not belong to any group and may appear anywhere.

> **DEFINITION.** A *group* of a `robots.txt` file is a maximal run of one or more `User-agent` lines followed by the rule lines up to the next `User-agent` line that follows a rule. The *product token* in a `User-agent` line is matched against a crawler's name without regard to case.

Take Profitina Laundry's sample file from this chapter's lab:

```
# Profitina Laundry (sample site for Chapter 10)
User-agent: *
Content-Signal: search=yes, ai-input=yes, ai-train=no
Allow: /
Disallow: /admin/

User-agent: GPTBot
User-agent: OAI-SearchBot
User-agent: ClaudeBot
User-agent: PerplexityBot
User-agent: Google-Extended
Allow: /
Disallow: /admin/

Sitemap: http://127.0.0.1:7100/sitemap.xml
```

It has two groups. The first names only `*` and has two rules. The second names five crawlers, because five `User-agent` lines follow each other with no rule between them, and has the same two rules. The `Sitemap` line belongs to no group. The `Content-Signal` line is not part of RFC 9309 at all; Section 10.5.4 explains where it comes from, and a conforming reader simply skips a key it does not know.

10.4.3 Choosing the groups: merging and the fallback

Let a be the name of a crawler, written in lower case, and let agents(g) be the set of lower-cased product tokens of group g. The set of groups that crawler a must obey is

$$G(a) = \begin{cases} \{\, g : a \in \text{agents}(g) \,\} & \text{if this set is not empty,} \\ \{\, g : * \in \text{agents}(g) \,\} & \text{otherwise.} \end{cases} \tag{10.1}$$

Equation (10.1) contains two rules that RFC 9309 states in so many words [7]. First, if *more than one* group names the crawler, their rules are combined into one list; a crawler does not stop at the first group that mentions it. Second, the `*` group is a *fallback*, not a default that is added to everything: a crawler that is named by any group ignores the `*` group completely.

The second rule is the one that bites. Suppose a file says

```
User-agent: *
Allow: /

User-agent: GPTBot
Disallow: /private/
```

A site owner reading it might think "everybody may read everything, and GPTBot may not read `/private/`". That is correct, but only by luck. Change the second group to `Disallow: /` and GPTBot may read nothing at all, even though the first group says `Allow: /` for "everybody". GPTBot is not part of "everybody" any more: it has a group of its own. Section 10.5 shows how exactly this rule, combined with a file the site owner never wrote, hid profitina.com from two crawlers.

10.4.4 Choosing the rule: longest match, allow on a tie

Once the groups are chosen, their rules form one list $R(a)$. For a path p, such as `/blog/post-1`, let $M(a,p)$ be the rules of $R(a)$ whose pattern matches p, and let $\ell(r)$ be the length in octets of the pattern of rule r. The deciding rule is

$$r^* = \arg\max_{r \in M(a,p)} (\ell(r),\ [\, r \text{ is Allow} \,]) \tag{10.2}$$

where the pairs are compared lexicographically: first by length, and only between equally long patterns by whether the rule is an `Allow` (counted as 1) or a `Disallow` (counted as 0). If $M(a,p)$ is empty, the path is allowed. If it is not, the path is allowed exactly when r^* is an `Allow` rule. Equation (10.2) is the whole of RFC 9309's precedence rule in one line: "the most specific match found MUST be used", and "if an allow rule and a disallow rule are equivalent, then the allow rule SHOULD be used" [7]. Google's documentation says the same for its crawlers [8].

> **EXAMPLE.** For the group `User-agent: *` with `Disallow: /` and `Allow: /blog/`, and the path `/blog/post-1`: both patterns match, with lengths 1 and 6. The pair for `Disallow: /` is (1,0) and the pair for `Allow: /blog/` is (6,1); the larger pair is (6,1), so the path is allowed. For the path `/about`, only `Disallow: /` matches, M has one element, and the path is blocked.

The order of the lines in the file plays no part in Equation (10.2). That is a real difference from an older, informal reading of `robots.txt` in which the first matching rule decided, and it is the difference that Python's own `urllib.robotparser` implemented for many years. Section 10.9.6 runs both readings side by side.

10.4.5 Wildcards

RFC 9309 gives two characters a special meaning inside a pattern [7]:

- `*` matches any sequence of characters, including none, so `Disallow: /*.pdf` blocks `/files/report.pdf` and `/a.pdf`;
- `$` at the very end anchors the pattern to the end of the path, so `Disallow: /*.pdf$` blocks `/a.pdf` but not `/a.pdf?download=1`.

Without a trailing `$`, a pattern matches every path that *begins* with it, which is why `Disallow: /admin` also blocks `/administrator`. The length $\ell(r)$ in Equation (10.2) is the length of the pattern as written, wildcards included.

The lab's reader implements the two wildcards by turning a pattern into a regular expression: the parts between `*` characters are escaped literally, the `*` becomes `.*`, and the pattern is anchored at the end only when it ended in `$`. Section 10.9.3 shows the dozen lines that do it.

10.4.6 When the file cannot be read

A crawler that asks for `/robots.txt` does not always get a file. RFC 9309 distinguishes the cases by HTTP status, and **Table 10.2** summarizes them [7], [8].

Table 10.2 How RFC 9309 tells crawlers to treat each kind of response to a request for `robots.txt`. Google additionally treats a 429 like a server error.

Response to GET `/robots.txt`	What a crawler must or may do	Why
2xx with a file	Parse it and obey it	The normal case
3xx redirect	Follow at least five redirects, then treat as unavailable	Sites move files
4xx, "unavailable" (404, 410, 403…)	May crawl everything, as if no file existed	The owner published no rules
5xx or no answer, "unreachable"	Must assume complete disallow	Rules may exist but cannot be read
Any response larger than 500 KiB	May ignore the rest; must parse at least 500 KiB	A bound on parsing work

The asymmetry between 4xx and 5xx is deliberate and easy to get backwards. A 404 means "there are no rules": the site did not publish a file, so nothing is forbidden. A 500 means "there are rules but I cannot show them to you": the safe assumption is that everything is forbidden. A site whose `robots.txt` breaks with a server error therefore disappears from well-behaved crawlers, and nothing on its visible pages looks wrong. Google adds a practical detail: for the first 12 hours of such errors it stops crawling the site while it keeps trying to fetch the file [8]. One of the four sample sites of the lab, `broken`, has exactly this fault.

RFC 9309 also allows crawlers to cache the file, but says they "SHOULD NOT use the cached version for more than 24 hours, unless the robots.txt file is unreachable" [7], and Google says it "generally caches the contents of robots.txt file for up to 24 hours" [8]. That matters in Section 10.7: a fix to `robots.txt` may take up to a day to reach a crawler even when your own caches are empty, so a further day of caching on top of it is a day you did not need to lose.

10.4.7 Reading and access are different things

Two limits of `robots.txt`, already stated in Chapter 2, deserve repeating because the incident of the next section is about the opposite mistake. `robots.txt` is a *request* that well-behaved crawlers honor, not an access control: a page that must be private needs authentication. And the file is public: a `Disallow` line advertises the path it hides. The mistake of Section 10.5 is not that a page was exposed; it is that pages meant for everyone were quietly withheld from some of the readers that mattered most.

10.5 The robots.txt Lesson: When the Served File Is Not Your File

10.5.1 What happened

profitina.com had a `robots.txt` that did what a GEO platform's own site should do: it allowed the search and AI crawlers by name. The file was correct in the theme and correct on the server. Yet for a while, two of the crawlers it welcomed, OpenAI's `GPTBot` and Anthropic's `ClaudeBot`, were told to stay away.

The cause was a feature of the content delivery network in front of the site, a *managed* `robots.txt`. When it is switched on, the edge does not simply pass the site's file through. It writes its own block of rules and puts it in front of the site's rules, and every visitor, every crawler included, receives the combination. Cloudflare's own documentation describes the behavior precisely: "Cloudflare will prepend our managed robots.txt before your existing robots.txt,

combining both into a single response" [9]. The managed block names a list of AI crawlers, among them `GPTBot`, `ClaudeBot`, `Google-Extended`, `Applebot-Extended`, `CCBot`, `Bytespider`, `Amazonbot` and `meta-externalagent`, and disallows them [9].

Nothing on the site's pages changed. Nothing in the team's repository was wrong. A person who opened the theme's `robots.txt` or the file on the server saw the intended rules. Only a request to the public address, `https://profitina.com/robots.txt`, showed what the world actually received. The team's rule that came out of the episode is short: *always verify the live URL, bypassing any cache* [3].

10.5.2 Why prepending is enough to block

Prepending sounds harmless. The site's own rules are still in the file, after all. **Figure 10.2** shows why it is not harmless, using Equation (10.1).

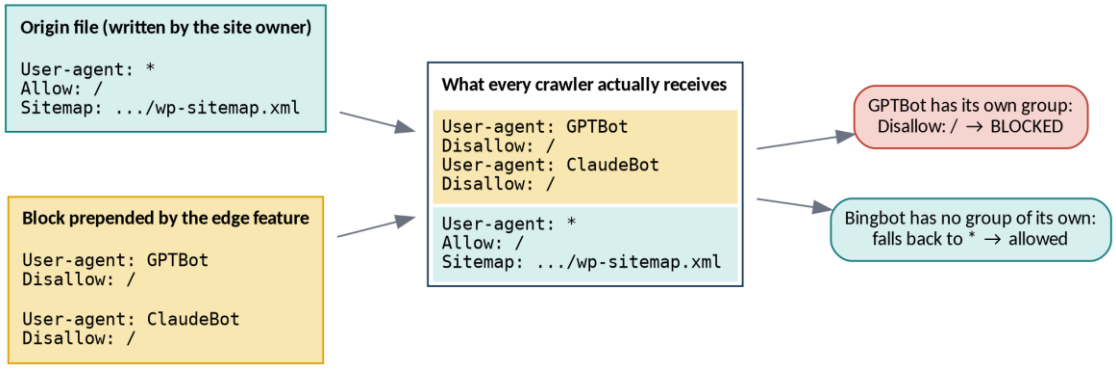


Figure 10.2 A managed block prepended at the edge. The origin's `*` group still allows everyone, but a crawler named in the managed block now has a group of its own and never looks at `*`; a crawler that is not named still falls back to `*`.

Suppose the site's own file said only `User-agent: *` and `Allow: /`, which is a common and perfectly reasonable way to say "everyone is welcome". After prepending, the served file has a group for `GPTBot` with `Disallow: /`, a group for `ClaudeBot` with `Disallow: /`, and the site's `*` group at the end. For `GPTBot`, the first case of Equation (10.1) applies: a group names it, so $G(\text{gptbot})$ is that group alone, and the `*` group is never consulted. Its only rule, `Disallow: /`, matches every path. `GPTBot` is blocked everywhere. `Bingbot`, which the managed block does not name, falls into the second case of Equation (10.1), obeys the `*` group, and is allowed.

What if the site's own file had named `GPTBot` explicitly, as profitina.com's file does today? Then Equation (10.1) merges the two groups that name it, and the rule list contains both `Disallow: /` from the edge and `Allow: /` from the site. Both patterns have length 1, so by Equation (10.2) the tie goes to `Allow`, and a crawler that follows RFC 9309 to the letter would be allowed. But three facts make that no comfort. First, the site's rules might be more specific, such as `Allow: /blog/`, and then the edge's `Disallow: /` wins for every path outside `/blog/`. Second, not every crawler implements merging and the tie rule; a crawler that stops at the first group naming it sees only the edge's `Disallow: /`. Third, and most important, a file that contains both "go away" and "welcome" for the same reader is a file whose meaning depends on the reader's parser. A site that wants to be quoted cannot leave that to chance.

> **WARNING.** A group that names a crawler takes that crawler out of the `*` group entirely. Any layer that adds a group for a crawler, whether a CDN feature, a security plugin or an SEO plugin, silently overrides what your `*` group says about it.

10.5.3 Defaults move, too

The managed `robots.txt` is not an exotic setting. Cloudflare introduced it on 1 July 2025, as part of a set of new controls over AI crawlers [10]. A year later, on 1 July 2026, it announced finer controls that separate AI traffic into search, agent and training uses, with new defaults from 15 September 2026 for newly onboarded domains that show ads: training and agent bots blocked on those pages, search bots allowed [11]. Whether any of this is right for a given site is the owner's decision; the point for an engineer is that the behavior of the layer in front of your site is set partly by someone else, and it changes over time. A check that passed in June may fail in September without anyone on your team touching anything.

10.5.4 The file profitina.com serves today

On 24 September 2026, the live file at `https://profitina.com/robots.txt` keeps `/wp-admin/` closed (with the usual exception for `/wp-admin/admin-ajax.php`), explicitly allows a long list of search and AI crawlers, including `GPTBot`, `ClaudeBot`, `Claude-Web` and `PerplexityBot`, points to the sitemap at `/wp-sitemap.xml`, and mentions the site's `llms.txt` [12]. It also carries a line that RFC 9309 does not define:

```
Content-Signal: search=yes, ai-input=yes, ai-train=no
```

This is a *content signal*, a machine-readable statement of how the owner would like the content to be *used*, as opposed to whether it may be *fetched*. The vocabulary, proposed by Cloudflare, has three keys: `search` (building a search index and showing results), `ai-input` (feeding content to a model at answer time, as in retrieval-augmented generation) and `ai-train` (training or fine-tuning models) [9]. profitina.com's line says: index us, quote us in answers, but please do not train on us. That combination is coherent with a GEO platform's goals, and it shows that "allow AI or not" is really several separate questions.

Two cautions keep the content signal in proportion. It is a statement of preference that a crawler may or may not read; it grants or withholds nothing by itself. And the access rules still decide: a crawler that is disallowed never reads the content, whatever the signal says.

10.5.5 The general lesson

The incident generalizes into a checklist that any site can apply and that the lab of Section 10.9 automates:

1. **Fetch the public URL, not the file.** What matters is what `https://your-site/robots.txt` returns to a stranger.
2. **Fetch it twice, once bypassing caches.** A request with a unique query string, such as `?audit-bypass=1`, usually misses every cache keyed on the full URL and reaches the origin. If the two answers differ, a cache holds an old copy.
3. **Compare with the origin itself.** If the host offers a way to reach the origin server directly, compare that too. If the cache-bypassing answer and the origin's answer differ, something between them *rewrites* the file; it is not a matter of stale copies at all.
4. **Evaluate the rules as a crawler would.** Do not eyeball the text; run it through a reader that implements Equation (10.1) and Equation (10.2), for every crawler you care about.
5. **Repeat on a schedule.** Layers change their defaults (Section 10.5.3). Chapter 9's scheduler is the natural home for such a check.

10.6 llms.txt: What It Is, and What It Is Not

10.6.1 The proposal

Section 2.7.5 introduced `llms.txt`, proposed on 3 September 2024 by Jeremy Howard as a Markdown file at a site's root that gives language models a short, curated map of the site [13]. Its format is easy to state precisely:

1. an H1 line with the name of the site, "the only required section";
2. a block quote with a short summary;
3. optionally, paragraphs or lists of further detail, but no headings;
4. optionally, sections headed with H2 containing lists of links, each `- [title](url): note`;
5. by convention, an `## Optional` section for links that can be skipped when context is short [13].

The proposal also suggests offering a clean Markdown version of important pages at the same address with `.md` appended. The idea is sensible: a language model's context window is small compared with a website, and a page's HTML is full of navigation, scripts and styling that are noise to a model.

profitina.com publishes an `llms.txt` [14]. Its content describes Profitina as a GEO platform that measures how ready brands are to be cited by AI answer engines from real site signals rather than by querying the engines, mentions the local two-lane Qwen models, the Indonesia-first design and UU PDP compliance, and states plainly that PROFIT is a utility token on Solana devnet, not a security and not an investment. It is, in other words, a faithful one-page summary of what the rest of the site says.

10.6.2 Adoption versus use

Here the honest numbers matter more than the idea, and there are two very different numbers.

The first is **adoption**: how many sites publish the file. A scan on 18 September 2026 of the Tranco list of 17 September 2026 found `llms.txt` (or `llms-full.txt`) on 93 of the top 1,000 domains, 9.3%, and on 830 of the top 10,000, 8.3% [15]. Among a broader set of more than three million monitored sites, one tracker counted 36,120 sites with the file in May 2026 against 4,088 a year earlier, an 8.8-fold increase [16].

The second is **use**: how often anyone reads the file. Ahrefs studied its server logs for 137,210 domains in May 2026. Of the files that existed, 97% received no request at all during the month [17]. Among the files that were requested, 96% of the requests came from bots rather than people, and 77% of the bots were not AI tools at all but SEO audit tools, general crawlers and technology profilers; the retrieval bots behind AI search products accounted for only 1.1% of all requests [17]. The study also observed that AI bots did not go looking for `llms.txt` files that did not exist; the file is fetched when something points to it, not proactively [17]. **Figure 10.3** puts the two sides next to each other.

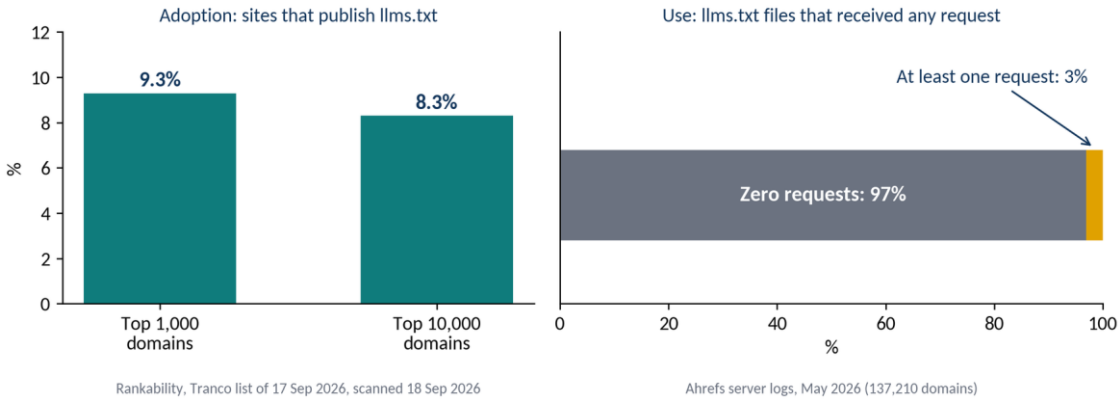


Figure 10.3 Adoption versus use of `llms.txt` in 2026. About one in eleven top sites publishes the file, but in May 2026 almost none of the files received a single request.

The adoption figures can be checked with one line of arithmetic, a habit worth keeping whenever a percentage and a count are both quoted: $93/1000 = 0.093 = 9.3\%$ and $830/10\,000 = 0.083 = 8.3\%$. The growth factor too: $36\,120/4\,088 \approx 8.84$.

The organization that ran the adoption scan states the limit of its own data in one sentence that deserves to be quoted in full: publishing the file "does not establish search-ranking improvements, AI citations, or training permissions" [15].

10.6.3 Profitina's position: cheap hygiene, no promises

Why publish a file that almost nobody reads? Because it costs almost nothing, because it describes the site honestly, and because some tools, such as coding assistants and agents that a person points at a site, do read it when asked. What Profitina does not do is claim that the file causes citations. That restraint is the same one that runs through the whole product: visibility results are "methodological estimates, not guarantees" [3], and a platform that measured readiness while overselling a file would be contradicting itself. The lab's auditor prints the same caveat under every report.

> **KEY INSIGHT.** Distinguish *adoption* (how many publish) from *use* (how many are read) and both from *effect* (whether it changes outcomes). For `llms.txt` in September 2026, adoption is modest, use is tiny, and effect is unproven. `robots.txt`, by contrast, has an effect every time a crawler arrives.

10.7 Two Cache Layers and the Purge Order

10.7.1 Why caches exist

A cache is a store of ready-made answers. Building a WordPress page means running PHP, querying a database and assembling a template; serving a stored copy means reading a file. On a small host the difference is between tens of milliseconds and hundreds, and under a burst of traffic it is the difference between a site that stays up and one that falls over. A content delivery network adds a second benefit: copies stored at the edge are served from a location close to the visitor, so the request does not cross oceans.

The cost of a cache is that a copy can be *stale*: the origin has changed, and the copy has not. Everything in this section is about controlling that cost.

10.7.2 Freshness in HTTP

HTTP caching is specified in RFC 9111 [18]. A response is stored together with its **age**, the time since the origin produced it, and it may be served without asking the origin as long as it is **fresh**:

$$\text{fresh} \iff \text{current\_age} < \text{freshness\_lifetime} \tag{10.3}$$

The **freshness lifetime** comes from the `Cache-Control` header that the origin (or a layer) sends. **Table 10.3** lists the directives that matter for a marketing site.

Table 10.3 The HTTP caching directives and headers the lab's auditor reads, as RFC 9111 defines them for shared caches.

Directive	Meaning for a shared cache (CDN, host cache)
`max-age=N`	Fresh for *N* seconds; applies to every cache, including the browser
`s-maxage=N`	Fresh for *N* seconds in *shared* caches only; overrides `max-age` there
`no-cache`	May be stored, but must be revalidated with the origin before each use
`no-store`	Must not be stored at all
`private`	Only the user's own browser may store it; shared caches must not
`public`	Shared caches may store it even where they otherwise would not
`ETag`, `Last-Modified` (headers)	*Validators*: let a cache ask "has it changed?" and receive a short 304 answer
`Age` (header)	Seconds the copy has already spent in caches

For a shared cache, then, the lifetime L of a response is

$$L = \begin{cases} s & \text{if } \texttt{s-maxage} = s \text{ is present,} \\ m & \text{else if } \texttt{max-age} = m \text{ is present,} \\ \text{a heuristic} & \text{otherwise.} \end{cases} \tag{10.4}$$

The last line of Equation (10.4) is worth noticing: a response with no `Cache-Control` at all does not mean "do not cache". RFC 9111 lets a cache *guess* a lifetime, typically from how long ago the resource was last modified [18]. A site that sends no caching headers has handed the decision to someone else.

Two newer headers refine the picture. RFC 9213 defines `CDN-Cache-Control`, a header with the same syntax as `Cache-Control` that only CDN caches obey; when present, a CDN cache "MUST ignore the Cache-Control and Expires header fields" [19]. This lets an origin say "browsers: one minute; my CDN: one hour" in a single response. And each layer usually reports what it did in a header of its own: Cloudflare sends `CF-Cache-Status` with values such as `HIT`, `MISS`, `EXPIRED`, `BYPASS` and `DYNAMIC` [20], and the LiteSpeed Cache plugin sends `X-LiteSpeed-Cache: hit` or `miss` [21].

10.7.3 What each layer caches by default

Defaults matter because nobody configures everything. Cloudflare's documentation states that its CDN "does not cache HTML or JSON by default", caches a fixed list of static file extensions such as images, scripts, style sheets and fonts, and that "by default Cloudflare caches a website's robots.txt" [22]. It also does not cache a response whose `Cache-Control` is `private`, `no-store`, `no-cache` or `max-age=0` [22]. A site that wants its HTML pages cached at the edge must say so with a rule, and once it does, those pages are subject to the same staleness as everything else.

The LiteSpeed Cache plugin, on the other hand, exists precisely to cache WordPress pages at the host, and it purges its copies automatically when a post is published or updated [21]. The two layers therefore have different habits: the inner layer knows about WordPress and purges itself on content changes; the outer layer knows nothing about WordPress and keeps what it was told to keep, for as long as it was told.

10.7.4 The purge order

After a change that the automatic purges do not cover, such as a new theme version or a change to `robots.txt`, both caches must be cleared by hand. The Profitina team's deployment notes put the order in capital letters: **LiteSpeed first, then Cloudflare**. **Figure 10.4** shows why.

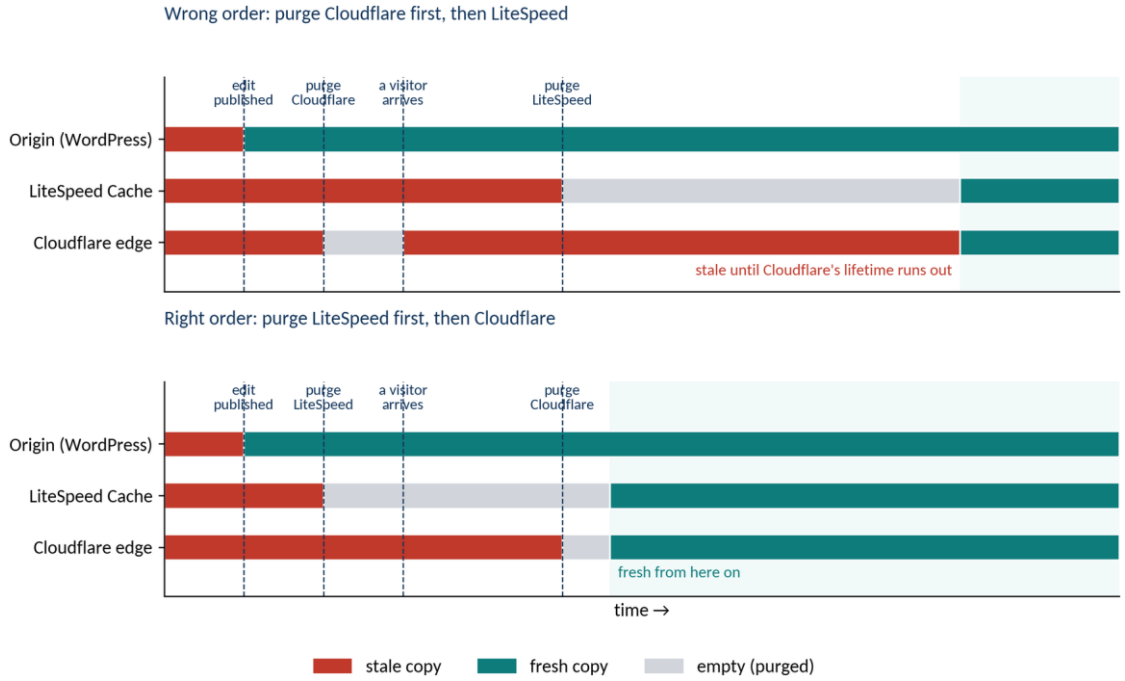


Figure 10.4 Purging two cache layers. In the wrong order, a visitor who arrives between the purges makes the edge refill itself from the still-stale inner cache, and the stale copy lives on for a full edge lifetime; in the right order, the stale window ends with the second purge.

Read the upper panel first. The edit is published at the origin. The operator purges Cloudflare. Before the operator gets to LiteSpeed, a visitor arrives. Cloudflare has nothing, so it asks the host; LiteSpeed still has the old page and serves it; Cloudflare stores that old page as a fresh copy, with a full lifetime ahead of it. Now the operator purges LiteSpeed. The host is clean, but the edge is not, and the edge will keep serving the old page until its lifetime runs out. Purging in this order did not just fail to help: it *renewed* the stale copy.

In the lower panel, LiteSpeed is purged first. A visitor arriving between the purges still gets the old page from Cloudflare, which has not been purged yet, but nothing new is stored anywhere, because Cloudflare did not need to ask. When Cloudflare is purged, the next request misses at the edge, misses at the host, reaches WordPress, and the new page fills both caches on its way back.

We can put a number on the difference. Let the edit happen at time t_0, the first purge at t_1, the second at t_2, and let T_C be the edge's lifetime for the page. In the right order the stale window ends with the second purge:

$$W_{\text{right}} = t_2 - t_0 \tag{10.5}$$

In the wrong order, a request at any time t_r with $t_1 < t_r < t_2$ refills the edge with the old copy, which then lives for T_C:

$$W_{\text{wrong}} = t_r + T_C - t_0 \;\leq\; t_2 + T_C - t_0 \tag{10.6}$$

EXAMPLE. A `robots.txt` fix is published at $t_0 = 0$. The operator purges one cache at $t_1 = 2$ minutes and the other at $t_2 = 5$ minutes. The edge keeps `robots.txt` for $T_C = 4$ hours, that is, 240 minutes. In the right order, Equation (10.5) gives $W = 5 - 0 = 5$ minutes. In the wrong order, a crawler that happens to ask at $t_r = 3$ minutes gives, by Equation (10.6), $W = 3 + 240 - 0 = 243$ minutes, just over four hours. Add the up to 24 hours for which a crawler may keep its own copy of `robots.txt` (Section 10.4.6), and a two-minute slip in the order becomes more than a day of the old rules.

A request between the purges is not a rare event on a public site. Crawlers, uptime monitors and link-preview bots ask for pages continuously. On a site with any traffic at all, the upper panel is the normal outcome of the wrong order, not an unlucky one.

DEFINITION. Two caches form *layers* when one of them (the outer) fetches from the other (the inner) on a miss. The *purge order* rule for layered caches is: purge from the inside out, the layer nearest the origin first and the layer nearest the visitor last.

10.7.5 Choosing lifetimes

The purge order limits the damage of a change; sensible lifetimes limit how often you need to purge at all. Three rules of thumb follow from the discussion so far, and the lab's auditor checks the first two:

- **HTML pages:** a lifetime of minutes to hours, not days, unless every publish reliably purges every layer. A month-long `s-maxage` on HTML means that a missed purge is a month-long mistake.
- **`robots.txt`:** no more than a day in any shared cache, because crawlers already keep their own copy for up to 24 hours [7], [8].
- **Validators:** send an `ETag` or `Last-Modified`, so that caches can revalidate cheaply instead of downloading the whole page again.

Static assets whose names change when their content changes, such as `style.3f2a.css`, can be cached for a year; that is what "cache busting" in Chapter 9's release process achieves for the application.

10.8 Structured Data on the Marketing Site

10.8.1 What the pages carry

Chapter 2 explained schema.org and JSON-LD and generated both for Profitina Laundry. profitina.com uses the same vocabulary on its own pages: FAQ structured data for its frequently asked questions and a `dateModified` date on its content, so that any program reading a page knows what it is, which questions it answers, and how recent it is. The vocabulary itself keeps evolving; schema.org's current release at the time of writing is version 30.1, of 16 September 2026 [23].

Three kinds of node matter most for a site like this:

- an **organization** node (`Organization`, or a more specific `LocalBusiness` for a shop), with at least `name` and `url`, so that the entity behind the site is unambiguous;
- **FAQ** nodes (`FAQPage` with `Question` and `Answer`), which package short, self-contained, quotable statements;
- a **page** node (`WebPage` or `Article`) with `dateModified`, which tells a reader how fresh the facts are.

10.8.2 After May 2026: FAQ markup without FAQ results

For years, many sites added `FAQPage` markup mainly because Google showed FAQ *rich results*, expandable questions under a search result. That incentive is gone. Google stopped showing FAQ rich results on 7 May 2026, with reporting and testing support following in June and API support ending in August 2026 [24]. Google also said that the markup can stay: "unused structured data doesn't cause problems for Search, and FAQPage is still a valid Schema.org type" [24].

For GEO the reasoning never depended on a visual feature of one search engine. A question with an explicit, short answer is exactly the kind of passage that the GEO study behind Chapter 2 found most likely to be used in generated answers, where adding citations, quotations and statistics raised a source's visibility by up to about 40% [25]. Structured data makes such passages unambiguous to a machine. It does not oblige any engine to quote them, and Profitina never says that it does.

10.8.3 Freshness belongs in the data

The `dateModified` property connects structured data with the caching story of Section 10.7. A cached page carries the `dateModified` of the moment it was cached. If a stale copy survives at the edge, the date inside it survives too, and a program that reads the page is told, truthfully as far as the copy goes, that nothing has changed since then. The lab uses exactly this: it compares the `dateModified` of the page as served with the `dateModified` of the page at the origin, and a difference is proof of a stale cache that no header could hide.

10.8.4 Three faults that make structured data useless

Structured data fails quietly, like `robots.txt`. A browser never shows it, so a broken block goes unnoticed until someone looks. The three faults the lab looks for are the commonest:

1. **Invalid JSON.** One trailing comma, one unescaped quote, and the whole block is rejected by any strict parser. The lab's `broken` sample site has a trailing comma.
2. **Missing identity.** An organization without `url` is an organization that cannot be linked to its website with certainty.
3. **Stale dates.** A `dateModified` that disagrees between the served page and the origin (Section 10.8.3).

Validity is only the floor. Chapter 2's rule, that structured data must say exactly what the visible page says, still applies, and no automatic check can confirm it; a person reading both can.

10.9 Lab: A GEO Hygiene Auditor in Python and Java

10.9.1 What we will build

The lab turns the checklist of Section 10.5.5, the caching rules of Section 10.7 and the structured-data faults of Section 10.8.4 into a program. The program is an *auditor*: given the address of a site,

it fetches a handful of public files, applies the rules of this chapter, and prints one line per finding, each marked `PASS`, `INFO`, `WARN` or `FAIL`, followed by a summary table. **Figure 10.5** shows the order of its requests.

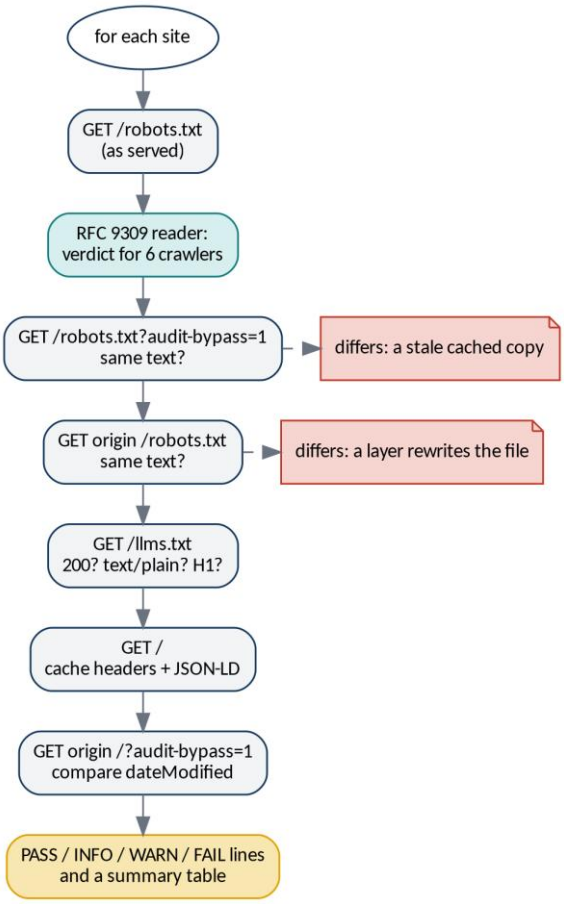


Figure 10.5 The auditor's requests for one site. Two comparisons tell a stale cached copy apart from a file that a layer rewrites; the last request fetches a fresh home page to compare `dateModified`.

A program that fetches other people's pages must be polite, and the auditor is: it makes one request per resource, waits at most five seconds for an answer, and says who it is in its `User-Agent` header. It never runs against third-party sites in bulk. To give it something realistic to audit without touching anyone's server, the lab ships four small **sample sites** and a server that runs them on your own computer, on the address `127.0.0.1`, which no other computer can reach. **Table 10.4** introduces them.

Table 10.4 The four sample sites of the lab. Two of them also expose their origin on a second port, so that the auditor can compare what visitors receive with what the origin holds.

Site	Public port	Origin port	What is wrong with it (by design)
`laundry`	7100	—	Nothing: Profitina Laundry, configured as this chapter recommends
`cdnblock`	7101	7105	An edge layer prepends a managed block to `robots.txt` (Section 10.5)
`stale`	7102	7106	HTML cached for a month; the cached home page is out of date; missing files return the home page with status 200
`broken`	7103	—	`robots.txt` answers 500; one JSON-LD block is not valid JSON; the organization has no `url`

All the code uses only the standard library of each language. The Python version needs Python 3.11 or newer and was written for 3.14; the Java version compiles with JDK 21 and runs on JDK

25. The folder layout, the commands for Windows, macOS and Linux, and the VS Code set-up are in the chapter's `README.md`:

```
Chapter10/
  sites/     laundry/  cdnblock/  stale/  broken/   (shared by both languages)
  python/    site_server.py  robots_rules.py  auditor.py  robots_quirk.py
  java/      SiteServer.java  RobotsRules.java  MiniJson.java  Auditor.java
```

10.9.2 The sample sites and their server

Each sample site is a folder of ordinary files plus one small configuration file, `_site.conf`. Here is the one for `stale`:

```
# HTML cached for a month at the edge; the cached home page is out of date.
# Missing files come back as the home page with status 200 (a "soft 404").
port = 7102
origin_port = 7106
soft404 = yes
header * = Cache-Control: public, max-age=604800, s-maxage=2592000
header /robots.txt = Cache-Control: public, max-age=604800
```

A line `header PATH = Name: value` adds a response header for one path, or for every path when `PATH` is `*`; a more specific line overrides the `*` line for the same header. A line `status PATH = CODE` forces a status code, which is how `broken` makes its `robots.txt` answer 500. `etag = yes` makes the server send an `ETag` computed from the file's content.

Two sub-folders simulate the layers of **Figure 10.1**. A file in `_edge/` is what the edge layer serves *instead* of the origin's file with the same name, on every request to the public port; that is how `cdnblock` gets its managed block. A file in `_cache/` is a stale copy that a cache still holds; it is served on the public port only when the URL has *no query string*, with the headers `X-Cache: HIT` and `Age: 5400`, exactly as a real cache keyed on the full URL would behave. Any request with a query string misses the cache and is answered from the origin with `X-Cache: MISS`. A site with an `origin_port` also answers on that second port with the plain origin files, never the `_edge` or `_cache` copies.

Listing 10.1 is the Python server. It reads each site's configuration, builds a request handler for each layer, and starts one `ThreadingHTTPServer` per port, each in a background thread.

Listing 10.1 `site_server.py`: serves the sample sites, simulating an edge layer (`_edge/`) and a stale cache (`_cache/`) in front of each origin.

```
"""Serve the sample sites of Chapter 10 on 127.0.0.1 (ports 7100-7109).

Each folder under sites/ is one site. Its _site.conf sets the port, extra
headers and status codes. Files in _edge/ are what an edge layer (a CDN)
serves instead of the origin file; files in _cache/ are stale cached copies,
served only when the URL has no query string.
usage: python site_server.py [sites_dir]
"""
import hashlib
import os
import sys
import threading
from http.server import BaseHTTPRequestHandler, ThreadingHTTPServer
from urllib.parse import urlsplit

TYPES = {".html": "text/html; charset=utf-8",
         ".txt": "text/plain; charset=utf-8",
         ".xml": "application/xml"}
CACHED_AGE = "5400"            # seconds a cached copy has sat at the edge


def read_conf(site_dir):
    conf = {"port": None, "origin_port": None, "etag": False,
            "soft404": False, "headers": [], "status": {}}
    with open(os.path.join(site_dir, "_site.conf"), encoding="utf-8") as f:
        for raw in f:
            line = raw.strip()
            if not line or line.startswith("#"):
                continue
            key, value = [s.strip() for s in line.split("=", 1)]
            if key.startswith("header "):
                name, val = value.split(":", 1)
                conf["headers"].append((key[7:].strip(), name, val.strip()))
            elif key.startswith("status "):
                conf["status"][key[7:].strip()] = int(value)
            elif key in ("port", "origin_port"):
                conf[key] = int(value)
            else:
                conf[key] = (value == "yes")
    return conf


def make_handler(site_dir, conf, layer):
    two_layers = conf["origin_port"] is not None
```

Listing 10.1 (Continued) *site_server.py: serves the sample sites, simulating an edge layer (_edge/) and a stale cache (_cache/) in front of each origin.*

```
    class Handler(BaseHTTPRequestHandler):
        def log_message(self, *args):          # keep the terminal quiet
            pass

        def do_GET(self):
            parts = urlsplit(self.path)
            path = parts.path
            name = "index.html" if path == "/" else path.lstrip("/")
            if name.startswith("_") or ".." in name:
                return self.reply(404, "text/plain", b"Not Found", path, {})
            if path in conf["status"]:
                code = conf["status"][path]
                return self.reply(code, "text/plain", b"Server Error", path, {})
            extra, file = {}, os.path.join(site_dir, name)
            edge = os.path.join(site_dir, "_edge", name)
            cached = os.path.join(site_dir, "_cache", name)
            if layer == "edge" and os.path.isfile(edge):
                file = edge                    # the edge rewrites this file
            elif layer == "edge" and not parts.query and os.path.isfile(cached):
                file = cached                  # a stale copy in the cache
                extra = {"X-Cache": "HIT", "Age": CACHED_AGE}
            elif layer == "edge" and two_layers:
                extra = {"X-Cache": "MISS"}
            if not os.path.isfile(file):
                if not conf["soft404"]:
                    return self.reply(404, "text/plain", b"Not Found", path, {})
                file = os.path.join(site_dir, "index.html")   # "soft 404"
            with open(file, "rb") as f:
                body = f.read()
            ctype = TYPES.get(os.path.splitext(file)[1], "text/plain")
            self.reply(200, ctype, body, path, extra)

        def do_HEAD(self):                     # same headers, no body
            self.do_GET()

        def reply(self, code, ctype, body, path, extra):
            headers = {"Content-Type": ctype}
            for pattern, hname, hval in conf["headers"]:
                if pattern == "*" or pattern == path:
                    headers[hname] = hval      # a path rule overrides "*"
            if conf["etag"] and code == 200:
                digest = hashlib.sha256(body).hexdigest()[:16]
                headers["ETag"] = '"' + digest + '"'
            headers.update(extra)
            headers["Content-Length"] = str(len(body))
            self.send_response(code)
```

Listing 10.1 (Continued) *site_server.py: serves the sample sites, simulating an edge layer (_edge/) and a stale cache (_cache/) in front of each origin.*

```
            for hname, hval in headers.items():
                self.send_header(hname, hval)
            self.end_headers()
            if self.command != "HEAD":
                self.wfile.write(body)

    return Handler


def start_all(sites_dir):
    """Start every site (and its origin, if any); return [(label, server)]."""
    started = []
    for site in sorted(os.listdir(sites_dir)):
        site_dir = os.path.join(sites_dir, site)
        if not os.path.isfile(os.path.join(site_dir, "_site.conf")):
            continue
        conf = read_conf(site_dir)
        layers = [("edge", conf["port"])]
        if conf["origin_port"] is not None:
            layers.append(("origin", conf["origin_port"]))
        for layer, port in layers:
            handler = make_handler(site_dir, conf, layer)
            server = ThreadingHTTPServer(("127.0.0.1", port), handler)
            threading.Thread(target=server.serve_forever, daemon=True).start()
            started.append((site + " " + layer, server))
    return started


def main():
    sites_dir = sys.argv[1] if len(sys.argv) > 1 else "../sites"
    servers = start_all(sites_dir)
    for label, server in servers:
        print("%-16s http://127.0.0.1:%d/" % (label, server.server_port))
    print("Serving; press Ctrl+C to stop.")
    try:
        threading.Event().wait()
    except KeyboardInterrupt:
        for _, server in servers:
            server.shutdown()


if __name__ == "__main__":
    main()
```

The heart of the server is the decision in `do_GET`, and it is worth reading in the order the code tests it. A path that starts with an underscore or contains `..` is refused, so a visitor can never read `_site.conf` or escape the site's folder. A forced status comes next. Then, on the public port only, an `_edge` file wins over everything; a `_cache` file wins if the URL has no query string; and a site with two layers labels every other answer as a cache `MISS`. Finally, a missing file is either a plain 404 or, when `soft404 = yes`, the home page with status 200, the behavior of many misconfigured content management systems and a trap for anyone who checks only the status code.

Listing 10.2 is the same server in Java, built on the JDK's `com.sun.net.httpserver.HttpServer`. The logic is line-for-line the same; the only additions are an explicit `Conf` class for the settings and a thread factory that marks the server threads as *daemon* threads, so that the Java Virtual Machine can exit as soon as `main` finishes.

Listing 10.2 `SiteServer.java`: the same sample-site server on the JDK's built-in HTTP server.

```
import com.sun.net.httpserver.HttpExchange;
import com.sun.net.httpserver.HttpServer;
import java.io.IOException;
import java.io.OutputStream;
import java.net.InetSocketAddress;
import java.nio.charset.StandardCharsets;
import java.nio.file.Files;
import java.nio.file.Path;
import java.security.MessageDigest;
import java.util.*;
import java.util.concurrent.Executors;

/** Serves the sample sites of Chapter 10 on 127.0.0.1 (ports 7100-7109).
 *  Files in _edge/ are what an edge layer serves instead of the origin file;
 *  files in _cache/ are stale copies, served only without a query string.
 *  usage: java SiteServer [sites_dir] */
public class SiteServer {
    static final String CACHED_AGE = "5400";   // seconds at the edge

    /** The settings read from one site's _site.conf. */
    static class Conf {
        Integer port, originPort;
        boolean etag, soft404;
        List<String[]> headers = new ArrayList<>();   // {pattern, name, value}
        Map<String, Integer> status = new HashMap<>();
    }

    static Conf readConf(Path siteDir) throws IOException {
        Conf c = new Conf();
        for (String raw : Files.readAllLines(siteDir.resolve("_site.conf"))) {
            String line = raw.strip();
            if (line.isEmpty() || line.startsWith("#")) continue;
            int eq = line.indexOf('=');
            String key = line.substring(0, eq).strip();
            String value = line.substring(eq + 1).strip();
            if (key.startsWith("header ")) {
                int colon = value.indexOf(':');
                c.headers.add(new String[] {key.substring(7).strip(),
                        value.substring(0, colon),
                        value.substring(colon + 1).strip()});
            } else if (key.startsWith("status ")) {
                c.status.put(key.substring(7).strip(), Integer.parseInt(value));
            } else if (key.equals("port")) {
                c.port = Integer.parseInt(value);
            } else if (key.equals("origin_port")) {
                c.originPort = Integer.parseInt(value);
```

Listing 10.2 (Continued) *SiteServer.java: the same sample-site server on the JDK's built-in HTTP server.*

```
            } else if (key.equals("etag")) {
                c.etag = value.equals("yes");
            } else if (key.equals("soft404")) {
                c.soft404 = value.equals("yes");
            }
        }
        return c;
    }

    static void handle(HttpExchange ex, Path dir, Conf conf, String layer)
            throws IOException {
        String path = ex.getRequestURI().getPath();
        String query = ex.getRequestURI().getRawQuery();
        String name = path.equals("/") ? "index.html" : path.substring(1);
        if (name.startsWith("_") || name.contains("..")) {
            reply(ex, 404, "text/plain", bytes("Not Found"), path, conf, Map.of());
            return;
        }
        if (conf.status.containsKey(path)) {
            reply(ex, conf.status.get(path), "text/plain", bytes("Server Error"),
                  path, conf, Map.of());
            return;
        }
        Map<String, String> extra = new LinkedHashMap<>();
        Path file = dir.resolve(name);
        Path edge = dir.resolve("_edge").resolve(name);
        Path cached = dir.resolve("_cache").resolve(name);
        boolean atEdge = layer.equals("edge");
        if (atEdge && Files.isRegularFile(edge)) {
            file = edge;                                  // the edge rewrites it
        } else if (atEdge && query == null && Files.isRegularFile(cached)) {
            file = cached;                                // a stale cached copy
            extra.put("X-Cache", "HIT");
            extra.put("Age", CACHED_AGE);
        } else if (atEdge && conf.originPort != null) {
            extra.put("X-Cache", "MISS");
        }
        if (!Files.isRegularFile(file)) {
            if (!conf.soft404) {
                reply(ex, 404, "text/plain", bytes("Not Found"), path, conf,
                      Map.of());
                return;
            }
            file = dir.resolve("index.html");             // a "soft 404"
        }
        String f = file.toString();
```

Listing 10.2 (Continued) *SiteServer.java: the same sample-site server on the JDK's built-in HTTP server.*

```
        String type = f.endsWith(".html") ? "text/html; charset=utf-8"
                : f.endsWith(".xml") ? "application/xml"
                : "text/plain; charset=utf-8";
        reply(ex, 200, type, Files.readAllBytes(file), path, conf, extra);
    }

    static void reply(HttpExchange ex, int code, String type, byte[] body,
                      String path, Conf conf, Map<String, String> extra)
            throws IOException {
        Map<String, String> h = new LinkedHashMap<>();
        h.put("Content-Type", type);
        for (String[] rule : conf.headers) {
            if (rule[0].equals("*") || rule[0].equals(path)) {
                h.put(rule[1], rule[2]);              // a path rule overrides *
            }
        }
        if (conf.etag && code == 200) {
            h.put("ETag", "\"" + sha256(body).substring(0, 16) + "\"");
        }
        h.putAll(extra);
        h.forEach((k, v) -> ex.getResponseHeaders().set(k, v));
        if (ex.getRequestMethod().equals("HEAD")) {   // same headers, no body
            ex.sendResponseHeaders(code, -1);
            return;
        }
        ex.sendResponseHeaders(code, body.length);
        try (OutputStream out = ex.getResponseBody()) {
            out.write(body);
        }
    }

    static byte[] bytes(String s) {
        return s.getBytes(StandardCharsets.UTF_8);
    }

    static String sha256(byte[] data) {
        try {
            byte[] d = MessageDigest.getInstance("SHA-256").digest(data);
            return HexFormat.of().formatHex(d);
        } catch (Exception e) {
            throw new IllegalStateException(e);
        }
    }
```

Listing 10.2 (Continued) *SiteServer.java: the same sample-site server on the JDK's built-in HTTP server.*

```
    /** Starts every site (and its origin, if any); returns label -> server. */
    static Map<String, HttpServer> startAll(Path sitesDir) throws IOException {
        Map<String, HttpServer> started = new LinkedHashMap<>();
        List<Path> sites = new ArrayList<>();
        try (var list = Files.list(sitesDir)) {
            list.filter(p -> Files.isRegularFile(p.resolve("_site.conf")))
                .sorted().forEach(sites::add);
        }
        for (Path dir : sites) {
            Conf conf = readConf(dir);
            Map<String, Integer> layers = new LinkedHashMap<>();
            layers.put("edge", conf.port);
            if (conf.originPort != null) layers.put("origin", conf.originPort);
            for (var layer : layers.entrySet()) {
                HttpServer server = HttpServer.create(
                        new InetSocketAddress("127.0.0.1", layer.getValue()), 0);
                server.createContext("/", ex -> {
                    try {
                        handle(ex, dir, conf, layer.getKey());
                    } finally {
                        ex.close();
                    }
                });
                server.setExecutor(Executors.newCachedThreadPool(r -> {
                    Thread t = new Thread(r);
                    t.setDaemon(true);                // never keeps the JVM alive
                    return t;
                }));
                server.start();
                started.put(dir.getFileName() + " " + layer.getKey(), server);
            }
        }
        return started;
    }

    public static void main(String[] args) throws Exception {
        Path sitesDir = Path.of(args.length > 0 ? args[0] : "../sites");
        Map<String, HttpServer> servers = startAll(sitesDir);
        servers.forEach((label, s) -> System.out.printf(
                "%-16s http://127.0.0.1:%d/%n", label, s.getAddress().getPort()));
        System.out.println("Serving; press Ctrl+C to stop.");
        Thread.currentThread().join();                // until Ctrl+C
    }
}
```

Start the Python server in one terminal with `python site_server.py ../sites` (on Windows, `py site_server.py ..\sites`). It prints the address of every layer it started:

```
broken edge      http://127.0.0.1:7103/
cdnblock edge    http://127.0.0.1:7101/
cdnblock origin  http://127.0.0.1:7105/
laundry edge     http://127.0.0.1:7100/
stale edge       http://127.0.0.1:7102/
stale origin     http://127.0.0.1:7106/
Serving; press Ctrl+C to stop.
```

In a second terminal, `curl -i` shows a response with its headers. First the public `robots.txt` of `cdnblock`, then the same file at its origin:

```
$ curl -i http://127.0.0.1:7101/robots.txt
HTTP/1.0 200 OK
Server: BaseHTTP/0.6 Python/3.11.15
Date: Thu, 24 Sep 2026 04:56:58 GMT
Content-Type: text/plain; charset=utf-8
Cache-Control: public, max-age=14400
ETag: "a75bf8f54f3e4133"
Content-Length: 294

# BEGIN block managed by the edge layer
User-agent: GPTBot
Disallow: /

User-agent: ClaudeBot
Disallow: /

User-agent: Google-Extended
Disallow: /
# END block managed by the edge layer

# Origin copy, written by the site owner
User-agent: *
Allow: /

Sitemap: http://127.0.0.1:7101/sitemap.xml

$ curl -i http://127.0.0.1:7105/robots.txt
HTTP/1.0 200 OK
Server: BaseHTTP/0.6 Python/3.11.15
Date: Thu, 24 Sep 2026 04:56:58 GMT
Content-Type: text/plain; charset=utf-8
Cache-Control: public, max-age=14400
ETag: "692d92fde37dbec1"
Content-Length: 108

# Origin copy, written by the site owner
User-agent: *
Allow: /

Sitemap: http://127.0.0.1:7101/sitemap.xml
```

This is Section 10.5 in miniature: the site owner's file is innocent, and the file the world receives blocks three crawlers. The `stale` site shows the other fault. `curl -I` asks for the headers only; without a query string the cache answers, and with one the origin does:

```
$ curl -I http://127.0.0.1:7102/
HTTP/1.0 200 OK
Server: BaseHTTP/0.6 Python/3.11.15
Date: Thu, 24 Sep 2026 04:56:58 GMT
Content-Type: text/html; charset=utf-8
Cache-Control: public, max-age=604800, s-maxage=2592000
X-Cache: HIT
Age: 5400
Content-Length: 511

$ curl -I "http://127.0.0.1:7102/?nocache=1"
HTTP/1.0 200 OK
Server: BaseHTTP/0.6 Python/3.11.15
Date: Thu, 24 Sep 2026 04:56:58 GMT
Content-Type: text/html; charset=utf-8
Cache-Control: public, max-age=604800, s-maxage=2592000
X-Cache: MISS
Content-Length: 517
```

The two lengths, 511 and 517 bytes, already tell us that the two answers are different pages. Press Ctrl+C in the first terminal to stop the server; the auditor below does not need it, because it can start the sites itself.

10.9.3 A robots.txt reader that follows RFC 9309

Chapter 2's `Robots` class implemented group selection and the longest-match rule without wildcards. This chapter needs the rest of RFC 9309: groups for the same crawler merged, the `*` group as a pure fallback, and the `*` and `$` wildcards. **Listing 10.3** does it in about sixty lines.

Listing 10.3 `robots_rules.py`: a reader that implements Equation (10.1) (merging, fallback to `*`) and Equation (10.2) (longest match, `Allow` on a tie), with the `*` and `$` wildcards.

```
"""A small robots.txt reader that follows RFC 9309.

Groups that name the same crawler are merged; a crawler with no group of
its own obeys the "*" group; the longest matching path wins, and on a tie
"allow" wins. Python's urllib.robotparser uses the first match instead.
"""
import re


class Robots:
    def __init__(self, text):
        self.groups = []           # [(set of agents, [(allow, path), ...])]
        self.sitemaps = []
        self.signals = []          # Content-Signal lines (not in RFC 9309)
        agents, rules, last_was_agent = None, None, False
        for raw in text.splitlines():
            line = raw.split("#", 1)[0].strip()
            if ":" not in line:
                continue
            key, value = line.split(":", 1)
            key, value = key.strip().lower(), value.strip()
            if key == "user-agent":
                if not last_was_agent:      # a new group starts here
                    agents, rules = set(), []
                    self.groups.append((agents, rules))
                agents.add(value.lower())
                last_was_agent = True
                continue
            last_was_agent = False
            if key in ("allow", "disallow") and rules is not None and value:
                rules.append((key == "allow", value))
            elif key == "sitemap":
                self.sitemaps.append(value)
            elif key == "content-signal":
                self.signals.append(value)

    def rules_for(self, agent):
        agent = agent.lower()
        mine = [r for agents, rules in self.groups if agent in agents
                for r in rules]
        if mine:
            return mine
        return [r for agents, rules in self.groups if "*" in agents
                for r in rules]
```

Listing 10.3 (Continued) *robots_rules.py: a reader that implements Equation (10.1) (merging, fallback to *) and Equation (10.2) (longest match, Allow on a tie), with the * and $ wildcards.*

```
    def allowed(self, agent, path="/"):
        best_len, best_allow = -1, True     # no matching rule: allowed
        for allow, pattern in self.rules_for(agent):
            if matches(pattern, path):
                n = len(pattern.encode("utf-8"))
                if n > best_len or (n == best_len and allow):
                    best_len, best_allow = n, allow
        return best_allow


def matches(pattern, path):
    """'*' matches any run of characters; a final '$' anchors the end."""
    anchored = pattern.endswith("$")
    body = pattern[:-1] if anchored else pattern
    regex = ".*".join(re.escape(part) for part in body.split("*"))
    return re.match(regex + ("$" if anchored else ""), path) is not None
```

Three details deserve comment. The constructor keeps a flag, `last_was_agent`, because a `User-agent` line starts a new group only when the previous significant line was *not* also a `User-agent` line; that is how five consecutive `User-agent` lines become one group, as in Section 10.4.2. The method `rules_for` is Equation (10.1) almost word for word: collect the rules of every group that names the crawler, and only if there are none, collect the rules of the `*` groups. And `allowed` is Equation (10.2): it walks the rules, keeps the best pair (length, is-allow) seen so far, and replaces it when a longer pattern matches or an equally long `Allow` does. An empty `Disallow:` line, which in `robots.txt` means "nothing is disallowed", is simply not added as a rule.

