STOP CREATING SLOP

Vibe Code Like a Pro

A PRACTICAL GUIDE FOR BEGINNERS

STOP CREATING SLOP

Vibe Code Like a Pro

Antony Kariuki

Build real software with AI · Without shipping slop

STOP CREATING SLOP: Vibe Code Like a Pro

Copyright © 2026 Antony Kariuki. All rights reserved.

First edition, version 1.0 · 2026–10–24 · ISBN to be assigned

No part of this publication may be reproduced, distributed, or transmitted in any form or by any means without the prior written permission of the author, except for brief quotations in a review or for personal study.

How to read the version notes

Software tools, model names, pricing, and interface details change frequently. Where a chapter states something version-dependent, it is marked **[verify current]**. The reasoning, standards, and practices in this book are designed to outlast the tools.

Typography and licensing

Set in Outfit, Plus Jakarta Sans, and JetBrains Mono, all licensed under the SIL Open Font License 1.1 and embedded in this edition. Outfit by Smartsheet Inc. and contributors; Plus Jakarta Sans by Tokotype; JetBrains Mono by JetBrains.

Standards referenced

OWASP Top 10:2021, W3C WCAG 2.1 (level AA), RFC 9110 (HTTP Semantics), RFC 6749 (OAuth 2.0), Kenya Data Protection Act 2019, and EU General Data Protection Regulation (Regulation 2016/679). Standards texts are quoted for reference only and remain the property of their issuing bodies.

Disclaimer

This book teaches engineering judgement, not legal, financial, or security assurance. Validate security, privacy, and compliance decisions for your own context before launch. Examples use a fictional Nairobi business, Mifugo Market, for illustration.

Produced with an HTML/CSS typesetting pipeline. Editable sources, assets, and QA records travel with this edition.

Contents

PARTS, CHAPTERS, AND SECTIONS

PART 1 · FOUNDATIONS 1

1. What Software Is and How Applications Work 2
1. Opening problem: The "magic box" misconception 3
2. Why this topic matters 3
3. Learning objectives 4
4. Prerequisites and what to skip 4
5. Plain-English explanation 5
6. Mental model: The restaurant counter (and where it breaks down) 5
7. Technical explanation with correct terminology 6
8. Concrete example: Loading the Mifugo Market homepage 8
9. Step-by-step demonstration: Watching a real request in your browser 9
10. AI collaboration example 10
11. The human's decisions and responsibilities (non-delegable) 10
12. How to inspect and verify results 11
13. Common failure modes and recovery 11
14. Professional standard 12
15. Hands-on exercise: Network detective (assessment level L2–L3) 13
16. Build challenge 13
17. Key takeaways 14
18. Glossary of new terms 14
19. Knowledge-check questions 16
20. Links to earlier and later chapters 17

2. Vibe Coding, Slop, and the Professional Builder 18
1. Opening problem: The "it runs, so it's fine" misconception 19
2. Why this topic matters 19
3. Learning objectives 20
4. Prerequisites and what to skip 21
5. Plain-English explanation 21
6. Mental model: The slop funnel 22
7. Technical explanation with correct terminology 22
8. Concrete example: A hypothetical AI-generated login form 23
9. Step-by-step demonstration: Spotting slop in three code snippets 25
10. AI collaboration example 27
11. The human's decisions and responsibilities (non-delegable) 27
12. How to inspect and verify AI output 28

13. Common failure modes and recovery 28
14. Professional standard 29
15. Hands-on exercise: Slop spotter (assessment level L2–L3) 30
16. Build challenge 32
17. Key takeaways 33
18. Glossary of new terms 33
19. Knowledge-check questions 35
20. Links to earlier and later chapters 36

PART 2 · PRODUCT BEFORE CODE 37

3. Finding a Problem Worth Solving 38
1. Opening problem — the misconception that wastes months of work 39
2. Why this topic matters in real software projects 40
3. Learning objectives 40
4. Prerequisites and what to skip 41
5. Plain-English explanation 41
6. Mental model — and where it breaks down 42
7. Technical explanation with correct terminology 42
8. Concrete example — validating Mifugo Market 44
9. Step-by-step demonstration — validating your own idea 45
10. AI collaboration example (prompt format P1–P10) 47
11. Human decisions and responsibilities (non-delegable judgment) 49
12. Inspect and verify AI output (problem-validation checklist) 51
13. Failure modes and recovery 51
14. Professional standard — quality threshold before coding 52
15. Hands-on exercise (L2 — Understanding) 52
16. Build challenge (L3 — Application) with acceptance criteria 53
17. Key takeaways 54
18. Glossary of new terms 55
19. Knowledge-check questions 57
20. Links to earlier and later chapters 58

4. Users, Journeys, and Requirements 60
1. Opening problem — designing for everyone means designing for no one 61
2. Why this topic matters in real software projects 62
3. Learning objectives 62
4. Prerequisites and what to skip 63
5. Plain-English explanation 63
6. Mental model — and where it breaks down 64
7. Technical explanation with correct terminology 64
8. Concrete example — Mifugo Market order journey 66
9. Step-by-step demonstration — creating your persona and journey 67

10. AI collaboration example (prompt format P1–P10) 68
11. Human decisions and responsibilities (non-delegable judgment) 70
12. Inspect and verify AI output (journey checklist) 72
13. Failure modes and recovery 72
14. Professional standard — quality threshold before design 73
15. Hands-on exercise (L3 — Application) 73
16. Build challenge (L3 — Application) with acceptance criteria 74
17. Key takeaways 76
18. Glossary of new terms 76
19. Knowledge-check questions 78
20. Links to earlier and later chapters 79

5. Specification, Acceptance Criteria, and Scope Control 80
1. Opening problem — a vague prompt produces slop 81
2. Why this topic matters in real software projects 82
3. Learning objectives 82
4. Prerequisites and what to skip 83
5. Plain-English explanation 83
6. Mental model — and where it breaks down 84
7. Technical explanation with correct terminology 84
8. Concrete example — Mifugo Market product catalog 85
9. Step-by-step demonstration — writing a spec and criteria 87
10. AI collaboration example (prompt format P1–P10) 88
11. Human decisions and responsibilities (non-delegable judgment) 90
12. Inspect and verify AI output (specification checklist) 92
13. Failure modes and recovery 92
14. Professional standard — quality threshold before building 93
15. Hands-on exercise (L3 — Application) 93
16. Build challenge (L3 — Application) with acceptance criteria 94
17. Key takeaways 95
18. Glossary of new terms 95
19. Knowledge-check questions 98
20. Links to earlier and later chapters 99

PART 3 · DIRECTING THE AI 100

6. Tools, Environments, and Your First Win 101
1. Opening problem — the trap of tool preparation 102
2. Why this topic matters in real software projects 103
3. Learning objectives 103
4. Prerequisites and what to skip 104
5. Plain-English explanation 104
6. Mental model — and where it breaks down 105

7. Technical explanation with correct terminology 106
8. Concrete example — Mifugo Market landing page 107
9. Step-by-step demonstration — your first win 108
10. AI collaboration example (prompt format P1–P10) 115
11. Human decisions and responsibilities (non-delegable judgment) 116
12. Inspect and verify AI output (first-win checklist) 117
13. Failure modes and recovery 118
14. Professional standard — quality threshold before going live 120
15. Hands-on exercise (L3 — Application) 120
16. Build challenge (L3 — Application) with acceptance criteria 121
17. Key takeaways 122
18. Glossary of new terms 123
19. Knowledge-check questions 126
20. Links to earlier and later chapters 127

7. Prompting, Context, and AI Collaboration 129
1. Opening problem — the one-line prompt trap 130
2. Why this topic matters in real software projects 131
3. Learning objectives 131
4. Prerequisites and what to skip 132
5. Plain-English explanation 132
6. Mental model — and where it breaks down 133
7. Technical explanation with correct terminology 134
8. Concrete example — adding a product catalog to Mifugo Market 135
9. Step-by-step demonstration — prompting for a catalog feature 135
10. AI collaboration example (prompt format P1–P10) 138
11. Human decisions and responsibilities (non-delegable judgment) 141
12. Inspect and verify AI output (prompting checklist) 142
13. Failure modes and recovery 142
14. Professional standard — quality threshold before accepting AI output 143
15. Hands-on exercise (L3 — Application) 144
16. Build challenge (L3 — Application) with acceptance criteria 145
17. Key takeaways 146
18. Glossary of new terms 147
19. Knowledge-check questions 148
20. Links to earlier and later chapters 150

8. The Diff Discipline: Reviewing AI-Generated Code 151
1. Opening problem — the "Accept All" habit 152
2. Why this topic matters in real software projects 153
3. Learning objectives 153
4. Prerequisites and what to skip 154
5. Plain-English explanation 154
6. Mental model — and where it breaks down 155

7. Technical explanation with correct terminology 156
8. Concrete example — reviewing an "add to cart" diff 157
9. Step-by-step demonstration — reviewing an AI diff 158
10. AI collaboration example (prompt format P1–P10) 160
11. Human decisions and responsibilities (non-delegable judgment) 161
12. Inspect and verify AI output (diff-review checklist) 162
13. Failure modes and recovery 163
14. Professional standard — quality threshold before accepting AI output 165
15. Hands-on exercise (L4 — Analysis) 165
16. Build challenge (L4 — Analysis) with acceptance criteria 167
17. Key takeaways 168
18. Glossary of new terms 169
19. Knowledge-check questions 171
20. Links to earlier and later chapters 172

PART 4 · APPLICATION STRUCTURE 174

9. Frontend: Interface, State, and User Experience 175
1. Opening problem — "frontend is just making it look pretty" 176
2. Why this topic matters in real software projects 177
3. Learning objectives 178
4. Prerequisites and what to skip 178
5. Plain-English explanation 179
6. Mental model — and where it breaks down 180
7. Technical explanation with correct terminology 181
8. Concrete example — Mifugo Market order inquiry form 182
9. Step-by-step demonstration — building the order inquiry form 183
10. AI collaboration example (prompt format P1–P10) 189
11. Human decisions and responsibilities (non-delegable judgment) 190
12. Inspect and verify AI output (frontend checklist) 192
13. Failure modes and recovery 192
14. Professional standard — quality threshold before going live 193
15. Hands-on exercise (L3 — Application) 194
16. Build challenge (L3 — Application) with acceptance criteria 196
17. Key takeaways 197
18. Glossary of new terms 197
19. Knowledge-check questions 199
20. Links to earlier and later chapters 200

10. Backend, Data, and APIs 202
1. Opening problem — "the backend is where the AI does everything" 203
2. Why this topic matters in real software projects 204
3. Learning objectives 204

4. Prerequisites and what to skip 205
5. Plain-English explanation 205
6. Mental model — and where it breaks down 206
7. Technical explanation with correct terminology 207
8. Concrete example — Mifugo Market product catalog API 208
9. Step-by-step demonstration — product catalog API 209
10. AI collaboration example (prompt format P1–P10) 213
11. Human decisions and responsibilities (non-delegable judgment) 218
12. Inspect and verify AI output (backend/API checklist) 218
13. Failure modes and recovery 219
14. Professional standard — quality threshold before connecting frontend and backend 219
15. Hands-on exercise (L3 — Application) 220
16. Build challenge (L4 — Analysis) with acceptance criteria 221
17. Key takeaways 222
18. Glossary of new terms 223
19. Knowledge-check questions 225
20. Links to earlier and later chapters 226

11. Access Control, Authentication, and Secure Connections 227
1. Opening problem — "logged in means they can see everything" 228
2. Why this topic matters in real software projects 229
3. Learning objectives 229
4. Prerequisites and what to skip 230
5. Plain-English explanation 231
6. Mental model — and where it breaks down 232
7. Technical explanation with correct terminology 232
8. Concrete example — Mifugo Market login and order privacy 234
9. Step-by-step demonstration — adding authentication and authorization 236
10. AI collaboration example (prompt format P1–P10) 238
11. Human decisions and responsibilities (non-delegable judgment) 241
12. Inspect and verify AI output (authentication checklist) 241
13. Failure modes and recovery 242
14. Professional standard — quality threshold before enabling login 244
15. Hands-on exercise (L3 — Application) 245
16. Build challenge (L4 — Analysis) with acceptance criteria 246
17. Key takeaways 247
18. Glossary of new terms 247
19. Knowledge-check questions 250
20. Links to earlier and later chapters 251

PART 5 · BUILD, TEST, PROTECT 253

12. Testing, Debugging, and Regression Prevention 254

1. Opening problem — "if it works once, it keeps working" 255
2. Why this topic matters in real software projects 256
3. Learning objectives 256
4. Prerequisites and what to skip 257
5. Plain-English explanation 257
6. Mental model — and where it breaks down 258
7. Technical explanation with correct terminology 259
8. Concrete example — Mifugo Market price calculation 260
9. Step-by-step demonstration — writing and running tests 260
10. AI collaboration example (prompt format P1–P10) 263
11. Human decisions and responsibilities (non-delegable judgment) 266
12. Inspect and verify AI output (testing checklist) 267
13. Failure modes and recovery 267
14. Professional standard — quality threshold before deploying 268
15. Hands-on exercise (L3 — Application) 268
16. Build challenge (L4 — Analysis) with acceptance criteria 269
17. Key takeaways 270
18. Glossary of new terms 270
19. Knowledge-check questions 272
20. Links to earlier and later chapters 273

13. Security, Privacy, and Data Protection 275

1. Opening problem — "HTTPS makes my app secure" 276
2. Why this topic matters in real software projects 277
3. Learning objectives 277
4. Prerequisites and what to skip 278
5. Plain-English explanation 278
6. Mental model — and where it breaks down 280
7. Technical explanation with correct terminology 280
8. Concrete example — Mifugo Market security hardening 282
9. Step-by-step demonstration — securing Mifugo Market 283
10. AI collaboration example (prompt format P1–P10) 285
11. Human decisions and responsibilities (non-delegable judgment) 288
12. Inspect and verify AI output (security checklist) 289
13. Failure modes and recovery 289
14. Professional standard — quality threshold before going live with user data 291
15. Hands-on exercise (L3 — Application) 291
16. Build challenge (L4 — Analysis) with acceptance criteria 293
17. Key takeaways 294
18. Glossary of new terms 294
19. Knowledge-check questions 297
20. Links to earlier and later chapters 298

PART 6 · SHIP AND OPERATE 299

14. Version Control, Deployment, and Operations 300
1. Opening problem — the misconceptions that cost real businesses 301
2. Why this topic matters in real software projects 302
3. Learning objectives 303
4. Prerequisites and what to skip 304
5. Plain-English explanation 306
6. Mental model — and where it breaks down 306
7. Technical explanation with correct terminology 307
8. Concrete example — Mifugo Market goes live 309
9. Step-by-step demonstration 312
10. AI collaboration example (prompt format P1–P10) 318
11. Human decisions and responsibilities (non-delegable judgment) 321
12. Inspect and verify AI output (deployment-related checklist) 323
13. Failure modes and recovery 323
14. Professional standard — quality threshold before going live 325
15. Hands-on exercise (L3 — Application) 327
16. Build challenge (L4 — Analysis) with acceptance criteria 328
17. Key takeaways 329
18. Glossary of new terms 329
19. Knowledge-check questions 333
20. Links to earlier and later chapters 334

PART 7 · PRODUCT & RESPONSIBILITY 336

15. From Prototype to Product: Validation, Viability, and Trust 337
1. Opening problem — "shipping means I'm done" 338
2. Why this topic matters in real software projects 339
3. Learning objectives 339
4. Prerequisites and what to skip 340
5. Plain-English explanation 341
6. Mental model — and where it breaks down 343
7. Technical explanation with correct terminology 343
8. Concrete example — Mifugo Market maturity assessment 345
9. Step-by-step demonstration — ship-readiness for your application 346
10. AI collaboration example (prompt format P1–P10) 349
11. Human decisions and responsibilities (non-delegable judgment) 352
12. Inspect and verify AI output (ship-readiness checklist) 352
13. Failure modes and recovery 353
14. Professional standard — quality threshold before launch 354
15. Hands-on exercise (L3 — Application) 354

16. Build challenge (L4 — Analysis) with acceptance criteria 356
17. Key takeaways 357
18. Glossary of new terms 357
19. Knowledge-check questions 359
20. Links to earlier and later chapters 361

PART 8 · CAPSTONE 363

16. Capstone Project — Ship Mifugo Market End-to-End 364
1. Opening problem — "I've learned the pieces but can't put them together" 365
2. Why this topic matters in real software projects 366
3. Learning objectives 366
4. Prerequisites and what to skip 368
5. Plain-English explanation 368
6. Mental model — and where it breaks down 369
7. Technical explanation with correct terminology 369
8. Concrete example — Mifugo Market capstone roadmap 371
9. Step-by-step demonstration — the four-week capstone 372
10. AI collaboration example (prompt format P1–P10) 375
11. Human decisions and responsibilities (non-delegable judgment) 378
12. Inspect and verify AI output (capstone checklist) 378
13. Failure modes and recovery 379
14. Professional standard — quality threshold for the capstone 379
15. Hands-on exercise (L6 — Creation) 381
16. Build challenge / capstone rubric (L6 — Creation) 381
17. Key takeaways 383
18. Glossary of new terms 384
19. Knowledge-check questions 385
20. Links to earlier and later chapters 386

17. Your Independent Practice 387
1. Opening problem — "the book ends here" 388
2. Why this topic matters in real software projects 389
3. Learning objectives 389
4. Prerequisites and what to skip 390
5. Plain-English explanation 391
6. Mental model — and where it breaks down 392
7. Technical explanation with correct terminology 392
8. Concrete example — Wanjiku's next project 394
9. Step-by-step demonstration — troubleshooting a failed deploy 394
10. AI collaboration example (prompt format P1–P10) 401
11. Human decisions and responsibilities (non-delegable judgment) 402
12. Inspect and verify AI output (ongoing practice checklist) 404

13. Failure modes and recovery 404
14. Professional standard — quality threshold for independent practice 405
15. Hands-on exercise (L3 — Application) 406
16. Build challenge (L4 — Analysis) with acceptance criteria 407
17. Key takeaways 409
18. Glossary of new terms 409
19. Knowledge-check questions 411
20. Links to earlier and later chapters 412

Appendix A · Glossary of Core Terms 414

Appendix B · Evidence Register 416

Appendix C · Standards and Security References 418

About the Author 420

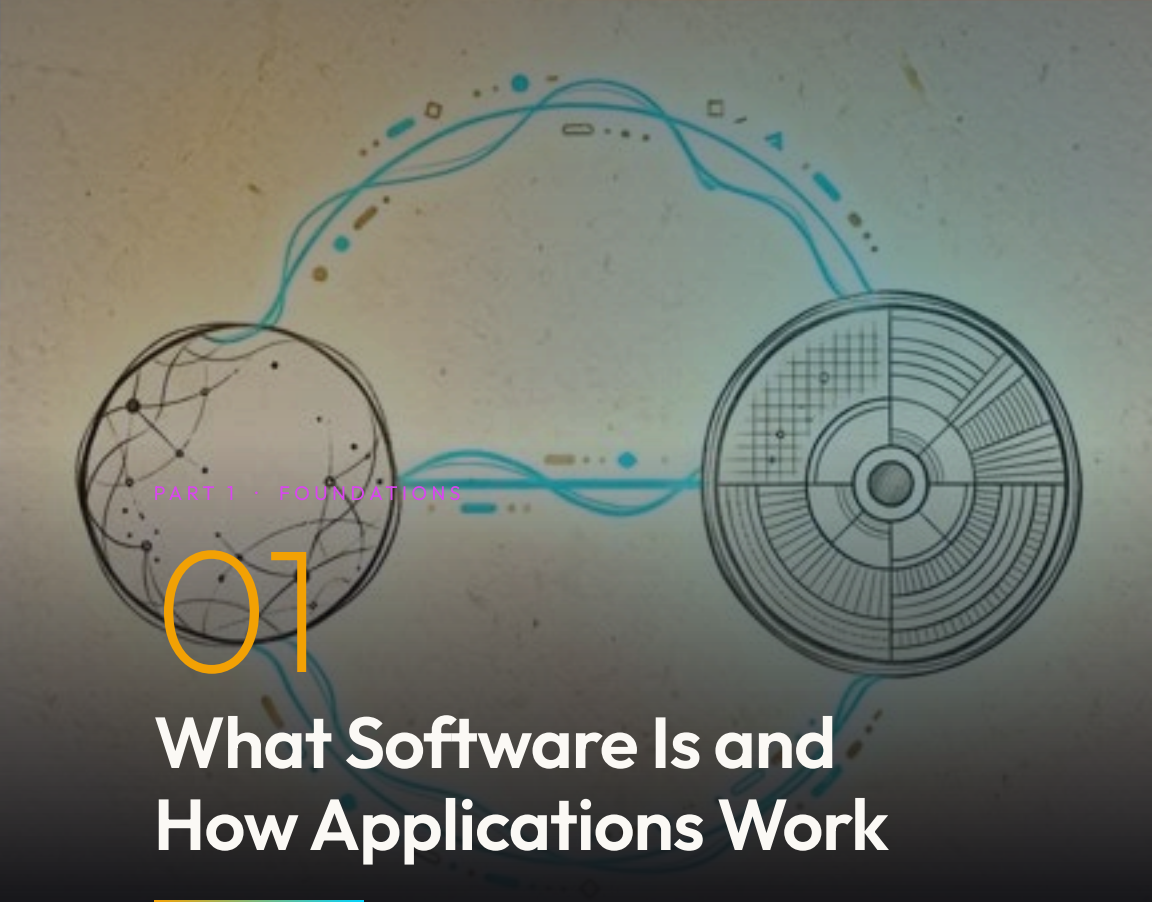
PART 1

1 Foundations

Software, AI, and You

01 What Software Is and How Applications Work

02 Vibe Coding, Slop, and the Professional Builder



PART 1 · FOUNDATIONS

01

What Software Is and How Applications Work

Part 1 — Foundations: Software, AI, and You · Reading and practice time: approximately 90 minutes (60 minutes reading + 30 minutes practice) · **Prerequisites:** None. You should be able to use a web browser, manage files, and follow written instructions.

01 Opening problem: The "magic box" misconception

SLOP DETECTOR — THE MISCONCEPTION

Misconception: *Software is magic — you type something, press a button, and it just works. If it breaks, you need a wizard to fix it.*

Many people who have never written code think of software as a black box: a mysterious thing that lives inside computers and phones, doing unknowable things. This misconception is dangerous when you start using AI to build software. If you think software is magic, you cannot tell whether the AI built something sound or something that will collapse under real use. You will not know where to look when it breaks, and you will not understand why a "simple change" sometimes takes days.

This chapter replaces the black-box model with a concrete mental model: the client-server cycle. By the end, you will be able to trace a single web page from your browser to a server and back — and you will know where to look when something goes wrong.

02 Why this topic matters

Before you can direct an AI to build an application, you need to understand what an application *is*. Without this model, you will not understand:

- Why your site works on your laptop but not on your friend's phone.
- Why a form submission can fail silently.
- Why you need a domain, a host, and environment variables before the world can use your app.
- What a developer means when they say "the frontend talks to the backend."

02 Why this topic matters (continued)

Software that runs on one machine is a demonstration. Software that reaches real users over the internet is an application. This chapter teaches you the difference.

03 Learning objectives

WHAT YOU WILL BE ABLE TO DO

By the end of this chapter, you can:

- Draw a labeled diagram showing the path from a browser to a server and back (browser → DNS → server → response → browser).
- Explain, in your own words, what happens between typing a URL and seeing a page.
- Distinguish the frontend (what runs in the browser) from the backend (what runs on the server).
- Define an HTTP request and an HTTP response, and give one example of each.
- Explain what "stateless" means in one sentence, and why it matters for logins.

04 Prerequisites and what to skip

BEFORE YOU START

Prerequisites: None. This is the first chapter.

What to skip: You do not need to know any programming language. You do not need to install anything beyond a web browser (which you already have). We will not write code in this chapter; we will observe how existing software behaves.

05 Plain-English explanation

Imagine you are standing at a restaurant counter. You place an order (a request). The kitchen prepares your meal (the server processes the request). The waiter brings the food back to your counter (a response). You eat it. If you want another meal, you place another order. The kitchen does not remember your previous order unless you tell it again or it writes it down.

A web application works the same way:

- **You (the client)** use a browser (Chrome, Firefox, Safari) to ask for something.
- **The server** is a computer somewhere else that stores the application and data.
- **The request** is your ask: "Show me the homepage," "Submit this order," "Delete this item."
- **The response** is the server's answer: here is the page, here is an error, here is the data you asked for.

This cycle happens every time you click a link, submit a form, or load an image on the web.

06 Mental model: The restaurant counter (and where it breaks down)

Analogy: The browser is the customer at the counter. The server is the kitchen. HTTP is the language they use to place orders and receive meals.

06 Mental model: The restaurant counter (and where it breaks down) (continued)

Where the analogy breaks down: A real waiter remembers that you are at table 4 and that you already ordered water. HTTP, by itself, remembers nothing. Each request is independent — the server does not know who you are from one request to the next unless something (a cookie, a session, a token) adds that memory. We will cover how applications add memory in later chapters; for now, understand that HTTP itself is **stateless**.

07 Technical explanation with correct terminology

Each term below is defined at first use and appears in the chapter glossary at the end.

Client The device or software that makes a request. In web applications, the client is usually a browser running on your phone or computer. A-01

Server A computer (or program running on one) that stores an application and answers client requests. A-01

Browser A specific kind of client: an application such as Chrome, Firefox, or Safari that renders web pages for people to read and interact with.

HTTP (Hypertext Transfer Protocol) The language clients and servers use to talk to each other. An HTTP conversation always has two parts: a request from the client, and a response from the server. A-01 · A-02

Request A message from the client to the server asking for something. It includes a method (what to do), a path (what resource), and sometimes data. A-03

Response The server's answer to a request. It includes a status code (did it work?) and usually a body (the page, data, or error message). A-03

07 Technical explanation with correct terminology

(continued)

Status code A three-digit number in every HTTP response telling the client what happened. Common examples: `200 OK` (success), `404 Not Found` (the resource does not exist), `500 Internal Server Error` (the server crashed). A-03

DNS (Domain Name System) The internet's address book. When you type `mifugo.co.ke`, DNS translates that name into an IP address — the actual location of the server. A-02

07 Technical explanation with correct terminology

(continued)

IP address A numeric address that identifies a specific computer on the internet, like `192.0.2.1`. Humans use domain names; machines use IP addresses.

Static content Files (HTML, CSS, images) that the server sends exactly as stored, without changing them per request.

Dynamic content Content the server generates on the fly, often by reading from a database or calculating a result, so different users can see different things at the same URL.

Frontend The part of an application that runs in the browser: HTML, CSS, and JavaScript. It displays pages and collects user input.

Backend The part of an application that runs on the server: it receives requests, processes data, talks to databases, and sends responses.

Runtime environment The software that executes code. Browsers are runtime environments for frontend code; Node.js is a common runtime environment for backend code.

Stateless Describing a protocol that does not remember previous requests. HTTP is stateless: each request stands alone. A-01

08 Concrete example: Loading the Mifugo Market homepage

IN PRACTICE

Mifugo Market is our running example throughout this book: a Nairobi-based fresh-produce ordering platform for small shops and households. Its owner, Wanjiku, wants customers to browse produce (sukuma wiki, tomatoes, avocados, mangoes) and place orders.

When a customer types `mifugo.co.ke` into a browser and presses Enter, here is what happens:

1. **DNS lookup:** The browser asks a DNS server: "What is the IP address for mifugo.co.ke?" The DNS server replies with an IP address.
2. **Connection:** The browser opens a connection to that IP address.
3. **Request:** The browser sends an HTTP `GET` request asking for the homepage.
4. **Processing:** The server receives the request, finds the homepage files, and assembles a response.
5. **Response:** The server sends back `200 OK` plus HTML, CSS, and JavaScript files.
6. **Rendering:** The browser receives the files, assembles them, and displays the Mifugo Market homepage.

Each image, stylesheet, and script on that page is usually a separate request/response cycle. The page you see is not one file; it is many conversations stitched together.

09 Step-by-step demonstration: Watching a real request in your browser

STEP BY STEP

You do not need to write code to see this happen. Your browser has built-in tools that show every request and response.

Environment: Any computer with Chrome, Firefox, Safari, or Edge. No installation required.

Procedure:

1. Open your browser and go to a public Kenyan business website (for example, a local supermarket, bank, or news site).
2. Right-click anywhere on the page and choose **Inspect** (or press `F12` / `Cmd+Option+I`).
3. Click the **Network** tab in the panel that opens.
4. Refresh the page (`F5` or `Cmd+R`).
5. You will see a list of every file the browser requested: the HTML document, CSS stylesheets, images, JavaScript files, and possibly API calls.
6. Click the first item (usually the HTML document). A detail panel opens.
7. Look for **Request Method** (probably `GET`), **Status Code** (probably `200`), and **Response Headers**.

Expected result: You will see a list of requests, each with a status code. The main page should show `200 OK`. Images and stylesheets should also show `200 OK` if they loaded successfully. A missing image might show `404 Not Found`.

Limitations: Some sites block inspection or load content after the page opens (you may need to scroll or click to trigger more requests). The Network tab clears when you navigate away unless you check "Preserve log."

09 Step-by-step demonstration: Watching a real request in your browser (continued)

What you are seeing: Every row in that Network tab is one HTTP request and one HTTP response — the same cycle described above, happening dozens of times per page load.

10 AI collaboration example

AI COLLABORATION — PROMPT FORMAT P1–P10

This chapter does not involve generating code with AI. It is observational only. AI collaboration begins in Chapter 6, when we scaffold the first static page for Mifugo Market. Until then, the skill to practice is *observation*: understanding what already exists before you ask an AI to change it.

11 The human's decisions and responsibilities (non-delegable)

YOUR DECISION — NOT THE AI'S

Even though this chapter involves no AI, it establishes a principle that runs through the whole book:

- **You decide what the software is supposed to do.** The AI can suggest, but you define the problem.
- **You decide where data lives.** Is it stored in the browser, on a server, or in a database? That choice affects security, cost, and reliability.
- **You decide what "works" means.** A page that loads is not the same as a page that serves real users safely.

No AI tool can make these decisions for you. They are yours.

12 How to inspect and verify results

QUALITY CHECK

Since this chapter is observational, verification means confirming what you saw in the Network tab:

- ✓ The main page request returned `200 OK` — or you can explain why it returned something else.
- ✓ You can identify at least three different resource types (HTML, CSS, image, JavaScript).
- ✓ You can explain that each resource required its own request/response cycle.

If you cannot see the Network tab, try a different browser or check whether your browser's developer tools are disabled by your organization.

13 Common failure modes and recovery

WHEN IT GOES WRONG

13 Common failure modes and recovery (continued)

FAILURE MODE	WHAT IT LOOKS LIKE	WHY IT HAPPENS	RECOVERY
Confusing browser with server	"The website is broken because my computer is slow"	The problem may be on the server, not your device	Test the same site on another device or network
Assuming HTTP remembers you	Logging in on one page, then being asked to log in again on the next	HTTP is stateless; memory requires cookies/sessions/tokens	Look for where the application stores login state (covered in later chapters)
Missing resources	Images or styles fail to load; page looks broken	The server returned `404` for those files	Check the Network tab for failed requests and their URLs

13 Common failure modes and recovery (continued)

FAILURE MODE	WHAT IT LOOKS LIKE	WHY IT HAPPENS	RECOVERY
Server error	`500 Internal Server Error`	Something went wrong on the server, not in your browser	Refresh once; if it persists, the problem is on the server side

14 Professional standard

PRO STANDARD

Quality threshold: Before you build or modify any application, you should be able to sketch how a request travels from a user's browser to your server and back, and name at least three common status codes. If you cannot draw this cycle, you are not ready to debug deployment issues or evaluate AI-generated backend code. A-01 · A-02 · A-03

15 Hands-on exercise: Network detective (assessment level L2–L3)

BUILD IT

Task: Visit three different websites. For each, open the browser's Network tab, reload the page, and record:

1. The name of the site.
2. The status code of the main page request (first item in the list).
3. Three different resource types that loaded (e.g., HTML document, image, stylesheet, script).
4. One request that returned something other than `200 OK` (if any), and its status code.

Constraints: Use real websites, not examples from this book. Do not use a site that requires you to log in.

Expected deliverable: A written list (notes or a document) with one entry per site, containing the four items above.

Success criteria: You can identify the status code and resource type for at least three requests across three different sites, and you can explain what `200 OK` and `404 Not Found` mean in your own words.

16 Build challenge

● PUSH FURTHER

Not applicable to this chapter. The first build challenge appears in Chapter 6, when you deploy a static page for Mifugo Market.

17 Key takeaways

KEY TAKEAWAYS

- Software is not magic. It is a chain of requests and responses between clients and servers.
- The browser (client) asks; the server answers. HTTP is the language they use.
- Every page load involves many separate requests — for HTML, images, stylesheets, scripts.
- HTTP is stateless: the server does not remember who you are between requests unless the application adds memory.
- Status codes tell you what happened: `200` means success, `404` means not found, `500` means server error.

18 Glossary of new terms

TERMS INTRODUCED IN THIS CHAPTER

18 Glossary of new terms (continued)

TERM	DEFINITION	EXAMPLE
Client	The device or software that makes a request (usually a browser).	Your laptop running Chrome.
Server	A computer or program that stores an application and answers client requests.	The computer hosting mifugo.co.ke.
Browser	An application that renders web pages for people.	Chrome, Firefox, Safari.
HTTP	The protocol clients and servers use to exchange requests and responses.	The "language" of the web.
Request	A message from client to server asking for a resource or action.	`GET /homepage`

18 Glossary of new terms (continued) (continued)

TERM	DEFINITION	EXAMPLE
Response	The server's answer to a request, including a status code and body.	`200 OK` + HTML page
Status code	A three-digit number indicating the result of a request.	`200 OK`, `404 Not Found`, `500 Internal Server Error`
DNS	The system that translates domain names into IP addresses.	`mifugo.co.ke` → `192.0.2.1`
IP address	A numeric address identifying a computer on the internet.	`192.0.2.1`
Static content	Files sent exactly as stored, unchanged per request.	An image file

18 Glossary of new terms (continued) (continued)

TERM	DEFINITION	EXAMPLE
Dynamic content	Content generated by the server for a specific request.	A personalized dashboard
Frontend	Code that runs in the browser (HTML, CSS, JavaScript).	The Mifugo Market product listing page
Backend	Code that runs on the server (receives requests, processes data).	The route that saves an order to a database
Runtime environment	Software that executes code.	Browser (frontend), Node.js (backend)

18 Glossary of new terms (continued)

TERM	DEFINITION	EXAMPLE
Stateless	A protocol that does not remember previous requests.	HTTP

19 Knowledge-check questions

KNOWLEDGE CHECK

Attempt these before reading the answers below.

1. What does HTTP stand for, and what does it do?
2. What is the difference between a client and a server?
3. What does a `404 Not Found` status code mean?
4. Why is HTTP described as "stateless"?
5. Name two types of resources (besides the HTML page itself) that a browser requests when loading a page.

ANSWERS AND EXPLANATIONS

(do not read until you have attempted the questions above)

1. **HTTP** stands for Hypertext Transfer Protocol. It is the language browsers and servers use to exchange requests and responses. A-01
2. A **client** makes requests (usually a browser). A **server** receives requests and sends responses (a computer hosting the application). A-01
3. `404 Not Found` means the server could not find the resource you asked for — for example, a page or image that does not exist at that URL. A-03
4. **Stateless** means the server does not remember previous requests. Each request stands alone unless the application adds memory (cookies, sessions, tokens). A-01
5. Examples: CSS stylesheets, images, JavaScript files, API data. Each is requested separately. A-02

20 Links to earlier and later chapters

WHERE THIS SITS IN THE BOOK

- **Earlier:** None (this is the first chapter).
- **Later:** Chapter 2 builds on this model to explain what AI-generated code is and is not. Chapters 9–10 show how frontend and backend exchange data over HTTP. Chapter 14 shows how to deploy an application so real users can reach it through this same request/response cycle.

PART 1 · FOUNDATIONS

02

Vibe Coding, Slop, and the Professional Builder

Part 1 — Foundations: Software, AI, and You · **Reading and practice time:** approximately 110 minutes (70 minutes reading + 40 minutes practice) · **Prerequisites:** Chapter 1 (you understand the client-server request/response cycle).

01 Opening problem: The "it runs, so it's fine" misconception

SLOP DETECTOR — THE MISCONCEPTION

Misconception: *If an AI wrote the code and the application runs on my screen, it is professional software ready for users.*

This is the most dangerous assumption a new AI-assisted builder can make. AI tools can generate a login form, a product page, or an order handler in seconds. The result may look polished and behave correctly in the simple cases you try. But "runs on my screen" is not the same as "safe for real users." Behind a working-looking surface, AI-generated code can contain hardcoded passwords, missing input validation, broken access controls, or silent data loss. None of these show up as an error message until a real user — or an attacker — finds them.

This chapter exists to establish one principle before you write or generate a single line of production code: **producing code is not the same as engineering a product.** The human remains responsible for understanding the problem, evaluating the AI's output, protecting users, and deciding whether software is ready to ship.

02 Why this topic matters

AI has dramatically lowered the barrier to producing code. You can describe an idea in plain language and have a working-looking application in minutes. But producing code is not engineering. A bridge that looks solid from a distance can still collapse under load. Software that looks functional can still:

02 Why this topic matters (continued)

- Leak customer data because passwords were stored in plaintext.
- Let one customer view another customer's orders by changing a number in the web address.
- Fail silently when a payment is submitted, taking money without recording the order.
- Be impossible to fix when the AI stumbles, because you never understood how it works.

02 Why this topic matters (continued)

If you ship software like this to real users, you are not building a product. You are creating **slop** — low-quality, unreviewed AI-generated code that looks plausible but is unsafe, untested, or unmaintainable.

03 Learning objectives

WHAT YOU WILL BE ABLE TO DO

By the end of this chapter, you can:

- Define **vibe coding** and state its origin (Andrej Karpathy, February 2025) and original scope (throwaway weekend projects).
- Define **slop** and give three examples of what it looks like in practice.
- List five things AI-generated code may get wrong (logic, security, data handling, missing context, pattern violations).
- State three non-delegable human responsibilities in AI-assisted software building.
- Decide, with reasoning, whether a given scenario is "vibe coding" or "professional AI-assisted building."

04 Prerequisites and what to skip

BEFORE YOU START

Prerequisites: Chapter 1 (client-server model, HTTP request/response). You need the mental model from Chapter 1 to understand where AI-generated code fits in an application.

What to skip: You do not need to install any AI coding tool yet. You do not need to write code. This chapter is observational and analytical: you will read code, classify it, and make decisions about it.

05 Plain-English explanation

Vibe coding is a style of building software where you describe what you want in everyday language, an AI writes the code, and you accept the result based on how it feels ("the vibe") rather than on careful review. The term was coined by Andrej Karpathy, an AI researcher and co-founder of OpenAI, in February 2025. He described it as "fully giving in to the vibes" and "forgetting that the code even exists." Importantly, Karpathy himself scoped vibe coding to **throwaway weekend projects** — experiments, not products for real users.

Slop is the name for the output when vibe coding is applied carelessly: code that runs but is insecure, untested, undocumented, or impossible to maintain. Slop looks like software. It behaves like software in the happy path. It is not professional software.

Professional AI-assisted building is different. You still use AI to generate code, but you treat every AI suggestion as a draft that must be reviewed, tested, and verified before it reaches users. You remain the director, the editor, and the person accountable for what ships.

06 Mental model: The slop funnel

Analogy: Think of AI-generated code like fast food prepared without an inspector in the kitchen. It looks like a meal. It tastes fine at first. But without inspection, you cannot tell whether the ingredients were stored safely, whether the cook washed their hands, or whether something will make you sick later.

Where the analogy breaks down: Unlike food poisoning, software slop does not always announce itself immediately. A security flaw can sit dormant for months before an attacker finds it. The analogy helps you understand the need for inspection, but it does not capture the silent, delayed nature of many software failures.

07 Technical explanation with correct terminology

Each term below is defined at first use and appears in the chapter glossary.

Vibe coding A style of AI-assisted development where a person describes a goal in natural language, an AI generates code, and the person accepts or iterates based on results rather than systematic review. Origin: Andrej Karpathy, February 2025; originally described for throwaway weekend projects. D-01

Slop Low-quality AI-generated code that runs but is insecure, untested, undocumented, or difficult to maintain. Slop is the risk this book exists to prevent.

AI-generated code Source code produced by an artificial intelligence model rather than written by a person. It may be syntactically correct (it runs) but logically or semantically wrong.

Human-in-the-loop The principle that a person must review, decide, and take responsibility for AI output rather than accepting it automatically.

07 Technical explanation with correct terminology

(continued)

Non-delegable judgment Decisions that the human builder must make personally: what problem to solve, what "safe enough" means, whether software is ready for real users. These cannot be assigned to an AI.

Throwaway project Software built for learning or experimentation, not intended for real users or long-term maintenance. Vibe coding's original scope.

Production system Software that real users depend on, with security, testing, backups, monitoring, and a plan for when things break.

08 Concrete example: A hypothetical AI-generated login form

IN PRACTICE

Below is a **hypothetical** example — labeled as such, not a real product — of what an AI might generate if asked to "make a login page." It runs. It looks reasonable. It is slop.

```HTML
<!— [Illustrative — not production-ready. Hypothetical example only.] —>
<form action="/login" method="POST">
  <input type="email" name="email" placeholder="Email" required>
  <input type="password" name="password" placeholder="Password" required>
  <button type="submit">Sign in</button>
</form>
```

And the corresponding server-side logic (pseudocode):

08 Concrete example: A hypothetical AI-generated login form (continued)

```text
# [Pseudocode — illustrative only, not executable as-is]
password = get_from_form("password")
if password == "admin123":
    let_user_in()
else:
    show_error("Wrong password")
```

What is wrong with this example:

08 Concrete example: A hypothetical AI-generated login form (continued)

1. **Hardcoded secret:** The password `admin123` is written directly into the code. Anyone who sees the code knows the password. It cannot be changed without editing and redeploying the application.
2. **No password hashing:** Passwords are compared in plaintext. If the code or database is exposed, every user's password is visible.
3. **No input validation:** Nothing checks whether the email is valid, whether the password meets a minimum length, or whether the input contains malicious commands.
4. **No rate limiting:** Someone can try thousands of passwords in rapid succession (a brute-force attack).
5. **No error handling that protects users:** Error messages might reveal whether an email address exists, helping attackers enumerate accounts.