The function `matches` handles the wildcards. `Disallow: /*.pdf$` becomes the regular expression `/.*\.pdf$`, and `re.match` anchors it at the start of the path, because every pattern in `robots.txt` matches from the beginning. **Listing 10.4** is the Java version, with records for rules and groups.

Listing 10.4 `RobotsRules.java`: the same RFC 9309 reader in Java.

```java
import java.nio.charset.StandardCharsets;
import java.util.*;
import java.util.regex.Pattern;

/** A small robots.txt reader that follows RFC 9309. Groups naming the same
 *  crawler are merged; a crawler with no group of its own obeys "*"; the
 *  longest matching path wins, and on a tie "allow" wins. */
public class RobotsRules {
    record Rule(boolean allow, String path) {}
    record Group(Set<String> agents, List<Rule> rules) {}

    final List<Group> groups = new ArrayList<>();
    final List<String> sitemaps = new ArrayList<>();
    final List<String> signals = new ArrayList<>();   // Content-Signal lines

    RobotsRules(String text) {
        Group current = null;
        boolean lastWasAgent = false;
        for (String raw : text.split("\r?\n")) {
            String line = raw.split("#", 2)[0].strip();
            int colon = line.indexOf(':');
            if (colon < 0) continue;
            String key = line.substring(0, colon).strip().toLowerCase();
            String value = line.substring(colon + 1).strip();
            if (key.equals("user-agent")) {
                if (!lastWasAgent) {                  // a new group starts
                    current = new Group(new HashSet<>(), new ArrayList<>());
                    groups.add(current);
                }
                current.agents().add(value.toLowerCase());
                lastWasAgent = true;
                continue;
            }
            lastWasAgent = false;
            if ((key.equals("allow") || key.equals("disallow"))
                    && current != null && !value.isEmpty()) {
                current.rules().add(new Rule(key.equals("allow"), value));
            } else if (key.equals("sitemap")) {
                sitemaps.add(value);
            } else if (key.equals("content-signal")) {
                signals.add(value);
            }
        }
    }
```

Listing 10.4 (Continued) *RobotsRules.java: the same RFC 9309 reader in Java.*

```
    List<Rule> rulesFor(String agent) {
        String a = agent.toLowerCase();
        List<Rule> mine = new ArrayList<>();
        for (Group g : groups) if (g.agents().contains(a)) mine.addAll(g.rules());
        if (!mine.isEmpty()) return mine;
        for (Group g : groups) if (g.agents().contains("*")) mine.addAll(g.rules());
        return mine;
    }

    boolean allowed(String agent, String path) {
        int bestLen = -1;
        boolean bestAllow = true;                       // no rule: allowed
        for (Rule r : rulesFor(agent)) {
            if (matches(r.path(), path)) {
                int n = r.path().getBytes(StandardCharsets.UTF_8).length;
                if (n > bestLen || (n == bestLen && r.allow())) {
                    bestLen = n;
                    bestAllow = r.allow();
                }
            }
        }
        return bestAllow;
    }

    boolean allowed(String agent) {
        return allowed(agent, "/");
    }

    /** '*' matches any run of characters; a final '$' anchors the end. */
    static boolean matches(String pattern, String path) {
        boolean anchored = pattern.endsWith("$");
        String body = anchored ? pattern.substring(0, pattern.length() - 1)
                               : pattern;
        StringBuilder regex = new StringBuilder();
        String[] parts = body.split("\\*", -1);
        for (int i = 0; i < parts.length; i++) {
            if (i > 0) regex.append(".*");
            regex.append(Pattern.quote(parts[i]));
        }
        if (!anchored) regex.append(".*");
        return Pattern.compile(regex.toString(), Pattern.DOTALL)
                      .matcher(path).matches();
    }
}
```

10.9.4 The auditor

Listing 10.5 is the auditor. It is the longest program of the chapter, but it is a sequence of small, independent checks, each a function that fetches what it needs and prints its findings through a `Report` object that remembers the worst status of each category.

Listing 10.5 `auditor.py`: checks `robots.txt`, `llms.txt`, cache headers and JSON-LD of each site and prints a report and a summary table.

```
"""Audit robots.txt, llms.txt, cache headers and JSON-LD of web sites.

usage: python auditor.py --local ../sites      (start the sample sites, audit)
       python auditor.py https://your-own-site.example ...
One request per resource, a timeout, and an honest User-Agent.
"""
import json
import os
import re
import sys
import urllib.error
import urllib.request

from robots_rules import Robots

BOTS = ["GPTBot", "OAI-SearchBot", "ClaudeBot", "PerplexityBot",
        "Google-Extended", "Bingbot"]
AGENT = "Chapter10-Auditor/1.0 (Profitina book lab)"
RANK = {"PASS": 0, "INFO": 1, "WARN": 2, "FAIL": 3}
LDJSON = re.compile(r'<script[^>]*type\s*=\s*["\']application/ld\+json["\']'
                    r'[^>]*>(.*?)</script>', re.I | re.S)
ENTITIES = {"Organization", "LocalBusiness", "Restaurant", "Store"}


def fetch(url):
    """Return (status, headers with lower-case names, text); 0 if no reply."""
    request = urllib.request.Request(url, headers={"User-Agent": AGENT})
    try:
        with urllib.request.urlopen(request, timeout=5) as r:
            status, headers, body = r.status, r.headers, r.read()
    except urllib.error.HTTPError as e:
        status, headers, body = e.code, e.headers, e.read()
    except (urllib.error.URLError, OSError):
        return 0, {}, ""
    lower = {k.lower(): v for k, v in headers.items()}
    return status, lower, body.decode("utf-8", "replace")


class Report:
    def __init__(self):
        self.worst = {}
```

Listing 10.5 (Continued) *auditor.py: checks robots.txt, llms.txt, cache headers and JSON-LD of each site and prints a report and a summary table.*

```
    def line(self, category, status, text, *details):
        print("[%s] %s" % (status, text))
        for d in details:
            print("       " + d)
        old = self.worst.get(category)
        if old is None or RANK[status] > RANK[old]:
            self.worst[category] = status


def media_type(headers):
    return headers.get("content-type", "").split(";")[0].strip()


def check_robots(rep, base, origin):
    status, _, text = fetch(base + "/robots.txt")
    if status >= 500 or status == 0:
        rep.line("robots", "FAIL", "robots.txt: HTTP %d; crawlers must assume "
                 '"disallow all"' % status)
        return
    if status >= 400:
        rep.line("robots", "WARN", "robots.txt: HTTP %d; crawlers assume "
                 '"allow all"' % status)
        return
    robots = Robots(text)
    blocked = [b for b in BOTS if not robots.allowed(b, "/")]
    extra = ["sitemap: " + s for s in robots.sitemaps] or ["no Sitemap line"]
    extra += ["content signal: " + s for s in robots.signals]
    if blocked:
        rep.line("robots", "FAIL", "robots.txt: HTTP 200; blocked for "
                 + ", ".join(blocked), *extra)
    else:
        rep.line("robots", "PASS", "robots.txt: HTTP 200; all %d crawlers "
                 "may fetch /" % len(BOTS), *extra)
    _, _, bypass = fetch(base + "/robots.txt?audit-bypass=1")
    if bypass != text:
        rep.line("robots", "WARN", "robots.txt: served copy differs from a "
                 "cache-bypass copy", "a stale copy is cached; purge the cache")
        return
    if origin:
        _, _, at_origin = fetch(origin + "/robots.txt")
        if at_origin != text:
            o = Robots(at_origin)
            diff = [b for b in BOTS if o.allowed(b) != robots.allowed(b)]
            rep.line("robots", "WARN", "robots.txt: served copy differs from "
                     "the origin copy", "origin allows " + ", ".join(diff),
                     "something in front of the origin rewrites the file")
```

Listing 10.5 (Continued) *auditor.py: checks robots.txt, llms.txt, cache headers and JSON-LD of each site and prints a report and a summary table.*

```
            return
    agree = "served, cache-bypass and origin" if origin else \
        "served and cache-bypass"
    rep.line("robots", "PASS", "robots.txt: %s copies agree" % agree)


def check_llms(rep, base):
    status, headers, text = fetch(base + "/llms.txt")
    kind = media_type(headers)
    if status == 404:
        rep.line("llms", "INFO", "llms.txt: absent (HTTP 404); optional")
        return
    if status != 200:
        rep.line("llms", "WARN", "llms.txt: HTTP %d" % status)
        return
    if kind not in ("text/plain", "text/markdown"):
        rep.line("llms", "WARN", "llms.txt: HTTP 200 but %s" % kind,
                 'probably a "soft 404" page, not an llms.txt file')
        return
    lines = [s for s in text.splitlines() if s.strip()]
    if not lines or not lines[0].startswith("# "):
        rep.line("llms", "WARN", "llms.txt: first line is not an H1 title")
        return
    summary = "summary" if any(s.startswith(">") for s in lines) \
        else "no summary"
    links = len(re.findall(r"\[[^\]]+\]\([^)]+\)", text))
    rep.line("llms", "PASS", 'llms.txt: HTTP 200 %s; H1 "%s", %s, %s'
             % (kind, lines[0][2:].strip(), summary, plural(links, "link")))


def plural(n, word):
    return "%d %s%s" % (n, word, "" if n == 1 else "s")


def human(seconds):
    if seconds < 3600:
        return "%d min" % (seconds // 60)
    if seconds < 86400:
        return "%d h" % (seconds // 3600)
    return "%d days" % (seconds // 86400)
```

Listing 10.5 (Continued) *auditor.py: checks robots.txt, llms.txt, cache headers and JSON-LD of each site and prints a report and a summary table.*

```
def check_cache(rep, path, headers):
    cc = headers.get("cache-control", "")
    directives = {}
    for part in cc.split(","):
        name, _, value = part.strip().lower().partition("=")
        if name:
            directives[name] = value
    validator = "ETag" if "etag" in headers else \
        "Last-Modified" if "last-modified" in headers else "no validator"
    layers = ["%s %s" % (n, headers[n.lower()]) for n in
              ("X-Cache", "CF-Cache-Status", "X-LiteSpeed-Cache", "Age")
              if n.lower() in headers]
    extra = ["layer headers: " + ", ".join(layers)] if layers else []
    if not cc:
        rep.line("cache", "INFO", "cache %s: no Cache-Control; %s" %
                 (path, validator), "caches may guess a lifetime", *extra)
        return
    if {"no-store", "no-cache", "private"} & directives.keys():
        rep.line("cache", "PASS", "cache %s: not kept by shared caches; %s"
                 % (path, validator), *extra)
        return
    ttl = int(directives.get("s-maxage") or directives.get("max-age") or 0)
    text = "cache %s: shared caches keep it %s; %s" % (path, human(ttl),
                                                       validator)
    if ttl > 86400 and path == "/robots.txt":
        rep.line("cache", "WARN", text, "crawlers re-read robots.txt within "
                 "24 h; a long cache delays every fix", *extra)
    elif ttl > 86400:
        rep.line("cache", "WARN", text, "HTML cached this long needs a purge "
                 "of every layer after each edit", *extra)
    else:
        rep.line("cache", "PASS", text, *extra)


def walk(node, found):
    """Collect every schema.org node (a dict with @type) in document order."""
    if isinstance(node, dict):
        if "@type" in node:
            found.append(node)
        for value in node.values():
            walk(value, found)
    elif isinstance(node, list):
        for value in node:
            walk(value, found)
```

Listing 10.5 (Continued) *auditor.py: checks robots.txt, llms.txt, cache headers and JSON-LD of each site and prints a report and a summary table.*

```
def types_of(node):
    t = node["@type"]
    return t if isinstance(t, list) else [t]


def date_modified(html):
    for block in LDJSON.findall(html):
        try:
            found = []
            walk(json.loads(block), found)
        except ValueError:
            continue
        for node in found:
            if "dateModified" in node:
                return node["dateModified"]
    return None


def check_jsonld(rep, html, fresh_html):
    blocks = LDJSON.findall(html)
    if not blocks:
        rep.line("jsonld", "WARN", "JSON-LD: none found on the home page")
        return
    nodes, bad = [], 0
    for block in blocks:
        try:
            walk(json.loads(block), nodes)
        except ValueError:
            bad += 1
    types = []
    for node in nodes:
        for t in types_of(node):
            if t not in types:
                types.append(t)
    details = ["types: " + (", ".join(types) or "none")]
    for node in nodes:
        if "FAQPage" in types_of(node):
            n = len(node.get("mainEntity", []))
            details.append("FAQPage with %d questions" % n)
    served = date_modified(html)
    details.append("dateModified " + (served or "absent"))
    if bad:
        rep.line("jsonld", "FAIL", "JSON-LD: %d of %d blocks are not valid "
                 "JSON; engines skip them" % (bad, len(blocks)), *details)
    else:
        rep.line("jsonld", "PASS", "JSON-LD: %s, all valid JSON"
```

Listing 10.5 (Continued) *auditor.py: checks robots.txt, llms.txt, cache headers and JSON-LD of each site and prints a report and a summary table.*

```
                % plural(len(blocks), "block"), *details)
    for node in nodes:
        kinds = [t for t in types_of(node) if t in ENTITIES]
        for key in ("name", "url"):
            if kinds and key not in node:
                rep.line("jsonld", "WARN", 'JSON-LD: %s has no "%s"'
                         % (kinds[0], key))
    fresh = date_modified(fresh_html)
    if served and fresh and served != fresh:
        rep.line("jsonld", "WARN", "JSON-LD: served page says dateModified "
                 "%s, origin says %s" % (served, fresh),
                 "a stale copy is served; purge the cache layers")


def audit(name, base, origin=None):
    print("== %s  %s" % (name, base))
    rep = Report()
    check_robots(rep, base, origin)
    check_llms(rep, base)
    _, home_headers, html = fetch(base + "/")
    _, robots_headers, _ = fetch(base + "/robots.txt")
    check_cache(rep, "/", home_headers)
    check_cache(rep, "/robots.txt", robots_headers)
    _, _, fresh = fetch((origin or base) + "/?audit-bypass=1")
    check_jsonld(rep, html, fresh)
    print()
    return rep.worst


def main():
    args = sys.argv[1:]
    targets, servers = [], []
    if args[:1] == ["--local"]:
        from site_server import read_conf, start_all
        sites_dir = args[1] if len(args) > 1 else "../sites"
        servers = start_all(sites_dir)
        confs = [(read_conf(os.path.join(sites_dir, site)), site)
                 for site in os.listdir(sites_dir)
                 if os.path.isfile(os.path.join(sites_dir, site, "_site.conf"))]
        for conf, site in sorted(confs, key=lambda c: c[0]["port"]):
            base = "http://127.0.0.1:%d" % conf["port"]
            origin = conf["origin_port"] and \
                "http://127.0.0.1:%d" % conf["origin_port"]
            targets.append((site, base, origin))
    else:
        for url in args:
```

Listing 10.5 (Continued) *auditor.py: checks robots.txt, llms.txt, cache headers and JSON-LD of each site and prints a report and a summary table.*

```
            url = url.rstrip("/")
            targets.append((url.split("//")[-1], url, None))
    if not targets:
        print(__doc__.strip())
        return
    results = [(name, audit(name, base, origin))
               for name, base, origin in targets]
    for _, server in servers:
        server.shutdown()
    print("%-10s %-7s %-7s %-7s %s" % ("site", "robots", "llms", "cache",
                                       "json-ld"))
    for name, worst in results:
        print("%-10s %-7s %-7s %-7s %s" % (name, *[worst.get(c, "-") for c in
              ("robots", "llms", "cache", "jsonld")]))
    print("\nNote on llms.txt: cheap hygiene, not a proven citation signal. In")
    print("May 2026, 97% of llms.txt files got zero requests (Ahrefs log study")
    print("of 137,210 domains).")


if __name__ == "__main__":
    main()
```

Reading the checks in the order `audit` calls them:

- **`fetch`** returns the status, the headers with lower-case names, and the body. An HTTP error such as 404 or 500 is not an exception for the auditor but an answer to report, so it is caught and returned like any other; only a site that does not answer at all becomes status 0.
- **`check_robots`** first applies **Table 10.2**: a server error or no answer is `FAIL`, because crawlers must then assume that everything is disallowed; a 4xx is only a `WARN`, because crawlers may then read everything. For a 2xx it parses the file with `Robots` and asks, for each of six crawlers, whether it may fetch `/`. Then come the two comparisons of Section 10.5.5: the served copy against a copy fetched with `?audit-bypass=1`, and, when an origin address is known, the served copy against the origin's. The first difference means a stale cache; the second means a layer that rewrites the file, and the auditor names the crawlers for which the origin's rules and the served rules disagree.
- **`check_llms`** separates four situations: no file (404, only `INFO`, since the file is optional), an error, a "file" that is really an HTML page served with status 200 (the soft 404 of the `stale` site), and a real file, which must start with an H1 line. For a real file it reports the title, whether there is a summary block quote, and the number of Markdown links.
- **`check_cache`** parses `Cache-Control` into directives and computes the shared-cache lifetime of Equation (10.4): `s-maxage` if present, else `max-age`. No header at all is `INFO`, since caches will guess; `no-store`, `no-cache` or `private` is a `PASS`; a lifetime longer than a day is a `WARN`, with a different explanation for HTML and for `robots.txt`. It also lists any cache-layer headers it recognizes (`X-Cache`, `CF-Cache-Status`, `X-LiteSpeed-Cache`, `Age`), which is how you can tell a `HIT` from a `MISS` on a real site.
- **`check_jsonld`** extracts every `<script type="application/ld+json">` block with a regular expression, parses each one with a strict JSON parser, and walks the result to collect every node that has an `@type`, including nodes nested inside others, such as the `PostalAddress` inside a `LocalBusiness`. It reports invalid blocks, the types found, the number of questions of any `FAQPage`, and the `dateModified`; warns about an organization-like node without `name` or `url`; and compares the served `dateModified` with the origin's, the check of Section 10.8.3.

The `main` function has two modes. With `--local` it reads the sample sites' configuration files, starts all their servers inside the same process, audits them in order of port, and shuts the servers

down before printing the summary. With one or more URLs it audits those sites instead, which is how you point it at your own site.

The Java version needs one thing that Python has built in: a JSON parser. **Listing 10.6** is a small, strict one, enough to decide whether a block is valid JSON and to walk its objects and arrays. It deliberately rejects a trailing comma, as every conforming parser must.

Listing 10.6 `MiniJson.java`: a tiny strict JSON reader for the Java auditor; anything that is not valid JSON, such as a trailing comma, throws an exception.

```
import java.util.*;

/** A tiny, strict JSON reader: objects become LinkedHashMap, arrays become
 *  ArrayList, numbers Double, plus String, Boolean and null. Anything that
 *  is not valid JSON (a trailing comma, for example) throws. */
public class MiniJson {
    private final String s;
    private int i;

    private MiniJson(String s) {
        this.s = s;
    }

    public static Object parse(String text) {
        MiniJson p = new MiniJson(text);
        Object value = p.value();
        p.space();
        if (p.i != p.s.length()) throw p.error("extra text");
        return value;
    }

    private IllegalArgumentException error(String what) {
        return new IllegalArgumentException(what + " at offset " + i);
    }

    private void space() {
        while (i < s.length() && " \t\r\n".indexOf(s.charAt(i)) >= 0) i++;
    }

    private char peek() {
        space();
        if (i >= s.length()) throw error("unexpected end");
        return s.charAt(i);
    }

    private void expect(char c) {
        if (peek() != c) throw error("expected '" + c + "'");
        i++;
    }
```

Listing 10.6 (Continued) *MiniJson.java: a tiny strict JSON reader for the Java auditor; anything that is not valid JSON, such as a trailing comma, throws an exception.*

```
    private Object value() {
        char c = peek();
        if (c == '{') return object();
        if (c == '[') return array();
        if (c == '"') return string();
        for (String word : new String[] {"true", "false", "null"}) {
            if (s.startsWith(word, i)) {
                i += word.length();
                return word.equals("null") ? null : Boolean.valueOf(word);
            }
        }
        return number();
    }

    private Map<String, Object> object() {
        Map<String, Object> map = new LinkedHashMap<>();
        expect('{');
        if (peek() == '}') {
            i++;
            return map;
        }
        while (true) {
            String key = string();
            expect(':');
            map.put(key, value());
            if (peek() == ',') {
                i++;
                continue;                    // the next key must follow
            }
            expect('}');
            return map;
        }
    }

    private List<Object> array() {
        List<Object> list = new ArrayList<>();
        expect('[');
        if (peek() == ']') {
            i++;
            return list;
        }
        while (true) {
            list.add(value());
            if (peek() == ',') {
                i++;
                continue;
```

Listing 10.6 (Continued) *MiniJson.java: a tiny strict JSON reader for the Java auditor; anything that is not valid JSON, such as a trailing comma, throws an exception.*

```
            }
            expect(']');
            return list;
        }
    }

    private String string() {
        expect('"');
        StringBuilder b = new StringBuilder();
        while (true) {
            if (i >= s.length()) throw error("unterminated string");
            char c = s.charAt(i++);
            if (c == '"') return b.toString();
            if (c < 0x20) throw error("control character in string");
            if (c != '\\') {
                b.append(c);
                continue;
            }
            if (i >= s.length()) throw error("unterminated string");
            char e = s.charAt(i++);
            switch (e) {
                case '"', '\\', '/' -> b.append(e);
                case 'b' -> b.append('\b');
                case 'f' -> b.append('\f');
                case 'n' -> b.append('\n');
                case 'r' -> b.append('\r');
                case 't' -> b.append('\t');
                case 'u' -> {
                    if (i + 4 > s.length()) throw error("bad \\u escape");
                    b.append((char) Integer.parseInt(s.substring(i, i + 4), 16));
                    i += 4;
                }
                default -> throw error("bad escape");
            }
        }
    }
```

Listing 10.6 (Continued) *MiniJson.java: a tiny strict JSON reader for the Java auditor; anything that is not valid JSON, such as a trailing comma, throws an exception.*

```
    private Double number() {
        int start = i;
        while (i < s.length() && "+-0123456789.eE".indexOf(s.charAt(i)) >= 0) i++;
        try {
            return Double.valueOf(s.substring(start, i));
        } catch (NumberFormatException e) {
            i = start;
            throw error("not a JSON value");
        }
    }
}
```

Listing 10.7 is the Java auditor. It uses `java.net.http.HttpClient` for the requests, pinned to HTTP/1.1 so that it talks to the sample servers the same way `urllib` does, and it prints exactly the same report as the Python version, character for character.

Listing 10.7 `Auditor.java`: the same auditor in Java; it prints exactly the same report as **Listing 10.5**.

```java
import com.sun.net.httpserver.HttpServer;
import java.net.URI;
import java.net.http.HttpClient;
import java.net.http.HttpRequest;
import java.net.http.HttpResponse;
import java.nio.file.Files;
import java.nio.file.Path;
import java.time.Duration;
import java.util.*;
import java.util.regex.Matcher;
import java.util.regex.Pattern;

/** Audits robots.txt, llms.txt, cache headers and JSON-LD of web sites.
 *  usage: java Auditor --local ../sites   (start the sample sites, audit)
 *         java Auditor https://your-own-site.example ...
 *  One request per resource, a timeout, and an honest User-Agent. */
public class Auditor {
    static final List<String> BOTS = List.of("GPTBot", "OAI-SearchBot",
            "ClaudeBot", "PerplexityBot", "Google-Extended", "Bingbot");
    static final String AGENT = "Chapter10-Auditor/1.0 (Profitina book lab)";
    static final List<String> RANK = List.of("PASS", "INFO", "WARN", "FAIL");
    static final Pattern LDJSON = Pattern.compile(
            "<script[^>]*type\\s*=\\s*[\"']application/ld\\+json[\"']"
            + "[^>]*>(.*?)</script>", Pattern.CASE_INSENSITIVE | Pattern.DOTALL);
    static final Set<String> ENTITIES = Set.of("Organization", "LocalBusiness",
            "Restaurant", "Store");
    static final HttpClient HTTP = HttpClient.newBuilder()
            .version(HttpClient.Version.HTTP_1_1)
            .connectTimeout(Duration.ofSeconds(5)).build();

    record Page(int status, Map<String, String> headers, String text) {}

    /** Returns status, headers with lower-case names, and text; 0 if no reply. */
    static Page fetch(String url) {
        try {
            HttpRequest req = HttpRequest.newBuilder(URI.create(url))
                    .header("User-Agent", AGENT)
                    .timeout(Duration.ofSeconds(5)).build();
            HttpResponse<String> r = HTTP.send(req,
                    HttpResponse.BodyHandlers.ofString());
            Map<String, String> h = new HashMap<>();
            r.headers().map().forEach((k, v) -> h.put(k.toLowerCase(), v.get(0)));
            return new Page(r.statusCode(), h, r.body());
        } catch (Exception e) {
            return new Page(0, Map.of(), "");
        }
```

Listing 10.7 (Continued) *Auditor.java: the same auditor in Java; it prints exactly the same report as Listing 10.5.*

```
    }

    /** Prints report lines and remembers the worst status per category. */
    static class Report {
        final Map<String, String> worst = new HashMap<>();

        void line(String category, String status, String text, String... details) {
            System.out.println("[" + status + "] " + text);
            for (String d : details) System.out.println("       " + d);
            String old = worst.get(category);
            if (old == null || RANK.indexOf(status) > RANK.indexOf(old)) {
                worst.put(category, status);
            }
        }
    }

    static String mediaType(Map<String, String> headers) {
        return headers.getOrDefault("content-type", "").split(";")[0].strip();
    }

    static List<String> blockedBy(RobotsRules robots) {
        List<String> out = new ArrayList<>();
        for (String b : BOTS) if (!robots.allowed(b)) out.add(b);
        return out;
    }

    static void checkRobots(Report rep, String base, String origin) {
        Page p = fetch(base + "/robots.txt");
        if (p.status() >= 500 || p.status() == 0) {
            rep.line("robots", "FAIL", "robots.txt: HTTP " + p.status()
                    + "; crawlers must assume \"disallow all\"");
            return;
        }
        if (p.status() >= 400) {
            rep.line("robots", "WARN", "robots.txt: HTTP " + p.status()
                    + "; crawlers assume \"allow all\"");
            return;
        }
        RobotsRules robots = new RobotsRules(p.text());
        List<String> blocked = blockedBy(robots);
        List<String> extra = new ArrayList<>();
        for (String s : robots.sitemaps) extra.add("sitemap: " + s);
        if (extra.isEmpty()) extra.add("no Sitemap line");
        for (String s : robots.signals) extra.add("content signal: " + s);
        String[] ex = extra.toArray(new String[0]);
        if (!blocked.isEmpty()) {
```

Listing 10.7 (Continued) *Auditor.java: the same auditor in Java; it prints exactly the same report as Listing 10.5.*

```
            rep.line("robots", "FAIL", "robots.txt: HTTP 200; blocked for "
                    + String.join(", ", blocked), ex);
        } else {
            rep.line("robots", "PASS", "robots.txt: HTTP 200; all " + BOTS.size()
                    + " crawlers may fetch /", ex);
        }
        String bypass = fetch(base + "/robots.txt?audit-bypass=1").text();
        if (!bypass.equals(p.text())) {
            rep.line("robots", "WARN", "robots.txt: served copy differs from a "
                    + "cache-bypass copy",
                    "a stale copy is cached; purge the cache");
            return;
        }
        if (origin != null) {
            String atOrigin = fetch(origin + "/robots.txt").text();
            if (!atOrigin.equals(p.text())) {
                RobotsRules o = new RobotsRules(atOrigin);
                List<String> diff = new ArrayList<>();
                for (String b : BOTS) {
                    if (o.allowed(b) != robots.allowed(b)) diff.add(b);
                }
                rep.line("robots", "WARN", "robots.txt: served copy differs from "
                        + "the origin copy",
                        "origin allows " + String.join(", ", diff),
                        "something in front of the origin rewrites the file");
                return;
            }
        }
        String agree = origin != null ? "served, cache-bypass and origin"
                                      : "served and cache-bypass";
        rep.line("robots", "PASS", "robots.txt: " + agree + " copies agree");
    }

    static void checkLlms(Report rep, String base) {
        Page p = fetch(base + "/llms.txt");
        String kind = mediaType(p.headers());
        if (p.status() == 404) {
            rep.line("llms", "INFO", "llms.txt: absent (HTTP 404); optional");
            return;
        }
        if (p.status() != 200) {
            rep.line("llms", "WARN", "llms.txt: HTTP " + p.status());
            return;
        }
        if (!kind.equals("text/plain") && !kind.equals("text/markdown")) {
            rep.line("llms", "WARN", "llms.txt: HTTP 200 but " + kind,
```

Listing 10.7 (Continued) *Auditor.java: the same auditor in Java; it prints exactly the same report as Listing 10.5.*

```
                    "probably a \"soft 404\" page, not an llms.txt file");
            return;
        }
        List<String> lines = new ArrayList<>();
        for (String s : p.text().split("\r?\n")) if (!s.isBlank()) lines.add(s);
        if (lines.isEmpty() || !lines.get(0).startsWith("# ")) {
            rep.line("llms", "WARN", "llms.txt: first line is not an H1 title");
            return;
        }
        boolean hasSummary = lines.stream().anyMatch(s -> s.startsWith(">"));
        int links = 0;
        Matcher m = Pattern.compile("\\[[^\\]]+\\]\\([^)]+\\)").matcher(p.text());
        while (m.find()) links++;
        rep.line("llms", "PASS", "llms.txt: HTTP 200 " + kind + "; H1 \""
                + lines.get(0).substring(2).strip() + "\", "
                + (hasSummary ? "summary" : "no summary") + ", "
                + plural(links, "link"));
    }

    static String plural(int n, String word) {
        return n + " " + word + (n == 1 ? "" : "s");
    }

    static String human(long seconds) {
        if (seconds < 3600) return (seconds / 60) + " min";
        if (seconds < 86400) return (seconds / 3600) + " h";
        return (seconds / 86400) + " days";
    }

    static void checkCache(Report rep, String path, Map<String, String> headers) {
        String cc = headers.getOrDefault("cache-control", "");
        Map<String, String> directives = new HashMap<>();
        for (String part : cc.split(",")) {
            String d = part.strip().toLowerCase();
            if (d.isEmpty()) continue;
            int eq = d.indexOf('=');
            directives.put(eq < 0 ? d : d.substring(0, eq),
                           eq < 0 ? "" : d.substring(eq + 1));
        }
        String validator = headers.containsKey("etag") ? "ETag"
                : headers.containsKey("last-modified") ? "Last-Modified"
                : "no validator";
        List<String> layers = new ArrayList<>();
        for (String n : List.of("X-Cache", "CF-Cache-Status", "X-LiteSpeed-Cache",
                                "Age")) {
            String v = headers.get(n.toLowerCase());
```

Listing 10.7 (Continued) *Auditor.java: the same auditor in Java; it prints exactly the same report as Listing 10.5.*

```
            if (v != null) layers.add(n + " " + v);
        }
        List<String> extra = new ArrayList<>();
        if (!layers.isEmpty()) {
            extra.add("layer headers: " + String.join(", ", layers));
        }
        if (cc.isEmpty()) {
            extra.add(0, "caches may guess a lifetime");
            rep.line("cache", "INFO", "cache " + path + ": no Cache-Control; "
                    + validator, extra.toArray(new String[0]));
            return;
        }
        if (directives.containsKey("no-store") || directives.containsKey("no-cache")
                || directives.containsKey("private")) {
            rep.line("cache", "PASS", "cache " + path + ": not kept by shared "
                    + "caches; " + validator, extra.toArray(new String[0]));
            return;
        }
        String age = directives.containsKey("s-maxage") ? directives.get("s-maxage")
                : directives.getOrDefault("max-age", "0");
        long ttl = Long.parseLong(age.isEmpty() ? "0" : age);
        String text = "cache " + path + ": shared caches keep it " + human(ttl)
                + "; " + validator;
        if (ttl > 86400 && path.equals("/robots.txt")) {
            extra.add(0, "crawlers re-read robots.txt within 24 h; a long cache "
                    + "delays every fix");
            rep.line("cache", "WARN", text, extra.toArray(new String[0]));
        } else if (ttl > 86400) {
            extra.add(0, "HTML cached this long needs a purge of every layer "
                    + "after each edit");
            rep.line("cache", "WARN", text, extra.toArray(new String[0]));
        } else {
            rep.line("cache", "PASS", text, extra.toArray(new String[0]));
        }
    }
```

Listing 10.7 (Continued) *Auditor.java: the same auditor in Java; it prints exactly the same report as Listing 10.5.*

```
    /** Collects every schema.org node (a map with @type) in document order. */
    @SuppressWarnings("unchecked")
    static void walk(Object node, List<Map<String, Object>> found) {
        if (node instanceof Map<?, ?> map) {
            if (map.containsKey("@type")) found.add((Map<String, Object>) map);
            for (Object v : map.values()) walk(v, found);
        } else if (node instanceof List<?> list) {
            for (Object v : list) walk(v, found);
        }
    }

    static List<String> typesOf(Map<String, Object> node) {
        Object t = node.get("@type");
        List<String> out = new ArrayList<>();
        if (t instanceof List<?> list) {
            for (Object x : list) out.add(String.valueOf(x));
        } else {
            out.add(String.valueOf(t));
        }
        return out;
    }

    static List<String> blocks(String html) {
        List<String> out = new ArrayList<>();
        Matcher m = LDJSON.matcher(html);
        while (m.find()) out.add(m.group(1));
        return out;
    }

    static String dateModified(String html) {
        for (String block : blocks(html)) {
            List<Map<String, Object>> found = new ArrayList<>();
            try {
                walk(MiniJson.parse(block), found);
            } catch (IllegalArgumentException e) {
                continue;
            }
            for (var node : found) {
                if (node.containsKey("dateModified")) {
                    return String.valueOf(node.get("dateModified"));
                }
            }
        }
        return null;
    }
```

Listing 10.7 (Continued) *Auditor.java: the same auditor in Java; it prints exactly the same report as Listing 10.5.*

```
    static void checkJsonLd(Report rep, String html, String freshHtml) {
        List<String> blocks = blocks(html);
        if (blocks.isEmpty()) {
            rep.line("jsonld", "WARN", "JSON-LD: none found on the home page");
            return;
        }
        List<Map<String, Object>> nodes = new ArrayList<>();
        int bad = 0;
        for (String block : blocks) {
            try {
                walk(MiniJson.parse(block), nodes);
            } catch (IllegalArgumentException e) {
                bad++;
            }
        }
        List<String> types = new ArrayList<>();
        for (var node : nodes) {
            for (String t : typesOf(node)) if (!types.contains(t)) types.add(t);
        }
        List<String> details = new ArrayList<>();
        details.add("types: "
                + (types.isEmpty() ? "none" : String.join(", ", types)));
        for (var node : nodes) {
            if (typesOf(node).contains("FAQPage")) {
                Object q = node.get("mainEntity");
                int n = q instanceof List<?> list ? list.size() : 0;
                details.add("FAQPage with " + n + " questions");
            }
        }
        String served = dateModified(html);
        details.add("dateModified " + (served != null ? served : "absent"));
        String[] d = details.toArray(new String[0]);
        if (bad > 0) {
            rep.line("jsonld", "FAIL", "JSON-LD: " + bad + " of " + blocks.size()
                    + " blocks are not valid JSON; engines skip them", d);
        } else {
            rep.line("jsonld", "PASS", "JSON-LD: " + plural(blocks.size(), "block")
                    + ", all valid JSON", d);
        }
        for (var node : nodes) {
            List<String> kinds = new ArrayList<>(typesOf(node));
            kinds.retainAll(ENTITIES);
            for (String key : List.of("name", "url")) {
                if (!kinds.isEmpty() && !node.containsKey(key)) {
                    rep.line("jsonld", "WARN", "JSON-LD: " + kinds.get(0)
                            + " has no \"" + key + "\"");
```

Listing 10.7 (Continued) *Auditor.java: the same auditor in Java; it prints exactly the same report as Listing 10.5.*

```
                }
            }
        }
        String fresh = dateModified(freshHtml);
        if (served != null && fresh != null && !served.equals(fresh)) {
            rep.line("jsonld", "WARN", "JSON-LD: served page says dateModified "
                    + served + ", origin says " + fresh,
                    "a stale copy is served; purge the cache layers");
        }
    }

    static Map<String, String> audit(String name, String base, String origin) {
        System.out.println("== " + name + "   " + base);
        Report rep = new Report();
        checkRobots(rep, base, origin);
        checkLlms(rep, base);
        Page home = fetch(base + "/");
        Page robots = fetch(base + "/robots.txt");
        checkCache(rep, "/", home.headers());
        checkCache(rep, "/robots.txt", robots.headers());
        String fresh = fetch((origin != null ? origin : base)
                + "/?audit-bypass=1").text();
        checkJsonLd(rep, home.text(), fresh);
        System.out.println();
        return rep.worst;
    }

    public static void main(String[] args) throws Exception {
        List<String[]> targets = new ArrayList<>();       // {name, base, origin}
        Map<String, HttpServer> servers = Map.of();
        if (args.length > 0 && args[0].equals("--local")) {
            Path sitesDir = Path.of(args.length > 1 ? args[1] : "../sites");
            servers = SiteServer.startAll(sitesDir);
            List<Path> sites = new ArrayList<>();
            try (var list = Files.list(sitesDir)) {
                list.filter(p -> Files.isRegularFile(p.resolve("_site.conf")))
                    .forEach(sites::add);
            }
            Map<Integer, String[]> byPort = new TreeMap<>();
            for (Path dir : sites) {
                SiteServer.Conf conf = SiteServer.readConf(dir);
                byPort.put(conf.port, new String[] {dir.getFileName().toString(),
                        "http://127.0.0.1:" + conf.port, conf.originPort == null
                        ? null : "http://127.0.0.1:" + conf.originPort});
            }
            targets.addAll(byPort.values());
```

Listing 10.7 (Continued) *Auditor.java: the same auditor in Java; it prints exactly the same report as Listing 10.5.*

```
        } else {
            for (String url : args) {
                String u = url.replaceAll("/+$", "");
                String name = u.substring(u.indexOf("//") + 2);
                targets.add(new String[] {name, u, null});
            }
        }
        if (targets.isEmpty()) {
            System.out.println("usage: java Auditor --local ../sites | URL ...");
            return;
        }
        List<Map<String, String>> results = new ArrayList<>();
        for (String[] t : targets) results.add(audit(t[0], t[1], t[2]));
        servers.values().forEach(s -> s.stop(0));
        System.out.printf("%-10s %-7s %-7s %-7s %s%n", "site", "robots", "llms",
                "cache", "json-ld");
        for (int k = 0; k < targets.size(); k++) {
            Map<String, String> w = results.get(k);
            System.out.printf("%-10s %-7s %-7s %-7s %s%n", targets.get(k)[0],
                    w.getOrDefault("robots", "-"), w.getOrDefault("llms", "-"),
                    w.getOrDefault("cache", "-"), w.getOrDefault("jsonld", "-"));
        }
        System.out.println("\nNote on llms.txt: cheap hygiene, not a proven "
                + "citation signal. In");
        System.out.println("May 2026, 97% of llms.txt files got zero requests "
                + "(Ahrefs log study");
        System.out.println("of 137,210 domains).");
    }
}
```

10.9.5 Running the auditor

In Python, from the `python` folder:

```
python auditor.py --local ../sites
```

In Java, from the `java` folder:

```
javac -d out *.java
java -cp out Auditor --local ../sites
```

Both print the following report. (The two outputs were compared with `diff` and are identical.)

```
== laundry  http://127.0.0.1:7100
[PASS] robots.txt: HTTP 200; all 6 crawlers may fetch /
       sitemap: http://127.0.0.1:7100/sitemap.xml
       content signal: search=yes, ai-input=yes, ai-train=no
[PASS] robots.txt: served and cache-bypass copies agree
[PASS] llms.txt: HTTP 200 text/plain; H1 "Profitina Laundry", summary, 3 links
[PASS] cache /: shared caches keep it 5 min; ETag
[PASS] cache /robots.txt: shared caches keep it 1 h; ETag
[PASS] JSON-LD: 3 blocks, all valid JSON
       types: LocalBusiness, PostalAddress, FAQPage, Question, Answer, WebPage
       FAQPage with 2 questions
       dateModified 2026-09-20

== cdnblock  http://127.0.0.1:7101
[FAIL] robots.txt: HTTP 200; blocked for GPTBot, ClaudeBot, Google-Extended
       sitemap: http://127.0.0.1:7101/sitemap.xml
[WARN] robots.txt: served copy differs from the origin copy
       origin allows GPTBot, ClaudeBot, Google-Extended
       something in front of the origin rewrites the file
[PASS] llms.txt: HTTP 200 text/plain; H1 "Kopi Kita Roastery", summary, 1 link
[PASS] cache /: shared caches keep it 10 min; ETag
       layer headers: X-Cache MISS
[PASS] cache /robots.txt: shared caches keep it 4 h; ETag
[PASS] JSON-LD: 1 block, all valid JSON
       types: Organization
       dateModified absent

== stale  http://127.0.0.1:7102
[PASS] robots.txt: HTTP 200; all 6 crawlers may fetch /
       no Sitemap line
[PASS] robots.txt: served, cache-bypass and origin copies agree
[WARN] llms.txt: HTTP 200 but text/html
       probably a "soft 404" page, not an llms.txt file
[WARN] cache /: shared caches keep it 30 days; no validator
       HTML cached this long needs a purge of every layer after each edit
       layer headers: X-Cache HIT, Age 5400
[WARN] cache /robots.txt: shared caches keep it 7 days; no validator
       crawlers re-read robots.txt within 24 h; a long cache delays every fix
       layer headers: X-Cache MISS
[PASS] JSON-LD: 2 blocks, all valid JSON
       types: Organization, WebPage
       dateModified 2026-08-01
[WARN] JSON-LD: served page says dateModified 2026-08-01, origin says 2026-09-18
       a stale copy is served; purge the cache layers

== broken  http://127.0.0.1:7103
[FAIL] robots.txt: HTTP 500; crawlers must assume "disallow all"
[INFO] llms.txt: absent (HTTP 404); optional
[INFO] cache /: no Cache-Control; no validator
       caches may guess a lifetime
[INFO] cache /robots.txt: no Cache-Control; no validator
       caches may guess a lifetime
[FAIL] JSON-LD: 1 of 2 blocks are not valid JSON; engines skip them
       types: Organization
       dateModified absent
[WARN] JSON-LD: Organization has no "url"

site       robots  llms    cache   json-ld
laundry    PASS    PASS    PASS    PASS
cdnblock   FAIL    PASS    PASS    PASS
stale      PASS    WARN    WARN    WARN
broken     FAIL    INFO    INFO    FAIL

Note on llms.txt: cheap hygiene, not a proven citation signal. In
May 2026, 97% of llms.txt files got zero requests (Ahrefs log study
of 137,210 domains).
```

Read it site by site, and compare each finding with the chapter.

laundry passes everything. All six crawlers may fetch /; the served copy of robots.txt equals a cache-bypassing copy; llms.txt has a title, a summary and three links; the HTML is kept for 5 minutes and robots.txt for 1 hour, both with an ETag; and the JSON-LD has three valid blocks, with a LocalBusiness and its PostalAddress, an FAQPage with two questions, and a WebPage with a dateModified.

cdnblock fails on robots.txt exactly as profitina.com once did. GPTBot, ClaudeBot and Google-Extended are blocked, although the origin allows all three, and the auditor says where to look: the served copy equals the cache-bypass copy, so it is not a stale cache, but it differs from the origin copy, so "something in front of the origin rewrites the file". Everything else about the site is healthy, which is precisely what makes this fault hard to notice from a browser.

stale passes on robots.txt but warns three times. Its llms.txt is an HTML page with status 200, the soft 404. Its home page may be kept by shared caches for 30 days and has no validator, and its robots.txt for 7 days, beyond the 24 hours that crawlers themselves allow. And the page actually served, a cache HIT that is 5,400 seconds old, says dateModified 2026-08-01, while the origin says 2026-09-18: the visitors and the crawlers are reading the "coming soon" page of six weeks ago.

broken fails twice. Its robots.txt answers 500, so every well-behaved crawler must treat the whole site as disallowed. One of its two JSON-LD blocks has a trailing comma and is invalid; the valid one describes an Organization without a url. It sends no caching headers at all, which the auditor reports as INFO: caches may guess.

The summary table condenses each category to its worst finding, and the note at the bottom repeats Section 10.6.2's caveat, so that nobody reads a PASS for llms.txt as a promise.

10.9.6 A surprise in Python's standard library

Python ships a robots.txt parser, urllib.robotparser. Why did the lab write its own? **Listing 10.8** answers by asking both the same two questions.

Listing 10.8 robots_quirk.py: the same two questions put to urllib.robotparser and to the RFC 9309 reader of **Listing 10.3**.

```
"""Compare urllib.robotparser (first match) with RFC 9309 (longest match)."""
import urllib.robotparser

from robots_rules import Robots

CASES = [
    ("longest match", "User-agent: *\nDisallow: /\nAllow: /blog/\n",
     "/blog/post-1"),
    ("merged groups", "User-agent: GPTBot\nDisallow: /private/\n\n"
     "User-agent: *\nAllow: /\n\nUser-agent: GPTBot\nAllow: /private/faq\n",
     "/private/faq"),
]

for title, text, path in CASES:
    stdlib = urllib.robotparser.RobotFileParser()
    stdlib.parse(text.splitlines())
    print("%-14s %-14s robotparser: %-5s RFC 9309: %s" % (
        title, path, stdlib.can_fetch("GPTBot", path),
        Robots(text).allowed("GPTBot", path)))
```

On Python 3.11, the version used to produce this book's outputs, it prints:

```
longest match  /blog/post-1   robotparser: False RFC 9309: True
merged groups  /private/faq   robotparser: False RFC 9309: True
```

In the first case, `Disallow: /` comes before `Allow: /blog/` in the file, and the older `robotparser` lets the *first* matching rule decide; RFC 9309's longest match allows the path. In the second case, `GPTBot` is named by two groups, and the older parser uses only the first of them. For many years this was the behavior of the standard library, and it is still what you get on older interpreters. Recent maintenance releases rewrote the module: the current source in the 3.13 and 3.14 branches ranks rules by match length, handles `*` and `$`, and merges groups for the same agent [26], so on a current Python 3.14 both columns print `True`. The lesson is not about Python in particular. It is that a library's behavior is a fact about a *version*, and a program that must follow a standard should either pin the version it relies on or test the behavior it needs, as this listing does. Java's standard library has no `robots.txt` parser at all, which is why both versions of the lab carry their own.

10.9.7 Pointing the auditor at a real site

Give the auditor one or more addresses instead of `--local`:

```
python auditor.py https://your-own-site.example
java -cp out Auditor https://your-own-site.example
```

Without an origin address, the `robots.txt` comparison can only detect stale cached copies, not rewriting layers; if your host gives you a way to reach the origin, such as a preview address or the server's own name, compare that too. Audit your own sites, or sites whose owners agree, and keep the request rate at the auditor's one request per resource. Remember also that the auditor's verdicts are about hygiene: a site that passes every check has removed obstacles to being read and understood, which is necessary for being quoted and not sufficient.

10.10 Profitina in Practice

Everything in this chapter describes the marketing site as it stood on 24 September 2026. What is live, and what is only a lesson, is easy to separate.

Live on profitina.com. A WordPress site with a custom bilingual theme and Polylang, Indonesian and English pages side by side, hosted with LiteSpeed Cache behind Cloudflare. A `robots.txt` that allows the search and AI crawlers by name, keeps `/wp-admin/` closed, points to `/wp-sitemap.xml`, mentions `llms.txt`, and states the content signal `search=yes, ai-input=yes, ai-train=no` [12]. An `llms.txt` that summarizes the platform honestly, including that PROFIT is a devnet utility token and not an investment [14]. FAQ structured data and `dateModified` on content pages. An embedded, read-only community widget, and legal links that point to the single home of the Privacy Policy and Terms in the application [1].

Live as a rule of the team. "Always verify the live URL, bypassing cache" after any change that touches crawler access [3]. Purge LiteSpeed first, Cloudflare second. Version every theme change. Prefix the theme's own names.

In the product. The same knowledge reaches Profitina's users through the application. The *Scan* step reads real signals of a site's readiness, among them whether AI crawlers may read it and whether its structured data is valid; the *Optimize* step's Technical Assistant helps with `robots.txt`, in particular with allowing AI crawlers, and with Bing sitemap submission; and every result carries the reminder that readiness estimates are not guarantees of what third-party

engines will do [3]. This chapter teaches the public, standard ground those signals stand on, with a lab that anyone can run.

Not claimed. Profitina does not claim that `llms.txt` causes citations, that FAQ markup produces search features after May 2026, or that any file guarantees an AI answer engine will quote a site. It does not query ChatGPT, Gemini or Perplexity to measure citations; those engines are the targets whose readiness is estimated, never components the platform depends on [3].

10.11 Common Misunderstandings

"My `robots.txt` is correct; I checked the file." You checked *a* file. Crawlers receive whatever the public URL returns after every layer in front of your server has had its say. Check the URL, with and without a cache-bypassing query string (Section 10.5.5).

"`User-agent: *` with `Allow: /` welcomes every crawler." It welcomes every crawler that has no group of its own. A single group anywhere in the file that names `GPTBot`, including one added by a CDN or a plugin, takes `GPTBot` out of the `*` group entirely (Equation (10.1)).

"The first matching rule wins." Under RFC 9309 the *longest* matching rule wins, and `Allow` wins a tie (Equation (10.2)). The order of lines does not matter. Some older parsers, including older versions of Python's `urllib.robotparser`, did use the first match (Section 10.9.6).

"If `robots.txt` breaks, crawlers will just read everything." Only if it breaks with a 4xx. A 5xx, or no answer, tells crawlers to assume that *everything* is disallowed (**Table 10.2**).

"No `Cache-Control` header means no caching." It means the caches choose a lifetime themselves, heuristically (Equation (10.4)).

"Purging both caches is enough; the order is a detail." In the wrong order a single request between the purges refills the outer cache with the old copy for a full lifetime (Equation (10.6)).

"`llms.txt` is how AI finds my site." In May 2026, 97% of `llms.txt` files received no request at all, and AI retrieval bots made 1.1% of the requests that did arrive [17]. It is cheap hygiene; access is decided by `robots.txt`.

"FAQ markup is pointless now that Google dropped FAQ rich results." The visual feature is gone; the markup is still valid, still harmless, and still an unambiguous statement of questions and answers for any program that reads the page [24].

"A `Content-Signal` line blocks AI training." It states a preference about use. Whether a crawler may fetch the page at all is still decided by the `Allow` and `Disallow` rules, and whether a company honors the signal is up to the company.

10.12 Chapter Summary

- profitina.com is a WordPress site with a custom bilingual theme (Polylang, real Indonesian and English pages) behind two cache layers: LiteSpeed Cache at the host and Cloudflare at the edge. The application at app.profitina.com is separate, and the legal texts live only there.
- A shared global name in WordPress's loop (`$id`) once mixed languages on the site's hero labels; the root-cause fix was a prefixed name (`$pf_id`). Code that runs inside a framework's loop must not borrow the framework's names.
- RFC 9309 selects groups by Equation (10.1): every group naming the crawler, merged, or else the `*` group; and it selects the rule by Equation (10.2): the longest match, with `Allow` winning a

tie. Wildcards `*` and `$` are part of the standard. A 4xx for `robots.txt` means "no rules"; a 5xx means "assume everything is disallowed".

- A CDN's managed `robots.txt` prepended groups for `GPTBot` and `ClaudeBot`, which silently overrode the site's welcome. The rule that came out of it: always verify the live URL, bypassing caches, and compare with the origin.
- `llms.txt` is published by 9.3% of the top 1,000 and 8.3% of the top 10,000 domains (September 2026), but 97% of the files got no request in May 2026. Profitina publishes one as hygiene and claims nothing more.
- HTTP freshness is Equation (10.3), with the shared-cache lifetime of Equation (10.4). With two layered caches, purge from the inside out; the stale window is then Equation (10.5) instead of the much longer Equation (10.6).
- FAQ, organization and page nodes with `dateModified` make the site's facts unambiguous; after 7 May 2026 FAQ markup no longer produces Google rich results but remains valid.
- The lab's auditor, in Python and Java, finds a rewritten `robots.txt`, a server error on `robots.txt`, a soft-404 `llms.txt`, month-long HTML caching, a stale cached page, and invalid JSON-LD on four local sample sites, and prints the same report in both languages.

10.13 Exercises

Exercise 10.1. Divide the following file into groups, and for each group list its product tokens and its rules. Which lines belong to no group?

```
User-agent: Bingbot
User-agent: Googlebot
Disallow: /search
Sitemap: https://shop.example/sitemap.xml
User-agent: *
Disallow: /cart/
Allow: /cart/help
```

Exercise 10.2. For the file of Exercise 10.1, decide by hand, using Equation (10.1) and Equation (10.2), whether each of the following requests is allowed: `Googlebot` for `/search?q=batik`; `Googlebot` for `/cart/`; `GPTBot` for `/cart/`; `GPTBot` for `/cart/help`; `GPTBot` for `/cart/helpdesk`.

Exercise 10.3. A group contains `Disallow: /*.pdf$` and `Allow: /public/`. Decide, with the length of each pattern, whether `/public/report.pdf`, `/public/report.pdf?v=2` and `/files/a.pdf` are allowed.

Exercise 10.4. A site owner's file is `User-agent: GPTBot` / `Allow: /blog/`, followed by `User-agent: *` / `Allow: /`. A CDN prepends `User-agent: GPTBot` / `Disallow: /`. (a) Under RFC 9309, may `GPTBot` fetch `/blog/post-1`? `/about`? (b) What would a crawler that obeys only the *first* group naming it decide? (c) What should the owner change?

Exercise 10.5. For each response to `GET /robots.txt`, say what an RFC 9309 crawler does: (a) 404; (b) 503; (c) 200 with an empty body; (d) 200 with an HTML error page that says "Page not found". For (d), what would the lab's `Robots` class conclude, and why is that a problem?

Exercise 10.6. A page is sent with `Cache-Control: public, max-age=600, s-maxage=86400`. (a) For how long is it fresh in a browser, and in a CDN? (b) The same response also carries `CDN-Cache-Control: max-age=60`. Now how long is it fresh in the CDN, according to RFC 9213? (c) Which lifetime does the lab's auditor report for the original header, and why?

Exercise 10.7. An edit is published at $t_0 = 0$. One cache is purged at $t_1 = 1$ minute and the other at $t_2 = 10$ minutes. The edge keeps the page for $T_C = 2$ hours. Compute the stale window in the right order and, in the wrong order, for a request at $t_r = 6$ minutes and for the worst possible request time.

Exercise 10.8. Ahrefs found valid `llms.txt` files on 38,360 of 137,210 domains, and 97% of those files received no requests in May 2026. (a) What share of all studied domains had a valid file? (b) About how many files received at least one request? (c) Why is it wrong to conclude that "3% of sites benefit from `llms.txt`"?

Exercise 10.9. Add Anthropic's `Claude-SearchBot` to the list of crawlers the auditor checks, in both languages. Predict the `robots.txt` line of each sample site before running, then run the auditor and compare.

Exercise 10.10. Change the auditor, in both languages, so that a `robots.txt` without any `Sitemap` line produces an extra `WARN` line. Which sample site's summary changes, and to what?

Exercise 10.11. Why does `check_jsonld` fetch the origin's home page with `?audit-bypass=1` rather than without a query string? What would go wrong for the `stale` site if it used the public address without a query string instead?

Exercise 10.12. Write an `llms.txt` for the `cdnblock` site's fictional coffee roaster, Kopi Kita Roastery, that the auditor accepts with a `PASS` and that has a summary, two sections of links, and an `Optional` section. Which parts of the file does the auditor check, and which parts can only a person check?

10.14 Solutions to the Exercises

Solution 10.1. There are two groups. Group 1 has the product tokens `bingbot` and `googlebot` (two consecutive `User-agent` lines) and one rule, `Disallow: /search`. The `Sitemap` line belongs to no group; it does not close group 1 either, but the following `User-agent` line starts a new group because the previous significant line of group 1 was a rule. Group 2 has the token `*` and the rules `Disallow: /cart/` and `Allow: /cart/help`.

Solution 10.2. `Googlebot` is named by group 1, so by Equation (10.1) it obeys only group 1 and never the `*` group. `/search?q=batik` begins with `/search`, so `Disallow: /search` matches and it is blocked. `/cart/` matches no rule of group 1, so it is *allowed* for `Googlebot`: the `Disallow: /cart/` of the `*` group does not apply to a crawler with its own group. `GPTBot` is named by no group, so it obeys group 2. `/cart/` matches `Disallow: /cart/` (length 6) only: blocked. `/cart/help` matches `Disallow: /cart/` (6) and `Allow: /cart/help` (10); the longer wins: allowed. `/cart/helpdesk` also begins with `/cart/help`, so the same two rules match and it is allowed too, a reminder that patterns match prefixes.

Solution 10.3. The pattern `/*.pdf$` has 7 octets and `/public/` has 8. For `/public/report.pdf` both match (the first because the path ends in `.pdf`), and the longer, `Allow: /public/` with 8 octets, wins: allowed. For `/public/report.pdf?v=2` the `$` pattern does not match, because the path does not end in `.pdf`; only `Allow: /public/` matches: allowed. For `/files/a.pdf` only `Disallow: /*.pdf$` matches: blocked.

Solution 10.4. (a) The served file has two groups naming `GPTBot`, the prepended one and the owner's. By Equation (10.1) they are merged into the rules `Disallow: /` (length 1) and `Allow: /blog/` (length 6), and the `*` group is ignored. For `/blog/post-1` both match and the longer `Allow`

wins: allowed. For `/about` only `Disallow: /` matches: blocked. The owner's `*` group, which says `Allow: /`, plays no part. (b) A crawler that obeys only the first group naming it sees `Disallow: /` alone and is blocked everywhere, including `/blog/`. (c) Turn off the CDN's managed `robots.txt` feature (or configure it to leave the crawlers the owner wants alone), and state the owner's intent for each crawler in full in its own group, for example `Allow: /` for `GPTBot`, so that no merged reading can shrink it. Then verify the public URL.

Solution 10.5. (a) A 404 means the file is "unavailable": the crawler may read everything. (b) A 503 is a server error, "unreachable": the crawler must assume that everything is disallowed (and Google pauses crawling for the first 12 hours while it retries). (c) An empty file has no groups and no rules, so every path is allowed. (d) The crawler receives an HTML page with status 200 and parses it as `robots.txt`. HTML lines have no `key: value` structure that forms groups, so `Robots` finds no groups and concludes that everything is allowed. That happens to be harmless here, but it means the site is not saying what it intended; if the page's text happened to contain the lines `User-agent: *` and `Disallow: /`, they would even be obeyed. The status code tells the crawler the wrong story, which is exactly the soft-404 problem that `check_llms` catches for `llms.txt`.

Solution 10.6. (a) A browser, a private cache, uses `max-age`: 600 seconds, 10 minutes. A CDN, a shared cache, uses `s-maxage` when present, by Equation (10.4): 86,400 seconds, one day. (b) RFC 9213 says a CDN cache that finds a valid `CDN-Cache-Control` must use it and ignore `Cache-Control`: the page is fresh in the CDN for 60 seconds, while browsers still keep it for 10 minutes. (c) The auditor reads only `Cache-Control` and reports the shared-cache lifetime, `s-maxage`: it prints `shared caches keep it 1 days` (its small `human` function does not bother with the singular), and because 86,400 seconds is not *greater* than one day, the line is a `PASS`. Adding `CDN-Cache-Control` support is a natural extension: read that header first when it is present, as the RFC says.

Solution 10.7. Right order, Equation (10.5): $W = t_2 - t_0 = 10 - 0 = 10$ minutes. Wrong order, Equation (10.6), with $t_r = 6$: $W = 6 + 120 - 0 = 126$ minutes. The worst case is a request just before the second purge, $t_r \to t_2 = 10$: $W \to 10 + 120 = 130$ minutes, a little over two hours instead of ten minutes.

Solution 10.8. (a) $38\,360/137\,210 \approx 0.280$, about 28% of the studied domains. (b) $3\% \times 38\,360 = 0.03 \times 38\,360 \approx 1\,151$ files. (c) A request is not a benefit. Most of the requests that did arrive came from bots that are not AI tools at all, such as SEO audit tools, and only 1.1% came from AI retrieval bots [17]; and even a request by an AI system says nothing about whether the site was then quoted. Use is necessary for an effect, not evidence of one.

Solution 10.9. In Python, add `"Claude-SearchBot"` to `BOTS`; in Java, add it to `List.of(...)` in `BOTS`. Prediction: no sample site names `Claude-SearchBot`, so it always falls back to the `*` group. `laundry` and `stale` allow everyone under `*`, so their line becomes `all 7 crawlers may fetch /`. `cdnblock`'s `*` group allows everyone and the managed block does not name `Claude-SearchBot`, so the blocked list stays `GPTBot, ClaudeBot, Google-Extended`. `broken` still fails with HTTP 500 before any crawler is considered. Running the modified Python auditor confirms it:

```
== laundry  http://127.0.0.1:7100
[PASS] robots.txt: HTTP 200; all 7 crawlers may fetch /
== cdnblock  http://127.0.0.1:7101
[FAIL] robots.txt: HTTP 200; blocked for GPTBot, ClaudeBot, Google-Extended
== stale  http://127.0.0.1:7102
[PASS] robots.txt: HTTP 200; all 7 crawlers may fetch /
== broken  http://127.0.0.1:7103
[FAIL] robots.txt: HTTP 500; crawlers must assume "disallow all"
```

Solution 10.10. In `check_robots` of **Listing 10.5**, insert before the line that fetches `?audit-bypass=1`:

```
    if not robots.sitemaps:
        rep.line("robots", "WARN", "robots.txt: no Sitemap line; add one so "
                 "crawlers find every page")
```

and in `checkRobots` of **Listing 10.7**, before the line that fetches the bypass copy:

```
        if (robots.sitemaps.isEmpty()) {
            rep.line("robots", "WARN", "robots.txt: no Sitemap line; add one so "
                    + "crawlers find every page");
        }
```

Only `stale` has a `robots.txt` without a `Sitemap` line (`broken` never gets that far, because its file answers 500). Its section gains one line, and its `robots` column in the summary changes from `PASS` to `WARN`; both languages print:

```
== stale  http://127.0.0.1:7102
[PASS] robots.txt: HTTP 200; all 6 crawlers may fetch /
       no Sitemap line
[WARN] robots.txt: no Sitemap line; add one so crawlers find every page
[PASS] robots.txt: served, cache-bypass and origin copies agree
```

Solution 10.11. The comparison is meant to contrast what visitors receive with what is *true now*. For a site with a known origin address, the origin never serves cached copies, so the query string is only a safeguard; for a site without one, the query string is the only way the auditor has of getting past a cache keyed on the full URL. If `check_jsonld` fetched the public address of `stale` without a query string, it would receive the same cached `HIT` as the served page, both dates would be `2026-08-01`, and the stale page would pass unnoticed. The whole point of the check is to ask a question the cache cannot answer.

Solution 10.12. For example:

```
# Kopi Kita Roastery

> Kopi Kita Roastery is a fictional coffee roaster in Malang that sells
> single-origin beans from Java, roasted every Monday.

Orders close on Sunday evening; delivery is within Java only.

## Products

- [Beans](http://127.0.0.1:7101/beans.html): origins, roast levels, prices
- [Brewing guide](http://127.0.0.1:7101/brew.html): grind and ratio per method

## Company

- [About](http://127.0.0.1:7101/about.html): who roasts, and where

## Optional

- [Blog](http://127.0.0.1:7101/blog.html): harvest notes and events
```

The auditor checks that the file is served with status 200 as `text/plain` (or `text/markdown`), that its first non-blank line is an H1 (`#`), whether a block-quote summary exists, and how many Markdown links it contains; this file earns `PASS` with "summary, 4 links". Only a person can check that the summary is true, that the links work and lead to pages that say what the notes promise, and that the file agrees with the rest of the site, the same "keep the facts true" rule that Chapter 2 applied to structured data.

References

[1] Profitina, "Profitina – Gather · Grow · Give," marketing site. [Online]. Available: https://profitina.com (accessed Sep. 24, 2026).

[2] W3Techs, "Usage statistics and market share of WordPress," Sep. 2026. [Online]. Available: https://w3techs.com/technologies/details/cm-wordpress (accessed Sep. 24, 2026).

[3] Profitina, *Profitina Whitepaper — Public Version v24*, Sep. 2026. [Online]. Available: https://app.profitina.com/static/whitepaper/

[4] WP SYNTEX, "Polylang," WordPress plugin directory, version 3.8.9. [Online]. Available: https://wordpress.org/plugins/polylang/ (accessed Sep. 24, 2026).

[5] Google Search Central, "Tell Google about localized versions of your page." [Online]. Available: https://developers.google.com/search/docs/specialty/international/localized-versions (accessed Sep. 24, 2026).

[6] WordPress Developer Resources, "setup_postdata()." [Online]. Available: https://developer.wordpress.org/reference/functions/setup_postdata/ (accessed Sep. 24, 2026).

[7] M. Koster, G. Illyes, H. Zeller, and L. Sassman, "Robots Exclusion Protocol," IETF RFC 9309, Sep. 2022.

[8] Google Search Central, "How Google interprets the robots.txt specification." [Online]. Available: https://developers.google.com/search/docs/crawling-indexing/robots/robots_txt (accessed Sep. 24, 2026).

[9] Cloudflare, "Managed robots.txt," Cloudflare Bots documentation. [Online]. Available: https://developers.cloudflare.com/bots/additional-configurations/managed-robots-txt/ (accessed Sep. 24, 2026).

[10] Cloudflare, "Control content use for AI training with Cloudflare's managed robots.txt and blocking for monetized content," *The Cloudflare Blog*, Jul. 1, 2025. [Online]. Available: https://blog.cloudflare.com/control-content-use-for-ai-training/

[11] Cloudflare, "Your site, your rules: new AI traffic options for all customers," *The Cloudflare Blog*, Jul. 1, 2026. [Online]. Available: https://blog.cloudflare.com/content-independence-day-ai-options/

[12] Profitina, "robots.txt," profitina.com. [Online]. Available: https://profitina.com/robots.txt (accessed Sep. 24, 2026).

[13] J. Howard, "The /llms.txt file," llmstxt.org, proposed Sep. 3, 2024. [Online]. Available: https://llmstxt.org/

[14] Profitina, "llms.txt," profitina.com. [Online]. Available: https://profitina.com/llms.txt (accessed Sep. 24, 2026).

[15] Rankability, "LLMS.txt adoption," data page, scan of Sep. 18, 2026 (Tranco list of Sep. 17, 2026). [Online]. Available: https://www.rankability.com/data/llms-txt-adoption/

[16] PPC Land, "llms.txt adoption rises 8.8x but 97% of files get zero AI requests," Jul. 2, 2026. [Online]. Available: https://ppc.land/llms-txt-adoption-rises-8-8x-but-97-of-files-get-zero-ai-requests/

[17] L. Linehan and X. Guan, "We analyzed 137K sites: 97% of llms.txt files never get read," *Ahrefs Blog*, Jun. 15, 2026. [Online]. Available: https://ahrefs.com/blog/llmstxt-study/

[18] R. Fielding, M. Nottingham, and J. Reschke, "HTTP Caching," IETF RFC 9111, Jun. 2022.

[19] S. Ludin, M. Nottingham, and Y. Wu, "Targeted HTTP Cache Control," IETF RFC 9213, Jun. 2022.

[20] Cloudflare, "Cloudflare cache responses," Cloudflare Cache documentation. [Online]. Available: https://developers.cloudflare.com/cache/concepts/cache-responses/ (accessed Sep. 24, 2026).

[21] LiteSpeed Technologies, "Troubleshooting guide," LSCache for WordPress documentation. [Online]. Available: https://docs.litespeedtech.com/lscache/lscwp/troubleshoot/ (accessed Sep. 24, 2026).

[22] Cloudflare, "Default cache behavior," Cloudflare Cache documentation. [Online]. Available: https://developers.cloudflare.com/cache/concepts/default-cache-behavior/ (accessed Sep. 24, 2026).

[23] Schema.org, "Schema.org releases," version 30.1, Sep. 16, 2026. [Online]. Available: https://schema.org/docs/releases.html

[24] Search Engine Journal, "Google drops FAQ rich results from search," May 10, 2026. [Online]. Available: https://www.searchenginejournal.com/google-drops-faq-rich-results-from-search/574429/

[25] P. Aggarwal, V. Murahari, T. Rajpurohit, A. Kalyan, K. Narasimhan, and A. Deshpande, "GEO: Generative Engine Optimization," in *Proc. 30th ACM SIGKDD Conf. Knowledge Discovery and Data Mining (KDD '24)*, 2024. arXiv:2311.09735.

[26] Python Software Foundation, "Lib/urllib/robotparser.py," CPython source, 3.14 branch. [Online]. Available: https://github.com/python/cpython/blob/3.14/Lib/urllib/robotparser.py (accessed Sep. 24, 2026).

Chapter 11

Privacy and Compliance by Design (UU PDP)

By the end of this chapter you will be able to explain Indonesia's Personal Data Protection Law, UU PDP No. 27/2022, in plain words: what counts as personal data, who is responsible for it, which rights a person holds over data about themselves, and what a company must do to keep it safe. You will know where the law stands in September 2026, with its implementing regulation, PP 33/2026, signed in July 2026 and taking effect in January 2027, and its supervisory agency still being formed, and you will be able to cite your sources. You will understand privacy by design as an engineering method rather than a slogan, and you will be able to follow each of the measures Profitina ships from the law to the code: passwords stored with Argon2id, consent that is explicit and recorded with a version and a time, login rate limiting, data export, changing an email address or password, full account erasure, and legal pages published as ordinary HTML. You will be able to compute why a slow password hash protects users and a fast one does not, and to reason about a sliding-window rate limiter by hand. You will know why a right that has no button is not yet a right a person can use. Finally, you will have built lab G7, a small application in Python and Java that logs consent, hashes passwords with PBKDF2 from the standard library (or Argon2id, when the optional package is installed), limits failed logins, exports a person's data as JSON, and erases an account so thoroughly that a test can prove nothing personal remains.

11.1 Where We Left Off

Chapter 10 looked at Profitina from the outside: the marketing site, its caches, and the files that tell crawlers what they may read. It ended with a principle that applies just as well to this chapter: what matters is what actually happens, not what a file or a document says should happen.

This chapter turns to the people who use the platform and the data they leave in it. Every service that lets people register collects personal data, if only an email address. In Indonesia, since October 2022, that collection has been governed by a comprehensive law, UU PDP, and since October 2024 every organization has been expected to comply with it in full. A student who builds a laundry app for a class and a company that runs a platform with thousands of users are subject to the same principles; what changes with size is the scale of the consequences.

Profitina was built with that law as a design input, not as a document written after the fact. Its roadmap contains a phase devoted to it, completed before and around the July 2026 launch: Argon2id password hashing, explicit consent, login rate limiting, self-service data rights, and legal pages [1]. Chapter 3 walked through the user's side of those features. Here we look at them from the engineer's side: what the law requires, how each requirement becomes a piece of design, and how to test that the design does what it promises.

The chapter has three parts. Sections 11.2 and 11.3 explain the law and its status in 2026. Sections 11.4 to 11.9 take privacy by design apart into concrete measures: minimal data, password hashing, consent, rate limiting, data-subject rights, and legal pages. Section 11.10 builds

lab G7, and Sections 11.11 to 11.15 close with Profitina in practice, common misunderstandings, a summary, exercises and full solutions.

This chapter explains law for engineers. It is not legal advice, and a company that processes personal data at scale should consult a lawyer. But engineers cannot leave the law to lawyers either: almost every obligation in UU PDP is, in the end, carried out by code.

11.2 UU PDP No. 27/2022 in Plain Language

11.2.1 One law, sixteen chapters

Undang-Undang Nomor 27 Tahun 2022 tentang Pelindungan Data Pribadi, the Personal Data Protection Law, usually shortened to UU PDP, was enacted and promulgated on 17 October 2022 [2]. It has 76 articles (*pasal*) in 16 chapters. It is Indonesia's first comprehensive data-protection statute: before it, rules about personal data were scattered across sector regulations. Readers who know Europe's GDPR will recognize its shape, but UU PDP is its own law, with its own terms, its own articles and its own enforcement.

The law gave organizations two years to adjust: Pasal 74 required every party that processes personal data to comply "at the latest two years" after the law took effect, that is, by 17 October 2024 [2]. Since then, compliance has not been optional.

11.2.2 What is personal data?

Personal data is data about a natural person who is identified, or who can be identified, on its own or combined with other information. The law divides it into two kinds (Pasal 4) [2]:

- **General personal data** (*data pribadi yang bersifat umum*): full name, sex, nationality, religion, marital status, and personal data that, combined, identifies a person. An email address such as `ayu@example.com` identifies its owner and is personal data.
- **Specific personal data** (*data pribadi yang bersifat spesifik*): health data and information, biometric data, genetic data, criminal records, data about children, personal financial data, and other data as regulated. Processing it carries a higher risk and stricter duties.

> **DEFINITION.** A *data subject* (*subjek data pribadi*) is the person the data is about. A *controller* (*pengendali data pribadi*) decides the purposes and means of processing. A *processor* (*prosesor data pribadi*) processes data on a controller's behalf. *Processing* covers the whole life of the data: collecting, analyzing, storing, correcting, displaying, transferring, and erasing or destroying it.

For a platform like Profitina, the platform is the controller for its users' accounts; a hosting company or an email-delivery service that handles data on its behalf is a processor. The distinction matters because the controller carries the obligations, and must make sure its processors meet them too.

11.2.3 The principles

Pasal 16 lists the principles every processing must follow [2]. Paraphrased, they say that collection is limited and specific, lawful and transparent; that processing matches its stated purpose; that the subject's rights are guaranteed; that data is accurate, complete, not misleading and up to date; that it is protected against unauthorized access, disclosure, alteration, misuse, destruction and loss; that the purpose, the activities and any failure of protection are

communicated; that data is destroyed or deleted when its retention period ends or at the subject's request; and that processing is done responsibly and can be demonstrated to be so.

The last clause, *can be demonstrated*, is the engineer's clause. A system that protects data but cannot show that it does, with a record of who consented to what and when, or a test that proves an erasure really erased, leaves its owner unable to answer the most basic question a regulator or a user can ask.

11.2.4 Lawful bases

Processing needs a basis. Pasal 20 lists six: explicit, valid consent of the subject for one or more specific purposes; performance of a contract to which the subject is a party; a legal obligation of the controller; the vital interests of the subject; a task carried out in the public interest, in public service, or under the controller's legal powers; and other legitimate interests, balanced against the subject's rights [2].

When the basis is consent, Pasal 21 lists what the person must be told first: the legality of the processing, its purpose, the type and relevance of the data, how long documents containing it are kept, details about the information collected, how long the processing lasts, and the subject's rights. Pasal 22 says consent is given in writing or in a recorded form, electronically or not, and that a request for consent must be clearly distinguishable from other matters, in a format that is understandable and easy to access, and in clear, plain language; consent that does not meet these conditions is void [2]. Section 11.6 turns these sentences into a registration form and a database record.

11.2.5 The rights of the data subject

Chapter IV of the law, Pasal 5 to 15, gives the person a set of rights [2]. The ones an ordinary user will want are:

- **Information** (Pasal 5): to know the identity of the controller, the legal basis and purpose of the request and use of the data, and the controller's accountability.
- **Correction** (Pasal 6): to complete, update and correct errors or inaccuracies in their data.
- **Access** (Pasal 7): to access and obtain a copy of their data.
- **Erasure** (Pasal 8): to end processing, and to delete or destroy their data.
- **Withdrawal of consent** (Pasal 9).
- **Objection to automated decisions** (Pasal 10): to object to decisions based solely on automated processing, including profiling, that have legal or similarly significant effects.
- **Portability** (Pasal 13): to obtain and use their data in a format that is structured, commonly used and readable by an electronic system, and to send it to another controller.

The law also gives controllers deadlines. For example, a controller must give the subject access to their data no later than 3×24 hours after the request (Pasal 32) [2]. A right with a 72-hour deadline is a right that a small team must be able to fulfill without a special project, which is one of the strongest arguments for making it self-service.

11.2.6 The controller's duties

The obligations of the controller fill most of the law. The ones that shape a system's design are these [2]:

- **Security** (Pasal 35): protect personal data with technical and operational measures appropriate to the risk.
- **Preventing unauthorized access** (Pasal 39): "Pengendali Data Pribadi wajib mencegah Data Pribadi diakses secara tidak sah", the controller must prevent personal data from being accessed unlawfully, with a reliable security system.
- **Impact assessment** (Pasal 34): assess the impact before processing that carries a high risk, such as large-scale processing of specific data or systematic monitoring.
- **Deletion and destruction** (Pasal 43 to 45): delete data that is no longer needed for its purpose, when consent is withdrawn, when the subject asks, or when it was obtained unlawfully; and tell the subject.
- **Breach notification** (Pasal 46): in a failure of personal-data protection, notify the subjects and the supervisory agency in writing no later than 3 × 24 hours, stating what data was exposed, when and how, and what is being done about it.
- **A data protection officer** (Pasal 53): required for processing for public services, for large-scale systematic monitoring, and for large-scale processing of specific data or criminal records.
- **Cross-border transfers** (Pasal 56): allowed only when the receiving country's protection is equal or higher, or with adequate safeguards, or with consent.

11.2.7 Sanctions

Violations carry administrative sanctions under Pasal 57: a written warning, temporary suspension of processing, deletion or destruction of the data, and administrative fines of up to 2% of annual revenue [2]. Separate criminal provisions, Pasal 67 to 73, punish unlawfully obtaining, disclosing or using personal data that does not belong to the offender, and creating false personal data, with imprisonment of four to six years and fines of up to six billion rupiah depending on the offence [2]. The numbers are worth knowing not as a threat but as a measure of how seriously the law takes the subject: the fine is scaled to revenue, so it does not become trivial for a large company.

11.3 Where the Law Stands in 2026

11.3.1 PP 33/2026, the implementing regulation

A law like UU PDP states principles and obligations; the procedures are set out in implementing regulations. For almost four years after enactment, the main implementing regulation was a draft. That changed in July 2026. *Peraturan Pemerintah Nomor 33 Tahun 2026 tentang Peraturan Pelaksanaan Undang-Undang Nomor 27 Tahun 2022 tentang Pelindungan Data Pribadi* was set and promulgated on 16 July 2026, as State Gazette 2026 No. 88, with 225 articles in 12 chapters, and it takes effect on 16 January 2027 [3].

Its chapters cover the classification of personal data, the bases for processing, the rights of data subjects, the obligations of controllers and processors, cross-border transfers, data protection officers, data protection impact assessments, compliance supervision, international cooperation, and administrative sanctions and dispute resolution [3]. One sentence of it summarizes a whole engineering discipline: "pengendali data pribadi wajib memiliki dasar pemrosesan data pribadi sebelum melakukan pemrosesan", the controller must have a basis for processing before it processes [3]. In other words: decide and record *why* before the first row is written.

The six months between promulgation and effect exist so that organizations can adjust. For a system that was designed to the law itself, the adjustment is mostly a matter of checking procedures against the regulation's details; for a system that was not, it may mean rebuilding parts of it.

11.3.2 The supervisory agency

UU PDP (Pasal 58 to 61) creates a supervisory agency, established by and responsible to the President, to supervise compliance, impose administrative sanctions and handle complaints. In September 2026 that agency does not yet exist. According to the Ministry of Communication and Digital Affairs (Komdigi), the draft Presidential Regulation (Perpres) establishing it completed its planning, drafting and harmonization and was submitted for the President's decision by letter on 20 May 2026 [4]. In the meantime Komdigi carries the supervisory functions: incident handling, receiving complaints about alleged violations and misuse of data, and consultation services [4]. Observers are cautious about the timing. One policy expert, quoted in September 2026, estimated that the agency would be able to enforce compliance standards effectively "in 2028, at the fastest", because even a signed regulation must be followed by the work of forming an institution [4], [5].

Figure 11.1 puts the dates together with Profitina's own.

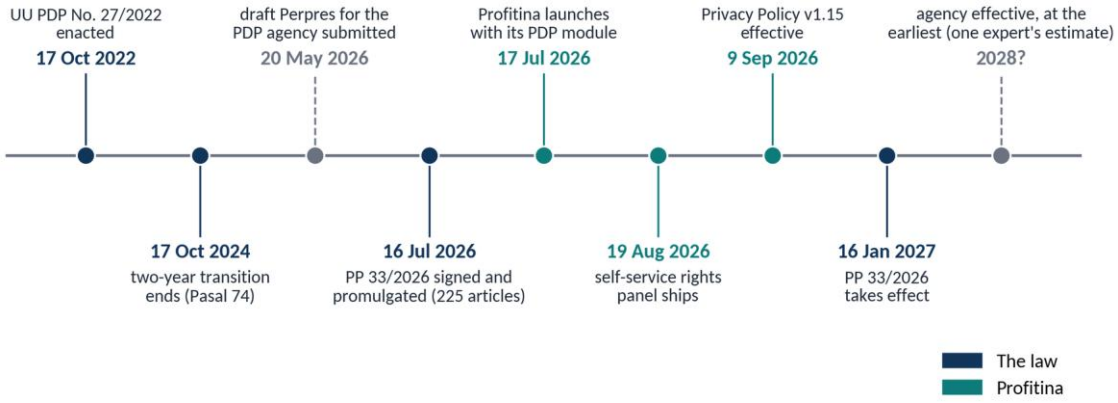


Figure 11.1 UU PDP from enactment to the implementing regulation, with the milestones of Profitina's own compliance work. The agency's date is an expert's estimate, not a schedule.

11.3.3 What this means for a small team

It would be easy to read "the agency is not yet operational" as "the law is not yet enforced". That reading is wrong in two ways. First, the law's obligations have applied since October 2024 whatever the state of the agency, and Komdigi supervises in the meantime. Second, and more important for an engineer, obligations that are designed in are cheap, and obligations that are added later are expensive. A consent log that has recorded every registration since launch cannot be reconstructed after the fact; a password table full of fast hashes cannot be upgraded without waiting for every user to log in again. The time to build for the law is before the first user arrives, and Profitina did.

11.4 Privacy by Design as an Engineering Method

11.4.1 The idea

"Privacy by design" is a phrase coined and developed by Ann Cavoukian, then Information and Privacy Commissioner of Ontario, who summarized it in seven foundational principles: be

proactive, not reactive; make privacy the default setting; embed privacy into the design; aim for full functionality, positive-sum rather than zero-sum; provide end-to-end security over the whole lifecycle; keep it visible and transparent; and respect the user [6]. The principles are old and general. What makes them useful is turning each into a decision that a developer can make, test and defend.

11.4.2 Minimize first

The cheapest data to protect is data you never collected. Profitina's registration asks for two things: an email address and a password [1], [7]. There is no phone field anywhere in the product, and an email address is never publishable on a profile. Community handles are neutral, a letter and eight hexadecimal digits by default, which the user can change, and they are checked so that they cannot be a student's registration number; names are masked for anonymous visitors. Profile extras, a photo and up to four social links, each need their own consent, are private by default, and can be withdrawn at any time.

Each of those choices removes a whole class of risk. A leaked table of email addresses is bad; a leaked table of email addresses, phone numbers, full names and student numbers is much worse. Every field that is not collected is a field that cannot be leaked, cannot be subpoenaed, and does not need a retention rule.

11.4.3 Write it down: the record of processing

Pasal 16's "can be demonstrated" requires a record of what is processed, why, on which basis, for how long and by whom. The usual name for that record is a *record of processing activities*, ROPA. Profitina has a house rule that ties it to development: every new feature that touches personal data must update the ROPA and, where needed, the Privacy Policy, in the same piece of work [1]. A feature is not finished while its privacy paperwork is missing, exactly as it is not finished while its tests are missing.

11.4.4 Private by default, everywhere

Minimization continues after registration. Scan and content history is private to its owner. Admin sessions and user sessions are mutually exclusive, so an administrator cannot browse as themselves and as a user at the same time [7]. Direct messages are one-to-one and administrators cannot read them, with one narrow exception: a frozen copy of a message that a participant has *reported*. Server errors are recorded without raw request bodies or headers, with the IP address hashed, and error logs are deleted after 90 days by a purge job that actually runs [1]. The phrase "that actually runs" is not decoration. A retention rule that exists only in a policy document is not a retention rule; a scheduled job with a test is.

11.4.5 One home for the legal text

Chapter 10 told how the Privacy Policy was moved to a single home in the application, with the marketing site summarizing and linking [1]. The engineering principle behind it, *remove duplication, do not synchronize it*, is also a privacy principle: a user who reads two versions of a policy cannot know which one binds the company, and a company that maintains two versions will one day contradict itself. Section 11.9 returns to the legal pages.

11.5 Passwords: Why Argon2id

11.5.1 Never store the password

A system that checks passwords must remember something about each password, but it must not remember the password itself. If it did, anyone who obtained the database, an attacker, a careless backup, a curious insider, would hold every user's password, and because people reuse passwords, the damage would spread to their email, their bank and their social media.

The standard answer is a **one-way function**: a function h that is easy to compute and practically impossible to invert. At registration the system stores h(password); at login it computes h of what was typed and compares. Encryption is the wrong tool, because encryption is designed to be reversed by whoever holds the key, and a key on the same server as the database is stolen together with it. Profitina's Privacy Policy states the principle in its security section: passwords are stored as a one-way hash with a modern algorithm (Argon2id or bcrypt), never as plain text [7].

11.5.2 Why a *fast* hash is the wrong one-way function

A general-purpose hash such as SHA-256 is one-way, but it is designed to be fast, and speed is exactly what an attacker wants. With a stolen table of hashes, the attacker does not need to invert anything: they simply guess, hash each guess, and look for a match. If a password is drawn from a set of N equally likely possibilities and the attacker can compute R hashes per second, the time to try them all is

$$T = \frac{N}{R} \tag{11.1}$$

and on average half of that finds a given password. The defender cannot change N, which the user chose, but can change R by choosing the function. That is the whole idea of a *password hashing function*: a deliberately slow one-way function with a tunable cost.

Table 11.1 shows what the choice does on one ordinary machine, the virtual server with a 2.1 GHz Xeon processor used to run this book's code, measured on a single core with Python 3.11.

Table 11.1 Measured speed of four ways to hash a password on one CPU core of the machine used for this book (September 2026). A real attacker is much faster with specialized hardware; the ratios are what matter.

Function (cost setting)	Time per hash	Hashes per second, R
SHA-256, salted, one pass	0.6 µs	about 1,670,000
PBKDF2-HMAC-SHA256, 100,000 iterations	31 ms	about 32
PBKDF2-HMAC-SHA256, 600,000 iterations	230 ms	about 4.3
Argon2id, 19 MiB, 2 passes, 1 lane	39 ms	about 26

EXAMPLE. A password of 8 lowercase letters has $N = 26^8 = 208\,827\,064\,576$ possibilities. With salted SHA-256 at $R \approx 1.67 \times 10^6$ per second, Equation (11.1) gives $T \approx 1.25 \times 10^5$ seconds, about 35 hours on one core. With PBKDF2 at 600,000 iterations, $R \approx 4.35$, so $T \approx 4.8 \times 10^{10}$ seconds, about 1,500 years on the same core. The password did not change; only the cost of each guess did.

The numbers in the example flatter the defender, because an attacker does not use one Python process on one core. Graphics processors and dedicated hardware compute SHA-256 and

PBKDF2 many thousands of times faster. What they are much less good at is computing functions that also need a lot of *memory* per guess, and that is where Argon2 comes in.

11.5.3 Salt

Before any of this, every password gets a **salt**: a random value, different for every stored password, that is hashed together with it and stored next to the result. Without a salt, two users with the same password have the same hash, and an attacker can precompute a table of hashes of common passwords once and look up every user at once. With a salt, every guess must be computed separately for every user. NIST's current guidelines require that passwords "SHALL be salted and hashed using a suitable password hashing scheme", with a salt of at least 32 bits [8]; the lab uses 128 bits (16 bytes).

11.5.4 Argon2id

Argon2 won the Password Hashing Competition in 2015 [9] and was standardized in RFC 9106 in September 2021 [10]. It is *memory-hard*: computing one hash requires filling and repeatedly reading a block of memory whose size the defender chooses, so an attacker who wants to run many guesses in parallel needs that much memory for each of them. It has three variants; **Argon2id**, a hybrid that resists both side-channel attacks and attacks by specialized hardware, is the one RFC 9106 recommends as the default [10].

Argon2id has three cost parameters: memory m, number of passes t, and parallelism p. The OWASP Password Storage Cheat Sheet lists five configurations it considers equally strong, from $m = 47104$ KiB (46 MiB) with $t = 1$ down to $m = 7168$ KiB (7 MiB) with $t = 5$, all with $p = 1$; a common middle choice is $m = 19456$ KiB (19 MiB), $t = 2$, $p = 1$ [11]. The same cheat sheet recommends 600,000 iterations for PBKDF2-HMAC-SHA256 when Argon2id is not available [11]. PBKDF2 itself is defined in RFC 8018 [12]; it is not memory-hard, but it is in the standard library of both Python (`hashlib.pbkdf2_hmac`) and Java (`PBKDF2WithHmacSHA256`), which is why lab G7 uses it by default and offers Argon2id as an option [13], [14].

11.5.5 A stored hash names its own recipe

Figure 11.2 shows the whole path from a typed password to a stored string, and back at login.

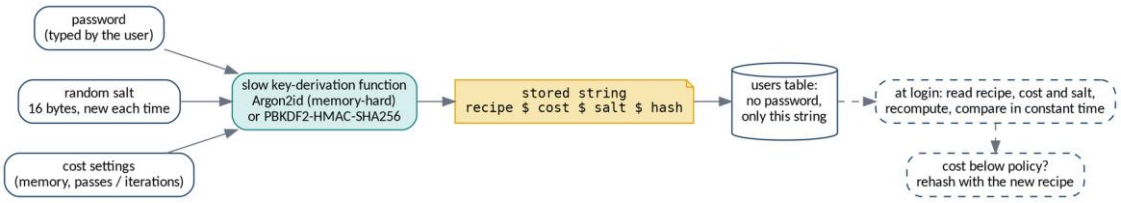


Figure 11.2 Storing a password. A random salt and the cost settings go into a slow key-derivation function; the stored string records the recipe, the cost, the salt and the result, so that login can recompute it and an outdated recipe can be upgraded.

The stored string is not just a hash. It is a small record: the name of the algorithm, its cost settings, the salt and the result, for example

```
pbkdf2_sha256$600000$<salt, base64>$<hash, base64>
$argon2id$v=19$m=19456,t=2,p=1$<salt, base64>$<hash, base64>
```

Because every stored hash names its own recipe, the recipe can change without a flag day. When the policy moves to a stronger setting, old hashes still verify with their old settings, and at the next successful login, the only moment the system ever sees the plain password again, the system recomputes the hash with the new recipe and replaces the old string. This is the *upgrade on login*

pattern, and Profitina uses it: legacy hashes are upgraded to Argon2id when their owners log in [1]. Lab G7 demonstrates it by moving from 100,000 to 600,000 PBKDF2 iterations (or to Argon2id) in the middle of its demo.

Two smaller details complete the picture. The comparison of the recomputed hash with the stored one must take the same time whatever the input, so that the time of a failed login reveals nothing about how many bytes matched; Python's `hmac.compare_digest` and Java's `MessageDigest.isEqual` do this. And the length limit matters: NIST's SP 800-63B-4, published in August 2025, requires passwords of at least 15 characters when a password is the only factor, and at least 8 when it is used with another factor, and it caps consecutive failed attempts at 100 per account [8]. Lab G7 uses the 15-character minimum.

11.6 Consent That Means Something

11.6.1 From Pasal 22 to a checkbox

Section 11.2.4 quoted the law: consent is given in writing or in a recorded form; the request must be distinguishable, understandable, and in plain language; consent that fails these conditions is void [2]. Translated into a registration form, that means four things:

1. **A separate, explicit act.** A checkbox the user ticks, not a sentence buried in the terms that "by registering you agree".
2. **Never pre-ticked.** A box that arrives ticked records the designer's choice, not the user's. Profitina's Privacy Policy says so directly: consent checkboxes are not pre-checked [7].
3. **Informed.** A short notice in both languages next to the box, with a link to the full policy, covering what Pasal 21 lists.
4. **Recorded.** A row that says who consented, to which version of the policy, and when.

The fourth point is what turns consent from an interface detail into evidence. Profitina logs every consent with the policy version and the time, as an audit trail [1]. When the Privacy Policy changes, as it did when version 1.15 took effect on 9 September 2026 [7], the log shows exactly which version each person accepted.

11.6.2 Designing the consent log

A consent log is a list of events, not a single field on the user record. The difference matters. A boolean `consented = true` says nothing about *what* was agreed, *when*, or whether it was later withdrawn; a list of events answers all three and never loses history. Lab G7's log has one row per event with five fields: the subject, the event (`given` or `withdrawn`), the policy version, the time, and a hash of the IP address from which it was given.

Why a *hash* of the IP address, and why a *keyed* one? An IP address is itself personal data. Keeping it in clear text in a log whose purpose is to prove consent would create a new privacy risk in the act of documenting privacy. A plain SHA-256 of an IPv4 address does not help much, because there are only about four billion addresses and all of them can be hashed in minutes. A keyed hash (HMAC) with a secret kept outside the database cannot be reversed that way by someone who has only the database. The lab keeps the first 16 hexadecimal digits, enough to tell events apart, and erases even that on account deletion.

11.6.3 Withdrawal

Pasal 9 gives the right to withdraw consent. Withdrawal must be as easy as giving, and it must be recorded too. In lab G7, as on Profitina, the definitive withdrawal is account deletion: the log gains a `withdrawn` event, and the earlier `given` event stays as proof that processing until then was lawful, but both are re-labeled with an anonymous tombstone identifier and stripped of the IP hash, so that the proof no longer identifies the person (Section 11.8.4).

Consent is also granular where the product is. Profitina's profile page asks for consent per item, one for the photo and one for each social link, and every one of them can be withdrawn on its own [1]. The general rule: a consent covers one purpose, and a person should never have to accept one thing to get another that has nothing to do with it.

11.6.4 Children

Pasal 25 requires the consent of a parent or guardian for processing children's data [2]. Profitina is not directed at children, and its policy says so, with parental consent where the law requires it [7]. The engineering lesson is that an age question is itself personal data; a service not aimed at children should say so plainly rather than collect birth dates it does not need.

11.7 Login Rate Limiting

11.7.1 The attacks

Two attacks aim at the login form. **Brute force** tries many passwords against one account. **Credential stuffing** tries username-password pairs leaked from *other* sites, betting that people reuse passwords; each account is tried only a few times, but millions of accounts are tried [15]. A slow hash (Section 11.5) makes each attempt cost the defender CPU time too, which is a third reason to limit attempts: without a limit, a flood of login requests becomes a denial-of-service attack on the server's processors.

Pasal 39 obliges the controller to prevent unauthorized access [2]; rate limiting is one of the standard technical measures that meet it. Profitina's Privacy Policy describes its own measure in these terms: login rate limiting against brute force and credential stuffing, with limits per account identity and, for anonymous visitors, per IP address, kept only in server memory and never written to the database [7]. The method is the same whatever the numbers, and this chapter teaches it with the lab's own, deliberately small ones.

11.7.2 Fixed windows and sliding windows

The simplest limiter counts failures per clock minute and resets the count at the start of each minute: a *fixed window*. Its weakness is the boundary. With a limit of 3 per minute, an attacker can fail 3 times at 09:00:59 and 3 more times at 09:01:00: six attempts in two seconds, twice the intended rate.

A **sliding window** has no boundary. It remembers the time of each recent failure for a key, and before each attempt it forgets the failures older than the window. Let $F_k(t)$ be the set of times of failures for key k within the window w before the current time t, and let n be the maximum number of failures allowed:

$$F_k(t) = \{\, s \in \text{failures}(k) : t - w < s \le t \,\}, \qquad \text{blocked}(k,t) \Leftrightarrow |F_k(t)| \ge n \tag{11.2}$$

When a key is blocked, it may try again as soon as its oldest remembered failure leaves the window:

$$\text{retry\_after}(k, t) = \min F_k(t) + w - t \tag{11.3}$$

Figure 11.3 follows the lab's demo with $n = 3$ and $w = 60$ seconds.

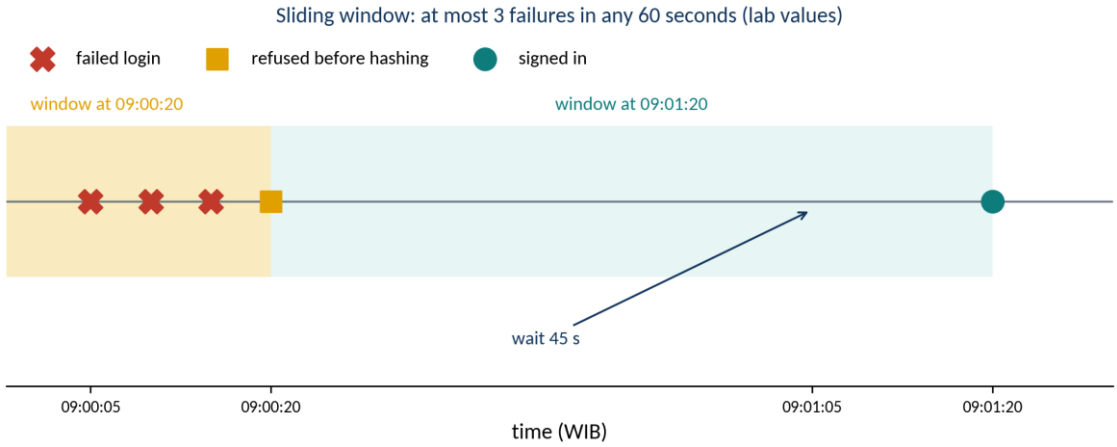


Figure 11.3 A sliding-window limiter with the lab's values. Three failures at 09:00:05, 09:00:10 and 09:00:15 block the fourth attempt at 09:00:20, even with the right password; the block lifts when the first failure leaves the window at 09:01:05.

> **EXAMPLE.** Failures are recorded at $s = 5{,}10{,}15$ seconds after 09:00:00. At $t = 20$, $F_k(20) = \{5{,}10{,}15\}$ has 3 elements, so by Equation (11.2) the key is blocked, and by Equation (11.3) it must wait $5 + 60 - 20 = 45$ seconds, exactly what the demo prints. At $t = 80$, the window is $(20{,}80]$, $F_k(80) = \emptyset$, and the login proceeds.

Storing the failure times in a double-ended queue makes each check cheap: new failures go on the right, expired ones leave on the left, and each time enters and leaves the queue once, so the work per attempt is amortized $O(1)$, with at most n times kept per key.

11.7.3 Four details that matter

Two keys, not one. Limiting only per account lets a credential-stuffing attack spread its guesses over many accounts, a few each; limiting only per IP address lets a distributed attack use many addresses against one account. The lab checks both, an `acct:` key for the email address and an `ip:` key for the (keyed-hashed) address, and refuses if either is blocked.

Refuse before hashing. A blocked attempt is rejected *before* the password is verified. Otherwise the limiter would still let an attacker burn the server's CPU with slow hashes, and a correct password would still produce a different response time than a wrong one.

Say the same thing. "Wrong email or password" is the only failure message, for an unknown email and a wrong password alike, so that the login form does not tell an attacker which email addresses have accounts. For the same reason the lab verifies a dummy hash when the email is unknown, so that both cases take about the same time.

Keep it in memory. A limiter's records are short-lived by nature. Keeping them in memory, as Profitina does [7], means they vanish when they are no longer useful, and they never become one more table of personal data in backups.

11.8 Rights That Are Actually Reachable

11.8.1 From articles to buttons

A right is reachable when an ordinary person can exercise it, without knowing the law, without writing to anyone, and without waiting. **Table 11.2** maps the rights of Section 11.2.5 to what Profitina ships [1], [7].

Table 11.2 Data-subject rights under UU PDP and the self-service feature, or channel, through which a Profitina user exercises each one.

Right (UU PDP)	How a Profitina user exercises it
Information (Pasal 5)	Privacy Policy and Terms as bilingual HTML pages at `app.profitina.com/legal/`, linked from every page
Correction (Pasal 6)	Change email or password in the *Akun & Data Pribadi* panel, confirmed with the password
Access (Pasal 7)	Download all account data as JSON from the same panel
Erasure (Pasal 8)	Delete the account permanently from the panel, confirmed with the password
Withdrawal of consent (Pasal 9)	Per-item profile consents can be switched off; deleting the account withdraws the rest
Objection to automated decisions (Pasal 10)	Contact info@profitina.com; product decisions about users are made by people
Portability (Pasal 13)	The JSON export is structured, common and machine-readable

Figure 11.4 shows the same rights in the life of an account, from collection to erasure, together with the obligations that surround them.

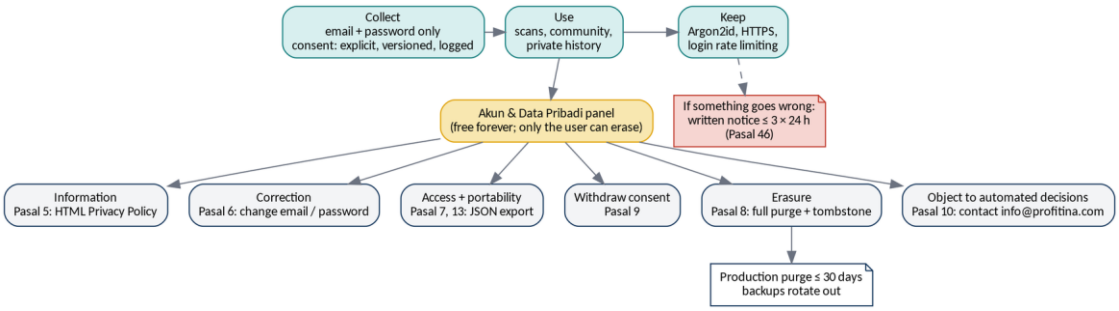


Figure 11.4 The life of personal data at Profitina: minimal collection, protected use, rights exercised from one panel, erasure followed by the production purge, and the breach-notification duty that stands behind it all.

Three commitments frame the panel. The rights, together with account safety and recovery, are **free forever**; they are never a Pro feature [1]. **Only the user can trigger erasure**: an administrator never deletes an account "on the user's behalf", because a deletion that someone else can start is also a deletion that someone else can start by mistake or in bad faith [7]. And every destructive action is **confirmed with the password**, so that a session left open on a shared computer cannot be used to erase or redirect an account.

11.8.2 An honest correction

Profitina's own history contains a lesson that is worth more than any table. The mechanisms for export, change and deletion existed in the application's code before 19 August 2026. What did not exist was a button: the functions could be reached through the programming interface, but a user of the web interface had no way to find them. On that date a dedicated panel, *Akun & Data Pribadi*,

shipped in the *Pantau* (Monitor) tab, and only from then on was it true to call the rights self-service [1].

The team's own record says it plainly, and the lesson generalizes. An engineer who looks at the code sees that the right is implemented. A user who looks at the screen sees that it is not there. The law is written for the user. A feature that the intended person cannot find does not exist for that person, and a test suite that calls the function directly will never notice. The remedy is an end-to-end test that starts where the user starts, at the screen, the kind of browser-level check, with Playwright driving Chromium, that Profitina's test suite uses.

11.8.3 Export: everything, and only mine

An export for the right of access and portability has two failure modes, and a good test checks both. It can leave something out: a table added later that the export function was never told about. And it can include too much: another user's reply, or the internal password hash, which is not the user's data in any useful sense and is a security risk in a downloaded file. Lab G7's test checks that Ayu's export contains her email address, her consent events and her post, and contains neither Budi's post nor any password hash.

11.8.4 Erasure: purge, and leave a tombstone only where others need one

Erasure is the hardest right to implement correctly, because personal data spreads. An account's email address is in the users table; its posts are in the posts table; its consent events are in the consent log; its sessions are in the session store; its failed-login records may be in the limiter; its identifier may be referenced by other people's replies. A deletion that removes the user row and nothing else leaves most of the personal data behind.

Profitina's erasure purges personal data across all modules and leaves an anonymous tombstone [1]. The tombstone solves a real dilemma. Suppose Ayu asked a question and Budi answered it. If Ayu's question disappears, Budi's answer points at nothing and the thread breaks; if it stays with her name, she has not been erased. The tombstone keeps an empty shell, with the writer removed and the text replaced by `[deleted]`, *only* where someone else's content points to it, and deletes everything else outright. **Figure 11.5** shows the rule applied to the lab's three tables.

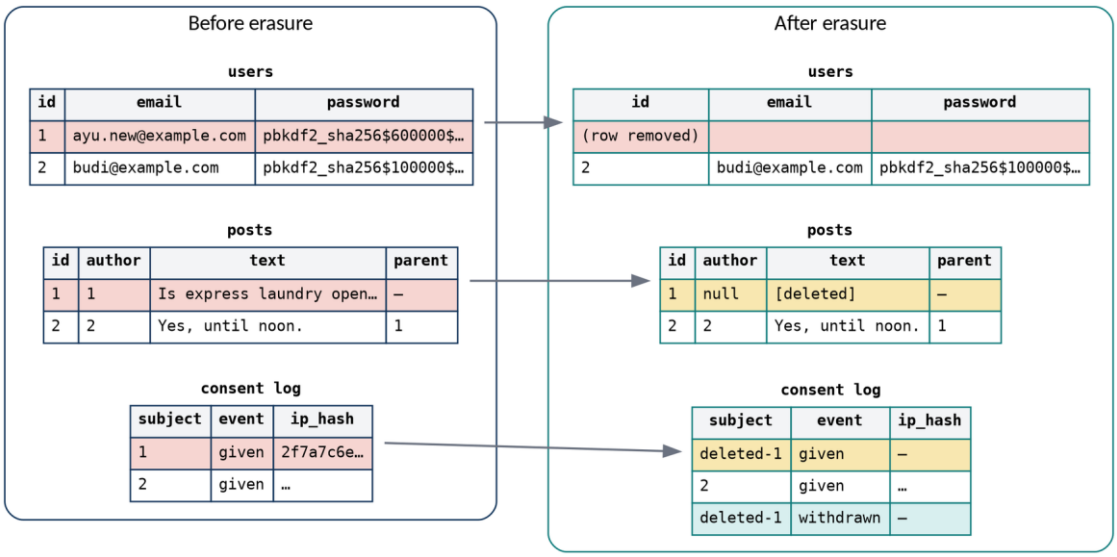


Figure 11.5 Erasure in lab G7. The user row goes; the post that another user replied to becomes an anonymous shell; the consent events keep only an anonymous identifier, a version and a time, and a `withdrawn` event is added.

On Profitina, production data is deleted or anonymized within 30 days of an account's deletion, and backups age out with their 90-day rotation cycle; error logs are kept at most 90 days [7]. The limits are part of the promise: erasure from a live database is immediate, but a system with backups must also say when the backups forget.

11.8.5 When something goes wrong

Pasal 46 requires written notice to affected users and to the supervisory agency within 3×24 hours of a failure of personal-data protection [2], and Profitina's policy repeats the commitment [7]. Seventy-two hours is short. Profitina's runbook divides it in advance: the first two hours for containment, hours 2 to 12 for assessing what was affected, and hours 12 to 48 for notification, with templates for both letters prepared before they are ever needed [1]. The engineering point is the same as for the consent log: the work that makes a deadline achievable is done long before the deadline starts.

11.9 HTML Legal Pages and a Single Source of Truth

A Privacy Policy is the concrete form of the right to information (Pasal 5), and the notice Pasal 21 requires before consent. It is also a document that a user must be able to read on a phone, in their own language, and link to. That is why Profitina publishes its legal texts as ordinary HTML pages, `/legal/privacy` and `/legal/terms`, rather than as downloadable PDF files [1], [7].

HTML serves each of the requirements better:

- **Readable everywhere.** An HTML page reflows on a small screen and works with screen readers and browser translation; a PDF of an A4 page does neither well.
- **Linkable to the paragraph.** Each section can have an anchor, so that the consent notice can link to exactly the paragraph it summarizes.
- **Versioned.** The page states its version and the date it takes effect. At the time of writing, the Privacy Policy is version 1.15, effective 9 September 2026, and the Terms and Conditions are version 1.1, effective 6 August 2026 [7], [16]. The application knows the current policy version, so the consent log can record it with every consent.
- **Bilingual in one place.** The Indonesian and English texts live together, in the same application, and change together.
- **One home.** The marketing site summarizes and links; it does not keep a copy (Section 10.2.3).

The policy itself follows the structure the law asks for: what is collected (account data, content, technical data, optional profile items, messages, scan history, transactions), why and on which basis, how long it is kept, who processes it on the company's behalf and under which safeguards across borders, which cookies are used (functional ones only: session, language and privacy-notice acknowledgment, no advertising cookies), which rights the user has and how to exercise them, and how to reach the company: info@profitina.com [7].

11.10 Lab G7: Consent Log, Export and Erasure

11.10.1 What we will build

Lab G7 is a small application that holds user accounts, posts and a consent log in memory and implements every measure of Sections 11.5 to 11.8: salted slow password hashes with upgrade on login, explicit and versioned consent, a sliding-window login limiter on two keys, export as

JSON, email and password change with password confirmation, and erasure with a tombstone. A demo program walks through a scenario with two users, Ayu and Budi, and a test suite proves ten properties, the most important being that after an erasure nothing personal about the erased user remains in any table.

The application keeps its tables in memory and uses a simulated clock, so that every run prints exactly the same thing and the reader can follow the times by hand. Nothing in it depends on a database; Exercise 11.12 suggests moving the tables into SQLite, which Python's standard library includes. The IP addresses in the demo, `203.0.113.7` and `198.51.100.23`, come from the blocks reserved for documentation, which never belong to real machines [17].

Everything is standard library only. The Python files need Python 3.11 or newer (written for 3.14); `passwords.py` can also use Argon2id if the optional `argon2-cffi` package is installed [18]. The Java files compile with JDK 21 and run on JDK 25:

```
Chapter11/
  python/  passwords.py  rate_limit.py  privacy_app.py  demo.py
           test_privacy_app.py
  java/    Pbkdf2Hasher.java  SlidingWindowLimiter.java  PrivacyApp.java
           Json.java  Demo.java  PrivacyAppTest.java
```

The chapter's `README.md` gives the commands for Windows, macOS and Linux, in a terminal and in VS Code.

11.10.2 Password hashing

Listing 11.1 contains two hashers with the same four methods: `recognizes` (is this stored string mine?), `hash`, `verify`, and `needs_rehash` (is this stored string weaker than my current policy?).

Listing 11.1 `passwords.py`: PBKDF2-HMAC-SHA256 from the standard library, and Argon2id through the optional `argon2-cffi` package; each stored hash names its recipe.

```
"""Password hashing for the privacy lab: PBKDF2 from the standard library,
and Argon2id when the optional argon2-cffi package is installed.

A stored hash names its own recipe, so the recipe can change later:
    pbkdf2_sha256$600000$<salt, base64>$<hash, base64>
"""
import base64
import hashlib
import hmac
import secrets

OWASP_PBKDF2_SHA256 = 600_000       # OWASP Password Storage Cheat Sheet


class Pbkdf2Hasher:
    def __init__(self, iterations=OWASP_PBKDF2_SHA256):
        self.iterations = iterations

    def recognizes(self, stored):
        return stored.startswith("pbkdf2_sha256$")

    def hash(self, password, salt=None):
        salt = salt if salt is not None else secrets.token_bytes(16)
        key = hashlib.pbkdf2_hmac("sha256", password.encode("utf-8"), salt,
                                  self.iterations, dklen=32)
        return "pbkdf2_sha256$%d$%s$%s" % (
            self.iterations, base64.b64encode(salt).decode(),
            base64.b64encode(key).decode())

    def verify(self, stored, password):
        try:
            scheme, iters, salt, key = stored.split("$")
        except ValueError:
            return False
        if scheme != "pbkdf2_sha256":
            return False
        again = hashlib.pbkdf2_hmac("sha256", password.encode("utf-8"),
                                    base64.b64decode(salt), int(iters),
                                    dklen=32)
        # compare_digest takes the same time wherever the first difference is
        return hmac.compare_digest(again, base64.b64decode(key))

    def needs_rehash(self, stored):
        parts = stored.split("$")
        return parts[0] != "pbkdf2_sha256" or int(parts[1]) < self.iterations
```

Listing 11.1 (Continued) *passwords.py: PBKDF2-HMAC-SHA256 from the standard library, and Argon2id through the optional argon2-cffi package; each stored hash names its recipe.*

```
class Argon2Hasher:
    """Argon2id with an OWASP configuration (19 MiB, 2 passes, 1 lane)."""

    def __init__(self):
        from argon2 import PasswordHasher          # pip install argon2-cffi
        self.ph = PasswordHasher(time_cost=2, memory_cost=19456,
                                 parallelism=1)

    def recognizes(self, stored):
        return stored.startswith("$argon2")

    def hash(self, password):
        return self.ph.hash(password)

    def verify(self, stored, password):
        from argon2.exceptions import VerificationError, InvalidHashError
        try:
            return self.ph.verify(stored, password)
        except (VerificationError, InvalidHashError):
            return False

    def needs_rehash(self, stored):
        return not stored.startswith("$argon2id$") or \
            self.ph.check_needs_rehash(stored)
```

`Pbkdf2Hasher.hash` draws 16 random bytes from `secrets`, the module meant for security-sensitive randomness, derives 32 bytes with `hashlib.pbkdf2_hmac`, and writes the recipe string of Section 11.5.5. `verify` splits the string, recomputes with the stored iteration count and salt, and compares with `hmac.compare_digest`. `needs_rehash` says yes when the stored string is not PBKDF2 at all or has fewer iterations than the current policy. `Argon2Hasher` wraps `argon2-cffi`'s `PasswordHasher` with the OWASP configuration of 19 MiB, 2 passes and 1 lane, and imports the package only when it is created, so that the rest of the lab works without it.