08 Concrete example: A hypothetical AI-generated login form (continued) (continued)

This code "runs." If you opened it in a browser, you could type an email and password and get a response. But it is not safe for real users.

09 Step-by-step demonstration: Spotting slop in three code snippets

STEP BY STEP

This chapter's demonstration is analytical, not constructive. You will review three short code snippets and classify each.

Environment: Any text editor or paper. No installation required.

Snippet A — Clean:

```HTML
<label for="email">Email address</label>
<input type="email" id="email" name="email" required>
```

Classification: Acceptable. It has a label, a proper input type, and a required attribute. It is a small, reviewable piece of frontend code.

Snippet B — Flawed (AI-generated-looking):

```TEXT
# [Pseudocode — illustrative only]
api_key = "sk_live_abc123def456"
send_request(url, api_key)
```

09 Step-by-step demonstration: Spotting slop in three code snippets (continued)

Classification: Reject. A hardcoded API key. Anyone with access to the code can impersonate the application. Keys must come from environment variables, never be written into source code. E-02 · G-05

Snippet C — Flawed:

09 Step-by-step demonstration: Spotting slop in three code snippets (continued)

```
TEXT
# [Pseudocode — illustrative only]
if user_id == requested_id:
    show_data()
else:
    show_data_anyway()  # bug: should deny access
```

Classification: Reject. Broken access control. A user can view another user's data by guessing or changing an ID in the web address. This is the most common serious web application security risk. G-02

Expected result: You can identify at least one flaw in each flawed snippet and explain why Snippet A is acceptable while Snippets B and C must not be accepted as-is.

10 AI collaboration example

AI COLLABORATION — PROMPT FORMAT P1–P10

Chapter 2 does not involve prompting an AI to generate code. The skill here is *reviewing* AI output, not producing it. AI collaboration examples begin in Chapter 7, where you learn to write structured prompts and maintain context files. The review mindset you build here applies to every AI-generated snippet you encounter from Chapter 6 onward.

11 The human's decisions and responsibilities (non-delegable)

YOUR DECISION — NOT THE AI'S

No matter how capable the AI becomes, these decisions remain yours:

1. **What problem are we solving, and for whom?** The AI can suggest features, but you define the purpose and the users.
2. **Is this output safe to show real users?** You must check for exposed secrets, missing validation, and broken access controls before any user touches the software.
3. **Do I understand this code well enough to fix it if it breaks?** If you cannot explain what a piece of code does, you should not deploy it.

An AI can draft code, suggest fixes, and explain errors. It cannot accept responsibility for a data breach, a failed payment, or a leaked customer list. That responsibility stays with you.

12 How to inspect and verify AI output

QUALITY CHECK

When an AI generates code, do not "Accept All." Instead:

1. ✓ **Read the diff.** Review every added and changed line. If you cannot explain what a line does, ask the AI to explain it — then verify the explanation yourself.
2. ✓ **Look for secrets.** Search for anything that looks like a key, password, or token (`sk_`, `secret`, `password`, `API_KEY`). If it is hardcoded, reject it. G-05
3. ✓ **Look for validation.** Does the code check that inputs are valid before using them? Unvalidated input is an injection risk. G-03
4. ✓ **Look for access control.** Does the code check that the current user may perform this action on this specific resource? G-02
5. ✓ **Run the tests.** No AI-generated code ships without a test suite passing. D-03

13 Common failure modes and recovery

WHEN IT GOES WRONG

13 Common failure modes and recovery (continued)

FAILURE MODE	WHAT IT LOOKS LIKE	WHY IT HAPPENS	RECOVERY
Assuming running code is correct	The app works in your browser, so you ship it	AI optimizes for "it runs," not "it is secure"	Review code against a checklist before deploying; never ship unreviewed AI output
Treating vibe coding as production-ready	You build a customer-facing app with no tests, no security review, no backups	Vibe coding was scoped to throwaway projects; it is not a production method	Use vibe coding for prototypes only; apply full review, testing, and security before shipping

13 Common failure modes and recovery (continued)

FAILURE MODE	WHAT IT LOOKS LIKE	WHY IT HAPPENS	RECOVERY
Not knowing what to look for in a diff	You accept AI changes because you do not understand them	You lack the mental model to spot flaws	Study Chapter 1 (request/response) and Chapter 8 (diff review); ask for explanations until you understand
Copy-pasting without context	The AI generates code that contradicts your existing files	The AI does not see your whole codebase	Maintain a context file (Chapter 7); review changes against existing patterns

14 Professional standard

● PRO STANDARD

Quality threshold: Before any AI-generated code reaches users, you must be able to answer "yes" to each of these:

14 Professional standard (continued)

- I have read every line of the generated code (or every diff).
- No secrets (passwords, API keys, tokens) are hardcoded.
- All user input is validated on the server.
- Users can only access their own data (object-level authorization).
- A test suite runs and passes.

14 Professional standard (continued)

- I can explain how the code works well enough to fix it if it breaks.

14 Professional standard (continued)

If any answer is "no," the code is slop, not software. Do not ship it.

D-03 · D-04 · G-02 · G-03 · G-05

15 Hands-on exercise: Slop spotter (assessment level L2–L3)

BUILD IT

Task: You are given three short code snippets (below). Classify each as **Acceptable**, **Flawed — revise**, or **Reject**. For each flawed snippet, list the specific problem and one way to fix it.

Snippet 1:

```HTML
<input type="text" name="username" placeholder="Username">
```

Snippet 2:

15 Hands-on exercise: Slop spotter (assessment level L2–L3) (continued)

```python
# [Pseudocode — illustrative only]
connection = connect("mydb", user="admin", password="letmein")
```

Snippet 3:

15 Hands-on exercise: Slop spotter (assessment level L2–L3) (continued)

```javascript
// [Illustrative — not production-ready]
function getOrder(orderId) {
  return database.query("SELECT * FROM orders WHERE id = " +
orderId);
}
```

Constraints: Work on paper or in a text editor. Do not ask an AI to classify these for you — the point is to practice your own judgment.

15 Hands-on exercise: Slop spotter (assessment level L2–L3) (continued)

Expected deliverable: A written classification for each snippet, with the flaw and fix identified for Snippets 2 and 3.

15 Hands-on exercise: Slop spotter (assessment level L2–L3) (continued)

Success criteria: You correctly identify Snippet 1 as acceptable (with a note that it lacks a label, which is a minor accessibility issue), Snippet 2 as rejected (hardcoded password — use environment variables), and Snippet 3 as rejected (SQL injection risk — use parameterized queries). G-03 · G-05

16 Build challenge

● PUSH FURTHER

Task: Write a "human responsibility contract" for yourself: a short, personal statement listing five things you will personally verify before shipping any AI-assisted application.

Constraints: Each item must be observable and checkable (not "be careful" or "be thorough"). Use the professional standard from Section 14 as a guide.

Expected deliverable: A written list of five items, each with a yes/no check.

Success criteria: All five items are specific enough that another person could read your list and verify whether you followed it. Example items: "No API keys appear in source code," "Every form validates input on the server," "A test suite passes before deploy."

17 Key takeaways

KEY TAKEAWAYS

- Vibe coding is a real practice, but its origin is throwaway weekend projects — not professional software.
- Slop is code that runs but is unsafe, untested, or unmaintainable. It looks like software; it is not engineered software.
- AI may get logic wrong, introduce security flaws, mishandle data, miss context, and violate patterns in your codebase.
- Three responsibilities never delegate to AI: defining the problem, judging safety for users, and understanding the code well enough to fix it.
- Before shipping, read every diff, check for secrets, validate input, enforce access control, and run tests.

18 Glossary of new terms

TERMS INTRODUCED IN THIS CHAPTER

18 Glossary of new terms (continued)

TERM	DEFINITION	EXAMPLE
Vibe coding	AI-assisted development where a person describes a goal in natural language and accepts AI-generated code based on results rather than systematic review.	"Just tell the AI what you want and accept whatever it makes."
Slop	Low-quality AI-generated code that runs but is insecure, untested, undocumented, or hard to maintain.	A login form with a hardcoded password.
AI-generated code	Source code produced by an AI model rather than written by a person.	Code suggested by Cursor, Copilot, or Claude Code.
Human-in-the-loop	A person reviews and takes responsibility for AI output.	You read the diff before accepting.
Non-delegable judgment	A decision only the human builder can make.	Deciding whether an app is safe for real users.

18 Glossary of new terms (continued)

TERM	DEFINITION	EXAMPLE
Throwaway project	Software built for learning, not for real users or long-term use.	A weekend experiment.
Production system	Software real users depend on, with security, testing, backups, and monitoring.	Mifugo Market when it serves paying customers.

19 Knowledge-check questions

KNOWLEDGE CHECK

Attempt these before reading the answers below.

1. What is vibe coding, and who coined the term?
2. Was vibe coding originally intended for production software or throwaway projects?
3. Name three things AI-generated code may get wrong.
4. Name one responsibility that cannot be delegated to an AI.
5. What is the difference between a throwaway project and a production system?

ANSWERS AND EXPLANATIONS

(do not read until you have attempted the questions above)

1. **Vibe coding** is a style of AI-assisted development where a person describes a goal in natural language and an AI generates code, with the person accepting or iterating based on results. The term was coined by Andrej Karpathy in February 2025. D-01
2. **Throwaway projects.** Karpathy himself scoped vibe coding to experiments, not products for real users. D-01
3. Examples: logic errors, security flaws (hardcoded secrets, broken access control), data handling problems (no validation, plaintext passwords), missing context, and pattern violations in your codebase. D-01 · D-05 · G-02 · G-03 · G-05
4. Examples: defining the problem to solve, judging whether software is safe for real users, understanding the code well enough to fix it if it breaks. [Constitution §5]
5. A **throwaway project** is built for learning and is not intended for real users. A **production system** serves real users and requires security, testing, backups, and monitoring.

20 Links to earlier and later chapters

WHERE THIS SITS IN THE BOOK

- **Earlier:** Chapter 1 (client-server model, HTTP, status codes). You need that model to understand where AI-generated code fits in an application and what can go wrong.
- **Later:** Chapter 6 introduces AI coding tools and your first deployed page. Chapter 7 teaches structured prompting and context management. Chapter 8 teaches diff review in depth. Chapters 12–13 cover testing and security — the gates that separate slop from professional software.

PART 2

2

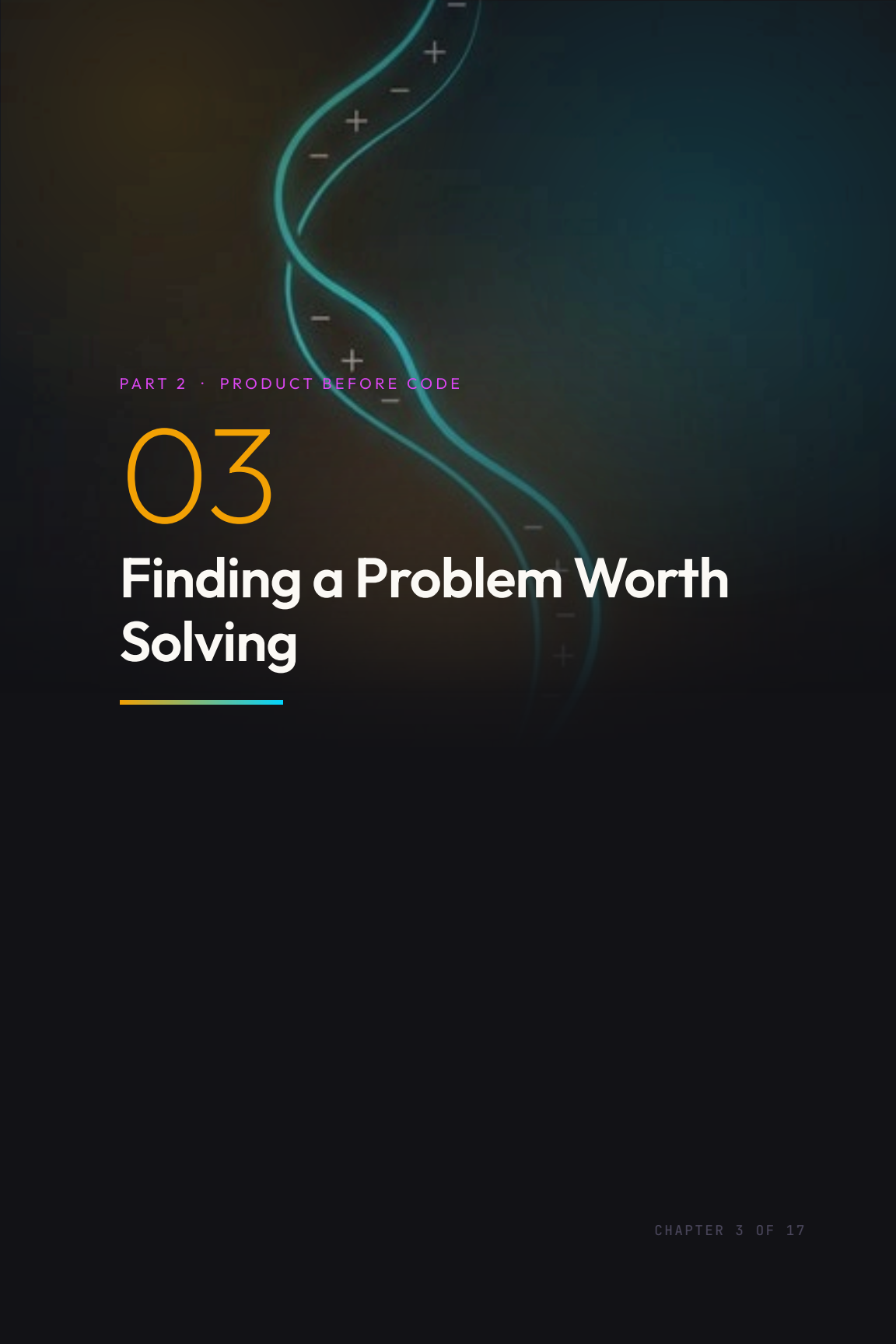
Product Before Code

From idea to a problem worth solving

03 Finding a Problem Worth Solving

04 Users, Journeys, and Requirements

05 Specification, Acceptance Criteria, and Scope Control



PART 2 · PRODUCT BEFORE CODE

03

Finding a Problem Worth Solving

01 Opening problem — the misconception that wastes months of work

SLOP DETECTOR — THE MISCONCEPTION

The scenario

Juma is a carpenter in Ruiru who notices that local furniture-makers struggle to find reliable suppliers of seasoned hardwood. He has seen AI coding tools generate working apps in minutes, so he tells an AI assistant: "Build me a marketplace app for buying and selling timber." The AI produces a slick-looking site with product listings, a shopping cart, and payment buttons. Juma publishes it. Two weeks later, he has exactly zero customers.

The problem? Juma fell in love with a solution before validating the problem. He never confirmed that furniture-makers actually want a digital marketplace, that they would change their current way of buying timber (phone calls to a trusted supplier), or that they would trust an app with their money. He built something nobody asked for.

The misconception

MISCONCEPTION	WHY IT IS DANGEROUS	THE CORRECTION
"My app idea is the starting point."	Starting with a solution means you may build something elegant that solves no real problem. AI makes it cheap to generate code, so the waste is invisible until you have spent weeks and money on nothing.	Start with a validated problem: who has it, why it matters, and what they do today about it. Code comes last, not first.
"If I build it, they will come."	AI-generated code can run perfectly and still have no users. Running code is not demand.	Validate demand before you write production code. A landing-page conversation with five real users beats a polished app with zero users.

02 Why this topic matters in real software projects

Most software projects fail not because the code is bad, but because they solve a problem nobody cares about. AI makes generating code fast and cheap, which makes the trap more dangerous: you can now build a fully working app for a non-existent problem in a weekend.

Constitution §3 identifies **scope paralysis** as a key beginner problem — not knowing what to learn first or what can be deferred. Problem discovery is the antidote. If you cannot write a one-page problem statement, you are not ready to prompt an AI to build anything.

The earlier you validate, the cheaper it is to change direction. Changing a problem statement on paper takes minutes. Changing a deployed, AI-generated application with real users takes weeks and risks losing their trust.

03 Learning objectives

WHAT YOU WILL BE ABLE TO DO

By the end of this chapter, the reader can:

- Write a one-page problem statement covering who, what problem, why it matters, and the current workaround.
- Identify one primary user for a product idea (for example, Wanjiku, the owner of Mifugo Market).
- List three existing alternatives or competitors that the user already relies on.
- Design one simple validation test (a conversation with five potential users, or a landing-page smoke test).
- Decide, with reasoning, whether to proceed, pivot, or abandon an idea.

04 Prerequisites and what to skip

BEFORE YOU START

Prerequisites

Chapters 1–2. You should understand what software is (Chapter 1) and the difference between vibe coding and professional building (Chapter 2), including that running code is not the same as a product people need.

What you can skip

No coding is required in this chapter. You will not install tools, write AI prompts for code, or deploy anything. Problem validation is a thinking and conversation activity, not a technical one.

05 Plain-English explanation

A problem statement is a promise you test

A **problem statement** is a short, written description of:

- **Who** has the problem (the user).
- **What** the problem is, in the user's own words.
- **Why** it matters (the cost of leaving it unsolved).
- **What** they do today to cope (the current workaround).

If you cannot fill in all four blanks, you do not yet understand the problem well enough to solve it with software.

Validation is looking for evidence before you build

Validation means testing whether real people actually have the problem you think they have, and whether they would use your proposed solution. It is not asking friends "is this a good idea?" — they will say yes to be nice. It is finding evidence of real behavior: people who are already trying to solve the problem, spending time or money on workarounds, and willing to try something new.

A smoke test is a cheap experiment

05 Plain-English explanation (continued)

A **smoke test** (in this context) is a small, low-cost experiment that checks demand before you build. Examples: a one-page website describing the product with an email sign-up, or a conversation with five potential users where you ask about their current workflow. If nobody signs up or admits the problem is painful, you have saved yourself months of building.

06 Mental model — and where it breaks down

Analogy: the broken well

Imagine a village where people walk two hours each way to fetch water from a river. An outside builder arrives and installs a modern electric pump next to the river. Nobody uses it. Why? Because the real problem was not "no pump near the river" — it was "water is far and heavy to carry," and the villagers' actual workaround was a communal donkey cart that also served as a social gathering. The builder solved the wrong problem because he fell in love with a technical solution instead of understanding the lived problem.

Where the analogy breaks down

Unlike a well, software can be changed cheaply after launch. But the lesson holds: if you build for a problem that does not exist in your users' lives, no amount of polish will make them use it. You must understand the problem from the user's perspective before you write code.

07 Technical explanation with correct terminology

Each term is defined at first use below and listed in the chapter glossary (Section 18).

07 Technical explanation with correct terminology

(continued)

- **Problem statement** — a one-page description of who has a problem, what it is, why it matters, and what they do today about it.
- **User need** — a specific difficulty or goal a real person has, expressed in their terms, not in features.

07 Technical explanation with correct terminology

(continued)

- **Workaround** — what a person currently does to cope with the problem (manual spreadsheets, phone calls, paper ledgers, doing nothing).
- **Validation** — testing whether real users have the problem and would use your solution, before you build it.

07 Technical explanation with correct terminology

(continued)

- **Smoke test** — a small, low-cost experiment to check demand (landing page, email sign-up, short conversations).
- **Competitor / alternative** — anything the user already uses to address the problem, including pen and paper, not just other apps.
- **Pivot** — changing direction based on evidence: adjusting the problem, the user, or the solution without starting from zero.

07 Technical explanation with correct terminology

(continued)

- **MVP (minimum viable product)** — the smallest version of the product that delivers real value to real users so you can learn. Introduced here; deepened in Chapter 15.

07 Technical explanation with correct terminology

(continued)

- **Prototype** — a quick, possibly non-functional model used to learn or communicate; not built for real users. (The demo/prototype/MVP/production distinction is addressed as labeled guidance — see research gap R-04 below.)

07 Technical explanation with correct terminology

(continued)

[security] — none (product discovery chapter)

08 Concrete example — validating Mifugo Market

IN PRACTICE

Mifugo Market (hypothetical, labeled) is a fresh-produce ordering platform for small shops and households in Nairobi. The owner, Wanjiku, wants customers to browse produce (sukuma wiki, tomatoes, avocados, mangoes) online and place orders she can fulfill.

Before any code, Wanjiku writes her problem statement:

> ***Who:*** *Small shop owners and households in Ruiru and Nairobi who buy fresh produce weekly.* ***Problem:*** *They spend time traveling to multiple markets to compare prices and stock, and they often arrive to find items sold out.* ***Why it matters:*** *Lost time and wasted trips cut into shop margins and frustrate households.* ***Current workaround:*** *Calling several suppliers by phone, or visiting markets in person early in the morning.*

She then lists her alternatives:

08 Concrete example — validating Mifugo Market

(continued)

1. Wakulima Market (physical, in-person only).
2. WhatsApp groups where suppliers post photos and prices informally.
3. Pen-and-paper order sheets kept by shop owners.

She plans validation: five short conversations with shop owners in Ruiru, asking how they currently order, what frustrates them, and whether they would try an online ordering tool. She will also put up a one-page landing page with an email sign-up for "Mifugo Market — order fresh produce online" and see if anyone signs up without her promoting it to friends.

09 Step-by-step demonstration — validating your own idea

STEP BY STEP

Follow these steps for your own product idea, or use Mifugo Market as a practice case.

Step 1: Write the problem statement (one page)

Answer these four questions in plain language:

1. Who has the problem? (Be specific: "small shop owners in Ruiru," not "everyone.")
2. What is the problem, in their words?
3. Why does it matter? (Time lost, money lost, frustration, risk.)
4. What do they do today to cope?

Step 2: List three alternatives

09 Step-by-step demonstration — validating your own idea (continued)

What do these people already use? Include non-digital options (paper, phone, word of mouth). If there are no alternatives at all, be skeptical — it may mean there is no real demand.

Step 3: Choose a validation test

Pick one:

- **Conversation test:** Talk to five real people in your target user group. Ask about their current workflow, not whether they like your idea. Listen for complaints, workarounds, and money already being spent.
- **Landing-page smoke test:** Publish a one-page description of the product with an email sign-up. Drive a few visitors (not friends) and see if anyone signs up.

Step 4: Decide: proceed, pivot, or abandon

09 Step-by-step demonstration — validating your own idea (continued)

Based on the evidence:

09 Step-by-step demonstration — validating your own idea (continued)

- **Proceed** if multiple users confirm the problem is painful and show interest in a solution.
- **Pivot** if the problem is real but different from what you assumed (different user, different pain point).

09 Step-by-step demonstration — validating your own idea (continued) (continued)

- **Abandon** if nobody confirms the problem or nobody is willing to try a solution.

10 AI collaboration example (prompt format P1–P10)

AI COLLABORATION — PROMPT FORMAT P1–P10

PROMPT FORMAT **A reusable prompt you can copy**

AI can help you structure your thinking, but it cannot validate the problem for you — only real users can.

P1 **P1 · PROBLEM IT SOLVES**
Turn raw notes from user conversations into a structured problem statement and validation plan.

P2 **P2 · WHEN TO USE IT**
Chapter 3, after you have done some conversations or have initial notes about a user problem.

P3 **P3 · REQUIRED INPUTS**
Your notes from user conversations, a description of your target user, and a list of current alternatives you have found.

10 AI collaboration example (prompt format P1–P10) (continued)

P4 **P4 · THE PROMPT**

TEXT

```
I am validating a product idea. Here are my notes from
conversations with potential users:
[USER_NOTES]

The target user is: [TARGET_USER]
The alternatives they currently use are: [ALTERNATIVES]

Help me write:
1. A one-page problem statement (who, what problem, why it
matters, current workaround).
2. Three validation questions I should ask next.
3. One smoke-test idea I could run this week.

Do not invent user quotes or statistics. Mark anything that
is an assumption as [ASSUMPTION]. Return plain text, not
code.
```

10 AI collaboration example (prompt format P1–P10) (continued)

P5 **P5 · WHAT A GOOD RESPONSE CONTAINS**

A structured problem statement; clearly labeled assumptions; concrete, testable validation questions; a feasible smoke test.

P6 **P6 · WHAT THE READER MUST INSPECT**

Confirm no user quotes or statistics are invented; confirm assumptions are flagged; confirm the problem statement uses the user's words, not AI-generated marketing language.

10 AI collaboration example (prompt format P1–P10) (continued) (continued)

P7 **P7 · WEAK OR INCOMPLETE RESPONSE**
Invents "research" or statistics; presents assumptions as facts; suggests building an app as the first validation step.

P8 **P8 · FOLLOW-UP WHEN INADEQUATE**
"Remove the invented statistics," "Which of these is an assumption versus something I observed?" "Show me a smoke test that does not require writing code."

10 AI collaboration example (prompt format P1–P10) (continued) (continued)

P9 **P9 · SAFETY/SECURITY CONSIDERATIONS**
Do not share real customers' personal data or confidential business information with the AI. Anonymize notes before pasting them in.

P10 **P10 · VERIFICATION STEP**
Compare the AI's problem statement against your actual conversation notes. Every claim should trace to something a real user said or did, not to the AI's imagination.

11 Human decisions and responsibilities (non-delegable judgment)

● YOUR DECISION — NOT THE AI'S

11 Human decisions and responsibilities (non-delegable judgment) (continued)

- **Whom to talk to.** You choose the real users; the AI cannot know your community.
- **What counts as evidence.** You judge whether a comment is a genuine pain point or polite encouragement.

11 Human decisions and responsibilities (non-delegable judgment) (continued)

- **Whether to proceed, pivot, or abandon.** This is a business judgment the AI cannot make. The AI can summarize your notes; it cannot decide whether to spend your money.

11 Human decisions and responsibilities (non-delegable judgment) (continued)

- **Whether a problem is worth solving with software.** Some problems are better solved with a conversation, a spreadsheet, or a printed form.

11 Human decisions and responsibilities (non-delegable judgment) (continued)

[Human decision-point box] Before you ask an AI to write any code, answer: Have I spoken to at least five real people in my target group? Do I know what they currently do to work around the problem? Would they try my solution? If any answer is no, do not write code yet.

12 Inspect and verify AI output (problem-validation checklist)

QUALITY CHECK

If you use AI to help with validation, confirm:

- ✓ [ ] No statistics or quotes were invented by the AI.
- ✓ [ ] Every assumption is labeled [ASSUMPTION].
- ✓ [ ] The problem statement uses the user's words, not marketing language.
- ✓ [ ] The validation test can be completed this week without writing code.
- ✓ [ ] The decision criteria (proceed/pivot/abandon) are stated in advance, not after seeing the result.

13 Failure modes and recovery

WHEN IT GOES WRONG

FAILURE	SYMPTOM	LIKELY CAUSE	RECOVERY
Solution-first bias	You have a feature list but no confirmed user	Started with the app idea, not the problem	Pause coding; write the problem statement; talk to users
Social desirability bias	Friends say "great idea" but no one signs up	Asking for opinions instead of observing behavior	Ask about current workarounds and past behavior, not future intentions
No alternatives found	You believe no one else solves this	You only looked for apps, not manual workarounds	Search for paper processes, phone-based systems, word-of-mouth
Validation paralysis	You keep researching and never build	Fear of making the wrong choice	Set a deadline: two weeks of validation, then decide

All scenarios in this section are hypothetical and labeled as such — they illustrate failure patterns, not real case studies.

14 Professional standard — quality threshold before coding

PRO STANDARD

An idea is ready for software development only when:

- The problem statement fits on one page and answers who, what, why, and the current workaround.
- You have spoken to at least five people in the target user group (or completed an equivalent smoke test).
- You can name three alternatives they currently use.
- You have a written proceed/pivot/abandon decision with reasoning.

These criteria are the author team's recommended practice for beginners, informed by product-discovery literature (see research gap R-01 below).

15 Hands-on exercise (L2 — Understanding)

BUILD IT

Task: Write a one-page problem statement for your own idea, or for Mifugo Market.

Constraints:

- Do not mention your app or any features.
- Use only what you know or can verify this week — label assumptions.
- Keep it to one page.

Deliverable: A one-page problem statement.

Success criteria:

15 Hands-on exercise (L2 — Understanding) (continued)

- Answers who, what problem, why it matters, current workaround.
- Names at least two alternatives the user already uses.
- Any assumption is labeled [ASSUMPTION].

Assessment level: L2 (Understanding — explains in their own words, gives an example)

Time estimate: 45–60 minutes.

16 Build challenge (L3 — Application) with acceptance criteria

● PUSH FURTHER

Task: Run one validation test and document the result.

Constraints:

- Do not write code.
- If you choose conversations, talk to at least three people (five is the target).
- If you choose a smoke test, publish one page and record sign-ups.

Acceptance criteria (Given–When–Then):

16 Build challenge (L3 — Application) with acceptance criteria (continued)

- **Given** a target user group and a problem statement, **When** you complete three conversations or launch a landing page, **Then** you have at least three dated notes or one live page with sign-up data.
- **Given** the validation results, **When** you review them against your proceed/pivot/abandon criteria, **Then** you have a written decision with reasoning.
- **Given** the decision is to proceed, **When** you move to Chapter 4, **Then** your problem statement becomes the basis for user personas and journeys.

Deliverable: Validation notes (conversation summaries or smoke-test results) plus a written proceed/pivot/abandon decision.

Assessment level: L3 (Application — performs a procedure with guidance)

Time estimate: 90–120 minutes.

17 Key takeaways

KEY TAKEAWAYS

17 Key takeaways (continued)

- Start with the problem, not the app. AI can build fast, but it cannot tell you what is worth building.
- A problem statement answers who, what, why, and the current workaround — on one page.
- Validation means evidence from real users, not opinions from friends.
- A smoke test is cheap; building the wrong app is expensive.
- Decide proceed/pivot/abandon based on evidence, and write down your reasoning.

18 Glossary of new terms

TERMS INTRODUCED IN THIS CHAPTER

18 Glossary of new terms (continued)

TERM	PLAIN-LANGUAGE DEFINITION	EXAMPLE	WHY IT MATTERS
Problem statement	A one-page description of who has a problem, what it is, why it matters, and what they do today.	"Small shop owners in Ruiru waste time traveling between markets to compare produce prices."	Keeps you focused on a real need before you build.
User need	A specific difficulty or goal a real person has, in their own words.	"I need to know which market has sukuma wiki in stock before I leave my shop."	Prevents building features nobody wants.
Workaround	What a person currently does to cope with the problem.	Calling three suppliers by phone every Monday morning.	Shows how painful the problem is and what you are competing against.

18 Glossary of new terms (continued) (continued)

TERM	PLAIN-LANGUAGE DEFINITION	EXAMPLE	WHY IT MATTERS
Validation	Testing whether real users have the problem and would use your solution, before building.	Five conversations with shop owners confirming they miss sales due to stock-outs.	Cheap evidence that you are solving a real problem.
Smoke test	A small, low-cost experiment to check demand.	A one-page site with an email sign-up for "Mifugo Market."	Tests interest without writing application code.
Competitor / alternative	Anything the user already uses to address the problem, including non-digital options.	Wakulima Market; WhatsApp groups; paper order sheets.	You are competing with all of these, not just other apps.

18 Glossary of new terms (continued) (continued)

TERM	PLAIN-LANGUAGE DEFINITION	EXAMPLE	WHY IT MATTERS
Pivot	Changing direction based on evidence: adjusting the problem, user, or solution.	Shifting from "marketplace for all produce" to "ordering tool for one supplier's regular customers."	Lets you correct course cheaply instead of sinking time into a dead end.

18 Glossary of new terms (continued)

TERM	PLAIN-LANGUAGE DEFINITION	EXAMPLE	WHY IT MATTERS
MVP	The smallest version that delivers real value to real users so you can learn.	A simple order form where Wanjiku lists stock and customers message her to order.	Introduced here; defined more fully in Chapter 15.

18 Glossary of new terms (continued) (continued)

TERM	PLAIN-LANGUAGE DEFINITION	EXAMPLE	WHY IT MATTERS
Prototype	A quick, possibly non-functional model used to learn or communicate, not for real users.	A sketch of the order form on paper.	Distinguished from MVP and production in Chapter 15 (labeled guidance; R-04 gap).

19 Knowledge-check questions

KNOWLEDGE CHECK

Attempt these before looking at the answers below.

Q1. What four elements must a one-page problem statement contain?

Q2. Why is asking friends "is this a good idea?" not reliable validation?

Q3. Name two non-digital alternatives that might compete with your software idea.

Q4. True or false: A smoke test requires you to build a working application first.

Q5. What is the difference between proceeding, pivoting, and abandoning?

[PAGE BREAK — ANSWER KEY]

Answers

A1. Who has the problem; what the problem is (in the user's words); why it matters; and what they do today to cope (the current workaround).

19 Knowledge-check questions (continued)

A2. Friends tend to be polite and encouraging rather than honest. They may say "great idea" without ever using or paying for the product. Reliable validation looks at actual behavior and current workarounds, not opinions.

A3. Any manual or informal process the user already relies on — for example, phone calls to suppliers, paper ledgers, WhatsApp groups, physical markets, word-of-mouth referrals.

A4. False. A smoke test is a low-cost experiment (a landing page, an email sign-up, short conversations) that checks demand before any application code is written.

19 Knowledge-check questions (continued)

A5. Proceed: evidence supports continuing. Pivot: the problem or user is different than assumed, so you adjust direction. Abandon: no evidence of a real problem, so you stop before writing code.

20 Links to earlier and later chapters

WHERE THIS SITS IN THE BOOK

Earlier chapters (prerequisites):

- **Chapter 1:** What software is — sets the context that code is a means to solve a problem, not an end in itself.
- **Chapter 2:** Vibe coding and slop — running code for a problem nobody wants is a form of slop.

Later chapters (where this is used next):

20 Links to earlier and later chapters (continued)

- **Chapter 4:** Users, Journeys, and Requirements — your primary user from this chapter becomes the persona; the problem becomes the journey.
- **Chapter 5:** Specification and Acceptance Criteria — your validated problem becomes the scope of version one.
- **Chapter 16:** Capstone — the problem statement you write here is refined into the capstone's opening artifact.

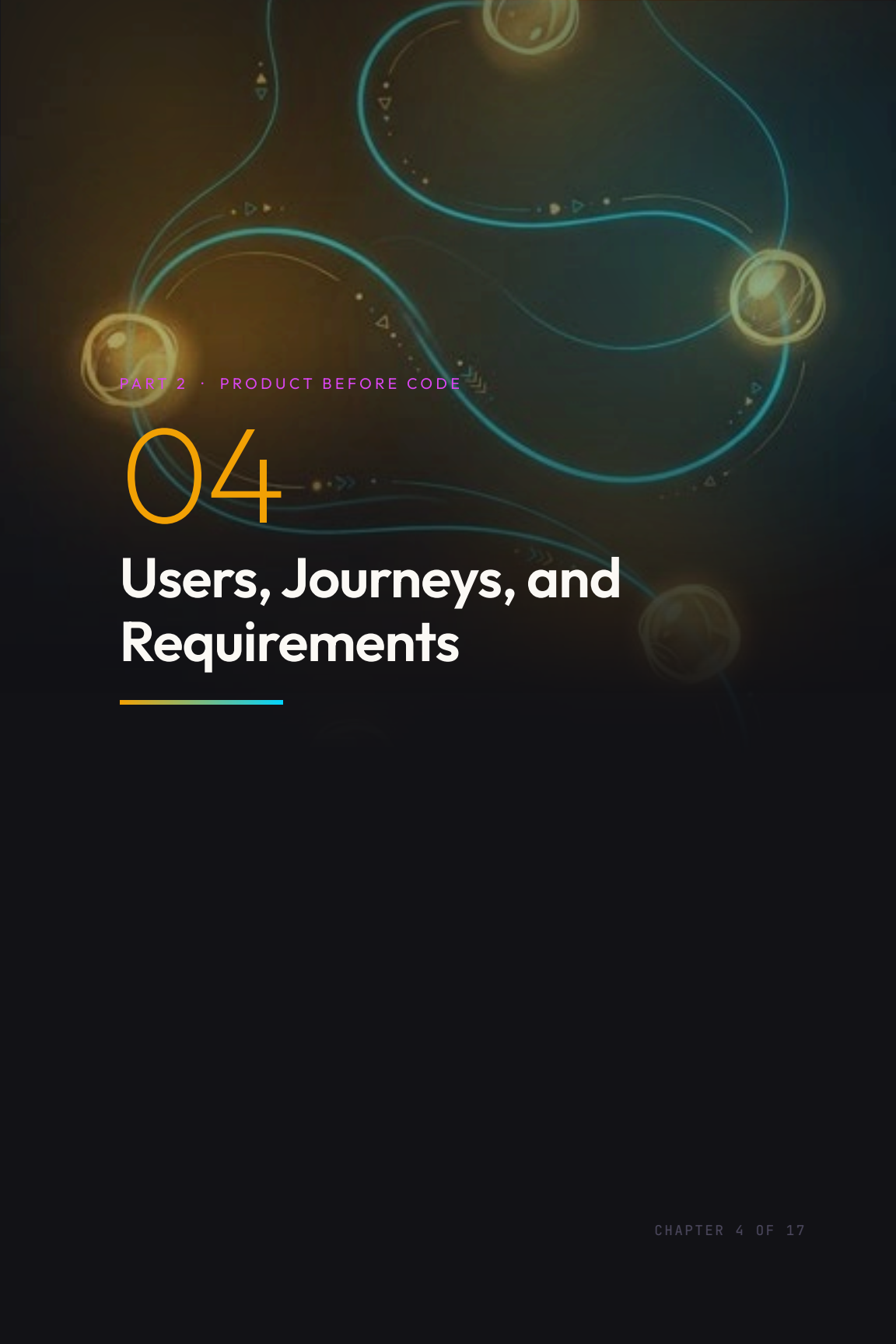
20 Links to earlier and later chapters (continued)

Research gap note: R-01 (problem-validation techniques) has no primary source located yet. The techniques in this chapter (conversations, smoke tests, problem statements) are presented as labeled practitioner guidance, not as industry standards. See research_gaps.md.

Time estimate: Approximately 140 minutes total (80 minutes reading, 60 minutes hands-on practice).

20 Links to earlier and later chapters (continued)

Evidence notes: Claims trace to R-01 (problem-validation gap — labeled guidance), Constitution §2 and §3. All scenarios are hypothetical. (User stories and acceptance criteria are covered in Chapters 4–5, not Chapter 3.)



PART 2 · PRODUCT BEFORE CODE

04

Users, Journeys, and Requirements

01 Opening problem — designing for everyone means designing for no one

SLOP DETECTOR — THE MISCONCEPTION

The scenario

Wanjiku has validated that small shop owners in Ruiru want a simpler way to order fresh produce. She tells her AI assistant: "Build an ordering app for everyone — shop owners, households, hotels, restaurants, and farmers." The AI generates a sprawling application with separate dashboards for each role, complex supplier onboarding, delivery logistics, and farmer verification. Six months later, the app is too complex to use, none of the intended users have tried it, and Wanjiku cannot explain who the primary user is.

She designed for "everyone" and ended up serving no one well.

The misconception

MISCONCEPTION	WHY IT IS DANGEROUS	THE CORRECTION
"Design for everyone."	A product that tries to serve every possible user ends up too complex for any of them. AI makes it easy to add features, so "everyone" quickly becomes an unmanageable feature list.	Choose one primary user and design for them first. Expand only after that user is well served.
"Features are a list of things the app should do."	A feature list describes the solution, not the user's need. It misses the steps, decisions, and edge cases that determine whether the app actually works in real life.	Express requirements as user journeys and user stories focused on what the user wants to achieve.

02 Why this topic matters in real software projects

The most common way software fails is not bad technology — it is building the wrong thing for the wrong people. Before you write code or design screens, you need to know:

- **Who** you are building for (a specific primary user, not "everyone").
- **What** they are trying to achieve (their goal, not your feature list).
- **The steps** they take to get there, including where things can go wrong (the journey).
- **What happens** when things go wrong (empty states, error states, edge cases).

This chapter teaches you to capture that understanding before code, so the AI has clear, user-centered requirements to work with instead of a vague request like "build me an ordering app."

03 Learning objectives

WHAT YOU WILL BE ABLE TO DO

By the end of this chapter, the reader can:

- → Create one primary user persona for Mifugo Market (name, goal, constraint).
- → Draw a user journey for "placing a produce order" with at least five steps and one decision point.
- → Write three user stories in "As a… I want… so that…" format.
- → Identify two edge cases (for example, an item out of stock, or an empty cart).

04 Prerequisites and what to skip

BEFORE YOU START

Prerequisites

Chapter 3. You should have a validated problem statement and a primary user in mind. Chapter 3's problem statement feeds directly into this chapter's persona and journey.

What you can skip

No coding or AI code-generation in this chapter. You will sketch, write, and think. Visual design and wireframing tools are not required — pen and paper or a simple drawing tool is enough.

05 Plain-English explanation

A persona is a stand-in for your primary user

A **persona** is a concrete, specific description of the main person you are building for. Instead of "small business owners" (too vague), you create someone like "Wanjiku, 38, runs a small grocery in Ruiru, orders produce twice a week, uses a basic smartphone, and has limited data." A persona keeps decisions grounded: "Would Wanjiku find this easy to use on her phone?"

A user journey is the path a user takes

A **user journey** maps the steps a person takes to accomplish a goal — from first hearing about your product to completing their task. It shows not just the happy path but also decision points ("is the item in stock?") and where things can break.

User stories capture needs, not features

A **user story** is a short sentence: "As a [user], I want [goal], so that [benefit]." It describes what the user needs to achieve and why, without prescribing the technology. This keeps the focus on value and makes it easier to test later.

05 Plain-English explanation (continued)

Edge cases are the moments that break assumptions

An **edge case** is a situation outside the normal flow: an item is out of stock, the internet cuts out, the cart is empty, a user enters a phone number in the wrong format. Professional software handles these gracefully; sloppy software ignores them until real users hit them.

06 Mental model — and where it breaks down

Analogy: a matatu route

Think of a user journey like a matatu route in Nairobi. The route has a start (the user's goal), stops along the way (each step), junctions where the driver must decide which way to go (decision points), and possible breakdowns or roadblocks (edge cases and error states). A good route gets most passengers to their destination smoothly and has a plan for when something goes wrong.