Listing 11.2 is the Java hasher. The JDK's `SecretKeyFactory` provides PBKDF2 under the standard name `PBKDF2WithHmacSHA256` [14]; `SecureRandom` provides the salt and `MessageDigest.isEqual` the constant-time comparison. For the same password, salt and iteration count, it produces exactly the same string as the Python version, which one of the tests checks.

Listing 11.2 `Pbkdf2Hasher.java`: the same PBKDF2 recipe with the JDK's `SecretKeyFactory`.

```
import java.security.MessageDigest;
import java.security.SecureRandom;
import java.util.Base64;
import javax.crypto.SecretKeyFactory;
import javax.crypto.spec.PBEKeySpec;

/** PBKDF2-HMAC-SHA256 from the JDK. A stored hash names its own recipe:
 *  pbkdf2_sha256$600000$<salt, base64>$<hash, base64>
 *  (Argon2id needs a library such as Bouncy Castle; see the README.) */
public class Pbkdf2Hasher {
    static final int OWASP_PBKDF2_SHA256 = 600_000;   // OWASP cheat sheet
    private static final SecureRandom RANDOM = new SecureRandom();
    final int iterations;

    Pbkdf2Hasher() {
        this(OWASP_PBKDF2_SHA256);
    }

    Pbkdf2Hasher(int iterations) {
        this.iterations = iterations;
    }

    boolean recognizes(String stored) {
        return stored.startsWith("pbkdf2_sha256$");
    }

    static byte[] derive(String password, byte[] salt, int iterations) {
        try {
            PBEKeySpec spec = new PBEKeySpec(password.toCharArray(), salt,
                                             iterations, 256);
            return SecretKeyFactory.getInstance("PBKDF2WithHmacSHA256")
                                   .generateSecret(spec).getEncoded();
        } catch (Exception e) {
            throw new IllegalStateException(e);
        }
    }

    String hash(String password) {
        byte[] salt = new byte[16];
        RANDOM.nextBytes(salt);
        return hash(password, salt);
    }
```

Listing 11.2 (Continued) *Pbkdf2Hasher.java: the same PBKDF2 recipe with the JDK's SecretKeyFactory.*

```
    String hash(String password, byte[] salt) {
        Base64.Encoder b64 = Base64.getEncoder();
        return "pbkdf2_sha256$" + iterations + "$" + b64.encodeToString(salt)
                + "$" + b64.encodeToString(derive(password, salt, iterations));
    }

    boolean verify(String stored, String password) {
        String[] p = stored.split("\\$");
        if (p.length != 4 || !p[0].equals("pbkdf2_sha256")) return false;
        Base64.Decoder b64 = Base64.getDecoder();
        byte[] again = derive(password, b64.decode(p[2]), Integer.parseInt(p[1]));
        // isEqual takes the same time wherever the first difference is
        return MessageDigest.isEqual(again, b64.decode(p[3]));
    }

    boolean needsRehash(String stored) {
        String[] p = stored.split("\\$");
        return !p[0].equals("pbkdf2_sha256") || Integer.parseInt(p[1]) < iterations;
    }
}
```

11.10.3 The sliding-window limiter

Listing 11.3 is Equation (11.2) and Equation (11.3) in code. Each key maps to a `deque` of failure times; `_trim` removes the times that have left the window; `retry_after` returns 0 or the waiting time; `record_failure` appends; `forget` drops a key after a successful login or an erasure.

Listing 11.3 `rate_limit.py`: a sliding-window limiter for failed logins, with a clock passed in so that tests can control time.

```
"""A sliding-window limiter for failed logins.

Each key (an account, or an IP address for anonymous traffic) keeps the
times of its recent failures. A key is blocked while it has max_failures
failures inside the last window_seconds. The numbers are the reader's
choice; the demo uses small ones so that the effect is easy to see.
"""
from collections import defaultdict, deque


class SlidingWindowLimiter:
    def __init__(self, max_failures, window_seconds, clock):
        self.max_failures = max_failures
        self.window = window_seconds
        self.clock = clock                      # a function returning seconds
        self.failures = defaultdict(deque)      # key -> times of failures

    def _trim(self, key):
        times, now = self.failures[key], self.clock()
        while times and times[0] <= now - self.window:
            times.popleft()                     # forget failures that aged out
        return times

    def retry_after(self, key):
        """0 if the key may try now, else seconds until it may."""
        times = self._trim(key)
        if len(times) < self.max_failures:
            return 0
        return int(times[0] + self.window - self.clock())

    def record_failure(self, key):
        self._trim(key).append(self.clock())

    def forget(self, key):
        self.failures.pop(key, None)
```

The clock is a function passed to the constructor rather than a call to the system clock inside the class. That one decision is what makes the limiter testable: the demo and the tests move time forward by assigning a number, instead of sleeping for a minute. **Listing 11.4** is the Java version, with a `LongSupplier` as the clock and an `ArrayDeque` per key.

Listing 11.4 `SlidingWindowLimiter.java`: the same limiter in Java.

```
import java.util.ArrayDeque;
import java.util.Deque;
import java.util.HashMap;
import java.util.Map;
import java.util.function.LongSupplier;

/** A sliding-window limiter for failed logins. A key (an account, or an IP
 *  address for anonymous traffic) is blocked while it has maxFailures
 *  failures inside the last windowSeconds. The numbers are the reader's. */
public class SlidingWindowLimiter {
    final int maxFailures;
    final long window;
    final LongSupplier clock;                       // returns seconds
    final Map<String, Deque<Long>> failures = new HashMap<>();

    SlidingWindowLimiter(int maxFailures, long windowSeconds, LongSupplier clock) {
        this.maxFailures = maxFailures;
        this.window = windowSeconds;
        this.clock = clock;
    }

    private Deque<Long> trim(String key) {
        Deque<Long> times = failures.computeIfAbsent(key, k -> new ArrayDeque<>());
        long now = clock.getAsLong();
        while (!times.isEmpty() && times.peekFirst() <= now - window) {
            times.pollFirst();                      // failures that aged out
        }
        return times;
    }

    /** 0 if the key may try now, else seconds until it may. */
    long retryAfter(String key) {
        Deque<Long> times = trim(key);
        if (times.size() < maxFailures) return 0;
        return times.peekFirst() + window - clock.getAsLong();
    }

    void recordFailure(String key) {
        trim(key).addLast(clock.getAsLong());
    }

    void forget(String key) {
        failures.remove(key);
    }
}
```

11.10.4 The application

Listing 11.5 is the application. Its tables are ordinary dictionaries and lists, and every method corresponds to one section of this chapter.

Listing 11.5 `privacy_app.py`: registration with explicit consent, rate-limited login with hash upgrade, export, correction, and erasure with a tombstone.

```
"""G7: a privacy-by-design mini app (in memory, standard library only).

Explicit consent with a version and a time; passwords stored as slow salted
hashes; failed logins rate-limited; self-service export, correction and
erasure. Erasure purges every personal field and leaves an anonymous
tombstone only where other people's content still points to it.
"""
import hashlib
import hmac
import json
import secrets
from datetime import datetime, timedelta, timezone

from passwords import Pbkdf2Hasher
from rate_limit import SlidingWindowLimiter

WIB = timezone(timedelta(hours=7))
MIN_PASSWORD = 15          # NIST SP 800-63B-4, password as the only factor
GENERIC = "wrong email or password"


class PrivacyError(Exception):
    pass


class PrivacyApp:
    def __init__(self, clock, secret, hasher=None, max_failures=3,
                 window_seconds=60, policy_version="1.0"):
        self.clock = clock
        self.secret = secret           # keys the IP hashes; not in the tables
        self.hasher = hasher or Pbkdf2Hasher()
        self.hashers = [self.hasher]   # older recipes still readable
        self.limiter = SlidingWindowLimiter(max_failures, window_seconds,
                                            clock)
        self.policy_version = policy_version
        self.users = {}        # id -> email, password (hash), created
        self.consents = []     # subject, event, version, at, ip_hash
        self.posts = {}        # id -> author, text, parent, at
        self.sessions = {}     # token -> user id
        self.tombstones = []   # tombstone, at
        self.next_user = self.next_post = self.next_tomb = 1
        self.dummy = self.hasher.hash(secrets.token_hex(16))
```

Listing 11.5 (Continued) *privacy_app.py: registration with explicit consent, rate-limited login with hash upgrade, export, correction, and erasure with a tombstone.*

```
    # ---- helpers ------------------------------------------------------
    def stamp(self):
        return datetime.fromtimestamp(self.clock(), WIB).isoformat()

    def ip_hash(self, ip):
        mac = hmac.new(self.secret, ip.encode("utf-8"), hashlib.sha256)
        return mac.hexdigest()[:16]

    def find(self, email):
        for uid, user in self.users.items():
            if user["email"] == email:
                return uid
        return None

    def verify(self, stored, password):
        for h in self.hashers:
            if h.recognizes(stored):
                return h.verify(stored, password)
        return False

    def user_of(self, token):
        if token not in self.sessions:
            raise PrivacyError("not logged in")
        return self.sessions[token]

    def check_password(self, uid, password):
        if not self.verify(self.users[uid]["password"], password):
            raise PrivacyError("password confirmation failed")

    def use_hasher(self, hasher):
        """Adopt a new recipe; old hashes are upgraded at the next login."""
        self.hasher = hasher
        self.hashers.insert(0, hasher)

    # ---- registration and login ---------------------------------------
    def register(self, email, password, consent, ip):
        email = email.strip().lower()
        if "@" not in email:
            raise PrivacyError("not an email address")
        if len(password) < MIN_PASSWORD:
            raise PrivacyError("password needs at least %d characters"
                               % MIN_PASSWORD)
        if consent is not True:                 # never assumed, never default
            raise PrivacyError("consent is required")
        if self.find(email) is not None:
            raise PrivacyError("email already registered")
```

Listing 11.5 (Continued) *privacy_app.py: registration with explicit consent, rate-limited login with hash upgrade, export, correction, and erasure with a tombstone.*

```
        uid, self.next_user = self.next_user, self.next_user + 1
        self.users[uid] = {"email": email, "created": self.stamp(),
                           "password": self.hasher.hash(password)}
        self.consents.append({"subject": uid, "event": "given",
                              "version": self.policy_version,
                              "at": self.stamp(), "ip_hash": self.ip_hash(ip)})
        return uid

    def login(self, email, password, ip):
        email = email.strip().lower()
        keys = ["acct:" + email, "ip:" + self.ip_hash(ip)]
        wait = max(self.limiter.retry_after(k) for k in keys)
        if wait:                            # refuse before touching the hash
            raise PrivacyError("too many attempts, wait %d s" % wait)
        uid = self.find(email)
        stored = self.users[uid]["password"] if uid else self.dummy
        if not self.verify(stored, password) or uid is None:
            for k in keys:
                self.limiter.record_failure(k)
            raise PrivacyError(GENERIC)          # same words either way
        self.limiter.forget(keys[0])
        if self.hasher.needs_rehash(stored):
            self.users[uid]["password"] = self.hasher.hash(password)
        token = secrets.token_hex(16)
        self.sessions[token] = uid
        return token

    # ---- ordinary use ------------------------------------------------------
    def post(self, token, text, parent=None):
        pid, self.next_post = self.next_post, self.next_post + 1
        self.posts[pid] = {"author": self.user_of(token), "text": text,
                           "parent": parent, "at": self.stamp()}
        return pid

    # ---- data-subject rights -----------------------------------------------
    def export(self, token):
        """Right of access and portability: everything held, as JSON."""
        uid = self.user_of(token)
        user = self.users[uid]
        data = {
            "exported_at": self.stamp(),
            "account": {"id": uid, "email": user["email"],
                        "created": user["created"]},
            "consents": [{k: c[k] for k in ("event", "version", "at",
                                            "ip_hash") if k in c}
                         for c in self.consents if c["subject"] == uid],
```

Listing 11.5 (Continued) *privacy_app.py: registration with explicit consent, rate-limited login with hash upgrade, export, correction, and erasure with a tombstone.*

```
            "posts": [{"id": pid, "text": p["text"], "parent": p["parent"],
                       "at": p["at"]}
                      for pid, p in self.posts.items() if p["author"] == uid],
        }
        return json.dumps(data, indent=2, ensure_ascii=False)

    def change_email(self, token, password, new_email):
        uid = self.user_of(token)
        self.check_password(uid, password)
        new_email = new_email.strip().lower()
        if "@" not in new_email or self.find(new_email) is not None:
            raise PrivacyError("that email cannot be used")
        self.users[uid]["email"] = new_email

    def change_password(self, token, old, new):
        uid = self.user_of(token)
        self.check_password(uid, old)
        if len(new) < MIN_PASSWORD:
            raise PrivacyError("password needs at least %d characters"
                               % MIN_PASSWORD)
        self.users[uid]["password"] = self.hasher.hash(new)
        for t in [t for t, u in self.sessions.items() if u == uid]:
            if t != token:
                del self.sessions[t]            # sign out every other device

    def delete_account(self, token, password):
        """Right to erasure. Only the account holder can start it."""
        uid = self.user_of(token)
        self.check_password(uid, password)
        tomb = "deleted-%d" % self.next_tomb
        self.next_tomb += 1
        mine = {pid for pid, p in self.posts.items() if p["author"] == uid}
        needed = {p["parent"] for p in self.posts.values()
                  if p["parent"] in mine and p["author"] != uid}
        for pid in mine:
            if pid in needed:           # others replied: keep an empty shell
                self.posts[pid] = {"author": None, "text": "[deleted]",
                                   "parent": self.posts[pid]["parent"],
                                   "at": self.posts[pid]["at"]}
            else:
                del self.posts[pid]
        for c in self.consents:
            if c["subject"] == uid:
                c["subject"] = tomb     # proof of consent, without the person
                c.pop("ip_hash", None)
        self.consents.append({"subject": tomb, "event": "withdrawn",
```

Listing 11.5 (Continued) *privacy_app.py: registration with explicit consent, rate-limited login with hash upgrade, export, correction, and erasure with a tombstone.*

```
                              "version": self.policy_version,
                              "at": self.stamp()})
        for t in [t for t, u in self.sessions.items() if u == uid]:
            del self.sessions[t]
        self.limiter.forget("acct:" + self.users[uid]["email"])
        del self.users[uid]
        self.tombstones.append({"tombstone": tomb, "at": self.stamp()})
        return tomb

    def dump(self):
        """Every stored table as one string, for the 'nothing left' test."""
        return json.dumps({"users": self.users, "consents": self.consents,
                           "posts": self.posts, "sessions": self.sessions,
                           "tombstones": self.tombstones})
```

Reading it method by method:

- `register` normalizes the email to lower case, checks that it looks like an address and that the password has at least 15 characters, and then insists that `consent is True`. Not “truthy”: the string `"yes"`, the number 1 and `None` are all refused, because consent must be an explicit act, and a program that accepts a default or a leftover value is exactly the pre-ticked box of Section 11.6.1. Only then does it create the user and append a `given` event with the policy version, the time and the keyed IP hash.
- `login` builds the two keys, asks the limiter first and refuses before touching any hash, verifies the password (against a dummy hash when the email is unknown), records a failure on both keys or forgets the account key on success, upgrades an outdated hash, and returns a random session token.
- `export` assembles the account, the consent events and the posts of the logged-in user, and nothing else, as indented JSON.
- `change_email` and `change_password` require the current password. A password change also signs out every other session of the user, because a password is often changed precisely when someone else may know the old one.
- `delete_account` requires the password, computes which of the user’s posts have replies by other people, keeps those as anonymous shells and deletes the rest, re-labels the consent events with a tombstone identifier and removes their IP hashes, appends a `withdrawn` event, removes every session and the limiter’s account key, deletes the user row, and records the tombstone.
- `dump` serializes every table into one string. It exists for the test that searches the whole store for traces of an erased user.

The Java application in **Listing 11.6** has the same structure, with small classes for users, consent events and posts, and a `PrivacyException` for refusals. It needs one helper that Python has built in: **Listing 11.7** writes maps and lists as JSON in the same two-space layout as Python’s `json.dumps(indent=2)`, so that the two exports are identical.

Listing 11.6 `PrivacyApp.java`: the same application in Java.

```
import java.nio.charset.StandardCharsets;
import java.security.SecureRandom;
import java.time.Instant;
import java.time.OffsetDateTime;
import java.time.ZoneOffset;
import java.time.format.DateTimeFormatter;
import java.util.*;
import java.util.function.LongSupplier;
import javax.crypto.Mac;
import javax.crypto.spec.SecretKeySpec;

/** G7: a privacy-by-design mini app (in memory, JDK only). Explicit consent
 *  with a version and a time; slow salted password hashes; rate-limited
 *  logins; self-service export, correction and erasure. Erasure purges every
 *  personal field and leaves an anonymous tombstone only where other
 *  people's content still points to it. */
public class PrivacyApp {
    static final int MIN_PASSWORD = 15;   // NIST SP 800-63B-4, single factor
    static final String GENERIC = "wrong email or password";
    static final ZoneOffset WIB = ZoneOffset.ofHours(7);
    private static final SecureRandom RANDOM = new SecureRandom();

    static class PrivacyException extends RuntimeException {
        private static final long serialVersionUID = 1L;

        PrivacyException(String message) {
            super(message);
        }
    }

    static class User {
        String email, password, created;
    }

    static class Consent {
        String subject, event, version, at, ipHash;   // ipHash may be null
    }

    static class Post {
        Integer author, parent;                       // author null: deleted
        String text, at;
    }
```

Listing 11.6 (Continued) *PrivacyApp.java: the same application in Java.*

```
    final LongSupplier clock;
    final byte[] secret;                // keys the IP hashes; not in the tables
    Pbkdf2Hasher hasher;
    final List<Pbkdf2Hasher> hashers = new ArrayList<>();
    final SlidingWindowLimiter limiter;
    String policyVersion = "1.0";               // the current policy
    final Map<Integer, User> users = new LinkedHashMap<>();
    final List<Consent> consents = new ArrayList<>();
    final Map<Integer, Post> posts = new LinkedHashMap<>();
    final Map<String, Integer> sessions = new LinkedHashMap<>();
    final List<String[]> tombstones = new ArrayList<>();    // {tombstone, at}
    int nextUser = 1, nextPost = 1, nextTomb = 1;
    final String dummy;

    PrivacyApp(LongSupplier clock, byte[] secret, Pbkdf2Hasher hasher,
               int maxFailures, long windowSeconds) {
        this.clock = clock;
        this.secret = secret;
        this.hasher = hasher;
        this.hashers.add(hasher);
        this.limiter = new SlidingWindowLimiter(maxFailures, windowSeconds, clock);
        this.dummy = hasher.hash(token());
    }

    // ---- helpers -----------------------------------------------------------
    String stamp() {
        return OffsetDateTime.ofInstant(Instant.ofEpochSecond(clock.getAsLong()),
                WIB).format(DateTimeFormatter.ISO_OFFSET_DATE_TIME);
    }

    String ipHash(String ip) {
        try {
            Mac mac = Mac.getInstance("HmacSHA256");
            mac.init(new SecretKeySpec(secret, "HmacSHA256"));
            byte[] d = mac.doFinal(ip.getBytes(StandardCharsets.UTF_8));
            return HexFormat.of().formatHex(d).substring(0, 16);
        } catch (Exception e) {
            throw new IllegalStateException(e);
        }
    }
```

Listing 11.6 (Continued) *PrivacyApp.java: the same application in Java.*

```
    static String token() {
        byte[] b = new byte[16];
        RANDOM.nextBytes(b);
        return HexFormat.of().formatHex(b);
    }

    Integer find(String email) {
        for (var e : users.entrySet()) {
            if (e.getValue().email.equals(email)) return e.getKey();
        }
        return null;
    }

    boolean verify(String stored, String password) {
        for (Pbkdf2Hasher h : hashers) {
            if (h.recognizes(stored)) return h.verify(stored, password);
        }
        return false;
    }

    int userOf(String token) {
        Integer uid = sessions.get(token);
        if (uid == null) throw new PrivacyException("not logged in");
        return uid;
    }

    void checkPassword(int uid, String password) {
        if (!verify(users.get(uid).password, password)) {
            throw new PrivacyException("password confirmation failed");
        }
    }

    /** Adopt a new recipe; old hashes are upgraded at the next login. */
    void useHasher(Pbkdf2Hasher h) {
        hasher = h;
        hashers.add(0, h);
    }
```

Listing 11.6 (Continued) *PrivacyApp.java: the same application in Java.*

```
    // ---- registration and login -------------------------------------------
    int register(String email, String password, Boolean consent, String ip) {
        email = email.strip().toLowerCase();
        if (!email.contains("@")) {
            throw new PrivacyException("not an email address");
        }
        if (password.length() < MIN_PASSWORD) {
            throw new PrivacyException("password needs at least " + MIN_PASSWORD
                    + " characters");
        }
        if (!Boolean.TRUE.equals(consent)) {       // never assumed or default
            throw new PrivacyException("consent is required");
        }
        if (find(email) != null) {
            throw new PrivacyException("email already registered");
        }
        int uid = nextUser++;
        User u = new User();
        u.email = email;
        u.created = stamp();
        u.password = hasher.hash(password);
        users.put(uid, u);
        Consent c = new Consent();
        c.subject = String.valueOf(uid);
        c.event = "given";
        c.version = policyVersion;
        c.at = stamp();
        c.ipHash = ipHash(ip);
        consents.add(c);
        return uid;
    }

    String login(String email, String password, String ip) {
        email = email.strip().toLowerCase();
        List<String> keys = List.of("acct:" + email, "ip:" + ipHash(ip));
        long wait = Math.max(limiter.retryAfter(keys.get(0)),
                             limiter.retryAfter(keys.get(1)));
        if (wait > 0) {                            // refuse before hashing
            throw new PrivacyException("too many attempts, wait " + wait + " s");
        }
        Integer uid = find(email);
        String stored = uid != null ? users.get(uid).password : dummy;
        if (!verify(stored, password) || uid == null) {
            for (String k : keys) limiter.recordFailure(k);
            throw new PrivacyException(GENERIC);   // same words either way
        }
```

Listing 11.6 (Continued) *PrivacyApp.java: the same application in Java.*

```
        limiter.forget(keys.get(0));
        if (hasher.needsRehash(stored)) {
            users.get(uid).password = hasher.hash(password);
        }
        String t = token();
        sessions.put(t, uid);
        return t;
    }

    // ---- ordinary use ------------------------------------------------------
    int post(String token, String text, Integer parent) {
        Post p = new Post();
        p.author = userOf(token);
        p.text = text;
        p.parent = parent;
        p.at = stamp();
        int pid = nextPost++;
        posts.put(pid, p);
        return pid;
    }

    // ---- data-subject rights -----------------------------------------------
    /** Right of access and portability: everything held, as JSON. */
    String export(String token) {
        int uid = userOf(token);
        User u = users.get(uid);
        Map<String, Object> data = new LinkedHashMap<>();
        data.put("exported_at", stamp());
        Map<String, Object> account = new LinkedHashMap<>();
        account.put("id", uid);
        account.put("email", u.email);
        account.put("created", u.created);
        data.put("account", account);
        List<Object> cs = new ArrayList<>();
        for (Consent c : consents) {
            if (!c.subject.equals(String.valueOf(uid))) continue;
            Map<String, Object> m = new LinkedHashMap<>();
            m.put("event", c.event);
            m.put("version", c.version);
            m.put("at", c.at);
            if (c.ipHash != null) m.put("ip_hash", c.ipHash);
            cs.add(m);
        }
        data.put("consents", cs);
        List<Object> ps = new ArrayList<>();
        for (var e : posts.entrySet()) {
```

Listing 11.6 (Continued) *PrivacyApp.java: the same application in Java.*

```
            Post p = e.getValue();
            if (!Integer.valueOf(uid).equals(p.author)) continue;
            Map<String, Object> m = new LinkedHashMap<>();
            m.put("id", e.getKey());
            m.put("text", p.text);
            m.put("parent", p.parent);
            m.put("at", p.at);
            ps.add(m);
        }
        data.put("posts", ps);
        return Json.write(data, "");
    }

    void changeEmail(String token, String password, String newEmail) {
        int uid = userOf(token);
        checkPassword(uid, password);
        newEmail = newEmail.strip().toLowerCase();
        if (!newEmail.contains("@") || find(newEmail) != null) {
            throw new PrivacyException("that email cannot be used");
        }
        users.get(uid).email = newEmail;
    }

    void changePassword(String token, String old, String fresh) {
        int uid = userOf(token);
        checkPassword(uid, old);
        if (fresh.length() < MIN_PASSWORD) {
            throw new PrivacyException("password needs at least " + MIN_PASSWORD
                    + " characters");
        }
        users.get(uid).password = hasher.hash(fresh);
        // sign out every other device
        sessions.entrySet().removeIf(e -> e.getValue() == uid
                                        && !e.getKey().equals(token));
    }
```

Listing 11.6 (Continued) *PrivacyApp.java: the same application in Java.*

```
    /** Right to erasure. Only the account holder can start it. */
    String deleteAccount(String token, String password) {
        int uid = userOf(token);
        checkPassword(uid, password);
        String tomb = "deleted-" + nextTomb++;
        Set<Integer> mine = new HashSet<>();
        posts.forEach((pid, p) -> {
            if (Integer.valueOf(uid).equals(p.author)) mine.add(pid);
        });
        Set<Integer> needed = new HashSet<>();
        for (Post p : posts.values()) {
            if (p.parent != null && mine.contains(p.parent)
                    && !Integer.valueOf(uid).equals(p.author)) needed.add(p.parent);
        }
        for (int pid : mine) {
            if (needed.contains(pid)) {           // others replied: empty shell
                Post p = posts.get(pid);
                p.author = null;
                p.text = "[deleted]";
            } else {
                posts.remove(pid);
            }
        }
        for (Consent c : consents) {
            if (c.subject.equals(String.valueOf(uid))) {
                c.subject = tomb;                 // proof, without the person
                c.ipHash = null;
            }
        }
        Consent w = new Consent();
        w.subject = tomb;
        w.event = "withdrawn";
        w.version = policyVersion;
        w.at = stamp();
        consents.add(w);
        sessions.values().removeIf(v -> v == uid);
        limiter.forget("acct:" + users.get(uid).email);
        users.remove(uid);
        tombstones.add(new String[] {tomb, stamp()});
        return tomb;
    }
```

Listing 11.6 (Continued) *PrivacyApp.java: the same application in Java.*

```
    /** Every stored table as one string, for the "nothing left" test. */
    String dump() {
        StringBuilder b = new StringBuilder();
        users.forEach((id, u) -> b.append(id).append(' ').append(u.email)
                .append(' ').append(u.password).append(' ').append(u.created)
                .append('\n'));
        for (Consent c : consents) {
            b.append(c.subject).append(' ').append(c.event).append(' ')
             .append(c.version).append(' ').append(c.at).append(' ')
             .append(c.ipHash).append('\n');
        }
        posts.forEach((id, p) -> b.append(id).append(' ').append(p.author)
                .append(' ').append(p.parent).append(' ').append(p.text)
                .append('\n'));
        sessions.forEach((t, id) ->
                b.append(t).append(' ').append(id).append('\n'));
        for (String[] t : tombstones) b.append(String.join(" ", t)).append('\n');
        return b.toString();
    }
}
```

Listing 11.7 `Json.java`: a small JSON writer with the same layout as Python's `json.dumps(indent=2)`.

```
import java.util.List;
import java.util.Map;

/** Writes maps, lists, strings, numbers, booleans and null as JSON with a
 *  two-space indent, the same layout as Python's json.dumps(indent=2). */
public class Json {
    static String write(Object v, String indent) {
        String in = indent + "  ";
        if (v instanceof Map<?, ?> map) {
            if (map.isEmpty()) return "{}";
            StringBuilder b = new StringBuilder("{\n");
            int i = 0;
            for (var e : map.entrySet()) {
                b.append(in).append(quote(String.valueOf(e.getKey()))).append(": ")
                 .append(write(e.getValue(), in));
                b.append(++i < map.size() ? ",\n" : "\n");
            }
            return b.append(indent).append('}').toString();
        }
        if (v instanceof List<?> list) {
            if (list.isEmpty()) return "[]";
            StringBuilder b = new StringBuilder("[\n");
            for (int i = 0; i < list.size(); i++) {
                b.append(in).append(write(list.get(i), in));
                b.append(i + 1 < list.size() ? ",\n" : "\n");
            }
            return b.append(indent).append(']').toString();
        }
        if (v instanceof String s) return quote(s);
        return String.valueOf(v);          // numbers, true, false, null
    }

    static String quote(String s) {
        StringBuilder b = new StringBuilder("\"");
        for (char c : s.toCharArray()) {
            switch (c) {
                case '"' -> b.append("\\\"");
                case '\\' -> b.append("\\\\");
                case '\n' -> b.append("\\n");
                case '\r' -> b.append("\\r");
                case '\t' -> b.append("\\t");
                default -> {
                    if (c < 0x20) b.append(String.format("\\u%04x", (int) c));
                    else b.append(c);
                }
            }
        }
        return b.append('"').toString();
    }
}
```

Listing 11.7 (Continued) *Json.java: a small JSON writer with the same layout as Python's json.dumps(indent=2).*

11.10.5 The demo

Listing 11.8 runs the scenario of **Figure 11.3** and **Figure 11.5**. It starts with a *legacy* recipe of 100,000 PBKDF2 iterations, registers two users, then moves the policy to 600,000 iterations (or to Argon2id with `--argon2`), so that the upgrade on login becomes visible. The helper `recipe` prints only the algorithm and its cost settings, never the salt or the hash.

Listing 11.8 `demo.py`: consent, hashing, rate limiting, export, correction and erasure in one scenario.

```
"""Walk through the G7 lab: consent, hashing, rate limit, export, erasure.

usage: python demo.py            (PBKDF2 from the standard library)
       python demo.py --argon2   (Argon2id; needs: pip install argon2-cffi)
"""
import sys

from passwords import Argon2Hasher, Pbkdf2Hasher
from privacy_app import PrivacyApp, PrivacyError


class Clock:
    def __init__(self, start):
        self.now = start

    def __call__(self):
        return self.now


def attempt(label, action, shown=None):
    try:
        result = action()
        print("%-38s -> %s" % (label, shown or result))
        return result
    except PrivacyError as e:
        print("%-38s -> refused: %s" % (label, e))
        return None


def recipe(stored):
    """The algorithm and its cost settings; never the salt or the hash."""
    parts = stored.split("$")
    return "$".join(parts[:2] if parts[0] else parts[:4])


def main():
    clock = Clock(1790215200)                # 2026-09-24 09:00:00 WIB
    legacy = Pbkdf2Hasher(iterations=100_000)
    app = PrivacyApp(clock, secret=b"lab-secret-kept-outside-the-db",
                     hasher=legacy, max_failures=3, window_seconds=60)
    ayu_pw, budi_pw = "kopi-susu-tanpa-gula-2026", "nasi-goreng-pedas-sekali"
    ayu_ip, budi_ip = "203.0.113.7", "198.51.100.23"
```

Listing 11.8 (Continued) *demo.py: consent, hashing, rate limiting, export, correction and erasure in one scenario.*

```
    print("== 1. Registration needs explicit consent")
    attempt("register ayu, consent box left empty",
            lambda: app.register("ayu@example.com", ayu_pw, False, ayu_ip))
    attempt("register ayu, consent box ticked",
            lambda: "user %d" % app.register("Ayu@Example.com", ayu_pw,
                                             True, ayu_ip))
    attempt("register budi, consent box ticked",
            lambda: "user %d" % app.register("budi@example.com", budi_pw,
                                             True, budi_ip))

    print("\n== 2. What is stored instead of the password")
    print("ayu's recipe: " + recipe(app.users[1]["password"]))
    new = Argon2Hasher() if "--argon2" in sys.argv else Pbkdf2Hasher()
    app.use_hasher(new)
    print("policy moves to: " + recipe(new.hash("x" * 16)))

    print("\n== 3. Login rate limiting (lab values: 3 failures in 60 s)")
    for guess in ("kopi-susu-2025", "kopi-susu-2024", "kopi-susu-2023"):
        clock.now += 5
        attempt("09:00:%02d ayu, wrong password" % (clock.now % 60),
                lambda: app.login("ayu@example.com", guess, ayu_ip))
    clock.now += 5
    attempt("09:00:20 ayu, right password",
            lambda: app.login("ayu@example.com", ayu_pw, ayu_ip), "signed in")
    clock.now += 60
    token = attempt("09:01:20 ayu, right password",
                    lambda: app.login("ayu@example.com", ayu_pw, ayu_ip),
                    "signed in")
    print("ayu's recipe after login: " + recipe(app.users[1]["password"]))
    budi = app.login("budi@example.com", budi_pw, budi_ip)

    print("\n== 4. Ordinary use")
    first = app.post(token, "Is express laundry open on Sunday?")
    app.post(budi, "Yes, until noon.", parent=first)
    print("ayu posted #%d; budi replied" % first)

    print("\n== 5. Right of access and portability: export")
    print(app.export(token))

    print("\n== 6. Right to correction: change email")
    attempt("change email, wrong password",
            lambda: app.change_email(token, "guess-guess-guess", "a@x.id"))
    attempt("change email, password confirmed",
            lambda: app.change_email(token, ayu_pw, "ayu.new@example.com"),
            "email is now ayu.new@example.com")
```

Listing 11.8 (Continued) *demo.py: consent, hashing, rate limiting, export, correction and erasure in one scenario.*

```
    print("\n== 7. Right to erasure")
    clock.now += 3600
    attempt("delete ayu's account",
            lambda: app.delete_account(token, ayu_pw))
    reply = app.posts[2]
    print("budi's reply still answers post #%d: %r" % (
        reply["parent"], app.posts[reply["parent"]]["text"]))

    print("\n== 8. What is left of ayu")
    dump = app.dump()
    for trace in ("ayu@example.com", "ayu.new@example.com", ayu_ip,
                  app.ip_hash(ayu_ip), "Is express laundry open"):
        print("%-30s %s" % (trace, "FOUND" if trace in dump else "not found"))
    for c in app.consents:
        print("consent log: %-10s %-9s v%s %s" % (c["subject"], c["event"],
                                                  c["version"], c["at"]))


if __name__ == "__main__":
    main()
```

Listing 11.9 is the Java demo.

Listing 11.9 `Demo.java`: the same scenario in Java.

```
import java.util.function.Supplier;

/** Walks through the G7 lab: consent, hashing, rate limit, export, erasure.
 *  usage: java Demo      (PBKDF2 from the JDK) */
public class Demo {
    static long now = 1790215200L;                  // 2026-09-24 09:00:00 WIB

    static <T> T attempt(String label, Supplier<T> action, String shown) {
        try {
            T result = action.get();
            System.out.printf("%-38s -> %s%n", label,
                              shown != null ? shown : result);
            return result;
        } catch (PrivacyApp.PrivacyException e) {
            System.out.printf("%-38s -> refused: %s%n", label, e.getMessage());
            return null;
        }
    }

    /** The algorithm and its cost settings; never the salt or the hash. */
    static String recipe(String stored) {
        String[] p = stored.split("\\$");
        return p[0] + "$" + p[1];
    }

    public static void main(String[] args) {
        Pbkdf2Hasher legacy = new Pbkdf2Hasher(100_000);
        PrivacyApp app = new PrivacyApp(() -> now,
                "lab-secret-kept-outside-the-db".getBytes(), legacy, 3, 60);
        String ayuPw = "kopi-susu-tanpa-gula-2026";
        String budiPw = "nasi-goreng-pedas-sekali";
        String ayuIp = "203.0.113.7", budiIp = "198.51.100.23";

        System.out.println("== 1. Registration needs explicit consent");
        attempt("register ayu, consent box left empty",
                () -> app.register("ayu@example.com", ayuPw, false, ayuIp), null);
        attempt("register ayu, consent box ticked",
                () -> "user " + app.register("Ayu@Example.com", ayuPw, true, ayuIp),
                null);
        attempt("register budi, consent box ticked",
                () -> "user " + app.register("budi@example.com", budiPw, true,
                                             budiIp),
                null);
```

Listing 11.9 (Continued) *Demo.java: the same scenario in Java.*

```
        System.out.println("\n== 2. What is stored instead of the password");
        System.out.println("ayu's recipe: " + recipe(app.users.get(1).password));
        Pbkdf2Hasher fresh = new Pbkdf2Hasher();
        app.useHasher(fresh);
        System.out.println("policy moves to: "
                + recipe(fresh.hash("x".repeat(16))));

        System.out.println(
                "\n== 3. Login rate limiting (lab values: 3 failures in 60 s)");
        for (String guess : new String[] {"kopi-susu-2025", "kopi-susu-2024",
                                          "kopi-susu-2023"}) {
            now += 5;
            attempt(String.format("09:00:%02d ayu, wrong password", now % 60),
                    () -> app.login("ayu@example.com", guess, ayuIp), null);
        }
        now += 5;
        attempt("09:00:20 ayu, right password",
                () -> app.login("ayu@example.com", ayuPw, ayuIp), "signed in");
        now += 60;
        String token = attempt("09:01:20 ayu, right password",
                () -> app.login("ayu@example.com", ayuPw, ayuIp), "signed in");
        System.out.println("ayu's recipe after login: "
                + recipe(app.users.get(1).password));
        String budi = app.login("budi@example.com", budiPw, budiIp);

        System.out.println("\n== 4. Ordinary use");
        int first = app.post(token, "Is express laundry open on Sunday?", null);
        app.post(budi, "Yes, until noon.", first);
        System.out.println("ayu posted #" + first + "; budi replied");

        System.out.println("\n== 5. Right of access and portability: export");
        System.out.println(app.export(token));

        System.out.println("\n== 6. Right to correction: change email");
        attempt("change email, wrong password", () -> {
            app.changeEmail(token, "guess-guess-guess", "a@x.id");
            return "";
        }, null);
        attempt("change email, password confirmed", () -> {
            app.changeEmail(token, ayuPw, "ayu.new@example.com");
            return "";
        }, "email is now ayu.new@example.com");
```

Listing 11.9 (Continued) *Demo.java: the same scenario in Java.*

```
        System.out.println("\n== 7. Right to erasure");
        now += 3600;
        attempt("delete ayu's account", () -> app.deleteAccount(token, ayuPw),
                null);
        PrivacyApp.Post reply = app.posts.get(2);
        System.out.println("budi's reply still answers post #" + reply.parent
                + ": '" + app.posts.get(reply.parent).text + "'");

        System.out.println("\n== 8. What is left of ayu");
        String dump = app.dump();
        for (String trace : new String[] {"ayu@example.com", "ayu.new@example.com",
                ayuIp, app.ipHash(ayuIp), "Is express laundry open"}) {
            System.out.printf("%-30s %s%n", trace,
                    dump.contains(trace) ? "FOUND" : "not found");
        }
        for (PrivacyApp.Consent c : app.consents) {
            System.out.printf("consent log: %-10s %-9s v%s %s%n", c.subject,
                    c.event, c.version, c.at);
        }
    }
}
```

Run `python demo.py` in the `python` folder, or `javac -d out *.java` and `java -cp out Demo` in the `java` folder. Both print the following, identically:

```
== 1. Registration needs explicit consent
register ayu, consent box left empty   -> refused: consent is required
register ayu, consent box ticked       -> user 1
register budi, consent box ticked      -> user 2

== 2. What is stored instead of the password
ayu's recipe: pbkdf2_sha256$100000
policy moves to: pbkdf2_sha256$600000

== 3. Login rate limiting (lab values: 3 failures in 60 s)
09:00:05 ayu, wrong password           -> refused: wrong email or password
09:00:10 ayu, wrong password           -> refused: wrong email or password
09:00:15 ayu, wrong password           -> refused: wrong email or password
09:00:20 ayu, right password           -> refused: too many attempts, wait 45 s
09:01:20 ayu, right password           -> signed in
ayu's recipe after login: pbkdf2_sha256$600000

== 4. Ordinary use
ayu posted #1; budi replied

== 5. Right of access and portability: export
{
  "exported_at": "2026-09-24T09:01:20+07:00",
  "account": {
    "id": 1,
    "email": "ayu@example.com",
    "created": "2026-09-24T09:00:00+07:00"
  },
  "consents": [
    {
      "event": "given",
      "version": "1.0",
      "at": "2026-09-24T09:00:00+07:00",
      "ip_hash": "2f7a7c6e655482f7"
    }
  ],
  "posts": [
    {
      "id": 1,
      "text": "Is express laundry open on Sunday?",
      "parent": null,
      "at": "2026-09-24T09:01:20+07:00"
    }
  ]
}

== 6. Right to correction: change email
change email, wrong password           -> refused: password confirmation failed
change email, password confirmed       -> email is now ayu.new@example.com

== 7. Right to erasure
delete ayu's account                   -> deleted-1
budi's reply still answers post #1: '[deleted]'

== 8. What is left of ayu
ayu@example.com               not found
ayu.new@example.com           not found
203.0.113.7                   not found
2f7a7c6e655482f7              not found
Is express laundry open       not found
consent log: deleted-1  given     v1.0 2026-09-24T09:00:00+07:00
consent log: 2          given     v1.0 2026-09-24T09:00:00+07:00
consent log: deleted-1  withdrawn v1.0 2026-09-24T10:01:20+07:00
```

Follow it section by section.

1. The first registration is refused because the consent box was left empty; the next two succeed. Ayu typed `Ayu@Example.com`, and the application stored `ayu@example.com`, so that she cannot accidentally end up with two accounts.
2. Ayu's stored string says `pbkdf2_sha256$100000`: the legacy recipe. The policy then moves to 600,000 iterations.
3. Three wrong passwords at 09:00:05, :10 and :15 are refused with the same message. At 09:00:20 Ayu types the *right* password and is still refused, without her password even being checked, and told to wait 45 seconds, exactly Equation (11.3). At 09:01:20 she signs in, and her stored string has been upgraded to `pbkdf2_sha256$600000`.
4. Ayu asks a question and Budi replies.
5. Ayu's export contains her account, her consent event with its version, time and IP hash, and her post; it contains neither Budi's reply nor any password hash.
6. An email change with a wrong password is refused; with the right password it succeeds.
7. An hour later Ayu deletes her account and receives the tombstone `deleted-1`. Budi's reply still points to post 1, which now reads `[deleted]`.
8. Searching every table for Ayu's old and new email addresses, her IP address, its keyed hash and the text of her question finds nothing. The consent log still proves that someone consented to version 1.0 at 09:00 and withdrew at 10:01, but it no longer says who.

With `argon2-cffi` installed, `python demo.py --argon2` prints the same lines except two, which show the Argon2id recipe of the OWASP configuration instead:

```
policy moves to: $argon2id$v=19$m=19456,t=2,p=1
ayu's recipe after login: $argon2id$v=19$m=19456,t=2,p=1
```

Ayu's PBKDF2 hash, verified with its own recipe, has been replaced by an Argon2id hash at her first successful login, which is the upgrade path Profitina uses for legacy hashes.

11.10.6 The tests

A privacy measure that is not tested is a promise, not a measure. **Listing 11.10** contains ten tests. Two check the hashing itself against outside references: the known answer for PBKDF2-HMAC-SHA256 published in RFC 7914 [19], and a stored string for a fixed salt that the Java version must reproduce character for character. The others check the behavior of this chapter: consent must be exactly `True`; unknown email and wrong password give the same message; the window blocks and releases; old hashes are upgraded; the export holds only the user's own data; changes need the password and sign out other sessions; and erasure leaves nothing personal behind.

Listing 11.10 `test_privacy_app.py`: ten tests, including a known-answer test for PBKDF2 and a search of every table after an erasure.

```
"""Tests for the G7 lab. Run: python -m unittest -v test_privacy_app"""
import json
import unittest

from passwords import Pbkdf2Hasher
from privacy_app import GENERIC, PrivacyApp, PrivacyError

PW = "kopi-susu-tanpa-gula-2026"
IP = "203.0.113.7"


class Clock:
    def __init__(self):
        self.now = 1790215200

    def __call__(self):
        return self.now


def new_app(clock=None):
    # Few iterations keep the tests fast; the demo uses the OWASP value.
    return PrivacyApp(clock or Clock(), b"test-secret",
                      hasher=Pbkdf2Hasher(iterations=1000))


class PrivacyTests(unittest.TestCase):
    def test_pbkdf2_known_answer(self):
        # RFC 7914, section 11: PBKDF2-HMAC-SHA256("passwd", "salt", 1, 64)
        import hashlib
        key = hashlib.pbkdf2_hmac("sha256", b"passwd", b"salt", 1, 64)
        self.assertTrue(key.hex().startswith("55ac046e56e3089fec1691c22544b605"))

    def test_recipe_is_the_same_in_java(self):
        # The Java lab computes exactly this string for the same inputs.
        expected = ("pbkdf2_sha256$1000$cHJvZml0aW5hLWxhYi0xNg==$"
                    "RJZs9e03R01zF3dwuFJoifK15hAmgK6TPoBFUVtqkrE=")
        h = Pbkdf2Hasher(iterations=1000)
        self.assertEqual(h.hash(PW, salt=b"profitina-lab-16"), expected)
        self.assertTrue(h.verify(expected, PW))
```

Listing 11.10 (Continued) *test_privacy_app.py: ten tests, including a known-answer test for PBKDF2 and a search of every table after an erasure.*

```
    def test_hash_is_salted_and_verifies(self):
        h = Pbkdf2Hasher(iterations=1000)
        a, b = h.hash(PW), h.hash(PW)
        self.assertNotEqual(a, b)                # a fresh salt every time
        self.assertTrue(h.verify(a, PW))
        self.assertFalse(h.verify(a, PW + "x"))
        self.assertNotIn(PW, a)

    def test_consent_must_be_explicitly_true(self):
        app = new_app()
        for value in (False, None, "yes", 1):
            with self.assertRaises(PrivacyError):
                app.register("ayu@example.com", PW, value, IP)
        uid = app.register("ayu@example.com", PW, True, IP)
        log = [c for c in app.consents if c["subject"] == uid]
        self.assertEqual(log[0]["event"], "given")
        self.assertEqual(log[0]["version"], "1.0")
        self.assertNotIn(IP, app.dump())         # only a keyed hash is kept

    def test_same_message_for_unknown_email_and_wrong_password(self):
        app = new_app()
        app.register("ayu@example.com", PW, True, IP)
        messages = []
        for email, pw in (("ayu@example.com", "wrong-password-123"),
                          ("nobody@example.com", PW)):
            with self.assertRaises(PrivacyError) as ctx:
                app.login(email, pw, "198.51.100." + str(len(messages)))
            messages.append(str(ctx.exception))
        self.assertEqual(messages, [GENERIC, GENERIC])

    def test_sliding_window_blocks_then_releases(self):
        clock = Clock()
        app = new_app(clock)
        app.register("ayu@example.com", PW, True, IP)
        for _ in range(3):
            clock.now += 1
            self.assertRaises(PrivacyError, app.login, "ayu@example.com",
                              "wrong-password-123", IP)
        with self.assertRaises(PrivacyError) as ctx:
            app.login("ayu@example.com", PW, IP)   # right password, still no
        self.assertIn("too many attempts", str(ctx.exception))
        clock.now += 60
        self.assertTrue(app.login("ayu@example.com", PW, IP))
```

Listing 11.10 (Continued) *test_privacy_app.py: ten tests, including a known-answer test for PBKDF2 and a search of every table after an erasure.*

```
    def test_old_hash_is_upgraded_at_login(self):
        app = new_app()
        uid = app.register("ayu@example.com", PW, True, IP)
        app.use_hasher(Pbkdf2Hasher(iterations=2000))
        self.assertIn("$1000$", app.users[uid]["password"])
        app.login("ayu@example.com", PW, IP)
        self.assertIn("$2000$", app.users[uid]["password"])

    def test_export_contains_only_own_data(self):
        app = new_app()
        app.register("ayu@example.com", PW, True, IP)
        app.register("budi@example.com", PW, True, "198.51.100.23")
        ayu = app.login("ayu@example.com", PW, IP)
        budi = app.login("budi@example.com", PW, "198.51.100.23")
        app.post(ayu, "hello from ayu")
        app.post(budi, "hello from budi")
        data = json.loads(app.export(ayu))
        self.assertEqual(data["account"]["email"], "ayu@example.com")
        self.assertEqual([p["text"] for p in data["posts"]], ["hello from ayu"])
        self.assertNotIn("budi", json.dumps(data))
        self.assertNotIn("password", json.dumps(data))

    def test_changes_need_the_password(self):
        app = new_app()
        app.register("ayu@example.com", PW, True, IP)
        t1 = app.login("ayu@example.com", PW, IP)
        t2 = app.login("ayu@example.com", PW, IP)
        self.assertRaises(PrivacyError, app.change_email, t1,
                          "not-the-password", "new@example.com")
        app.change_password(t1, PW, "a-brand-new-long-passphrase")
        self.assertIn(t1, app.sessions)
        self.assertNotIn(t2, app.sessions)       # other devices signed out

    def test_delete_leaves_nothing_personal(self):
        app = new_app()
        app.register("ayu@example.com", PW, True, IP)
        app.register("budi@example.com", PW, True, "198.51.100.23")
        ayu = app.login("ayu@example.com", PW, IP)
        budi = app.login("budi@example.com", PW, "198.51.100.23")
        first = app.post(ayu, "a question from ayu")
        app.post(ayu, "a note nobody answered")
        reply = app.post(budi, "an answer from budi", parent=first)
        app.change_email(ayu, PW, "ayu.new@example.com")
        stored_hash = app.users[1]["password"]
        ip_hash = app.ip_hash(IP)
        self.assertRaises(PrivacyError, app.delete_account, ayu, "wrong-one")
```

Listing 11.10 (Continued) *test_privacy_app.py: ten tests, including a known-answer test for PBKDF2 and a search of every table after an erasure.*

```
        tomb = app.delete_account(ayu, PW)
        dump = app.dump()
        for trace in ("ayu@example.com", "ayu.new@example.com", IP, ip_hash,
                      stored_hash, "a question from ayu", "nobody answered"):
            self.assertNotIn(trace, dump)
        self.assertEqual(app.posts[first]["text"], "[deleted]")
        self.assertEqual(app.posts[reply]["parent"], first)  # thread intact
        events = [c["event"] for c in app.consents if c["subject"] == tomb]
        self.assertEqual(events, ["given", "withdrawn"])
        self.assertFalse(any("ayu" in k for k in app.limiter.failures))
        self.assertRaises(PrivacyError, app.export, ayu)     # session gone
        self.assertIn("budi@example.com", dump)              # others untouched


if __name__ == "__main__":
    unittest.main()
```

The last test is the one the rest of the lab exists for. It builds a realistic small history (two users, a question, an unanswered note, a reply, an email change), erases Ayu, serializes every table, and asserts that none of seven traces is present: both email addresses, the IP address, its keyed hash, the stored password hash, and the text of both of her posts. It also checks what must *remain*: the reply thread is intact, the tombstone has exactly a `given` and a `withdrawn` event, Budi's data is untouched, and Ayu's session no longer works.

The tests set the PBKDF2 iterations to 1,000 so that they run in a fraction of a second; the demo and the default constructor use the OWASP value. Run them with `python -m unittest test_privacy_app`:

```
..........
----------------------------------------------------------------------
Ran 10 tests in 0.021s

OK
```

Java has no test framework in its standard library, so **Listing 11.11** carries a ten-line runner: each test is a method, a table lists the tests with their names, and the runner prints `ok` or `FAILED` for each and exits with status 1 if anything failed, which is all a build script needs.

Listing 11.11 `PrivacyAppTest.java`: the same ten properties in Java, with a minimal runner.

```
import java.util.HexFormat;
import java.util.List;

/** Tests for the G7 lab, with a ten-line runner instead of a framework.
 *  Run: java -cp out PrivacyAppTest */
public class PrivacyAppTest {
    static final String PW = "kopi-susu-tanpa-gula-2026";
    static final String IP = "203.0.113.7";
    static long now;

    interface Test {
        void run() throws Exception;
    }

    static void check(boolean condition, String what) {
        if (!condition) throw new AssertionError(what);
    }

    static void refused(Runnable action, String what) {
        try {
            action.run();
        } catch (PrivacyApp.PrivacyException e) {
            return;
        }
        throw new AssertionError("not refused: " + what);
    }

    static PrivacyApp newApp() {
        now = 1790215200L;
        // Few iterations keep the tests fast; the demo uses the OWASP value.
        return new PrivacyApp(() -> now, "test-secret".getBytes(),
                              new Pbkdf2Hasher(1000), 3, 60);
    }

    static void pbkdf2KnownAnswer() {
        // RFC 7914, section 11: PBKDF2-HMAC-SHA256("passwd", "salt", 1, 64)
        byte[] key = Pbkdf2Hasher.derive("passwd", "salt".getBytes(), 1);
        check(HexFormat.of().formatHex(key).startsWith("55ac046e56e3089fec1691c2"),
              "known answer");
    }
```

Listing 11.11 (Continued) *PrivacyAppTest.java: the same ten properties in Java, with a minimal runner.*

```
    static void recipeIsTheSameInPython() {
        String expected = "pbkdf2_sha256$1000$cHJvZml0aW5hLWxhYi0xNg==$"
                + "RJZs9e03R01zF3dwuFJoifK15hAmgK6TPoBFUVtqkrE=";
        Pbkdf2Hasher h = new Pbkdf2Hasher(1000);
        check(h.hash(PW, "profitina-lab-16".getBytes()).equals(expected), "same");
        check(h.verify(expected, PW), "verifies");
    }

    static void hashIsSaltedAndVerifies() {
        Pbkdf2Hasher h = new Pbkdf2Hasher(1000);
        String a = h.hash(PW), b = h.hash(PW);
        check(!a.equals(b), "fresh salt");
        check(h.verify(a, PW) && !h.verify(a, PW + "x"), "verify");
        check(!a.contains(PW), "no plain text");
    }

    static void consentMustBeExplicitlyTrue() {
        PrivacyApp app = newApp();
        for (Boolean value : new Boolean[] {false, null}) {
            refused(() -> app.register("ayu@example.com", PW, value, IP),
                    "consent");
        }
        app.register("ayu@example.com", PW, true, IP);
        check(app.consents.get(0).event.equals("given"), "logged");
        check(app.consents.get(0).version.equals("1.0"), "version");
        check(!app.dump().contains(IP), "only a keyed hash of the IP");
    }

    static void sameMessageForUnknownEmailAndWrongPassword() {
        PrivacyApp app = newApp();
        app.register("ayu@example.com", PW, true, IP);
        String[] msg = new String[2];
        try {
            app.login("ayu@example.com", "wrong-password-123", "198.51.100.1");
        } catch (PrivacyApp.PrivacyException e) {
            msg[0] = e.getMessage();
        }
        try {
            app.login("nobody@example.com", PW, "198.51.100.2");
        } catch (PrivacyApp.PrivacyException e) {
            msg[1] = e.getMessage();
        }
        check(PrivacyApp.GENERIC.equals(msg[0])
              && PrivacyApp.GENERIC.equals(msg[1]),
              "same message");
    }
```

Listing 11.11 (Continued) *PrivacyAppTest.java: the same ten properties in Java, with a minimal runner.*

```
    static void slidingWindowBlocksThenReleases() {
        PrivacyApp app = newApp();
        app.register("ayu@example.com", PW, true, IP);
        for (int i = 0; i < 3; i++) {
            now += 1;
            refused(() -> app.login("ayu@example.com", "wrong-password-123", IP),
                    "wrong password");
        }
        refused(() -> app.login("ayu@example.com", PW, IP), "blocked");
        now += 60;
        check(app.login("ayu@example.com", PW, IP) != null, "released");
    }

    static void oldHashIsUpgradedAtLogin() {
        PrivacyApp app = newApp();
        int uid = app.register("ayu@example.com", PW, true, IP);
        app.useHasher(new Pbkdf2Hasher(2000));
        check(app.users.get(uid).password.contains("$1000$"), "old recipe");
        app.login("ayu@example.com", PW, IP);
        check(app.users.get(uid).password.contains("$2000$"), "upgraded");
    }

    static void exportContainsOnlyOwnData() {
        PrivacyApp app = newApp();
        app.register("ayu@example.com", PW, true, IP);
        app.register("budi@example.com", PW, true, "198.51.100.23");
        String ayu = app.login("ayu@example.com", PW, IP);
        String budi = app.login("budi@example.com", PW, "198.51.100.23");
        app.post(ayu, "hello from ayu", null);
        app.post(budi, "hello from budi", null);
        String json = app.export(ayu);
        check(json.contains("\"email\": \"ayu@example.com\""), "own email");
        check(json.contains("hello from ayu") && !json.contains("budi"),
              "own only");
        check(!json.contains("password"), "no hash in the export");
    }
```

Listing 11.11 (Continued) *PrivacyAppTest.java: the same ten properties in Java, with a minimal runner.*

```
    static void changesNeedThePassword() {
        PrivacyApp app = newApp();
        app.register("ayu@example.com", PW, true, IP);
        String t1 = app.login("ayu@example.com", PW, IP);
        String t2 = app.login("ayu@example.com", PW, IP);
        refused(() -> app.changeEmail(t1, "not-the-password", "new@example.com"),
                "email change");
        app.changePassword(t1, PW, "a-brand-new-long-passphrase");
        check(app.sessions.containsKey(t1), "this device stays");
        check(!app.sessions.containsKey(t2), "other devices signed out");
    }

    static void deleteLeavesNothingPersonal() {
        PrivacyApp app = newApp();
        app.register("ayu@example.com", PW, true, IP);
        app.register("budi@example.com", PW, true, "198.51.100.23");
        String ayu = app.login("ayu@example.com", PW, IP);
        String budi = app.login("budi@example.com", PW, "198.51.100.23");
        int first = app.post(ayu, "a question from ayu", null);
        app.post(ayu, "a note nobody answered", null);
        int reply = app.post(budi, "an answer from budi", first);
        app.changeEmail(ayu, PW, "ayu.new@example.com");
        String storedHash = app.users.get(1).password;
        String ipHash = app.ipHash(IP);
        refused(() -> app.deleteAccount(ayu, "wrong-one"), "wrong password");
        String tomb = app.deleteAccount(ayu, PW);
        String dump = app.dump();
        for (String trace : List.of("ayu@example.com", "ayu.new@example.com", IP,
                ipHash, storedHash, "a question from ayu", "nobody answered")) {
            check(!dump.contains(trace), "left behind: " + trace);
        }
        check(app.posts.get(first).text.equals("[deleted]"), "tombstone post");
        check(app.posts.get(reply).parent == first, "thread intact");
        long events = app.consents.stream().filter(c -> c.subject.equals(tomb))
                .map(c -> c.event).filter(e -> e.equals("given")
                        || e.equals("withdrawn")).count();
        check(events == 2, "given and withdrawn");
        check(app.limiter.failures.keySet().stream()
                 .noneMatch(k -> k.contains("ayu")),
              "limiter forgot the account");
        refused(() -> app.export(ayu), "session gone");
        check(dump.contains("budi@example.com"), "others untouched");
    }
```

Listing 11.11 (Continued) *PrivacyAppTest.java: the same ten properties in Java, with a minimal runner.*

```
    public static void main(String[] args) {
        Object[][] tests = {
            {"pbkdf2 known answer", (Test) PrivacyAppTest::pbkdf2KnownAnswer},
            {"recipe is the same in Python",
                (Test) PrivacyAppTest::recipeIsTheSameInPython},
            {"hash is salted and verifies",
                (Test) PrivacyAppTest::hashIsSaltedAndVerifies},
            {"consent must be explicitly true",
                (Test) PrivacyAppTest::consentMustBeExplicitlyTrue},
            {"same message for unknown email and wrong password",
                (Test) PrivacyAppTest::sameMessageForUnknownEmailAndWrongPassword},
            {"sliding window blocks then releases",
                (Test) PrivacyAppTest::slidingWindowBlocksThenReleases},
            {"old hash is upgraded at login",
                (Test) PrivacyAppTest::oldHashIsUpgradedAtLogin},
            {"export contains only own data",
                (Test) PrivacyAppTest::exportContainsOnlyOwnData},
            {"changes need the password",
                (Test) PrivacyAppTest::changesNeedThePassword},
            {"delete leaves nothing personal",
                (Test) PrivacyAppTest::deleteLeavesNothingPersonal},
        };
        int failed = 0;
        for (Object[] t : tests) {
            try {
                ((Test) t[1]).run();
                System.out.println("ok      " + t[0]);
            } catch (Throwable e) {
                failed++;
                System.out.println("FAILED  " + t[0] + ": " + e);
            }
        }
        System.out.println(tests.length + " tests, " + failed + " failed");
        if (failed > 0) System.exit(1);
    }
}
```

```
ok      pbkdf2 known answer
ok      recipe is the same in Python
ok      hash is salted and verifies
ok      consent must be explicitly true
ok      same message for unknown email and wrong password
ok      sliding window blocks then releases
ok      old hash is upgraded at login
ok      export contains only own data
ok      changes need the password
ok      delete leaves nothing personal
10 tests, 0 failed
```

The test named “recipe is the same in Python” is a small demonstration of why a self-describing hash format matters beyond one program: a hash written by one implementation, in one language, verifies in another. A team that moves a service from Python to Java, or the other way round, does not have to ask its users to reset their passwords.

11.11 Profitina in Practice

The measures of this chapter live in a dedicated data-protection module of Profitina’s application, alongside the scan, community and PROFIT modules described in Chapter 9 [1]. What is live, as of 24 September 2026:

Live. Registration with email and password only; a consent checkbox that is never pre-ticked, with a short bilingual notice; every consent logged with the policy version and the time. Passwords stored with Argon2id, with legacy hashes upgraded at login. Login rate limiting per account identity and, for anonymous traffic, per IP address, held in memory only. Email verification by one-time code as the first anti-bot layer. The *Akun & Data Pribadi* panel in the *Pantau* tab, since 19 August 2026, with JSON export, email change, password change and permanent account deletion, each confirmed with the password; erasure that purges personal data across all modules and leaves an anonymous tombstone; production data deleted or anonymized within 30 days of deletion. Profiles with per-item consent, private by default; neutral handles; no phone fields; email never publishable. Direct messages that administrators cannot read except for a frozen copy of a reported message. Error records without raw bodies or headers, with hashed IP addresses, purged after 90 days. The Privacy Policy (version 1.15, effective 9 September 2026) and the Terms (version 1.1) as bilingual HTML in the application, with the marketing site linking to them. A breach runbook for the 72-hour deadline, with prepared notices. One contact address for everything: info@profitina.com [1], [7].

House rules. Every feature that touches personal data updates the ROPA and, where needed, the Privacy Policy. Rights, safety and account recovery stay free forever. Only the user can start an erasure. Academic features process published bibliographic records, never profile, photo or community data, and a person who is not found gets no score [1].

Still ahead. Compliance is a continuing process, not a certificate: PP 33/2026 takes effect in January 2027, the supervisory agency is still being formed, and Profitina, like every controller, will check its procedures against the regulation's details as they are applied.

11.12 Common Misunderstandings

"The agency does not exist yet, so UU PDP is not enforced." The law's obligations have applied in full since 17 October 2024, and Komdigi supervises in the transition [4]. Waiting for an agency is waiting to be found unprepared.

"We encrypt passwords." Passwords should be *hashed* with a slow, salted password-hashing function, not encrypted. Encryption is designed to be reversed by whoever holds the key (Section 11.5.1).

"SHA-256 is secure, so it is fine for passwords." SHA-256 is a secure hash for checking that data has not changed, and much too fast for passwords (**Table 11.1**, Equation (11.1)).

"A consent checkbox that is ticked by default is still consent." It records the designer's choice. Pasal 22 requires an explicit, recorded act, and consent that fails the conditions is void [2].

"We have `consented = true` in the users table, so we can prove consent." A boolean does not say what was agreed, when, or under which version, and it forgets withdrawals. Consent is a log of events (Section 11.6.2).

"Deleting the user row is deleting the user." Personal data lives in posts, logs, sessions, caches and backups too. Erasure is a purge across every table, with a tombstone only where other people's content needs one, and a stated time for backups (**Figure 11.5**).

"Rate limiting is about stopping one attacker." Credential stuffing uses many accounts and many addresses at a low rate each. Limit per account *and* per address, refuse before hashing, and keep the error message the same (Section 11.7.3).

"The export function exists, so users have the right of access." If users cannot find it, they do not have it. Profitina's own record shows the difference between implemented and reachable (Section 11.8.2).

"Hashing IP addresses makes them anonymous." A plain hash of an IPv4 address can be reversed by hashing all four billion addresses. Use a keyed hash with a secret kept outside the data, keep only what the purpose needs, and erase it with the account.

11.13 Chapter Summary

- UU PDP No. 27/2022, enacted 17 October 2022 with a two-year transition to 17 October 2024, defines general and specific personal data (Pasal 4), controllers and processors, principles (Pasal 16), lawful bases (Pasal 20), consent requirements (Pasal 21, 22), data-subject rights (Pasal 5 to 13), security duties (Pasal 35, 39), deletion (Pasal 43 to 45), 72-hour breach notification (Pasal 46), and sanctions up to 2% of annual revenue (Pasal 57), plus criminal provisions.
- PP 33/2026, the implementing regulation, was set and promulgated on 16 July 2026 (225 articles, 12 chapters) and takes effect on 16 January 2027. The supervisory agency's Perpres was submitted on 20 May 2026; Komdigi supervises meanwhile; one expert expects an effective agency in 2028 at the earliest.
- Privacy by design means concrete decisions: collect only email and password, keep things private by default, record processing in a ROPA, purge logs with jobs that actually run, and keep one home for the legal text.
- Passwords are stored with a slow, salted, self-describing hash. Equation (11.1) explains why: the defender controls the cost of each guess. Argon2id (RFC 9106) is memory-hard; PBKDF2 at 600,000 iterations is the standard-library fallback; hashes are upgraded at login.
- Consent is an explicit, recorded event with a version and a time, never pre-ticked, withdrawable, and granular where the product is.
- A sliding-window limiter (Equation (11.2), Equation (11.3)) on both account and address keys, checked before hashing and with one error message, defends against brute force and credential stuffing.
- Rights are reachable when a person can use them from the interface: Profitina's *Akun & Data Pribadi* panel provides export, correction and erasure, free forever, with erasure only by the user and a tombstone only where others need it.
- Lab G7 implements all of it in Python and Java and proves with tests that consent is explicit, the limiter works, hashes upgrade, the export is complete and exclusive, and erasure leaves nothing personal behind.

11.14 Exercises

Exercise 11.1. Classify each item as general personal data, specific personal data, or not personal data under Pasal 4: (a) a customer's full name; (b) a fingerprint template used to unlock a laundry locker; (c) a customer's bank balance; (d) the laundry's opening hours; (e) the medical allergy a customer mentions so that a certain detergent is avoided; (f) the religion stated on a membership form.

Exercise 11.2. For each processing activity of a laundry app, name the most suitable lawful basis from Pasal 20: (a) storing the customer's phone number to call when the order is ready; (b)

keeping invoices for the period tax law requires; (c) sending a monthly promotional newsletter; (d) keeping server logs for a few weeks to detect attacks.

Exercise 11.3. A password consists of 10 lowercase letters. Using Equation (11.1) and the rates of **Table 11.1**, compute the time to try every possibility with salted SHA-256 and with PBKDF2 at 600,000 iterations. Then explain in two sentences why the real advantage of the slow function is smaller than the ratio suggests, and why that argues for Argon2id.

Exercise 11.4. Two users choose the same password. Explain what their stored strings look like with and without a salt, and what an attacker who has stolen the table can do in each case.

Exercise 11.5. A limiter allows $n = 3$ failures in $w = 60$ seconds for one key. Attempts arrive at $t = 0{,}10{,}20$ (wrong password), 30 (right password), 61 (wrong), 65 (right) and 71 (right). Using Equation (11.2) and Equation (11.3), say for each attempt whether it is checked or refused, and give the waiting time for each refusal. Assume refused attempts are not recorded as failures.

Exercise 11.6. A fixed-window limiter allows 3 failures per clock minute. Show a sequence of attempts that achieves 6 failures within 2 seconds. What does the sliding window of Equation (11.2) allow for the same sequence?

Exercise 11.7. The posts table holds: post 1 by Ayu (no parent), post 2 by Budi replying to 1, post 3 by Ayu replying to 2, post 4 by Ayu (no parent), post 5 by Ayu replying to 4. Ayu erases her account. Using the rule of `delete_account`, which rows are deleted and which become `[deleted]` shells? Is Budi's reply still attached to something?

Exercise 11.8. The Privacy Policy changes from version 1.0 to 1.1. Write, in Python and in Java, a function `needs_consent` (Java: `needsConsent`) that says whether a logged-in user's latest `given` event is for the current version, and a function `accept_policy` (Java: `acceptPolicy`) that records a new `given` event. Demonstrate them.

Exercise 11.9. A breach is discovered on Monday at 21:00 WIB. By when must the written notices of Pasal 46 be sent? Using the phases of Profitina's runbook (containment in hours 0 to 2, assessment in hours 2 to 12, notification in hours 12 to 48), give the clock time at which each phase ends.

Exercise 11.10. Before 19 August 2026, Profitina's export and deletion functions existed but had no button. Describe a test that would have failed at that time and passed afterwards, and explain why the lab's unit tests could not detect the gap.

Exercise 11.11. A server verifies 1,000 logins per hour with PBKDF2 at 600,000 iterations (230 ms each on one core, **Table 11.1**). How much CPU time per hour does that cost? An attacker sends 100 login requests per second. What happens with and without "refuse before hashing"?

Exercise 11.12. Move lab G7's tables into SQLite (Python's `sqlite3` module) and write the erasure as one transaction of SQL statements. Show the tables after erasing Ayu from the history of the lab's last test.

11.15 Solutions to the Exercises

Solution 11.1. (a) General personal data: a full name is listed in Pasal 4. (b) Specific: biometric data. (c) Specific: personal financial data. (d) Not personal data: it is about a business, not a person. (e) Specific: health information, even when mentioned casually; it deserves the stricter care, and the app should consider whether it needs to store it at all or only a "use detergent B" flag. (f) General: religion is listed among the general categories.

Solution 11.2. (a) Performance of the contract: calling about an order is part of delivering the service the customer asked for. (b) A legal obligation of the controller. (c) Consent: marketing is not necessary for the contract, so it needs its own explicit, separate, withdrawable consent. (d) Legitimate interest: security logging protects the service and its users, provided it is limited to what is necessary and kept only briefly, and the balance with the subject's rights is documented.

Solution 11.3. $N = 26^{10} = 141\,167\,095\,653\,376 \approx 1.41 \times 10^{14}$. With SHA-256, $T = N/R \approx 1.41 \times 10^{14}/1.67 \times 10^{6} \approx 8.4 \times 10^{7}$ seconds, about 2.7 years on one core. With PBKDF2 at 600,000 iterations, $T \approx 1.41 \times 10^{14}/4.35 \approx 3.2 \times 10^{13}$ seconds, about a million years. The ratio is the same as in Section 11.5.2, about 3.8×10^{5}. Real attackers use graphics processors or dedicated hardware that compute SHA-256 and PBKDF2 far faster than a CPU core and in massive parallel, so the absolute times shrink dramatically for both; Argon2id's memory requirement limits how many guesses such hardware can run in parallel, which keeps the defender's advantage where it counts.

Solution 11.4. Without a salt, both users' stored strings are identical, so an attacker learns at once that they share a password; worse, the attacker can precompute hashes of millions of common passwords once and look up every user of every stolen table in that one list. With a 16-byte random salt per user, the two stored strings differ completely, identical passwords are invisible, and every guess must be recomputed separately for every user's salt, multiplying the attacker's work by the number of users.

Solution 11.5. $t = 0$, 10, 20: checked, wrong; the failure set becomes $\{0\}$, $\{0,10\}$, $\{0,10,20\}$. $t = 30$: $F = \{0,10,20\}$ has 3 elements, so refused before the password is checked, with retry_after $= 0 + 60 - 30 = 30$ seconds. $t = 61$: the window is $(1,61]$, so failure 0 has left; $F = \{10,20\}$, the attempt is checked, it is wrong, and F becomes $\{10,20,61\}$. $t = 65$: 3 failures in $(5,65]$, refused, retry_after $= 10 + 60 - 65 = 5$ seconds. $t = 71$: the window is $(11,71]$, $F = \{20,61\}$, checked, right password: signed in, and the account key is forgotten. Running the lab's limiter on the same sequence prints exactly these decisions.

Solution 11.6. Fail 3 times at 09:00:59 (allowed: first three of that minute) and 3 times at 09:01:00 (allowed: first three of the new minute): 6 failures in about 1 second. With the sliding window, after the three failures at 09:00:59 the set F contains three times within the last 60 seconds until 09:01:59, so every attempt until then is refused; only 3 failures occur.

Solution 11.7. Ayu's posts are 1, 3, 4 and 5. A post is kept as a shell only if a post *by someone else* has it as its parent. Post 1 has Budi's reply 2, so it becomes a `[deleted]` shell. Post 3 has no replies: deleted. Post 4 has a reply, but it is Ayu's own post 5: deleted, together with 5, which has no replies. Budi's post 2 is still attached to post 1, the shell; the thread reads "[deleted]" followed by Budi's answer. Budi's post 2 no longer has a reply, since Ayu's post 3 is gone.

Solution 11.8. **Listing 11.12** and **Listing 11.13** scan the consent log for the user's latest `given` event and compare its version with the current policy version; accepting appends a new event, so that the log keeps both.

Listing 11.12 `sol_policy.py`: asking for consent again after a policy change, on top of lab G7.

```
"""Exercise 11.8: ask for consent again when the policy version changes."""
from passwords import Pbkdf2Hasher
from privacy_app import PrivacyApp


def needs_consent(app, token):
    uid = app.user_of(token)
    given = [c["version"] for c in app.consents
             if c["subject"] == uid and c["event"] == "given"]
    return not given or given[-1] != app.policy_version


def accept_policy(app, token, ip):
    uid = app.user_of(token)
    app.consents.append({"subject": uid, "event": "given",
                         "version": app.policy_version,
                         "at": app.stamp(), "ip_hash": app.ip_hash(ip)})


clock = lambda: 1790215200
app = PrivacyApp(clock, b"s", hasher=Pbkdf2Hasher(iterations=1000))
app.register("ayu@example.com", "kopi-susu-tanpa-gula-2026", True, "203.0.113.7")
token = app.login("ayu@example.com", "kopi-susu-tanpa-gula-2026", "203.0.113.7")
print("needs consent under 1.0:", needs_consent(app, token))
app.policy_version = "1.1"
print("needs consent under 1.1:", needs_consent(app, token))
accept_policy(app, token, "203.0.113.7")
print("after accepting 1.1:   ", needs_consent(app, token))
print([c["version"] for c in app.consents])
```

```
needs consent under 1.0: False
needs consent under 1.1: True
after accepting 1.1:    False
['1.0', '1.1']
```

Listing 11.13 `SolPolicy.java`: the same in Java (the field `policyVersion` of `PrivacyApp` is deliberately not `final`).

```java
/** Exercise 11.8: ask for consent again when the policy version changes. */
public class SolPolicy {
    static boolean needsConsent(PrivacyApp app, String token) {
        String uid = String.valueOf(app.userOf(token));
        String last = null;
        for (PrivacyApp.Consent c : app.consents) {
            if (c.subject.equals(uid) && c.event.equals("given")) last = c.version;
        }
        return last == null || !last.equals(app.policyVersion);
    }

    static void acceptPolicy(PrivacyApp app, String token, String ip) {
        PrivacyApp.Consent c = new PrivacyApp.Consent();
        c.subject = String.valueOf(app.userOf(token));
        c.event = "given";
        c.version = app.policyVersion;
        c.at = app.stamp();
        c.ipHash = app.ipHash(ip);
        app.consents.add(c);
    }

    public static void main(String[] args) {
        PrivacyApp app = new PrivacyApp(() -> 1790215200L, "s".getBytes(),
                                        new Pbkdf2Hasher(1000), 3, 60);
        String pw = "kopi-susu-tanpa-gula-2026", ip = "203.0.113.7";
        app.register("ayu@example.com", pw, true, ip);
        String token = app.login("ayu@example.com", pw, ip);
        System.out.println("needs consent under 1.0: " + needsConsent(app, token));
        app.policyVersion = "1.1";
        System.out.println("needs consent under 1.1: " + needsConsent(app, token));
        acceptPolicy(app, token, ip);
        System.out.println("after accepting 1.1:    " + needsConsent(app, token));
    }
}
```

```
needs consent under 1.0: false
needs consent under 1.1: true
after accepting 1.1:    false
```

A real interface would call `needs_consent` after login and, if it returns true, show the new notice with an unticked box before anything else.

Solution 11.9. $3 \times 24 = 72$ hours after Monday 21:00 WIB is Thursday 21:00 WIB, the latest moment for the notices. Containment ends at Monday 23:00 (hour 2); assessment ends at Tuesday 09:00 (hour 12); the runbook's notification phase ends at Wednesday 21:00 (hour 48), a full day before the legal deadline, which is the margin a runbook should leave.

Solution 11.10. An end-to-end test drives a real browser: log in as a test user, open the *Pantau* tab, look for the *Akun & Data Pribadi* panel and its "export" and "delete" controls by their visible labels, press export, and check that a JSON file with the user's email is downloaded. Before 19 August 2026 it fails at the step "find the panel"; afterwards it passes. Unit tests like the lab's call `export` and `delete_account` directly, so they prove the functions work and say nothing about whether a person can reach them.

Solution 11.11. $1000 \times 0.23\ \text{s} = 230$ seconds of CPU per hour, about 6.4% of one core: affordable. At 100 requests per second, hashing every attempt would need $100 \times 0.23 = 23$ core-seconds per second, far more than a small server has, so honest logins would queue behind the attack and the service would stall. With "refuse before hashing", every request beyond the limit costs only a dictionary lookup and a queue trim, and the CPU stays free for real users. (A network-level limit in front of the application adds a further layer.)

Solution 11.12. One table per kind of record, and the erasure of **Listing 11.5** as SQL inside one transaction, so that either everything is erased or nothing is. **Listing 11.14** builds the history, erases Ayu (user 1) and prints what remains.

Listing 11.14 `sol_sqlite.py`: lab G7's tables in SQLite, with erasure as one transaction.

```
"""Exercise 11.12: the erasure of lab G7 as one SQLite transaction."""
import sqlite3

db = sqlite3.connect(":memory:")
db.executescript("""
CREATE TABLE users (id INTEGER PRIMARY KEY, email TEXT UNIQUE NOT NULL,
                    password TEXT NOT NULL, created TEXT NOT NULL);
CREATE TABLE posts (id INTEGER PRIMARY KEY, author INTEGER REFERENCES users(id),
                    text TEXT NOT NULL, parent INTEGER REFERENCES posts(id),
                    at TEXT NOT NULL);
CREATE TABLE consents (id INTEGER PRIMARY KEY, subject TEXT NOT NULL,
                       event TEXT NOT NULL, version TEXT NOT NULL,
                       at TEXT NOT NULL, ip_hash TEXT);
CREATE TABLE sessions (token TEXT PRIMARY KEY, user_id INTEGER NOT NULL);
INSERT INTO users VALUES (1, 'ayu@example.com', 'pbkdf2_sha256$...', 't0'),
                         (2, 'budi@example.com', 'pbkdf2_sha256$...', 't0');
INSERT INTO posts VALUES (1, 1, 'a question from ayu', NULL, 't1'),
                         (2, 1, 'a note nobody answered', NULL, 't1'),
                         (3, 2, 'an answer from budi', 1, 't2');
INSERT INTO consents VALUES (1, '1', 'given', '1.0', 't0', '2f7a7c6e655482f7'),
                            (2, '2', 'given', '1.0', 't0', 'aaaa');
INSERT INTO sessions VALUES ('tok-a', 1), ('tok-b', 2);
""")

def erase(db, uid, tomb, now):
    with db:                                    # one transaction: all or nothing
        db.execute("""UPDATE posts SET author = NULL, text = '[deleted]'
                      WHERE author = ? AND id IN (SELECT parent FROM posts
                      WHERE parent IS NOT NULL AND author IS NOT ?)""",
                   (uid, uid))
        db.execute("DELETE FROM posts WHERE author = ?", (uid,))
        db.execute("UPDATE consents SET subject = ?, ip_hash = NULL "
                   "WHERE subject = ?", (tomb, str(uid)))
        db.execute("INSERT INTO consents (subject, event, version, at) "
                   "VALUES (?, 'withdrawn', '1.0', ?)", (tomb, now))
        db.execute("DELETE FROM sessions WHERE user_id = ?", (uid,))
        db.execute("DELETE FROM users WHERE id = ?", (uid,))

erase(db, 1, "deleted-1", "t3")
for table in ("users", "posts", "consents", "sessions"):
    for row in db.execute("SELECT * FROM " + table):
        print(table, row)
```

```
users (2, 'budi@example.com', 'pbkdf2_sha256$...', 't0')
posts (1, None, '[deleted]', None, 't1')
posts (3, 2, 'an answer from budi', 1, 't2')
consents (1, 'deleted-1', 'given', '1.0', 't0', None)
consents (2, '2', 'given', '1.0', 't0', 'aaaa')
consents (3, 'deleted-1', 'withdrawn', '1.0', 't3', None)
sessions ('tok-b', 2)
```

Ayu's user row, her session, her unanswered note and her IP hash are gone; her question survives only as a shell because Budi's answer points to it; the consent log proves consent and withdrawal under the anonymous identifier `deleted-1`. The order of the first two statements matters: shells must be made before the remaining posts by the user are deleted.

References

[1] Profitina, *Profitina Whitepaper — Public Version v24*, Sep. 2026. [Online]. Available: https://app.profitina.com/static/whitepaper/

[2] Republik Indonesia, *Undang-Undang Nomor 27 Tahun 2022 tentang Pelindungan Data Pribadi*, Lembaran Negara 2022 No. 196, Oct. 17, 2022. [Online]. Available: https://peraturan.bpk.go.id/Details/229798/uu-no-27-tahun-2022

[3] Meridian Hukum, "Peraturan Pemerintah Nomor 33 Tahun 2026 tentang Peraturan Pelaksanaan Undang-Undang Nomor 27 Tahun 2022 tentang Pelindungan Data Pribadi" (LN 2026 No. 88, TLN No. 7190). [Online]. Available: https://meridianhukum.com/peraturan/pp-no-33-tahun-2026 (accessed Sep. 24, 2026).

[4] Bisnis.com, "Pembentukan Badan PDP dipercepat, implementasi UU PDP dinilai belum efektif," Sep. 13, 2026. [Online]. Available: https://teknologi.bisnis.com/read/20260913/84/2003698/pembentukan-badan-pdp-dipercepat-implementasi-uu-pdp-dinilai-belum-efektif

[5] CNBC Indonesia, "Lembaga perlindungan data belum ada di RI, padahal aturannya sudah ada," Sep. 16, 2026. [Online]. Available: https://www.cnbcindonesia.com/tech/20260916115609-37-768328/lembaga-perlindungan-data-belum-ada-di-ri-padahal-aturannya-sudah-ada

[6] A. Cavoukian, *Privacy by Design: The 7 Foundational Principles*. Toronto, ON, Canada: Information and Privacy Commissioner of Ontario, 2009 (rev. 2011).

[7] Profitina, "Kebijakan Privasi / Privacy Policy," version 1.15, effective Sep. 9, 2026. [Online]. Available: https://app.profitina.com/legal/privacy (accessed Sep. 24, 2026).

[8] National Institute of Standards and Technology, *Digital Identity Guidelines: Authentication and Authenticator Management*, NIST SP 800-63B-4, Aug. 2025.

[9] Password Hashing Competition, "Password Hashing Competition and our recommendation for hashing passwords: Argon2," 2015. [Online]. Available: https://www.password-hashing.net/

[10] A. Biryukov, D. Dinu, D. Khovratovich, and S. Josefsson, "Argon2 Memory-Hard Function for Password Hashing and Proof-of-Work Applications," IETF RFC 9106, Sep. 2021.

[11] OWASP Foundation, "Password Storage Cheat Sheet," *OWASP Cheat Sheet Series*. [Online]. Available: https://cheatsheetseries.owasp.org/cheatsheets/Password_Storage_Cheat_Sheet.html (accessed Sep. 24, 2026).

[12] K. Moriarty, B. Kaliski, and A. Rusch, "PKCS #5: Password-Based Cryptography Specification Version 2.1," IETF RFC 8018, Jan. 2017.

[13] Python Software Foundation, "hashlib — Secure hashes and message digests," *The Python Standard Library*. [Online]. Available: https://docs.python.org/3/library/hashlib.html

[14] Oracle, "Java Security Standard Algorithm Names," *Java SE 25 Documentation*. [Online]. Available: https://docs.oracle.com/en/java/javase/25/docs/specs/security/standard-names.html

[15] OWASP Foundation, "Credential Stuffing Prevention Cheat Sheet," *OWASP Cheat Sheet Series*. [Online]. Available: https://cheatsheetseries.owasp.org/cheatsheets/Credential_Stuffing_Prevention_Cheat_Sheet.html (accessed Sep. 24, 2026).

[16] Profitina, "Syarat dan Ketentuan / Terms and Conditions," version 1.1, effective Aug. 6, 2026. [Online]. Available: https://app.profitina.com/legal/terms (accessed Sep. 24, 2026).

[17] J. Arkko, M. Cotton, and L. Vegoda, "IPv4 Address Blocks Reserved for Documentation," IETF RFC 5737, Jan. 2010.

[18] H. Schlawack, "argon2-cffi: Argon2 for Python," version 25.1.0. [Online]. Available: https://argon2-cffi.readthedocs.io/

[19] C. Percival and S. Josefsson, "The scrypt Password-Based Key Derivation Function," IETF RFC 7914, Aug. 2016, sec. 11, "Test Vectors for PBKDF2 with HMAC-SHA-256."

Chapter 12
PROFIT: The Utility Coin on Solana

By the end of this chapter you will be able to say, in one careful sentence, what a utility coin is and what it is not, and why PROFIT belongs to the first kind and not to the second. You will know enough about Solana, the public blockchain on which PROFIT's contract lives, to read a token's settings in a block explorer and understand every field: the mint, the token accounts, the six decimals, the mint authority, and the freeze authority that PROFIT does not have. You will be able to separate, without hesitation, what is live in the September 2026 release (a devnet contract, earning, spending, transfers between users, buying through QRIS) from what is on the roadmap (mainnet, selling, a non-custodial exchange). And you will build Lab G8, a small but complete simulator in Python and Java that reproduces the public emission table of the Profitina whitepaper, derives its hidden power law, models circulating supply as a geometric series with a closed form, finds the point where burning overtakes emission, and plays with a toy version of the Rupiah floor. Every number in the lab is illustrative. None of it is financial advice, and nothing in this chapter predicts or promises a price.

12.1 Where We Left Off

Chapter 11 ended with a promise kept: the personal-data rights of UU PDP are not a paragraph in a policy but buttons a user can press, and the most important of them, erasure, cannot be triggered by anyone but the user. Before that, Chapter 8 explained how Profitina pays for itself with two tiers, Free and Pro, and it mentioned several times, briefly and with a warning box, a thing called PROFIT: users earn it by contributing, they can redeem it for Pro time, it is burned when spent, and "Chapter 12 explains why burning matters". This is Chapter 12.

PROFIT deserves its own chapter for two reasons that pull in opposite directions.

The first reason is that it is genuinely interesting to teach. A token with a decaying emission schedule and several burn mechanisms is a small dynamical system: a quantity that grows by one rule and shrinks by another, and whose long-run behavior can be computed exactly with nothing more than the formula for a geometric series. Anyone who has met recurrences in an algorithms course, or iterative methods in numerical methods, will recognize the mathematics immediately, and anyone who has not will meet it here gently, one step at a time.

The second reason is that it is genuinely easy to misunderstand. The words "coin", "token", "blockchain" and "Rupiah" appearing in the same paragraph make many readers think of speculation, of prices going up, of getting in early. That is precisely the reading this chapter is written to prevent. PROFIT in the September 2026 release is a utility coin on Solana's devnet, a public test network whose tokens have no real monetary value; it cannot be sold for Rupiah; and the public whitepaper describes it as "not a security, not electronic money, not a promise of returns" [1]. We will return to this sentence whenever it matters, not out of nervousness but because it is the single most useful fact in the chapter.

So the chapter moves in the order a careful reader needs. We first define a utility coin and place it in Indonesia's legal landscape (Section 12.2). We then learn just enough Solana to understand what "a devnet SPL token with six decimals and no freeze authority" means (Sections 12.3 and

12.4), and we list PROFIT's public facts (Section 12.5). The middle of the chapter is mathematics and code: emission (Section 12.6), burning (Section 12.7), supply as a geometric series (Section 12.8), the break-even point (Section 12.9), and the oracle floor (Section 12.10). The last part returns to the product: what a user can do with PROFIT and the guardrails around each action (Section 12.11), why the referral program has exactly one tier (Section 12.12), and a table that separates what is live from what is planned (Section 12.13).

12.2 What a Utility Coin Is, and What It Is Not

12.2.1 Hollow pride

Start with an ordinary evening on any social platform. Someone answers a stranger's question carefully, with a source. Someone else corrects a mistake in a popular post, politely. A third person reports a scam link before anyone clicks it. All three have done something useful for a community, and all three receive the same reward: a few likes, perhaps a follower, a small warm feeling that is gone by morning. Profitina's founding vision calls this *hollow pride*: a reward that is real but cannot be kept, spent or built upon.

PROFIT is Profitina's attempt to give that kind of contribution a reward with a little more substance. A helpful post, useful feedback that improves the local model (Chapter 7), a new channel that others actually use, a friend who joins and verifies their email: each earns a small amount of PROFIT, and PROFIT can be used inside the platform for real services. The tagline of the whole product, *Gather · Grow · Give*, says the same thing in three words.

Notice what this story does *not* say. It does not say that PROFIT will be worth a lot one day, or that early users will be rich, or that the coin is a way to invest in Profitina. It says that PROFIT is a way to recognize and use contributions. That is what makes it a utility coin.

> **DEFINITION.** A *utility coin* (or utility token) is a digital unit whose purpose is to be *used* inside a product: to pay for services, unlock features, or, eventually, to take part in governance. Its holders are users of a product, not owners of a business, and nothing about holding it entitles them to profits, interest or a future price.

12.2.2 Four things PROFIT is not

It helps to define PROFIT by what it is not, because each "not" rules out a specific misunderstanding. The public whitepaper uses these words, and the app repeats them [1], [2].

- **Not a security.** A security, in the everyday legal sense, is an investment in an enterprise with an expectation of profit from the efforts of others: a share, a bond, a unit in a fund. PROFIT gives no claim on Profitina's revenue, assets or decisions, and it is not offered as an investment.
- **Not electronic money.** Electronic money is a stored value denominated in Rupiah that can be used to pay third parties, like a balance in a licensed e-wallet. PROFIT is not denominated in Rupiah, cannot be used to pay anyone outside the platform, and cannot be converted back to Rupiah in the September 2026 release.
- **Not a promise of returns.** Nothing about PROFIT guarantees that its reference value will rise, stay the same, or even exist in a given form in the future.
- **Not a meme.** A meme coin exists to be traded on attention alone. PROFIT exists because the platform needs a unit for rewarding and paying for services, and its rules are designed to *slow down* the kind of rapid creation and trading that meme coins thrive on.

> **WARNING.** This chapter describes a product feature and teaches mathematics with illustrative numbers. It is not financial advice, not an offer to buy or sell anything, and not a prediction of any price. PROFIT in the September 2026 release is a Solana devnet token with no real monetary value [1].

12.2.3 The Indonesian frame

Money and crypto assets are regulated activities in Indonesia, and a responsible platform starts from the law, not from the technology. Three facts frame everything that follows.

First, the Rupiah is the currency for payments in Indonesia. The Currency Law, UU No. 7 of 2011, requires the Rupiah for payments and other obligations settled in money within the country [3]. A token cannot legally replace the Rupiah as a means of payment, and PROFIT does not try to: every purchase of a real service in Profitina is priced and paid either in Rupiah through QRIS or in PROFIT as a product credit, never as a substitute currency in the wider economy.

Second, crypto assets are supervised by the Financial Services Authority (OJK). Since 10 January 2025, under Government Regulation No. 49 of 2024, supervision of crypto assets moved from the commodity-futures regulator Bappebti to OJK, which regulates their trading as *digital financial assets* under OJK Regulation No. 27 of 2024 [4]. Any future step that would let PROFIT be traded for Rupiah therefore sits inside OJK's framework, and Bank Indonesia remains the regulator of payment systems such as QRIS.

Third, the order of events matters. Profitina's roadmap places mainnet and any form of exchange *after* a token audit and licensing, not before [1]. This is why selling PROFIT is disabled today, and why this chapter keeps saying "roadmap" about everything that involves turning PROFIT back into Rupiah.

> **KEY INSIGHT.** The law comes first and the design follows it. Disabling the sale of PROFIT until mainnet and regulatory clarity is not a missing feature; it is the feature that keeps a utility coin a utility coin.

12.3 Solana in Plain Words

12.3.1 A shared notebook that nobody can quietly edit

Imagine a notebook that records who owns what. Thousands of people around the world keep identical copies of it. When someone wants to add a line ("move 5 units from Ani's account to Budi's"), they sign the request with a secret key that only they hold, and the keepers of the notebook check the signature, check that Ani actually has 5 units, and add the line to their copies in the same order. Once enough of them agree, the line is final: to change it, one would have to persuade most of the keepers to rewrite history together, which the system is designed to make impractical.

That notebook is a *blockchain*, the keepers are *validators*, the signed requests are *transactions*, and the order-keeping protocol is *consensus*. Solana is one such public blockchain, designed to process many transactions quickly and cheaply. For a reward coin that hands out amounts such as 0.01 at a time, cheap and fast matters: a fee of several thousand Rupiah per transfer would make small rewards absurd.

12.3.2 Accounts, programs, transactions

Solana organizes its state as *accounts*. An account is a small record identified by a 32-byte public address, written in a text form called base58 (the long strings of letters and digits you see in explorers). Every account holds a balance of the native currency, SOL, and may hold data. Some accounts hold *programs*: code that runs on the chain. Programs on Solana are stateless; they read and write data held in *other* accounts that are passed to them.

A *transaction* is a list of instructions, each addressed to a program, together with the signatures of the accounts that approve it. When the Token Program (Section 12.4) receives an instruction "transfer 250,000 base units from account A to account B", it checks that the owner of A signed, that A holds enough, and then updates both accounts.

Fees are paid in *lamports*, the smallest unit of SOL; one SOL is 10^9 lamports. According to Solana's own documentation, the base fee is 5,000 lamports per signature, and it is split: **50% is burned and 50% goes to the validator**; optional *prioritization fees*, which a sender adds to get ahead in busy moments, go entirely to validators [5]. It is worth pausing on the first half of that rule. The network that PROFIT lives on already destroys part of every fee it collects. Burning is not an exotic idea invented for PROFIT; it is part of the plumbing of the chain itself, and Section 12.7 will meet it again.

12.3.3 Three clusters, one of them for practice

Solana runs as several separate public networks, called *clusters*, with the same software and different purposes. **Table 12.1** summarizes them in the words of the official documentation [6].

Table 12.1 Solana's three public clusters, as described in the official documentation in September 2026.

Cluster	Purpose	Are its tokens real?	Public RPC endpoint
Mainnet	The permissionless, persistent cluster for users, builders, validators and token holders	Yes: SOL issued on mainnet is real	`https://api.mainnet.solana.com`
Devnet	"A playground for anyone who wants to take Solana for a test drive, as a user, token holder, app developer, or validator"	No; a faucet hands out test SOL	`https://api.devnet.solana.com`
Testnet	Where core contributors stress-test new releases, focused on performance and validator behavior	No; a faucet hands out test SOL	`https://api.testnet.solana.com`

PROFIT's contract lives on **devnet**. This one fact explains most of what follows: the contract is real and public, anyone can inspect it, and it has no monetary value, because nothing on devnet does. Profitina's brand banner and its version string use the word "testnet" in the everyday sense of "not mainnet yet"; the precise name of the cluster is devnet, and this book uses the precise name.

12.3.4 The state of the network in 2026

A reader who remembers news of Solana outages from 2021 and 2022 deserves an update. As of August 2026, Solana's mainnet had run for **30 consecutive months, about 913 days, without a network-wide outage**; the last full halt was on 6 February 2024 and was resolved in under five hours [7]. The same report credits three changes for the streak: the QUIC networking protocol with stake-weighted quality of service, a fee market in which priority fees now make up about 88% of daily fee revenue, and client diversity, in particular Firedancer, an independent validator

client that went live on mainnet in late 2025 [7]. Having more than one independent implementation of a validator means that a single bug is less likely to stop the whole network.

The network also keeps changing. The official changelog of 18 September 2026 lists, among the feature gates for mainnet, *Transaction V1* (a new transaction format that, among other things, makes deploying programs about four times cheaper in fees) and a reduction of the *slot time* to 250 milliseconds; a slot is the short window during which one validator, the current leader, may produce a block [8]. The same changelog shows Agave and Firedancer client releases moving in step [8]. Versions and timings like these age within weeks; a reader should treat them as a snapshot of September 2026 and check the changelog for the present.

> **KEY INSIGHT.** For PROFIT, the properties of Solana that matter most are dull ones: fees that are tiny compared with a 0.01 reward, a separate devnet for honest testing, and a token standard that every wallet and explorer already understands. Speed records are nice; boring reliability is what a reward system needs.

12.4 Tokens on Solana: Mints, Accounts and Authorities

12.4.1 The mint and the token accounts

On Solana, a token such as PROFIT is not a program of its own. It is data managed by a shared, widely used program, the *Token Program*, which already knows how to create tokens, move them and burn them. Two kinds of account hold that data, and **Figure 12.1** shows how they relate.

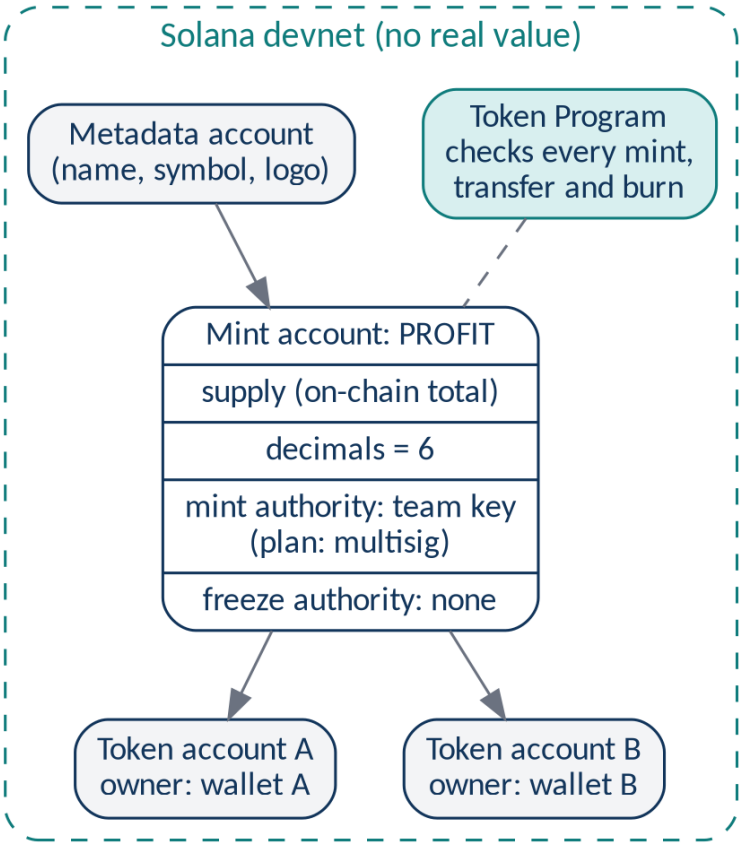


Figure 12.1 A token on Solana: one mint account describes the token (supply, decimals and two authorities), each holder has a token account pointing to the mint, and a metadata account gives it a name and logo. The Token Program enforces the rules for every change.

The **mint account** describes the token as a whole. The Solana documentation lists its fields: the *supply* (the total number of units in existence), the *decimals* (how many decimal places a unit has), the *mint authority* ("the account authorized to create new units of the token, increasing the supply") and the *freeze authority* ("the account authorized to freeze tokens in a Token Account, preventing them from being transferred or burned") [9].

A **token account** records how many units of one particular mint one particular owner holds. A wallet that holds three different tokens has three token accounts. The usual one for a given wallet and mint is the *associated token account*, whose address is derived from the owner's address and the mint's address, so that anyone can compute where a wallet's PROFIT lives without asking [9].

A third account, from the Metaplex *Token Metadata* program, attaches a name, a symbol and an image to the mint, so that wallets can display "PROFIT" with Profitina's logo instead of a bare address [10].

12.4.2 Decimals, and why amounts are whole numbers

The field that surprises beginners most is *decimals*. On the chain, every amount is a whole number of *base units*. The decimals field only says how to display that number to a human. If a mint has d decimals and a token account holds n base units, the amount a person sees is

$$a = \frac{n}{10^d} \tag{12.1}$$

PROFIT has $d = 6$, so one PROFIT is $10^6 = 1{,}000{,}000$ base units, and the smallest possible amount is 0.000001 PROFIT, which the whitepaper calls one µP (micro-PROFIT) [1]. A referral reward of 0.25 PROFIT is stored as the whole number 250,000; a base reward of 0.01 PROFIT is 10,000.

Why go to this trouble instead of storing 0.25 directly? Because computers store most fractional numbers in binary floating point, and binary cannot represent 0.01 or 0.1 exactly, just as a decimal expansion cannot represent 1/3 exactly. The error is tiny for one number, but a ledger adds millions of numbers, and tiny errors add up. The first step of Lab G8 makes this visible. **Listing 12.1** parses decimal strings into whole micro-PROFIT, formats them back, rejects an amount with seven decimals, and then adds 0.01 one hundred times in two ways.

Listing 12.1 `01_units.py`: PROFIT amounts as whole numbers of micro-PROFIT, with Equation (12.1) applied in both directions.

```
"""Lab G8, step 1: keep PROFIT amounts as whole numbers of micro-PROFIT.

A token with 6 decimals stores every amount as an integer count of its
smallest unit. We do the same, and we show why a float is the wrong tool.
"""

DECIMALS = 6
SCALE = 10 ** DECIMALS            # 1 PROFIT = 1,000,000 micro-PROFIT


def to_micro(text):
    """Parse a decimal string such as '0.25' into micro-PROFIT, exactly."""
    whole, _, frac = text.strip().partition(".")
    if len(frac) > DECIMALS:
        raise ValueError(f"{text!r} has more than {DECIMALS} decimals")
    if not (whole or "0").isdigit() or not (frac or "0").isdigit():
        raise ValueError(f"{text!r} is not a plain decimal number")
    return int(whole or "0") * SCALE + int(frac.ljust(DECIMALS, "0") or "0")


def to_text(micro):
    """Format micro-PROFIT back as a decimal string with 6 places."""
    return f"{micro // SCALE}.{micro % SCALE:06d}"


def main():
    print("1 PROFIT =", SCALE, "micro-PROFIT")
    for text in ["0.25", "0.01", "50", "0.000001"]:
        micro = to_micro(text)
        print(f"{text:>9} PROFIT -> {micro:>9} micro -> {to_text(micro)}")

    try:
        to_micro("0.0000001")
    except ValueError as err:
        print("rejected:", err)
```

Listing 12.1 (Continued) *01_units.py: PROFIT amounts as whole numbers of micro-PROFIT, with Equation (12.1) applied in both directions.*

```
    # One hundred rewards of 0.01 PROFIT, added two ways.
    as_float = 0.0
    as_micro = 0
    for _ in range(100):
        as_float += 0.01
        as_micro += to_micro("0.01")
    print("float sum of 100 x 0.01 =", as_float)
    print("micro sum of 100 x 0.01 =", to_text(as_micro))
    print("float sum == 1.0 ?", as_float == 1.0)
    print("micro sum == 1 PROFIT ?", as_micro == SCALE)


if __name__ == "__main__":
    main()
```

The Java version, **Listing 12.2**, does the same with `long` integers and prints exactly the same text.

Listing 12.2 `Units.java`: the same conversion in Java, with a `long` count of micro-PROFIT.

```
/** Lab G8, step 1: keep PROFIT amounts as whole numbers of micro-PROFIT. */
public class Units {
    static final int DECIMALS = 6;
    static final long SCALE = 1_000_000L;          // 1 PROFIT = 10^6 micro

    /** Parse a decimal string such as "0.25" into micro-PROFIT, exactly. */
    static long toMicro(String text) {
        String t = text.strip();
        int dot = t.indexOf('.');
        String whole = dot < 0 ? t : t.substring(0, dot);
        String frac = dot < 0 ? "" : t.substring(dot + 1);
        if (frac.length() > DECIMALS) {
            throw new IllegalArgumentException(
                "'" + text + "' has more than " + DECIMALS + " decimals");
        }
        if (!(whole.isEmpty() ? "0" : whole).matches("\\d+")
                || !(frac.isEmpty() ? "0" : frac).matches("\\d+")) {
            throw new IllegalArgumentException(
                "'" + text + "' is not a plain decimal number");
        }
        String padded = (frac + "000000").substring(0, DECIMALS);
        long w = whole.isEmpty() ? 0 : Long.parseLong(whole);
        return w * SCALE + Long.parseLong(padded);
    }

    /** Format micro-PROFIT back as a decimal string with 6 places. */
    static String toText(long micro) {
        return (micro / SCALE) + "." + String.format("%06d", micro % SCALE);
    }

    public static void main(String[] args) {
        System.out.println("1 PROFIT = " + SCALE + " micro-PROFIT");
        for (String text : new String[] {"0.25", "0.01", "50", "0.000001"}) {
            long micro = toMicro(text);
            System.out.printf("%9s PROFIT -> %9d micro -> %s%n",
                              text, micro, toText(micro));
        }
        try {
            toMicro("0.0000001");
        } catch (IllegalArgumentException err) {
            System.out.println("rejected: " + err.getMessage());
        }
        // One hundred rewards of 0.01 PROFIT, added two ways.
        double asFloat = 0.0;
        long asMicro = 0;
        for (int i = 0; i < 100; i++) {
```

Listing 12.2 (Continued) *Units.java: the same conversion in Java, with a long count of micro-PROFIT.*

```
            asFloat += 0.01;
            asMicro += toMicro("0.01");
        }
        System.out.println("float sum of 100 x 0.01 = " + asFloat);
        System.out.println("micro sum of 100 x 0.01 = " + toText(asMicro));
        System.out.println("float sum == 1.0 ? "
                           + (asFloat == 1.0 ? "True" : "False"));
        System.out.println("micro sum == 1 PROFIT ? "
                           + (asMicro == SCALE ? "True" : "False"));
    }
}
```

Both programs print:

```
1 PROFIT = 1000000 micro-PROFIT
     0.25 PROFIT ->    250000 micro -> 0.250000
     0.01 PROFIT ->     10000 micro -> 0.010000
       50 PROFIT ->  50000000 micro -> 50.000000
 0.000001 PROFIT ->         1 micro -> 0.000001
rejected: '0.0000001' has more than 6 decimals
float sum of 100 x 0.01 = 1.0000000000000007
micro sum of 100 x 0.01 = 1.000000
float sum == 1.0 ? False
micro sum == 1 PROFIT ? True
```

The last four lines are the lesson. Added one hundred times as a binary floating-point number, 0.01 does not make 1.0 but 1.0000000000000007, and the comparison with 1.0 is false. Added as the whole number 10,000, it makes exactly 1,000,000, which is exactly one PROFIT. The IEEE 754 standard that nearly every processor follows defines this behavior precisely [11]; it is not a bug in Python or Java but a property of binary fractions. A ledger that must never lose or invent a micro-PROFIT counts in whole base units, exactly as the chain does.

12.4.3 The two authorities

The two authorities in the mint account are where trust lives, and they deserve slow reading.

The **mint authority** can create new units. Whoever holds its key can increase the supply. For a reward coin, some mint authority must exist while rewards are still being issued; the question is who holds it and under what controls. A single key held by one person is a single point of failure: lose it and no more rewards can be issued, leak it and anyone can print coins. The standard answer on Solana is a *multisig*, an account that requires several of a set of keys to sign before it acts; Squads is a widely used multisig platform for managing Solana treasuries and tokens [12]. Profitina's stated plan is to move PROFIT's mint authority to a multisig. A mint authority can also be *revoked* entirely, setting it to none, after which no new units can ever be created; that is the natural end state for a token whose maximum supply has been fully distributed.

The **freeze authority** can freeze a holder's token account so that its tokens cannot be moved or burned [9]. It has legitimate uses (a regulated asset may be required to freeze accounts under a court order), and it is also a power to trap people's holdings at will. The freeze authority can be set to none when the mint is created, and once it is none, nobody can ever freeze a PROFIT account: not an attacker, not a court acting through the team, not the team itself.

PROFIT's freeze authority is none [1]. This is not a promise about intentions; it is a property of the deployed mint that anyone can check in a block explorer in a few seconds. Of all the facts in this chapter, it is the one that most directly protects holders, and it cost nothing to provide except the decision to give up a power.

> **EXAMPLE.** To check a token's settings yourself, open the Solana Explorer, switch the cluster selector to *Devnet*, and paste the mint address from **Table 12.2** into the search box. The page shows the current supply, the decimals, the mint authority and the freeze authority. For PROFIT the freeze authority field reads that there is none. Doing this once is worth more than reading this paragraph twice.

12.5 PROFIT's Public Facts

We can now read PROFIT's public description with understanding. **Table 12.2** collects what the public whitepaper, version 24 of September 2026, states about the coin, together with what the app shows [1], [2].

Table 12.2 PROFIT's public facts, from the public whitepaper v24 and the app, as of September 2026.

Property	Value in the September 2026 release
Nature	Utility coin: fuel for AI services and feature access, eventually governance; "not a meme, not a security"
Chain and standard	Solana, SPL token, Metaplex metadata; the logo is Profitina's logo
Cluster	Devnet; the contract is live and publicly verifiable
Devnet mint address	`6f1dNQPhYXPo78LVLGD2WpzBH2mzCm77BG9E8PFnWweL`
Decimals	6 (1 PROFIT = 1,000,000 µP)
Freeze authority	None
Mint authority	Planned move to a multisig
Supply	A locked maximum supply
Emission	Decaying: the reward multiplier halves every time the number of human users grows tenfold
Burns	Spend-burn, transfer-burn, buy-and-burn from QRIS revenue, anti-bot penalty burn
Oracle	Floor = Rupiah reserve ÷ circulating PROFIT; initial reference 1 PROFIT = Rp1,000, a model estimate, not a market price
Buying	Through QRIS, Rp50,000 to Rp1,000,000, credited at the oracle price after an administrator verifies the payment
Selling or redeeming for Rupiah	Disabled until mainnet and regulatory clarity
Mainnet	Roadmap: Phase 3, labeled Q4 2026, after a token audit and licensing

Three entries in the table deserve a sentence each before we move on.

The **cluster** entry is the foundation of everything else: the contract is real, but devnet tokens have no value [6], so nothing in the table describes money.

The **supply** entry says that the maximum supply is locked. A fixed ceiling means that emission, however it is scheduled, cannot go on forever; the schedule decides only how fast the ceiling is approached, and burning decides how much of what was emitted is still circulating.

The **oracle** entry uses the word "floor", and the word "model estimate" next to the Rupiah figure. Section 12.10 explains the floor formula and shows, with a small proof, why a purchase at the floor price does not move it. The Rupiah figure is a reference used by the platform to price its own services, not a quote anyone can trade at.

> **WARNING.** The mint address in **Table 12.2** is a devnet address. A token with the same name on mainnet, or any "PROFIT" offered for sale anywhere in the September 2026 release, is not Profitina's coin. The only official contact for questions is info@profitina.com.

12.6 Emission That Decays

12.6.1 The problem a decaying schedule solves

Suppose a platform paid every active user a fixed 1 PROFIT per day, forever. With 1,000 users it would create 1,000 PROFIT a day; with a million users, a million a day. The amount created would grow in exact proportion to the community, and with a locked maximum supply the ceiling would be reached in a hurry, leaving nothing to reward the people who arrive later. Worse, a fixed per-user reward invites fake users: every extra account, real or not, adds the same amount.

A *decaying* schedule answers both problems. It pays generously while the community is small and the platform needs its first contributors most, and it pays each person less as the community grows, so that the total created per day still grows, but much more slowly than the number of people.

> **DEFINITION.** *Emission* is the creation of new units of a token, here as rewards for contributions. An *emission multiplier* is a number between 0 and 1 by which every base reward is multiplied; a *decaying* emission schedule makes that multiplier shrink as some measure of the platform, here the number of human users, grows.

12.6.2 The public rule

The public whitepaper states the rule in words: the multiplier **halves every time the number of human users grows tenfold** [1]. It illustrates the rule with a small table, which we reproduce as **Table 12.3**.

Table 12.3 The whitepaper's illustrative emission table: the multiplier halves for every tenfold growth of users.