Where the analogy breaks down

Unlike a matatu passenger, a software user can abandon your app at any step with no cost to themselves. If a step is confusing or fails, they simply leave and may not return. Every step in the journey must earn the user's continued attention.

07 Technical explanation with correct terminology

Each term is defined at first use below and listed in the chapter glossary (Section 18).

07 Technical explanation with correct terminology

(continued)

- **Persona** — a specific, concrete description of the primary user you are designing for (name, role, goal, constraint).
- **Primary user** — the one person or group the product is built for first; others come later.
- **User journey** — a map of the steps a user takes to accomplish a goal, including decision points and possible problems.
- **Decision point** — a step where the user must make a choice that changes what happens next.

07 Technical explanation with correct terminology

(continued)

- **Edge case** — a situation outside the normal flow (out of stock, empty cart, lost connection).
- **User story** — "As a [user], I want [goal], so that [benefit]." A short, user-centered statement of need.
- **Acceptance criteria** — the conditions that must be met for a user story to be considered complete (preview; full treatment in Chapter 5).
- **Empty state** — what the screen shows when there is nothing to display (for example, "no orders yet").

07 Technical explanation with correct terminology

(continued)

- **Error state** — what the screen shows when something goes wrong (for example, "could not connect to the server").

08 Concrete example — Mifugo Market order journey

IN PRACTICE

Primary user (persona): Wanjiku, 38, owner of a small grocery in Ruiru. She orders fresh produce twice a week for her shop. She uses a basic Android phone with limited data. Her goal: restock quickly so she does not disappoint customers.

User journey — placing a produce order:

1. Wanjiku opens Mifugo Market on her phone.
2. She browses the produce list and checks which items are in stock.
3. She adds sukuma wiki and tomatoes to her cart.
4. She reviews her cart and enters her delivery address.
5. She places the order and receives a confirmation.
6. Wanjiku receives her delivery and checks the items against her order.

Decision point: At step 2, if an item is out of stock, does she choose a substitute, wait, or order elsewhere? The journey must handle this.

Edge cases:

- The item she wants is out of stock when she tries to order.
- Her cart is empty when she tries to check out.
- Her internet connection cuts out while placing the order.

09 Step-by-step demonstration — creating your persona and journey

STEP BY STEP

Step 1: Create one primary persona

Fill in these fields:

- **Name:** (give them a name — it makes them real)
- **Role:** (what do they do?)
- **Goal:** (what are they trying to achieve with your product?)
- **Constraint:** (what limits them? limited data, basic phone, limited time, limited budget)

Write one paragraph describing a typical day and how your product fits into it.

Step 2: Draw the user journey

On paper or in a simple drawing tool, lay out the steps:

1. Start: what triggers the user to begin?
2. Steps: what does the user do, in order?
3. Decision point: where must the user choose, and what are the options?
4. End: how does the user know they succeeded?
5. Edge cases: where could things go wrong at each step?

Aim for at least five steps and one decision point.

Step 3: Write three user stories

Use the format: "As a [user], I want [goal], so that [benefit]."

Example for Mifugo Market:

09 Step-by-step demonstration — creating your persona and journey (continued)

- "As a shop owner, I want to see which produce items are in stock, so that I do not order items I cannot get."
- "As a shop owner, I want to review my cart before ordering, so that I can check the total cost."
- "As a shop owner, I want to receive an order confirmation, so that I know my order was received."

Step 4: Identify two edge cases

For each user story, ask: what could go wrong? Write down two edge cases and one sentence on how the app should respond.

10 AI collaboration example (prompt format P1–P10)

AI COLLABORATION — PROMPT FORMAT P1–P10

PROMPT FORMAT **A reusable prompt you can copy**

AI can help draft user stories and journeys, but it cannot know your users. You must supply the real detail.

P1 **P1 · PROBLEM IT SOLVES**
Turn a validated problem and persona into user stories and a journey outline.

P2 **P2 · WHEN TO USE IT**
Chapter 4, after you have a persona and problem statement from Chapter 3.

P3 **P3 · REQUIRED INPUTS**
Your persona description, your problem statement, and any notes from user conversations.

10 AI collaboration example (prompt format P1–P10) (continued)

P4 **P4 · THE PROMPT**

```
TEXT
I am designing a product for this primary user:
[PERSONA_DESCRIPTION]

The problem they have is:
[PROBLEM_STATEMENT]

Help me draft:
1. A user journey for their main goal, with at least five
steps and one decision point.
2. Three user stories in "As a / I want / so that" format.
3. Two edge cases and how the app should respond to each.

Base everything on the user description above. Mark
anything that is an assumption as [ASSUMPTION]. Do not
invent user names, statistics, or quotes.
```

10 AI collaboration example (prompt format P1–P10) (continued)

P5 **P5 · WHAT A GOOD RESPONSE CONTAINS**

A journey with steps and a decision point; three user stories in the correct format; two edge cases with responses; assumptions labeled.

P6 **P6 · WHAT THE READER MUST INSPECT**

Confirm no invented quotes or statistics; confirm assumptions are flagged; confirm the user stories focus on the user's goal, not on technology features.

10 AI collaboration example (prompt format P1–P10) (continued) (continued)

P7 **P7 · WEAK OR INCOMPLETE RESPONSE**
Suggests features ("I want a React dashboard") instead of goals; invents user data; skips edge cases.

P8 **P8 · FOLLOW-UP WHEN INADEQUATE**
"Rewrite the user stories without mentioning any technology," "Add a decision point where the item is out of stock," "What assumption are you making here?"

10 AI collaboration example (prompt format P1–P10) (continued) (continued)

P9 **P9 · SAFETY/SECURITY CONSIDERATIONS**
Do not paste real customers' personal data or confidential business information into the AI. Anonymize any notes.

P10 **P10 · VERIFICATION STEP**
Compare each user story against your conversation notes. Every story should describe something a real user said they needed, not something the AI imagined.

11 Human decisions and responsibilities (non-delegable judgment)

● **YOUR DECISION — NOT THE AI'S**

11 Human decisions and responsibilities (non-delegable judgment) (continued)

- **Who the primary user is.** You choose; the AI cannot know your market.
- **Which journey matters most.** You decide which goal to design for first.

11 Human decisions and responsibilities (non-delegable judgment) (continued)

- **Which edge cases to handle now versus later.** You prioritize based on how often they happen and how painful they are.
- **What counts as "done" for a user story.** You set the acceptance criteria (Chapter 5).

11 Human decisions and responsibilities (non-delegable judgment) (continued)

[Human decision-point box] Before you ask an AI to generate screens or code, answer: Who is my primary user? What is their main goal? What are the five steps they take? If you cannot answer, do not build yet.

12 Inspect and verify AI output (journey checklist)

QUALITY CHECK

If you use AI to help draft journeys or stories, confirm:

- ✓ [ ] The journey has at least five steps and one decision point.
- ✓ [ ] User stories follow "As a / I want / so that" and describe user goals, not features.
- ✓ [ ] Edge cases are included (out of stock, empty cart, connection lost).
- ✓ [ ] No invented statistics, quotes, or user names.
- ✓ [ ] Assumptions are labeled [ASSUMPTION].

13 Failure modes and recovery

WHEN IT GOES WRONG

FAILURE	SYMPTOM	LIKELY CAUSE	RECOVERY
Designing for everyone	Persona is vague ("small businesses")	No primary user chosen	Pick one specific person; design for them first
Feature-list thinking	Stories read like "I want a database"	Focus on solution instead of need	Rewrite stories using the user's goal and benefit
Missing edge cases	Journey only shows the happy path	Not considering what can go wrong	Add out-of-stock, empty-cart, and connection-lost steps
No decision point	Journey is a straight line with no choices	Oversimplified	Add a real choice the user must make

All scenarios in this section are hypothetical and labeled as such.

14 Professional standard — quality threshold before design

PRO STANDARD

You are ready to move to screens and code only when:

- You have one named primary persona with a goal and a constraint.
- Your user journey has at least five steps, one decision point, and identified edge cases.
- You have three user stories in "As a / I want / so that" format.
- Edge cases have a stated response (even if the response is "show a friendly message").

These criteria are the author team's recommended practice for beginners, informed by user-story and user-flow literature (B-01, C-01, C-02).

15 Hands-on exercise (L3 — Application)

BUILD IT

Task: Create a persona, draw a user journey, and write three user stories for your own idea or for Mifugo Market.

Constraints:

- One primary persona only — do not design for multiple user types yet.
- Pen and paper or a simple drawing tool is sufficient; no design software required.
- No code.

Deliverable: One persona, one journey diagram (five steps + one decision point), three user stories, two edge cases.

Success criteria:

15 Hands-on exercise (L3 — Application) (continued)

- Persona has a name, role, goal, and constraint.
- Journey has five or more steps and at least one decision point.
- User stories use "As a / I want / so that" format.
- Two edge cases are named with a response for each.

Assessment level: L3 (Application — performs a procedure with guidance)

Time estimate: 70–90 minutes.

16 Build challenge (L3 — Application) with acceptance criteria

PUSH FURTHER

Task: Review your journey for completeness and prepare it to become acceptance criteria in Chapter 5.

Constraints:

- Do not write code.
- Use only the persona and journey you created in the exercise.

Acceptance criteria (Given–When–Then):

16 Build challenge (L3 — Application) with acceptance criteria (continued)

- **Given** a primary persona is defined, **When** another person reads the persona, **Then** they can describe who the user is and what their main constraint is.
- **Given** the user journey is reviewed, **When** the journey is traced step by step, **Then** every step has a clear next action and the decision point has at least two options.
- **Given** edge cases are identified, **When** an edge case occurs (for example, item out of stock), **Then** the journey shows what the user sees and what happens next.
- **Given** user stories are written, **When** each story is read, **Then** it follows "As a / I want / so that" and contains no technology references.

Deliverable: A reviewed journey document plus a note of any gaps found and how they will be fixed.

Assessment level: L3 (Application — performs a procedure with guidance)

16 Build challenge (L3 — Application) with acceptance criteria (continued)

Time estimate: 45–60 minutes.

17 Key takeaways

KEY TAKEAWAYS

- Design for one primary user, not "everyone."
- A persona makes your user concrete: name, role, goal, constraint.
- A user journey maps the steps, decision points, and where things can break.
- User stories express needs ("so that…"), not features.
- Edge cases (out of stock, empty cart, lost connection) are where professional software separates itself from slop.

18 Glossary of new terms

TERMS INTRODUCED IN THIS CHAPTER

18 Glossary of new terms (continued)

TERM	PLAIN-LANGUAGE DEFINITION	EXAMPLE	WHY IT MATTERS
Persona	A specific, concrete description of the primary user you are designing for.	"Wanjiku, 38, grocery owner in Ruiru, orders produce twice a week on a basic Android phone."	Keeps design decisions grounded in a real person.
Primary user	The one person or group the product is built for first.	Wanjiku the shop owner (not hotels, farmers, or households — yet).	Prevents building a sprawling product that serves no one well.
User journey	A map of the steps a user takes to accomplish a goal.	Open app → browse → add to cart → checkout → confirm → receive delivery.	Shows the full experience, not just individual screens.
Decision point	A step where the user must choose, changing what happens next.	Item out of stock: choose substitute, wait, or order elsewhere.	Forces you to design for real user choices, not just the happy path.

18 Glossary of new terms (continued) (continued)

TERM	PLAIN-LANGUAGE DEFINITION	EXAMPLE	WHY IT MATTERS
Edge case	A situation outside the normal flow.	Cart is empty at checkout; internet cuts out during payment.	Professional software handles these gracefully; slop ignores them.
User story	"As a [user], I want [goal], so that [benefit]."	"As a shop owner, I want to see stock levels, so that I do not order unavailable items."	Captures user need without prescribing technology.
Acceptance criteria	Conditions that must be met for a story to be complete. (Full treatment in Chapter 5.)	"Given an item is out of stock, Then it shows 'Sold out' and cannot be added to cart."	Turns stories into testable requirements.

18 Glossary of new terms (continued) (continued)

TERM	PLAIN-LANGUAGE DEFINITION	EXAMPLE	WHY IT MATTERS
Empty state	What the screen shows when there is nothing to display.	"No orders yet" screen.	Prevents confusing blank screens.

18 Glossary of new terms (continued)

TERM	PLAIN-LANGUAGE DEFINITION	EXAMPLE	WHY IT MATTERS
Error state	What the screen shows when something goes wrong.	"Could not connect to the server. Try again."	Tells users what happened and what to do next.

19 Knowledge-check questions

KNOWLEDGE CHECK

Attempt these before looking at the answers below.

Q1. What is the difference between a primary user and "everyone"?

Q2. What three parts does a user story contain?

Q3. Name two edge cases that could occur in a produce-ordering journey.

Q4. What is a decision point in a user journey?

Q5. True or false: A user journey should only show the happy path where everything works.

[PAGE BREAK — ANSWER KEY]

Answers

A1. "Everyone" is too vague and leads to a complex product that serves no one well. A primary user is one specific person or group you design for first (for example, Wanjiku, the Ruiru grocery owner).

A2. "As a [user], I want [goal], so that [benefit]."

A3. Any two of: item out of stock, empty cart at checkout, internet connection lost during order, wrong phone number format, item price changed between browsing and checkout.

A4. A step where the user must make a choice that changes what happens next (for example, whether to choose a substitute when an item is out of stock).

A5. False. A good user journey includes error paths and edge cases, not just the happy path.

20 Links to earlier and later chapters

WHERE THIS SITS IN THE BOOK

Earlier chapters (prerequisites):

- **Chapter 3:** Finding a Problem Worth Solving — your validated problem statement and primary user feed directly into this chapter's persona and journey.

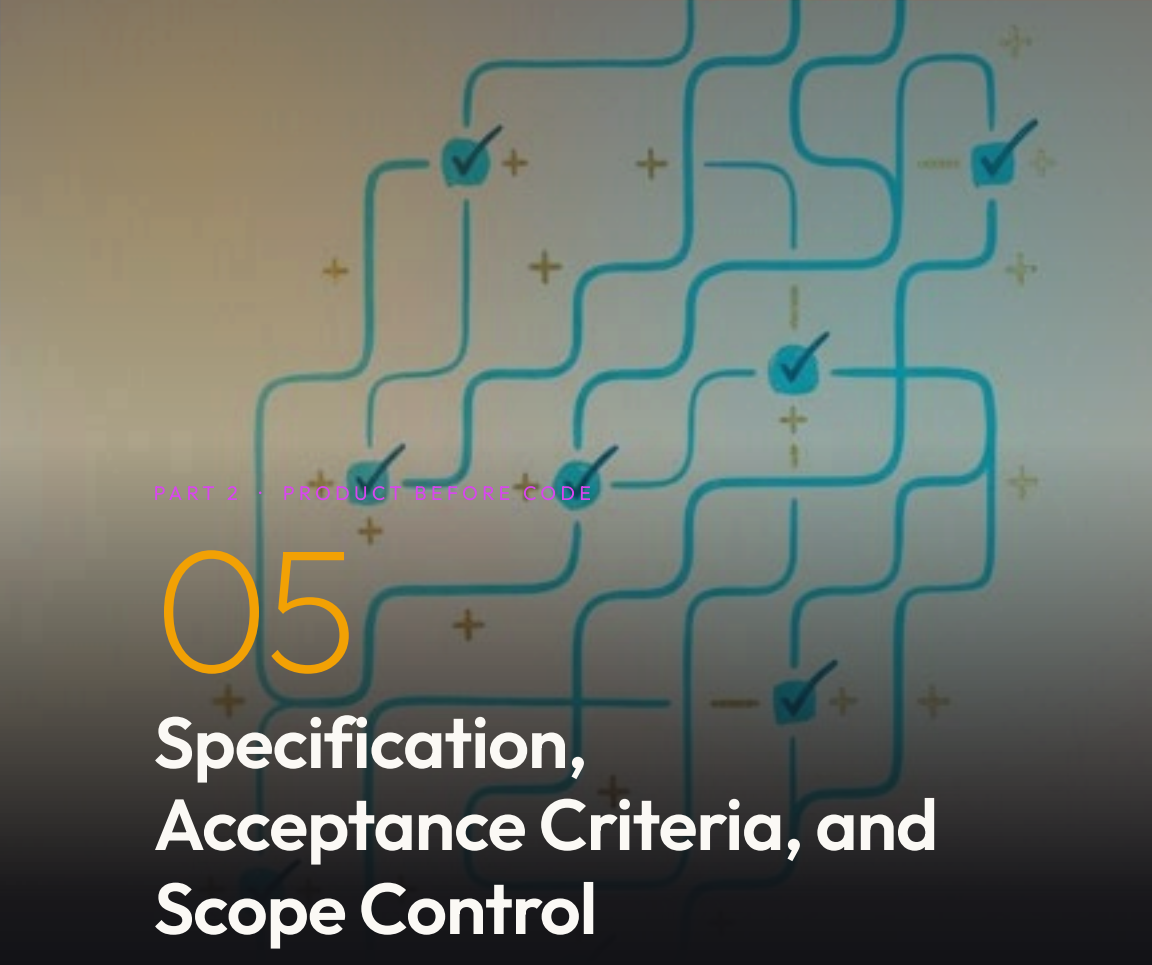
Later chapters (where this is used next):

- **Chapter 5:** Specification and Acceptance Criteria — your user stories become acceptance criteria; your edge cases become test scenarios.
- **Chapter 9:** Frontend — your journey becomes the screens and flows you build.
- **Chapter 12:** Testing — your edge cases become test cases.

Research note: User-story format (B-01, Atlassian) and user-flow/IA concepts (C-01, C-02) are established practices. The persona and journey techniques are presented as the author team's recommended practice for beginners.

Time estimate: Approximately 120 minutes total (70 minutes reading, 50 minutes hands-on practice).

Evidence notes: Claims trace to B-01 (Atlassian user stories), C-01/C-02 (MockFlow IA and user flows), Constitution §4 (product learning outcome). All scenarios are hypothetical.



PART 2 · PRODUCT BEFORE CODE

05

Specification, Acceptance Criteria, and Scope Control

01 Opening problem — a vague prompt produces slop

SLOP DETECTOR — THE MISCONCEPTION

The scenario

Wanjiku has a validated problem and a primary user. She opens her AI coding assistant and types: "Build me a produce ordering app for my shop in Nairobi." The AI generates an application with user accounts, supplier onboarding, delivery tracking, payment integration, and analytics. It looks impressive. But when Wanjiku tries to use it, she cannot simply list today's stock and let a customer place an order — the core of her business. The AI built everything except what she actually needed, because she never defined what "done" looks like.

She gave the AI a vague idea and accepted whatever came back. That is how slop is made.

The misconception

MISCONCEPTION	WHY IT IS DANGEROUS	THE CORRECTION
"I'll describe my idea and the AI will build the right thing."	Without acceptance criteria, there is no definition of success. You cannot tell whether the AI's output is correct, so you accept code that may be wrong, incomplete, or insecure.	Write a specification with testable acceptance criteria first. Criteria become the test you run against AI output.
"More acceptance criteria are better."	Twenty criteria per story are impossible to test and maintain. The AI will miss some, and you will not notice.	Aim for 3–5 testable criteria per story (B-04).

02 Why this topic matters in real software projects

A specification is the bridge between a problem and code. It answers: what are we building, for whom, within what constraints, and how will we know it works?

AI coding assistants are powerful, but they are literal. They build exactly what you ask for — and nothing more. If you ask for "an ordering app," you get an AI's idea of an ordering app. If you ask for "a page where Wanjiku can list produce items with name, price, and stock status, and customers can add items to a cart only when stock is available," you get something you can actually verify.

Acceptance criteria turn vague intentions into observable pass/fail checks. They also protect you from scope creep: once you define what is in version one, everything else is explicitly out of scope.

This chapter directly corrects misconception M-01 ("if the AI wrote it and it runs, it's professional software") — because running code without acceptance criteria cannot be verified as correct.

03 Learning objectives

WHAT YOU WILL BE ABLE TO DO

By the end of this chapter, the reader can:

- → Write a one-page specification for a Mifugo Market feature (goal, users, scope, constraints).
- → Write 3–5 acceptance criteria per user story using Given-When-Then.
- → State what is out of scope for the first version.
- → Explain how acceptance criteria become tests.

04 Prerequisites and what to skip

BEFORE YOU START

Prerequisites

Chapters 3–4. You need a validated problem statement, a primary persona, and user stories from Chapter 4.

What you can skip

No coding or tool setup in this chapter. You will write documents, not software. The formal Given-When-Then syntax is introduced here; no testing tools are required yet (testing is Chapter 12).

05 Plain-English explanation

A specification is a one-page answer to six questions

For each feature, write down:

1. **Goal:** what does this feature achieve for the user?
2. **Users:** who uses it?
3. **Scope:** what is included in version one?
4. **Constraints:** what limits apply (budget, device, data, time)?
5. **Acceptance criteria:** how do we know it works?
6. **Out of scope:** what is deliberately not included?

Acceptance criteria use Given-When-Then

Given-When-Then is a simple format:

- **Given** [the starting context]
- **When** [the user takes an action]
- **Then** [the expected outcome]

Each criterion should be testable: you can read it and say "yes, this works" or "no, it does not."

Out of scope is as important as in scope

05 Plain-English explanation (continued)

Every feature list should end with what is *not* being built in version one. This prevents the AI (and you) from adding "just one more thing" until the project never ships.

06 Mental model — and where it breaks down

Analogy: a market stall order slip

Imagine Wanjiku supplies produce to a shop. The shop owner sends an order slip. A good slip says: "5 kg sukuma wiki, 3 kg tomatoes, deliver by 8 a.m., pay on delivery." A bad slip says: "some vegetables." The good slip can be checked against the delivery — either the right items arrived or they did not. The bad slip cannot be checked, so disagreements are inevitable.

Acceptance criteria are your order slip to the AI. They make the request checkable.

Where the analogy breaks down

Unlike a paper order slip, software acceptance criteria can be automated into tests (Chapter 12). But the principle is the same: the criteria must be specific enough that someone else could verify them without asking you.

07 Technical explanation with correct terminology

Each term is defined at first use below and listed in the chapter glossary (Section 18).

07 Technical explanation with correct terminology

(continued)

- **Specification** — a one-page document describing a feature's goal, users, scope, constraints, acceptance criteria, and what is out of scope.
- **Acceptance criteria** — the conditions that must be met for a feature or user story to be considered complete.
- **Given-When-Then** — a format for writing acceptance criteria: Given [context], When [action], Then [outcome].
- **Out of scope** — items deliberately excluded from the current version.

07 Technical explanation with correct terminology

(continued)

- **Definition of done** — the shared agreement on what must be true before a feature is considered finished.
- **Scope creep** — the tendency for a project to grow beyond its original boundaries, one small addition at a time.
- **Testable** — a criterion that can be checked as pass or fail without opinion.

08 Concrete example — Mifugo Market product catalog

IN PRACTICE

Feature: Customer browses the produce catalog.

Specification (one page):

08 Concrete example — Mifugo Market product catalog (continued)

- **Goal:** Customers can see what produce is available and its price before ordering.
- **Users:** Customers (shop owners and households).
- **Scope (v1):** View list of produce items; each item shows name, price, and stock status; out-of-stock items cannot be added to cart.

08 Concrete example — Mifugo Market product catalog (continued)

- **Constraints:** Works on a basic smartphone; loads on limited data; no login required for browsing.
- **Out of scope (v1):** Search, filtering by category, user accounts, delivery scheduling, reviews.

08 Concrete example — Mifugo Market product catalog (continued)

Acceptance criteria (Given-When-Then):

08 Concrete example — Mifugo Market product catalog (continued)

1. Given the customer is on the homepage, When they view the catalog, Then they see produce items with name, price, and stock status.

08 Concrete example — Mifugo Market product catalog (continued) (continued)

1. Given an item is in stock, When the customer clicks "Add to cart," Then the item is added and the cart count increases.
2. Given an item is out of stock, When the customer views it, Then it shows "Sold out" and cannot be added to cart.

08 Concrete example — Mifugo Market product catalog (continued)

1. Given the customer adds items to the cart, When they view the cart, Then they see the total cost.
2. Given the page loads on a basic smartphone with limited data, Then the catalog appears within a reasonable time (performance criterion).

09 Step-by-step demonstration — writing a spec and criteria

STEP BY STEP

Step 1: Pick one feature

Choose one user story from Chapter 4. For Mifugo Market: "As a shop owner, I want to add a new produce item, so that customers can order it."

Step 2: Write the one-page specification

Answer the six questions above. Keep it to one page. Be specific about constraints (device, data, offline behavior).

Step 3: Write 3–5 acceptance criteria

09 Step-by-step demonstration — writing a spec and criteria (continued)

Use Given-When-Then. Each criterion must be testable. Aim for 3–5 per story (B-04); more than that is hard to maintain.

Step 4: Mark what is out of scope

List what version one will not do. This is your protection against scope creep.

Step 5: Review against the checklist

Use the inspection checklist in Section 12 before you show the spec to an AI or build anything.

10 AI collaboration example (prompt format P1–P10)

AI COLLABORATION — PROMPT FORMAT P1–P10

PROMPT FORMAT **A reusable prompt you can copy**

AI can help draft specifications and criteria, but you must define the feature first — the AI should not invent your product.

P1 **P1 · PROBLEM IT SOLVES**
Turn a user story into a specification with testable acceptance criteria.

P2 **P2 · WHEN TO USE IT**
Chapter 5, after you have a user story from Chapter 4.

P3 **P3 · REQUIRED INPUTS**
The user story, your persona, and any constraints (device, data, timeline).

10 AI collaboration example (prompt format P1–P10) (continued)

P4 **P4 · THE PROMPT**

```
TEXT
I am writing a specification for this feature:
[USER_STORY FROM CHAPTER 4]

Primary user: [PERSONA]
Constraints: [DEVICE, DATA, TIMELINE, BUDGET]

Help me write:
1. A one-page specification (goal, users, scope,
constraints, out of scope).
2. Four acceptance criteria in Given-When-Then format.
3. What is explicitly out of scope for version one.

Make every criterion testable (pass/fail, no opinion words
like "fast" or "easy"). Mark assumptions as [ASSUMPTION].
Do not invent user data or statistics.
```

10 AI collaboration example (prompt format P1–P10) (continued)

P5 **P5 · WHAT A GOOD RESPONSE CONTAINS**

One-page spec; 3–5 Given-When-Then criteria that are testable; an out-of-scope list; assumptions labeled.

P6 **P6 · WHAT THE READER MUST INSPECT**

Confirm every criterion is testable; confirm no opinion words ("fast," "user-friendly," "modern"); confirm out of scope is explicit; confirm no invented statistics.

10 AI collaboration example (prompt format P1–P10) (continued) (continued)

P7 **P7 · WEAK OR INCOMPLETE RESPONSE**
Criteria like "the app should be fast" or "the interface should be intuitive"; no out-of-scope list; more than five criteria per story; criteria that dictate implementation ("use React") rather than behavior.

P8 **P8 · FOLLOW-UP WHEN INADEQUATE**
"Rewrite criterion 2 so it is pass/fail," "Add an out-of-scope section," "Which of these is an assumption?"

10 AI collaboration example (prompt format P1–P10) (continued) (continued)

P9 **P9 · SAFETY/SECURITY CONSIDERATIONS**
Do not paste real customer data or confidential pricing into the AI. Anonymize examples.

P10 **P10 · VERIFICATION STEP**
Read each acceptance criterion and ask: "Could I verify this with a yes/no answer?" If not, rewrite it.

11 Human decisions and responsibilities (non-delegable judgment)

● YOUR DECISION — NOT THE AI'S

11 Human decisions and responsibilities (non-delegable judgment) (continued)

- **What is in version one.** You decide the scope; the AI will happily add everything.
- **What is out of scope.** You must explicitly exclude items; otherwise scope creep is inevitable.

11 Human decisions and responsibilities (non-delegable judgment) (continued)

- **Whether a criterion is testable.** You judge whether "fast" means "under 2 seconds" or something else.
- **Whether to proceed to code.** You confirm the spec is complete before asking the AI to build.

11 Human decisions and responsibilities (non-delegable judgment) (continued)

[Human decision-point box] Before you prompt the AI to build a feature, answer: Do I have a one-page spec? Are there 3–5 testable acceptance criteria? Is out of scope listed? If any answer is no, write the spec first.

12 Inspect and verify AI output (specification checklist)

QUALITY CHECK

If you use AI to help with specifications, confirm:

- ✓ [ ] The spec answers all six questions (goal, users, scope, constraints, criteria, out of scope).
- ✓ [ ] There are 3–5 acceptance criteria per story.
- ✓ [ ] Every criterion uses Given-When-Then and is testable (pass/fail).
- ✓ [ ] No opinion words (fast, easy, modern, beautiful).
- ✓ [ ] Out of scope is explicitly listed.
- ✓ [ ] Assumptions are labeled [ASSUMPTION].

13 Failure modes and recovery

WHEN IT GOES WRONG

FAILURE	SYMPTOM	LIKELY CAUSE	RECOVERY
Vague criteria	"The app should be fast and easy to use"	Opinion words instead of measurable outcomes	Rewrite with specific, testable thresholds
Too many criteria	15+ criteria per story	Trying to cover every possibility	Trim to 3–5; move the rest to future versions
No out-of-scope list	Project keeps growing	Nothing is explicitly excluded	Add an out-of-scope section before coding
Criteria dictate implementation	"Use React and Firebase"	Solution mixed with requirement	Rewrite to describe behavior, not technology
No connection to user stories	Criteria test features nobody asked for	Spec written without the persona	Trace each criterion back to a user story

All scenarios in this section are hypothetical and labeled as such.

14 Professional standard — quality threshold before building

PRO STANDARD

A feature is ready for AI implementation only when:

- The specification fits on one page and answers all six questions.
- There are 3–5 acceptance criteria per user story.
- Every criterion is testable (pass/fail, no opinion words).
- Out of scope is explicitly listed.
- Each criterion traces back to a user story and the validated problem.

These criteria are the author team's recommended practice, informed by acceptance-criteria literature (B-02, B-03, B-04).

15 Hands-on exercise (L3 — Application)

BUILD IT

Task: Write a one-page specification and acceptance criteria for one Mifugo Market feature (or your own validated feature).

Constraints:

- One feature only.
- No code.
- One page maximum for the specification.

Deliverable: One specification with 3–5 Given-When-Then acceptance criteria and an out-of-scope list.

Success criteria:

15 Hands-on exercise (L3 — Application) (continued)

- Spec answers all six questions.
- 3–5 acceptance criteria, all testable (pass/fail).
- Out of scope explicitly listed.
- No opinion words in criteria.

Assessment level: L3 (Application — performs a procedure with guidance)

Time estimate: 60–75 minutes.

16 Build challenge (L3 — Application) with acceptance criteria

PUSH FURTHER

Task: Review your specification against the checklist and convert one acceptance criterion into a test you could run.

Constraints:

- Use the specification from the exercise above.
- Do not write code; write the test in plain language.

Acceptance criteria (Given–When–Then):

- **Given** a completed specification, **When** it is reviewed against the Section 12 checklist, **Then** every item is checked off or a gap is noted with a fix plan.
- **Given** one acceptance criterion, **When** it is rewritten as a test, **Then** the test has a clear pass/fail result and requires no opinion to judge.
- **Given** the out-of-scope list, **When** someone proposes adding an out-of-scope item to version one, **Then** there is a recorded decision to defer it.

16 Build challenge (L3 — Application) with acceptance criteria (continued)

Deliverable: A checklist review of your spec, plus one acceptance criterion rewritten as a plain-language test.

Assessment level: L3 (Application — performs a procedure with guidance)

Time estimate: 45–60 minutes.

17 Key takeaways

KEY TAKEAWAYS

- A vague prompt produces slop; a specification produces verifiable software.
- Acceptance criteria use Given-When-Then and must be testable (pass/fail).
- Aim for 3–5 criteria per user story; more is hard to maintain.
- Out of scope is as important as in scope — it prevents scope creep.
- Acceptance criteria become your tests in Chapter 12.

18 Glossary of new terms

TERMS INTRODUCED IN THIS CHAPTER

18 Glossary of new terms (continued)

TERM	PLAIN-LANGUAGE DEFINITION	EXAMPLE	WHY IT MATTERS
Specification	A one-page document describing a feature's goal, users, scope, constraints, acceptance criteria, and out of scope.	"Feature: browse catalog. Goal: customers see available produce. Users: shop owners. Out of scope: search, reviews."	Makes the request clear and checkable before code is written.

18 Glossary of new terms (continued)

TERM	PLAIN-LANGUAGE DEFINITION	EXAMPLE	WHY IT MATTERS
Acceptance criteria	Conditions that must be met for a feature to be complete.	"Given an item is out of stock, Then it shows 'Sold out' and cannot be added to cart."	Turns a vague feature into something you can verify.

18 Glossary of new terms (continued)

TERM	PLAIN-LANGUAGE DEFINITION	EXAMPLE	WHY IT MATTERS
Given-When-Then	A format: Given [context], When [action], Then [outcome].	"Given the customer is on the homepage, When they view the catalog, Then they see items with name, price, and stock."	Forces clarity about state, action, and outcome.

18 Glossary of new terms (continued)

TERM	PLAIN-LANGUAGE DEFINITION	EXAMPLE	WHY IT MATTERS
Out of scope	Items deliberately excluded from the current version.	"Search and filtering are out of scope for version one."	Prevents scope creep and keeps version one shippable.

18 Glossary of new terms (continued)

TERM	PLAIN-LANGUAGE DEFINITION	EXAMPLE	WHY IT MATTERS
Definition of done	The shared agreement on what must be true before a feature is finished.	All acceptance criteria pass; code reviewed; tested on a phone.	Everyone agrees on when work is complete.

18 Glossary of new terms (continued)

TERM	PLAIN-LANGUAGE DEFINITION	EXAMPLE	WHY IT MATTERS
Scope creep	A project growing beyond its original boundaries.	Adding delivery tracking, reviews, and analytics to a simple ordering app.	The enemy of shipping; killed by an explicit out-of-scope list.
Testable	A criterion that can be checked as pass or fail without opinion.	"Page loads within 3 seconds" is testable; "the app is fast" is not.	Lets you verify AI output objectively.

19 Knowledge-check questions

KNOWLEDGE CHECK

Attempt these before looking at the answers below.

Q1. What six questions should a one-page specification answer?

Q2. What is the Given-When-Then format?

Q3. How many acceptance criteria should you aim for per user story?

Q4. Why is "the app should be fast" a bad acceptance criterion?

Q5. What is out of scope, and why does it matter?

[PAGE BREAK — ANSWER KEY]

Answers

A1. Goal, users, scope, constraints, acceptance criteria, and out of scope.

A2. Given [the starting context], When [the user takes an action], Then [the expected outcome].

A3. 3–5 per user story (B-04). More than that is hard to test and maintain.

A4. "Fast" is an opinion word — it cannot be verified as pass or fail without a specific threshold (for example, "loads within 3 seconds").

A5. Out of scope is what is deliberately not included in version one. It matters because it prevents scope creep and keeps the first version shippable.

20 Links to earlier and later chapters

WHERE THIS SITS IN THE BOOK

Earlier chapters (prerequisites):

- **Chapter 3:** Finding a Problem Worth Solving — the validated problem statement is the basis for the specification.
- **Chapter 4:** Users, Journeys, and Requirements — user stories become the features you specify here.

Later chapters (where this is used next):

- **Chapter 7:** Prompting — your specification becomes the context for structured AI prompts.
- **Chapter 8:** Diff review — acceptance criteria are the checklist you use to review AI-generated code.
- **Chapter 12:** Testing — acceptance criteria become test cases.
- **Chapter 16:** Capstone — the specification and criteria from this chapter are part of the capstone deliverables.

Evidence notes: Claims trace to B-02/B-03/B-04 (Atlassian, Visual Paradigm acceptance criteria), Constitution §3 and §4. All scenarios are hypothetical.

Time estimate: Approximately 120 minutes total (70 minutes reading, 50 minutes hands-on practice).

PART 3

3 Directing the AI

Tools, context, and the diff discipline

06 Tools, Environments, and Your First Win

07 Prompting, Context, and AI Collaboration

08 The Diff Discipline: Reviewing AI-Generated Code

PART 3 · DIRECTING THE AI

06

Tools, Environments, and Your First Win

01 Opening problem — the trap of tool preparation

● SLOP DETECTOR — THE MISCONCEPTION

The scenario

Wanjiku is ready to build Mifugo Market. Before she writes a single line of code, she spends two weeks researching which AI coding editor is "best," which database to use, which hosting platform to choose, and which framework to learn. She installs three different tools, creates accounts on four platforms, and watches hours of tutorial videos. She has not yet built a single page.

When she finally opens an AI editor, she is overwhelmed by options and unsure where to start. The tool research became a form of procrastination — and she still has nothing live.

The misconception

MISCONCEPTION	WHY IT IS DANGEROUS	THE CORRECTION
"I need to research and install the perfect stack before I can start."	Tool choice matters less than starting. Time spent comparing tools is time not spent validating the problem or building a page.	Pick one accessible, well-supported stack and ship a small page first. You can change tools later; you cannot recover lost time.
"I need to master the terminal before I can deploy."	Deployment and version control are hard to teach without some terminal comfort, but you only need a few commands to get started.	Learn the minimum Git commands (init, add, commit, push) and expand as needed.

02 Why this topic matters in real software projects

Every application needs a place to live and a way to track changes. Until you have:

- an editor where you can write and review code,
- version control (Git) so you can undo mistakes,
- and a live URL that real people can visit,

you do not have a software project — you have a file on your laptop.

This chapter gives you your **early win**: by the end, you will have a static landing page for Mifugo Market, tracked in Git, and live on the internet at a public URL. Constitution §5 names this the early-win principle — confidence built before the harder material.

It also establishes habits you will use in every later chapter: committing before and after AI sessions, keeping secrets out of Git, and deploying through a platform rather than from your laptop.

03 Learning objectives

WHAT YOU WILL BE ABLE TO DO

By the end of this chapter, the reader can:

- Install an AI coding editor (principle: AI chat, inline editing, and diff review; tool-specific steps labeled "verify current").
- Create a Git repository and make a first commit.
- Build a static one-page site for Mifugo Market (HTML/CSS).
- Deploy it to a PaaS with a custom domain or a free subdomain.
- Set one environment variable in the platform dashboard.
- Explain what `.gitignore` excludes and why `.env` is never committed.

04 Prerequisites and what to skip

BEFORE YOU START

Prerequisites

Chapters 1–5. You should have a validated problem statement and a product idea (Mifugo Market or your own) from Chapters 3–5.

What you can skip

Deep terminal mastery. You only need the Git commands taught here. Backend, database, and authentication come in Chapters 10–11 — this chapter is static HTML/CSS only.

05 Plain-English explanation

An AI coding editor is your workspace

An **AI coding editor** is a code editor (like Cursor, VS Code with Copilot, or similar) that lets you chat with an AI about your code, ask it to make changes, and review those changes as a diff before accepting them. You do not need to pick the "perfect" one — you need one that can show you a diff before you accept AI changes.

Git is your save-point system

Git records every change to your code as a **commit** — a labeled save point. If an AI edit breaks your page, you can return to the last commit that worked. A **repository** (or repo) is the folder Git tracks, with its full history.

`.gitignore` keeps secrets and clutter out

A `.gitignore` file tells Git which files and folders to never track — for example, `node_modules` (installed packages), `.env` (secrets), and database files. If you commit `.env`, anyone who sees your repository can see your secrets.

`.env` holds local configuration

05 Plain-English explanation (continued)

A `.env` file holds environment variables for development (database paths, secret keys). It is convenient locally but must never be committed to Git. In production, you set these variables in the hosting platform's dashboard instead.

A PaaS hosts your application

A **PaaS** (Platform as a Service) — for example, Render — manages the servers, building, and deployment for you. You connect your GitHub repository, and every time you push code, the platform rebuilds and redeploys your application. You get a public URL (a free subdomain like `mifugomarket.onrender.com`, or a custom domain like `mifugomarket.co.ke`).

05 Plain-English explanation (continued)

Environment variables in production

In production, secrets live in the platform dashboard, not in your code. You set `NODE_ENV=production`, database URLs, and signing keys there, and your application reads them at runtime.

06 Mental model — and where it breaks down

Analogy: a workshop with a time machine

Think of your editor as a workbench, Git as a time machine that lets you return to any earlier state of your project, and the PaaS as a storefront window that anyone on the street can look into. You build at the workbench, save snapshots with the time machine, and display the result in the window.

Where the analogy breaks down

06 Mental model — and where it breaks down (continued)

Unlike a physical workshop, the PaaS storefront updates automatically every time you save a new snapshot to GitHub. There is no separate "publish" button — your `git push` is the publish button. This is convenient, but it also means a mistake is live immediately unless you check it first (staging previews are covered in Chapter 14).

07 Technical explanation with correct terminology

Each term is defined at first use below and listed in the chapter glossary (Section 18).

07 Technical explanation with correct terminology

(continued)

- **AI coding editor** — a code editor with AI chat, inline editing, and diff review (for example, Cursor).
- **Context file** — a file (CLAUDE.md, GEMINI.md, or AGENTS.md) that tells the AI about your project conventions, stack, and rules.
- **Git** — version-control software that records every change as a commit. H-01
- **Repository (repo)** — the folder Git tracks, with its full history.
- **Commit** — a labeled save point with a message describing what changed.
- **Push** — sending your commits to a remote repository such as GitHub.
- **.gitignore** — a file listing folders and files Git must never track.
- **.env** — a local file holding development environment variables; never committed.
- **.env.example** — a template with placeholder values (not real secrets), committed to the repo.
- **Static site** — a website made of HTML, CSS, and JavaScript files served exactly as stored, with no backend.

07 Technical explanation with correct terminology

(continued) (continued)

- **PaaS** — a hosting platform that manages servers, building, and deployment (for example, Render).
- **Deploy** — publishing your application to a public URL.
- **Custom domain** — a domain name you own (for example, `mifugomarket.co.ke`).
- **Free subdomain** — a URL provided by the platform (for example, `mifugomarket.onrender.com`).
- **Environment variable** — a value set outside the code that the application reads at runtime.
- **Build command** — the command the platform runs to install dependencies (for example, `npm install`).

07 Technical explanation with correct terminology

(continued)

- **Start command** — the command the platform runs to start your application.

07 Technical explanation with correct terminology

(continued)

[security] — `.gitignore`, `.env`, Environment variable (protect secrets)

08 Concrete example — Mifugo Market landing page

IN PRACTICE

Wanjiku creates a static landing page for Mifugo Market. It shows:

08 Concrete example — Mifugo Market landing page (continued)

- The business name and tagline ("Fresh produce, delivered to your shop in Ruiru and Nairobi").
- A short list of produce (sukuma wiki, tomatoes, avocados, mangoes) with prices.

08 Concrete example — Mifugo Market landing page (continued)

- A contact placeholder (phone number or WhatsApp link).
- A simple, mobile-friendly layout.

08 Concrete example — Mifugo Market landing page (continued)

She does not build a backend, a database, or user accounts yet. This is version zero: one page, live on the internet, proving the idea can reach customers. She will add the backend and ordering in Chapters 9–11.

09 Step-by-step demonstration — your first win

STEP BY STEP

Phase A — Set up the AI coding editor

1. Install an AI coding editor (for example, Cursor). [Tool-specific — verify current; tool_comparison_framework.md]
2. Create a new folder for your project: `mifugo-market/`.
3. Create a context file at the project root, for example `AGENTS.md`:

09 Step-by-step demonstration — your first win

(continued)

```text
# Mifugo Market
A fresh-produce ordering platform for small shops and households
in Nairobi.
Stack: static HTML/CSS landing page now; Node.js/Express backend
later.
Rules:
```

09 Step-by-step demonstration — your first win

(continued)

```text
- Never hardcode secrets.
- Commit before and after every AI session.
- Review every diff before accepting.
```

09 Step-by-step demonstration — your first win

(continued)

1. Open the folder in your editor and start a chat with the AI.

09 Step-by-step demonstration — your first win

(continued)

[verify current] — Editor installation steps, keyboard shortcuts, and context-file names (CLAUDE.md / GEMINI.md / AGENTS.md) vary by tool and change over time. Confirm against your editor's current documentation and the companion website.

09 Step-by-step demonstration — your first win

(continued) (continued)

[tool_comparison_framework.md, T]

09 Step-by-step demonstration — your first win

(continued)

Phase B — Generate the static landing page

Ask your AI assistant:

09 Step-by-step demonstration — your first win

(continued)

> *"Create a simple, mobile-friendly HTML landing page for Mifugo Market, a fresh-produce ordering platform in Nairobi. Include the business name, tagline, a list of produce items with prices, and a contact placeholder. Use semantic HTML and basic CSS. Return a single HTML file with embedded CSS."*

Review the output before accepting. Save it as `index.html`.

Open `index.html` in your browser to confirm it loads and looks right on your screen.

Phase C — Initialize Git and make your first commit

In your terminal (inside the editor or a separate terminal window):

09 Step-by-step demonstration — your first win

(continued) (continued)

```bash
# Initialize a Git repository
git init

# Create .gitignore before you commit anything
cat > .gitignore << 'EOF'
node_modules/
.env
.env.local
.DS_Store
EOF

# Stage your files
git add index.html .gitignore AGENTS.md

# Make your first commit
git commit -m "feat: initial static landing page"
```

You now have a save point. If the next AI edit breaks the page, you can return here.

09 Step-by-step demonstration — your first win

(continued)

[verify current] — Git commands are stable (git-scm.com H-01), but terminal behavior and editor-integrated Git UIs differ; check the companion website for editor-specific steps.

Phase D — Create a GitHub repository and push

09 Step-by-step demonstration — your first win

(continued) (continued)

1. Create a new, empty repository on GitHub (do not initialize with a README).
2. Copy the commands GitHub shows for adding a remote and pushing:

09 Step-by-step demonstration — your first win

(continued) (continued)

```bash
git remote add origin https://github.com/YOUR_USERNAME/mifugo-market.git
git branch -M main
git push -u origin main
```

Your code is now backed up on GitHub.

Phase E — Deploy to Render

09 Step-by-step demonstration — your first win

(continued) (continued)

Render is used here because it offers a usable free tier that does not require a credit card to start, and it supports static sites and Node.js applications with environment variables set in its dashboard. [Tool selection per Constitution §8; tool_comparison_framework.md]

Deploy steps (Render-specific — verify current):

09 Step-by-step demonstration — your first win

(continued)

1. Push your committed code to GitHub (done in Phase D).
2. Log into Render and connect your GitHub repository.
3. Render detects a static site. Choose "Static Site" as the service type.
4. Set the **build command** if needed (for a pure HTML site, this may be empty).
5. Choose a free `onrender.com` subdomain, or add a custom domain (for example, `mifugomarket.co.ke`).
6. If using a custom domain, configure DNS through your domain registrar: add an A record or CNAME pointing to Render.
7. Trigger the first deployment and watch the build log.
8. Visit the live URL and confirm the landing page loads.

[verify current] — Render's dashboard layout, free-tier limits, and domain configuration steps change over time. Treat these steps as illustrative and confirm them against Render's official documentation and the companion website before following them. H-02

Phase F — Set one environment variable

[verify current] — Static HTML files cannot directly read platform environment variables without a build step that replaces placeholders like `[[SITE_NAME]]`. This example demonstrates the principle of separating configuration from code; Render's actual method for passing environment variables to a static site may require a build process. Confirm Render's current behavior in Render's official documentation and the companion website before relying on it. H-02

Even a static page can use an environment variable to demonstrate the principle. In your HTML, add a small script that reads a variable:

09 Step-by-step demonstration — your first win

(continued)

```HTML
<script>
  const siteName = "[[SITE_NAME]]";
  document.getElementById("site-title").textContent = siteName;
</script>
```

Then in Render's dashboard, add an environment variable:

- **Key:** `SITE_NAME`
- **Value:** `Mifugo Market`

Redeploy. Confirm the page title reflects the variable — or note that this step may not work as written until Render's current static-site environment-variable behavior is confirmed.

Phase G — Add a second commit

Make a small change (for example, update the tagline), review the diff, accept it, and commit:

```BASH
git add index.html
git commit -m "feat: update tagline with SITE_NAME variable"
git push origin main
```

Render redeploys automatically from the push.

10 AI collaboration example (prompt format P1–P10)

AI COLLABORATION — PROMPT FORMAT P1–P10

PROMPT FORMAT **A reusable prompt you can copy**

P1 **P1 · PROBLEM IT SOLVES**

Generate a static landing page and configure it for deployment without exposing secrets.

P2 **P2 · WHEN TO USE IT**

Chapter 6, when creating your first static page.

P3 **P3 · REQUIRED INPUTS**

Your project folder, the business name and tagline, and the list of produce items to display.

P4 **P4 · THE PROMPT**

```text
Create a static HTML landing page for Mifugo Market, a
fresh-produce ordering platform in Nairobi.

Requirements:
- Semantic HTML with embedded CSS (single file,
index.html).
- Mobile-friendly layout.
- Show: business name, tagline, produce list with prices,
contact placeholder.
- Do not include any secrets, API keys, or real
credentials.
- Return the file as a code block with language tag html.
- Reference the project conventions in AGENTS.md.
```

P5 **P5 · WHAT A GOOD RESPONSE CONTAINS**

A single HTML file with embedded CSS; mobile-friendly layout; all required content; no secrets; clearly marked code block.

10 AI collaboration example (prompt format P1–P10) (continued)

P6 **P6 · WHAT THE READER MUST INSPECT**
Open the file in a browser; confirm it loads; confirm no API keys or secrets appear; confirm the layout works on a narrow screen.

P7 **P7 · WEAK OR INCOMPLETE RESPONSE**
Includes placeholder API keys; suggests connecting to a backend; returns multiple files without explanation; does not mention mobile layout.

P8 **P8 · FOLLOW-UP WHEN INADEQUATE**
"Remove the API key placeholder," "Make it a single HTML file," "Show me how it looks on a phone screen."

P9 **P9 · SAFETY/SECURITY CONSIDERATIONS**
Never let the AI add real credentials, even as "placeholders" to be filled in later. Use `[PLACEHOLDER]` only, and never commit anything that looks like a secret.

P10 **P10 · VERIFICATION STEP**
Open `index.html` in your browser; confirm the page loads and displays the produce list; confirm `git ls-files | grep .env` returns nothing (no env file tracked).

11 Human decisions and responsibilities (non-delegable judgment)

● YOUR DECISION — NOT THE AI'S

11 Human decisions and responsibilities (non-delegable judgment) (continued)

- **Which editor and platform to use.** You choose based on your needs; the AI can suggest but not decide for you.
- **What goes in `.gitignore`.** You decide which files must never be tracked (`.env`, secrets, database files).
- **When to deploy.** You judge whether the page is ready to be public.

11 Human decisions and responsibilities (non-delegable judgment) (continued)

- **Whether to use a custom domain or free subdomain.** You own the domain decision and any cost.

11 Human decisions and responsibilities (non-delegable judgment) (continued)

[Human decision-point box] Before your first deploy, answer: Is `.env` in `.gitignore`? Have I tested the page locally? Do I know the live URL? If any answer is no, do not deploy yet.

12 Inspect and verify AI output (first-win checklist)

QUALITY CHECK

Before accepting AI-generated code or deploying:

12 Inspect and verify AI output (first-win checklist)

(continued)

- ✓ [ ] The page opens in a browser without errors.
- ✓ [ ] The layout works on a narrow screen (mobile-friendly).
- ✓ [ ] No secrets, API keys, or credentials appear in the code.
- ✓ [ ] `.gitignore` excludes `node_modules`, `.env`, and `.DS_Store`.

12 Inspect and verify AI output (first-win checklist)

(continued)

- ✓ [ ] The commit history has at least one meaningful commit before deploy.
- ✓ [ ] You can state the live URL.

12 Inspect and verify AI output (first-win checklist)

(continued)

[Related to H-01, H-02, H-03, Q-06]

13 Failure modes and recovery

WHEN IT GOES WRONG

13 Failure modes and recovery (continued)

FAILURE	SYMPTOM	LIKELY CAUSE	RECOVERY
Page does not load after deploy	Blank screen or 404	Wrong build/start command; file not committed; wrong branch deployed	Check Render build log; confirm `index.html` is committed and on `main`

13 Failure modes and recovery (continued)

FAILURE	SYMPTOM	LIKELY CAUSE	RECOVERY
Secret committed to Git	`.env` or API key appears in repository history	`.env` not added to `.gitignore` before first commit	Rotate the key immediately; remove from code; treat as compromised [G-05, S-01]
Cannot push to GitHub	Authentication error	Not logged into Git; wrong remote URL	Authenticate with GitHub; check `git remote -v`

13 Failure modes and recovery (continued)

FAILURE	SYMPTOM	LIKELY CAUSE	RECOVERY
Deploy fails silently	No error, but old version showing	Push went to wrong branch; Render not connected to `main`	Confirm branch is `main`; check Render dashboard for deploy status
`.env` read as undefined in production	Variable works locally but not live	Variable set in `.env` locally, not in platform dashboard	Set the variable in Render's dashboard; redeploy

13 Failure modes and recovery (continued)

All scenarios in this section are hypothetical and labeled as such.

14 Professional standard — quality threshold before going live

PRO STANDARD

Your first deployment is ready only when:

- **Version-control discipline (Q-06):** Commit before and after AI sessions; `.gitignore` excludes `node_modules`, `.env`, build output. H-01
- **Secrets excluded (S-01):** No hardcoded secrets; `.env` in `.gitignore`; production secrets in the platform dashboard only. G-05
- **Locally tested:** The page opens correctly in your browser before you push. [S-08 spirit]
- **Live URL reachable:** The deployed URL loads over HTTPS. [S-02]

15 Hands-on exercise (L3 — Application)

BUILD IT

Task: Deploy your own static landing page (Mifugo Market or your own idea) to a public URL.

Constraints:

- Commit all work with Git before deploying, and push to GitHub first.
- Do not commit `.env` or any secrets.
- Set one environment variable in the platform dashboard.
- Test the page locally before deploying.

Deliverable: A live public URL (free subdomain or custom domain) showing your deployed landing page.

Success criteria:

15 Hands-on exercise (L3 — Application) (continued)

- The URL loads in a browser over HTTPS.
- The page displays your business name and tagline.
- `.env` is not tracked by Git (`git ls-files | grep .env` returns nothing).
- At least one environment variable is set in the platform dashboard.

Assessment level: L3 (Application — performs a procedure with guidance)

Time estimate: 60–90 minutes.

16 Build challenge (L3 — Application) with acceptance criteria

● PUSH FURTHER

Task: Make a small change to your landing page, commit it, push, and confirm the platform redeploys automatically. Document the deploy URL and the environment variable you set.

Constraints:

- Do not ask the AI to regenerate the whole page.
- Make the smallest possible change (for example, update the tagline or add one produce item).
- Commit before and after the AI edit.

Acceptance criteria (Given–When–Then):

16 Build challenge (L3 — Application) with acceptance criteria (continued)

- **Given** a change is committed and pushed to GitHub, **When** the platform detects the push, **Then** it redeploys automatically and the new content appears at the live URL.
- **Given** an environment variable is set in the platform dashboard, **When** the page loads, **Then** it reflects the variable value.
- **Given** the Git history is reviewed, **Then** there are at least three commits (initial page, context file, and the change) with meaningful messages.

Deliverable: A note with the live URL, the environment variable name, and the number of commits in your history.

Assessment level: L3 (Application — performs a procedure with guidance)

Time estimate: 45–60 minutes.

17 Key takeaways

KEY TAKEAWAYS

17 Key takeaways (continued)

- Pick one accessible stack and ship a small page first — tool research is not progress.
- Git is your safety net: commit before and after every AI session.
- `.env` and secrets never go in Git; production secrets live in the platform dashboard.
- A PaaS (Render) deploys automatically from GitHub — your `git push` is the publish button.
- Test locally before you deploy, and confirm the live URL loads over HTTPS.
- Your live URL, Git repo, and `.gitignore` are the foundation every later chapter builds on.

18 Glossary of new terms

TERMS INTRODUCED IN THIS CHAPTER

18 Glossary of new terms (continued)

TERM	PLAIN-LANGUAGE DEFINITION	EXAMPLE	WHY IT MATTERS	TAG
AI coding editor	A code editor with AI chat, inline editing, and diff review.	Cursor	Lets you direct AI and review changes before accepting.	
Context file	A file telling the AI about your project conventions, stack, and rules.	`AGENTS.md`	Keeps AI output consistent with your project.	
Commit	A labeled save point in Git with a message describing what changed.	`git commit -m "feat: initial landing page"`	Lets you return to a known-good state.	

18 Glossary of new terms (continued) (continued)

TERM	PLAIN-LANGUAGE DEFINITION	EXAMPLE	WHY IT MATTERS	TAG
Push	Sending commits to a remote repository such as GitHub.	`git push origin main`	Backs up work and triggers PaaS redeploys.	
`.gitignore`	A file listing folders and files Git must never track.	`node_modules/` , `.env` , `.DS_Store`	Keeps secrets and clutter out of your repository.	[security]

18 Glossary of new terms (continued) (continued)

TERM	PLAIN-LANGUAGE DEFINITION	EXAMPLE	WHY IT MATTERS	TAG
`.env`	A local file holding development environment variables.	`SESSION_SECRET=dev-value`	Convenient locally, but must never be committed to Git.	[security]

18 Glossary of new terms (continued)

TERM	PLAIN-LANGUAGE DEFINITION	EXAMPLE	WHY IT MATTERS	TAG
`.env.example`	A template with placeholder values (not real secrets).	`SESSION_SECRET=[CHANGE_ME]`	Shows others which variables to set without exposing secrets.	[security]

18 Glossary of new terms (continued) (continued)

TERM	PLAIN-LANGUAGE DEFINITION	EXAMPLE	WHY IT MATTERS	TAG
Static site	A website of HTML/CSS/JS files served exactly as stored.	`index.html`	Simple, cheap to host, and a good first win.	
PaaS	A hosting platform managing servers, building, and deployment.	Render	Deploys your app without you managing infrastructure.	

18 Glossary of new terms (continued) (continued)

TERM	PLAIN-LANGUAGE DEFINITION	EXAMPLE	WHY IT MATTERS	TAG
Deploy	Publishing your application to a public URL.	`git push` → Render rebuilds	Makes your app reachable by anyone with the URL.	
Custom domain	A domain name you own.	`mifugomarket.co.ke`	Professional, memorable address.	

18 Glossary of new terms (continued) (continued)

TERM	PLAIN-LANGUAGE DEFINITION	EXAMPLE	WHY IT MATTERS	TAG
Free subdomain	A platform-provided URL.	`mifugomarket.onrender.com`	Free, no domain purchase needed to start.	

18 Glossary of new terms (continued)

TERM	PLAIN-LANGUAGE DEFINITION	EXAMPLE	WHY IT MATTERS	TAG
Environment variable	A value set outside the code that the app reads at runtime.	`SITE_NAME=Mifugo Market`	Separates configuration from code; keeps secrets out of Git.	[security]
Build command	The command a platform runs to install dependencies.	`npm install`	Wrong command means the app will not build.	
Start command	The command a platform runs to start your application.	`node server.js`	Must match how your app actually starts.	

19 Knowledge-check questions

KNOWLEDGE CHECK

Attempt these before looking at the answers below.

Q1. Why should `.env` never be committed to Git?

Q2. What does `git push` do in the context of a PaaS deployment?

Q3. Name three things `.gitignore` should exclude for a web project.

Q4. True or false: You must master the terminal before you can deploy to a PaaS.

Q5. What is a context file, and what does it do?

[PAGE BREAK — ANSWER KEY]

Answers

19 Knowledge-check questions (continued)

A1. Because anyone who sees your repository (including future AI sessions or collaborators) could copy your secrets, and secrets committed to Git are there forever in the history. [S-01, G-05]

A2. It sends your commits to GitHub; once connected to a PaaS like Render, the push triggers a rebuild and redeploy of your application. H-02

A3. Any three of: `node_modules/`, `.env`, `.env.local`, `.DS_Store`, database files (`*.db`), build output (`dist/`, `build/`). [Q-06]

A4. False. You only need a few Git commands (init, add, commit, push); you can expand terminal comfort over time.

19 Knowledge-check questions (continued)

A5. A context file (for example, `AGENTS.md`, `CLAUDE.md`, or `GEMINI.md`) tells the AI about your project conventions, stack, and rules, so its output stays consistent with your project.

20 Links to earlier and later chapters

WHERE THIS SITS IN THE BOOK

Earlier chapters (prerequisites):

- **Chapter 3:** Finding a Problem Worth Solving — your validated problem gives you something worth putting on a landing page.
- **Chapter 4:** Users, Journeys, and Requirements — your primary user informs what the landing page should say.
- **Chapter 5:** Specification and Acceptance Criteria — your page is the first feature with its own acceptance criteria.

Later chapters (where this is used next):

20 Links to earlier and later chapters (continued)

- **Chapter 7:** Prompting, Context, and AI Collaboration — your context file from this chapter grows as the project grows.
- **Chapter 8:** The Diff Discipline — the commits you make here are what you review as diffs.
- **Chapter 10:** Backend, Data, and APIs — your static page becomes the frontend that calls backend endpoints.
- **Chapter 14:** Version Control, Deployment, and Operations — the Git and deploy habits from this chapter are deepened with branches, staging, and rollback.

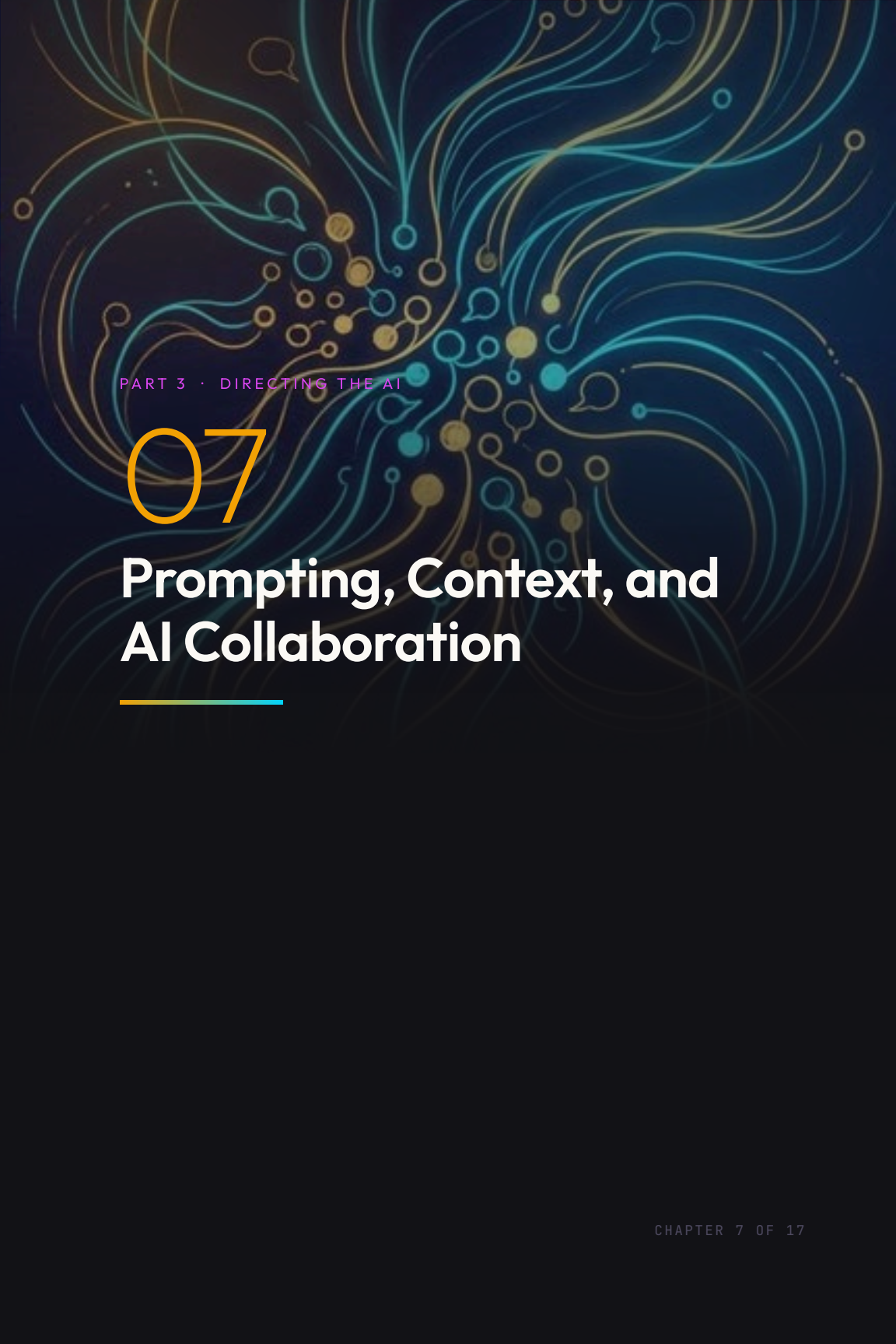
20 Links to earlier and later chapters (continued)

Companion website: Current editor, PaaS, and domain pricing; Kenyan payment accessibility; updated deploy steps — flagged "verify current — see companion website" because platform-specific instructions date quickly. [T]

Evidence notes: Claims trace to H-01 (Git), H-02 (PaaS deployment), H-03 (environment variables), tool_comparison_framework.md (tool selection and regional notes), Q-06 (version-control discipline). All scenarios are hypothetical.

20 Links to earlier and later chapters (continued)

Time estimate: Approximately 180 minutes total (90 minutes reading, 90 minutes hands-on practice).



PART 3 · DIRECTING THE AI

07

Prompting, Context, and AI Collaboration

01 Opening problem — the one-line prompt trap

SLOP DETECTOR — THE MISCONCEPTION

The scenario

Wanjiku has her Mifugo Market landing page live. She wants to add a product catalog so customers can see what produce is in stock. She opens her AI coding assistant and types: "Add an API that lists products."

The AI generates a backend route, a database connection, and a frontend fetch call — all in one response. It looks impressive, but it hardcodes a database password, assumes a database that does not exist yet, and writes frontend code that does not match her static HTML structure. Wanjiku accepts it anyway because it "runs." Two days later, nothing works and she cannot tell which part is broken.

She used a one-line prompt, expected the AI to remember her project, and dumped too much into one request. That is how AI-generated slop begins.

The misconception

MISCONCEPTION	WHY IT IS DANGEROUS	THE CORRECTION
"I just need to prompt better."	Prompting is necessary but insufficient. You must also understand the code, define requirements, and verify output.	Prompting + context + review. The human remains the director. D-01 · D-02
"The AI remembers everything from last session."	AI models are stateless; they forget previous conversations once the context window fills or the session ends.	Maintain a context file and re-state constraints each session.
"Paste the whole codebase and ask for a feature."	Overloading the context window causes context rot and lost-in-the-middle; the AI misses relevant details.	Break features into small chunks and reference specific files.

02 Why this topic matters in real software projects

AI coding assistants can generate code fast, but they generate *plausible* code, not necessarily *correct* code. The difference between a working prototype and professional software is whether the human directs the AI with clear context, constraints, and review — or copies and pastes until something "mostly works."

This chapter teaches you to prompt deliberately: structure your prompts, manage context, break work into AI-sized chunks, and know when to start a fresh session. These skills are used in every chapter that follows, because every chapter asks you to direct the AI rather than delegate to it.

03 Learning objectives

WHAT YOU WILL BE ABLE TO DO

By the end of this chapter, the reader can:

- Write a prompt with role, context, constraints, and desired output format.
- Maintain a context file for a project (CLAUDE.md / GEMINI.md / AGENTS.md).
- Break a feature into AI-sized chunks small enough to implement reliably.
- State three context-window limitations: statelessness, context rot, and lost-in-the-middle.
- Decide when to start a fresh AI session versus continue an existing one.

04 Prerequisites and what to skip

BEFORE YOU START

Prerequisites

Chapters 1–6. You should have an AI coding editor installed, a Git repository, a context file (started in Chapter 6), and a static landing page.

What you can skip

No specific model benchmarking. The book does not rank current AI models (Constitution §6 exclusion). Model names, context window sizes, and pricing change quarterly — treat them as "verify current" on the companion website.

05 Plain-English explanation

A good prompt has four parts

Every important prompt should include:

- **Role:** who the AI should act as (for example, "You are a careful frontend developer").
- **Context:** what the project is, which files are involved, and what has already been decided.
- **Constraints:** what the AI must not do (for example, "do not hardcode API keys," "do not change existing styles").
- **Output format:** what you want back (for example, "return a unified diff," "return only the changed file").

Context is more important than model size

Research shows that how you structure and maintain context matters more than the raw size of the model's context window D-02. A well-maintained context file (AGENTS.md / CLAUDE.md / GEMINI.md) that describes your project, conventions, and rules produces better results than pasting everything into the chat.

05 Plain-English explanation (continued)

Three context-window limitations

- **Statelessness:** The AI does not remember past sessions unless you remind it. Each new chat starts fresh.
- **Context rot:** As a conversation gets long, the AI loses track of earlier constraints (for example, "never hardcode keys" may be forgotten after twenty messages) D-01.
- **Lost-in-the-middle:** The AI pays less attention to information buried in the middle of a long prompt; the beginning and end get more weight D-01.

Chunking keeps the AI focused

05 Plain-English explanation (continued)

Break a feature into chunks small enough for the AI to implement reliably: one route, one form, one validation rule. Do not ask the AI to "build the whole app" in one prompt. Small chunks are easier to review, test, and roll back.

06 Mental model — and where it breaks down

Analogy: a junior assistant with a short memory

Think of the AI as a brilliant junior assistant who has read every programming book but forgets everything you told them yesterday. You must hand them a folder (the context file) at the start of each meeting, point to the specific page you want them to edit, and check their work before they touch anything else.

Where the analogy breaks down

Unlike a junior assistant, the AI cannot take initiative, notice a missing requirement on its own, or refuse an unsafe request reliably. You must supply all judgment and accept full responsibility.

07 Technical explanation with correct terminology

Each term is defined at first use below and listed in the chapter glossary (Section 18).

- **Prompt** — the instructions you give an AI coding assistant.
- **Role** — the persona or stance you ask the AI to adopt.
- **Context** — project background, relevant files, and prior decisions the AI needs to know.
- **Constraints** — rules the AI must follow (what it must not do).
- **Output format** — how you want the AI's response structured (diff, code block, list).
- **Context window** — the amount of text the AI can process at once.
- **Statelessness** — the AI does not retain memory across separate sessions.
- **Context rot** — loss of important details as a conversation grows long.
- **Lost-in-the-middle** — the AI overlooks information in the middle of a long prompt.
- **Context file** — a project file (AGENTS.md / CLAUDE.md / GEMINI.md) describing conventions, stack, and rules.
- **Chunking** — breaking a feature into small, independently implementable pieces.
- **AI-sized chunk** — a piece of work small enough for the AI to implement and review reliably.

[AI] — Context window, Statelessness, Context rot, Lost-in-the-middle, Chunking (AI-specific limitations)

08 Concrete example — adding a product catalog to Mifugo Market

IN PRACTICE

Wanjiku wants a backend endpoint that returns her produce list as JSON, and a frontend that displays it. Instead of one prompt ("add a product catalog"), she breaks it into chunks:

Chunk 1 — Context: She reviews her AGENTS.md context file and confirms the stack (static HTML frontend; Node.js/Express backend coming next).

Chunk 2 — Backend route: One prompt to add a `GET /api/products` route returning hardcoded product JSON, with a constraint to use environment variables for any configuration.

Chunk 3 — Frontend fetch: One prompt to add JavaScript that calls the endpoint and displays products, with a constraint to handle errors gracefully.

Chunk 4 — Review: She reviews each diff before accepting (Chapter 8 discipline), tests locally, then commits.

Each chunk is small enough that she can verify the output completely.

09 Step-by-step demonstration — prompting for a catalog feature

STEP BY STEP

Step 1: Review your context file

Open AGENTS.md (or CLAUDE.md / GEMINI.md). Confirm it describes your project, stack, and rules. Add any missing details:

09 Step-by-step demonstration — prompting for a catalog feature (continued)

```
TEXT
# Mifugo Market — context
Stack: static HTML/CSS frontend; Node.js/Express backend
(planned).
Rules: never hardcode secrets; review every diff; commit
before/after AI sessions.
Current state: landing page live at mifugomarket.onrender.com.
Next: product catalog endpoint GET /api/products.
```

09 Step-by-step demonstration — prompting for a catalog feature (continued)

Step 2: Write a structured prompt for one chunk

Instead of "add a product catalog," prompt for one piece:

09 Step-by-step demonstration — prompting for a catalog feature (continued)

```
TEXT
Role: You are a careful backend developer.
Context: Mifugo Market is a fresh-produce ordering platform in
Nairobi. We are adding a product catalog. The frontend is static
HTML/CSS; the backend will be Node.js/Express. See AGENTS.md for
project conventions.

Task: Add a GET /api/products route that returns a JSON list of
produce items (name, price, stock_status). Use hardcoded data
for now; no database yet.

Constraints:
```

09 Step-by-step demonstration — prompting for a catalog feature (continued) (continued)

```
TEXT
- Do not hardcode any secrets or API keys.
- Return a unified diff showing only the changed files.
- Include error handling: if something goes wrong, return HTTP
500 with a JSON error message.
- Do not modify the frontend in this step.

Output format: Unified diff with file paths.
```

09 Step-by-step demonstration — prompting for a catalog feature (continued) (continued)

Step 3: Review the diff

Open the diff the AI generates. Check:

- [ ] No secrets or hardcoded credentials.
- [ ] The route matches your project structure.
- [ ] Error handling is present.
- [ ] No unrelated files were changed.

Accept only what passes review.

Step 4: Test locally

09 Step-by-step demonstration — prompting for a catalog feature (continued)

[Important] The backend endpoint `GET /api/products` cannot be tested yet — the backend server itself is built in Chapter 10. At this point you have a static landing page only. For now, confirm the diff looks right (the route exists, returns JSON, has error handling, no secrets) and accept it if it does. You will test the endpoint locally in Chapter 10.

Step 5: Commit and move to the next chunk

```bash
git add <changed-files>
git commit -m "feat: add GET /api/products route"
```

Then prompt for the next chunk (frontend display).

[verify current] — Model-specific context window sizes, pricing, and rate limits change quarterly. The principles (context file, chunking, diff review) are stable; the numbers are not. Check the companion website for current figures. [R-06, T]

10 AI collaboration example (prompt format P1–P10)

AI COLLABORATION — PROMPT FORMAT P1–P10

PROMPT FORMAT **A reusable prompt you can copy**

P1 **P1 · PROBLEM IT SOLVES**

Add a feature to an existing project without introducing slop or breaking existing code.

10 AI collaboration example (prompt format P1–P10) (continued)

P2 **P2 · WHEN TO USE IT**
Chapter 7 and every chapter after, whenever you direct the AI to modify code.

P3 **P3 · REQUIRED INPUTS**
Your context file, the specific file(s) to change, the feature description, and your constraints.

10 AI collaboration example (prompt format P1–P10) (continued)

P4

10 AI collaboration example (prompt format P1–P10) (continued)

P4 · THE PROMPT

10 AI collaboration example (prompt format P1–P10) (continued)

```
TEXT
Role: You are a careful [frontend/backend] developer.
Context: [PROJECT_DESCRIPTION]. Relevant file: [FILE_PATH].
Project conventions: see [CONTEXT_FILE].
Task: [FEATURE_DESCRIPTION].
Constraints:
- Do not hardcode any secrets or API keys.
- Do not change files unrelated to this task.
```

10 AI collaboration example (prompt format P1–P10) (continued)

```
TEXT
- Return a unified diff with file paths.
- Include error handling where user input is involved.
Output format: Unified diff.
```

10 AI collaboration example (prompt format P1–P10) (continued)

P5 **P5 · WHAT A GOOD RESPONSE CONTAINS**
A unified diff limited to the relevant files; constraints honored; no secrets; error handling included.

P6 **P6 · WHAT THE READER MUST INSPECT**
Confirm the diff touches only the intended files; confirm no secrets appear; confirm the change matches the task; run tests if available.

10 AI collaboration example (prompt format P1–P10) (continued)

P7 **P7 · WEAK OR INCOMPLETE RESPONSE**
Suggests rewriting the whole app; hardcodes a key; changes unrelated files; ignores constraints.

P8 **P8 · FOLLOW-UP WHEN INADEQUATE**
"Limit the diff to [FILE] only," "Remove the hardcoded key," "Show me the diff for just this route."

P9 **P9 · SAFETY/SECURITY CONSIDERATIONS**
Never paste real customer data or secrets into the prompt. Anonymize examples. Review auth-related changes with extra care (Chapter 8).

P10 **P10 · VERIFICATION STEP**
Run the feature locally; confirm it works; confirm no unrelated behavior broke; commit the change.

11 Human decisions and responsibilities (non-delegable judgment)

YOUR DECISION — NOT THE AI'S

- **What to build.** You define the feature; the AI implements it.
- **How to chunk the work.** You decide what counts as one AI-sized chunk.
- **When to start a fresh session.** You notice context rot and decide whether to continue or start over.
- **Whether to accept the output.** You review the diff and accept or reject; the AI cannot veto you.

11 Human decisions and responsibilities (non-delegable judgment) (continued)

[Human decision-point box] Before every AI session, answer: Do I have a context file? Have I broken this feature into chunks? Do I know which file I am asking the AI to change? If any answer is no, do not prompt yet.

12 Inspect and verify AI output (prompting checklist)

● QUALITY CHECK

Before accepting AI-generated code:

- ✓ [ ] The prompt included role, context, constraints, and output format.
- ✓ [ ] The diff touches only the intended files.
- ✓ [ ] No secrets, API keys, or credentials appear.
- ✓ [ ] Constraints were honored (no unrelated changes).
- ✓ [ ] The output was tested locally before committing.
- ✓ [ ] The context file was updated if conventions changed.

[Related to D-01, D-02, D-03, D-05]

13 Failure modes and recovery

● WHEN IT GOES WRONG

13 Failure modes and recovery (continued)

FAILURE	SYMPTOM	LIKELY CAUSE	RECOVERY
Context rot	AI forgets "no hardcoded keys" after many messages	Long conversation; stateless model	Restate constraints; start a fresh session with context file F-07
Lost-in-the-middle	AI misses a requirement buried in a long prompt	Information in middle of prompt	Put the most important constraints at the start and end

13 Failure modes and recovery (continued)

FAILURE	SYMPTOM	LIKELY CAUSE	RECOVERY
Overloaded prompt	AI builds the wrong feature or skips parts	Asked for too much at once	Break into smaller chunks; one feature per prompt
Whole-app regeneration	AI suggests rewriting everything	Poor prompt framing	Decline; ask for the smallest change to one file
False confidence	AI says the code is correct when it is not	AI review is not a gate	You review the diff; AI review is a second opinion only F-08

13 Failure modes and recovery (continued)

All scenarios in this section are hypothetical and labeled as such.

14 Professional standard — quality threshold before accepting AI output

● PRO STANDARD

You accept AI-generated code only when:

14 Professional standard — quality threshold before accepting AI output (continued)

- The prompt included role, context, constraints, and output format D-03 .
- The diff was reviewed and touches only intended files D-04 .
- No secrets appear in the output [S-01, G-05].

14 Professional standard — quality threshold before accepting AI output (continued)

- The change was tested locally F-01 .
- You can explain what the code does — if you cannot, you do not accept it [M-02].

15 Hands-on exercise (L3 — Application)

BUILD IT

Task: Write three prompts for the same feature (one bad one-line prompt, two progressively better structured prompts) and rank their likely output quality.

Constraints:

- Use the same feature (for example, "add email validation to the order form").
- Do not run the AI unless you have access; simulated ranking is acceptable.
- Label each prompt's weaknesses.

Deliverable: Three prompts with a written quality ranking and reasoning for each.

15 Hands-on exercise (L3 — Application) (continued)

Success criteria:

- Prompt 1 is a one-liner (demonstrates the anti-pattern).
- Prompt 2 adds context and constraints.
- Prompt 3 adds role, output format, and file references.
- Each prompt is followed by a one-sentence quality assessment.

Assessment level: L2 (Understanding — explains in their own words, gives an example)

Time estimate: 40–50 minutes.

16 Build challenge (L3 — Application) with acceptance criteria

● PUSH FURTHER

Task: Maintain your context file and use it to direct one AI chunk for Mifugo Market (or your project).

Constraints:

- Update AGENTS.md / CLAUDE.md / GEMINI.md before prompting.
- Prompt for one AI-sized chunk only (not the whole app).
- Review the diff before accepting.

Acceptance criteria (Given–When–Then):

16 Build challenge (L3 — Application) with acceptance criteria (continued)

- **Given** an updated context file, **When** you prompt for a feature chunk, **Then** the prompt references the context file and specific file paths.
- **Given** the AI returns a diff, **When** you review it, **Then** you either accept, request revision, or reject it with specific reasons.
- **Given** the change is accepted, **When** you commit it, **Then** the commit message describes what changed and the context file is updated if conventions changed.

Deliverable: Updated context file, the prompt used, the diff review notes, and the commit.

Assessment level: L3 (Application — performs a procedure with guidance)

Time estimate: 60–75 minutes.

17 Key takeaways

KEY TAKEAWAYS

- Structure every prompt: role, context, constraints, output format.
- Maintain a context file; it matters more than model size.
- Know the limits: statelessness, context rot, lost-in-the-middle.
- Break work into AI-sized chunks; never ask for the whole app at once.
- You review and accept or reject — the AI does not decide.
- Start a fresh session when context rot sets in.

18 Glossary of new terms

TERMS INTRODUCED IN THIS CHAPTER

18 Glossary of new terms (continued)

TERM	PLAIN-LANGUAGE DEFINITION	EXAMPLE	WHY IT MATTERS	TAG
Prompt	The instructions you give an AI coding assistant.	"Add a GET /api/products route..."	The quality of your prompt shapes the quality of the output.	[AI]
Role	The persona or stance you ask the AI to adopt.	"You are a careful backend developer."	Sets the tone and standard for the AI's output.	[AI]
Context	Project background, relevant files, and prior decisions the AI needs.	AGENTS.md describing your stack and rules.	Without context, the AI guesses and produces slop.	[AI]
Constraints	Rules the AI must follow (what it must not do).	"Do not hardcode API keys."	Keeps the AI from introducing security flaws.	[AI, security]
Output format	How you want the AI's response structured.	"Return a unified diff."	Makes the output reviewable and usable.	[AI]
Context window	The amount of text the AI can process at once.	A few thousand words of conversation and files.	Limits how much you can tell the AI at one time.	[AI]
Statelessness	The AI does not remember past sessions.	A new chat starts with no memory of last week.	You must re-state context each session.	[AI]

18 Glossary of new terms (continued) (continued)

TERM	PLAIN-LANGUAGE DEFINITION	EXAMPLE	WHY IT MATTERS	TAG
Context rot	Loss of important details as a conversation grows long.	The AI forgets "no hardcoded keys" after 20 messages.	Restate constraints or start fresh.	[AI]

18 Glossary of new terms (continued)

TERM	PLAIN-LANGUAGE DEFINITION	EXAMPLE	WHY IT MATTERS	TAG
Lost-in-the-middle	The AI overlooks information in the middle of a long prompt.	A constraint buried in paragraph 5 is ignored.	Put key constraints at start and end.	[AI]
Context file	A project file describing conventions, stack, and rules.	AGENTS.md, CLAUDE.md, GEMINI.md.	Keeps AI output consistent across sessions.	[AI]
Chunking	Breaking a feature into small, independently implementable pieces.	One route per prompt, not the whole app.	Small chunks are easier to review and test.	[AI]
AI-sized chunk	A piece of work small enough for the AI to implement reliably.	"Add the products route" rather than "build the marketplace."	Prevents overloaded prompts and incomplete output.	[AI]

19 Knowledge-check questions

KNOWLEDGE CHECK

Attempt these before looking at the answers below.

Q1. What four parts should every important prompt contain?

Q2. What is context rot, and what do you do about it?

19 Knowledge-check questions (continued)

Q3. True or false: The AI remembers your project from one session to the next.

Q4. What is an AI-sized chunk?

Q5. What is a context file, and which filenames are common?

19 Knowledge-check questions (continued)

[PAGE BREAK — ANSWER KEY]

Answers

A1. Role, context, constraints, and output format.

19 Knowledge-check questions (continued)

A2. Context rot is the loss of important details (such as "never hardcode keys") as a conversation grows long. You restate constraints periodically or start a fresh session with your context file.