Human users	Multiplier	PROFIT per day at 1 PROFIT per user per day
1,000	0.125	125
100,000	0.03125	3,125
1,000,000	0.015625	15,625

To turn the words into a formula, let u be the number of human users and let $m(u)$ be the multiplier. "Halves for every tenfold growth" means that multiplying u by 10 multiplies m by $1/2$. The number of tenfold growths between a reference size u_0 and u is $\log_{10}(u/u_0)$, because $10^{\log_{10}(u/u_0)} = u/u_0$. So the multiplier has been halved $\log_{10}(u/u_0)$ times:

$$m(u) = (1/2)^{\log_{10}(u/u_0)} \tag{12.2}$$

Which reference u_0 reproduces the table? At $u = 1{,}000$ the multiplier is $0.125 = (1/2)^3$, so $\log_{10}(1000/u_0) = 3$, which gives $u_0 = 1$. Check the other rows: $\log_{10}100{,}000 = 5$ and $(1/2)^5 = 0.03125$; $\log_{10}1{,}000{,}000 = 6$ and $(1/2)^6 = 0.015625$. All three rows agree. From now on we take $u_0 = 1$ and write $m(u) = (1/2)^{\log_{10}u}$.

If each user earns a base of r PROFIT per day before the multiplier, the total created per day is

$$E(u) = r \cdot u \cdot m(u) \tag{12.3}$$

and with the table's $r = 1$: $E(1{,}000) = 1{,}000 \times 0.125 = 125$, $E(100{,}000) = 100{,}000 \times 0.03125 = 3{,}125$, $E(1{,}000{,}000) = 1{,}000{,}000 \times 0.015625 = 15{,}625$, exactly the right-hand column.

These are illustrations of the rule, not a forecast of Profitina's actual daily emission. Real rewards are small amounts for specific actions (Section 12.6.5), and the whitepaper's rule is the only part of the emission engine this book uses.

12.6.3 The hidden power law

Equation (12.3) hides a surprise that becomes visible with one identity about logarithms. For positive numbers a and u,

$$a^{\log_{10} u} = u^{\log_{10} a} \tag{12.4}$$

The proof is two lines. Take $\log_{10}$ of the left side: $\log_{10}(a^{\log_{10} u}) = \log_{10} u \cdot \log_{10} a$. Take $\log_{10}$ of the right side: $\log_{10}(u^{\log_{10} a}) = \log_{10} a \cdot \log_{10} u$. The two logarithms are the same product, and $\log_{10}$ is one-to-one, so the two sides are equal.

Apply Equation (12.4) with $a = 1/2$. Since $\log_{10} 1/2 = -\log_{10} 2$, we get $m(u) = u^{-\log_{10} 2}$, and therefore

$$E(u) = r \cdot u^{1-\log_{10} 2} = r \cdot u^{\alpha}, \qquad \alpha = 1 - \log_{10} 2 \approx 0.69897 \tag{12.5}$$

In plain words: the daily emission is a *power* of the number of users, with an exponent of about 0.7. It still grows as the community grows, but sublinearly. Three consequences follow at once.

- **Tenfold users, fivefold emission.** Multiplying u by 10 multiplies E by $10^{\alpha} = 10 \cdot 10^{-\log_{10} 2} = 10/2 = 5$. Each decade of growth creates five times as much per day, not ten times.
- **The gap with linear emission grows.** The ratio of linear emission u to decaying emission u^{α} is $u^{1-\alpha} = u^{\log_{10} 2} = 2^{\log_{10} u}$. At a million users that is $2^6 = 64$: the decaying schedule creates **64 times less** per day than a linear one, the figure the whitepaper quotes [1].
- **Per-user rewards fall slowly.** Each user's share is $m(u) = u^{-0.301}$. Doubling the community lowers each person's multiplier by the factor $2^{-0.301} \approx 0.81$, about 19%. Early contributors are rewarded more, but late ones are not shut out.

Figure 12.2 draws both schedules on logarithmic axes, where a power law becomes a straight line whose slope is the exponent: slope 1 for the linear schedule, slope $\alpha \approx 0.7$ for the decaying one.

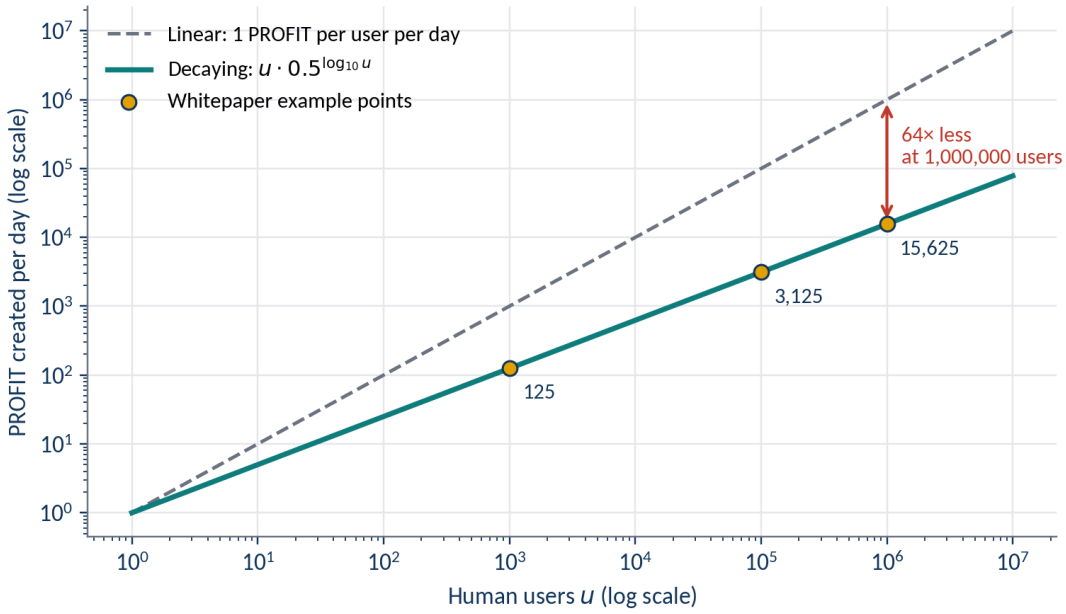


Figure 12.2 Daily emission under a linear schedule and under the whitepaper's decaying schedule, on logarithmic axes. The whitepaper's three example points lie on the decaying line; at a million users the gap is a factor of 64.

12.6.4 Lab G8, step 2: reproducing the table

The second program of the lab computes Equation (12.2), Equation (12.3) and Equation (12.5) for every power of ten from 1 to 10 million users, and prints the linear schedule and the ratio beside them. **Listing 12.3** is the Python version.

Listing 12.3 `02_emission.py`: the decaying multiplier of Equation (12.2), the daily emission of Equation (12.3) and its power-law form Equation (12.5).

```
"""Lab G8, step 2: a decaying emission multiplier (illustrative, not financial).

Public rule used here: the multiplier halves every time the number of human
users grows tenfold, m(u) = 0.5 ** log10(u / U0). With U0 = 1 and a base of
1 PROFIT per user per day, this reproduces the whitepaper's example table.
"""
import math

U0 = 1              # reference user count (the reader may change it)
BASE = 1.0          # PROFIT per user per day before the multiplier


def multiplier(users):
    return 0.5 ** math.log10(users / U0)


def emission(users):
    """PROFIT created per day when every user earns the base reward."""
    return users * BASE * multiplier(users)


def main():
    alpha = 1 - math.log10(2)
    print(f"exponent alpha = 1 - log10(2) = {alpha:.6f}")
    print(f"{'users':>10} {'multiplier':>11} {'decaying/day':>13}"
          f" {'linear/day':>11} {'ratio':>6} {'u^alpha':>11}")
    for power in range(0, 8):
        u = 10 ** power
        m = multiplier(u)
        e = emission(u)
        linear = u * BASE
        print(f"{u:>10,} {m:>11.7f} {e:>13,.3f} {linear:>11,.0f}"
              f" {linear / e:>6.0f} {u ** alpha:>11,.3f}")
    # Growth of the daily emission per tenfold growth of users.
    print(f"emission grows x{emission(10**6) / emission(10**5):.3f}"
          " when users grow x10")


if __name__ == "__main__":
    main()
```

Listing 12.4 is the Java version. The only care needed is to set the locale, so that the thousands separator is a comma on every machine; otherwise an Indonesian Windows installation would print 1.000.000 and the outputs would differ.

Listing 12.4 `Emission.java`: the same table in Java, formatted identically.

```java
import java.util.Locale;

/** Lab G8, step 2: a decaying emission multiplier (illustrative, not financial). */
public class Emission {
    static final double U0 = 1;         // reference user count
    static final double BASE = 1.0;     // PROFIT per user per day

    static double multiplier(double users) {
        return Math.pow(0.5, Math.log10(users / U0));
    }

    /** PROFIT created per day when every user earns the base reward. */
    static double emission(double users) {
        return users * BASE * multiplier(users);
    }

    public static void main(String[] args) {
        Locale.setDefault(Locale.US);
        double alpha = 1 - Math.log10(2);
        System.out.printf("exponent alpha = 1 - log10(2) = %.6f%n", alpha);
        System.out.printf("%10s %11s %13s %11s %6s %11s%n", "users",
            "multiplier", "decaying/day", "linear/day", "ratio", "u^alpha");
        for (int power = 0; power < 8; power++) {
            long u = (long) Math.pow(10, power);
            double m = multiplier(u);
            double e = emission(u);
            double linear = u * BASE;
            System.out.printf("%,10d %11.7f %,13.3f %,11.0f %6.0f %,11.3f%n",
                u, m, e, linear, linear / e, Math.pow(u, alpha));
        }
        // Growth of the daily emission per tenfold growth of users.
        System.out.printf("emission grows x%.3f when users grow x10%n",
            emission(1e6) / emission(1e5));
    }
}
```

Both print:

```
exponent alpha = 1 - log10(2) = 0.698970
     users  multiplier  decaying/day  linear/day  ratio     u^alpha
         1   1.0000000         1.000           1      1       1.000
        10   0.5000000         5.000          10      2       5.000
       100   0.2500000        25.000         100      4      25.000
     1,000   0.1250000       125.000       1,000      8     125.000
    10,000   0.0625000       625.000      10,000     16     625.000
   100,000   0.0312500     3,125.000     100,000     32   3,125.000
 1,000,000   0.0156250    15,625.000   1,000,000     64  15,625.000
10,000,000   0.0078125    78,125.000  10,000,000    128  78,125.000
emission grows x5.000 when users grow x10
```

Read the output column by column. The multiplier halves on every row, as the rule says. The rows for 1,000, 100,000 and 1,000,000 users reproduce **Table 12.3** exactly. The ratio column doubles on every row and reaches 64 at a million. And the last column, computed with the power u^{α} instead of the multiplier, agrees with the decaying column to every printed digit, which is Equation (12.5) confirmed numerically. The final line confirms the "tenfold users, fivefold emission" rule.

The multiplier is printed with seven decimals rather than six for a reason worth knowing. At ten million users it is 0.0078125 exactly, and rounding that to six decimals is a tie: Python rounds ties to even (0.007812) while Java's formatter rounds them up (0.007813). Printing seven decimals avoids the tie and keeps the two outputs identical. Small details like this are why the lab's two languages are always checked byte for byte.

12.6.5 From the schedule to real rewards

In the product, nobody receives “1 PROFIT per day”. People receive small amounts for specific, useful actions, each multiplied by the emission multiplier and by a local-AI usefulness score, and each capped. **Table 12.4** lists the earning actions and caps as the public whitepaper states them [1].

Table 12.4 Earning actions, base rewards and caps as listed in the public whitepaper. Every base reward is multiplied by the emission multiplier and by the usefulness score; daily caps reset at 00:00 WIB.

Action	Base reward (PROFIT)	Cap
Feedback on a scan	0.01 to 0.05	10 per day
A post	0.02	1
A reply, repost or like received	0.01	2 / 1 / 5
Creating a channel	0.05	2
A referral (both sides, after email verification)	0.25	once per referred person; 50 in total

Two rules from earlier chapters complete the picture. Only email-verified humans earn (Chapter 3), and the usefulness score gives trivial or bot-like posts 0.00, with escrow and clawback for anything later judged abusive; duplicate content earns nothing and links to the earlier post (Chapter 5). The daily caps reset at 00:00 WIB, the rule that Chapter 8 explained at length. The multiplier of Equation (12.2) is therefore not a salary but a dial: it turns every reward down a little each time the community grows tenfold.

12.7 Four Ways PROFIT Leaves: The Burns

12.7.1 What burning means

To *burn* a token is to destroy units of it permanently. On Solana the Token Program has a burn instruction that removes units from a token account and reduces the mint’s supply by the same amount; after a burn, those units exist nowhere. Burning is the opposite of minting, and it is the only way the circulating amount of a fixed-supply token can shrink.

Why would a platform destroy its own reward coin? Because emission and burning together decide how much PROFIT circulates, and a reward that is created freely but never destroyed only piles up. The whitepaper’s design gives PROFIT four burns [1], shown as flows in **Figure 12.3**.

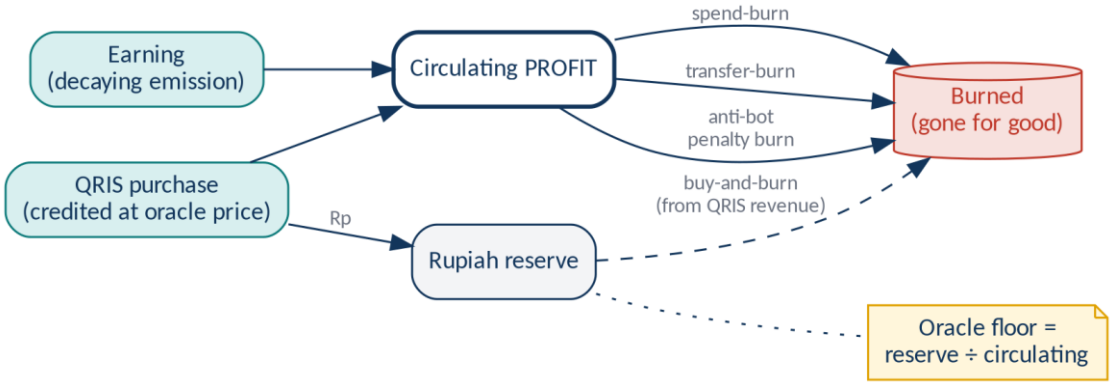


Figure 12.3 How PROFIT enters and leaves circulation. Earning and QRIS purchases add to circulation; four burns remove PROFIT permanently; the oracle floor divides the Rupiah reserve by what circulates.

12.7.2 The four burns

1. **Spend-burn.** When a user spends PROFIT on a service, such as a scan top-up or redeeming Pro time, the PROFIT spent is burned rather than moved into a company wallet. This follows the idea that Ethereum introduced with EIP-1559, in which the base fee of every transaction "is always burned" [13], and that Solana applies to half of each base fee [5]: using the system consumes some of its unit.
2. **Transfer-burn.** Since users can send PROFIT to one another (Section 12.11), each transfer carries a small fee, and the fee is burned. A transfer fee that went to the platform would be a toll; a burned fee is a small brake on churning coins back and forth.
3. **Buy-and-burn.** A share of real Rupiah revenue from QRIS is used, by design, to remove PROFIT from circulation. This ties part of the burning to the platform's actual commercial activity rather than to token activity alone.
4. **Anti-bot penalty burn.** Rewards held in escrow for activity later judged to be bot-like or abusive are clawed back and burned, not redistributed. Cheating therefore does not merely fail to pay; it removes what it tried to take.

The whitepaper adds an invariant for the mature system: **once mature, burn is at least emission**, $B \geq E$ [1]. In a young system with few users and little spending, emission dominates, as it must if anyone is to earn anything. The design goal is that as usage grows, spending and transfers grow with the number of people, while emission grows only as $u^{0.7}$, so that burning eventually catches up. Section 12.9 computes when, under simple assumptions.

> **KEY INSIGHT.** Burning is not a trick to raise a price. In a utility coin it is bookkeeping with a purpose: every burn is tied to a real use (spending, transferring, revenue, or catching abuse), so the amount in circulation follows the amount of real activity.

12.8 Supply as a Geometric Series

12.8.1 A model simple enough to solve

We now build a model of circulating supply that is simple enough to solve exactly and rich enough to teach something. Make three assumptions, each a deliberate simplification.

1. The number of users is fixed, so the daily emission E is a constant (for 100,000 users under Equation (12.3), $E = 3{,}125$ PROFIT per day).
2. Each day, a fixed fraction b of whatever is circulating is burned through spending and transfers. A burn rate of $b = 0.005$ means that half a percent of the circulating supply is used up each day.
3. The day starts with the burn and ends with the emission, and circulation starts at zero.

Let S_t be the circulating supply at the end of day t, with $S_0 = 0$. Then each day

$$S_{t+1} = (1 - b)\, S_t + E \tag{12.6}$$

This is a *first-order linear recurrence*: tomorrow's value is a fixed multiple of today's plus a constant. Before solving it, compute a few days by hand with $E = 3{,}125$ and $b = 0.005$, so $1 - b = 0.995$:

- $S_1 = 0.995 \times 0 + 3{,}125 = 3{,}125$
- $S_2 = 0.995 \times 3{,}125 + 3{,}125 = 3{,}109.375 + 3{,}125 = 6{,}234.375$
- $S_3 = 0.995 \times 6{,}234.375 + 3{,}125 = 6{,}203.203125 + 3{,}125 = 9{,}328.203125$

Each day adds a little less than 3,125, because the burn grows with the supply.

12.8.2 Unrolling the recurrence

Write $q = 1 - b$ to keep the algebra short and apply Equation (12.6) to itself:

$$S_t = qS_{t-1} + E = q(qS_{t-2} + E) + E = q^2 S_{t-2} + qE + E$$

Continue until S_0 appears. After t steps,

$$S_t = q^t S_0 + E\,(1 + q + q^2 + \cdots + q^{t-1})$$

The bracket is a *geometric series*: a sum in which each term is the previous term times the same ratio q. Its closed form is one of the most useful identities in mathematics [14]. Call the sum G and multiply it by q:

$$G = 1 + q + \cdots + q^{t-1}, \qquad qG = q + q^2 + \cdots + q^t$$

Subtract the second line from the first. Every term except the first of G and the last of qG cancels, leaving $G - qG = 1 - q^t$, so for $q \neq 1$

$$\sum_{k=0}^{t-1} q^k = \frac{1 - q^t}{1 - q} \tag{12.7}$$

With $S_0 = 0$ and $1 - q = b$, the supply after t days is

$$S_t = E \cdot \frac{1 - (1 - b)^t}{b} \tag{12.8}$$

Check it against the hand computation: $S_3 = 3{,}125 \times (1 - 0.995^3)/0.005 = 625{,}000 \times (1 - 0.985074875) = 625{,}000 \times 0.014925125 = 9{,}328.203125$. The closed form and the step-by-step computation agree to the last digit.

12.8.3 The steady state and how fast it is reached

Because $0 < 1 - b < 1$, the power $(1 - b)^t$ shrinks toward zero as t grows, and Equation (12.8) approaches a limit:

$$S_\infty = \lim_{t \to \infty} S_t = \frac{E}{b} \tag{12.9}$$

This *steady state* has a direct meaning: it is the supply at which the daily burn $b \cdot S$ exactly equals the daily emission E. Below it, emission wins and the supply rises; above it, burning wins and the supply falls. With $E = 3{,}125$ and $b = 0.005$, $S_\infty = 625{,}000$ PROFIT. Doubling the burn rate halves the steady state; halving the emission halves it too.

How long until the supply is close to the steady state? The fraction of the steady state reached after t days is $S_t / S_\infty = 1 - (1 - b)^t$. Setting this equal to 0.9 and solving for t with logarithms gives the day on which 90% is first reached:

$$t_{90} = \left\lceil \frac{\ln 0.1}{\ln(1 - b)} \right\rceil \tag{12.10}$$

For $b = 0.005$: $\ln 0.1 = -2.302585$ and $\ln 0.995 = -0.005012542$, so the quotient is 459.36 and $t_{90} = 460$ days. A small burn rate means a high steady state that is approached slowly; a large one means a low steady state reached quickly.

12.8.4 Lab G8, step 3: a ledger in micro-PROFIT against the closed form

The third program simulates Equation (12.6) day by day as a ledger would, in whole micro-PROFIT, and compares it with Equation (12.8). The burn is computed with integer division, because a ledger cannot burn a fraction of a base unit; that is the one place where the simulation and the formula must differ, and the program measures by how much. It also checks a conservation law that every honest ledger satisfies: on every day, supply equals everything emitted minus everything burned. **Listing 12.5** is the Python version.

Listing 12.5 `03_supply_sim.py`: the recurrence Equation (12.6) run as an integer ledger, checked against the closed form Equation (12.8), with the steady state Equation (12.9) and t_{90} from Equation (12.10).

```
"""Lab G8, step 3: circulating supply under daily emission and burning.

Illustrative, not financial. Every day the platform emits E PROFIT and burns
a fixed share b of what is circulating. The ledger works in whole micro-PROFIT;
the closed form S_t = E * (1 - (1 - b)**t) / b checks it.
usage: python 03_supply_sim.py [users] [burn_bps] [days]
"""
import math
import sys

SCALE = 1_000_000                       # micro-PROFIT per PROFIT
CHECKPOINTS = [0, 30, 90, 180, 365, 730]


def half_up(x):
    """Round half away from zero, the same way in Python and Java."""
    return math.floor(x + 0.5)


def daily_emission_micro(users):
    """Whitepaper example rule: 1 PROFIT/user/day x 0.5**log10(users)."""
    profit = users * 0.5 ** math.log10(users)
    return half_up(profit * SCALE)


def simulate(e_micro, burn_bps, days):
    """Return lists of supply, total emitted and total burned, in micro."""
    supply, emitted, burned = [0], [0], [0]
    s = em = bu = 0
    for _ in range(days):
        burn = s * burn_bps // 10_000   # floor: a ledger cannot burn a fraction
        s = s - burn + e_micro
        em += e_micro
        bu += burn
        supply.append(s)
        emitted.append(em)
        burned.append(bu)
    return supply, emitted, burned


def closed_form(e, b, t):
    return e * (1 - (1 - b) ** t) / b
```

Listing 12.5 (Continued) *03_supply_sim.py: the recurrence Equation (12.6) run as an integer ledger, checked against the closed form Equation (12.8), with the steady state Equation (12.9) and t{90} from Equation (12.10).*

```
def bar(value, top, width=40):
    return "#" * half_up(width * value / top)


def main():
    users = int(sys.argv[1]) if len(sys.argv) > 1 else 100_000
    burn_bps = int(sys.argv[2]) if len(sys.argv) > 2 else 50
    days = int(sys.argv[3]) if len(sys.argv) > 3 else 730
    e_micro = daily_emission_micro(users)
    e, b = e_micro / SCALE, burn_bps / 10_000
    steady = e / b
    print(f"users={users:,}  emission E={e:,.6f} PROFIT/day"
          f"  burn b={b:.4f}/day  steady state E/b={steady:,.0f}")
    supply, emitted, burned = simulate(e_micro, burn_bps, days)
    print(f"{'day':>4} {'ledger supply':>18} {'closed form':>18}"
          f" {'gap (micro)':>11}")
    for t in CHECKPOINTS:
        if t > days:
            break
        exact = closed_form(e, b, t)
        gap = supply[t] - half_up(exact * SCALE)
        print(f"{t:>4} {supply[t] / SCALE:>18,.6f} {exact:>18,.6f}"
              f" {gap:>11,}")
    print("conservation: supply == emitted - burned on every day:",
          all(s == em - bu for s, em, bu in zip(supply, emitted, burned)))
    print(f"after {days} days: emitted {emitted[-1] / SCALE:,.0f},"
          f" burned {burned[-1] / SCALE:,.0f}")
    print("supply as a share of the steady state (each # = 2.5%):")
    for t in range(0, days + 1, 73):
        print(f"{t:>4} |{bar(supply[t] / SCALE, steady):<40}|"
              f" {100 * supply[t] / SCALE / steady:5.1f}%")
    print("how the burn rate shapes the curve:")
    for bps in (20, 50, 100):
        rate = bps / 10_000
        t90 = math.ceil(math.log(0.1) / math.log(1 - rate))
        print(f"  b={rate:.4f}: steady state {e / rate:>12,.0f} PROFIT,"
              f" 90% reached on day {t90}")


if __name__ == "__main__":
    main()
```

Two details in the listing deserve comment. The helper `half_up` rounds halves away from zero with `floor(x + 0.5)`, because Python's built-in `round` rounds ties to even while Java's `Math.round` rounds them up, and the two versions must agree. And the program accepts three optional arguments (users, burn rate in basis points, days), so the reader can try other scenarios without editing the code. **Listing 12.6** is the Java version.

Listing 12.6 `SupplySim.java`: the same ledger simulation in Java, with arrays of `long` micro-PROFIT.

```java
import java.util.Locale;

/**
 * Lab G8, step 3: circulating supply under daily emission and burning.
 * Illustrative, not financial. usage: java SupplySim [users] [burn_bps] [days]
 */
public class SupplySim {
    static final long SCALE = 1_000_000L;            // micro-PROFIT per PROFIT
    static final int[] CHECKPOINTS = {0, 30, 90, 180, 365, 730};

    /** Round half away from zero, the same way in Python and Java. */
    static long halfUp(double x) {
        return (long) Math.floor(x + 0.5);
    }

    /** Whitepaper example rule: 1 PROFIT/user/day x 0.5^log10(users). */
    static long dailyEmissionMicro(long users) {
        double profit = users * Math.pow(0.5, Math.log10(users));
        return halfUp(profit * SCALE);
    }

    static double closedForm(double e, double b, int t) {
        return e * (1 - Math.pow(1 - b, t)) / b;
    }

    static String bar(double value, double top) {
        return "#".repeat((int) halfUp(40 * value / top));
    }

    public static void main(String[] args) {
        Locale.setDefault(Locale.US);
        long users = args.length > 0 ? Long.parseLong(args[0]) : 100_000;
        int burnBps = args.length > 1 ? Integer.parseInt(args[1]) : 50;
        int days = args.length > 2 ? Integer.parseInt(args[2]) : 730;
        long eMicro = dailyEmissionMicro(users);
        double e = (double) eMicro / SCALE, b = burnBps / 10_000.0;
        double steady = e / b;
        System.out.printf("users=%,d  emission E=%,.6f PROFIT/day  burn b=%.4f"
            + "/day  steady state E/b=%,.0f%n", users, e, b, steady);
```

Listing 12.6 (Continued) *SupplySim.java: the same ledger simulation in Java, with arrays of long micro-PROFIT.*

```
        long[] supply = new long[days + 1];
        long[] emitted = new long[days + 1];
        long[] burned = new long[days + 1];
        for (int t = 1; t <= days; t++) {
            long s = supply[t - 1];
            long burn = s * burnBps / 10_000;   // floor: no fractional burn
            supply[t] = s - burn + eMicro;
            emitted[t] = emitted[t - 1] + eMicro;
            burned[t] = burned[t - 1] + burn;
        }
        System.out.printf("%4s %18s %18s %11s%n",
            "day", "ledger supply", "closed form", "gap (micro)");
        for (int t : CHECKPOINTS) {
            if (t > days) break;
            double exact = closedForm(e, b, t);
            long gap = supply[t] - halfUp(exact * SCALE);
            System.out.printf("%4d %,18.6f %,18.6f %,11d%n",
                t, (double) supply[t] / SCALE, exact, gap);
        }
        boolean conserved = true;
        for (int t = 0; t <= days; t++) {
            conserved &= supply[t] == emitted[t] - burned[t];
        }
        System.out.println("conservation: supply == emitted - burned on every"
            + " day: " + (conserved ? "True" : "False"));
        System.out.printf("after %d days: emitted %,.0f, burned %,.0f%n", days,
            (double) emitted[days] / SCALE, (double) burned[days] / SCALE);
        System.out.println("supply as a share of the steady state"
            + " (each # = 2.5%):");
        for (int t = 0; t <= days; t += 73) {
            double v = (double) supply[t] / SCALE;
            System.out.printf("%4d |%-40s| %5.1f%%%n",
                t, bar(v, steady), 100 * v / steady);
        }
        System.out.println("how the burn rate shapes the curve:");
        for (int bps : new int[] {20, 50, 100}) {
            double rate = bps / 10_000.0;
            long t90 = (long) Math.ceil(Math.log(0.1) / Math.log(1 - rate));
            System.out.printf("  b=%.4f: steady state %,12.0f PROFIT,"
                + " 90%% reached on day %d%n", rate, e / rate, t90);
        }
    }
}
```

With no arguments (100,000 users, a burn of 50 basis points, that is 0.5% per day, for 730 days), both print:

```
users=100,000  emission E=3,125.000000 PROFIT/day  burn b=0.0050/day  steady state
E/b=625,000
 day      ledger supply        closed form gap (micro)
   0           0.000000           0.000000           0
  30      87,259.880067      87,259.880053          14
  90     226,931.983853     226,931.983816          37
 180     371,466.967245     371,466.967187          58
 365     524,699.178701     524,699.178622          79
 730     608,903.592463     608,903.592370          93
conservation: supply == emitted - burned on every day: True
after 730 days: emitted 2,281,250, burned 1,672,346
supply as a share of the steady state (each # = 2.5%):
   0 |                                        |   0.0%
  73 |############                            |  30.6%
 146 |####################                    |  51.9%
 219 |############################            |  66.6%
 292 |################################        |  76.9%
 365 |####################################    |  84.0%
 438 |########################################|  88.9%
 511 |########################################|  92.3%
 584 |########################################|  94.6%
 657 |########################################|  96.3%
 730 |########################################|  97.4%
how the burn rate shapes the curve:
  b=0.0020: steady state    1,562,500 PROFIT, 90% reached on day 1151
  b=0.0050: steady state      625,000 PROFIT, 90% reached on day 460
  b=0.0100: steady state      312,500 PROFIT, 90% reached on day 230
```

The output has five parts, and each confirms something.

- The first line computes E from the users and $S_\infty = E/b = 625{,}000$ from Equation (12.9).
- The table compares the ledger with the closed form. The *gap* column, in micro-PROFIT, is never negative and grows slowly to 93 micro-PROFIT after two years: the ledger always floors the burn, so it burns a fraction of a micro-PROFIT less each day than the formula, and the leftover fractions accumulate. Ninety-three millionths of a PROFIT on a supply of 608,903 is a relative difference of about 1.5×10^{-10}. The formula and the ledger agree, and the program says precisely how closely.
- The conservation line is `True`: on each of the 731 days, supply equals emitted minus burned, to the micro-PROFIT.
- The bar chart shows the supply as a share of the steady state every 73 days. It rises quickly at first and flattens, reaching 84.0% at one year and 97.4% at two.
- The last three lines show the steady state and t_{90} for three burn rates. At $b = 0.005$, $t_{90} = 460$, as computed by hand above.

The same program with other arguments explores other worlds. Here it is for a million users and a burn of 1% per day over one year (`python 03_supply_sim.py 1000000 100 365`, or `java SupplySim 1000000 100 365`):

```
users=1,000,000  emission E=15,625.000000 PROFIT/day  burn b=0.0100/day  steady state
E/b=1,562,500
 day      ledger supply        closed form gap (micro)
   0           0.000000           0.000000           0
  30     406,718.166592     406,718.166581          11
  90     930,106.292719     930,106.292690          29
 180   1,306,550.047376   1,306,550.047331          45
 365   1,522,628.180592   1,522,628.180543          49
conservation: supply == emitted - burned on every day: True
after 365 days: emitted 5,703,125, burned 4,180,497
supply as a share of the steady state (each # = 2.5%):
   0 |                                        |   0.0%
  73 |####################                    |  52.0%
 146 |###############################         |  76.9%
 219 |####################################    |  88.9%
 292 |########################################  |  94.7%
 365 |########################################## |  97.4%
how the burn rate shapes the curve:
  b=0.0020: steady state    7,812,500 PROFIT, 90% reached on day 1151
  b=0.0050: steady state    3,125,000 PROFIT, 90% reached on day 460
  b=0.0100: steady state    1,562,500 PROFIT, 90% reached on day 230
```

Tenfold users made the emission five times larger, 15,625 instead of 3,125, as Equation (12.5) predicts, and the steady state is $15{,}625/0.01 = 1{,}562{,}500$. The faster burn also brings the curve close to that steady state much sooner: 97.4% of it after one year, where the slower burn of the first run needed two.

Figure 12.4 draws the model for three burn rates, from the same integer ledger, over three years.

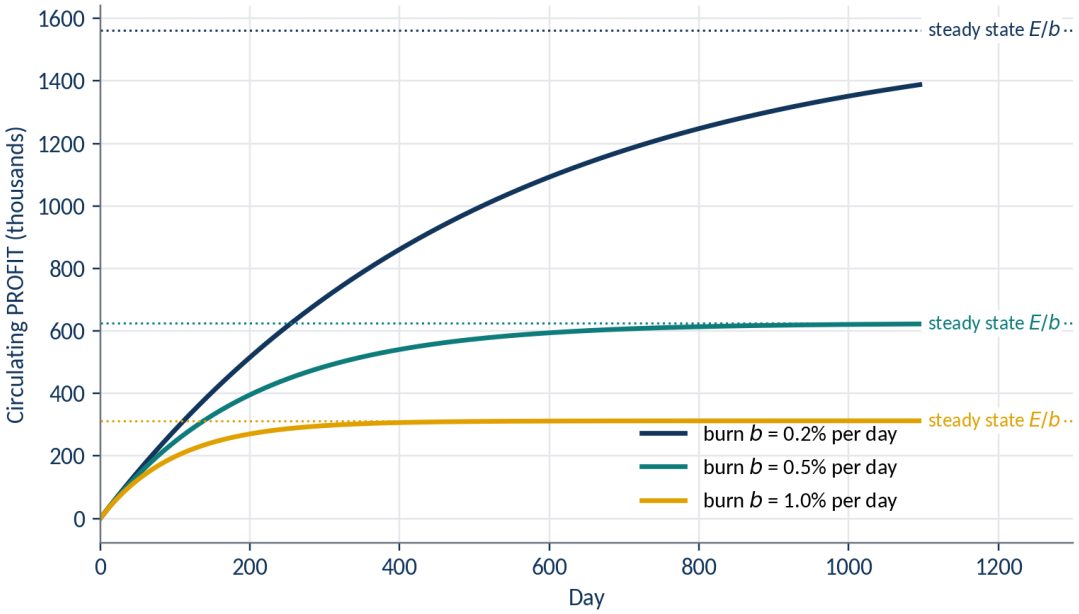


Figure 12.4 Circulating supply under a constant emission of 3,125 PROFIT per day and three daily burn rates, from the integer ledger of Lab G8. Each curve approaches its own steady state E/b; a higher burn rate gives a lower steady state that is reached sooner. Illustrative values only.

WARNING. The model has fixed users, a fixed emission and a burn that is a fixed share of supply. Real platforms have none of these. The model is useful because it can be solved exactly and reveals the *shape* of the dynamics; it does not forecast PROFIT's actual supply, and nothing in it concerns a price.

12.9 When Does Burning Overtake Emission?

12.9.1 Per user, the question becomes simple

Section 12.7 quoted the design invariant for the mature system: burn at least equal to emission. When does a growing platform get there? The steady-state model of Section 12.8 fixed the number of users; now we let it vary and ask a different question.

Suppose that, on average, each active user causes β PROFIT to be burned per day through spending and transfer fees. Then the daily burn grows in proportion to the users, $B(u) = \beta u$, while the daily emission grows as $E(u) = u \cdot m(u)$ by Equation (12.3) with $r = 1$. Divide both by u: burn per user is the constant β, emission per user is $m(u) = u^{-\log_{10}2}$, which falls as the community grows. So burning overtakes emission exactly when

$$m(u) \le \beta \Leftrightarrow u^{-\log_{10}2} \le \beta \Leftrightarrow u \ge u^* = \beta^{-1/\log_{10}2} \tag{12.11}$$

The last step raises both sides of $u^{-\log_{10}2} \le \beta$ to the power $-1/\log_{10}2$, which is negative and therefore reverses the inequality. Since $1/\log_{10}2 \approx 3.3219$, the break-even user count is $u^* = \beta^{-3.3219}$.

Work one case by hand. For $\beta = 0.02$ PROFIT per user per day: $\log_{10}0.02 = -1.69897$, and $-1.69897 \times (-3.3219) = 5.6439$, so $u^* = 10^{5.6439} \approx 440{,}409$ users. Below about 440 thousand users, emission exceeds burning; above it, burning wins.

There is a pleasant consequence of the power law. At $u^*/10$ the per-user emission is twice β (one halving fewer), and at $10u^*$ it is half of β. So the ratio of burn to emission is 0.5 one decade below the break-even point, 1 at it, and 2 one decade above, whatever β is.

12.9.2 Lab G8, step 4: closed form, bisection and a toy oracle

The fourth program computes u^* twice: once by Equation (12.11) and once by *bisection*, the simplest reliable method for solving an equation numerically [15]. Bisection keeps an interval in which the answer is known to lie, looks at the midpoint, and throws away the half that cannot contain the answer; after 60 halvings of an interval of width 12 (in powers of ten), the remaining width is $12/2^{60} \approx 10^{-17}$, far below what a `double` can distinguish. Solving on $x = \log_{10}u$ rather than on u itself keeps the interval small and the steps even. Having two independent ways to get the same number is the numerical analyst's version of a test: if they disagree, one of them is wrong.

The same program also runs the toy oracle of Section 12.10, so we list it here in full and discuss the second half of its output in that section. **Listing 12.7** is the Python version.

Listing 12.7 `04_breakeven_oracle.py`: the break-even user count by Equation (12.11) and by bisection, and a toy oracle floor.

```
"""Lab G8, step 4: where burning overtakes emission, and a toy oracle floor.

Illustrative, not financial. Per user and per day, the platform emits
m(u) = 0.5 ** log10(u) PROFIT and burns beta PROFIT through spending.
Burn >= emission once m(u) <= beta, i.e. u >= beta ** (-1 / log10(2)).
"""
import math

SCALE = 1_000_000                          # micro-PROFIT per PROFIT
LOG2 = math.log10(2)


def per_user_emission(users):
    return 0.5 ** math.log10(users)


def break_even_closed(beta):
    return beta ** (-1 / LOG2)


def break_even_bisect(beta, lo=0.0, hi=12.0, steps=60):
    """Bisection on x = log10(u) for m(10**x) - beta = 0."""
    for _ in range(steps):
        mid = (lo + hi) / 2
        if per_user_emission(10 ** mid) > beta:
            lo = mid                       # still emitting more than burning
        else:
            hi = mid
    return 10 ** hi


def show_break_even():
    print("per-user burn beta -> users u* where burn >= emission")
    print(f"  {'beta':>6} {'u* closed form':>15} {'u* bisection':>15}"
          f" {'burn/emission at u*/10':>23} {'u*':>5} {'10u*':>5}")
    for beta in (0.005, 0.01, 0.02, 0.05):
        u_star = break_even_closed(beta)
        u_bis = break_even_bisect(beta)
        r = [beta / per_user_emission(u_star * k) for k in (0.1, 1, 10)]
        print(f"  {beta:>6} {u_star:>15,.0f} {u_bis:>15,.0f}"
              f" {r[0]:>23.2f} {r[1]:>5.2f} {r[2]:>5.2f}")


def floor_price(reserve_rp, circulating_micro):
    return reserve_rp * SCALE / circulating_micro
```

Listing 12.7 (Continued) *04_breakeven_oracle.py: the break-even user count by Equation (12.11) and by bisection, and a toy oracle floor.*

```
def show_oracle():
    print("toy oracle: floor = rupiah reserve / circulating PROFIT")
    reserve = 100_000_000                  # rupiah
    circ = 100_000 * SCALE                 # micro-PROFIT
    steps = [("start", 0, 0)]
    steps.append(("community emission +5,000", 0, 5_000 * SCALE))
    steps.append(("spend-burn -3,000", 0, -3_000 * SCALE))
    steps.append(("QRIS purchase Rp1,000,000", 1_000_000, None))
    steps.append(("transfer-burn -200", 0, -200 * SCALE))
    print(f"  {'event':<27} {'reserve (Rp)':>14} {'circulating':>14}"
          f" {'floor (Rp)':>10}")
    for name, add_rp, add_micro in steps:
        if add_micro is None:              # buy at the current floor price
            add_micro = add_rp * circ // reserve
        reserve += add_rp
        circ += add_micro
        print(f"  {name:<27} {reserve:>14,} {circ / SCALE:>14,.2f}"
              f" {floor_price(reserve, circ):>10,.2f}")


def main():
    show_break_even()
    show_oracle()


if __name__ == "__main__":
    main()
```

Listing 12.8 is the Java version. The array of `Long` objects, with `null` for the purchase step, mirrors Python's `None`: it marks the one event whose PROFIT amount is not given but computed from the current floor.

Listing 12.8 `BreakEvenOracle.java`: the same two computations in Java.

```
import java.util.Locale;

/**
 * Lab G8, step 4: where burning overtakes emission, and a toy oracle floor.
 * Illustrative, not financial. Burn >= emission once 0.5^log10(u) <= beta,
 * i.e. u >= beta^(-1 / log10 2).
 */
public class BreakEvenOracle {
    static final long SCALE = 1_000_000L;          // micro-PROFIT per PROFIT
    static final double LOG2 = Math.log10(2);

    static double perUserEmission(double users) {
        return Math.pow(0.5, Math.log10(users));
    }

    static double breakEvenClosed(double beta) {
        return Math.pow(beta, -1 / LOG2);
    }

    /** Bisection on x = log10(u) for m(10^x) - beta = 0. */
    static double breakEvenBisect(double beta) {
        double lo = 0.0, hi = 12.0;
        for (int i = 0; i < 60; i++) {
            double mid = (lo + hi) / 2;
            if (perUserEmission(Math.pow(10, mid)) > beta) {
                lo = mid;                  // still emitting more than burning
            } else {
                hi = mid;
            }
        }
        return Math.pow(10, hi);
    }

    static void showBreakEven() {
        System.out.println(
            "per-user burn beta -> users u* where burn >= emission");
        System.out.printf("  %6s %15s %15s %23s %5s %5s%n", "beta",
            "u* closed form", "u* bisection", "burn/emission at u*/10", "u*",
            "10u*");
        for (double beta : new double[] {0.005, 0.01, 0.02, 0.05}) {
            double uStar = breakEvenClosed(beta);
            double uBis = breakEvenBisect(beta);
            double[] r = new double[3];
            double[] k = {0.1, 1, 10};
            for (int i = 0; i < 3; i++) {
                r[i] = beta / perUserEmission(uStar * k[i]);
```

Listing 12.8 (Continued) *BreakEvenOracle.java: the same two computations in Java.*

```
            }
            System.out.printf("  %6s %,15.0f %,15.0f %23.2f %5.2f %5.2f%n",
                Double.toString(beta), uStar, uBis, r[0], r[1], r[2]);
        }
    }

    static double floorPrice(long reserveRp, long circulatingMicro) {
        return (double) reserveRp * SCALE / circulatingMicro;
    }

    static void showOracle() {
        System.out.println(
            "toy oracle: floor = rupiah reserve / circulating PROFIT");
        long reserve = 100_000_000L;              // rupiah
        long circ = 100_000L * SCALE;             // micro-PROFIT
        String[] names = {"start", "community emission +5,000",
            "spend-burn -3,000", "QRIS purchase Rp1,000,000",
            "transfer-burn -200"};
        long[] addRp = {0, 0, 0, 1_000_000L, 0};
        Long[] addMicro = {0L, 5_000L * SCALE, -3_000L * SCALE, null,
            -200L * SCALE};
        System.out.printf("  %-27s %14s %14s %10s%n",
            "event", "reserve (Rp)", "circulating", "floor (Rp)");
        for (int i = 0; i < names.length; i++) {
            long add = addMicro[i] == null
                ? addRp[i] * circ / reserve       // buy at the current floor
                : addMicro[i];
            reserve += addRp[i];
            circ += add;
            System.out.printf("  %-27s %,14d %,14.2f %,10.2f%n", names[i],
                reserve, (double) circ / SCALE, floorPrice(reserve, circ));
        }
    }

    public static void main(String[] args) {
        Locale.setDefault(Locale.US);
        showBreakEven();
        showOracle();
    }
}
```

Both print:

```
per-user burn beta -> users u* where burn >= emission
    beta  u* closed form    u* bisection  burn/emission at u*/10    u*  10u*
   0.005      44,040,900      44,040,900                    0.50  1.00  2.00
    0.01       4,404,090       4,404,090                    0.50  1.00  2.00
    0.02         440,409         440,409                    0.50  1.00  2.00
    0.05          20,986          20,986                    0.50  1.00  2.00
toy oracle: floor = rupiah reserve / circulating PROFIT
  event                        reserve (Rp)    circulating floor (Rp)
  start                         100,000,000     100,000.00   1,000.00
  community emission +5,000     100,000,000     105,000.00     952.38
  spend-burn -3,000             100,000,000     102,000.00     980.39
  QRIS purchase Rp1,000,000     101,000,000     103,020.00     980.39
  transfer-burn -200            101,000,000     102,820.00     982.30
```

The first table shows the closed form and bisection agreeing to the last printed digit for four burn rates, and the three ratio columns are 0.50, 1.00 and 2.00 on every row, as predicted. The numbers also show how sensitive u^* is to β: halving the per-user burn from 0.01 to 0.005 multiplies the break-even community by $2^{3.32} = 10$. **Figure 12.5** draws the same thing: the falling line of per-user emission and three horizontal lines of per-user burn, crossing at the break-even points.

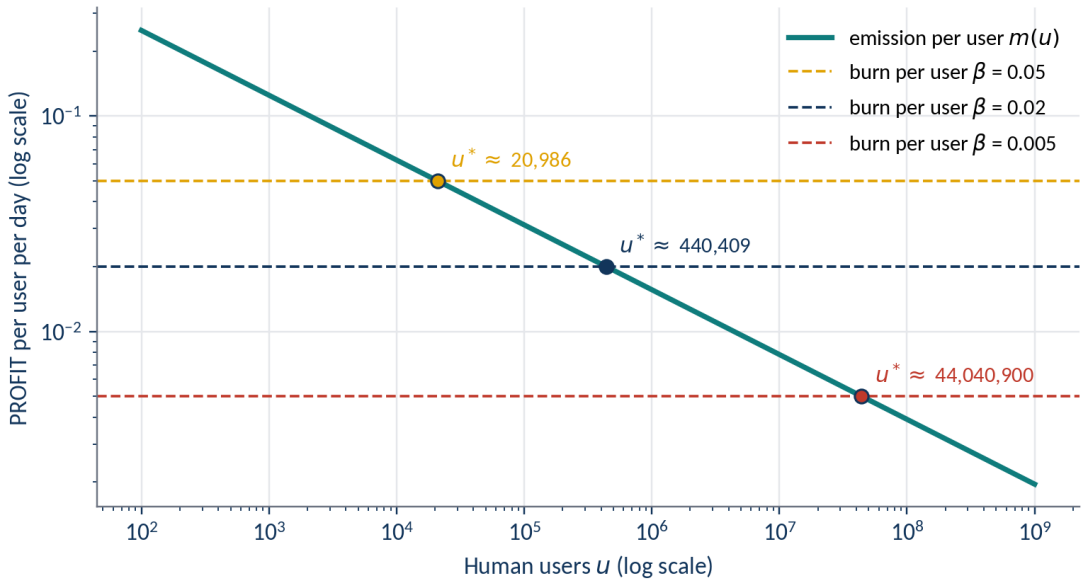


Figure 12.5 Per-user emission $m(u)$ falls as the community grows; a constant per-user burn β is a horizontal line. Burning overtakes emission to the right of each crossing u^*. Illustrative values of β.

WARNING. The values of β in the lab are the reader's inputs, chosen to make the mathematics visible. They are not measurements of Profitina, and the break-even points are not predictions of when anything will happen to PROFIT.

12.10 The Oracle: A Floor, Not a Price

12.10.1 What the oracle computes

A platform that sells PROFIT for Rupiah through QRIS, and prices some services in PROFIT, needs a reference for how many Rupiah one PROFIT is worth *inside the platform*. Profitina calls the mechanism that provides it an *oracle*, and the public whitepaper states its formula: the floor is the Rupiah reserve divided by the circulating PROFIT [1]. Writing R for the reserve in Rupiah and C for the circulating supply,

$$f = \frac{R}{C} \tag{12.12}$$

The initial reference is **1 PROFIT = Rp1,000**, described in the whitepaper and in the app as a "model estimate, not a market price or promise" [1], [2]. During testnet the value is stabilized at or above the initial reference, and the app displays a 7-day moving average with a safeguard against manipulation; two-way pricing belongs to mainnet [2].

The word *floor* is chosen with care. If every circulating PROFIT could be exchanged for its share of the reserve, each would receive exactly f Rupiah; the floor is what the reserve covers, not what anyone would pay. And since selling is disabled, nobody can in fact exchange PROFIT for Rupiah in the September 2026 release. The floor is a unit of account for the platform's own prices.

12.10.2 What moves the floor, and what does not

Equation (12.12) has only two ingredients, so only two kinds of event move it.

- **Emission lowers the floor**, because C grows while R stays.

- **Burns raise the floor**, because C shrinks while R stays.
- **A purchase at the floor price leaves the floor unchanged.** A user who pays x Rupiah at the floor price $f = R/C$ receives x/f PROFIT. The reserve becomes $R + x$ and the supply $C + x/f$, so the new floor is

$$f' = \frac{R + x}{C + x/f} = \frac{R + x}{C + xC/R} = \frac{R + x}{C\,(R + x)/R} = \frac{R}{C} = f \tag{12.13}$$

In words: buying at the floor adds reserve and supply in exactly the proportion that the floor already has. A purchase therefore cannot be used to push the floor up or down, which is one reason the design credits purchases at the oracle price.

The second half of the lab output walks through five events on a toy platform with Rp100,000,000 in reserve and 100,000 PROFIT circulating, so the floor starts at the reference Rp1,000:

- Community emission of 5,000 PROFIT raises C to 105,000 and lowers the floor to $100{,}000{,}000/105{,}000 = 952.38$.
- A spend-burn of 3,000 lowers C to 102,000 and raises the floor to 980.39.
- A QRIS purchase of Rp1,000,000 at the floor price adds 1,020 PROFIT (computed in whole micro-PROFIT) and Rp1,000,000 of reserve. The floor stays at 980.39, as Equation (12.13) promises.
- A transfer-burn of 200 raises the floor to 982.30.

Every one of these numbers is invented for the lesson. The point is the direction of each arrow, not its size.

> **WARNING.** The floor is not a market price, not a redemption value, and not a promise of any future value. In the September 2026 release there is no market in which PROFIT can be sold for Rupiah at this or any other price [1].

12.11 Buying, Spending and Transferring: The Guardrails

A coin that can be earned, bought, spent and sent to others has four doors, and each door needs a lock that keeps the coin a utility. This section walks through the doors in the order a user meets them.

12.11.1 Buying through QRIS

QRIS is Bank Indonesia's national QR standard for payments, and by mid-2026 it had 65.77 million users and 44.86 million merchant acceptance points, 96.68% of them micro, small and medium enterprises [16]. A user who wants PROFIT chooses an amount between Rp50,000 and Rp1,000,000, pays by scanning the QRIS code with any participating bank or e-wallet app, and receives PROFIT at the oracle price once an administrator verifies the payment [1], [2]. Chapter 8 explained why a person verifies purchases at this stage: purchase volume is small, the rule "measure interest before building payment" applies, and a manual check is cheap to build and easy to audit.

Buying is the only door through which Rupiah enters. There is no door through which Rupiah leaves: **selling or redeeming PROFIT for Rupiah is disabled** until mainnet and regulatory clarity [1].

12.11.2 Spending on real services

PROFIT can be spent on real services in the app: extra scan quota for the day (a top-up is valid for the purchase day only, in WIB), the Academic Citation Scan, and Pro time (Chapter 8). Prices are configured by the platform and shown before the user confirms; a purchase that fails is refunded; spent PROFIT is burned [1]. Around spending, the whitepaper sets four guardrails, and each deserves a sentence.

1. **No minting from spending.** Spending never creates new PROFIT anywhere; a spend only moves units to the burn.
2. **Purchases never raise rank.** Leaderboards measure contribution, weighted toward quality (Chapter 4). Buying or spending PROFIT cannot buy a better position.
3. **No bypass of Fair-Play or safety.** No amount of PROFIT unlocks posting without the checks of Chapter 5: Link Safety, the usefulness score, duplicate detection, moderation.
4. **Rights stay free forever.** Exporting data, correcting it and deleting an account (Chapter 11), and every safety feature, are never priced in PROFIT or in Rupiah.

There is also a use that does not spend at all: creating a channel requires a **balance of at least 50 PROFIT, which is never deducted** (Chapter 4). The balance is a signal of participation, not a fee.

12.11.3 Transferring between users

Since late August 2026 users can send PROFIT to each other by handle [1]. Four rules shape the door.

- **A small fee, burned.** Each transfer carries a small fee, and the fee is burned (the transfer-burn of Section 12.7).
- **Only organically earned PROFIT can be transferred.** PROFIT that was bought with Rupiah, or granted by an administrator, cannot be sent to another user. The reason is a loophole this closes: if bought PROFIT could be transferred, one person could pay Rupiah, send the PROFIT to a second person, and the second could use it as if it were money received from the first, turning the platform into an informal channel for moving Rupiah value between people. Restricting transfers to earned PROFIT keeps transfers a way of sharing recognition, not a payment rail.
- **A public transfer log.** Transfers appear in a public log, with names masked for anonymous visitors. Openness about the flow of a reward coin discourages collusion and makes patterns visible to everyone.
- **Escrow for unverified recipients.** PROFIT sent to a user who has not yet verified an email is held in escrow until they do, consistent with the rule that only verified humans take part in the economy.

12.11.4 Administrator grants, in public

Administrators can grant PROFIT at their discretion for contests, collaboration or compensation. Grants have per-transaction and daily caps and are drawn from a **lifetime pool capped at 0.01% of the maximum supply**; their history is public, and granted PROFIT is excluded from every leaderboard [1]. A power that can create value for chosen people is exactly the kind of power that should be small, capped and visible, and this design makes it all three.

> **KEY INSIGHT.** Every guardrail answers the same question: could this door be used to turn PROFIT into a way of moving money or buying status? When the answer is yes, the door is narrowed until the answer is no.

12.12 One Tier of Referral, and Why Not More

12.12.1 The rule

A user who invites a friend receives **0.25 PROFIT, and so does the friend**, after the friend verifies their email; each pair earns this once, for life, and a person can be rewarded for at most 50 referrals [1]. A separate Referral Leaderboard shows who has brought people in. That is the whole program: one tier, a small symmetric reward, paid only for a verified human.

12.12.2 Why a second tier would be a different thing

Many referral programs pay not only for the friends you invite but also for the friends *they* invite, and so on down several levels. Profitina deliberately does not, and the whitepaper gives the reason in plain words: multi-level tiers are "structurally similar to pyramid schemes" [1].

The mathematics shows why. Suppose every participant recruits k new people. At level 0 there is one person; at level 1 there are k; at level 2, k^2; at level n, k^n. The number of people needed to fill n levels below one person is again a geometric series, by Equation (12.7):

$$\sum_{i=1}^{n} k^i = k\,\frac{k^n - 1}{k - 1} \tag{12.14}$$

With $k = 5$ and only ten levels, that is $5 \times (5^{10} - 1)/4 = 12{,}207{,}030$ people, more people than live in the city of Jakarta; with $k = 10$ and ten levels, it is over eleven billion, more than the population of the Earth. A reward structure that pays people for the recruitment of their recruits can only keep paying while this exponential tree keeps growing, and it must stop growing long before the tenth level. Those who join late, at the bottom, pay for those at the top. That is the definition of a pyramid, whatever it is called, and Indonesia's Trade Law, UU No. 7 of 2014, prohibits distributors from using a pyramid scheme [17].

A one-tier referral grows linearly with the people each user personally invites, is capped, and pays both sides the same small amount. It rewards introducing a friend to a useful product, and nothing else.

> **EXAMPLE.** Ani invites Budi, and Budi verifies his email: Ani and Budi each receive 0.25 PROFIT (times the usual multipliers). Budi then invites Citra, who verifies: Budi and Citra each receive 0.25 PROFIT. Ani receives nothing for Citra. If Ani invites Budi again with a second account, nothing happens: the pair has already been rewarded, and a second account for the same person is exactly the kind of activity the Fair-Play checks of Chapter 5 are built to catch.

12.13 Live Today, and on the Roadmap

Everything about PROFIT that involves value must be sorted into two boxes that are never confused. **Table 12.5** does the sorting for the September 2026 release.

Table 12.5 What is live and what is on the roadmap for PROFIT, as of September 2026.