19 Knowledge-check questions (continued)

A3. False. AI models are stateless; they do not retain memory across separate sessions unless you remind them via a context file.

19 Knowledge-check questions (continued)

A4. A piece of work small enough for the AI to implement and review reliably — for example, one route or one form, not a whole application.

19 Knowledge-check questions (continued)

A5. A context file (commonly AGENTS.md, CLAUDE.md, or GEMINI.md) describes your project conventions, stack, and rules so the AI's output stays consistent across sessions.

20 Links to earlier and later chapters

WHERE THIS SITS IN THE BOOK

Earlier chapters (prerequisites):

- **Chapter 6:** Tools, Environments, and Your First Win — your context file and editor setup come from here.
- **Chapter 8:** The Diff Discipline — the diff review skills you practice here are applied formally in Chapter 8.

Later chapters (where this is used next):

- **Chapter 9–11:** Frontend, Backend, Access Control — you use structured prompting for every feature you build.
- **Chapter 16:** Capstone — the context file from this chapter is a required capstone deliverable.

Research gap note: R-06 (model context window specifics) has no primary source located for exact current numbers. Context-window principles are established D-01 · D-02; specific sizes and pricing are "verify current" on the companion website. [T]

Evidence notes: Claims trace to D-01 (arXiv AI-assisted coding limitations), D-02 (Packmind/ACE context engineering), D-03/D-04/D-05 (AI coding best practices and failure modes), Constitution §5 and §8. All scenarios are hypothetical.

Time estimate: Approximately 150 minutes total (80 minutes reading, 70 minutes hands-on practice).

PART 3 · DIRECTING THE AI

08

The Diff Discipline: Reviewing AI-Generated Code

01 Opening problem — the "Accept All" habit

SLOP DETECTOR — THE MISCONCEPTION

The scenario

Wanjiku asks her AI assistant to add an "add to cart" button to Mifugo Market. The AI generates a diff touching five files: the cart button, the checkout route, the product page, the stylesheet, and a configuration file she does not recognize. She clicks "Accept All" without reading the changes. The cart works, but so does a hidden change that writes every order to a public log file, and the configuration change disables HTTPS on her production site.

She does not notice until a customer complains that their order details appeared in a Google search.

Accepting AI output without reviewing the diff is how slop reaches production.

The misconception

MISCONCEPTION	WHY IT IS DANGEROUS	THE CORRECTION
"Accept All on every AI suggestion."	You cannot tell good generated code from bad, and you may accept security flaws, broken logic, or unrelated changes.	Review every diff; classify changes as safe, needs review, or reject.
"The AI's own review is good enough."	AI code reviewers produce false positives and negatives; they are not a gate.	A human always owns final approval, especially for auth, crypto, secrets, and deployment config. D-04 · F-08

02 Why this topic matters in real software projects

AI coding assistants generate code that *looks* right but can be subtly wrong: missing validation, hardcoded secrets, wrong logic, or changes to files you did not ask it to touch. Because the AI is confident and fluent, it is easy to trust it blindly.

The diff is the record of exactly what changed — line by line, file by file. Learning to read and review a diff is the skill that separates a builder who directs AI from a prompt-giver who hopes for the best. Constitution §1 names this human responsibility: the reader must evaluate AI output, not delegate judgment to it.

This chapter is the gate that protects every chapter after it. No AI-generated code in this book is accepted without a diff review.

03 Learning objectives

WHAT YOU WILL BE ABLE TO DO

By the end of this chapter, the reader can:

- Open and read a diff (added/removed lines, file paths).
- Classify changes as safe / needs-review / reject.
- Identify three categories of AI errors: logic, security, and data handling.
- Reject or request revision of AI output with specific feedback.
- State which code paths require mandatory human review (auth, crypto, secrets, deployment config).

04 Prerequisites and what to skip

BEFORE YOU START

Prerequisites

Chapters 1–7. You should have an AI coding editor, a Git repository, and experience making commits (Chapter 6) and writing structured prompts (Chapter 7).

What you can skip

Writing production code yourself in this chapter. You read and review AI-generated snippets; you do not implement cryptography or authentication from scratch.

05 Plain-English explanation

A diff is a change log, line by line

A **diff** shows exactly what changed between two versions of a file. Lines starting with `+` were added; lines starting with `-` were removed; unchanged lines show context. Each file changed is listed with its path.

Classify before you accept

For every change in a diff, ask:

- **Safe:** I understand what this does, it matches my intent, and it introduces no risk. (Example: changing button text.)
- **Needs review:** I am not sure what this does, or it touches something important. (Example: a new route handler.)
- **Reject:** This introduces a risk, changes something unrelated, or I cannot explain it. (Example: a hardcoded password or a config change to disable HTTPS.)

Three categories of AI errors

05 Plain-English explanation (continued)

- **Logic errors:** The code runs but does the wrong thing (for example, a discount applied to the wrong items, or an order total miscalculated).
- **Security errors:** The code is insecure (for example, a hardcoded API key, missing authorization, no input validation).
- **Data-handling errors:** The code mishandles data (for example, saving passwords in plaintext, logging sensitive information, no empty-state handling).

Mandatory human review paths

Some code paths always need a human's eyes, no matter how confident the AI seems:

05 Plain-English explanation (continued)

- Authentication and authorization logic
- Cryptographic implementations (password hashing, signing keys)
- Sensitive data handling (personal data, payment data)
- Deployment configuration (environment variables, HTTPS settings, domain setup)

06 Mental model — and where it breaks down

Analogy: an architect's blueprint review

Think of AI-generated code like a blueprint drawn by an assistant. The assistant is fast and can draw many options, but you are the architect who signs off. You check that the load-bearing walls are where they should be, that the exits are safe, and that nothing was added without your approval. You do not build from a blueprint you have not read.

Where the analogy breaks down

06 Mental model — and where it breaks down (continued)

Unlike a blueprint, code can be changed after it is "built," and a bad deploy can be rolled back. But the principle holds: you do not accept changes you have not read, especially in safety-critical areas.

07 Technical explanation with correct terminology

Each term is defined at first use below and listed in the chapter glossary (Section 18).

- **Diff** — a line-by-line record of changes between two versions of a file or set of files.
- **Added line** — a line starting with `+` in a diff; new code being introduced.
- **Removed line** — a line starting with `-` in a diff; code being deleted.
- **Context line** — an unchanged line in a diff, shown for context (usually starting with a space).
- **File path** — the location of a changed file in the repository.
- **Hunk** — a group of related changes within one file in a diff.
- **Unified diff** — a common diff format showing file paths, line numbers, and `+` / `-` lines together.
- **Safe change** — a change you understand, that matches your intent, and introduces no risk.
- **Needs-review change** — a change you are unsure about, or that touches something important.
- **Reject** — a change you decline because it introduces risk, is unrelated, or cannot be explained.
- **Security-critical path** — code handling auth, crypto, secrets, or deployment config; requires mandatory human review.
- **False confidence** — trusting AI output because it looks fluent, even when it is wrong.

[security] — Security-critical path, Secrets, Authentication/authorization, Deployment config

08 Concrete example — reviewing an "add to cart" diff

IN PRACTICE

The AI generates a diff for an "add to cart" feature touching three files:

File 1 — `public/cart.js` (safe):

```
DIFF
+function addToCart(productId, quantity) {
+  cart.push({ id: productId, qty: quantity });
+  updateCartBadge();
+}
```

You understand this. It adds an item to the cart. Classify: **safe**.

File 2 — `routes/cart.js` (needs review):

```
DIFF
+app.post('/cart/add', (req, res) => {
+  const { productId, quantity } = req.body;
+  db.query(`INSERT INTO cart (product_id, qty) VALUES
(${productId}, ${quantity})`);
+  res.json({ ok: true });
+});
```

You notice two problems: no input validation (what if `quantity` is negative?) and raw SQL interpolation (injection risk). Classify: **needs review** — request revision with validation and parameterized queries.

File 3 — `config.js` (reject):

```
DIFF
-const FORCE_HTTPS = true;
+const FORCE_HTTPS = false;
+const ADMIN_KEY = "sk_live_abc123secret";
```

08 Concrete example — reviewing an "add to cart" diff (continued)

This disables HTTPS and hardcodes a secret key. Classify: **reject**. Restore HTTPS, remove the key, and store it in an environment variable instead.

Wanjiku accepts File 1, requests revisions for File 2, and rejects File 3 entirely.

09 Step-by-step demonstration — reviewing an AI diff

STEP BY STEP

Step 1: Open the diff

When the AI suggests changes, your editor shows a diff view. Read it file by file, hunk by hunk. Do not scroll past anything you do not understand.

Step 2: Classify each change

Go through each file and mark it:

CLASSIFICATION	QUESTION TO ASK	ACTION
Safe	Do I understand this? Does it match my intent? Is there no risk?	Accept
Needs review	Am I unsure? Does it touch something important?	Request specific revision
Reject	Does it introduce risk? Change something unrelated? I cannot explain it?	Reject and explain why

Step 3: Check the security-critical paths

For any file touching auth, crypto, secrets, or deployment config, apply extra scrutiny:

09 Step-by-step demonstration — reviewing an AI diff (continued)

- Are secrets read from environment variables, not hardcoded? [S-01, G-05]
- Are passwords hashed, never stored plaintext? [S-05]
- Is HTTPS enforced? [S-02]
- Is authorization checked on every request, not assumed? [S-04]

Step 4: Give specific feedback

If you request a revision, be specific:

09 Step-by-step demonstration — reviewing an AI diff (continued)

- Weak: "This is wrong, fix it."
- Strong: "In routes/cart.js, the SQL query interpolates user input directly. Use parameterized queries instead, and validate that quantity is a positive integer."

Step 5: Accept, reject, or request revision

Accept only what passes review. Commit accepted changes:

```bash
git add <accepted-files>
git commit -m "feat: add to cart with validated input"
```

Step 6: Run tests after accepting

Run your test suite (Chapter 12) to confirm nothing else broke. This catches regression (F-06).

10 AI collaboration example (prompt format P1–P10)

AI COLLABORATION — PROMPT FORMAT P1–P10

PROMPT FORMAT **A reusable prompt you can copy**

P1 · PROBLEM IT SOLVES
Get AI-generated code reviewed and revised before accepting it.

P2 · WHEN TO USE IT
Chapter 8 and every chapter after, whenever the AI generates code.

P3 · REQUIRED INPUTS
The diff the AI produced, the file paths involved, and your constraints (no secrets, validation required, etc.).

P4 · THE PROMPT

TEXT

```
Review this AI-generated diff for Mifugo Market:
[DIFF_CONTENT]

Files changed: [FILE_LIST]

Classify each change as safe / needs-review / reject. For
each needs-review or reject item, explain:
1. What the risk is (logic, security, or data handling).
2. The specific revision needed.

Constraints:
- Do not suggest hardcoding any secrets.
- Do not suggest disabling HTTPS or authentication.
- Return your review as a bulleted list, one item per file.
```

P5 · WHAT A GOOD RESPONSE CONTAINS
A per-file classification; specific risks named; concrete revision steps; no secrets suggested.

10 AI collaboration example (prompt format P1–P10) (continued)

P6 **P6 · WHAT THE READER MUST INSPECT**
Confirm the AI's classification matches your own reading; confirm it did not miss a hardcoded key or missing validation; confirm you still make the final decision.

P7 **P7 · WEAK OR INCOMPLETE RESPONSE**
Says "everything looks fine" without examining; misses a hard-coded key; suggests a whole-app rewrite.

P8 **P8 · FOLLOW-UP WHEN INADEQUATE**
"You missed the hardcoded key in line 12," "Show me the revision for routes/cart.js only," "Why did you classify config.js as safe?"

P9 **P9 · SAFETY/SECURITY CONSIDERATIONS**
Do not paste real secrets into the prompt for review. If a secret is already in the diff, treat it as compromised and rotate it.

P10 **P10 · VERIFICATION STEP**
Apply the AI's suggested revisions, then review the revised diff yourself before accepting.

11 Human decisions and responsibilities (non-delegable judgment)

● YOUR DECISION — NOT THE AI'S

11 Human decisions and responsibilities (non-delegable judgment) (continued)

- **What counts as safe.** You decide; the AI cannot judge your risk tolerance.
- **Whether to accept or reject.** Final approval is always yours.
  D-03 · F-08
- **Which paths need extra review.** Auth, crypto, secrets, and deployment config always get your eyes.
- **Whether the code is maintainable.** If you cannot explain it, you cannot maintain it — reject it.

11 Human decisions and responsibilities (non-delegable judgment) (continued)

[Human decision-point box] Before accepting any AI-generated change, answer: Have I read every line of the diff? Do I understand what each changed file does? Are secrets, auth, and HTTPS handled correctly? If any answer is no, do not accept.

12 Inspect and verify AI output (diff-review checklist)

QUALITY CHECK

For every AI-generated diff, confirm:

12 Inspect and verify AI output (diff-review checklist) (continued)

- ✓ [ ] I have read every added and removed line.
- ✓ [ ] Each file is classified safe / needs-review / reject.
- ✓ [ ] No secrets, API keys, or credentials are hardcoded. [S-01, G-05]
- ✓ [ ] Authentication and authorization logic is reviewed. [S-04, E-03]
- ✓ [ ] HTTPS and deployment config are intact. [S-02]
- ✓ [ ] Input validation is present where user data is involved. [S-03]
- ✓ [ ] No unrelated files were changed.

12 Inspect and verify AI output (diff-review checklist) (continued)

- ✓ [ ] I can explain what the code does.

12 Inspect and verify AI output (diff-review checklist) (continued)

[Related to D-03, D-04, Q-07]

13 Failure modes and recovery

● WHEN IT GOES WRONG

13 Failure modes and recovery (continued)

FAILURE	SYMPTOM	LIKELY CAUSE	RECOVERY
Accept All habit	Broken or insecure feature reaches production	Diff not read; blind acceptance	Revert the commit; re-review; accept only what passes
Missed secret	API key appears in source or logs	Not scanning diff for secrets	Rotate the key immediately; remove from code [G-05, S-01]

13 Failure modes and recovery (continued)

FAILURE	SYMPTOM	LIKELY CAUSE	RECOVERY
Security-critical path skipped	Auth or crypto changed without review	Assumed AI knows best	Human review mandatory for auth/crypto/secrets/deploy D-04
Unrelated changes	Files you did not ask about were modified	Prompt too broad	Reject unrelated changes; re-prompt with narrower scope

13 Failure modes and recovery (continued)

FAILURE	SYMPTOM	LIKELY CAUSE	RECOVERY
AI review false confidence	AI says code is safe but it is not	Overtrusting AI review	You own final approval; AI review is a second opinion F-08

13 Failure modes and recovery (continued)

All scenarios in this section are hypothetical and labeled as such.

14 Professional standard — quality threshold before accepting AI output

PRO STANDARD

AI-generated code is accepted only when:

- Every line of the diff has been read and classified.
- No secrets appear in code or configuration. [S-01, G-05]
- Security-critical paths (auth, crypto, secrets, deployment config) have human review. D-04
- Input validation is present where user data is involved. [S-03]
- Tests pass after the change is applied. F-01 · F-02
- You can explain what the code does. [M-02]

15 Hands-on exercise (L4 — Analysis)

BUILD IT

Task: Review three AI-generated snippets and classify each as safe / needs-review / reject, with reasons.

Constraints:

- Do not ask the AI to regenerate anything.
- Work only with the snippets provided.
- Write specific revision notes for needs-review and reject items.

Deliverable: A classification table (three snippets) with reasons and revision notes.

Snippet A (login form):

HTML

```
<input type="password" name="password">
<button type="submit">Login</button>
```

15 Hands-on exercise (L4 — Analysis) (continued)

And backend:

```JS
app.post('/login', (req, res) ⇒ {
  const { password } = req.body;
  db.query(`SELECT * FROM users WHERE password =
'${password}'`);
});
```

Snippet B (cart button):

```JS
function addToCart(productId) {
  cart.push(productId);
  updateBadge();
}
```

Snippet C (config):

15 Hands-on exercise (L4 — Analysis) (continued)

```JS
const API_KEY = "sk_live_9f8e7d6c5b4a";
const FORCE_HTTPS = false;
```

Success criteria:

15 Hands-on exercise (L4 — Analysis) (continued)

(continued)

- Snippet A classified reject (SQL injection, plaintext password comparison, no hashing).
- Snippet B classified safe (simple, understandable, matches intent).
- Snippet C classified reject (hardcoded secret, HTTPS disabled).
- Each classification includes a one-sentence reason.

Assessment level: L4 (Analysis — investigates evidence, diagnoses a failure)

Time estimate: 30–40 minutes.

16 Build challenge (L4 — Analysis) with acceptance criteria

● PUSH FURTHER

Task: Review an AI-generated "add to cart" diff for Mifugo Market, identify the flaws, request specific revisions, and accept only the safe parts.

Constraints:

- Do not ask the AI to regenerate the whole app.
- Make the smallest possible revision requests.
- Commit only what passes review.

Acceptance criteria (Given–When–Then):

16 Build challenge (L4 — Analysis) with acceptance criteria (continued)

- **Given** an AI-generated diff for the cart feature, **When** you review it line by line, **Then** each file is classified safe / needs-review / reject with a reason.
- **Given** a needs-review item, **When** you request a revision, **Then** the request is specific (names the file, the line, and the fix).
- **Given** a reject item (for example, hardcoded secret or disabled HTTPS), **When** you decline it, **Then** you state the security risk and do not accept the change.
- **Given** the accepted changes are applied, **When** you run the test suite, **Then** all tests pass (no regression). F-02

Deliverable: A diff-review log with classifications, revision requests, and the final accepted commit.

16 Build challenge (L4 — Analysis) with acceptance criteria (continued)

Assessment level: L4 (Analysis — investigates evidence, diagnoses a failure, plans recovery)

Time estimate: 50–60 minutes.

17 Key takeaways

KEY TAKEAWAYS

17 Key takeaways (continued)

- Read every line of every diff before accepting.
- Classify changes as safe / needs-review / reject.
- Watch for three AI error categories: logic, security, data handling.

17 Key takeaways (continued)

- Auth, crypto, secrets, and deployment config always need human review.
- Give specific feedback, not vague "fix this" comments.
- If you cannot explain the code, you do not accept it.

17 Key takeaways (continued)

- Tests run after every accepted change to catch regression.

18 Glossary of new terms

TERMS INTRODUCED IN THIS CHAPTER

18 Glossary of new terms (continued)

TERM	PLAIN-LANGUAGE DEFINITION	EXAMPLE	WHY IT MATTERS	TAG
Diff	A line-by-line record of changes between two versions.	`+` lines added, `-` lines removed	Shows exactly what the AI changed so you can review it.	
Added line	A line starting with `+` in a diff.	`+cart.push(item);`	New code being introduced; must be understood before accepting.	

18 Glossary of new terms (continued) (continued)

TERM	PLAIN-LANGUAGE DEFINITION	EXAMPLE	WHY IT MATTERS	TAG
Removed line	A line starting with `-` in a diff.	`-const HTTPS = true;`	Code being deleted; check nothing important was removed.	
Context line	An unchanged line in a diff, shown for context.	Lines starting with a space	Helps you locate where the change fits in the file.	

18 Glossary of new terms (continued) (continued)

TERM	PLAIN-LANGUAGE DEFINITION	EXAMPLE	WHY IT MATTERS	TAG
File path	The location of a changed file.	`routes/cart.js`	Tells you which file was changed and where.	
Hunk	A group of related changes in one file.	A block of `+` and `-` lines with context	Keeps related changes together for easier review.	

18 Glossary of new terms (continued) (continued)

TERM	PLAIN-LANGUAGE DEFINITION	EXAMPLE	WHY IT MATTERS	TAG
Unified diff	A standard diff format with file paths, line numbers, and `+` / `-`.	The format Git and most editors show	The standard format you will review in your editor.	
Safe change	A change you understand and that introduces no risk.	Changing button text	Can be accepted after reading.	

18 Glossary of new terms (continued) (continued)

TERM	PLAIN-LANGUAGE DEFINITION	EXAMPLE	WHY IT MATTERS	TAG
Needs-review change	A change you are unsure about or that touches something important.	A new route handler	Request specific revision before accepting.	

18 Glossary of new terms (continued)

TERM	PLAIN-LANGUAGE DEFINITION	EXAMPLE	WHY IT MATTERS	TAG
Reject	A change you decline due to risk, unrelated changes, or lack of understanding.	Hardcoded secret	Do not accept; explain why and request a fix.	
Security-critical path	Code handling auth, crypto, secrets, or deployment config.	Login route, `.env` handling	Always gets mandatory human review.	[security]
False confidence	Trusting AI output because it looks fluent.	Accepting code that "looks right" but is insecure	The main reason diff review exists.	[AI]

19 Knowledge-check questions

KNOWLEDGE CHECK

Attempt these before looking at the answers below.

Q1. What does a line starting with `+` mean in a diff?

Q2. Name the three categories of AI errors this chapter teaches.

Q3. Which code paths always require mandatory human review?

Q4. True or false: An AI's own review of its code is sufficient; you do not need to read the diff.

19 Knowledge-check questions (continued)

Q5. What should you do if you cannot explain what an AI-generated change does?

[PAGE BREAK — ANSWER KEY]

Answers

A1. It means a line was added (new code being introduced).

A2. Logic errors (runs but does the wrong thing), security errors (insecure code), and data-handling errors (mishandles data).

19 Knowledge-check questions (continued)

A3. Authentication and authorization logic, cryptographic implementations, sensitive data handling, and deployment configuration.

A4. False. AI review produces false positives and negatives; a human always owns final approval.

A5. Reject it. If you cannot explain the code, you cannot maintain it or verify it is safe.

20 Links to earlier and later chapters

WHERE THIS SITS IN THE BOOK

Earlier chapters (prerequisites):

- **Chapter 6:** Tools, Environments, and Your First Win — the commits you make here are what you review as diffs.
- **Chapter 7:** Prompting, Context, and AI Collaboration — structured prompts produce diffs worth reviewing.

Later chapters (where this is used next):

20 Links to earlier and later chapters (continued)

- **Chapter 9–13:** Every AI-generated feature in these chapters must pass this diff-review bar before being accepted.
- **Chapter 14:** Deployment — deployment configuration changes are security-critical paths requiring human review.
- **Chapter 16:** Capstone — the capstone rubric requires all AI-generated code to be diff-reviewed.

20 Links to earlier and later chapters (continued)

Evidence notes: Claims trace to D-03/D-04/D-05 (AI coding best practices and failure modes), E-03/E-04 (authn/authz), G-05 (secrets), S-01–S-05 (security practices), F-08 (AI review false confidence). All scenarios are hypothetical.

Time estimate: Approximately 130 minutes total (70 minutes reading, 60 minutes hands-on practice).

PART 4

4

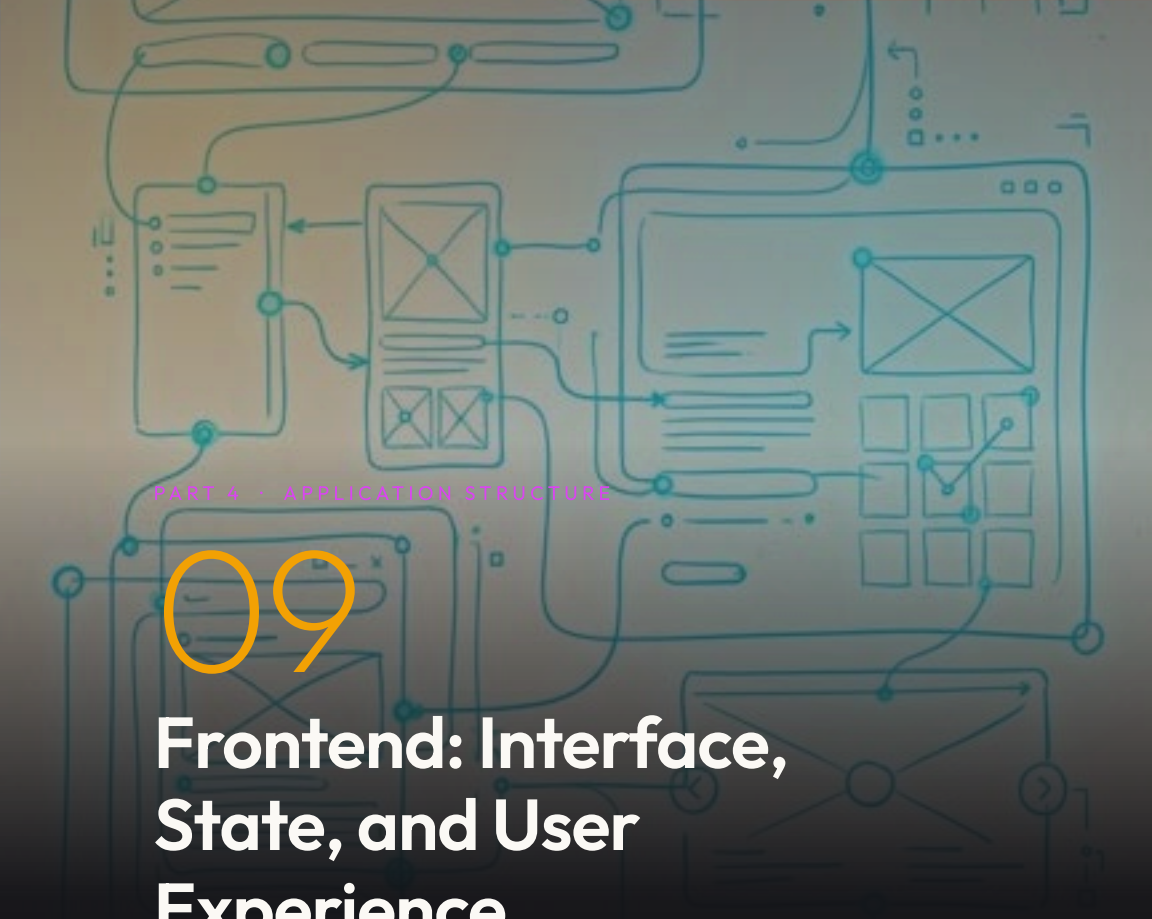
Application Structure

Frontend, backend, identity

09 Frontend: Interface, State, and User Experience

10 Backend, Data, and APIs

11 Access Control, Authentication, and Secure Connections



PART 4 · APPLICATION STRUCTURE

09

Frontend: Interface, State, and User Experience

01 Opening problem — "frontend is just making it look pretty"

SLOP DETECTOR — THE MISCONCEPTION

The scenario

Wanjiku's Mifugo Market landing page is live. She now wants customers to place produce orders through a form on the page. She asks her AI assistant to "make an order form." The AI generates a form with input fields and a submit button. It looks clean. But when Wanjiku tests it:

- Submitting an empty form sends blank data to the server (no validation).
- After clicking submit, nothing happens for several seconds — no spinner, no message (no loading state).
- When the server is down, the page shows nothing at all (no error state).
- When there are no past orders, the page shows a blank white space (no empty state).
- The text is gray on white, hard to read; the submit button is tiny on a phone (WCAG failures).

The form "looks pretty" but fails the people who need to use it. Frontend is not decoration — it is the part of the software users actually interact with, and slop here is immediately visible.

The misconception

01 Opening problem — "frontend is just making it look pretty" (continued)

MISCONCEPTION	WHY IT IS DANGEROUS	THE CORRECTION
"Frontend is just making it look pretty."	The frontend is where users meet your product. Without validation, loading, empty, and error states, users cannot tell whether their action succeeded or failed.	Frontend state and feedback are functional requirements, not decoration.
"If the user makes a mistake, that's their problem."	Users will mistype, lose connection, and submit empty forms. Software must handle these gracefully.	Validate input, show clear errors, and guide users toward success.

02 Why this topic matters in real software projects

The frontend is the interface between your application and the people who use it. Every decision here — whether a button is reachable, whether an error is explained, whether the page works on a cheap phone — shapes whether a real customer can complete an order or gives up and goes elsewhere.

Constitution §5 lists accessibility and responsiveness as professional standards, not optional extras. WCAG 2.1 AA (contrast 4.5:1, keyboard operability, visible focus, alt text) and mobile-first design (viewport, fluid units, touch targets) are the baseline for a credible public application.

This chapter teaches you to build a frontend that communicates with the user in every state: idle, validating, loading, success, error, and empty.

03 Learning objectives

WHAT YOU WILL BE ABLE TO DO

By the end of this chapter, the reader can:

- Build a form with validation, loading state, empty state, and error message.
- Make a page responsive (mobile-first, viewport meta tag, touch targets ≥44 px).
- Meet WCAG 2.1 AA basics (contrast 4.5:1, keyboard operability, visible focus, alt text, labels on inputs).
- Explain frontend state (what the user sees right now) versus backend state.
- Direct the AI to generate a frontend feature and review it against accessibility criteria.

04 Prerequisites and what to skip

BEFORE YOU START

Prerequisites

Chapters 1–8. You should have a static landing page (Chapter 6), understand HTTP request/response (Chapter 1), and know how to review AI diffs (Chapter 8).

What you can skip

No backend or database in this chapter. The form submits to a placeholder; the backend connection comes in Chapter 10. Advanced JavaScript frameworks are excluded — plain HTML/CSS/JavaScript only (Constitution §6, stack decision).

05 Plain-English explanation

State is what the user sees right now

Frontend state describes the current condition of the screen: Is the form empty? Is it being submitted? Did it succeed? Did it fail? Is there nothing to show? Each condition should look and feel different so the user is never left wondering.

Loading, empty, and error states are promises to the user

- **Loading state:** "I am working on it." A spinner or message while data fetches or a form submits.
- **Empty state:** "There is nothing here." A helpful message when there are no orders, no products, or no results.
- **Error state:** "Something went wrong." A clear message explaining what happened and what to try next.

Without these, users click repeatedly, assume the app is broken, or abandon it.

Responsive design means it works on any screen

Mobile-first means you design for small screens first (where most of your users in Nairobi and Ruiru will be), then add layouts for larger screens. The viewport meta tag tells browsers to scale the page correctly on phones. Touch targets should be at least 44×44 pixels so fingers can tap them accurately.

Accessibility means everyone can use it

WCAG 2.1 AA is a recognized international standard C-04. Key basics:

05 Plain-English explanation (continued)

- **Contrast:** text needs at least 4.5:1 contrast against its background (3:1 for large text).
- **Keyboard:** everything must be operable by keyboard alone.
- **Focus:** the keyboard focus indicator must be visible.
- **Color:** color is never the only way to convey information.
- **Labels:** every input has a visible, associated label.
- **Alt text:** images have descriptions for screen readers.

06 Mental model — and where it breaks down

Analogy: a market stall counter

Think of your frontend as a market stall counter. The customer (user) walks up and wants to buy produce. A good counter has:

- A clear price list (labels).
- A way to hand over money and get confirmation (form + submit).
- A sign saying "sold out" when there is nothing (empty state).
- A sign saying "come back tomorrow" when the supplier is late (error state).
- A counter height that works for standing and seated customers (accessibility).
- Room for a wheelchair or a cart to approach (keyboard access).

Where the analogy breaks down

Unlike a physical stall, a web form can fail silently — the customer clicks submit and nothing happens, with no explanation. A good frontend never leaves the user in silence.

07 Technical explanation with correct terminology

Each term is defined at first use below and listed in the chapter glossary (Section 18).

- **Frontend state** — the current condition of what the user sees (idle, validating, loading, success, error, empty).
- **Loading state** — a screen or indicator showing that an action is in progress.
- **Empty state** — what is shown when there is nothing to display (no orders, no products).
- **Error state** — what is shown when something goes wrong, with a clear message.
- **Validation** — checking that user input meets rules before it is submitted.
- **Responsive design** — a layout that adapts to different screen sizes.
- **Mobile-first** — designing for small screens first, then adding larger-screen layouts.
- **Viewport meta tag** — an HTML tag telling browsers how to scale the page on mobile devices.
- **Touch target** — the tappable area of a button or link; minimum 44×44 pixels recommended.
- **Contrast ratio** — the difference in brightness between text and its background; 4.5:1 minimum for normal text (WCAG 2.1 AA).
- **Keyboard operability** — all functions usable with a keyboard alone.
- **Visible focus** — a visible indicator showing which element currently has keyboard focus.
- **Alt text** — a text description of an image for screen readers.
- **Form label** — visible text associated with an input field.
- **Backend state** — data and logic stored on the server (covered in Chapters 10–11).

07 Technical explanation with correct terminology

(continued)

[security] — Form validation (input is untrusted; validated on the server in Chapter 10/13)

08 Concrete example — Mifugo Market order inquiry form

IN PRACTICE

Wanjiku adds an "Order Inquiry" form to her landing page. Customers can request produce without a backend yet — the form collects their name, email, phone, and items of interest, and shows feedback at every step:

- **Idle:** empty form with labels and placeholder hints.
- **Validating:** if they submit with missing fields, inline error messages appear under each field.
- **Loading:** a spinner appears while the form "submits" (simulated).
- **Success:** a confirmation message: "Thank you, Wanjiku's Market has received your order."
- **Error:** if the submit fails, a message: "Could not send your order. Please try again or call 0700-000-000."
- **Empty:** when there are no past inquiries to show (for a future order history section), a message: "No orders yet."

The form meets WCAG basics: labels on every input, 4.5:1 contrast, visible focus ring, tap targets ≥44 px, and works on a narrow phone screen.

09 Step-by-step demonstration — building the order inquiry form

STEP BY STEP

Step 1: Write the HTML form with labels

09 Step-by-step demonstration — building the order inquiry form (continued)

HTML

```
<form id="order-form" novalidate>
  <div class="field">
    <label for="name">Full name</label>
    <input type="text" id="name" name="name" required
minlength="2">
    <span class="error" id="name-error"></span>
  </div>
  <div class="field">
    <label for="email">Email address</label>
    <input type="email" id="email" name="email" required>
    <span class="error" id="email-error"></span>
  </div>
  <div class="field">
    <label for="phone">Phone number</label>
    <input type="tel" id="phone" name="phone" required pattern="
[0-9]{9,12}">
    <span class="error" id="phone-error"></span>
  </div>
  <div class="field">
    <label for="items">Items of interest</label>
    <textarea id="items" name="items" required></textarea>
    <span class="error" id="items-error"></span>
  </div>
  <button type="submit" id="submit-btn">Send order
inquiry</button>
  <div id="loading" class="hidden" aria-live="polite">Sending…
</div>
  <div id="success" class="hidden" role="status" aria-
live="polite">
    Thank you! Your order inquiry has been received.
  </div>
```

09 Step-by-step demonstration — building the order inquiry form (continued) (continued)

```HTML
  <div id="error" class="hidden" role="alert" aria-
live="polite">
    Could not send your order. Please try again or call 0700-
000-000.
  </div>
</form>
```

09 Step-by-step demonstration — building the order inquiry form (continued)

Step 2: Add CSS for states, responsiveness, and accessibility

Key rules:

09 Step-by-step demonstration — building the order inquiry form (continued)

```CSS
/* Mobile-first: single column, full-width inputs */
.field input, .field textarea {
  width: 100%;
  padding: 12px;
  font-size: 16px;
  border: 2px solid #444;
  border-radius: 4px;
}

/* Visible focus (WCAG 2.4.7) */
.field input:focus, .field textarea:focus {
  outline: 3px solid #0066cc;
```

09 Step-by-step demonstration — building the order inquiry form (continued) (continued)

```css
  outline-offset: 2px;
}

/* Touch targets ≥44px */
button {
  min-height: 48px;
  padding: 0 24px;
  font-size: 16px;
}

/* Error messages */
.error {
```

09 Step-by-step demonstration — building the order inquiry form (continued) (continued)

```css
  color: #b00020;
  font-size: 14px;
  display: none;
}
.field input:invalid:not(:placeholder-shown) {
  border-color: #b00020;
}

/* Loading/success/error states */
#loading, #success, #error {
  padding: 12px;
```

09 Step-by-step demonstration — building the order inquiry form (continued)

```css
  margin-top: 12px;
  border-radius: 4px;
}
#loading { background: #eef; }
#success { background: #efe; color: #060; }
#error { background: #fee; color: #b00; }
.hidden { display: none; }

/* Contrast: body text ≥4.5:1 */
body { color: #1a1a1a; background: #ffffff; font-family: system-ui, sans-serif; }

/* Responsive: larger screens get a centered max-width */
@media (min-width: 600px) {
  form { max-width: 500px; margin: 0 auto; }
}
```

09 Step-by-step demonstration — building the order inquiry form (continued)

Contrast check: `#1a1a1a` on `#ffffff` is approximately 15:1, well above 4.5:1 C-04.

Step 3: Add JavaScript for validation, loading, and states

09 Step-by-step demonstration — building the order inquiry form (continued)

JS

```
const form = document.getElementById('order-form');
const loading = document.getElementById('loading');
const success = document.getElementById('success');
const error = document.getElementById('error');
const submitBtn = document.getElementById('submit-btn');

form.addEventListener('submit', async (event) => {
  event.preventDefault();

  // Clear previous errors
  document.querySelectorAll('.error').forEach(e =>
(e.textContent = ''));

  // Client-side validation
  let valid = true;
  const name = document.getElementById('name');
  if (name.value.trim().length < 2) {
    document.getElementById('name-error').textContent = 'Please
enter your name.';
    valid = false;
  }
  const email = document.getElementById('email');
  if (!email.value.includes('@')) {
    document.getElementById('email-error').textContent = 'Please
enter a valid email.';
    valid = false;
  }
  if (!valid) return;

  // Loading state
  loading.classList.remove('hidden');
  success.classList.add('hidden');
  error.classList.add('hidden');
  submitBtn.disabled = true;
```

09 Step-by-step demonstration — building the order inquiry form (continued)

```JS
  // Simulate submission (replace with real fetch in Chapter 10)
  await new Promise((resolve) => setTimeout(resolve, 1500));

  // Success state
  loading.classList.add('hidden');
  success.classList.remove('hidden');
  form.reset();
  submitBtn.disabled = false;
});
```

09 Step-by-step demonstration — building the order inquiry form (continued)

Step 4: Test on a narrow screen

Open the page in your browser. Resize the window to phone width (~375 px). Confirm:

- Inputs are full-width and easy to tap.
- The focus ring is visible when you Tab through fields.
- The error message appears when you submit an empty form.
- The spinner appears during the simulated submit.
- The success message appears after submit.

10 AI collaboration example (prompt format P1–P10)

AI COLLABORATION — PROMPT FORMAT P1–P10

PROMPT FORMAT **A reusable prompt you can copy**

P1 **P1 · PROBLEM IT SOLVES**
Generate a frontend form with validation, loading, empty, and error states that meets accessibility basics.

P2 **P2 · WHEN TO USE IT**
Chapter 9, when building an order or contact form.

P3 **P3 · REQUIRED INPUTS**
The form fields needed, the page it will live on, and the acceptance criteria from Chapter 5.

P4 **P4 · THE PROMPT**

```text
Role: You are a careful frontend developer focused on
accessibility.
Context: Mifugo Market landing page (static HTML/CSS/JS).
See AGENTS.md for conventions.
Task: Add an order inquiry form with fields: name, email,
phone, items of interest.
Requirements:
- Mobile-first layout; viewport meta tag present.
- Labels on every input; visible focus ring; contrast
≥4.5:1.
- Client-side validation with inline error messages.
- Loading, success, and error states shown to the user.
- Empty state message for "no orders yet" (as a placeholder
section).
- Touch targets ≥44px.
- Return a single HTML file with embedded CSS and JS.
- Do not include a backend; form submission is simulated.
- Reference AGENTS.md for project conventions.
```

10 AI collaboration example (prompt format P1–P10) (continued)

P5 **P5 · WHAT A GOOD RESPONSE CONTAINS**
Single HTML file; labels and focus states; validation messages; loading/success/error divs; mobile-first CSS; no backend code.

P6 **P6 · WHAT THE READER MUST INSPECT**
Open in browser; test on narrow screen; Tab through fields to check focus; submit empty form to check validation; confirm contrast is readable; confirm no backend or API keys appear.

P7 **P7 · WEAK OR INCOMPLETE RESPONSE**
No labels; color as the only error indicator; no loading state; no error message; hardcoded API keys; desktop-only layout.

P8 **P8 · FOLLOW-UP WHEN INADEQUATE**
"Add labels to every input," "Add a loading spinner," "Make the layout mobile-first," "Remove the API key."

P9 **P9 · SAFETY/SECURITY CONSIDERATIONS**
Client-side validation is not security — the real validation happens on the backend (Chapter 10/13). Do not treat frontend validation as access control.

P10 **P10 · VERIFICATION STEP**
Submit the form empty; confirm inline errors appear; submit valid data; confirm loading spinner then success message; confirm page works on phone width.

11 Human decisions and responsibilities (non-delegable judgment)

● YOUR DECISION — NOT THE AI'S

11 Human decisions and responsibilities (non-delegable judgment) (continued)

- **Which fields are required.** You decide what information is necessary (data minimization, Chapter 13).
- **What the error messages say.** You write user-friendly messages, not technical jargon.

11 Human decisions and responsibilities (non-delegable judgment) (continued)

- **Whether the form meets accessibility needs.** You test keyboard access and contrast; the AI may guess wrong.

11 Human decisions and responsibilities (non-delegable judgment) (continued)

- **When frontend validation is enough versus when backend validation is required.** You know frontend validation can be bypassed (Chapter 13).

11 Human decisions and responsibilities (non-delegable judgment) (continued)

[Human decision-point box] Before you consider a form complete, answer: Does every input have a label? Can I Tab through the form without a mouse? Is there a loading state? Is there an error message? Is there an empty state? If any answer is no, keep refining.

12 Inspect and verify AI output (frontend checklist)

QUALITY CHECK

Before accepting AI-generated frontend code:

- ✓ [ ] Every input has a visible, associated label.
- ✓ [ ] Focus ring is visible when Tabbing through the page.
- ✓ [ ] Contrast meets 4.5:1 for body text (check with a contrast tool).
- ✓ [ ] Touch targets are at least 44×44 pixels.
- ✓ [ ] The page works on a narrow screen (mobile-first).
- ✓ [ ] Form has validation, loading, success, and error states.
- ✓ [ ] Empty state is considered (what shows when there is nothing).
- ✓ [ ] No color-only information (errors are not red alone — they also have text).
- ✓ [ ] No hardcoded secrets or API keys in frontend code.

[Related to C-04, C-05, C-06, Q-04, Q-05]

13 Failure modes and recovery

WHEN IT GOES WRONG

13 Failure modes and recovery (continued)

FAILURE	SYMPTOM	LIKELY CAUSE	RECOVERY
No validation	Empty form submits successfully	Validation missing or client-side only	Add required fields and inline validation; validate on server in Chapter 10
No loading state	User clicks submit twice; nothing happens	No feedback during submission	Add spinner and disable button during submit
No error state	Server failure shows blank page	No error handling	Add error div with clear message and recovery action
No empty state	Blank screen when no data	Not considered	Add "No orders yet" message with next steps
Color-only errors	Errors shown only in red text	Color used as sole information carrier	Add text error messages and icons

13 Failure modes and recovery (continued)

FAILURE	SYMPTOM	LIKELY CAUSE	RECOVERY
Touch targets too small	Buttons hard to tap on phone	Buttons too small or close together	Enlarge to ≥44px; add spacing

13 Failure modes and recovery (continued)

All scenarios in this section are hypothetical and labeled as such.

14 Professional standard — quality threshold before going live

● PRO STANDARD

A frontend form is ready only when:

14 Professional standard — quality threshold before going live (continued)

- **WCAG 2.1 AA basics:** contrast ≥4.5:1, keyboard operable, visible focus, labels on inputs, alt text on images [C-04, Q-05].
- **Mobile-first responsive:** viewport meta tag, fluid layout, touch targets ≥44px [C-05, C-06, Q-04].

14 Professional standard — quality threshold before going live (continued)

- **All four states:** validation, loading, empty, and error are present and tested.
- **Frontend validation is not security:** real validation happens on the backend (Chapter 10/13) [S-03].

14 Professional standard — quality threshold before going live (continued)

- **No secrets in frontend code:** no API keys or credentials in HTML/JS [S-01].

15 Hands-on exercise (L3 — Application)

BUILD IT

Task: Add an order inquiry form to your deployed static landing page (Mifugo Market or your own).

Constraints:

15 Hands-on exercise (L3 — Application) (continued)

- Use the page you deployed in Chapter 6.
- No backend; simulate submission with JavaScript.
- Do not commit `.env` or secrets.

Deliverable: A deployed page with a working order inquiry form.

Success criteria:

15 Hands-on exercise (L3 — Application) (continued)

- Form has name, email, phone, and items fields with labels.
- Submitting empty fields shows inline error messages.
- Loading spinner appears during submission.
- Success and error messages appear appropriately.
- Page is mobile-friendly (test on narrow screen).

15 Hands-on exercise (L3 — Application) (continued)

- Contrast and focus states meet WCAG basics.

15 Hands-on exercise (L3 — Application) (continued)

Assessment level: L3 (Application — performs a procedure with guidance)

Time estimate: 90–110 minutes.

16 Build challenge (L3 — Application) with acceptance criteria

● PUSH FURTHER

Task: Complete an accessibility self-check on your form and fix any issues found.

Constraints:

- Use only keyboard (no mouse) to navigate the form.
- Use a free contrast checker to verify text contrast.
- Document any issues and fixes.

Acceptance criteria (Given-When-Then):

- **Given** the form is open, **When** the user tabs through all fields, **Then** every field receives visible focus in a logical order.
- **Given** the form is viewed on a phone-width screen, **When** the user taps each field, **Then** each touch target is at least 44×44 pixels.
- **Given** body text and background are checked with a contrast tool, **Then** contrast is at least 4.5:1.
- **Given** an error occurs, **When** it is shown, **Then** it is communicated in text, not color alone.

Deliverable: An accessibility self-checklist (completed) plus a note of fixes made.

Assessment level: L3 (Application — performs a procedure with guidance)

Time estimate: 45–60 minutes.

17 Key takeaways

KEY TAKEAWAYS

- Frontend is functional, not decorative: validation, loading, empty, and error states are required.
- Mobile-first design means small screens first, then larger layouts.
- WCAG 2.1 AA basics: contrast 4.5:1, keyboard access, visible focus, labels, alt text.
- Frontend validation helps users; backend validation protects the system (Chapter 10/13).
- Test with a keyboard and a narrow screen before considering a form complete.

18 Glossary of new terms

TERMS INTRODUCED IN THIS CHAPTER

18 Glossary of new terms (continued)

TERM	PLAIN-LANGUAGE DEFINITION	EXAMPLE	WHY IT MATTERS	TAG
Frontend state	The current condition of what the user sees.	Loading, success, error, empty	Keeps the user informed at every step.	
Loading state	A screen or indicator showing an action is in progress.	"Sending..." spinner	Prevents users from clicking twice or abandoning.	
Empty state	What is shown when there is nothing to display.	"No orders yet"	Prevents confusing blank screens.	
Error state	What is shown when something goes wrong, with a clear message.	"Could not send your order"	Tells users what happened and what to do next.	

18 Glossary of new terms (continued) (continued)

TERM	PLAIN-LANGUAGE DEFINITION	EXAMPLE	WHY IT MATTERS	TAG
Validation	Checking user input meets rules before submission.	"Please enter a valid email"	Catches mistakes early; server validates for security (Ch 10/13).	
Responsive design	A layout that adapts to different screen sizes.	Single column on phone, wider on desktop	Works on any device your customers use.	
Mobile-first	Designing for small screens first, then adding larger layouts.	Base styles for phones; media queries for desktops	Most users in Kenya access the web on phones.	

18 Glossary of new terms (continued)

TERM	PLAIN-LANGUAGE DEFINITION	EXAMPLE	WHY IT MATTERS	TAG
Viewport meta tag	HTML tag telling browsers how to scale on mobile.	`<meta name="viewport" content="width=device-width, initial-scale=1">`	Prevents zoomed-out, unusable mobile views.	
Touch target	The tappable area of a button or link.	≥44×44 pixels recommended	Fingers can tap accurately on touchscreens.	
Contrast ratio	Brightness difference between text and background.	4.5:1 minimum for normal text	Readable for users with low vision (WCAG 2.1 AA).	

18 Glossary of new terms (continued) (continued)

TERM	PLAIN-LANGUAGE DEFINITION	EXAMPLE	WHY IT MATTERS	TAG
Keyboard operability	All functions usable with a keyboard alone.	Tabbing through form fields	Essential for users who cannot use a mouse.	
Visible focus	A visible indicator showing which element has keyboard focus.	Outline ring around the focused input	Lets keyboard users know where they are.	
Alt text	A text description of an image for screen readers.	"Sukuma wiki, KSh 40 per kg"	Makes images accessible to blind users.	
Form label	Visible text associated with an input field.	"Email address" next to the email input	Tells users what to enter; required for accessibility.	

18 Glossary of new terms (continued)

TERM	PLAIN-LANGUAGE DEFINITION	EXAMPLE	WHY IT MATTERS	TAG
Backend state	Data and logic stored on the server.	User accounts, order history	Covered in Chapters 10–11.	

19 Knowledge-check questions

KNOWLEDGE CHECK

Attempt these before looking at the answers below.

Q1. Name the four frontend states every form should handle.

Q2. What is the minimum contrast ratio for normal text under WCAG 2.1 AA?

Q3. What is the recommended minimum size for a touch target?

19 Knowledge-check questions (continued)

Q4. True or false: Frontend validation alone is enough to secure a form.

Q5. What does the viewport meta tag do?

[PAGE BREAK — ANSWER KEY]

Answers

19 Knowledge-check questions (continued)

A1. Validation (checking input), loading (action in progress), empty (nothing to show), and error (something went wrong).

A2. 4.5:1 for normal text; 3:1 for large text C-04.

A3. At least 44×44 CSS pixels, with about 8 pixels of spacing between targets C-05.

19 Knowledge-check questions (continued)

A4. False. Frontend validation helps users but can be bypassed. Real validation happens on the backend (Chapter 10/13).

A5. It tells the browser how to scale and size the page on mobile devices, preventing zoomed-out views.

20 Links to earlier and later chapters

WHERE THIS SITS IN THE BOOK

Earlier chapters (prerequisites):

20 Links to earlier and later chapters (continued)

- **Chapter 6:** Tools, Environments, and Your First Win — your static landing page and deploy pipeline come from here.
- **Chapter 8:** The Diff Discipline — the form's AI-generated code must pass diff review before acceptance.

Later chapters (where this is used next):

20 Links to earlier and later chapters (continued)

- **Chapter 10:** Backend, Data, and APIs — the form will submit to a real backend endpoint.
- **Chapter 13:** Security, Privacy, and Data Protection — input validation and data minimization apply to form data.

20 Links to earlier and later chapters (continued)

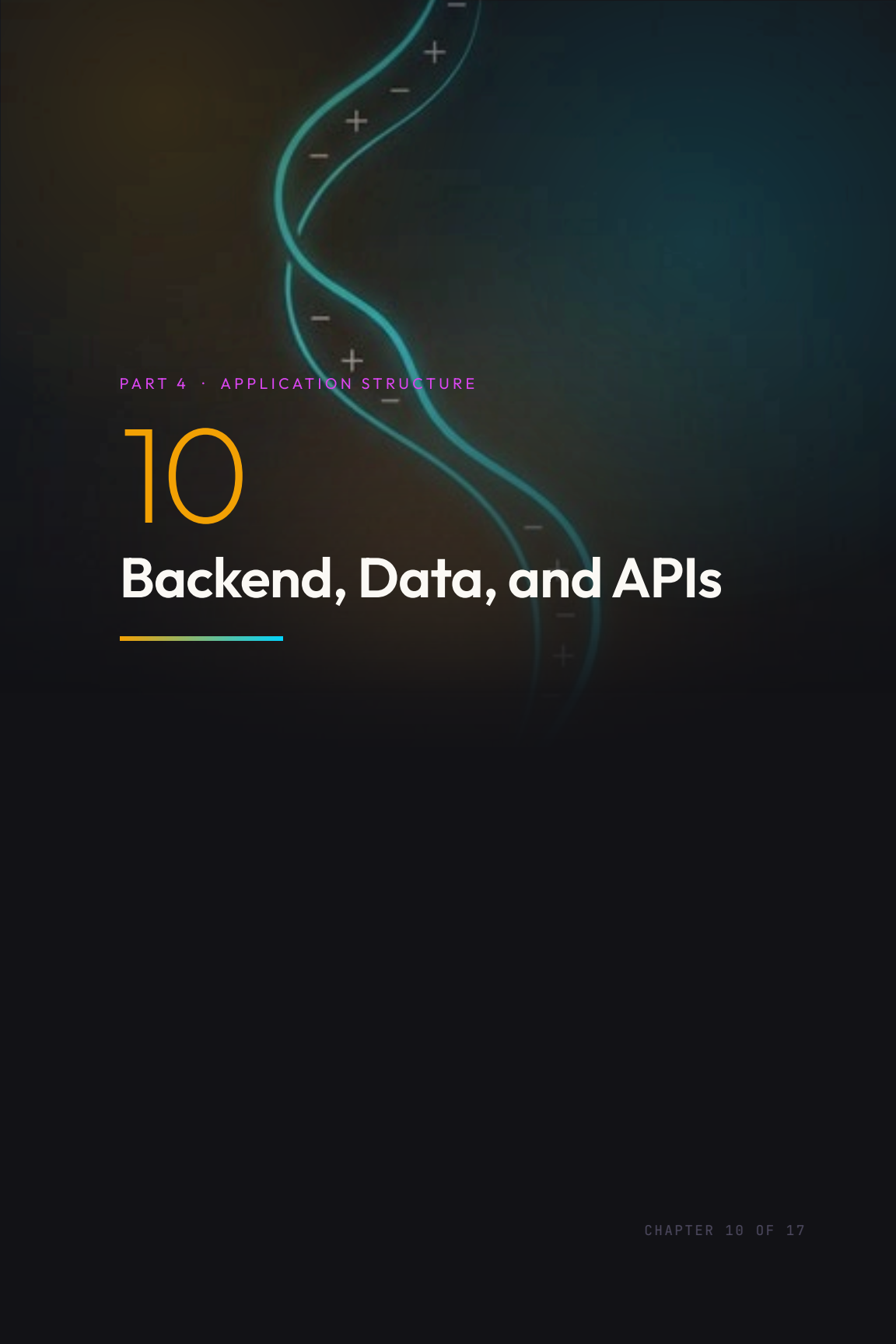
- **Chapter 16:** Capstone — the accessible, responsive form is part of the final Mifugo Market.

20 Links to earlier and later chapters (continued)

Evidence notes: Claims trace to C-04 (WCAG 2.1 AA), C-05 (touch targets), C-06 (mobile-first responsive design), Q-04/Q-05 (quality practices), S-03 (input validation). All scenarios are hypothetical.

20 Links to earlier and later chapters (continued)

Time estimate: Approximately 200 minutes total (90 minutes reading, 110 minutes hands-on practice).



PART 4 · APPLICATION STRUCTURE

10

Backend, Data, and APIs

01 Opening problem — "the backend is where the AI does everything"

SLOP DETECTOR — THE MISCONCEPTION

The scenario

Wanjiku's Mifugo Market landing page has an order inquiry form (Chapter 9). Now she wants customers to browse actual produce and place real orders. She tells her AI assistant: "Build me a backend with a database and API." The AI generates a Node.js server, a database file, and several API routes — all in one response. It looks like a complete application, but Wanjiku does not know what any of it does. She cannot explain where data is stored, how the frontend talks to the server, or what happens when a request fails.

She has code that runs, but no understanding of the system. That is not a product — that is slop she cannot debug or secure.

The misconception

MISCONCEPTION	WHY IT IS DANGEROUS	THE CORRECTION
"The backend is where the AI does everything; I don't need to understand data or APIs."	Without understanding the backend, you cannot debug failures, secure data, or verify that the AI built what you asked for.	You need a working mental model of backend, database, and API — enough to read, review, and direct.
"JSON is just a document."	JSON is structured data sent between frontend and backend. Treating it like a document leads to broken parsing and mismatched fields.	JSON is data with a predictable structure; each field has a meaning.

02 Why this topic matters in real software projects

Every application that stores or retrieves information needs a backend: a server that receives requests, talks to a database, and sends responses. The frontend (Chapter 9) is what users see; the backend is where the application's logic and data live.

Constitution §7 requires that readers understand HTTP, JSON, REST, and where API keys belong (never in frontend code). Without this, you cannot build anything that remembers data between visits or integrates with other services.

This chapter gives you a working model: a backend endpoint that returns product data as JSON, a frontend that fetches and displays it, and a clear picture of how request and response flow between them.

03 Learning objectives

WHAT YOU WILL BE ABLE TO DO

By the end of this chapter, the reader can:

- Explain what a database does and why an application needs one.
- Define a simple data model for Mifugo Market (products, orders, users).
- Explain REST API and JSON.
- Call a public API and read the response.
- Distinguish frontend request from backend response.
- State where API keys belong (header, environment variable, never URL or client code).

04 Prerequisites and what to skip

BEFORE YOU START

Prerequisites

Chapters 1–9. You should understand HTTP request/response (Chapter 1), frontend state and forms (Chapter 9), and diff review (Chapter 8).

What you can skip

No database writes by the reader in this chapter. Database writes are practiced in the capstone (Chapter 16). SQL internals are introduced conceptually only.

05 Plain-English explanation

A backend is a server that answers requests

Your frontend runs in the user's browser. The backend runs on a server and waits for HTTP requests (Chapter 1). When a customer clicks "View products," the frontend sends a request to the backend; the backend reads data and sends back a response, usually in JSON format.

A database remembers information

A **database** stores data so it survives between visits and between server restarts. For Mifugo Market, the database stores products (name, price, stock), orders, and users. Without a database, every refresh would lose all data.

REST and JSON: how frontend and backend talk

REST is a style of web API where the frontend sends an HTTP request to an **endpoint** (a URL like `/api/products`), and the backend sends back a response. **JSON** is the format for that data — a text structure of key-value pairs that both sides understand.

Example JSON response:

05 Plain-English explanation (continued)

```JSON
[
  { "id": 1, "name": "Sukuma wiki", "price": 40, "stock_status": "in stock" },
  { "id": 2, "name": "Tomatoes", "price": 60, "stock_status": "in stock" },
  { "id": 3, "name": "Avocados", "price": 100, "stock_status": "sold out" }
]
```

API keys belong in headers and environment variables

05 Plain-English explanation (continued)

An **API key** identifies the application making a request (not the user — see Chapter 11). It should be sent in an `Authorization` header, never in the URL (URLs get logged), and never hardcoded in source code. In production it lives in an environment variable on the platform dashboard [E-02, S-06].

06 Mental model — and where it breaks down

Analogy: a restaurant kitchen

The frontend is the dining room: customers place orders and receive food. The backend is the kitchen: it receives the order, prepares it (queries the database), and sends the food back. JSON is the order ticket: a structured slip that both sides understand. The database is the pantry: where ingredients are stored.

Where the analogy breaks down

Unlike a kitchen, a backend does not forget orders between shifts unless you design it that way — and it can serve thousands of customers simultaneously, not just those in one room. The mental model helps for request/response, but scale and persistence are additional backend concerns.

07 Technical explanation with correct terminology

Each term is defined at first use below and listed in the chapter glossary (Section 18).

07 Technical explanation with correct terminology

(continued)

- **Backend** — server-side code that receives HTTP requests, processes them, and returns responses.
- **Database** — structured storage that persists data between requests and server restarts.
- **Data model** — the shape of your data: tables, records, and fields (for example, products with name, price, stock).
- **Table / Record** — a table is a collection (for example, products); a record is one item in it.
- **REST API** — a web API style where clients send HTTP requests to endpoints and receive responses.
- **API endpoint** — a specific URL path (for example, `GET /api/products`).
- **JSON** — JavaScript Object Notation, a text format for structured data.
- **Request** — a message from client to server asking for a resource or action.
- **Response** — the server's answer, with a status code and body.
- **Status code** — a three-digit result (200 OK, 404 Not Found, 403 Forbidden, 500 error).
- **API key** — a value identifying the application making a request (not the user).
- **Environment variable** — a value set outside code, read at runtime (for secrets and configuration).
- **Frontend request** — the browser asking the backend for something.

07 Technical explanation with correct terminology

(continued) (continued)

- **Backend response** — the server answering with data or a status.
- **Server-to-server** — the backend calling another service (where API keys belong).
- **Client-side** — code running in the browser (where secrets must never go).

07 Technical explanation with correct terminology

(continued)

[security] — API key, Environment variable (secrets never in frontend code)

08 Concrete example — Mifugo Market product catalog API

IN PRACTICE

Wanjiku adds a backend to Mifugo Market:

- **Data model (products):** id, name, price (KSh), stock_status.
- **Backend endpoint:** `GET /api/products` returns the product list as JSON.
- **Frontend:** the landing page fetches `/api/products` and displays each product with price and stock status.
- **Public API call:** for learning, Wanjiku also calls a public fake-store API and displays one field (for example, a product title) on her page — showing how to integrate with another service without a key.

She verifies:

08 Concrete example — Mifugo Market product catalog API (continued)

- The endpoint returns JSON, not HTML.
- Each product has name, price, and stock_status.
- Out-of-stock items show "sold out" and cannot be added to cart (from Chapter 9).

09 Step-by-step demonstration — product catalog API

STEP BY STEP

Step 1: Define the data model

On paper, list your tables and fields:

products:

- id (number)
- name (text)
- price (number, KSh)
- stock_status (text: "in stock" / "sold out")

orders (planned for capstone):

- id, customer_id, product_id, quantity, date, status

users (planned for Chapter 11):

- id, name, email, password_hash, role

Step 2: Generate the backend endpoint

Ask your AI assistant:

09 Step-by-step demonstration — product catalog API (continued)

```
TEXT
Role: You are a careful backend developer.
Context: Mifugo Market, a fresh-produce ordering platform.
Static frontend exists (Chapter 9).
Task: Add a Node.js/Express backend with a GET /api/products
endpoint returning JSON product data.
Data model (hardcoded for now): products = [{ id:1, name:'Sukuma
wiki', price:40, stock_status:'in stock' }, ...].
Constraints:
- Do not hardcode any secrets or API keys.
- Return JSON with status 200.
- Handle errors: if something fails, return status 500 with a
JSON error message.
- Output format: unified diff showing changed files.
```

09 Step-by-step demonstration — product catalog API (continued)

Review the diff (Chapter 8). Confirm:

- No secrets hardcoded.
- The route returns JSON.
- Error handling is present.

[Action] Save the AI-generated backend code as a file named `server.js` in your project root. This file is your backend application. Confirm the file exists before proceeding.

Step 3: Connect the frontend

In your frontend JavaScript, fetch the endpoint:

09 Step-by-step demonstration — product catalog API (continued) (continued)

```JS
async function loadProducts() {
  try {
    const response = await fetch('/api/products');
    if (!response.ok) throw new Error('Failed to load
products');
    const products = await response.json();
    displayProducts(products);
  } catch (err) {
    document.getElementById('product-list').textContent =
      'Could not load products. Please try again later.';
  }
}
```

Step 4: Call a public API (learning exercise)

09 Step-by-step demonstration — product catalog API (continued) (continued)

Call a public API without a key to see how integration works. For example, a public fake-store endpoint:

09 Step-by-step demonstration — product catalog API (continued)

```JS
async function loadPublicExample() {
  const response = await
fetch('https://fakestoreapi.com/products/1');
  const product = await response.json();
```

09 Step-by-step demonstration — product catalog API (continued) (continued)

```
JS
  document.getElementById('public-example').textContent =
product.title;
}
```

09 Step-by-step demonstration — product catalog API (continued) (continued)

This demonstrates calling another service and reading JSON — no API key needed for this public example.

[verify current] — Public API endpoints and availability change; the example is illustrative. Confirm against the provider's documentation before relying on it. [tool_comparison_framework.md, T]

09 Step-by-step demonstration — product catalog API (continued) (continued)

Step 5: Test locally and commit

09 Step-by-step demonstration — product catalog API (continued) (continued)

```
BASH
# Start the backend locally
node server.js

# In another terminal, test the endpoint
```

09 Step-by-step demonstration — product catalog API (continued) (continued) (continued)

BASH

```
curl http://localhost:3000/api/products
```

09 Step-by-step demonstration — product catalog API (continued) (continued)

BASH

```
# Commit
git add server.js public/index.html
git commit -m "feat: add GET /api/products product catalog"
```

09 Step-by-step demonstration — product catalog API (continued) (continued)

[verify current] — Port numbers, start commands, and backend structure depend on your stack; confirm against your editor and platform documentation. [tool_comparison_framework.md, T]

10 AI collaboration example (prompt format P1–P10)

AI COLLABORATION — PROMPT FORMAT P1–P10

PROMPT FORMAT **A reusable prompt you can copy**

10 AI collaboration example (prompt format P1–P10) (continued)

P1 **P1 · PROBLEM IT SOLVES**
Add a backend API endpoint safely, with no secrets and proper error handling.

10 AI collaboration example (prompt format P1–P10) (continued)

P2 **P2 · WHEN TO USE IT**
Chapter 10, when creating your first backend route.

10 AI collaboration example (prompt format P1–P10) (continued)

P3 **P3 · REQUIRED INPUTS**
Your project folder, the data model, and the frontend file that will consume the endpoint.

10 AI collaboration example (prompt format P1–P10) (continued)

P4

10 AI collaboration example (prompt format P1–P10) (continued)

P4 · THE PROMPT

10 AI collaboration example (prompt format P1–P10) (continued)

TEXT

```
Role: You are a careful backend developer.
Context: Mifugo Market (Chapter 10). Frontend at
public/index.html. Backend stack: Node.js/Express.
```

10 AI collaboration example (prompt format P1–P10) (continued)

TEXT

```
Task: Add a GET /api/products endpoint returning JSON product
data.
Data model: products = [{ id, name, price, stock_status }].
Constraints:
```

10 AI collaboration example (prompt format P1–P10) (continued)

```
TEXT
- No hardcoded secrets or API keys.
- Return JSON with appropriate status codes (200 success, 500
error).
```

10 AI collaboration example (prompt format P1–P10) (continued)

```
TEXT
- Error handling: return JSON error messages, not stack traces
to the user.
- Output format: unified diff with file paths.
```

10 AI collaboration example (prompt format P1–P10) (continued)

```
TEXT
- Do not modify the frontend in this step.
```

10 AI collaboration example (prompt format P1–P10) (continued)

P5 **P5 · WHAT A GOOD RESPONSE CONTAINS**
Unified diff; JSON route; status codes; error handling; no secrets; file paths included.

10 AI collaboration example (prompt format P1–P10) (continued)

P6 **P6 · WHAT THE READER MUST INSPECT**
Confirm no secrets; confirm status codes; confirm error handling; test the endpoint locally.

P7 **P7 · WEAK OR INCOMPLETE RESPONSE**
Hardcodes a key; returns HTML instead of JSON; no error handling; changes unrelated files.

P8 **P8 · FOLLOW-UP WHEN INADEQUATE**
"Remove the hardcoded key," "Return JSON not HTML," "Show only the server file diff."

P9 **P9 · SAFETY/SECURITY CONSIDERATIONS**
No secrets in code; errors should not leak stack traces to users (security misconfiguration, S-05). Production keys go in the platform dashboard (Chapter 14).

P10 **P10 · VERIFICATION STEP**
Run the backend locally; curl `GET /api/products`; confirm JSON returns with status 200; confirm no stack traces on error.

11 Human decisions and responsibilities (non-delegable judgment)

● YOUR DECISION — NOT THE AI'S

- **What data to store.** You decide what the application remembers (products, orders, users).
- **Where API keys go.** You ensure keys are in environment variables and headers, never in frontend code or URLs. [E-02, S-06]
- **What to show when the API fails.** You decide the user-facing error message.
- **Which data is public versus private.** You decide what the product catalog shows to anyone versus what requires login (Chapter 11).

[Human decision-point box] Before you connect a backend, answer: What data does my app need to remember? Where will API keys live (never in frontend code)? What should users see when the server fails? If any answer is no, decide first.

12 Inspect and verify AI output (backend/API checklist)

● QUALITY CHECK

Before accepting AI-generated backend code:

- ✓ [ ] No hardcoded secrets or API keys. [S-01, G-05, E-02]
- ✓ [ ] API keys sent in Authorization header, never in URL. [S-06]
- ✓ [ ] Routes return JSON with status codes (200, 404, 500).
- ✓ [ ] Error handling returns JSON messages, not stack traces. [S-05]
- ✓ [ ] Data model matches your specification (Chapter 5).
- ✓ [ ] Frontend fetch matches the backend endpoint path.
- ✓ [ ] Tested locally with curl or browser before committing.

12 Inspect and verify AI output (backend/API checklist) (continued)

[Related to E-01, E-02, G-05, S-05]

13 Failure modes and recovery

WHEN IT GOES WRONG

FAILURE	SYMPTOM	LIKELY CAUSE	RECOVERY
Backend won't start	"port already in use" or module not found	Wrong port; missing dependency	Check port; install missing package; correct start command
Endpoint returns HTML	Frontend expects JSON, gets HTML	Wrong Content-Type or route returns a page	Set `Content-Type: application/json`; return `res.json()`
Empty JSON array	Frontend shows nothing	No data or wrong route	Check database/data; confirm route path matches fetch URL
CORS error	Browser blocks frontend-to-backend request	Frontend and backend on different origins	Configure CORS on backend; or serve both from same origin
Stack trace leaked to user	Error page shows internal code	No error handling	Wrap routes in try/catch; return generic JSON error

All scenarios in this section are hypothetical and labeled as such.

14 Professional standard — quality threshold before connecting frontend and backend

PRO STANDARD

A backend endpoint is ready only when:

14 Professional standard — quality threshold before connecting frontend and backend (continued)

- It returns JSON with appropriate HTTP status codes. A-03
- Error handling returns user-safe messages, not stack traces. [S-05]
- No hardcoded secrets; keys in environment variables only. [S-01, E-02]

14 Professional standard — quality threshold before connecting frontend and backend (continued)

- It has been tested locally (curl or browser) before deploy.
- The frontend fetch path matches the backend route exactly.

15 Hands-on exercise (L3 — Application)

BUILD IT

Task: Add a backend `GET /api/products` endpoint to your project and display the product list on your frontend.

Constraints:

- No database writes yet (hardcoded product data is fine for now).
- No secrets in code.
- Test the endpoint locally before deploying.

Deliverable: A working backend endpoint and frontend product list.

Success criteria:

15 Hands-on exercise (L3 — Application) (continued)

- `GET /api/products` returns JSON with status 200.
- Frontend displays product names, prices, and stock status.
- Error handling returns JSON (not stack traces) on failure.
- No hardcoded secrets or API keys in any file.

Assessment level: L3 (Application — performs a procedure with guidance)

Time estimate: 100–120 minutes.

16 Build challenge (L4 — Analysis) with acceptance criteria

● PUSH FURTHER

Task: Call a public API from your backend, display one field on your frontend, and document where an API key would go (without using a real one).

Constraints:

- Use a public API that requires no key for this exercise.
- Do not hardcode any key.
- Document where a key would be stored if the API required one.

Acceptance criteria (Given–When–Then):

16 Build challenge (L4 — Analysis) with acceptance criteria (continued)

- **Given** a public API endpoint, **When** your backend calls it and returns JSON, **Then** your frontend displays at least one field from the response.
- **Given** the API is unreachable, **When** the frontend requests data, **Then** a user-friendly error message is shown (not a crash).
- **Given** an API key were required, **When** you review your code, **Then** you can state exactly where it would go (environment variable, Authorization header — never frontend code or URL).

Deliverable: A working public API call integrated into your frontend, plus a one-paragraph note on API-key placement.

Assessment level: L4 (Analysis — investigates evidence, diagnoses a failure)

Time estimate: 60–75 minutes.

17 Key takeaways

KEY TAKEAWAYS

- The backend receives requests, talks to a database, and returns responses; the frontend displays them.
- JSON is the language frontend and backend use to exchange data.
- REST endpoints are URLs like `GET /api/products`.
- API keys identify the application, not the user; they go in headers and environment variables, never in frontend code or URLs.
- Test endpoints locally before connecting the frontend.

18 Glossary of new terms

TERMS INTRODUCED IN THIS CHAPTER

18 Glossary of new terms (continued)

TERM	PLAIN-LANGUAGE DEFINITION	EXAMPLE	WHY IT MATTERS	TAG
Backend	Server-side code that receives requests and returns responses.	Node.js/Express server	Where application logic and data live.	
Database	Structured storage that persists data between restarts.	Products, orders, users	Remembers data so it is not lost on refresh.	
Data model	The shape of your data: tables, records, fields.	products: id, name, price, stock_status	Defines what your app can remember.	
Table / Record	A collection of items; one item in it.	products table; one product record	Organizes data into related groups.	
REST API	Web API style: clients send HTTP requests to endpoints.	`GET /api/products`	Standard way for frontend and backend to talk.	
API endpoint	A specific URL path for an action.	`/api/products`	The address the frontend calls for data.	
JSON	Text format for structured data.	`[{"name":"Sukuma wiki","price":40}]`	Both frontend and backend understand it.	
Request	Message from client to server.	`GET /api/products`	The ask: what the frontend wants.	
Response	Server's answer with status code and body.	`200 OK` + JSON	The answer: data or status.	

18 Glossary of new terms (continued) (continued)

TERM	PLAIN-LANGUAGE DEFINITION	EXAMPLE	WHY IT MATTERS	TAG
Status code	Three-digit result of a request.	200 OK, 404 Not Found, 500 Error	Tells the frontend whether it succeeded.	

18 Glossary of new terms (continued)

TERM	PLAIN-LANGUAGE DEFINITION	EXAMPLE	WHY IT MATTERS	TAG
API key	Value identifying the application making a request.	`sk_live_...`	Identifies the app, not the user (Chapter 11).	[security]
Environment variable	Value set outside code, read at runtime.	`API_KEY` in platform dashboard	Keeps secrets out of source code.	[security]
Frontend request	Browser asking the backend for something.	`fetch('/api/products')`	The client side of the conversation.	
Backend response	Server answering with data or status.	JSON product list	The server side of the conversation.	
Server-to-server	Backend calling another service.	Backend calls a public API	Where API keys properly belong.	[security]
Client-side	Code running in the browser.	HTML/JS order form	Secrets must never live here.	[security]

19 Knowledge-check questions

KNOWLEDGE CHECK

Attempt these before looking at the answers below.

Q1. What is JSON used for in a web application?

Q2. Where should an API key never be placed?

Q3. What is the difference between a frontend request and a backend response?

Q4. Name two HTTP status codes and what they mean.

Q5. True or false: A database is required for every application.

[PAGE BREAK — ANSWER KEY]

Answers

A1. JSON is used to exchange structured data between the frontend and the backend (for example, a product list sent from server to browser).

A2. Never in frontend code, never in a URL query string, and never committed to Git. It goes in an environment variable and is sent in an Authorization header [E-02, S-06].

A3. A frontend request is the browser asking the server for something; a backend response is the server's answer with data and a status code.

A4. Any two of: 200 OK (success), 404 Not Found (resource missing), 403 Forbidden (not allowed), 500 Internal Server Error (server problem) A-03 .

A5. False. Simple static sites do not need a database; applications that remember data (products, orders, users) do.

20 Links to earlier and later chapters

WHERE THIS SITS IN THE BOOK

Earlier chapters (prerequisites):

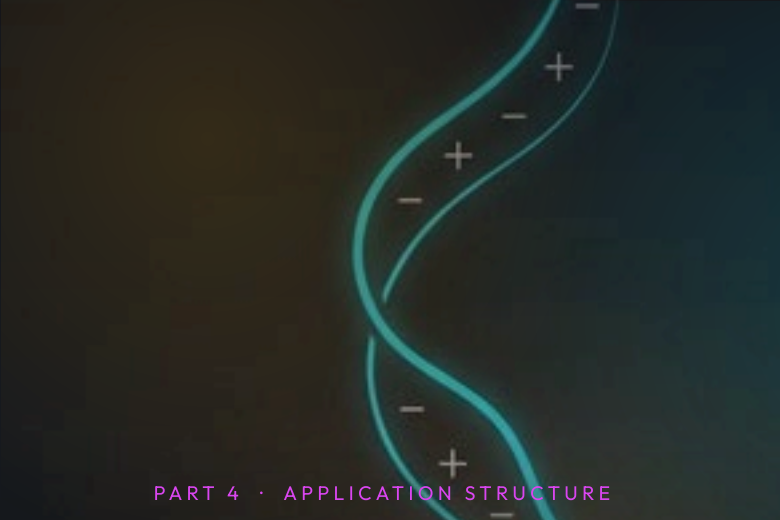
- **Chapter 1:** What Software Is — HTTP request/response is the foundation of backend communication.
- **Chapter 6:** Tools, Environments, and Your First Win — Git, deploy, and environment variables come from here.
- **Chapter 9:** Frontend — the page that now calls the backend endpoint.
- **Chapter 8:** Diff Discipline — backend code must pass diff review before acceptance.

Later chapters (where this is used next):

- **Chapter 11:** Access Control and Authentication — API endpoints get login and role-based access.
- **Chapter 13:** Security — input validation and secrets management for backend data.
- **Chapter 14:** Deployment — backend environment variables go in the platform dashboard.
- **Chapter 16:** Capstone — the full backend with products, orders, and users.

Evidence notes: Claims trace to A-03 (HTTP methods/status codes), E-01/E-02 (API auth and key placement), E-03 (authn vs authz), G-05 (secrets), S-01/S-05/S-06 (security practices). All scenarios are hypothetical.

Time estimate: Approximately 220 minutes total (100 minutes reading, 120 minutes hands-on practice).



PART 4 · APPLICATION STRUCTURE

11

Access Control, Authentication, and Secure Connections

01 Opening problem — "logged in means they can see everything"

SLOP DETECTOR — THE MISCONCEPTION

The scenario

Mifugo Market now has customer login. Wanjiku wants customers to see only their own orders, while she (the owner) sees all orders. She asks her AI assistant to add login. The AI generates sign-up and sign-in forms, and after login, customers can view any order by changing the order ID in the URL (for example, `?order_id=3` to view someone else's order). Wanjiku does not test this — she assumes "if they're logged in, they can see their own data."

A customer discovers they can view another customer's order by editing the URL. Private order details — phone numbers, addresses, order history — are exposed. This is broken access control, the most serious web application risk (OWASP A01:2021) G-01 · G-02 .

Authentication (who you are) is not the same as authorization (what you may do). Logging in does not mean you can see everything.

The misconception

01 Opening problem — "logged in means they can see everything" (continued)

MISCONCEPTION	WHY IT IS DANGEROUS	THE CORRECTION
"If they're logged in, they can see everything."	Users access other users' data by changing IDs in URLs (broken access control).	Authorize every request; validate ownership of each resource (object-level authorization) [E-03, E-04, S-04].

01 Opening problem — "logged in means they can see everything" (continued) (continued)

MISCONCEPTION	WHY IT IS DANGEROUS	THE CORRECTION
"JWTs are encrypted, so I can put user data in them."	JWTs are signed, not necessarily encrypted; anyone can decode the payload.	Never put secrets or sensitive data in a JWT payload [E-01, M-08].
"API keys identify the user."	API keys identify the application, not the person.	API keys for apps; OAuth/JWT/sessions for users [E-01, E-02, M-07].

02 Why this topic matters in real software projects

Authentication and authorization are the gatekeepers of your application. Get them wrong, and strangers can read customer data, place orders as other people, or take over accounts. Constitution §4 requires that readers implement sign-up/sign-in with password hashing and enforce object-level authorization — this chapter covers both.

OWASP A01 Broken Access Control is the top web application risk G-02 . A07 Identification and Authentication Failures is also in the Top 10 G-01 . These are not theoretical — they are the mistakes that cause real data breaches.

This chapter teaches you to separate identity from permission, hash passwords, use HTTPS, and keep secrets out of code.

03 Learning objectives

WHAT YOU WILL BE ABLE TO DO

By the end of this chapter, the reader can:

03 Learning objectives (continued)

- Distinguish authentication (who) from authorization (may do what).
- Implement sign-up/sign-in with password hashing (conceptual; AI-generated, reviewed per Chapter 8).
- Implement password reset flow conceptually.

03 Learning objectives (continued)

- Enforce object-level authorization (a user sees only their own orders).
- Explain why JWTs are signed, not necessarily encrypted.
- Set up HTTPS.

03 Learning objectives (continued)

- Store secrets in environment variables, never in code or URLs.

04 Prerequisites and what to skip

BEFORE YOU START

Prerequisites

Chapters 1–10. You should understand HTTP/HTTPS (Chapter 1), backend endpoints (Chapter 10), and diff review (Chapter 8).

What you can skip

Implementing cryptography from scratch. You use established libraries via AI, then review the output. Multi-factor authentication is mentioned as aspirational only. OAuth third-party login is mentioned conceptually; the book teaches email/password with hashing as the core pattern.

05 Plain-English explanation

Authentication: who are you?

Authentication verifies identity. When a customer signs up or signs in with email and password, the application checks: is this the person who owns this account? It is the first gate.

Authorization: what may you do?

Authorization checks whether the authenticated user has permission for a specific action. A logged-in customer may view their own orders but not another customer's. The owner may view all orders. Authorization happens on every request, on the server — never assumed from the frontend [S-04].

Password hashing: never store passwords in plaintext

When a customer creates a password, the application must not store it as-is. It runs the password through a **hashing function** (bcrypt or argon2) and stores only the hash. When the customer logs in, the application hashes the supplied password and compares it to the stored hash. If the database is breached, attackers cannot read the actual passwords [S-05, G-01].

HTTPS: encrypt data in transit

HTTPS (HTTP over TLS) encrypts data traveling between the browser and the server. Passwords, tokens, and personal data must never travel over plain HTTP. Render provides HTTPS automatically for deployed apps [A-01, G-01, S-02].

JWTs: signed, not necessarily encrypted

A **JWT** (JSON Web Token) is a stateless token that says "this user is authenticated." It is signed so the server can verify it was not tampered with, but the payload is typically readable by anyone. Never put passwords or sensitive data in a JWT payload [E-01, M-08].

Object-level authorization: validate ownership

05 Plain-English explanation (continued)

For every request to view or edit a resource (for example, an order), the backend checks: does the authenticated user own this resource? If order #5 belongs to customer A, customer B's request for order #5 returns 403 Forbidden, even though customer B is logged in [E-04, S-01].

Secrets in environment variables

Password hashing keys, session signing secrets, and database URLs live in environment variables on the platform dashboard — never in source code, never in URLs [G-05, H-03, S-01].

06 Mental model — and where it breaks down

Analogy: a club with a bouncer and room keys

Authentication is the bouncer at the door checking ID: are you on the guest list? Authorization is the room key: you may enter the lounge, but not the office. Your key only opens your own locker — not anyone else's.

Where the analogy breaks down

In software, an attacker does not need to be physically present. They can try thousands of passwords, guess URLs, or send forged requests. Authorization must be enforced on every single request, not just at the door.

07 Technical explanation with correct terminology

Each term is defined at first use below and listed in the chapter glossary (Section 18).

07 Technical explanation with correct terminology

(continued)

- **Authentication** — verifying who a user is (sign-up, sign-in, password check).
- **Authorization** — checking what an authenticated user may do for a specific action.
- **Sign-up / Sign-in** — creating an account and proving identity to access it.

07 Technical explanation with correct terminology

(continued)

- **Password hashing** — transforming a password into a stored hash using bcrypt or argon2; plaintext passwords are never stored [S-05].
- **Session / Token** — a server-issued value that remembers a logged-in user across requests.
- **JWT (JSON Web Token)** — a signed, stateless token representing authentication; payload is readable, signature verifies integrity E-01.

07 Technical explanation with correct terminology

(continued)

- **OAuth 2.0** — a delegation protocol for third-party access (for example, "sign in with Google"); scoped, time-limited tokens E-01.
- **Role-based access control (RBAC)** — permissions grouped by role (viewer, editor, owner) E-04.

07 Technical explanation with correct terminology

(continued)

- **Object-level authorization** — checking that the current user owns the specific resource they request (order #5 belongs to customer A) [E-04, S-01].
- **Least privilege** — granting only the permissions needed for each role [E-04, S-01].
- **HTTPS** — HTTP over TLS; encrypts data in transit [A-01, S-02].

07 Technical explanation with correct terminology

(continued)

- **Environment variable** — a value set outside code, read at runtime; holds secrets in production G-05 · H-03 .
- **403 Forbidden** — the status code returned when a user is authenticated but not authorized for a resource A-03 .