Capability	Status in the September 2026 release
PROFIT SPL token on Solana devnet, publicly verifiable	Live (since late July 2026)
6 decimals; freeze authority none	Live, properties of the deployed mint
Earning with a decaying multiplier, caps and the usefulness score	Live
Spend-burn, transfer-burn, anti-bot penalty burn	Live
Buy-and-burn from QRIS revenue	Part of the design; meaningful only with real revenue
Oracle floor with the Rp1,000 initial reference, 7-day average shown in the app	Live, as a model estimate
Buying PROFIT through QRIS (Rp50,000 to Rp1,000,000), verified by an administrator	Live
Spending on scan top-ups, the Academic Citation Scan and Pro time	Live
Transfers between users, earned PROFIT only, public log, escrow	Live (since late August 2026)
One-tier referral, 0.25 PROFIT each side	Live (since August 2026)
Administrator grants from a capped, public pool	Live
Mint authority moved to a multisig	Planned
Selling or redeeming PROFIT for Rupiah	Disabled until mainnet and regulatory clarity
PROFIT on Solana mainnet	Roadmap: Phase 3, labeled Q4 2026, after a token audit and licensing
Ultralight DEX: non-custodial swaps through the Jupiter aggregator	Roadmap: Phase 3, same gate
Governance by PROFIT holders	Stated as an eventual use; no design published

Two lines in the table need a word of honesty that the table cannot hold.

The first is buy-and-burn. The mechanism is part of the design, but its size depends on revenue, and Chapter 8 recorded the August 2026 finding plainly: zero PROFIT purchases across the whole production history at that time [1]. A burn fed by revenue is only as large as the revenue that feeds it.

The second is the Phase 3 date. “Q4 2026” is the label in the whitepaper, and the whitepaper itself attaches the gate: an audit of the token and licensing under the regulators of Section 12.2.3 [1]. A gate is a condition, not a calendar. If the conditions are not met by a date, the date moves; the conditions do not.

> **DEFINITION.** A *non-custodial* exchange lets two parties swap tokens directly from their own wallets, through on-chain programs, without handing their tokens to an intermediary that holds them. An *aggregator* such as Jupiter searches several such venues and routes a swap through the best available path.

12.14 Profitina in Practice

Everything in this chapter corresponds to the live product as follows, in the September 2026 release.

- **Where the user sees PROFIT.** The app has a PROFIT area that shows the user’s balance, the current oracle reference with its 7-day moving average and the caveat “model estimate, not a

market price", the QRIS purchase form, the list of services that can be paid in PROFIT with their prices, the transfer form and the public transfer log [2]. Rewards appear in the dashboard's notifications as they are credited (Chapter 4).

- **Where the chain appears.** The devnet mint of **Table 12.2** is the on-chain face of the token, and anyone can inspect it in the Solana Explorer with the cluster set to devnet [18]. The platform's ledger records every reward, spend, burn and transfer in whole micro-PROFIT, the same unit the mint uses.
- **How the rules are enforced.** Earning passes through the email-verification gate, the daily caps that reset at 00:00 WIB, the usefulness score and duplicate detection (Chapters 3, 5 and 8). Spending passes through the four guardrails of Section 12.11. Transfers pass through the earned-only rule, the burned fee and the public log.
- **What stays human.** Buying through QRIS is credited after an administrator verifies the payment; administrator grants are capped and public; product decisions, including any change to the rules of the economy, are taken by people and published in the whitepaper, whose version 24 is current in September 2026 [1].
- **What stays out.** No price chart, no "buy now" banner, no promise. The only numbers the app shows next to Rupiah are the purchase amounts and the model estimate, each with its caveat.

Lab G8 reproduces none of Profitina's code or configuration. It uses only the rule the whitepaper publishes (the multiplier halves for every tenfold growth of users) and the reader's own inputs for everything else. A reader who wants to practice the same ideas on a product of their own can take Profitina Laundry, the teaching case study of this book series: a loyalty point that customers earn per kilogram washed, that halves its earning rate as the customer base grows tenfold, and that is burned when redeemed for a free ironing, is the same system in miniature, with the same geometric series and the same guardrails.

12.15 Common Misunderstandings

MISUNDERSTANDING 1: "PROFIT is an investment in Profitina." It is not. PROFIT is a utility coin: it rewards contributions and pays for services inside the platform. It gives no share in the company, no claim on revenue and no promise of returns [1].

MISUNDERSTANDING 2: "The Rp1,000 figure is PROFIT's price." It is a model estimate used as an initial reference for the oracle floor of Equation (12.12). There is no market for PROFIT, and selling is disabled until mainnet and regulatory clarity.

MISUNDERSTANDING 3: "Devnet means the token is fake." The contract is real, public and verifiable; what devnet means is that tokens there have no monetary value. That is the purpose of a development cluster [6].

MISUNDERSTANDING 4: "A decaying emission means rewards stop." Emission keeps growing with the community, as $u^{0.7}$ by Equation (12.5); what decays is each user's multiplier. Ten times the users means five times the daily emission, not zero.

MISUNDERSTANDING 5: "Burning exists to push the price up." In a utility coin, each burn is tied to a real use: spending, transferring, revenue or caught abuse. Burning keeps circulation in step with activity; the oracle floor moves as a consequence, and the floor is not a price.

MISUNDERSTANDING 6: "Buying lots of PROFIT raises the floor." A purchase at the floor price leaves the floor exactly where it was, as Equation (12.13) proves.

MISUNDERSTANDING 7: "Transfers let me pay a friend." Only organically earned PROFIT can be transferred, so transfers share recognition, not Rupiah value. Bought and granted PROFIT cannot be sent.

MISUNDERSTANDING 8: "More referral levels would just be more generous." More levels create an exponential tree, Equation (12.14), that cannot keep growing; the late joiners pay for the early ones. One tier is a design choice, not a limitation.

MISUNDERSTANDING 9: "Floating-point numbers are fine for balances." They are not: 0.01 added a hundred times is not 1.0 in binary floating point. Ledgers count whole base units, as Equation (12.1) and Lab G8 show.

12.16 Chapter Summary

- A utility coin is used inside a product; it is not a security, not electronic money, not a promise of returns and not a meme. PROFIT is a utility coin, and in the September 2026 release it is a Solana devnet SPL token with no monetary value [1].
- In Indonesia the Rupiah is the currency for payments [3], and crypto assets have been supervised by OJK since 10 January 2025 [4]. PROFIT's mainnet and any exchange come only after a token audit and licensing; selling is disabled until then.
- Solana stores state in accounts, runs stateless programs, and charges fees in lamports; half of each base fee is burned [5]. Devnet is its public practice cluster [6]. As of August 2026 mainnet had run 30 months without a network-wide outage [7].
- A token is a mint account (supply, decimals, mint authority, freeze authority) plus token accounts [9]. Amounts are whole base units, displayed by Equation (12.1); PROFIT has 6 decimals. Its freeze authority is none; its mint authority is planned to move to a multisig.
- Emission decays: the multiplier Equation (12.2) halves for every tenfold growth of users, which reproduces the whitepaper's table (**Table 12.3**). By Equation (12.4), daily emission is the power law Equation (12.5) with exponent $\alpha \approx 0.699$: tenfold users give fivefold emission, and 64 times less than linear at a million users.
- Four burns remove PROFIT: spend-burn, transfer-burn, buy-and-burn and the anti-bot penalty burn; the mature design aims at burn at least equal to emission.
- With constant emission and a proportional burn, supply follows Equation (12.6), whose solution via the geometric series Equation (12.7) is Equation (12.8); it approaches the steady state Equation (12.9), reaching 90% after t_{90} days by Equation (12.10). An integer ledger agrees with the closed form to within a few dozen micro-PROFIT and conserves every unit.
- Per user, burning overtakes emission beyond u^* of Equation (12.11); bisection confirms the closed form.
- The oracle floor is Equation (12.12). Emission lowers it, burns raise it, and a purchase at the floor leaves it unchanged, Equation (12.13). It is a model estimate, not a price.
- Guardrails: buying via QRIS verified by a person; spending with four guardrails; transfers of earned PROFIT only, with a burned fee, a public log and escrow; grants from a capped public pool; one-tier referral, because multi-level structures grow like Equation (12.14).
- **Table 12.5** separates what is live from the roadmap: mainnet and the Ultralight DEX are Phase 3, gated by audit and licensing.

12.17 Exercises

1. **Base units.** Using Equation (12.1) with $d = 6$, write 0.07 PROFIT and 12.345678 PROFIT as whole numbers of micro-PROFIT. What does `to_micro` in **Listing 12.1** do with the text `"0.1234567"`, and why is that the right behavior for a ledger?

2. **The multiplier by hand.** Compute $m(50{,}000)$ from Equation (12.2) with $u_0 = 1$, then $E(50{,}000)$ from Equation (12.3) with $r = 1$. Check your answer with the power-law form Equation (12.5).

3. **The growing gap.** Prove that the ratio of linear to decaying emission is $2^{\log_{10} u}$, and use it to compute the ratio at 10^8 users.

4. **Supply by hand.** With $E = 1{,}000$ PROFIT per day and $b = 0.01$, compute S_1 to S_4 from Equation (12.6) with $S_0 = 0$. Then compute S_4 from Equation (12.8), the steady state from Equation (12.9), and t_{90} from Equation (12.10).

5. **Starting from a nonzero supply.** Show that for any starting supply S_0, the solution of Equation (12.6) is $S_t = S_\infty + (S_0 - S_\infty)(1-b)^t$ with $S_\infty = E/b$. What happens if $S_0 = S_\infty$? If $S_0 > S_\infty$?

6. **Break-even.** Use Equation (12.11) to find u^* for $\beta = 0.03$ PROFIT per user per day. Without a calculator, what is the ratio of burn to emission at $10u^*$?

7. **The floor.** A toy platform has a reserve of Rp50,000,000 and 40,000 PROFIT circulating. Compute the floor by Equation (12.12). Compute it again after a burn of 4,000 PROFIT, and separately after an emission of 6,000 PROFIT. Then show, using Equation (12.13), what a purchase of Rp2,000,000 at the original floor does.

8. **Pyramids.** Using Equation (12.14), how many people are needed to fill 12 levels below one person if each recruits 3? Explain in two sentences why a one-tier referral does not have this problem.

9. **Run the lab.** Run `03_supply_sim.py` (or `SupplySim`) with the arguments `1000 20 730`: a community of 1,000 users and a burn of 0.2% per day for two years. Report E, the steady state and the share of it reached after two years, and explain why this run is much further from its steady state than the default run.

10. **Live or roadmap?** For each statement, say whether it is live in the September 2026 release, on the roadmap, or neither: (a) a user sells 100 PROFIT for Rp100,000; (b) a user spends PROFIT on a scan top-up; (c) the team freezes a suspicious user's PROFIT account; (d) a user sends earned PROFIT to a friend; (e) PROFIT is swapped for SOL through the Ultralight DEX.

12.18 Solutions to the Exercises

Solution 1. One PROFIT is 10^6 micro-PROFIT, so $n = a \times 10^6$. For $a = 0.07$: $n = 70{,}000$. For $a = 12.345678$: $n = 12{,}345{,}678$. The text `"0.1234567"` has seven digits after the point, so `to_micro` raises `ValueError` with the message that it has more than 6 decimals. That is right because the chain cannot represent a seventh decimal: silently rounding would create or destroy value, while refusing makes the caller decide.

Solution 2. $\log_{10} 50{,}000 = \log_{10} 5 + 4 = 0.69897 + 4 = 4.69897$, so $m(50{,}000) = 0.5^{4.69897} = 2^{-4.69897} \approx 0.0385$. Then $E = 50{,}000 \times 0.0385 \approx 1{,}925$ PROFIT per day. By Equation (12.5), $E = 50{,}000^{0.69897} = 10^{4.69897 \times 0.69897} = 10^{3.2844} \approx 1{,}925$. The two forms agree (to more digits, both give 1,925.04).

Solution 3. Linear emission is u, decaying emission is $u \cdot m(u)$, so the ratio is $1/m(u) = 2^{\log_{10} u}$, because $m(u) = (1/2)^{\log_{10} u}$ by Equation (12.2). At $u = 10^8$, $\log_{10} u = 8$ and the ratio is $2^8 = 256$.

Solution 4. $q = 0.99$. $S_1 = 1{,}000$; $S_2 = 0.99 \times 1{,}000 + 1{,}000 = 1{,}990$; $S_3 = 0.99 \times 1{,}990 + 1{,}000 = 1{,}970.1 + 1{,}000 = 2{,}970.1$; $S_4 = 0.99 \times 2{,}970.1 + 1{,}000 = 2{,}940.399 + 1{,}000 = 3{,}940.399$. Closed form: $0.99^4 = 0.96059601$, so $S_4 = 1{,}000 \times (1 - 0.96059601)/0.01 = 100{,}000 \times 0.03940399 = 3{,}940.399$, the same. Steady state: $S_\infty = 1{,}000/0.01 = 100{,}000$. $t_{90} = \lceil \ln 0.1 / \ln 0.99 \rceil = \lceil -2.302585 / -0.010050 \rceil = \lceil 229.1 \rceil = 230$ days, which matches the last line of the lab's output for $b = 0.01$.

Solution 5. Unrolling as in Section 12.8.2 gives $S_t = q^t S_0 + E(1 - q^t)/b$ with $q = 1 - b$. Since $E/b = S_\infty$, this is $S_t = q^t S_0 + S_\infty - S_\infty q^t = S_\infty + (S_0 - S_\infty) q^t$. If $S_0 = S_\infty$, the second term is zero for every t: the supply never changes, because each day's burn $bS_\infty = E$ exactly cancels the emission; S_∞ is a *fixed point* of the recurrence. If $S_0 > S_\infty$, the second term is positive and shrinks by the factor q each day, so the supply *falls* toward S_∞: burning exceeds emission until the steady state is reached.

Solution 6. $\log_{10} 0.03 = -1.52288$; multiplied by $-1/\log_{10} 2 = -3.32193$ it gives 5.05889, so $u^* = 10^{5.05889} \approx 114{,}523$ users. At $10u^*$ the multiplier has halved once more, so per-user emission is $\beta/2$ and the ratio of burn to emission is 2.

Solution 7. Floor: 50,000,000/40,000 = Rp1,250. After burning 4,000: 50,000,000/36,000 ≈ Rp1,388.89, higher. After emitting 6,000 (starting again from 40,000): 50,000,000/46,000 ≈ Rp1,086.96, lower. A purchase of Rp2,000,000 at Rp1,250 gives 2,000,000/1,250 = 1,600 PROFIT; the new floor is 52,000,000/41,600 = 1,250, unchanged, as Equation (12.13) guarantees.

Solution 8. $3 \times (3^{12} - 1)/2 = 3 \times 531{,}440/2 = 797{,}160$ people. A one-tier referral pays only for the friends a person personally invites, is capped at 50, and pays both sides equally, so nobody is paid for the recruitment done by others and nothing depends on an ever-growing tree.

Solution 9. The run prints:

```
users=1,000  emission E=125.000000 PROFIT/day  burn b=0.0020/day  steady state E/b=62,500
 day      ledger supply        closed form gap (micro)
   0           0.000000           0.000000           0
  30       3,643.252892       3,643.252878          14
  90      10,305.020319      10,305.020277          42
 180      18,910.945544      18,910.945467          77
 365      32,402.696669      32,402.696532         137
 730      48,006.437382      48,006.437184         198
conservation: supply == emitted - burned on every day: True
after 730 days: emitted 91,250, burned 43,244
supply as a share of the steady state (each # = 2.5%):
   0 |                                        |   0.0%
  73 |#####                                   |  13.6%
 146 |##########                              |  25.3%
 219 |###############                         |  35.5%
 292 |###################                     |  44.3%
 365 |#######################                 |  51.8%
 438 |###########################             |  58.4%
 511 |###############################         |  64.0%
 584 |###################################     |  68.9%
 657 |#######################################   |  73.2%
 730 |###########################################|  76.8%
how the burn rate shapes the curve:
  b=0.0020: steady state       62,500 PROFIT, 90% reached on day 1151
  b=0.0050: steady state       25,000 PROFIT, 90% reached on day 460
  b=0.0100: steady state       12,500 PROFIT, 90% reached on day 230
```

With 1,000 users, $E = 125$ PROFIT per day (the first row of **Table 12.3**); with $b = 0.002$ the steady state is 125/0.002 = 62,500; after 730 days the supply is 48,006, 76.8% of the steady

state. The default run reached 97.4% in the same time because its burn rate was 0.5% instead of 0.2%. By Equation (12.10), the time scale is set by b alone: with $b = 0.002$, 90% takes 1,151 days, more than three years, whatever the emission.

Solution 10. (a) Neither live nor available: selling is disabled until mainnet and regulatory clarity. (b) Live. (c) Impossible: the freeze authority is none, so nobody can freeze a PROFIT account; account-level safety actions in the app (such as moderation) never touch a user's PROFIT. (d) Live, with a burned fee, the public log, and escrow if the friend has not verified an email. (e) Roadmap: Phase 3, after a token audit and licensing.

References

[1] Profitina, *Profitina Whitepaper — Public Version v24* (ID/EN), Sep. 2026. [Online]. Available: https://app.profitina.com/static/whitepaper/

[2] Profitina, *Profitina — Dasbor GEO*, PROFIT area and plan cards, release 0.14.5-testnet. [Online]. Available: https://app.profitina.com (accessed Sep. 24, 2026).

[3] Republik Indonesia, *Undang-Undang Nomor 7 Tahun 2011 tentang Mata Uang* (Law No. 7 of 2011 on Currency), 2011.

[4] Herbert Smith Freehills Kramer, "OJK assumes regulatory oversight of digital financial assets," May 15, 2025. [Online]. Available: https://www.hsfkramer.com/notes/fintech/2025-posts/ojk-assumes-regulatory-oversight-of-digital-financial-assets

[5] Solana Foundation, "Fees," *Solana Documentation*. [Online]. Available: https://solana.com/docs/core/fees (accessed Sep. 24, 2026).

[6] Solana Foundation, "Clusters and Public RPC Endpoints," *Solana Documentation*. [Online]. Available: https://solana.com/docs/references/clusters (accessed Sep. 24, 2026).

[7] Solana Compass, "Solana Mainnet 30-Month Uptime Streak: Firedancer, QUIC, and Priority Fees," Aug. 10, 2026. [Online]. Available: https://solanacompass.com/news/solana-marks-30-consecutive-months-without-a-network-wide-outage

[8] Solana Foundation, "Solana Changelog: September 18, 2026," Sep. 2026. [Online]. Available: https://solana.com/news/solana-changelog-september-18-2026

[9] Solana Foundation, "Tokens on Solana," *Solana Documentation*. [Online]. Available: https://solana.com/docs/tokens (accessed Sep. 24, 2026).

[10] Metaplex Foundation, "Token Metadata," *Metaplex Documentation*. [Online]. Available: https://www.metaplex.com/docs/smart-contracts/token-metadata (accessed Sep. 24, 2026).

[11] IEEE, *IEEE Standard for Floating-Point Arithmetic*, IEEE Std 754-2019, Jul. 2019.

[12] Squads, "Welcome to Squads Multisig," *Squads Documentation*. [Online]. Available: https://docs.squads.so/main (accessed Sep. 24, 2026).

[13] V. Buterin, E. Conner, R. Dudley, M. Slipper, I. Norden, and A. Bakhta, "EIP-1559: Fee market change for ETH 1.0 chain," Ethereum Improvement Proposals, no. 1559, Apr. 2019. [Online]. Available: https://eips.ethereum.org/EIPS/eip-1559

[14] R. L. Graham, D. E. Knuth, and O. Patashnik, *Concrete Mathematics: A Foundation for Computer Science*, 2nd ed. Reading, MA, USA: Addison-Wesley, 1994, ch. 2.

[15] R. L. Burden, J. D. Faires, and A. M. Burden, *Numerical Analysis*, 10th ed. Boston, MA, USA: Cengage Learning, 2016, sec. 2.1.

[16] ANTARA News, "QRIS reaches 65.77 mln users, accepted by 44.86 mln merchants: BI," Aug. 21, 2026. [Online]. Available: https://en.antaranews.com/news/428168/qris-reaches-6577-mln-users-accepted-by-4486-mln-merchants-bi

[17] Republik Indonesia, *Undang-Undang Nomor 7 Tahun 2014 tentang Perdagangan* (Law No. 7 of 2014 on Trade), art. 9, 2014.

[18] Solana Foundation, “Solana Explorer,” devnet cluster. [Online]. Available: https://explorer.solana.com/?cluster=devnet (accessed Sep. 24, 2026).

Chapter 13

The Profitina Group and the Road Ahead

By the end of this chapter you will be able to describe the Profitina Group as a whole: an umbrella of sibling products that share six common layers, each product with its own job. You will know what Profitina Capital does with risk statistics such as value at risk, expected shortfall, maximum drawdown and concentration, and you will be able to compute each of them by hand on a ten-day example. You will be able to follow Profitina TTM as it turns a paragraph of text into a short film, and Profitina AI as it routes a request between a model on the user's own machine and a cloud provider, and you will understand why Profitina Laundry, a fictional laundry business, is the place where readers of this book series practice. You will be able to name the three principles that bind all of these products (local-first AI; the app advises and the user decides; tests as a gate) and point to where each one shows up. You will build Lab G9 in Python and Java: a checker that asks the public OpenAlex catalogue how complete and consistent a researcher's record is, refuses to invent a result for a name it cannot find, and never ranks anyone. And you will see what comes next on the road, with a clear line between what has happened and what is only planned. The chapter, and the book, end with a short closing word.

13.1 Where We Left Off

Chapter 12 finished the tour of Profitina's economy. PROFIT turned out to be a utility coin with a small, precise set of rules: a devnet token with six decimals and no freeze authority, an emission multiplier that halves for every tenfold growth of users, four burns, a Rupiah floor that a purchase at the floor price cannot move, and a table that separates what is live from what waits for an audit and a license. Before that, Chapters 2 to 11 walked through the product itself: the answer-engine shift and GEO, the scan and its fixes, the community, safety, local AI, the feedback loop, the business model, the architecture and its release gate, the marketing site, and privacy by design.

All of that was one product, app.profitina.com, with its marketing site profitina.com. But Profitina is also the name of a small family of products, built by the same people with the same habits. Some of those products share code with each other; one of them exists only to teach. This last chapter steps back to look at the family, the principles that hold it together, and the road ahead.

The chapter has a lab like every other, and a fitting one. The Researcher Visibility Report, one of the newest features of Profitina (Chapter 3), measures how complete and findable a scholar's record is in OpenAlex, the open catalogue of the world's research. Lab G9 builds a small, transparent version of the same idea. It is a good last lab because it gathers so many threads of the book: a public API used politely, honest failure when the network is not there, a negative control that refuses to invent results, string normalization as a data-quality tool, and a report that helps a person fix something without ranking them against anyone.

13.2 One Umbrella, Several Houses

13.2.1 The shape of the group

Picture a family compound in a Javanese village: several houses around one yard, each household with its own kitchen and its own work, but one well, one gate and one set of rules about the yard. The *Profitina Group* is organized like that. **Figure 13.1** shows it.

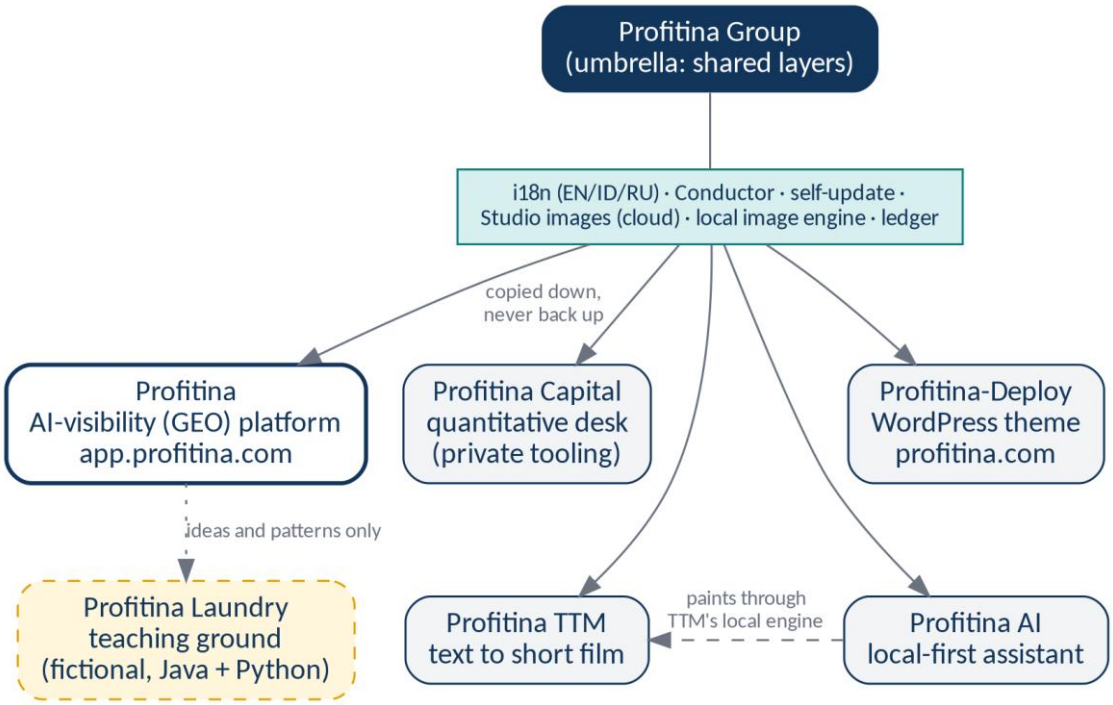


Figure 13.1 The Profitina Group. Shared layers live at the top and are copied down into each product, never back up. Profitina Laundry, the teaching ground, borrows only ideas and patterns, never code or data.

At the top sits the group itself, a single repository that holds the pieces every product needs. Below it are the products: Profitina, the AI-visibility platform this book is about; Profitina-Deploy, the WordPress theme of profitina.com (Chapter 10); Profitina Capital, a quantitative desk; Profitina TTM, which turns text into short films; and Profitina AI, a local-first assistant app. Off to the side, with a dashed border because it belongs to the classroom rather than to production, is Profitina Laundry.

13.2.2 Six shared layers

The group holds six layers that more than one product needs. Each is a building block that would be wasteful, and risky, to write several times.

1. **User-interface languages.** English, Indonesian and Russian dictionaries for the interface, with the same keys in every language. Profitina itself guards its ID and EN dictionaries with tests that fail if a key exists in one language and not the other: in September 2026, 1,531 dictionary keys had to be symmetric between the two languages.
2. **A "Conductor".** A component that decides which AI model should handle a given piece of work, based on the kind of task, the models available and their recent health.
3. **Self-update.** The machinery by which a desktop product learns that a newer version exists and installs it safely.

4. **A Studio image engine (cloud).** A uniform way to ask several cloud image services for a picture.
5. **A built-in local image engine.** The same interface, answered by a diffusion model running on the user's own graphics card.
6. **A "ledger" of resting providers.** A small shared memory of which AI providers are temporarily resting after an authentication error, a quota limit or an outage, so that one product's bad experience with a provider saves the others from repeating it.

The sixth layer deserves a sentence of theory, because it is a textbook pattern in disguise. When a remote service fails, a naive client retries immediately, fails again, retries again, and turns one outage into a storm of useless requests. A *circuit breaker* instead records the failure and stops calling the service for a while; after a cool-down it lets one trial request through, and only if that succeeds does normal traffic resume [1]. The ledger is a circuit breaker whose memory is shared across products.

13.2.3 One rule for sharing

Shared code creates a new danger: two copies that drift apart until nobody knows which is right. The group answers with one strict rule: **change in Group, copy to the units, never the reverse.** A fix to a shared layer is made once, at the top, and then copied down into each product; a product never patches its copy of a shared layer and leaves the others behind. The rule is the same one Chapter 11 met in a different form for the Privacy Policy: "remove duplication, don't synchronize it". Where duplication cannot be removed, as with code that must ship inside several separate products, it must at least flow in one direction only.

Two more habits keep the group honest. Work happens on two computers, with a strict routine of pulling the latest version before starting and synchronizing after finishing, so that neither machine ever holds changes the other has not seen. And every commit passes through a *secret scanner* that refuses to record anything that looks like a key, a token or a password. The second habit matters more than it seems: a secret that reaches version history has to be treated as leaked, however quickly it is deleted afterwards.

> **KEY INSIGHT.** A family of products stays coherent through a few rules that are easy to state and hard to break: one direction for shared code, one routine for two machines, and a gate that stops secrets before they are written down.

13.3 Profitina Capital: A Quantitative Desk

13.3.1 What it is

Profitina Capital is the group's quantitative execution desk: a local application for managing a crypto portfolio across exchanges, lending, staking and liquidity pools. It is private tooling. It is not a public product, it does not manage anyone else's money, and nothing in this section is financial advice. It appears in this book because the mathematics inside it is the mathematics of an ordinary numerical-methods or algorithms course, applied with discipline.

Its September 2026 release, version 3.8, added a *risk desk* with value at risk and expected shortfall at 95% and 99%, volatility, maximum drawdown, sensitivity (beta) to the prices of BTC and SOL, correlations, concentration measured by the Herfindahl–Hirschman index, stress scenarios and estimates of impermanent loss for liquidity pools. It also added performance attribution (time-weighted and money-weighted return, the Sharpe, Sortino and Calmar ratios),

monthly statements, transaction-cost analysis (slippage and implementation shortfall), policy limits that raise alerts when breached, and checks on the health of the data feeding all of it.

That is a long list. The rest of this section takes five of its items and computes each by hand on one tiny example, because each is a formula that any reader can understand and none of them is magic.

13.3.2 Five measures on ten days

Take a portfolio whose daily returns over ten days, in percent, were

$$r = (1.2,\ -0.8,\ 0.5,\ -2.1,\ 0.9,\ -0.3,\ 1.5,\ -1.4,\ 0.7,\ -0.6)$$

Value at risk. *Value at risk* at confidence level c answers the question "what loss is exceeded only on the worst $(1-c)$ fraction of days?". With historical data, sort the returns from worst to best and read off the loss at the $(1-c)$ quantile q_{1-c}:

$$\mathrm{VaR}_c = -\,q_{1-c}(r) \tag{13.1}$$

Sorted, our returns are $-2.1, -1.4, -0.8, -0.6, -0.3, 0.5, 0.7, 0.9, 1.2, 1.5$. At $c = 80\%$, the worst 20% of ten days is two days, and the quantile sits at the second-worst return, -1.4. So $\mathrm{VaR}_{80\%} = 1.4\%$: on four days out of five, the loss was no worse than 1.4%. (Real desks use 95% or 99% and hundreds of days; ten days and 80% keep the arithmetic visible.)

Expected shortfall. VaR says where the bad days begin, not how bad they are. *Expected shortfall*, also called conditional value at risk (CVaR), is the average loss on the days at or beyond the VaR threshold [2]:

$$\mathrm{CVaR}_c = -\,\mathbb{E}[\,r \mid r \le q_{1-c}(r)\,] \tag{13.2}$$

Here the two worst days are -2.1 and -1.4, so $\mathrm{CVaR}_{80\%} = (2.1 + 1.4)/2 = 1.75\%$. Expected shortfall is always at least as large as VaR, and it is the more honest number when losses have long tails.

Maximum drawdown. Start the portfolio at 100 and compound the returns: 100, 101.20, 100.39, 100.89, 98.77, 99.66, 99.36, 100.85, 99.44, 100.14, 99.54. The *drawdown* at each moment is the fall from the highest value seen so far, and the *maximum drawdown* is the largest such fall:

$$\mathrm{MDD} = \max_{t} \frac{\max_{s \le t} V_s - V_t}{\max_{s \le t} V_s} \tag{13.3}$$

The peak is 101.20 after day 1; the lowest point after it is 98.77 after day 4; so $\mathrm{MDD} = (101.20 - 98.77)/101.20 \approx 2.40\%$. Drawdown is the number that matches how people actually feel risk: not the average wobble but the worst stretch from a high to a low.

Sharpe ratio. The *Sharpe ratio* divides the average excess return by its standard deviation, measuring return per unit of variability [3]:

$$\mathrm{SR} = \frac{\bar{r} - r_f}{\sigma_r} \tag{13.4}$$

With a risk-free rate $r_f = 0$ for simplicity, the mean is $\bar{r} = -0.04\%$ per day and the sample standard deviation is $\sigma_r \approx 1.187\%$, so SR ≈ -0.034 per day. A negative Sharpe ratio says, correctly, that these ten days lost a little money while bouncing around a lot.

Concentration. If a portfolio holds assets with weights $w_1, \ldots, w_n$ (fractions of its value that add up to 1), the *Herfindahl–Hirschman index* is the sum of the squared weights:

$$\text{HHI} = \sum_{i=1}^{n} w_i^2 \tag{13.5}$$

For weights 0.5,0.3,0.2: HHI $= 0.25 + 0.09 + 0.04 = 0.38$. The index is 1 for a portfolio in a single asset and $1/n$ for n equal weights, so its reciprocal $1/0.38 \approx 2.6$ can be read as "this portfolio behaves like 2.6 equally weighted assets", fewer than the three it holds.

> **WARNING.** These numbers describe ten invented days. They are here to teach formulas, not to recommend any asset, strategy or product. Risk statistics describe the past; they do not promise anything about the future.

13.3.3 What Capital teaches

For a student, Capital is a collection of exercises that happen to be real. Sorting and quantiles are the heart of VaR and expected shortfall. A single pass with a running maximum computes the maximum drawdown in linear time, a small and satisfying algorithm. Correlations and betas are least-squares problems. Stress scenarios are "what if" recomputations over a changed input. And policy limits with alerts are the same pattern as Profitina's own monitoring (Chapter 9): measure, compare with a threshold the user chose, and tell a human.

Two items on the list are less familiar and worth one paragraph each. *Transaction-cost analysis* asks how much a trade actually cost compared with the price at the moment the decision was made. The difference, called *implementation shortfall*, includes the fees paid, the *slippage* between the expected and the executed price, and the cost of any part of the order that could not be filled. It is the trading version of a lesson every engineer learns: the plan and the result are different measurements, and only the second is real. *Data-health checks* ask, before any statistic is computed, whether the prices and balances feeding it are fresh, complete and plausible: a missing day, a stale quote or a balance that jumped by a factor of a thousand is flagged rather than silently averaged in. A risk number computed from broken data is worse than no number, because it looks like knowledge. The principle is the same as the negative control of Lab G9 and the "silent zero" of Section 13.4.3: when the input is not trustworthy, say so instead of producing a confident answer.

13.4 Profitina TTM: From Text to a Short Film

13.4.1 The pipeline

Profitina TTM (text to multimedia; version 2.8.1 in September 2026) takes a text or a script and produces a short film. It works as a pipeline of stages, each with a clear input and output.

1. **Planner.** Splits the text into scenes and decides roughly how long each should last.
2. **Director.** Gives the film a shape. It arranges the scenes along six story beats (*hook, setup, rise, turn, climax, landing*), detects the mood of each scene, and picks lighting and lens recipes to match.
3. **Painting.** Produces an image for each shot.

4. **Voice.** Reads the narration aloud.
5. **Score and ambience.** Generates music and background sound procedurally, from rules rather than recordings.
6. **Render.** Assembles everything into a video with ffmpeg, the standard open-source toolkit for audio and video [4].

The six beats are an old storytelling idea in modern clothes. A viewer is caught by a hook, oriented by a setup, carried by rising tension, surprised by a turn, held at a climax, and set down gently at a landing. A pipeline that knows these beats can place a close-up where the story turns and a wide shot where it lands.

13.4.2 Painting in two lanes

Since version 2.8, TTM paints in a *hybrid* way. Cloud image engines are tried first, and a built-in local diffusion engine running on the laptop's own graphics card, an RTX 4060, joins in whenever it is at least as good and not slower. The local engine uses fast, few-step diffusion models such as SDXL-Lightning, which distills a large image model so that it can produce a picture in a handful of steps instead of dozens [5]. The rule "join when at least as good and not slower" should sound familiar: it is the same rule as Profitina's Content Engine quality gate (Chapter 6), which serves an AI draft only when it beats the template. In both places, a model earns its place by measurement.

TTM also has a 2.5D "camera". From a single still image it estimates depth with Depth Anything V2, a monocular depth-estimation model [6], and then moves a virtual camera slightly so that near objects shift more than far ones. The eye reads this *parallax* as depth, and a still picture seems to breathe.

13.4.3 Fail loudly

The most important design decision in TTM is not about images at all. Early versions, when a painting engine failed, quietly rendered a smooth colored gradient in place of the missing picture, and the film finished "successfully" with blank shots in it. TTM now **fails loudly**: if a shot cannot be painted, the run stops or reports exactly which shot is missing and why, rather than rendering a silent placeholder. Profitina's own engineering history records the same lesson as "the silent zero": a `try`/`except` around a failed count would once have displayed a convincing 0, and the rule since then is that errors must be loud. A result that looks finished but is not is worse than an honest error, because nobody knows to fix it.

13.5 Profitina AI: A Local-First Assistant

13.5.1 What it does

Profitina AI (version 1.7.2 in September 2026) is an assistant application that runs in a browser and as a Windows desktop app. A user chats with it; behind the chat, the Conductor of Section 13.2.2 decides whether a request goes to a model running locally through Ollama (for example a Qwen model) or to a cloud provider. It supports threads, and agents that work on a task together; tools for files, processes and Office documents (find and replace inside a `.docx`, building a table of contents, exporting to PDF); scheduled tasks; the group's interface languages; and a Studio for images that can paint through TTM's local engine.

Local-first means that the local model is the default and the cloud is the exception the user chooses. It does not mean "local only". That distinction is exactly where Profitina AI and Profitina,

the product of this book, differ: Profitina AI may call cloud providers when its user asks it to, while **Profitina, the GEO platform, uses no cloud AI at all**. Its models run on its own server, through Ollama, in the two lanes of Chapter 6, and user data never leaves the server for AI processing. The two products are siblings, and Profitina AI is not part of app.profitina.com.

How does a conductor decide? At the level this book describes, it weighs three plain questions for every request. *May this data leave the machine?* If the user has not chosen a cloud provider for it, the answer is no, and the request stays local whatever else is true. *Which available model is fit for this kind of task?* A short classification needs a small, fast model; a long document needs a larger one; an image needs an image engine. *Which of those is healthy right now?* This is where the shared ledger of Section 13.2.2 earns its place: a provider that has just returned a quota or authentication error is marked as resting and skipped until its cool-down ends, so the user is not made to wait for a failure that another product has already seen. A scheduled task, such as a weekly summary of a folder of documents, passes through the same three questions at the time it runs, not at the time it was created, because the answers can change in between.

13.5.2 Never stop silently

After version 1.7.0, Profitina AI adopted a motto that sounds like TTM's: *never stop silently*. Every turn of the conversation ends with an explicit card that says what was done and what was not; loop breakers stop an agent that repeats itself; and whatever partial output exists is always returned, even when a task cannot be completed. A user should never be left staring at a spinner, wondering whether anything is happening.

Put the three sibling products side by side and the pattern is unmistakable. Profitina says "not sure yet" when it cannot detect a category, and "no model available" when the local model is down (Chapters 3 and 6). TTM fails loudly instead of painting gradients. Profitina AI never stops silently. These are not three coincidences. They are one principle: **tell the user the truth about what happened**, which the next section returns to.

13.6 Profitina Laundry: The Teaching Ground

Profitina Laundry is a small, fictional laundry business, and the application that runs it. It exists only to teach. The Web Programming and Framework-Based Programming books of this series build it, chapter by chapter, in both Java and Python, and the material planned around it includes use cases, an entity-relationship diagram, hundreds of sample transactions and a README for each chapter.

Why invent a laundry when a real platform is right there? Three reasons.

The first is safety. A real product has real users, real data and real secrets. A reader practicing on it could leak something, break something, or be tempted to copy what should not be copied. A fictional laundry has nothing to leak: its customers do not exist, its orders are samples, and its code belongs to the reader.

The second is size. A laundry is small enough to understand completely: customers bring clothes, the shop weighs them, washes, irons, charges by the kilogram, and hands them back. Every concept of a web application (accounts, orders, prices, statuses, receipts, reports) appears in it without the thousand details of a production platform.

The third is honesty about what is shared. Profitina Laundry has no connection to the code or data of the real product. What it borrows from Profitina is ideas and patterns: a scheduled job, a

daily quota that resets at 00:00 WIB, a state machine for an order, a consent log, a structured-data block for a local business. Several labs in this book pointed to it for exactly that reason: Chapter 2's JSON-LD generator describes the laundry, Chapter 8's plans and quotas could belong to a laundry's express service, and Chapter 12 suggested a laundry loyalty point that halves its earning rate as the customer base grows tenfold.

There is a quieter fourth reason: stability. A production platform changes under the reader's feet; Profitina's own codebase received 391 commits between 3 July and late September 2026, and a chapter that described its internals line by line would be out of date before it was printed. A teaching ground changes only when the book that uses it changes, so an exercise written today still works when a student meets it next year.

> **DEFINITION.** A *teaching ground* is a deliberately small, fictional system on which learners practice real techniques without real users, real data or real risk. It shares ideas with production systems, never code or data.

13.7 The Principles That Bind Them

Five products, three languages of interface, two computers, one teaching ground. What makes them one family is not a logo but a small set of principles that show up, in different clothes, in every one of them. **Figure 13.2** draws the three that matter most as a single loop.

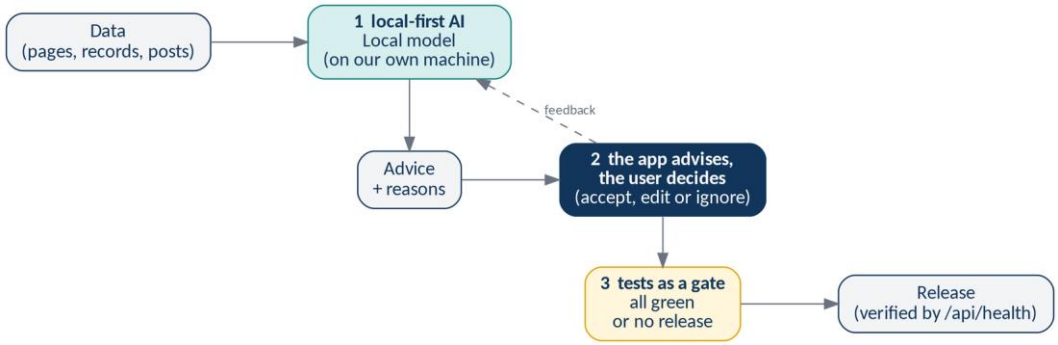


Figure 13.2 The three principles as one loop. Data goes to a local model; the model gives advice with reasons; a person decides; tests stand between every change and its release, which counts as done only when the health check confirms it.

13.7.1 Local-first AI

The first principle is that AI runs, by default, on hardware the product controls. In Profitina, the GEO platform, this is absolute: Ollama with Qwen models in two lanes, no hosted-AI code paths, no user data sent out for AI processing (Chapter 6). In Profitina AI it is a default with a visible exception: local unless the user chooses a cloud provider. In TTM it takes the form of the local diffusion engine that joins whenever it is as good and as fast as the cloud.

Why local? Chapter 6 gave three reasons, and they generalize to the whole group. *Privacy*: data that never leaves the machine cannot leak from someone else's. *Independence*: a product whose core depends on a paid third-party API inherits that API's prices, limits and outages; the group's own "ledger" of resting providers exists because cloud providers do rest. *Honesty about cost*: a local model has a cost that can be measured on one's own machine, including the small, surprising ones, such as the 2.25-second delay that dropped to 0.17 seconds once Profitina called Ollama at `127.0.0.1` instead of `localhost` on Windows.

Local-first has a price, and it is fair to name it: local models are smaller than the largest cloud models, and a single machine has a single GPU. The group pays that price with measurement (the quality gates of Chapter 6 and TTM's "at least as good and not slower") rather than with hope.

13.7.2 The app advises; the user decides

The second principle is that software proposes and a person disposes. Profitina's scan gives a score and a prioritized list of fixes, with the methodology shown; the user decides what to change on their own site. The Content Engine writes a draft; the user edits and publishes it, or not. The feedback pipeline curates suggestions with a local model, but "product decisions are always made by a human" (Chapter 7). Capital raises an alert when a policy limit is breached; it does not trade on its own initiative because of it. The researcher checker of Lab G9 reports gaps with advice; it does not grade the person.

This principle is also a legal one. UU PDP gives people the right to object to decisions based solely on automated processing (Chapter 11), and a product built around "the user decides" meets that right by design rather than by exception. And it is an ethical one: a system that advises can be wrong without harm, because a person stands between its mistake and the world.

13.7.3 Tests as a gate

The third principle is that nothing reaches users unless the automated tests are green. In Profitina this is concrete and measured. At the time of writing, 24 September 2026, the full SQLite test suite for release 0.14.5 had **1,577 tests passed and 1 skipped**, and a priority subset run against PostgreSQL 16 had **440 passed**, alongside JavaScript guards for the user interface. The release itself is a six-step gate (Chapter 9): lock the exact version to release, back up, run the tests, check that nothing changed during the tests, copy, restart, and declare success **only when `/api/health` answers with the new version**.

A test suite is a gate rather than a report because of what happens when it fails: nothing ships. A report can be read later, argued with, or ignored; a gate cannot be walked around. The same idea runs through the rest of the group in smaller forms: the secret scanner at every commit, the byte-for-byte comparison of the Python and Java outputs in every lab of this book, the negative control of the Researcher Visibility Report.

13.7.4 And one more: tell the truth about what happened

Sections 13.4 and 13.5 found a fourth principle that runs underneath the other three: *say what actually happened*. "Not sure yet." "No model available." "Fail loudly." "Never stop silently." "Visibility results are methodological estimates, not guarantees." "Model estimate, not a market price." "Zero PROFIT purchases." Each is a sentence a less careful product would have avoided, and each made the product more trustworthy by being said.

> **KEY INSIGHT.** The principles are cheap to state and expensive to keep. They are kept by mechanisms: a local model instead of an API key, a button that waits for a person, a gate that stops a release, a sentence that admits a limit.

13.8 Lab G9: Checking a Researcher Record with OpenAlex

13.8.1 The feature behind the lab

A researcher's work can only be cited by an AI answer engine, or found by a colleague, if it can be found at all, and found as the work of the right person. In scholarly catalogues this goes wrong in dull, fixable ways: works without a DOI, a missing ORCID identifier (the persistent identifier that distinguishes one researcher from another with the same name), and, most often, the same institution written in many different ways.

Profitina's *Researcher Visibility Report* (Chapter 3) measures exactly these things, using **OpenAlex** [7]. It names concrete gaps with steps to fix them, for example "No ORCID linked" or "Your affiliation is written in N different spellings". It processes published works and their bibliographic metadata, never a person's address, photo or social links. And it has a permanent **negative control**: fictional names always return "not found", never a plausible score; in the September 2026 release, 8 of 8 fictional test subjects stayed at 0.0 [7]. The organization-level version, *Organization Visibility*, applied to one Indonesian university, found **466 different spellings of the affiliation in 500 sampled works**, and 244 of 607 researchers without a linked ORCID [7]. It deliberately reports only aggregate gaps that an organization can close, and never a leaderboard of people.

Lab G9 builds a small, transparent checker in the same spirit. It is not Profitina's report: it uses only public fields, simple rules chosen for teaching, and thresholds the reader can change.

13.8.2 OpenAlex in brief

OpenAlex is a free and open catalogue of scholarly works, researchers, sources, institutions and topics, created by the nonprofit OurResearch and launched in 2022 as an open successor to the discontinued Microsoft Academic Graph [8]. It is reachable through a REST API that returns JSON. Three calls are enough for the lab:

- `GET /authors?search=<name>` searches researcher records by name and returns a page of candidates.
- `GET /authors/<id>` returns one researcher record by its OpenAlex identifier, such as `A5048491430`.
- `GET /works?filter=author.id:<id>` lists the works attributed to that record, each with its DOI, its open-access status and its *authorships*: for each person on the paper, the name as printed and the affiliation strings as printed.

Access rules changed in 2026. In January 2026 OpenAlex announced API keys and usage-based pricing, "still free for most users" [9]. As of September 2026 its documentation describes a daily budget: a request for a single record by its identifier is free, a thousand list-and-filter requests cost USD 0.10 of budget and a thousand searches USD 1; without a key a user has USD 0.10 of budget per day, and a free key raises it to USD 1; more than 100 requests per second, or an exhausted budget, returns `429 Too Many Requests` [10]. The lab's programs read an optional key from the environment variable `OPENALEX_API_KEY` and add it to every request; the few requests of a single check fit comfortably in the keyless budget.

13.8.3 What the checker checks

Figure 13.3 shows the steps.

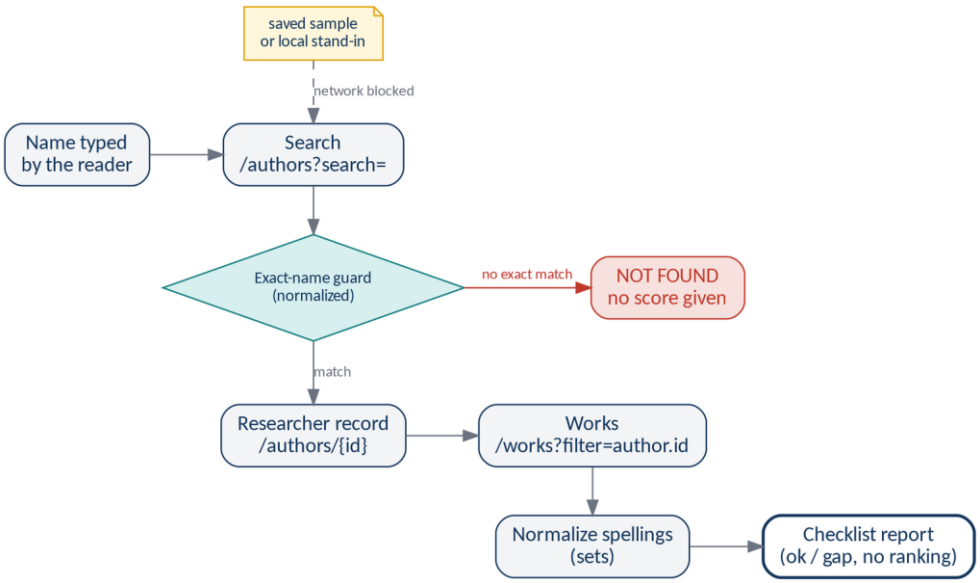


Figure 13.3 Lab G9. A name is searched; only an exact match after normalization passes the guard, and anything else is reported as not found with no score. When the network is blocked, a saved sample or a local stand-in answers the same three calls.

1. **Search** for the typed name and receive candidates.
2. **Guard.** Accept a candidate only if its display name, or one of its recorded alternative names, matches the typed name exactly after normalization. Otherwise print NOT FOUND and give no score. This is the negative control: a search engine that is asked about a person who does not exist will happily return the closest real person, and a checker that trusted the top result would score a stranger.
3. **Record.** Fetch the researcher record: ORCID, number of works, recorded name variants.
4. **Works.** Fetch the works and count those with a DOI and those that are open access.
5. **Normalize.** Collect the affiliation strings this researcher's name carries on those works and count how many really differ after normalization, against the number of distinct institutions OpenAlex resolved them to.
6. **Report** each result as ok or gap, with no overall score and no comparison with anyone.

For the rates, the checker needs a percentage rounded to a whole number, and, as in Lab G8, it must round the same way in Python and Java. With integers p (the part) and w (the whole), the percentage rounded half up is

$$\mathrm{pct}(p, w) = \left\lfloor \frac{200p + w}{2w} \right\rfloor \tag{13.6}$$

because $\frac{200p+w}{2w} = \frac{100p}{w} + \frac{1}{2}$, and taking the floor of a number plus one half rounds it to the nearest integer, halves upward. Integer division computes the floor directly, so no floating point is involved at all.

The teaching thresholds are simple and visible in the code: DOI coverage of at least 90% is ok, open access of at least 50% is ok, at most two distinct name forms is ok, and no more distinct affiliation spellings than institutions is ok. They are a starting point for discussion, not a standard; Exercise 9 asks the reader to argue for different ones.

13.8.4 Step 1: counting spellings with a set

How many ways is the same thing written? The answer is the size of a *set* of normalized strings. Normalization should remove differences that do not matter and keep those that do. For affiliation strings, the lab lowercases, removes accents (so that “Département” and “Departement” match), turns every punctuation mark into a space, and squeezes runs of spaces into one. For personal names it does the same and then sorts the words, so that “Piwowar, Heather” and “Heather Piwowar” become the same key. Accent removal uses Unicode normalization form NFKD, which splits a letter such as “é” into “e” followed by a combining accent that can then be dropped [11].

Listing 13.1 is the Python version, with a small demonstration on six name forms and six affiliation strings for one Indonesian university.

Listing 13.1 `normalize.py`: two normalizers and a set that counts the spellings that really differ.

```
"""Lab G9, step 1: how many ways is the same thing written?

Two small normalizers: one for affiliation strings, one for personal names.
A set of normalized strings counts the spellings that really differ.
"""
import unicodedata


def norm_text(s):
    """Lowercase, drop accents, turn punctuation into spaces, squeeze spaces."""
    s = unicodedata.normalize("NFKD", s)
    s = "".join(c for c in s if not unicodedata.combining(c))
    s = "".join(c if c.isalnum() else " " for c in s.lower())
    return " ".join(s.split())


def norm_name(s):
    """Like norm_text, but word order does not matter ('Piwowar, Heather')."""
    return " ".join(sorted(norm_text(s).split()))


def spellings(strings, key):
    """Return (distinct raw strings, distinct keys)."""
    return len(set(strings)), len({key(s) for s in strings})


def main():
    names = ["Heather Piwowar", "Piwowar, Heather", "Heather, Piwowar",
             "Heather A. Piwowar", "Heather A Piwowar", "H. Piwowar"]
    for n in names:
        print(f"{n:<20} -> {norm_name(n)!r}")
    raw, keys = spellings(names, norm_name)
    print(f"names: {raw} raw spellings, {keys} after normalization")

    affiliations = [
        "Institut Teknologi Sepuluh Nopember",
        "Institut Teknologi Sepuluh Nopember, Surabaya",
        "INSTITUT TEKNOLOGI SEPULUH NOPEMBER",
        "Institut Teknologi Sepuluh Nopember.",
        "Département d'Informatique, Institut Teknologi Sepuluh Nopember",
        "ITS Surabaya",
    ]
    for a in affiliations:
        print(f"{norm_text(a)!r}")
    raw, keys = spellings(affiliations, norm_text)
    print(f"affiliations: {raw} raw spellings, {keys} after normalization")
```

Listing 13.1 (Continued) *normalize.py: two normalizers and a set that counts the spellings that really differ.*

```
if __name__ == "__main__":
    main()
```

Listing 13.2 is the Java version. `java.text.Normalizer` performs the NFKD decomposition, and the regular expression `\p{M}` removes the combining marks [12].

Listing 13.2 `Normalize.java`: the same normalizers in Java.

```
import java.text.Normalizer;
import java.util.Arrays;
import java.util.HashSet;
import java.util.List;
import java.util.Set;
import java.util.function.Function;

/** Lab G9, step 1: how many ways is the same thing written? */
public class Normalize {
    /** Lowercase, drop accents, turn punctuation into spaces, squeeze spaces. */
    static String normText(String s) {
        String d = Normalizer.normalize(s, Normalizer.Form.NFKD)
                             .replaceAll("\\p{M}", "");
        StringBuilder b = new StringBuilder();
        for (char c : d.toLowerCase().toCharArray()) {
            b.append(Character.isLetterOrDigit(c) ? c : ' ');
        }
        return String.join(" ", b.toString().trim().split("\\s+"));
    }

    /** Like normText, but word order does not matter ("Piwowar, Heather"). */
    static String normName(String s) {
        String[] words = normText(s).split(" ");
        Arrays.sort(words);
        return String.join(" ", words);
    }

    /** Return {distinct raw strings, distinct keys}. */
    static int[] spellings(List<String> strings, Function<String, String> key) {
        Set<String> keys = new HashSet<>();
        for (String s : strings) keys.add(key.apply(s));
        return new int[] {new HashSet<>(strings).size(), keys.size()};
    }

    public static void main(String[] args) {
        List<String> names = List.of("Heather Piwowar", "Piwowar, Heather",
            "Heather, Piwowar", "Heather A. Piwowar", "Heather A Piwowar",
            "H. Piwowar");
        for (String n : names) {
            System.out.printf("%-20s -> '%s'%n", n, normName(n));
        }
        int[] r = spellings(names, Normalize::normName);
        System.out.printf("names: %d raw spellings, %d after normalization%n",
                          r[0], r[1]);
```

Listing 13.2 (Continued) *Normalize.java: the same normalizers in Java.*

```
        List<String> affiliations = List.of(
            "Institut Teknologi Sepuluh Nopember",
            "Institut Teknologi Sepuluh Nopember, Surabaya",
            "INSTITUT TEKNOLOGI SEPULUH NOPEMBER",
            "Institut Teknologi Sepuluh Nopember.",
            "Département d'Informatique, Institut Teknologi Sepuluh Nopember",
            "ITS Surabaya");
        for (String a : affiliations) {
            System.out.println("'" + normText(a) + "'");
        }
        r = spellings(affiliations, Normalize::normText);
        System.out.printf(
            "affiliations: %d raw spellings, %d after normalization%n",
            r[0], r[1]);
    }
}
```

Both print:

```
Heather Piwowar      -> 'heather piwowar'
Piwowar, Heather     -> 'heather piwowar'
Heather, Piwowar     -> 'heather piwowar'
Heather A. Piwowar   -> 'a heather piwowar'
Heather A Piwowar    -> 'a heather piwowar'
H. Piwowar           -> 'h piwowar'
names: 6 raw spellings, 3 after normalization
'institut teknologi sepuluh nopember'
'institut teknologi sepuluh nopember surabaya'
'institut teknologi sepuluh nopember'
'institut teknologi sepuluh nopember'
'departement d informatique institut teknologi sepuluh nopember'
'its surabaya'
affiliations: 6 raw spellings, 4 after normalization
```

Six name forms collapse to three keys: “heather piwowar”, “a heather piwowar” (with the middle initial) and “h piwowar” (with only an initial). Six affiliation strings collapse to four. Normalization merged the capitalized, the punctuated and the plain versions of the university’s name; it correctly kept “…, Surabaya” as a separate form, because the extra word is real information; and it could not know that “ITS Surabaya” is an abbreviation of the same institution. That last case is the honest limit of string normalization: some variants need knowledge, not rules, which is exactly why OpenAlex resolves affiliation strings to institution identifiers, and why the checker reports raw strings *and* institutions side by side.