07 Technical explanation with correct terminology

(continued)

[security] — Authentication, Authorization, Password hashing, JWT, Object-level authorization, HTTPS, Environment variable, 403 Forbidden

08 Concrete example — Mifugo Market login and order privacy

IN PRACTICE

Mifugo Market now has:

08 Concrete example — Mifugo Market login and order privacy (continued)

- **Sign-up:** customers create an account with email and password. Passwords are hashed before storage.
- **Sign-in:** customers sign in and receive a session/token.
- **Customer view:** a customer sees only their own orders. `GET /orders/:id` returns 403 if the order belongs to another customer.

08 Concrete example — Mifugo Market login and order privacy (continued)

- **Owner view:** Wanjiku (owner role) sees all orders.
- **HTTPS:** the deployed app is served over TLS; passwords and tokens never travel over plain HTTP.
- **Secrets:** the session-signing key lives in Render's dashboard, not in code.

08 Concrete example — Mifugo Market login and order privacy (continued)

Wanjiku tests object-level authorization: she logs in as customer A, notes her order ID, then changes the URL to customer B's order ID. The server returns 403. The fix works.

09 Step-by-step demonstration — adding authentication and authorization

STEP BY STEP

Step 1: Require HTTPS

Confirm your deployment is served over HTTPS (Render provides this automatically). Never send passwords or tokens over plain HTTP [S-02].

Step 2: Generate sign-up with password hashing

[Action] Before the generated code can run, install the hashing library:

```bash
npm install bcrypt
```

(Or use `npm install argon2` if you prefer argon2 instead of bcrypt.)

Ask your AI assistant:

```text
Role: You are a careful backend developer focused on security.
Context: Mifugo Market (Chapter 11). Users table: id, name,
email, password_hash, role.
Task: Add a POST /signup route that hashes the password with
bcrypt/argon2 before storing.
Constraints:
- Never store plaintext passwords.
- Validate email format and password strength (min 8 chars).
- Return unified diff; no hardcoded secrets.
- Hashing uses an established library (bcrypt or argon2), not
custom crypto.
```

Review the diff (Chapter 8). Confirm:

09 Step-by-step demonstration — adding authentication and authorization (continued)

- Password is hashed before storage.
- No plaintext password appears anywhere.
- Email and password validation present.

Step 3: Generate sign-in

Add a POST /signin route that:

- Looks up the user by email.
- Compares the supplied password hash with the stored hash.
- Issues a session/token on success.
- Returns 401 on failure (without revealing whether the email exists).

Step 4: Enforce object-level authorization

Add a middleware that checks ownership on `GET /orders/:id`:

```
TEXT
Given a request for order #5 by customer A:
- Look up order #5.
- Check: does order #5.customer_id === current customer's id?
- If yes: return the order (200).
- If no: return 403 Forbidden.
```

Test it: log in as two different customers; confirm each sees only their own orders.

Step 5: Set up HTTPS and environment variables

- Confirm the deployed app uses HTTPS.
- Set the session-signing secret in Render's dashboard (environment variable), not in code `G-05 · H-03`.
- Confirm `.env` is in `.gitignore` and not committed.

09 Step-by-step demonstration — adding authentication and authorization (continued)

[verify current] — bcrypt/argon2 library names, JWT library behavior, and Render's HTTPS setup may change; confirm against current documentation. [tool_comparison_framework.md, T]

10 AI collaboration example (prompt format P1–P10)

AI COLLABORATION — PROMPT FORMAT P1–P10

PROMPT FORMAT **A reusable prompt you can copy**

P1 **P1 · PROBLEM IT SOLVES**
Add sign-up/sign-in with password hashing and object-level authorization without introducing security flaws.

P2 **P2 · WHEN TO USE IT**
Chapter 11, when adding authentication to Mifugo Market.

P3 **P3 · REQUIRED INPUTS**
Your backend file structure, the users table schema, and the orders route to protect.

10 AI collaboration example (prompt format P1–P10) (continued)

P4 **P4 · THE PROMPT**

```
TEXT
Role: You are a careful backend developer focused on
authentication and authorization.
Context: Mifugo Market (Chapter 11). Users: id, name,
email, password_hash, role. Orders: id, customer_id, ...
Task: Add POST /signup (password hashed with
bcrypt/argon2), POST /signin, and object-level
authorization on GET /orders/:id.
Constraints:
- Never store plaintext passwords; use bcrypt or argon2.
- Return 403 when a user requests another user's order.
- No hardcoded secrets; session secret from environment
variable.
- JWTs (if used) are signed, never contain sensitive
payload data.
- Return unified diffs with file paths.
- No stack traces leaked to users.
```

P5 **P5 · WHAT A GOOD RESPONSE CONTAINS**

Sign-up/sign-in routes; password hashing; 403 authorization check; env var for secrets; no hardcoded keys; unified diffs.

10 AI collaboration example (prompt format P1–P10) (continued)

P6 **P6 · WHAT THE READER MUST INSPECT**

Confirm hashing is used; confirm 403 check exists; confirm no plaintext passwords; confirm secrets are env vars; test 403 behavior.

10 AI collaboration example (prompt format P1–P10) (continued) (continued)

P7 **P7 · WEAK OR INCOMPLETE RESPONSE**
Stores plaintext passwords; no authorization check; hardcodes a secret; puts data in JWT payload; suggests disabling HTTPS.

10 AI collaboration example (prompt format P1–P10) (continued) (continued)

P8 **P8 · FOLLOW-UP WHEN INADEQUATE**
"Hash the password before storing," "Add the 403 ownership check," "Move the secret to an environment variable."

10 AI collaboration example (prompt format P1–P10) (continued) (continued)

P9 **P9 · SAFETY/SECURITY CONSIDERATIONS**
Authentication and authorization are security-critical paths requiring mandatory human review (Chapter 8, D-04). Review every line.

10 AI collaboration example (prompt format P1–P10) (continued) (continued)

P10 **P10 · VERIFICATION STEP**
Sign up, sign in, view your own order (200), try another customer's order ID (expect 403), confirm HTTPS on the deployed app.

11 Human decisions and responsibilities (non-delegable judgment)

YOUR DECISION — NOT THE AI'S

- **Which roles exist.** You decide: customer, owner, admin — and what each may do.
- **Whether password strength rules are enough.** You set the policy.
- **Whether to use JWTs or sessions.** You choose based on your architecture.
- **Whether an endpoint needs authorization.** You check every route that touches user data.
- **Whether to use third-party login (OAuth).** You weigh the complexity and data-sharing implications.

[Human decision-point box] Before enabling login, answer: Do I hash passwords? Do I check ownership on every order endpoint? Is HTTPS on? Are secrets in the dashboard, not code? If any answer is no, do not enable login yet.

12 Inspect and verify AI output (authentication checklist)

QUALITY CHECK

Before accepting AI-generated auth code:

12 Inspect and verify AI output (authentication checklist) (continued)

- ✓ [ ] Passwords are hashed (bcrypt/argon2), never plaintext. [S-05]
- ✓ [ ] Sign-up validates email and password strength.
- ✓ [ ] Sign-in returns 401 on failure without revealing whether the email exists.
- ✓ [ ] Every order/user data endpoint checks object-level authorization (403 for wrong user). [S-01, E-04]
- ✓ [ ] No hardcoded secrets or signing keys. [S-01, G-05]
- ✓ [ ] JWTs (if used) contain no sensitive payload data. [E-01, M-08]

12 Inspect and verify AI output (authentication checklist) (continued)

- ✓ [ ] HTTPS is enforced on the deployed app. [S-02]
- ✓ [ ] Test: logged-in customer A cannot view customer B's order.

12 Inspect and verify AI output (authentication checklist) (continued)

[Related to D-04, E-03, G-01, G-02]

13 Failure modes and recovery

● WHEN IT GOES WRONG

13 Failure modes and recovery (continued)

FAILURE	SYMPTOM	LIKELY CAUSE	RECOVERY
Passwords in plaintext	Database breach exposes all passwords	No hashing library used	Hash all passwords; force password reset; notify users

13 Failure modes and recovery (continued)

FAILURE	SYMPTOM	LIKELY CAUSE	RECOVERY
Broken object-level authorization	Customer A views customer B's order by URL ID	No ownership check on endpoint	Add ownership check; return 403; retest

13 Failure modes and recovery (continued)

FAILURE	SYMPTOM	LIKELY CAUSE	RECOVERY
Hardcoded session secret	Secret exposed in repository	Secret in source code	Rotate the secret immediately; move to env var [G-05, S-01]

13 Failure modes and recovery (continued)

FAILURE	SYMPTOM	LIKELY CAUSE	RECOVERY
JWT with sensitive payload	Anyone decodes user data from token	Data stored in JWT payload	Remove sensitive data from JWT; store server-side

13 Failure modes and recovery (continued)

FAILURE	SYMPTOM	LIKELY CAUSE	RECOVERY
No HTTPS	Passwords travel in plain text	Deployment not using TLS	Enable HTTPS (Render provides automatically); redeploy

13 Failure modes and recovery (continued)

FAILURE	SYMPTOM	LIKELY CAUSE	RECOVERY
Sign-in reveals email existence	Error differs for wrong email vs wrong password	Implementation detail	Return the same generic failure message for both

13 Failure modes and recovery (continued)

All scenarios in this section are hypothetical and labeled as such.

14 Professional standard — quality threshold before enabling login

PRO STANDARD

Authentication is ready only when:

- Passwords are hashed (bcrypt/argon2), never plaintext [S-05, G-01].
- Object-level authorization is enforced on every user-data endpoint (403 test passes) [S-01, E-04].
- HTTPS is enforced on the deployed app [S-02, A-01].
- No hardcoded secrets; signing keys in environment variables [S-01, G-05].
- JWTs (if used) contain no sensitive payload data [E-01, M-08].
- Sign-in failure messages do not leak whether an email exists.

15 Hands-on exercise (L3 — Application)

BUILD IT

Task: Add sign-up/sign-in with password hashing and object-level authorization to Mifugo Market (or your project).

Constraints:

- Use an established hashing library via AI-generated code, then review the diff (Chapter 8).
- No plaintext passwords anywhere.
- Test that a logged-in customer cannot view another customer's order (403).

Deliverable: Working sign-up/sign-in with authorization enforced.

Success criteria:

- Sign-up hashes the password before storage.
- Sign-in works and issues a session/token.
- `GET /orders/:id` returns 403 when the order belongs to another user.
- HTTPS is enabled on the deployed app.
- Secrets are in the platform dashboard, not in code.

Assessment level: L3 (Application — performs a procedure with guidance)

Time estimate: 100–120 minutes.

16 Build challenge (L4 — Analysis) with acceptance criteria

● PUSH FURTHER

Task: Test object-level authorization and document the result. Confirm JWTs (if used) contain no sensitive data. Confirm HTTPS is live.

Constraints:

- Do not ask the AI to regenerate the auth system.
- Make the smallest possible test: two customer accounts, two orders.
- Document findings in writing.

Acceptance criteria (Given–When–Then):

- **Given** two customers each with their own order, **When** customer A requests customer B's order ID, **Then** the server returns 403 Forbidden.
- **Given** a JWT is used, **When** its payload is decoded, **Then** it contains no password, secret, or sensitive personal data.
- **Given** the deployed app is checked, **Then** it is served over HTTPS (padlock in browser).
- **Given** secrets are reviewed, **Then** none appear in source code or Git history.

Deliverable: A written authorization test report plus confirmation of HTTPS and no secrets in code.

Assessment level: L4 (Analysis — investigates evidence, diagnoses a failure)

Time estimate: 45–60 minutes.

17 Key takeaways

KEY TAKEAWAYS

- Authentication is who you are; authorization is what you may do. They are separate.
- Passwords are hashed, never stored plaintext.
- Every request to user data must check ownership (object-level authorization).
- JWTs are signed, not encrypted; never put secrets in a JWT payload.
- HTTPS is non-negotiable; secrets live in environment variables, never in code.
- Auth and crypto are security-critical paths: human review is mandatory.

18 Glossary of new terms

TERMS INTRODUCED IN THIS CHAPTER

18 Glossary of new terms (continued)

TERM	PLAIN-LANGUAGE DEFINITION	EXAMPLE	WHY IT MATTERS	TAG
Authentication	Verifying who a user is.	Sign-in with email and password	The first gate: are you who you say?	[security]
Authorization	Checking what an authenticated user may do.	Customer views own orders only	The second gate: are you allowed?	[security]
Sign-up / Sign-in	Creating an account and proving identity.	Register with email; log in	How customers access their own data.	[security]

18 Glossary of new terms (continued) (continued)

TERM	PLAIN-LANGUAGE DEFINITION	EXAMPLE	WHY IT MATTERS	TAG
Password hashing	Transforming a password into a stored hash.	bcrypt/argon2 hash stored, not plaintext	Protects passwords if the database is breached.	[security]
Session / Token	A value remembering a logged-in user across requests.	Cookie or JWT after sign-in	Lets the user stay logged in between pages.	[security]

18 Glossary of new terms (continued) (continued)

TERM	PLAIN-LANGUAGE DEFINITION	EXAMPLE	WHY IT MATTERS	TAG
JWT	A signed, stateless authentication token.	`eyJhbGciOiJIUzI1NiJ9...`	Verifies identity without storing sessions server-side.	[security]

18 Glossary of new terms (continued)

TERM	PLAIN-LANGUAGE DEFINITION	EXAMPLE	WHY IT MATTERS	TAG
OAuth 2.0	A delegation protocol for third-party access.	"Sign in with Google"	Lets users log in via another service.	[security]
RBAC	Permissions grouped by role (viewer, editor, owner).	Owner sees all orders; customer sees own	Limits what each role can do.	[security]

18 Glossary of new terms (continued) (continued)

TERM	PLAIN-LANGUAGE DEFINITION	EXAMPLE	WHY IT MATTERS	TAG
Object-level authorization	Checking ownership of the specific resource requested.	Order #5 belongs to customer A; customer B gets 403	Prevents users viewing each other's data.	[security]
Least privilege	Granting only the permissions needed.	Customer cannot access admin routes	Limits damage from a compromised account.	[security]

18 Glossary of new terms (continued) (continued)

TERM	PLAIN-LANGUAGE DEFINITION	EXAMPLE	WHY IT MATTERS	TAG
HTTPS	HTTP over TLS; encrypts data in transit.	`https://mifugomarket.co.ke`	Protects passwords and tokens from eavesdropping.	[security]

18 Glossary of new terms (continued) (continued)

TERM	PLAIN-LANGUAGE DEFINITION	EXAMPLE	WHY IT MATTERS	TAG
Environment variable	A value set outside code, read at runtime.	`SESSION_SECRET` in Render dashboard	Keeps secrets out of source code.	[security]

18 Glossary of new terms (continued) (continued) (continued)

TERM	PLAIN-LANGUAGE DEFINITION	EXAMPLE	WHY IT MATTERS	TAG
403 Forbidden	Status code: authenticated but not authorized.	Customer B requests customer A's order	Tells the user they are logged in but not allowed.	[security]

19 Knowledge-check questions

KNOWLEDGE CHECK

Attempt these before looking at the answers below.

Q1. What is the difference between authentication and authorization?

Q2. Why should passwords never be stored in plaintext?

Q3. What status code should a server return when a logged-in user requests another user's order?

Q4. True or false: JWTs are encrypted, so it is safe to put user data in them.

Q5. Where should the session-signing secret live in production?

[PAGE BREAK — ANSWER KEY]

Answers

A1. Authentication verifies identity (who you are); authorization checks permission for a specific action (what you may do).

A2. If the database is breached, plaintext passwords are immediately usable. Hashing (bcrypt/argon2) means attackers only get hashes, not the actual passwords [S-05].

A3. 403 Forbidden — the user is authenticated but not authorized for that resource [A-03, S-01].

19 Knowledge-check questions (continued)

A4. False. JWTs are signed, not necessarily encrypted; the payload is readable by anyone. Never put secrets in a JWT payload [E-01, M-08].

A5. In the production environment variable dashboard (for example, Render's dashboard), never in source code or Git [G-05, H-03, S-01].

20 Links to earlier and later chapters

WHERE THIS SITS IN THE BOOK

Earlier chapters (prerequisites):

- **Chapter 1:** What Software Is — HTTP and HTTPS foundations.
- **Chapter 8:** The Diff Discipline — auth code must pass diff review before acceptance.
- **Chapter 10:** Backend, Data, and APIs — the endpoints that now need authorization.

Later chapters (where this is used next):

- **Chapter 12:** Testing — tests for login and authorization behavior.
- **Chapter 13:** Security, Privacy, and Data Protection — input validation and secrets apply to auth data.
- **Chapter 14:** Deployment — HTTPS and environment variables in production.
- **Chapter 16:** Capstone — authenticated marketplace with role-based views.

Evidence notes: Claims trace to E-01/E-02/E-03/E-04 (API auth, authn vs authz, least privilege), G-01/G-02 (OWASP A01/A07), G-05 (secrets), S-01–S-06 (security practices), M-07–M-09 (misconceptions). All scenarios are hypothetical.

20 Links to earlier and later chapters (continued)

Time estimate: Approximately 220 minutes total (100 minutes reading, 120 minutes hands-on practice).

PART 5

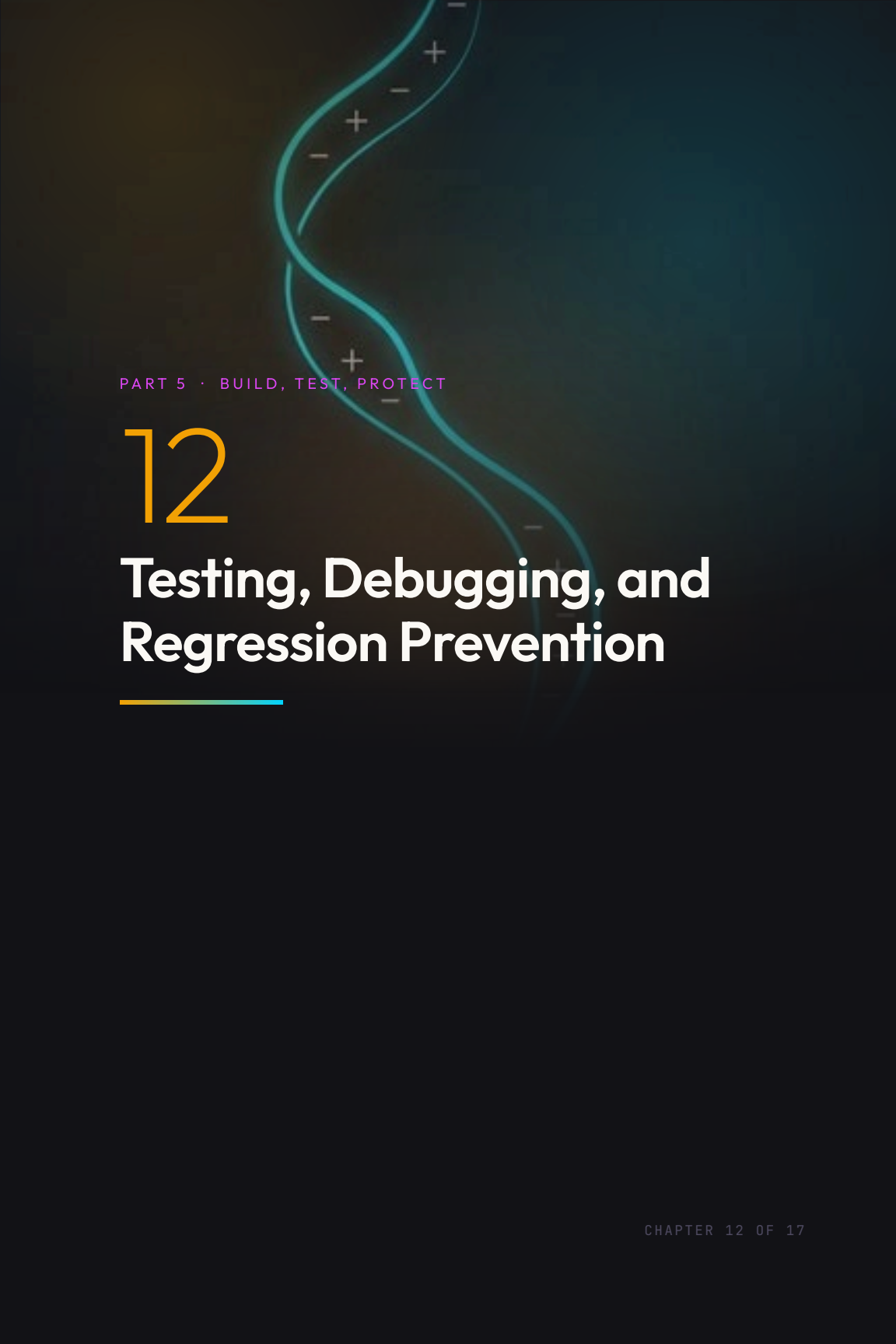
5

Build, Test, Protect

Tests, debugging, and data protection

12 Testing, Debugging, and Regression Prevention

13 Security, Privacy, and Data Protection



PART 5 · BUILD, TEST, PROTECT

12

Testing, Debugging, and Regression Prevention

01 Opening problem — "if it works once, it keeps working"

SLOP DETECTOR — THE MISCONCEPTION

The scenario

Mifugo Market now has a product catalog and customer login. Wanjiku asks her AI assistant to add a discount feature: "10% off orders over KSh 1,000." The AI generates the discount logic and it works when she tries it once. She commits and deploys.

The next week, a customer orders KSh 1,200 of produce and is charged full price. Another customer gets a discount on a KSh 900 order. The discount sometimes works and sometimes does not — and Wanjiku has no tests to tell her why. She considers asking the AI to regenerate the whole application, but she does not know what changed or how to go back.

She never wrote a test, never defined regression, and never verified after the AI edit. That is how working software silently breaks.

The misconception

MISCONCEPTION	WHY IT IS DANGEROUS	THE CORRECTION
"If it works once, it keeps working."	Software breaks when data changes, AI edits touch unrelated code, or environments differ. No tests means you discover failure from users, not from your own checks.	Write tests for critical paths; run them before every deploy.
"Testing is only for professional teams with QA departments."	Even a solo builder needs to know the price calculation is correct before charging real customers.	A few unit tests and one smoke test are enough to start.

02 Why this topic matters in real software projects

Testing is the safety net that lets you trust AI-generated code. Because AI can introduce subtle errors (wrong logic, missing validation, regressions), you need observable proof that your application still works after every change.

Constitution §4 lists testing as a required competency: the reader can write and run tests for critical paths and explain what regression is. This chapter teaches unit tests, integration tests, end-to-end smoke tests, the test pyramid, and how to debug by gathering evidence rather than guessing.

Without testing, every AI edit is a gamble. With testing, you know what works — and you can prove it.

03 Learning objectives

WHAT YOU WILL BE ABLE TO DO

By the end of this chapter, the reader can:

- Write unit tests for one critical path (for example, price calculation, stock check).
- Write one end-to-end smoke test (for example, place an order; homepage loads).
- Explain the test pyramid (~70/20/10) and why more unit than E2E tests.
- Define regression and give an example.
- Run tests before every deploy.
- Debug a failing test by reading the error message and reproducing the issue.

04 Prerequisites and what to skip

BEFORE YOU START

Prerequisites

Chapters 1–11. You should have a working application with a price calculation (or similar logic), a backend, and a Git repository.

What you can skip

Complex testing frameworks and advanced E2E tooling. One simple test runner is enough to start (Chapter 11 addresses tool selection lightly; frameworks change, categories are stable [R-11]).

05 Plain-English explanation

A test checks one thing and reports pass or fail

A **test** is a small program that runs a piece of your application and checks whether the result matches what you expected. If it matches, the test passes. If not, it fails and tells you what went wrong.

Unit tests check small pieces in isolation

A **unit test** checks one function or calculation by itself — for example, "given a subtotal of KSh 1,200 and a 10% discount rule, the total is KSh 1,080." Unit tests are fast, cheap, and numerous.

Integration tests check pieces working together

An **integration test** checks that two or more parts work together — for example, the frontend form sends data to the backend and the backend saves it.

End-to-end (E2E) and smoke tests check the whole flow

An **end-to-end test** checks the complete user journey — for example, open the homepage, add an item to cart, and place an order. A **smoke test** is a lightweight E2E check that the critical functions work (homepage loads, order form submits) after a build or deploy F-03.

05 Plain-English explanation (continued)

The test pyramid: more unit, fewer E2E

The **test pyramid** recommends roughly 70% unit tests, 20% integration tests, 10% end-to-end tests F-01. Unit tests are fast and cheap; E2E tests are slow and brittle. Most of your testing effort should be at the bottom of the pyramid.

Regression: when a fix breaks something that worked

A **regression** is when a change breaks previously working functionality F-02. Regression testing means re-running existing tests after every change to confirm nothing broke. Regression is a purpose, not a separate test level — it applies at unit, integration, and E2E levels.

06 Mental model — and where it breaks down

Analogy: checking each ingredient before cooking

Think of tests like checking each ingredient before cooking a meal: you confirm the produce is fresh (unit test), the stove works (integration test), and you taste the final dish (smoke test). If you skip these, you serve a bad meal and only find out when the customer complains.

Where the analogy breaks down

Unlike food, software can be re-tested instantly and automatically after every change. A test suite run in seconds gives you confidence that a new AI edit did not break the price calculation or the checkout flow.

07 Technical explanation with correct terminology

Each term is defined at first use below and listed in the chapter glossary (Section 18).

- **Test** — a small program that runs code and checks whether the result matches expectations; reports pass or fail.
- **Unit test** — a test of one function or calculation in isolation.
- **Integration test** — a test of two or more parts working together.
- **End-to-end (E2E) test** — a test of the complete user journey from start to finish.
- **Smoke test** — a lightweight check that critical functions work after a build or deploy F-03.
- **Test pyramid** — the recommended distribution: ~70% unit, ~20% integration, ~10% E2E F-01.
- **Regression** — a change that breaks previously working functionality F-02.
- **Regression testing** — re-running existing tests after a change to confirm nothing broke; a purpose, not a level F-02.
- **Pass / fail** — the result of a test.
- **Assertion** — a statement in a test that checks a condition (for example, "total should equal 1080").
- **Error message** — the output when a test fails, describing what went wrong.
- **Reproduce** — making a bug happen reliably before trying to fix it.
- **Stack trace** — a report of the function calls leading to an error, used in debugging.
- **Test runner** — a tool that executes your tests and reports results.

[security] — none directly (testing chapter; security testing is Chapter 13)

08 Concrete example — Mifugo Market price calculation

IN PRACTICE

Wanjiku's Mifugo Market applies a 10% discount on orders over KSh 1,000. She writes unit tests:

1. **Subtotal KSh 500, no discount:** total = KSh 500.
2. **Subtotal KSh 1,000, no discount** (the rule is "over KSh 1,000," not "KSh 1,000 or more"): total = KSh 1,000.
3. **Subtotal KSh 1,200, 10% discount:** total = KSh 1,080.
4. **Out-of-stock item cannot be ordered:** the order is rejected with a clear message.

She also writes one smoke test: open the homepage, confirm at least one product appears.

She runs the tests before every change. When the AI edits the discount logic, she re-runs them — if a test turns red, she knows immediately and reverts or fixes.

09 Step-by-step demonstration — writing and running tests

STEP BY STEP

Step 1: Identify one critical path

Pick the one calculation or flow that, if wrong, would cost money or lose trust. For Mifugo Market: the price calculation with discount.

Step 2: Write unit tests

Using your test runner (for example, Node.js with a built-in or simple test framework), write tests like:

09 Step-by-step demonstration — writing and running tests (continued)

```
JS
// Example structure (framework-agnostic)
test('no discount under KSh 1000', () ⇒ {
  const result = calculateTotal(500, 0.10, 1000);
  assert.equal(result, 500);
});

test('no discount exactly at KSh 1000', () ⇒ {
  const result = calculateTotal(1000, 0.10, 1000);
  assert.equal(result, 1000);
});
```

09 Step-by-step demonstration — writing and running tests (continued)

```
JS

test('10% discount over KSh 1000', () ⇒ {
  const result = calculateTotal(1200, 0.10, 1000);
  assert.equal(result, 1080);
});

test('out of stock cannot be ordered', () ⇒ {
  const result = placeOrder({ itemId: 3, quantity: 1 }); //
avocados, sold out
  assert.equal(result.status, 'rejected');
```

09 Step-by-step demonstration — writing and running tests (continued)

```
JS
  assert.equal(result.message, 'Item out of stock');
});
```

09 Step-by-step demonstration — writing and running tests (continued)

[verify current] — Test framework names, syntax, and assertions change (Jest, Vitest, Node's built-in test runner, etc.). The testing categories and pyramid are stable; the tools are "verify current" on the companion website [R-11, tool_comparison_framework.md, T].

Step 3: Run the tests

```
BASH
npm test
```

Confirm all tests pass. If any fail, fix the code before proceeding.

Step 4: Introduce a regression intentionally

Change the discount rule (for example, "over KSh 1,000" to "KSh 1,000 or more") and run the tests again. Test 2 should now fail — this proves your tests catch regressions. Revert the change and confirm tests pass again.

Step 5: Write one smoke test

A lightweight check that the critical flow works:

09 Step-by-step demonstration — writing and running tests (continued) (continued)

```JS
test('homepage loads and shows products', async () => {
  const response = await fetch('http://localhost:3000/');
  assert.equal(response.status, 200);
  const html = await response.text();
  assert.ok(html.includes('Sukuma wiki'));
});
```

Step 6: Run tests before every deploy

09 Step-by-step demonstration — writing and running tests (continued)

Make it a habit: `git commit` → `npm test` → all pass → `git push` → deploy. No deploy on red.

10 AI collaboration example (prompt format P1–P10)

AI COLLABORATION — PROMPT FORMAT P1–P10

PROMPT FORMAT **A reusable prompt you can copy**

P1 **P1 · PROBLEM IT SOLVES**
Generate tests for a critical path without introducing slop.

P2 **P2 · WHEN TO USE IT**
Chapter 12, when adding tests to Mifugo Market.

P3 **P3 · REQUIRED INPUTS**
The function or route to test, the expected behavior, and your test runner setup.

10 AI collaboration example (prompt format P1–P10) (continued)

P4

10 AI collaboration example (prompt format P1–P10) (continued)

P4 · THE PROMPT

10 AI collaboration example (prompt format P1–P10) (continued)

TEXT

```
Role: You are a careful test engineer.
Context: Mifugo Market (Chapter 12). Function:
calculateTotal(subtotal, discountRate, threshold).
Task: Write unit tests for the discount logic: no discount under
threshold, no discount exactly at threshold, discount applied
above threshold, out-of-stock rejection.
Constraints:
- Use the project's test runner.
- Each test checks one behavior.
- Tests are independent (do not depend on each other).
```

10 AI collaboration example (prompt format P1–P10) (continued)

```
TEXT
- No hardcoded secrets or test data that looks like real
customer data.
- Return test code in a code block with language tag.
```

10 AI collaboration example (prompt format P1–P10) (continued)

P5 **P5 · WHAT A GOOD RESPONSE CONTAINS**
Independent unit tests; clear assertions; edge cases (exactly at threshold); no real customer data.

P6 **P6 · WHAT THE READER MUST INSPECT**
Run the tests; confirm they pass; confirm each test checks one behavior; confirm tests are independent.

10 AI collaboration example (prompt format P1–P10) (continued)

P7 **P7 · WEAK OR INCOMPLETE RESPONSE**
Tests that depend on each other; only happy-path tests (no edge cases); hardcoded real-looking customer data; no assertions.

P8 **P8 · FOLLOW-UP WHEN INADEQUATE**
"Add an edge case at exactly the threshold," "Make tests independent," "Add an assertion for the error message."

10 AI collaboration example (prompt format P1–P10) (continued) (continued)

P9 **P9 · SAFETY/SECURITY CONSIDERATIONS**
Do not use real customer data in tests; use fabricated examples. Tests should not write to production data.

P10 **P10 · VERIFICATION STEP**
Run `npm test`; all tests pass; intentionally break the discount logic and confirm a test fails.

11 Human decisions and responsibilities (non-delegable judgment)

YOUR DECISION — NOT THE AI'S

- **Which paths are critical.** You decide what to test first (price, checkout, login).
- **What counts as "pass."** You define expected results; the AI may guess wrong.
- **When to run tests.** You run them before every deploy, without exception.
- **Whether to fix or revert on failure.** You decide based on the evidence.

[Human decision-point box] Before every deploy, answer: Have the tests run and passed? Do I have tests for the critical path? If any answer is no, do not deploy.

12 Inspect and verify AI output (testing checklist)

QUALITY CHECK

Before accepting AI-generated tests or code:

- ✓ [ ] Tests cover the critical path (price calculation, checkout, login).
- ✓ [ ] Tests include edge cases (exactly at threshold, out of stock, empty cart).
- ✓ [ ] Tests are independent (one test's result does not affect another).
- ✓ [ ] Tests run and pass locally before committing.
- ✓ [ ] No real customer data in test data.
- ✓ [ ] Test failures produce clear error messages.

[Related to F-01, F-02, F-03, Q-01, Q-02, Q-03]

13 Failure modes and recovery

WHEN IT GOES WRONG

FAILURE	SYMPTOM	LIKELY CAUSE	RECOVERY
Test passes but feature broken	Users report a bug that tests missed	Tests only covered happy path	Add edge-case tests; test failure modes
Flaky test	Test passes sometimes, fails others	Test depends on timing or external service	Make test deterministic; mock external calls
Regression after AI edit	Old feature broke after new feature	AI change had side effects	Revert the commit; re-run tests; re-implement narrowly F-06
Test depends on another test	Tests pass together, fail individually	Shared state between tests	Isolate tests; each sets up its own data
No tests for error paths	Feature fails on invalid input	Only happy-path tested	Add tests for invalid input and error responses

All scenarios in this section are hypothetical and labeled as such.

14 Professional standard — quality threshold before deploying

PRO STANDARD

Code is deployable only when:

- **Tests written for critical paths:** unit tests for calculations, one smoke test for the main flow F-01 · F-03 .
- **Test pyramid respected:** mostly unit tests, few E2E F-01 .
- **Tests run before every deploy:** no deploy on red [S-08, Q-02].
- **Regression tested:** existing tests re-run after each change F-02 .
- **Tests are independent and deterministic** [Q-02].

15 Hands-on exercise (L3 — Application)

BUILD IT

Task: Write unit tests for one critical path in your project (price calculation, stock check, or similar) plus one smoke test.

Constraints:

- No database writes needed (use hardcoded data if necessary).
- Tests must be independent.
- Run the tests locally before committing.

Deliverable: A test file with 3+ unit tests and 1 smoke test, all passing.

Success criteria:

- 3+ unit tests covering normal and edge cases.
- 1 smoke test checking a critical flow.
- All tests pass (`npm test` green).
- Tests are independent of each other.

15 Hands-on exercise (L3 — Application) (continued)

Assessment level: L3 (Application — performs a procedure with guidance)

Time estimate: 90–110 minutes.

16 Build challenge (L4 — Analysis) with acceptance criteria

● PUSH FURTHER

Task: Intentionally break one feature, confirm a test catches it, fix it, and confirm the test passes again (regression demonstration).

Constraints:

- Do not ask the AI to regenerate the app.
- Make the smallest possible breaking change.
- Document the before/after test results.

Acceptance criteria (Given–When–Then):

- **Given** a working feature with passing tests, **When** a breaking change is introduced, **Then** at least one test fails (red).
- **Given** a failing test, **When** the change is reverted or fixed, **Then** the test passes again (green).
- **Given** the full test suite is run after the fix, **Then** all tests pass (no regression elsewhere). F-02

Deliverable: A written regression test report showing the failing test, the fix, and the passing result.

Assessment level: L4 (Analysis — investigates evidence, diagnoses a failure)

Time estimate: 45–60 minutes.

17 Key takeaways

KEY TAKEAWAYS

- Tests give you observable proof that your application works — especially important with AI-generated code.
- The test pyramid: ~70% unit, ~20% integration, ~10% E2E; start with unit tests.
- Regression is when a change breaks something that worked; re-run tests after every change.
- No deploy on red: tests must pass before every deploy.
- Debug by reproducing the issue and reading the error message, not by guessing.

18 Glossary of new terms

TERMS INTRODUCED IN THIS CHAPTER

18 Glossary of new terms (continued)

TERM	PLAIN-LANGUAGE DEFINITION	EXAMPLE	WHY IT MATTERS	TAG
Test	A small program that runs code and checks whether the result matches expectations.	"Price of KSh 1,200 with 10% discount = KSh 1,080"	Gives you proof the code works, not just hope.	
Unit test	A test of one function or calculation in isolation.	Testing calculateTotal() alone	Fast, cheap, and numerous; the base of the pyramid.	
Integration test	A test of two or more parts working together.	Frontend form + backend endpoint	Confirms pieces connect correctly.	

18 Glossary of new terms (continued) (continued)

TERM	PLAIN-LANGUAGE DEFINITION	EXAMPLE	WHY IT MATTERS	TAG
End-to-end (E2E) test	A test of the complete user journey.	Open homepage → add to cart → place order	Confirms the whole flow works for a real user.	
Smoke test	A lightweight check that critical functions work after build/deploy.	Homepage loads; order form submits	Quick confidence check after every change.	F-03
Test pyramid	Recommended distribution: ~70% unit, ~20% integration, ~10% E2E.	Most tests are unit tests	Unit tests are fast; E2E tests are slow and brittle.	F-01
Regression	A change that breaks previously working functionality.	Discount broke after adding a new feature	The reason you re-run tests after every change.	F-02

18 Glossary of new terms (continued) (continued)

TERM	PLAIN-LANGUAGE DEFINITION	EXAMPLE	WHY IT MATTERS	TAG
Regression testing	Re-running existing tests after a change to confirm nothing broke.	Running all tests after an AI edit	A purpose applied at every test level, not a separate level.	F-02
Pass / fail	The result of a test.	5 tests pass, 1 fails	Tells you whether to proceed or fix.	

18 Glossary of new terms (continued)

TERM	PLAIN-LANGUAGE DEFINITION	EXAMPLE	WHY IT MATTERS	TAG
Assertion	A statement in a test checking a condition.	`assert.equal(result, 1080)`	The actual check the test performs.	
Error message	Output when a test fails, describing what went wrong.	"Expected 1080, got 1200"	Tells you what to fix.	
Reproduce	Making a bug happen reliably before fixing it.	"Add KSh 1,200 to cart and observe discount not applied"	You cannot fix what you cannot reliably trigger.	
Stack trace	A report of function calls leading to an error.	A list of files and line numbers	Helps locate where the error started.	
Test runner	A tool that executes tests and reports results.	`npm test`	Runs your test suite and reports pass/fail.	

19 Knowledge-check questions

KNOWLEDGE CHECK

Attempt these before looking at the answers below.

Q1. What is the test pyramid distribution?

Q2. What is regression, and why do you re-run tests after every change?

Q3. Name the three main test types this chapter teaches.

Q4. True or false: E2E tests should be the majority of your tests.

19 Knowledge-check questions (continued)

Q5. What does "no deploy on red" mean?

[PAGE BREAK — ANSWER KEY]

Answers

A1. Approximately 70% unit, 20% integration, 10% end-to-end F-01.

19 Knowledge-check questions (continued)

A2. Regression is when a change breaks previously working functionality. You re-run tests after every change to confirm nothing broke F-02.

A3. Unit tests, integration tests, and end-to-end tests (plus smoke tests as a lightweight E2E check).

A4. False. E2E tests are the smallest portion (~10%); they are slow and brittle. Most tests should be unit tests.

19 Knowledge-check questions (continued)

A5. "No deploy on red" means you never deploy when tests are failing; all tests must pass before you push to production [S-08].

20 Links to earlier and later chapters

WHERE THIS SITS IN THE BOOK

Earlier chapters (prerequisites):

20 Links to earlier and later chapters (continued)

- **Chapter 10:** Backend, Data, and APIs — the endpoints you now test.
- **Chapter 8:** The Diff Discipline — every change is reviewed before tests run.
- **Chapter 11:** Access Control — login and authorization behavior should be tested.

Later chapters (where this is used next):

20 Links to earlier and later chapters (continued)

- **Chapter 13:** Security, Privacy, and Data Protection — security checks join the pre-deploy gate alongside tests.
- **Chapter 14:** Version Control, Deployment, and Operations — tests run before every deploy; rollback if tests fail in production.

20 Links to earlier and later chapters (continued)

- **Chapter 16:** Capstone — the test suite (3+ unit tests + 1 smoke test) is a required deliverable.

20 Links to earlier and later chapters (continued)

Evidence notes: Claims trace to F-01 (Martin Fowler test pyramid), F-02 (regression testing), F-03 (smoke tests), Q-01–Q-03 (quality practices). All scenarios are hypothetical.

Time estimate: Approximately 200 minutes total (90 minutes reading, 110 minutes hands-on practice).

PART 5 · BUILD, TEST, PROTECT

13

Security, Privacy, and Data Protection

01 Opening problem — "HTTPS makes my app secure"

SLOP DETECTOR — THE MISCONCEPTION

The scenario

Mifugo Market is live with login, orders, and a backend. Wanjiku has HTTPS enabled (the padlock shows in the browser) and believes her application is secure. She has not validated any user input: the search field accepts anything, including text that looks like database commands. She has not set up authorization checks on her API routes beyond login. She copied a privacy policy from another website but collects more data than she needs and has no process for deleting a customer's data when asked.

One day, a customer's order details are exposed because anyone can guess an order ID in the URL and view it. Wanjiku thought HTTPS was enough — but HTTPS only protects data in transit. It does not fix broken access control, injection, or excessive data collection.

The misconceptions

MISCONCEPTION	WHY IT IS DANGEROUS	THE CORRECTION
"HTTPS makes my app secure."	HTTPS encrypts data in transit; it does not stop access-control flaws, injection, or insecure design.	HTTPS is necessary but not sufficient; you also need authorization, validation, and secure design [G-02, G-03, G-04, M-06].
"Input validation is optional for internal tools."	Internal tools are still attacked; unvalidated input leads to injection and data corruption.	Validate and sanitize all input, internal or external [G-03, S-03, M-10].
"A privacy policy covers everything."	Copying a policy does not implement data minimization, consent, or erasure.	Data minimization, consent, and erasure are operational practices [G-06, G-07, M-11].