13.8.5 Step 2: the checker

Listing 13.3 is the checker in Python. It has two sources behind one interface: `HttpSource` talks to the API (or to any server that answers the same three calls), and `SampleSource` answers from saved files, imitating a fuzzy name search by returning every saved record that shares at least one word with the query. Everything after the source, from the guard to the report, is the same code whichever source is used.

Listing 13.3 `researcher_check.py`: the OpenAlex researcher-record checker, with the exact-name guard, the rates of Equation (13.6) and an honest network failure.

```
"""Lab G9: an OpenAlex researcher-record checker (a teaching tool, not a ranking).

usage: python researcher_check.py [--sample | --base URL] "Full Name"
  --sample    read the saved responses in ../samples instead of the network
  --base URL  another API root, e.g. http://127.0.0.1:7130 (sample_server.py)
Set OPENALEX_API_KEY to use a free OpenAlex key (optional).
"""
import json
import os
import sys
import urllib.error
import urllib.parse
import urllib.request

from normalize import norm_name, norm_text

API = "https://api.openalex.org"
SAMPLES = os.path.join(os.path.dirname(os.path.abspath(__file__)),
                       "..", "samples")


class SampleSource:
    """Answers from saved JSON files, imitating a fuzzy name search."""
    label = "saved sample (fetched from OpenAlex on 2026-09-24)"

    def _authors(self):
        for f in sorted(os.listdir(SAMPLES)):
            if f.startswith("author_") and f.endswith(".json"):
                with open(os.path.join(SAMPLES, f), encoding="utf-8") as fh:
                    yield json.load(fh)

    def search(self, name):
        words = set(norm_text(name).split())
        hits = []
        for a in self._authors():
            known = [a["display_name"]] + a["display_name_alternatives"]
            if any(words & set(norm_text(k).split()) for k in known):
                hits.append(a)
        return hits

    def author(self, short_id):
        with open(os.path.join(SAMPLES, f"author_{short_id}.json"),
                  encoding="utf-8") as fh:
            return json.load(fh)
```

Listing 13.3 (Continued) *researcher_check.py: the OpenAlex researcher-record checker, with the exact-name guard, the rates of Equation (13.6) and an honest network failure.*

```
    def works(self, short_id):
        with open(os.path.join(SAMPLES, f"works_{short_id}.json"),
                  encoding="utf-8") as fh:
            return json.load(fh)["results"]


class HttpSource:
    """Answers from the OpenAlex REST API (or a local stand-in)."""

    def __init__(self, base):
        self.base = base.rstrip("/")
        self.label = self.base

    def _get(self, path, params):
        key = os.environ.get("OPENALEX_API_KEY")
        if key:
            params = dict(params, api_key=key)
        url = f"{self.base}{path}?{urllib.parse.urlencode(params)}"
        req = urllib.request.Request(
            url, headers={"User-Agent": "profitina-book-lab-g9/1.0"})
        with urllib.request.urlopen(req, timeout=20) as resp:
            return json.load(resp)

    def search(self, name):
        return self._get("/authors", {"search": name, "per_page": 5})["results"]

    def author(self, short_id):
        return self._get(f"/authors/{short_id}", {})

    def works(self, short_id):
        return self._get("/works", {"filter": f"author.id:{short_id}",
                                    "per_page": 100})["results"]


def pct(part, whole):
    return (200 * part + whole) // (2 * whole) if whole else 0
```

Listing 13.3 (Continued) *researcher_check.py: the OpenAlex researcher-record checker, with the exact-name guard, the rates of Equation (13.6) and an honest network failure.*

```
def exact_match(name, candidates):
    """Negative control: accept only a candidate whose name really matches."""
    want = norm_name(name)
    for a in candidates:
        known = [a["display_name"]] + a.get("display_name_alternatives", [])
        if want in {norm_name(k) for k in known}:
            return a
    return None


def check(source, name):
    print("OpenAlex researcher-record check (a teaching tool, not a ranking)")
    print(f"query : {name}")
    print(f"source: {source.label}")
    candidates = source.search(name)
    found = exact_match(name, candidates)
    if found is None:
        print(f"result: NOT FOUND ({len(candidates)} candidate(s), none with"
              " this exact name). No score is given.")
        return 1
    short_id = found["id"].rsplit("/", 1)[-1]
    a = source.author(short_id)
    works = source.works(short_id)
    print(f"match : {a['display_name']}  {a['id']}")

    def line(ok, text):
        print(f"  [{'ok ' if ok else 'gap'}] {text}")

    line(bool(a.get("orcid")), f"ORCID linked: {a.get('orcid') or 'none'}")
    line(a["works_count"] > 0, f"works indexed: {a['works_count']}")
    names = [a["display_name"]] + a["display_name_alternatives"]
    n_keys = len({norm_name(n) for n in names})
    line(n_keys <= 2, f"name spellings: {len(set(names))} raw,"
                      f" {n_keys} after normalization")
    n = len(works)
    with_doi = sum(1 for w in works if w.get("doi"))
    with_oa = sum(1 for w in works if w["open_access"]["is_oa"])
    print(f"  works examined: {n}")
    line(pct(with_doi, n) >= 90, f"DOI present: {with_doi}/{n}"
                                 f" ({pct(with_doi, n)}%)")
    line(pct(with_oa, n) >= 50, f"open access: {with_oa}/{n}"
                                f" ({pct(with_oa, n)}%)")
    raw, insts = [], set()
    for w in works:
        for au in w["authorships"]:
            if au["author"]["id"] == a["id"]:
```

Listing 13.3 (Continued) *researcher_check.py: the OpenAlex researcher-record checker, with the exact-name guard, the rates of Equation (13.6) and an honest network failure.*

```
                raw += au["raw_affiliation_strings"]
                insts |= {i["display_name"] for i in au["institutions"]}
    n_norm = len({norm_text(r) for r in raw})
    line(n_norm <= len(insts), f"affiliation strings: {len(raw)} raw,"
         f" {n_norm} after normalization, for {len(insts)} institutions")
    return 0


def main(argv):
    source, args = HttpSource(API), list(argv)
    if args and args[0] == "--sample":
        source, args = SampleSource(), args[1:]
    elif len(args) > 1 and args[0] == "--base":
        source, args = HttpSource(args[1]), args[2:]
    if len(args) != 1:
        print(__doc__.strip())
        return 2
    try:
        return check(source, args[0])
    except (urllib.error.URLError, OSError) as err:
        print(f"network unavailable: {err}")
        print("Nothing was scored; rerun with --sample to use the saved"
              " responses.")
        return 2


if __name__ == "__main__":
    sys.exit(main(sys.argv[1:]))
```

The Java version needs a JSON reader, because the Java standard library has none. **Listing 13.4** is a small recursive-descent parser: one method per kind of JSON value, each consuming exactly the characters of that value and calling the others for nested values. It reads objects into `LinkedHashMap`, arrays into `ArrayList`, integers into `Long` and other numbers into `Double`, and it can also write values back, which the stand-in server of Step 3 needs.

Listing 13.4 `Json.java`: a small recursive-descent JSON reader and writer, enough for the lab.

```
import java.util.ArrayList;
import java.util.LinkedHashMap;
import java.util.List;
import java.util.Map;

/**
 * A small JSON reader and writer, enough for the OpenAlex fields of Lab G9.
 * Objects become LinkedHashMap, arrays ArrayList, numbers Long or Double.
 */
public final class Json {
    private final String s;
    private int i;

    private Json(String s) {
        this.s = s;
    }

    public static Object parse(String text) {
        Json p = new Json(text);
        Object v = p.value();
        p.ws();
        if (p.i != p.s.length()) throw p.error("trailing characters");
        return v;
    }

    private RuntimeException error(String what) {
        return new IllegalArgumentException(what + " at offset " + i);
    }

    private void ws() {
        while (i < s.length() && Character.isWhitespace(s.charAt(i))) i++;
    }

    private Object value() {
        ws();
        if (i >= s.length()) throw error("unexpected end");
        char c = s.charAt(i);
        if (c == '{') return object();
        if (c == '[') return array();
        if (c == '"') return string();
        if (s.startsWith("true", i)) { i += 4; return Boolean.TRUE; }
        if (s.startsWith("false", i)) { i += 5; return Boolean.FALSE; }
        if (s.startsWith("null", i)) { i += 4; return null; }
        return number();
    }
```

Listing 13.4 (Continued) *Json.java: a small recursive-descent JSON reader and writer, enough for the lab.*

```
    private Map<String, Object> object() {
        Map<String, Object> m = new LinkedHashMap<>();
        i++;                                          // skip '{'
        ws();
        if (s.charAt(i) == '}') { i++; return m; }
        while (true) {
            ws();
            String key = string();
            ws();
            if (s.charAt(i++) != ':') throw error("expected ':'");
            m.put(key, value());
            ws();
            char c = s.charAt(i++);
            if (c == '}') return m;
            if (c != ',') throw error("expected ',' or '}'");
        }
    }

    private List<Object> array() {
        List<Object> a = new ArrayList<>();
        i++;                                          // skip '['
        ws();
        if (s.charAt(i) == ']') { i++; return a; }
        while (true) {
            a.add(value());
            ws();
            char c = s.charAt(i++);
            if (c == ']') return a;
            if (c != ',') throw error("expected ',' or ']'");
        }
    }

    private String string() {
        if (s.charAt(i++) != '"') throw error("expected a string");
        StringBuilder b = new StringBuilder();
        while (true) {
            char c = s.charAt(i++);
            if (c == '"') return b.toString();
            if (c != '\\') { b.append(c); continue; }
            char e = s.charAt(i++);
            switch (e) {
                case 'n' -> b.append('\n');
                case 't' -> b.append('\t');
                case 'r' -> b.append('\r');
                case 'b' -> b.append('\b');
                case 'f' -> b.append('\f');
```

Listing 13.4 (Continued) *Json.java: a small recursive-descent JSON reader and writer, enough for the lab.*

```
                case 'u' -> {
                    b.append((char) Integer.parseInt(s.substring(i, i + 4), 16));
                    i += 4;
                }
                default -> b.append(e);             // \" \\ \/
            }
        }
    }

    private Object number() {
        int start = i;
        while (i < s.length() && "+-0123456789.eE".indexOf(s.charAt(i)) >= 0) i++;
        String t = s.substring(start, i);
        if (t.isEmpty()) throw error("unexpected character");
        if (t.matches("-?\\d+")) return Long.parseLong(t);
        return Double.parseDouble(t);
    }

    /** Write a value back as compact JSON text. */
    public static String write(Object v) {
        if (v == null) return "null";
        if (v instanceof String str) {
            StringBuilder b = new StringBuilder("\"");
            for (char c : str.toCharArray()) {
                if (c == '"' || c == '\\') b.append('\\').append(c);
                else if (c < 0x20) b.append(String.format("\\u%04x", (int) c));
                else b.append(c);
            }
            return b.append('"').toString();
        }
        if (v instanceof Map<?, ?> m) {
            List<String> parts = new ArrayList<>();
            for (Map.Entry<?, ?> e : m.entrySet()) {
                parts.add(write(e.getKey().toString()) + ":" + write(e.getValue()));
            }
            return "{" + String.join(",", parts) + "}";
        }
        if (v instanceof List<?> l) {
            List<String> parts = new ArrayList<>();
            for (Object o : l) parts.add(write(o));
            return "[" + String.join(",", parts) + "]";
        }
        return v.toString();                        // numbers and booleans
    }
}
```

Listing 13.5 is the checker itself in Java, built on `java.net.http.HttpClient` [13] and the parser above.

Listing 13.5 `ResearcherCheck.java`: the same checker in Java, with a `Source` interface and two implementations.

```java
import java.io.IOException;
import java.net.URI;
import java.net.URLEncoder;
import java.net.http.HttpClient;
import java.net.http.HttpRequest;
import java.net.http.HttpResponse;
import java.nio.charset.StandardCharsets;
import java.nio.file.Files;
import java.nio.file.Path;
import java.time.Duration;
import java.util.ArrayList;
import java.util.HashSet;
import java.util.List;
import java.util.Map;
import java.util.Set;
import java.util.stream.Stream;

/**
 * Lab G9: an OpenAlex researcher-record checker (a teaching tool, not a ranking).
 * usage: java ResearcherCheck [--sample | --base URL] "Full Name"
 * Run it from the java folder so that ../samples is found.
 */
public class ResearcherCheck {
    static final String API = "https://api.openalex.org";
    static final Path SAMPLES = Path.of("..", "samples");

    interface Source {
        String label();
        List<Map<String, Object>> search(String name) throws IOException;
        Map<String, Object> author(String shortId) throws IOException;
        List<Map<String, Object>> works(String shortId) throws IOException;
    }

    @SuppressWarnings("unchecked")
    static Map<String, Object> obj(Object o) {
        return (Map<String, Object>) o;
    }

    @SuppressWarnings("unchecked")
    static List<Object> list(Object o) {
        return (List<Object>) o;
    }
```

Listing 13.5 (Continued) *ResearcherCheck.java: the same checker in Java, with a Source interface and two implementations.*

```
    static List<Map<String, Object>> objects(Object o) {
        List<Map<String, Object>> out = new ArrayList<>();
        for (Object x : list(o)) out.add(obj(x));
        return out;
    }

    static List<String> knownNames(Map<String, Object> a) {
        List<String> known = new ArrayList<>();
        known.add((String) a.get("display_name"));
        Object alt = a.get("display_name_alternatives");
        if (alt != null) for (Object x : list(alt)) known.add((String) x);
        return known;
    }

    /** Answers from saved JSON files, imitating a fuzzy name search. */
    static class SampleSource implements Source {
        public String label() {
            return "saved sample (fetched from OpenAlex on 2026-09-24)";
        }

        static Map<String, Object> read(Path p) throws IOException {
            return obj(Json.parse(Files.readString(p, StandardCharsets.UTF_8)));
        }

        public List<Map<String, Object>> search(String name) throws IOException {
            Set<String> words = Set.of(Normalize.normText(name).split(" "));
            List<Path> files;
            try (Stream<Path> st = Files.list(SAMPLES)) {
                files = st.filter(p -> p.getFileName().toString()
                                        .matches("author_.*\\.json"))
                          .sorted().toList();
            }
            List<Map<String, Object>> hits = new ArrayList<>();
            for (Path f : files) {
                Map<String, Object> a = read(f);
                boolean hit = false;
                for (String k : knownNames(a)) {
                    for (String w : Normalize.normText(k).split(" ")) {
                        hit |= words.contains(w);
                    }
                }
                if (hit) hits.add(a);
            }
            return hits;
        }
```

Listing 13.5 (Continued) *ResearcherCheck.java: the same checker in Java, with a Source interface and two implementations.*

```
        public Map<String, Object> author(String shortId) throws IOException {
            return read(SAMPLES.resolve("author_" + shortId + ".json"));
        }

        public List<Map<String, Object>> works(String shortId) throws IOException {
            return objects(read(SAMPLES.resolve("works_" + shortId + ".json"))
                           .get("results"));
        }
    }

    /** Answers from the OpenAlex REST API (or a local stand-in). */
    static class HttpSource implements Source {
        final String base;
        final HttpClient client = HttpClient.newBuilder()
            .connectTimeout(Duration.ofSeconds(20)).build();

        HttpSource(String base) {
            this.base = base.replaceAll("/+$", "");
        }

        public String label() {
            return base;
        }

        Map<String, Object> get(String path, String query) throws IOException {
            String key = System.getenv("OPENALEX_API_KEY");
            if (key != null && !key.isEmpty()) query += "&api_key=" + enc(key);
            HttpRequest req = HttpRequest.newBuilder(URI.create(
                    base + path + "?" + query))
                .header("User-Agent", "profitina-book-lab-g9/1.0")
                .timeout(Duration.ofSeconds(20)).build();
            try {
                HttpResponse<String> resp = client.send(req,
                    HttpResponse.BodyHandlers.ofString(StandardCharsets.UTF_8));
                if (resp.statusCode() != 200) {
                    throw new IOException("HTTP Error " + resp.statusCode());
                }
                return obj(Json.parse(resp.body()));
            } catch (InterruptedException e) {
                Thread.currentThread().interrupt();
                throw new IOException("interrupted");
            }
        }
```

Listing 13.5 (Continued) *ResearcherCheck.java: the same checker in Java, with a Source interface and two implementations.*

```
        static String enc(String s) {
            return URLEncoder.encode(s, StandardCharsets.UTF_8);
        }

        public List<Map<String, Object>> search(String name) throws IOException {
            return objects(get("/authors", "search=" + enc(name) + "&per_page=5")
                           .get("results"));
        }

        public Map<String, Object> author(String shortId) throws IOException {
            return get("/authors/" + shortId, "");
        }

        public List<Map<String, Object>> works(String shortId) throws IOException {
            return objects(get("/works", "filter=" + enc("author.id:" + shortId)
                               + "&per_page=100").get("results"));
        }
    }

    static long pct(long part, long whole) {
        return whole == 0 ? 0 : (200 * part + whole) / (2 * whole);
    }

    /** Negative control: accept only a candidate whose name really matches. */
    static Map<String, Object> exactMatch(String name,
                                          List<Map<String, Object>> cands) {
        String want = Normalize.normName(name);
        for (Map<String, Object> a : cands) {
            for (String k : knownNames(a)) {
                if (Normalize.normName(k).equals(want)) return a;
            }
        }
        return null;
    }

    static void line(boolean ok, String text) {
        System.out.println("  [" + (ok ? "ok " : "gap") + "] " + text);
    }
```

Listing 13.5 (Continued) *ResearcherCheck.java: the same checker in Java, with a Source interface and two implementations.*

```
    static int check(Source source, String name) throws IOException {
        System.out.println(
            "OpenAlex researcher-record check (a teaching tool, not a ranking)");
        System.out.println("query : " + name);
        System.out.println("source: " + source.label());
        List<Map<String, Object>> candidates = source.search(name);
        Map<String, Object> found = exactMatch(name, candidates);
        if (found == null) {
            System.out.println("result: NOT FOUND (" + candidates.size()
                + " candidate(s), none with this exact name). No score is given.");
            return 1;
        }
        String id = (String) found.get("id");
        String shortId = id.substring(id.lastIndexOf('/') + 1);
        Map<String, Object> a = source.author(shortId);
        List<Map<String, Object>> works = source.works(shortId);
        System.out.println("match : " + a.get("display_name") + "  " + a.get("id"));

        Object orcid = a.get("orcid");
        line(orcid != null, "ORCID linked: " + (orcid == null ? "none" : orcid));
        long worksCount = (Long) a.get("works_count");
        line(worksCount > 0, "works indexed: " + worksCount);
        List<String> names = knownNames(a);
        Set<String> nameKeys = new HashSet<>();
        for (String n : names) nameKeys.add(Normalize.normName(n));
        line(nameKeys.size() <= 2, "name spellings: " + new HashSet<>(names).size()
             + " raw, " + nameKeys.size() + " after normalization");

        int n = works.size(), withDoi = 0, withOa = 0;
        for (Map<String, Object> w : works) {
            if (w.get("doi") != null) withDoi++;
            if ((Boolean) obj(w.get("open_access")).get("is_oa")) withOa++;
        }
        System.out.println("  works examined: " + n);
        line(pct(withDoi, n) >= 90, "DOI present: " + withDoi + "/" + n
             + " (" + pct(withDoi, n) + "%)");
        line(pct(withOa, n) >= 50, "open access: " + withOa + "/" + n
             + " (" + pct(withOa, n) + "%)");
```

Listing 13.5 (Continued) *ResearcherCheck.java: the same checker in Java, with a Source interface and two implementations.*

```
        List<String> raw = new ArrayList<>();
        Set<String> insts = new HashSet<>();
        for (Map<String, Object> w : works) {
            for (Map<String, Object> au : objects(w.get("authorships"))) {
                if (!obj(au.get("author")).get("id").equals(a.get("id"))) continue;
                for (Object r : list(au.get("raw_affiliation_strings"))) {
                    raw.add((String) r);
                }
                for (Map<String, Object> i : objects(au.get("institutions"))) {
                    insts.add((String) i.get("display_name"));
                }
            }
        }
        Set<String> normed = new HashSet<>();
        for (String r : raw) normed.add(Normalize.normText(r));
        line(normed.size() <= insts.size(), "affiliation strings: " + raw.size()
             + " raw, " + normed.size() + " after normalization, for "
             + insts.size() + " institutions");
        return 0;
    }

    public static void main(String[] args) {
        Source source = new HttpSource(API);
        List<String> rest = new ArrayList<>(List.of(args));
        if (!rest.isEmpty() && rest.get(0).equals("--sample")) {
            source = new SampleSource();
            rest.remove(0);
        } else if (rest.size() > 1 && rest.get(0).equals("--base")) {
            source = new HttpSource(rest.get(1));
            rest = rest.subList(2, rest.size());
        }
        if (rest.size() != 1) {
            System.out.println("usage: java ResearcherCheck [--sample | --base URL]"
                + " \"Full Name\"");
            System.exit(2);
        }
        try {
            System.exit(check(source, rest.get(0)));
        } catch (IOException err) {
            System.out.println("network unavailable: " + err);
            System.out.println("Nothing was scored; rerun with --sample to use the"
                + " saved responses.");
            System.exit(2);
        }
    }
}
```

13.8.6 Running it, and what happened to the network

The programs were run on 24 September 2026. The machine used to prepare this chapter reaches the internet through a proxy, and the proxy refused connections to `api.openalex.org`. The checker did what it was written to do. Python printed:

```
OpenAlex researcher-record check (a teaching tool, not a ranking)
query : Heather Piwowar
source: https://api.openalex.org
network unavailable: <urlopen error Tunnel connection failed: 403 Forbidden>
Nothing was scored; rerun with --sample to use the saved responses.
```

and Java, whose HTTP client words the same failure differently, printed:

```
OpenAlex researcher-record check (a teaching tool, not a ranking)
query : Heather Piwowar
source: https://api.openalex.org
network unavailable: java.io.IOException: Tunnel failed, got: 403
Nothing was scored; rerun with --sample to use the saved responses.
```

Nothing was scored, nothing was invented, and the message says exactly what to do next. This is the lab's first result, and in a sense its most important one.

The runs below therefore use **saved responses**, in the folder `samples` next to the code. The researcher chosen is Heather Piwowar, a co-founder of OurResearch and of OpenAlex itself, whose record is public and whose work on open data and open access is well known [14]. The file `author_A5048491430.json` is a trimmed copy of her researcher record as returned by `GET /authors/A5048491430` on 24 September 2026, keeping only the fields the checker reads. The file `works_A5048491430.json` holds five of her works, each fetched from the API on the same day by its DOI and trimmed to her own authorship; e-mail addresses that appeared inside two affiliation strings were removed. The live list endpoint would return all of her indexed works, 99 at that date; the sample has five, and the report says "works examined: 5" so that nobody mistakes the sample for the whole. Each file begins with a `_note` field that says all of this, so the data carries its own provenance.

With the saved sample, `python researcher_check.py --sample "Heather Piwowar"` and `java ResearcherCheck --sample "Heather Piwowar"` both print:

```
OpenAlex researcher-record check (a teaching tool, not a ranking)
query : Heather Piwowar
source: saved sample (fetched from OpenAlex on 2026-09-24)
match : Heather Piwowar  https://openalex.org/A5048491430
  [ok ] ORCID linked: https://orcid.org/0000-0003-1613-5981
  [ok ] works indexed: 99
  [gap] name spellings: 7 raw, 3 after normalization
  works examined: 5
  [ok ] DOI present: 5/5 (100%)
  [ok ] open access: 5/5 (100%)
  [gap] affiliation strings: 9 raw, 8 after normalization, for 6 institutions
```

Read the report line by line. The ORCID is linked, which is the single most useful thing a researcher can do for their own findability. Ninety-nine works are indexed. The record carries seven name forms that normalize to three, the same three we saw in Step 1: a `gap`, but a mild and very common one. All five sampled works have a DOI and all five are open access, unsurprising for a researcher who has written about open access for years. And the last line is the interesting one: across five works, her affiliations appear as nine raw strings, eight after normalization, which OpenAlex resolved to six institutions. Two strings for the University of Pittsburgh differ only by a final period and merge; the rest are genuinely different, including a full sentence of biography that a publisher placed in an affiliation field. A human reading those strings would see four or five real places; a machine sees eight spellings. This is, in miniature, the problem that produced 466 spellings in 500 works at one university.

Now the negative control. A query that shares a word with a real record but is not that person's name:

```
OpenAlex researcher-record check (a teaching tool, not a ranking)
query : Piwowar
source: saved sample (fetched from OpenAlex on 2026-09-24)
result: NOT FOUND (1 candidate(s), none with this exact name). No score is given.
```

and a name invented for this book:

```
OpenAlex researcher-record check (a teaching tool, not a ranking)
query : Sekar Laundrawati
source: saved sample (fetched from OpenAlex on 2026-09-24)
result: NOT FOUND (0 candidate(s), none with this exact name). No score is given.
```

The first query had a candidate, the saved record matched the word “Piwowar”, and the guard rejected it because “piwowar” is not the normalized form of any of her recorded names. The second had no candidate at all. Both end the same way: NOT FOUND, no score given, and a non-zero exit code that a script can test.

13.8.7 Step 3: a local stand-in for the API

The saved-sample mode tests everything except the HTTP code. To exercise that too, without the internet, the lab includes a tiny server that answers the same three calls from the saved files on `127.0.0.1:7130`. **Listing 13.6** is the Python version, built on `http.server` [15].

Listing 13.6 `sample_server.py`: a local stand-in for the three OpenAlex calls, answering from the saved samples.

```
"""Lab G9, step 3: a local stand-in for the three OpenAlex calls we use.

It answers from the saved files in ../samples, so the HTTP code path of
researcher_check.py can be tried without the internet.
usage: python sample_server.py [port]      (default 7130; Ctrl+C to stop)
"""
import json
import sys
import urllib.parse
from http.server import BaseHTTPRequestHandler, ThreadingHTTPServer

from researcher_check import SampleSource

SOURCE = SampleSource()


class Handler(BaseHTTPRequestHandler):
    def do_GET(self):
        url = urllib.parse.urlsplit(self.path)
        q = urllib.parse.parse_qs(url.query)
        try:
            if url.path == "/authors" and "search" in q:
                hits = SOURCE.search(q["search"][0])
                body = {"meta": {"count": len(hits)}, "results": hits}
            elif url.path.startswith("/authors/"):
                body = SOURCE.author(url.path.split("/")[-1])
            elif url.path == "/works" and "filter" in q:
                short_id = q["filter"][0].split(":", 1)[1]
                results = SOURCE.works(short_id)
                body = {"meta": {"count": len(results)}, "results": results}
            else:
                return self.send_error(404, "unknown route")
        except FileNotFoundError:
            return self.send_error(404, "no saved sample for this id")
        data = json.dumps(body).encode("utf-8")
        self.send_response(200)
        self.send_header("Content-Type", "application/json; charset=utf-8")
        self.send_header("Content-Length", str(len(data)))
        self.end_headers()
        self.wfile.write(data)

    def log_message(self, fmt, *args):
        print("served", self.path.split("?")[0], file=sys.stderr)
```

Listing 13.6 (Continued) *sample_server.py: a local stand-in for the three OpenAlex calls, answering from the saved samples.*

```
def main():
    port = int(sys.argv[1]) if len(sys.argv) > 1 else 7130
    server = ThreadingHTTPServer(("127.0.0.1", port), Handler)
    print(f"sample OpenAlex stand-in on http://127.0.0.1:{port}")
    try:
        server.serve_forever()
    except KeyboardInterrupt:
        server.server_close()


if __name__ == "__main__":
    main()
```

Listing 13.7 is the Java version, built on the JDK's `com.sun.net.httpserver` package.

Listing 13.7 `SampleServer.java`: the same stand-in server in Java.

```
import com.sun.net.httpserver.HttpExchange;
import com.sun.net.httpserver.HttpServer;
import java.io.IOException;
import java.io.OutputStream;
import java.net.InetSocketAddress;
import java.net.URLDecoder;
import java.nio.charset.StandardCharsets;
import java.nio.file.NoSuchFileException;
import java.util.LinkedHashMap;
import java.util.List;
import java.util.Map;

/**
 * Lab G9, step 3: a local stand-in for the three OpenAlex calls we use.
 * It answers from ../samples. usage: java SampleServer [port]   (default 7130)
 */
public class SampleServer {
    static final ResearcherCheck.SampleSource SOURCE =
        new ResearcherCheck.SampleSource();

    static Map<String, String> query(String raw) {
        Map<String, String> q = new LinkedHashMap<>();
        if (raw == null) return q;
        for (String pair : raw.split("&")) {
            int eq = pair.indexOf('=');
            if (eq > 0) {
                String k = pair.substring(0, eq), v = pair.substring(eq + 1);
                q.put(URLDecoder.decode(k, StandardCharsets.UTF_8),
                      URLDecoder.decode(v, StandardCharsets.UTF_8));
            }
        }
        return q;
    }

    static Map<String, Object> page(List<Map<String, Object>> results) {
        Map<String, Object> body = new LinkedHashMap<>();
        body.put("meta", Map.of("count", results.size()));
        body.put("results", results);
        return body;
    }
```

Listing 13.7 (Continued) *SampleServer.java: the same stand-in server in Java.*

```
    static void handle(HttpExchange ex) throws IOException {
        String path = ex.getRequestURI().getPath();
        Map<String, String> q = query(ex.getRequestURI().getRawQuery());
        System.err.println("served " + path);
        Object body;
        int status = 200;
        try {
            if (path.equals("/authors") && q.containsKey("search")) {
                body = page(SOURCE.search(q.get("search")));
            } else if (path.startsWith("/authors/")) {
                body = SOURCE.author(path.substring(path.lastIndexOf('/') + 1));
            } else if (path.equals("/works") && q.containsKey("filter")) {
                String shortId = q.get("filter").split(":", 2)[1];
                body = page(SOURCE.works(shortId));
            } else {
                status = 404;
                body = Map.of("error", "unknown route");
            }
        } catch (NoSuchFileException e) {
            status = 404;
            body = Map.of("error", "no saved sample for this id");
        }
        byte[] data = Json.write(body).getBytes(StandardCharsets.UTF_8);
        ex.getResponseHeaders().set("Content-Type",
                                    "application/json; charset=utf-8");
        ex.sendResponseHeaders(status, data.length);
        try (OutputStream out = ex.getResponseBody()) {
            out.write(data);
        }
    }

    public static void main(String[] args) throws IOException {
        int port = args.length > 0 ? Integer.parseInt(args[0]) : 7130;
        HttpServer server = HttpServer.create(
            new InetSocketAddress("127.0.0.1", port), 0);
        server.createContext("/", SampleServer::handle);
        server.start();
        System.out.println("sample OpenAlex stand-in on http://127.0.0.1:" + port);
    }
}
```

Start either server in one terminal (`python sample_server.py` or `java SampleServer`), and in a second terminal run the checker against it with `--base http://127.0.0.1:7130`. The checker prints:

```
OpenAlex researcher-record check (a teaching tool, not a ranking)
query : Heather Piwowar
source: http://127.0.0.1:7130
match : Heather Piwowar  https://openalex.org/A5048491430
  [ok ] ORCID linked: https://orcid.org/0000-0003-1613-5981
  [ok ] works indexed: 99
  [gap] name spellings: 7 raw, 3 after normalization
  works examined: 5
  [ok ] DOI present: 5/5 (100%)
  [ok ] open access: 5/5 (100%)
  [gap] affiliation strings: 9 raw, 8 after normalization, for 6 institutions
```

and the server logs the three requests it answered:

```
sample OpenAlex stand-in on http://127.0.0.1:7130
served /authors
served /authors/A5048491430
served /works
```

The report is identical to the saved-sample report except for the `source` line, which proves that the HTTP path, the JSON parsing and the query encoding all work. The two languages are interchangeable here as well: the Python checker against the Java server, and the Java checker against the Python server, produce the same report. Stop the server with Ctrl+C when you are done. On a machine that can reach OpenAlex, running the checker with no option at all queries the live catalogue, and the numbers will be those of the day you run it.

> **WARNING.** Use the checker on your own record, or on a public scholar for learning, never to compare or rank people. A record's gaps say something about metadata, not about the quality of anyone's research. Keep requests few, identify your program with a clear User-Agent, and use an API key for anything more than a handful of checks.

13.8.8 Taking the checker further

The checker is deliberately small, and every limit it has is an invitation. Five extensions are worth doing, in roughly this order, and each one teaches something that the earlier chapters prepared.

Disambiguate with ORCID. Accept an optional `--orcid` argument, and when it is given, let the guard accept only a candidate whose `orcid` field matches it. Two real people with the same name then stop being a problem (Exercise 7 asks why they are one now). This is the same move that Chapter 11 made for personal data: identify a person by a stable identifier they control, not by a string that others also carry.

Read every page. The live works endpoint returns results in pages. A researcher with 400 works needs several requests, and the checker must follow the paging instructions in the response until there are no more. The loop is short, but it must stop on its own, count what it has seen, and say so in the report, so that "works examined" is always true.

Be polite under pressure. When OpenAlex answers `429 Too Many Requests`, the right response is not to retry at once. Wait, then retry, and double the wait each time up to a limit: *exponential backoff*. After a few failures, stop and report, exactly like the circuit breaker of Section 13.2.2. Add a small on-disk cache as well, so that checking the same person twice in one day costs no second request; this is the same politeness Chapter 3 asked of the readiness checker.

Break the rates down. Open access has several colors (gold, green, hybrid, bronze), and a researcher who can do nothing about a publisher's policy can often deposit an accepted manuscript in a repository. Printing the breakdown turns one number into advice a person can act on. The same holds for DOIs: listing the titles of the works without one is more useful than a percentage.

Aggregate, never rank. Organization Visibility works at the level of an institution: it counts spellings and missing identifiers across all of its researchers and reports the total. A reader who extends the lab in that direction should keep the same line: sum the gaps, show how many can be closed, and never print a list of people ordered by anything. The report that found 466 spellings in 500 works helped an entire university fix its metadata precisely because it pointed at a shared habit, not at individuals.

Each extension keeps the three principles. The checker stays a local program with no hidden service; it advises, and the researcher decides what to fix; and each extension deserves a test that pins down its behavior, starting with the most important one of all, that a name which does not exist still returns NOT FOUND.

13.9 What Comes Next

Figure 13.4 places the road ahead after the road behind, with a clear change of line where fact ends and plan begins.

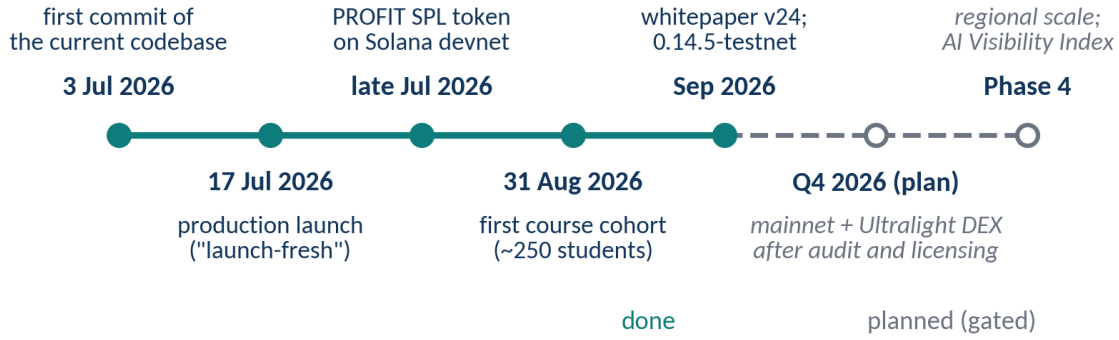


Figure 13.4 The road behind (solid) and the road ahead (dashed). Everything to the left of September 2026 has happened; everything to the right is planned and gated.

13.9.1 The road behind

The dates on the solid line are the ones this book has told: the first commit of the current codebase on 3 July 2026; the production launch on 17 July, with the database reset to zero ("launch-fresh"); the PROFIT token on Solana devnet in late July; the first university cohort of about 250 students on 31 August; and, in September, the public whitepaper v24 and release 0.14.5-testnet, committed on 24 September 2026, the day this chapter was written. Between those dates lie the features of Chapters 3 to 12, each one shipped, measured and, where it fell short, corrected.

13.9.2 The road ahead

Phase 3: PROFIT mainnet and the Ultralight DEX. The whitepaper labels this phase Q4 2026 and attaches its gate in the same sentence: a token audit and licensing under Indonesia's regulators [7]. Chapter 12 explained why the gate matters more than the label. When the conditions are met, PROFIT moves to mainnet, the mint authority to a multisig, and non-custodial swaps through the Jupiter aggregator become possible. Until then, selling stays disabled.

Phase 4: regional scale and an index. After leadership at home, the whitepaper plans expansion to the region and a "Profitina AI Visibility Index" [7]. The order is deliberate: the advantages the whitepaper claims (the Indonesian language, UU PDP compliance, QRIS) are local ones, and they must be real at home before they are exported.

Privacy law coming into force. The implementing regulation of UU PDP, PP No. 33 of 2026, takes effect on 16 January 2027 (Chapter 11). A product built on the principles of this book should meet it without drama, and the release gate is where that will be checked.

The model loop. The feedback pipeline of Chapter 7 is live, and it can one day feed a fine-tuned local model. That step remains on the roadmap; when it comes, it will be released the way everything else is: measured against the current model, decided by a person, and shipped through the gate.

The teaching ground. Profitina Laundry's planned material (use cases, the entity-relationship diagram, the sample transactions and the per-chapter READMEs) grows with the books that use it.

13.9.3 What will not change

Plans move. Dates slip, phases merge, features are removed when measurement says they should be (Chapter 8 told how the Agency tier went). What should not move are the things this book has tried to show are principles rather than preferences: the model runs locally; the app advises and the user decides; nothing ships unless the tests are green and the health check says so; the product says what actually happened; the rights of Chapter 11 are free forever; and no paid price is ever written next to a promise of something not yet built. A reader who comes back to Profitina in a year should expect a different feature list and the same principles.

13.9.4 The labs, one last time

The road ahead for a reader is shorter and more certain than the product's: it runs through the labs. **Table 13.1** lists every lab of the book with the idea it teaches and the principle of Section 13.7 it serves best. Each is small enough to read in an evening and complete enough to extend for a semester.

Table 13.1 The labs of the book, the idea each teaches, and the principle it serves best.

Chapter	Lab	The idea	Principle it serves best
2	G6: JSON-LD, robots.txt and llms.txt generator	Machine-readable facts about a business	The app advises; the user publishes
3	G1: mini AI-readiness checker	Observable signals, explained one by one	Tell the truth: every check shows its evidence
4	G2: live feed over Server-Sent Events	One-way push, invalidate versus refetch	Tests as a gate for real-time behavior
5	G3: rule-based link screener	A deterministic rule floor, measured on a labeled corpus	Measure before believing
6	G5: local model over Ollama's HTTP API	Honest fallback when no model is present	Local-first AI
7	Feedback pipeline	Submit, curate, dataset, measure, release	The user decides what ships
8	Plans, quotas and the subscription state machine	Days counted in WIB; no double credit	Tests as a gate
9	G4: scheduler and health check	A priority queue of due jobs; success only after health	Tests as a gate
10	Cache-header and robots auditor	Verify the live URL, bypassing the cache	Tell the truth about what is actually served
11	G7: consent log, export and delete	Rights as buttons; nothing personal left	The user decides about their own data
12	G8: emission and burn simulator	Power laws, geometric series, integer ledgers	Tell the truth: illustrative, never a price
13	G9: researcher-record checker	Negative control and normalized spellings	Tell the truth: not found means no score

Read down the last column and the three principles, with the fourth underneath them, appear again and again. That repetition is the point. A principle is learned the way a program is: by running it many times on different inputs until its behavior is familiar.

13.10 Profitina in Practice

In the September 2026 release, this chapter maps onto real products as follows.

- **The group.** One umbrella repository with six shared layers (interface languages, the Conductor, self-update, cloud and local image engines, the ledger of resting providers), copied down into the products and never back up; two machines with a pull-before, sync-after routine; a secret scanner at every commit.
- **The products.** Profitina (the GEO platform at app.profitina.com, release 0.14.5-testnet); Profitina-Deploy (the profitina.com theme); Profitina Capital 3.8 (private quantitative tooling with a risk desk); Profitina TTM 2.8.1 (text to short film, hybrid painting, fail loudly); Profitina AI 1.7.2 (local-first assistant, never stop silently). Profitina does not use Profitina AI or any cloud AI.
- **The teaching ground.** Profitina Laundry, fictional, in Java and Python, sharing ideas and never code or data.
- **The principles.** Local-first AI; the app advises and the user decides; tests as a gate (1,577 passed and 1 skipped on the full SQLite suite, 440 passed on the PostgreSQL 16 subset, at release 0.14.5); tell the truth about what happened.
- **The Researcher Visibility Report and Organization Visibility.** Built on OpenAlex, with a permanent negative control (8 of 8 fictional names at 0.0) and aggregate-only organization reports (466 spellings in 500 works; 244 of 607 without ORCID at one university) [7].
- **The road ahead.** Phase 3 (mainnet and the Ultralight DEX) gated by audit and licensing; Phase 4 (regional scale and an index) after domestic leadership.

Lab G9 reproduces none of the Researcher Visibility Report's code, weights or thresholds. It uses three public API calls, rules any reader can inspect, and a saved sample whose provenance is written inside it.

13.11 Common Misunderstandings

MISUNDERSTANDING 1: "Profitina uses Profitina AI, so it uses cloud AI." Profitina AI is a sibling product. Profitina, the GEO platform, runs its own local models through Ollama and uses no cloud AI at all.

MISUNDERSTANDING 2: "Local-first means local-only." In Profitina AI it means local by default, cloud when the user chooses. Only the GEO platform is strictly local.

MISUNDERSTANDING 3: "Profitina Laundry is a demo of Profitina's code." It is a fictional teaching application with no connection to the real product's code or data. It shares ideas and patterns only.

MISUNDERSTANDING 4: "Profitina Capital gives investment advice." It is private tooling that computes risk statistics. Statistics describe the past; the examples in this chapter are invented and advise nothing.

MISUNDERSTANDING 5: "If the search returns someone, the checker should score them." A search engine returns the nearest match even for a name that does not exist. The exact-name guard is what stops the checker from scoring a stranger; not found must mean no score.

MISUNDERSTANDING 6: "A researcher with more affiliation spellings is a worse researcher." Spellings measure metadata hygiene, mostly set by publishers and indexers, not the quality of research. The checker reports gaps to fix, never a rank.

MISUNDERSTANDING 7: "When the API is unreachable, returning zeros is harmless." Zeros would look like a real, bad result. The checker prints the network error and scores nothing, and the saved sample is labeled as a sample.

> **MISUNDERSTANDING 8: "Q4 2026 is the mainnet date."** It is a label with a gate attached: an audit and licensing. The gate decides; the label follows.

13.12 Chapter Summary

- The Profitina Group is an umbrella of sibling products sharing six layers, copied down and never back up (**Figure 13.1**); the ledger of resting providers is a shared circuit breaker.
- Profitina Capital 3.8 is a private quantitative desk with a risk desk. On ten invented days we computed value at risk (Equation (13.1)), expected shortfall (Equation (13.2)), maximum drawdown (Equation (13.3)), the Sharpe ratio (Equation (13.4)) and the Herfindahl–Hirschman index (Equation (13.5)).
- Profitina TTM 2.8.1 turns text into a short film through a planner, a director with six story beats, hybrid painting with a local diffusion engine, voice, procedural sound and ffmpeg, and it fails loudly.
- Profitina AI 1.7.2 is a local-first assistant with a Conductor, tools and a Studio; it never stops silently. It is not part of app.profitina.com, which uses no cloud AI.
- Profitina Laundry is the fictional teaching ground: safe, small, and sharing only ideas.
- Three principles bind the group (**Figure 13.2**): local-first AI; the app advises and the user decides; tests as a gate. Underneath runs a fourth: tell the truth about what happened.
- Lab G9 checks a researcher record in OpenAlex: search, an exact-name guard as negative control, the record, the works, normalized spellings counted with sets, and a report of `ok` and `gap` with rates by Equation (13.6) (**Figure 13.3**). When the network was blocked the checker said so and scored nothing; saved, labeled samples and a local stand-in server exercised every path.
- The labs of the book, each mapped to the principle it serves best, are collected in **Table 13.1**.
- The road ahead (**Figure 13.4**): Phase 3 (mainnet and the Ultralight DEX) after audit and licensing; Phase 4 (regional scale and an index) after domestic leadership; PP 33/2026 from 16 January 2027. Plans move; principles should not.

13.13 Exercises

1. **Concentration.** Compute the Herfindahl–Hirschman index of Equation (13.5) for weights 0.4,0.4,0.2 and its reciprocal. Then compute it for five equal weights. Which portfolio is more concentrated, and by how much?

2. **VaR and expected shortfall.** For the ten daily returns of Section 13.3.2, compute $\mathrm{VaR}_{90\%}$ and $\mathrm{CVaR}_{90\%}$ by Equation (13.1) and Equation (13.2), and then $\mathrm{VaR}_{70\%}$ and $\mathrm{CVaR}_{70\%}$.

3. **Maximum drawdown in one pass.** Describe an algorithm that computes Equation (13.3) for a series of n values in $O(n)$ time and $O(1)$ extra memory, and apply it to the values 100,110,99,105,90,120,108.

4. **Normalize by hand.** Apply `norm_text` of **Listing 13.1** to the four strings `"Universitas Indonesia"`, `"UNIVERSITAS INDONESIA, Depok"`, `"Universitas  Indonesia."` (with two spaces) and `"Univ. Indonesia"`. How many raw spellings are there, and how many after normalization?

5. **Where normalization stops.** The names “Soekarno” and “Sukarno” refer to the same person in two Indonesian spelling systems. What does `norm_name` make of them, and what kind of information, rather than a rule, would be needed to match them?
6. **Rounding the rates.** Use Equation (13.6) to compute pct(1,8), pct(1,3) and pct(2,3) exactly with integers. What does the program return when $w = 0$, and why is a guard needed there?
7. **The limits of the guard.** The exact-name guard stops the checker from scoring a stranger when the name does not exist. What goes wrong when two different real researchers have exactly the same name? Suggest one field of the researcher record that would help, and explain how.
8. **Name the principle.** For each practice, name the principle of Section 13.7 it serves: (a) TTM’s local diffusion engine joins only when it is at least as good and not slower; (b) a release counts as done only when `/api/health` reports the new version; (c) the Content Engine produces a draft that the user edits and publishes; (d) Profitina AI returns partial output when a task cannot finish; (e) the scan’s category detection says “not sure yet”.
9. **Thresholds.** The checker calls DOI coverage below 90% a gap. Argue for a different threshold for a researcher in a field where much work appears in books and national journals without DOIs, and say what the report should print so that the reader understands the choice.
10. **Word order.** Run the checker with the saved sample and the query `"piwowar, heather"`. Is it found? Which function makes that happen?

13.14 Solutions to the Exercises

Solution 1. $0.4^2 + 0.4^2 + 0.2^2 = 0.16 + 0.16 + 0.04 = 0.36$, reciprocal $1/0.36 \approx 2.78$. Five equal weights of 0.2: $5 \times 0.04 = 0.20$, reciprocal 5. The first portfolio is more concentrated: it behaves like fewer than three equal assets, while the second behaves like exactly five; its index is $0.36/0.20 = 1.8$ times as high.

Solution 2. Sorted returns: $-2.1, -1.4, -0.8, -0.6, -0.3, 0.5, 0.7, 0.9, 1.2, 1.5$. At 90% the worst 10% is one day, so $q_{10\%} = -2.1$: $\mathrm{VaR}_{90\%} = 2.1\%$, and the average of the losses at or beyond it is the same single day, $\mathrm{CVaR}_{90\%} = 2.1\%$. At 70% the worst 30% is three days: $q_{30\%} = -0.8$, so $\mathrm{VaR}_{70\%} = 0.8\%$ and $\mathrm{CVaR}_{70\%} = (2.1 + 1.4 + 0.8)/3 \approx 1.43\%$. With so few days, the two measures coincide at 90%; with more data they separate, which is why real desks need long histories.

Solution 3. Keep two variables: the running peak P (the largest value seen so far) and the best drawdown D found so far, initially $P = V_0$ and $D = 0$. For each next value V: set $P = \max(P, V)$, then $D = \max(D, (P - V)/P)$. Each value is visited once and only two numbers are stored, so the time is $O(n)$ and the extra memory $O(1)$. On 100,110,99,105,90,120,108: the peak becomes 110, the drawdown at 99 is 10%, at 105 it is 4.5%, at 90 it is $20/110 \approx 18.18\%$; the peak then becomes 120 and the drawdown at 108 is 10%. So MDD $\approx 18.18\%$.

Solution 4. The keys are `'universitas indonesia'`, `'universitas indonesia depok'`, `'universitas indonesia'` and `'univ indonesia'`. Four raw spellings become three after normalization: the capitalization, the trailing period and the double space disappear; the extra word “Depok” and the abbreviation “Univ.” remain, correctly for the first and unavoidably for the second.

Solution 5. `norm_name` returns `'soekarno'` and `'sukarno'`: two different keys, because the difference is in the letters themselves, not in case, accents, punctuation or word order. Matching them needs knowledge: a list of recorded alternative names (OpenAlex's `display_name_alternatives` is such a list for researchers), a persistent identifier such as ORCID that ties both spellings to one person, or a rule specific to the old Indonesian spelling (oe for u), which is knowledge written as a rule and would need care not to merge names that really differ.

Solution 6. pct(1,8) = ⌊(200 + 8)/16⌋ = ⌊13⌋ = 13 (12.5% rounds up to 13). pct(1,3) = ⌊(200 + 3)/6⌋ = ⌊33.83⌋ = 33. pct(2,3) = ⌊(400 + 3)/6⌋ = ⌊67.17⌋ = 67. When $w = 0$ the formula would divide by zero; both programs return 0 in that case, and the report still prints the fraction (for example `0/0`) so that the reader sees that there was nothing to count rather than a real 0%.

Solution 7. If two real researchers share the exact name, both pass the guard, and the checker takes the first candidate, which may be the wrong person. The guard protects against inventing a person, not against confusing two real ones. The ORCID field helps: if the user supplies their ORCID, the checker can accept only the candidate whose `orcid` matches it, which identifies one person uniquely. The last known institution is a weaker but useful second signal.

Solution 8. (a) Local-first AI, kept honest by measurement. (b) Tests as a gate. (c) The app advises; the user decides. (d) Tell the truth about what happened ("never stop silently"). (e) Tell the truth about what happened, and also the app advises and the user decides, since the user is asked to pick the category.

Solution 9. In fields where books, chapters and national journals often lack DOIs, 90% would flag almost everyone and teach nothing. A threshold such as 60%, or a report that shows the share without an `ok`/`gap` label for such fields, is more useful. Whatever the choice, the report should print the threshold next to the result, for example `DOI present: 12/20 (60%) - threshold 60% for this field`, so the reader can see that the label is a choice and disagree with it.

Solution 10. The run prints:

```
OpenAlex researcher-record check (a teaching tool, not a ranking)
query : piwowar, heather
source: saved sample (fetched from OpenAlex on 2026-09-24)
match : Heather Piwowar  https://openalex.org/A5048491430
  [ok ] ORCID linked: https://orcid.org/0000-0003-1613-5981
  [ok ] works indexed: 99
  [gap] name spellings: 7 raw, 3 after normalization
  works examined: 5
  [ok ] DOI present: 5/5 (100%)
  [ok ] open access: 5/5 (100%)
  [gap] affiliation strings: 9 raw, 8 after normalization, for 6 institutions
```

It is found. The sample source returns the record because the query shares words with it, and the guard accepts it because `norm_name` sorts the words: `"piwowar, heather"` and `"Heather Piwowar"` both become `'heather piwowar'`.

13.15 A Closing Word

This book began in a classroom and ends in one. Its first chapter told how a platform grew out of decades of teaching; its last lab asks a public catalogue whether a researcher can be found, and refuses to answer when it does not know. In between, it has tried to show one thing many times: that a real product is built from ideas a student can understand, a geometric series here, a set of normalized strings there, a state machine, a priority queue, a consent log, a release gate, and from the discipline to use them honestly.

Profitina will change after this book is printed. Versions will move past 0.14.5, the whitepaper past v24, the test count past 1,577, and some of the plans in Section 13.9 will arrive late or in a different shape. That is as it should be. What this book hopes will last is the way of working behind the numbers: measure before believing, tell the user what actually happened, let people decide, and let nothing ship until the tests say it may.

If you built the labs, you now own small working versions of the ideas behind a live platform, in two languages, with outputs you have checked yourself. Take them further. Point the readiness checker at your own site, the screener at your own messages, the simulator at a loyalty scheme for Profitina Laundry, the researcher checker at your own name. When something in them surprises you, or when you find a mistake in these pages, write to info@profitina.com. *Gather, grow, give*: the tagline asks for all three, and the last one is yours now.

References

[1] M. T. Nygard, *Release It! Design and Deploy Production-Ready Software*, 2nd ed. Raleigh, NC, USA: Pragmatic Bookshelf, 2018, ch. 5, “Stability Patterns.”

[2] R. T. Rockafellar and S. Uryasev, “Optimization of conditional value-at-risk,” *Journal of Risk*, vol. 2, no. 3, pp. 21–41, 2000.

[3] W. F. Sharpe, “The Sharpe ratio,” *The Journal of Portfolio Management*, vol. 21, no. 1, pp. 49–58, 1994.

[4] FFmpeg Developers, “FFmpeg Documentation.” [Online]. Available: https://ffmpeg.org/documentation.html

[5] S. Lin, A. Wang, and X. Yang, “SDXL-Lightning: Progressive adversarial diffusion distillation,” arXiv:2402.13929, 2024.

[6] L. Yang, B. Kang, Z. Huang, Z. Zhao, X. Xu, J. Feng, and H. Zhao, “Depth Anything V2,” arXiv:2406.09414, 2024.

[7] Profitina, *Profitina Whitepaper — Public Version v24* (ID/EN), Sep. 2026. [Online]. Available: https://app.profitina.com/static/whitepaper/

[8] J. Priem, H. Piwowar, and R. Orr, “OpenAlex: A fully-open index of scholarly works, authors, venues, institutions, and concepts,” arXiv:2205.01833, 2022.

[9] OpenAlex, “API keys required starting Feb 13 (and some new endpoints!),” *openalex-users* mailing list, Jan. 14, 2026. [Online]. Available: https://groups.google.com/g/openalex-users/c/rI1GIAySpVQ

[10] OpenAlex, “Authentication,” *OpenAlex API Reference*. [Online]. Available: https://developers.openalex.org/api-reference/authentication (accessed Sep. 24, 2026).

[11] Unicode Consortium, “Unicode Normalization Forms,” Unicode Standard Annex #15. [Online]. Available: https://www.unicode.org/reports/tr15/

[12] Oracle, “Class Normalizer,” *Java SE 21 & JDK 21 API Specification*, 2023. [Online]. Available: https://docs.oracle.com/en/java/javase/21/docs/api/java.base/java/text/Normalizer.html

[13] Oracle, “Class HttpClient,” *Java SE 21 & JDK 21 API Specification*, 2023. [Online]. Available: https://docs.oracle.com/en/java/javase/21/docs/api/java.net.http/java/net/http/HttpClient.html

[14] H. Piwowar, J. Priem, V. Larivière, J. P. Alperin, L. Matthias, B. Norlander, A. Farley, J. West, and S. Haustein, “The state of OA: a large-scale analysis of the prevalence and impact of Open Access articles,” *PeerJ*, vol. 6, p. e4375, 2018, doi: 10.7717/peerj.4375.

[15] Python Software Foundation, “http.server — HTTP servers,” *The Python Standard Library*. [Online]. Available: https://docs.python.org/3/library/http.server.html