02 Why this topic matters in real software projects

Security and privacy are not features you add at the end — they are conditions of being allowed to serve real users. A single exposed customer record, one injected database command, or one unauthorized data collection can destroy trust and expose you to legal liability under the Kenya Data Protection Act 2019 G-06 and GDPR-informed principles G-07.

OWASP Top 10:2021 lists the most critical web application risks G-01. For a small application like Mifugo Market, the most relevant are broken access control (A01), injection (A03), insecure design (A04), cryptographic failures (A02), and security misconfiguration (A05).

This chapter teaches you to validate input, manage secrets, apply a pre-deploy security checklist, and handle personal data responsibly under Kenya DPA and GDPR-aware principles.

03 Learning objectives

WHAT YOU WILL BE ABLE TO DO

By the end of this chapter, the reader can:

- Name the OWASP Top 10:2021 risks most relevant to a small web app (broken access control, injection, insecure design, cryptographic failures, security misconfiguration).
- Validate and sanitize all user input.
- State what data Mifugo Market collects and why (data minimization).
- Describe how to honor an erasure request conceptually.
- Run a pre-deploy security checklist (secrets not in code, HTTPS on, authorization on every route, inputs validated).

04 Prerequisites and what to skip

BEFORE YOU START

Prerequisites

Chapters 1–12. You should have a working application with authentication (Chapter 11), a backend (Chapter 10), and tests (Chapter 12).

What you can skip

Implementing cryptography from scratch. No jurisdiction-specific legal advice — Kenya DPA and GDPR are covered as awareness, not legal counsel (Constitution §6). Readers must consult a qualified attorney for their jurisdiction.

05 Plain-English explanation

Treat all user input as untrusted

Every piece of data that comes from a user — a form field, a search query, a URL parameter — could be malicious or malformed. **Input validation** checks that the data meets your rules before you use it. **Sanitization** cleans or neutralizes dangerous content. Validation happens on the server; frontend validation helps users but can be bypassed [S-03, G-03].

The OWASP Top 10 risks that matter most

For a small web app, the most relevant risks from OWASP Top 10:2021 G-01 are:

05 Plain-English explanation (continued)

- **A01 Broken Access Control:** users access data or actions they should not (for example, viewing another customer's order by changing a URL ID). G-02
- **A02 Cryptographic Failures:** sending passwords or tokens over HTTP; storing passwords in plaintext. G-01
- **A03 Injection:** untrusted data interpreted as commands (SQL injection, command injection, XSS). G-03
- **A04 Insecure Design:** security was never designed in; auth added as an afterthought. G-04
- **A05 Security Misconfiguration:** debug mode left on in production; default passwords; open ports. G-01

Data minimization: collect only what you need

Collect only the personal data necessary for your stated purpose, and delete it when you no longer need it. If you do not need a customer's ID number, do not ask for it. Less data means less risk if something goes wrong [G-06, S-12].

Consent and erasure

05 Plain-English explanation (continued)

You must obtain express consent before processing personal data, and customers can withdraw consent at any time. If a customer asks you to delete their data (the **right to erasure** or "right to be forgotten"), you should do so without undue delay, subject to legal exceptions [G-06, G-07, S-13].

Secrets and HTTPS

API keys, database URLs, and signing keys live in environment variables on the platform dashboard — never in code or Git. HTTPS encrypts data in transit [G-05, S-01, S-02].

06 Mental model — and where it breaks down

Analogy: a shop with a strong lock but open windows

HTTPS is like a strong lock on the front door — it protects what travels in and out. But if you leave the back window open (no input validation) or leave the customer ledger on the counter (no access control), the lock does not matter. Security is every door and window, not just the front one.

Where the analogy breaks down

Unlike a physical shop, attackers can try thousands of attacks automatically from anywhere in the world. You must assume your application will be probed and design every layer defensively.

07 Technical explanation with correct terminology

Each term is defined at first use below and listed in the chapter glossary (Section 18).

07 Technical explanation with correct terminology

(continued)

- **OWASP Top 10:2021** — the ten most critical web application security risks G-01.
- **Broken access control (A01)** — users access data or actions they should not G-02.
- **Injection (A03)** — untrusted data interpreted as code commands (SQL injection, XSS) G-03.
- **Insecure design (A04)** — security controls never created; requires threat modeling G-04.
- **Cryptographic failures (A02)** — weak or missing encryption; plaintext passwords G-01.
- **Security misconfiguration (A05)** — debug mode on in production; default settings G-01.
- **Input validation** — checking user input meets rules before use [S-03, G-03].
- **Sanitization** — cleaning or neutralizing dangerous content in input [S-03].

07 Technical explanation with correct terminology

(continued)

- **Parameterized queries** — database queries that treat user input as data, not commands (prevents SQL injection).
- **Data minimization** — collecting only personal data necessary for the purpose [G-06, S-12].
- **Consent** — express, informed, specific agreement to process personal data [G-06, P-01].
- **Right to erasure** — the right to have personal data deleted without undue delay [G-06, G-07, S-13].
- **Privacy notice** — telling users what you collect, why, and how long you keep it [G-06, P-06].

07 Technical explanation with correct terminology

(continued) (continued)

- **Security checklist** — a pre-deploy list verifying secrets, HTTPS, authorization, validation [S-01, S-02, S-08].

07 Technical explanation with correct terminology

(continued)

[security] — all terms above (OWASP risks, validation, secrets, HTTPS, privacy)

08 Concrete example — Mifugo Market security hardening

IN PRACTICE

Wanjiku applies security to Mifugo Market:

08 Concrete example — Mifugo Market security hardening (continued)

- **Input validation:** the order form validates name, email, phone, and quantity on the server; rejects invalid input with HTTP 400 and a clear message.
- **Sanitization:** search input is treated as text, not commands; no raw SQL interpolation.
- **Access control:** every order endpoint checks ownership (403 for wrong user) [S-01, E-04].
- **HTTPS:** enforced on the deployed app; no credentials over plain HTTP [S-02].
- **Secrets:** session key and database URL in Render's dashboard; `.env` in `.gitignore`; not committed [S-01, G-05].
- **Data minimization:** the sign-up form collects name, email, and phone only — not ID numbers or other unnecessary data.
- **Privacy notice:** a plain-language page explains what data is collected, why, and how long it is kept.
- **Erasure process:** Wanjiku knows how to delete a customer's data on request (conceptual; database deletion).

09 Step-by-step demonstration — securing Mifugo Market

STEP BY STEP

Step 1: Add input validation to all user-facing forms

For every form field, validate on the server:

09 Step-by-step demonstration — securing Mifugo Market (continued)

- Name: required, reasonable length.
- Email: required, valid format.
- Phone: required, valid pattern.
- Quantity: required, positive integer.

Reject invalid input with HTTP 400 and a clear error message. Never trust frontend validation alone [S-03].

09 Step-by-step demonstration — securing Mifugo Market (continued)

Step 2: Use parameterized queries

When writing database queries, never interpolate user input directly into SQL. Use parameterized queries so user input is treated as data, not commands (prevents SQL injection, A03).

Step 3: Enforce authorization on every route

09 Step-by-step demonstration — securing Mifugo Market (continued)

Every endpoint that reads or writes user data checks: does the current user own this resource? Return 403 if not [S-01, E-04].

Step 4: Confirm HTTPS and secrets

- Deployed app served over HTTPS [S-02].
- Secrets in the platform dashboard, not in code [S-01, G-05].
- `.env` in `.gitignore`; not committed [Q-06].

09 Step-by-step demonstration — securing Mifugo Market (continued)

Step 5: Draft a privacy notice

Write a plain-language notice stating:

09 Step-by-step demonstration — securing Mifugo Market (continued)

- What data you collect (name, email, phone, order history).
- Why (to process orders and contact customers).
- How long you keep it.
- How customers can withdraw consent or request deletion.

Step 6: Run the pre-deploy security checklist

Complete every item in Section 12 before deploying.

10 AI collaboration example (prompt format P1–P10)

AI COLLABORATION — PROMPT FORMAT P1–P10

PROMPT FORMAT **A reusable prompt you can copy**

P1 **P1 · PROBLEM IT SOLVES**
Add server-side input validation and sanitization to a form without introducing security flaws.

P2 **P2 · WHEN TO USE IT**
Chapter 13, when securing Mifugo Market forms and API endpoints.

P3 **P3 · REQUIRED INPUTS**
The form or endpoint code, the validation rules, and the acceptance criteria from Chapter 5.

10 AI collaboration example (prompt format P1–P10) (continued)

P4

10 AI collaboration example (prompt format P1–P10) (continued)

P4 · THE PROMPT

10 AI collaboration example (prompt format P1–P10) (continued)

```text
Role: You are a careful backend developer focused on security.
Context: Mifugo Market (Chapter 13). Order form with fields:
name, email, phone, quantity.
Task: Add server-side input validation and sanitization for the
order form.
Validation rules:
- name: required, max 100 chars
- email: required, valid format
- phone: required, valid pattern
- quantity: required, positive integer
```

10 AI collaboration example (prompt format P1–P10) (continued)

```
TEXT
Constraints:
- Validate on the server, not only the frontend.
- Reject invalid input with HTTP 400 and a clear error message.
- Use parameterized queries for any database access.
- No hardcoded secrets or API keys.
- Do not log passwords or sensitive data.
- Return unified diffs with file paths.
- Reference AGENTS.md for project conventions.
```

10 AI collaboration example (prompt format P1–P10) (continued)

P5 **P5 · WHAT A GOOD RESPONSE CONTAINS**
Server-side validation; HTTP 400 errors; parameterized queries; no secrets; no logging of sensitive data; unified diffs.

P6 **P6 · WHAT THE READER MUST INSPECT**
Confirm validation runs on the server; confirm no secrets; confirm error messages are clear but do not leak internals; confirm parameterized queries are used.

P7 **P7 · WEAK OR INCOMPLETE RESPONSE**
Validation only in JavaScript (bypassable); no error messages; raw SQL interpolation; hardcoded keys; suggests disabling validation for testing.

P8 **P8 · FOLLOW-UP WHEN INADEQUATE**
"Move validation to the backend route," "Add HTTP 400 responses," "Use parameterized queries," "Remove the hardcoded key."

10 AI collaboration example (prompt format P1–P10) (continued) (continued)

P9 **P9 · SAFETY/SECURITY CONSIDERATIONS**
Validation is not authentication — combine with authorization from Chapter 11. Never log passwords or secrets. Review input-handling code carefully (security-critical path).

P10 **P10 · VERIFICATION STEP**
Submit invalid data through the form; confirm HTTP 400 and error message; confirm no data is saved for invalid input.

11 Human decisions and responsibilities (non-delegable judgment)

YOUR DECISION — NOT THE AI'S

- **What data to collect.** You decide what is necessary (data minimization); do not collect "just in case."
- **How long to keep data.** You set retention and deletion rules.
- **How to handle erasure requests.** You decide the process for deleting a customer's data.
- **Whether the app is safe to show users.** You run the security checklist and make the go/no-go call.

[Human decision-point box] Before deploying with real user data, answer: Is all input validated on the server? Are secrets out of code? Is HTTPS on? Is authorization on every route? Do I have a privacy notice and an erasure process? If any answer is no, do not deploy with real data.

12 Inspect and verify AI output (security checklist)

● QUALITY CHECK

Before deploying, confirm every item:

- ✓ [ ] No hardcoded secrets or API keys in any file. [S-01, G-05]
- ✓ [ ] `.env` in `.gitignore`; not committed. [Q-06]
- ✓ [ ] Production served over HTTPS. [S-02, A-01]
- ✓ [ ] Authorization enforced on every user-data route (403 test passes). [S-01, E-04]
- ✓ [ ] All user input validated on the server. [S-03, G-03]
- ✓ [ ] Parameterized queries used for database access.
- ✓ [ ] Error messages do not leak stack traces or internals. [S-05]
- ✓ [ ] Debug mode off in production. [S-05]
- ✓ [ ] Privacy notice present; data minimization applied. [G-06, S-12]
- ✓ [ ] Erasure process documented conceptually.

[Related to G-01–G-07, S-01–S-06, Q-06]

13 Failure modes and recovery

● WHEN IT GOES WRONG

13 Failure modes and recovery (continued)

FAILURE	SYMPTOM	LIKELY CAUSE	RECOVERY
Data exposed via URL	Customer A views customer B's order	Missing authorization check	Add ownership check; return 403; audit logs
SQL injection	Database error or data leak from search field	Raw SQL interpolation	Use parameterized queries; sanitize input
Secret leaked	API key found in Git history	`.env` committed before `.gitignore`	Rotate the key immediately; remove from code [G-05, S-01]
No erasure process	Customer requests deletion; no way to comply	No deletion procedure	Build a deletion process; respond without undue delay G-06 · G-07

13 Failure modes and recovery (continued)

FAILURE	SYMPTOM	LIKELY CAUSE	RECOVERY
Excessive data collection	Collecting ID numbers not needed	No data minimization	Stop collecting unnecessary data; delete what you have
Stack trace leaked	Error page shows internal code	Debug mode on in production	Disable debug; return generic error messages [S-05]

13 Failure modes and recovery (continued)

All scenarios in this section are hypothetical and labeled as such.

14 Professional standard — quality threshold before going live with user data

PRO STANDARD

An application handling real user data is ready only when:

- **Secrets excluded:** no hardcoded secrets; `.env` in `.gitignore`; production secrets in platform dashboard [S-01, G-05].
- **HTTPS everywhere:** production served over TLS; no credentials over plain HTTP [S-02, A-01].
- **Authorization on every route:** object-level authorization enforced; 403 test passes [S-01, E-04].
- **Input validated on server:** all user input validated and sanitized [S-03, G-03].
- **Parameterized queries:** database access uses parameterized queries `G-03`.
- **Error handling:** no stack traces leaked to users [S-05].
- **Privacy notice and minimization:** what you collect and why is stated; only necessary data collected [G-06, S-12].
- **Erasure process:** you can delete a customer's data on request [G-06, G-07, S-13].
- **Pre-deploy security checklist complete** (Section 12).

15 Hands-on exercise (L3 — Application)

BUILD IT

Task: Add input validation and sanitization to all user-facing forms in Mifugo Market (or your project), and draft a privacy notice.

Constraints:

15 Hands-on exercise (L3 — Application) (continued)

- Validate on the server, not only the frontend.
- No secrets in code.
- Privacy notice in plain language.

Deliverable: Validated forms plus a one-paragraph privacy notice.

Success criteria:

15 Hands-on exercise (L3 — Application) (continued)

- Server-side validation on all forms with HTTP 400 for invalid input.
- No hardcoded secrets or API keys.
- Privacy notice states what data is collected, why, and how long.
- `.env` not tracked by Git.

15 Hands-on exercise (L3 — Application) (continued)

Assessment level: L3 (Application — performs a procedure with guidance)

Time estimate: 100–120 minutes.

16 Build challenge (L4 — Analysis) with acceptance criteria

● PUSH FURTHER

Task: Complete the pre-deploy security checklist on your deployed application and identify one OWASP risk relevant to your app with its mitigation.

Constraints:

- Do not ask the AI to regenerate the app.
- Work with your actual deployed code.
- Document findings in writing.

Acceptance criteria (Given-When-Then):

- **Given** the pre-deploy security checklist, **When** each item is reviewed against the deployed app, **Then** every item is checked off or a gap is noted with a fix plan.
- **Given** the application, **When** one OWASP Top 10:2021 risk is identified as relevant, **Then** a written mitigation is documented (for example, broken access control → object-level authorization checks).
- **Given** an erasure request scenario, **When** the process is reviewed, **Then** the steps to delete a customer's data are described.

Deliverable: A completed security checklist plus a one-page OWASP risk + mitigation write-up.

Assessment level: L4 (Analysis — investigates evidence, diagnoses a failure)

Time estimate: 60–75 minutes.

17 Key takeaways

KEY TAKEAWAYS

- HTTPS is necessary but not sufficient: you also need authorization, validation, and secure design.
- Treat all user input as untrusted; validate on the server.
- OWASP Top 10:2021 risks most relevant to small apps: broken access control, injection, insecure design, cryptographic failures, security misconfiguration.
- Collect only necessary data; get consent; honor erasure requests.
- Secrets live in the platform dashboard, never in code.
- Run the full pre-deploy security checklist before deploying with real user data.

18 Glossary of new terms

TERMS INTRODUCED IN THIS CHAPTER

18 Glossary of new terms (continued)

TERM	PLAIN-LANGUAGE DEFINITION	EXAMPLE	WHY IT MATTERS	TAG
OWASP Top 10:2021	The ten most critical web application security risks.	A01 broken access control, A03 injection	A checklist of the dangers to defend against.	[security]
Broken access control (A01)	Users access data or actions they should not.	Viewing another customer's order by URL ID	The most serious web app risk.	[security]
Injection (A03)	Untrusted data interpreted as commands.	SQL injection via a search field	Can expose or corrupt your database.	[security]

18 Glossary of new terms (continued) (continued)

TERM	PLAIN-LANGUAGE DEFINITION	EXAMPLE	WHY IT MATTERS	TAG
Insecure design (A04)	Security never built in; added as afterthought.	Auth added after the app is built	Cannot be fixed by perfect implementation alone.	[security]
Cryptographic failures (A02)	Weak or missing encryption; plaintext passwords.	Passwords stored in plain text	Exposes credentials if the database is breached.	[security]
Security misconfiguration (A05)	Debug mode on in production; default settings.	Stack traces shown to users	Leaks internals and aids attackers.	[security]

18 Glossary of new terms (continued) (continued)

TERM	PLAIN-LANGUAGE DEFINITION	EXAMPLE	WHY IT MATTERS	TAG
Input validation	Checking user input meets rules before use.	Rejecting a negative quantity	Stops bad data before it causes harm.	[security]
Sanitization	Cleaning or neutralizing dangerous input content.	Treating search text as text, not commands	Neutralizes injection attempts.	[security]

18 Glossary of new terms (continued)

TERM	PLAIN-LANGUAGE DEFINITION	EXAMPLE	WHY IT MATTERS	TAG
Parameterized queries	Database queries treating input as data, not commands.	Prepared statements with placeholders	Prevents SQL injection.	[security]
Data minimization	Collecting only personal data necessary for the purpose.	Not asking for ID numbers when not needed	Less data means less risk.	[security]
Consent	Express, informed, specific agreement to process data.	Customer ticks "I agree" before sign-up	Required by Kenya DPA and GDPR-informed principles.	[security]
Right to erasure	The right to have personal data deleted on request.	Deleting a customer's account and orders	Customers can withdraw and be forgotten.	[security]
Privacy notice	Plain-language statement of what data you collect and why.	"We collect your email to confirm orders"	Transparency for users; legal awareness.	[security]
Security checklist	A pre-deploy list verifying secrets, HTTPS, authorization, validation.	Section 12 checklist	Ensures nothing is forgotten before going live.	[security]

19 Knowledge-check questions

KNOWLEDGE CHECK

Attempt these before looking at the answers below.

Q1. Name three OWASP Top 10:2021 risks relevant to a small web app.

Q2. Why is frontend validation not enough for security?

Q3. What is data minimization?

Q4. What should you do when a customer asks you to delete their personal data?

Q5. Name three items on the pre-deploy security checklist.

[PAGE BREAK — ANSWER KEY]

Answers

A1. Any three of: broken access control (A01), injection (A03), insecure design (A04), cryptographic failures (A02), security misconfiguration (A05) G-01.

A2. Frontend validation can be bypassed by an attacker; real validation must happen on the server [S-03, G-03].

A3. Collecting only the personal data necessary for your stated purpose, and deleting it when no longer needed [G-06, S-12].

A4. Delete the data without undue delay, subject to legal exceptions (for example, legal obligations or claims) [G-06, G-07, S-13].

A5. Any three of: no hardcoded secrets; `.env` in `.gitignore`; HTTPS on; authorization on every route; input validated on server; parameterized queries; no stack traces leaked; privacy notice present.

20 Links to earlier and later chapters

WHERE THIS SITS IN THE BOOK

Earlier chapters (prerequisites):

- **Chapter 10:** Backend, Data, and APIs — the endpoints you now secure.
- **Chapter 11:** Access Control — authorization and password hashing feed into this chapter.
- **Chapter 8:** Diff Discipline — security-critical code must pass diff review.
- **Chapter 9:** Frontend — form validation connects to server-side validation here.

Later chapters (where this is used next):

- **Chapter 14:** Version Control, Deployment, and Operations — the security checklist is part of the pre-deploy gate; secrets go in the platform dashboard.
- **Chapter 16:** Capstone — the pre-deploy security checklist and privacy notice are required capstone deliverables.

Evidence notes: Claims trace to G-01–G-07 (OWASP, Kenya DPA, GDPR, secrets), S-01–S-06 (security practices), E-01–E-04 (API auth), Q-06 (version control). Kenya DPA and GDPR are covered as awareness, not legal advice; consult a qualified attorney for your jurisdiction. All scenarios are hypothetical.

Time estimate: Approximately 220 minutes total (100 minutes reading, 120 minutes hands-on practice).

PART 6

6

Ship and Operate

Version control, deployment, operations

14 Version Control, Deployment, and Operations

PART 6 · SHIP AND OPERATE

14

Version Control, Deployment, and Operations

01 Opening problem — the misconceptions that cost real businesses

SLOP DETECTOR — THE MISCONCEPTION

The scenario

Wanjiku runs Mifugo Market, a fresh-produce ordering platform for small shops and households in Nairobi. The application works on her laptop. Customers can browse sukuma wiki, tomatoes, and avocados, and place orders. She uploads the files to a hosting platform once, gets a live URL, and considers the job done.

Two weeks later, she asks her AI coding assistant to change the checkout flow. The AI regenerates parts of the code, and she accepts the changes without reviewing the diff. The next morning, customers trying to pay for their orders get an error page. Wanjiku has no Git history showing what changed, no way to see the previous working version, no staging area where the change could have been caught first, and no rollback plan. During peak market hours in Ruiru and Kiambu, Mifugo Market is down, and customers cannot place orders.

Three dangerous misconceptions

01 Opening problem — the misconceptions that cost real businesses (continued)

#	MISCONCEPTION	WHY IT IS DANGEROUS	THE CORRECTION
M-12	"I don't need Git because the AI can regenerate the code."	Regeneration is not deterministic. You lose history and cannot roll back to the exact version that worked.	Git is the safety net; commit before and after every AI session. H-01

01 Opening problem — the misconceptions that cost real businesses (continued) (continued)

#	MISCONCEPTION	WHY IT IS DANGEROUS	THE CORRECTION
M-14	"Deployment is a one-time event."	You deploy once and walk away, with no monitoring, backups, or update plan.	Deployment is the beginning of monitoring, maintenance, patching, and cost management. [Constitution §3]

01 Opening problem — the misconceptions that cost real businesses (continued) (continued)

#	MISCONCEPTION	WHY IT IS DANGEROUS	THE CORRECTION
M-16	"It works on my machine, so it's done."	Environment differences, real data, and real traffic reveal hidden failures that never appeared locally.	Test in staging, smoke test after deploy, and monitor production. F-02 · F-03

01 Opening problem — the misconceptions that cost real businesses (continued) (continued)

These misconceptions are the difference between a hobby project and a professional application real users can depend on.

02 Why this topic matters in real software projects

Deployment is where software meets real people. Until your application lives at a public URL that anyone can visit, it is an experiment, not a product. After deployment, new risks appear:

02 Why this topic matters in real software projects

(continued)

- **Downtime you do not see.** If the application crashes at 7 a.m. in Nairobi when shop owners are ordering stock, you may not know until a customer calls or messages you.
- **Changes you cannot undo.** Without version control, an AI-generated edit that breaks checkout cannot be cleanly reversed. You are left guessing what changed.

02 Why this topic matters in real software projects

(continued)

- **Secrets that leak.** Database URLs and signing keys left in code or committed to Git can be copied by anyone who sees the repository, giving them access to your customer data.
- **Costs you do not track.** A free hosting tier can quietly become a paid bill, or a domain can expire and take your application offline.

02 Why this topic matters in real software projects

(continued)

Constitution §3 states the central point directly: publishing an app is not the end — it is the beginning of maintenance, monitoring, security patching, and cost management. This chapter teaches you how to deploy safely, operate responsibly, and recover when things go wrong.

03 Learning objectives

WHAT YOU WILL BE ABLE TO DO

By the end of this chapter, the reader can:

03 Learning objectives (continued)

- Use Git (`init`, `add`, `commit`, `branch`, `merge`, `revert`, `push`, `pull`, `.gitignore`) as a safety net before and after every AI coding session.
- Deploy a full-stack Node.js/Express application to a PaaS (Render) using a custom domain or a free subdomain.

03 Learning objectives (continued)

- Configure environment variables in the production dashboard — never in code, never in a committed `.env` file.
- Explain staging versus production and why to verify a release in staging before serving real users. [Presented as guidance; see Q-07 below.]

03 Learning objectives (continued)

- Set up basic error monitoring and uptime awareness so failures are noticed quickly.
- Estimate monthly hosting cost for a small application.
- Roll back a bad deployment to the last known-good version.

04 Prerequisites and what to skip

BEFORE YOU START

Prerequisites

Chapters 1–13. In particular:

04 Prerequisites and what to skip (continued)

- **Chapter 1:** HTTP request/response and what happens when a browser visits a URL.
- **Chapter 6:** Git `init` / `add` / `commit`, your first PaaS deploy, `.env`, and `.gitignore`.
- **Chapter 10:** The backend and API routes you are now deploying.

04 Prerequisites and what to skip (continued)

- **Chapter 11:** Authentication and HTTPS, which must survive the move to production.
- **Chapter 12:** Tests you must run before every deploy.
- **Chapter 13:** The pre-deploy security checklist you must complete before going live.

04 Prerequisites and what to skip (continued)

What you can skip

Advanced DevOps infrastructure — Kubernetes, Terraform, multi-region clusters, and auto-scaling — is excluded from this book per Constitution §6. A first application does not need it. Basic deployment, domains, environment variables, backups, and rollback are sufficient and are covered here.

05 Plain-English explanation

Version control is a time machine for your code

Git records every change you make as a **commit** — a labeled save point with a message describing what changed and why. If an AI-generated edit breaks the checkout flow, you can look at the history, see exactly what changed, and return to the last commit that worked. Without Git, every change overwrites the past and there is no way back.

Deployment is publishing to the internet

Your laptop is not a good home for an application real users depend on: it goes to sleep, loses internet, and is not reachable from a phone. Deployment copies your application to a computer that stays online — a server managed by a hosting platform — so anyone with your URL can use it from anywhere, including a mobile phone in Nairobi or Mombasa.

Operations is keeping it alive

Deployment is day one. Operations is everything after: watching error logs, renewing your domain name, paying the hosting bill, backing up the database, and having a plan to roll back when a release breaks something.

06 Mental model — and where it breaks down

Analogy: dress rehearsal versus opening night

Think of **staging** as a restaurant's dress rehearsal: the kitchen runs through service with staff pretending to be customers, and every dish is checked before real diners arrive. Think of **production** as opening night: real customers, real money, real reputation. You do not serve a new dish to paying customers without testing it in rehearsal first.

Where the analogy breaks down

06 Mental model — and where it breaks down (continued)

Unlike a restaurant, software can be updated while customers are using it, and a bad deployment can usually be rolled back in minutes — you cannot "un-serve" a meal once it has been eaten. Staging reduces risk, but it does not remove the need for care in production. A change that passes staging can still behave differently with real data and real traffic, which is why monitoring and rollback plans remain essential.

07 Technical explanation with correct terminology

Each term is defined at first use below and listed in the chapter glossary (Section 18).

07 Technical explanation with correct terminology

(continued)

- **Git** — version-control software that records every change to your code as a commit. `H-01`
- **Repository (repo)** — the folder Git tracks, containing your code and its full history.
- **Commit** — a labeled save point with a message describing what changed.
- **Branch** — a separate line of development, so you can experiment without disturbing the main code.
- **Merge** — combining one branch (for example, a feature branch) into another (usually `main`).
- **Revert** — a new commit that undoes an earlier commit while keeping the history intact.
- **Push / Pull** — sending your commits to a remote repository such as GitHub, or fetching updates from it.
- **.gitignore** — a file listing folders and files Git must never track (for example, `node_modules`, `.env`, database files).

07 Technical explanation with correct terminology

(continued) (continued)

- **Staging environment** — a private, production-like environment used for final verification before release. [Presented as guidance; H-04 is a research gap — see Q-07.]
- **Production environment** — the live environment serving real users at your public URL.
- **Environment variable** — a value set outside the code (database URL, secret key) that the application reads at runtime. Keeps configuration and secrets out of the repository.
- **.env** — a local file holding environment variables during development; must be in `.gitignore` and never committed.
- **.env.example** — a template file with placeholder values (not real secrets) committed to the repo so others know which variables to set.

07 Technical explanation with correct terminology

(continued)

- **PaaS (Platform as a Service)** — a hosting platform that manages servers, building, and deployment for you (for example, Render).
- **Build command** — the command the platform runs to install dependencies (for example, `npm install`).
- **Start command** — the command the platform runs to start your application (for example, `node server.js` or `npm start`).
- **Custom domain** — a domain name you own (for example, `mifugomarket.co.ke`) that points to your application.
- **DNS (Domain Name System)** — the system that translates a domain name into an IP address.
- **A record / CNAME** — DNS records that point a domain to a server's IP address or to another domain name.

07 Technical explanation with correct terminology

(continued) (continued)

- **Rollback** — returning the application to a previous version after a bad deployment.
- **Uptime** — the percentage of time your application is reachable and working.
- **Backup** — a copy of your database and code stored separately, for recovery after data loss.

07 Technical explanation with correct terminology

(continued)

[security] — Environment variable, `.gitignore`, Staging/Production, Rollback (protect secrets and recover from bad releases)

08 Concrete example — Mifugo Market goes live

IN PRACTICE

Wanjiku's Mifugo Market code lives in a Git repository on GitHub. She connects Render to that repository. Every `git push` to the `main` branch triggers Render to rebuild and redeploy the application automatically. She sets production secrets — the database path and a session-signing key — in Render's dashboard, not in the code. Before a change reaches customers, she checks it in a staging preview. After deploy, she watches Render's build logs for errors.

This chapter walks through that process end to end: Git workflow for AI-assisted work, deploying to Render, configuring production environment variables, staging preview, rollback, monitoring, and cost estimation.

Diagram — the deployment pipeline

08 Concrete example — Mifugo Market goes live

(continued)

Alt text: A left-to-right flowchart of the deployment pipeline. Boxes in order: "Local development" → "Git commit" → "git push to GitHub" → "Render builds and builds a preview" → "Staging preview — test checkout flow" → decision diamond "Tests and preview pass?" with a "No" path looping back to "Fix and retry" and a "Yes" path continuing to "Production deploy — live URL" → "Monitoring: build logs and uptime" → decision diamond "Bad deployment?" with a "Yes" path to "git revert — rollback to previous version" (which loops back to production deploy) and a "No" path ending at "Application live and operating".

08 Concrete example — Mifugo Market goes live

(continued)

```text
Deploy pipeline for Mifugo Market

[Local development]
      |
      v
[Git commit]
      |
      v
[git push to GitHub]
      |
      v
[Render builds; preview environment created]
      |
      v
[Staging preview — test checkout flow]
      |
```

08 Concrete example — Mifugo Market goes live

(continued) (continued)

```
TEXT
      v
[Tests and preview pass?] --No⟶ [Fix and retry] --(back to Git
commit)
      |
     Yes
      |
      v
[Production deploy – live URL]
      |
      v
[Monitoring: build logs, uptime]
      |
      v
[Bad deployment?] --Yes⟶ [git revert – rollback] --(back to
Production deploy)
      |
```

08 Concrete example — Mifugo Market goes live

(continued) (continued)

```
TEXT
     No
      |
      v
```

08 Concrete example — Mifugo Market goes live

(continued) (continued)

```
TEXT
[Application live and operating]
```

08 Concrete example — Mifugo Market goes live

(continued)

This pipeline repeats for every change. The staging preview is the dress rehearsal; production is opening night.

09 Step-by-step demonstration

STEP BY STEP

Phase A — Git as a safety net for AI-assisted development

Follow these rules for every session with your AI coding editor:

1. **Commit before every AI session.** Create a save point so you can return to the state before the AI touched anything.
2. **Review every diff before accepting.** See Chapter 8 for the diff-review checklist.
3. **Commit after every accepted AI change**, with a message describing what changed and why.
4. **Use a branch for experiments.** Try out a new checkout flow on a branch named `experiment/new-checkout`; merge into `main` only after review and tests pass.
5. **Never commit secrets.** `.env`, `node_modules`, build output, and database files belong in `.gitignore`.

ESSENTIAL GIT COMMANDS

09 Step-by-step demonstration (continued)

```bash
# See what has changed and what is ready to commit
git status

# Stage all changes for the next commit
git add .

# Save a labeled save point
git commit -m "feat: add input validation to order checkout"

# Send commits to GitHub (also triggers Render to redeploy, once connected)
git push origin main
```

WORKING ON A BRANCH AND MERGING SAFELY

09 Step-by-step demonstration (continued)

```bash
# Create and switch to a new branch for an experiment
git checkout -b experiment/new-checkout

# ... make changes, run tests, review diff ...

# Switch back to main
git checkout main

# Merge the experiment only if tests pass
git merge experiment/new-checkout
```

ROLLING BACK A BAD CHANGE

If a commit breaks the application, `git revert` creates a new commit that undoes it without deleting history:

09 Step-by-step demonstration (continued) (continued)

```bash
# Undo the changes introduced by a specific commit
git revert <commit-hash>

# Send the reversal to GitHub; Render redeploys the previous version
git push origin main
```

RECOMMENDED `.GITIGNORE` FOR A NODE.JS APPLICATION

09 Step-by-step demonstration (continued) (continued)

```text
node_modules/
.env
.env.local
*.db
mifugo.db
dist/
build/
.DS_Store
```

09 Step-by-step demonstration (continued)

[verify current] — Git commands are stable (git-scm.com), but GUI behavior in AI editors changes; check the companion website for current editor-specific steps. H-01

Phase B — Deploying Mifugo Market to Render

09 Step-by-step demonstration (continued) (continued)

Render is a PaaS: you connect your GitHub repository, and it builds, deploys, and hosts your application. Render is used here because it offers a usable free tier that does not require a credit card to start, and it supports Node.js applications with environment variables set in its dashboard. [Tool selection per Constitution §8; tool_comparison_framework.md]

DEPLOY STEPS (RENDER-SPECIFIC — VERIFY CURRENT)

09 Step-by-step demonstration (continued)

1. Push your committed code to GitHub.
2. Log into Render and connect your GitHub repository.
3. Render detects that this is a Node.js application. Set:
4. **Build command:** `npm install`
5. **Start command:** `node server.js` (or `npm start`, matching your `package.json`)
6. Add **environment variables** in Render's dashboard (not in the repository):
7. `NODE_ENV=production`
8. `DATABASE_URL` — the path or connection string to your database
9. `SESSION_SECRET` — a long, random string used to sign sessions
10. Choose a free `onrender.com` subdomain, or add a custom domain (for example, `mifugomarket.co.ke`).
11. If using a custom domain, configure DNS through your domain registrar: add an A record or CNAME pointing to Render.
12. Trigger the first deployment and watch the build log.
13. Visit the live URL and smoke test: homepage loads, product list shows, order form submits without a server error.

09 Step-by-step demonstration (continued) (continued)

[verify current] — Render's dashboard layout, build/start command defaults, free-tier limits, and domain configuration steps change over time. Treat these steps as illustrative and confirm them against Render's official documentation and the companion website before following them. H-02

REGIONAL NOTE FOR KENYAN READERS

09 Step-by-step demonstration (continued)

Many PaaS free tiers require no payment card to begin, but paid upgrades often require an international credit card; mobile money (M-Pesa) integration is rare on global developer platforms. [tool_comparison_framework.md, Section 5; R-09] Hosting costs are usually quoted in USD — convert to KES and budget for exchange-rate fluctuation. If Render's paid tier is not accessible, the same principles apply on other Git-connected platforms; check the companion website for current regional alternatives.

Phase C — Staging preview and rollback

Render can build a **preview environment** for every pull request — a temporary, production-like copy of your application with the proposed changes. Use it as the dress rehearsal before opening night:

1. Create a branch, push your changes, and open a pull request on GitHub.
2. Render builds a preview URL. Test the full checkout flow there: browsing, adding to cart, placing an order.
3. Merge into `main` only after the preview passes tests and review.
4. Render redeploys production automatically from `main`.

ROLLBACK WHEN PRODUCTION BREAKS

09 Step-by-step demonstration (continued) (continued)

If a deployment breaks a feature:

```bash
# Revert the offending commit
git revert <commit-hash>
git push origin main
```

Render detects the new commit on `main` and redeploys the previous working version. This is faster and safer than trying to "fix forward" under pressure. [F-06 — regression after AI edit]

09 Step-by-step demonstration (continued)

[verify current] — Preview environment availability and limits vary by platform and pricing tier; confirm Render's current offering before relying on it. H-02

Phase D — Monitoring, uptime, and cost

After deployment, operations begins:

09 Step-by-step demonstration (continued) (continued)

- **Watch build logs.** After every `git push`, check Render's deploy log for errors or warnings.
- **Know when it fails.** Set up basic error awareness — platform notifications, or at minimum a weekly check that the homepage and order form still load.
- **Estimate monthly cost.** Render's free tier covers small applications at no cost. A custom `.co.ke` domain typically costs roughly KES 1,000–2,500 per year (pricing varies by registrar; convert from USD where quoted). As traffic grows, a paid platform tier may be needed. Record your estimate and review it quarterly.
- **Renew your domain.** Set a calendar reminder before expiry; an expired domain takes the application offline.

10 AI collaboration example (prompt format P1–P10)

AI COLLABORATION — PROMPT FORMAT P1–P10

PROMPT FORMAT **A reusable prompt you can copy**

When configuring deployment, you can ask your AI assistant for help — but you must inspect the output before applying it.

P1 **P1 · PROBLEM IT SOLVES**
Configure Render deployment files and start command for a Node.js/Express application without exposing secrets.

P2 **P2 · WHEN TO USE IT**
Chapter 14, when deploying Mifugo Market for the first time.

P3 **P3 · REQUIRED INPUTS**
Your repository path, the application's start command, and the list of environment variables the application needs.

10 AI collaboration example (prompt format P1–P10) (continued)

P4

10 AI collaboration example (prompt format P1–P10) (continued)

P4 · THE PROMPT

10 AI collaboration example (prompt format P1–P10) (continued)

TEXT

```
Review my Node.js/Express app at [REPO_PATH]. I want to deploy
it to Render.

Suggest:
1. A render.yaml file (or equivalent Render dashboard settings)
with the build command and start command.
2. Which environment variables must be set in Render's dashboard
(not in code): NODE_ENV, DATABASE_URL, SESSION_SECRET.
3. A .env.example file with placeholder values for local
development.
```

10 AI collaboration example (prompt format P1–P10) (continued)

```
TEXT
Do not hardcode any secrets. Do not suggest putting keys in URL
parameters. Return changes as unified diffs. Reference the
context file at [CONTEXT_FILE] for project conventions.
```

10 AI collaboration example (prompt format P1–P10) (continued)

P5 **P5 · WHAT A GOOD RESPONSE CONTAINS**
Build/start command guidance; an environment-variable list for the dashboard; a `.env.example` with placeholder values only; unified diffs; no hardcoded secrets.

P6 **P6 · WHAT THE READER MUST INSPECT**
Confirm the start command matches your app; confirm no secret values appear in the diff; confirm `.env` is listed in `.gitignore`.

P7 **P7 · WEAK OR INCOMPLETE RESPONSE**
Suggests putting `SESSION_SECRET` in source code; suggests a `?secret=` URL parameter; omits `DATABASE_URL`; does not mention `.gitignore`.

P8 **P8 · FOLLOW-UP WHEN INADEQUATE**
"Move the secret to Render's dashboard," "Show me the `.gitignore`," "What exact start command should I use for this app structure?"

P9 **P9 · SAFETY/SECURITY CONSIDERATIONS**
Never commit `.env`. If a key is accidentally committed, rotate it immediately — treating it as compromised. Production secrets live only in the platform dashboard. [G-05, S-01]

10 AI collaboration example (prompt format P1–P10) (continued) (continued)

P10 **P10 · VERIFICATION STEP**

Deploy to Render; confirm the application starts; confirm environment variables are read (for example, `NODE_ENV=production` appears in logs or a debug route); confirm `.env` is not tracked by Git:

BASH

```
git ls-files | grep .env
```

This command should return nothing.

11 Human decisions and responsibilities (non-delegable judgment)

YOUR DECISION — NOT THE AI'S

The AI can suggest deployment settings, but the following decisions belong to you:

11 Human decisions and responsibilities (non-delegable judgment) (continued)

- **When to deploy.** You judge whether tests pass and the application is ready; the AI cannot judge ship-readiness.
- **Which values are secrets.** You identify database URLs, signing keys, and API keys, and you ensure they never enter the repository.
- **The rollback decision.** You decide whether to fix forward or revert when a deployment breaks something.
- **Cost budget.** You set a monthly hosting budget and watch for unexpected charges.
- **Domain and DNS.** You own the domain registration and DNS records; the AI can suggest but cannot manage them.
- **Backup decision.** You decide how often to back up the database and how to restore it if needed.

11 Human decisions and responsibilities (non-delegable judgment) (continued)

[Human decision-point box] Before every production deploy, answer: Have the tests passed? Are secrets excluded from the repository? Are environment variables set in the dashboard? Is there a rollback plan? If any answer is no, do not deploy.

12 Inspect and verify AI output (deployment-related checklist)

● QUALITY CHECK

Before accepting AI-generated deployment configuration, confirm:

- ✓ [ ] The start command runs the application locally (`npm start` works without errors).
- ✓ [ ] The build command succeeds (`npm install` completes).
- ✓ [ ] No `.env`, secret files, or credentials appear in the diff.
- ✓ [ ] `.gitignore` excludes `node_modules`, `.env`, `*.db`, and build output.
- ✓ [ ] Environment variable names in the code match those set in Render's dashboard.
- ✓ [ ] HTTPS is enabled (Render provides it automatically for Render subdomains and custom domains).
- ✓ [ ] The build log was reviewed after deploy; no unhandled crashes.

[Related to S-01, S-02, Q-06]

13 Failure modes and recovery

● WHEN IT GOES WRONG

13 Failure modes and recovery (continued)

FAILURE	SYMPTOM	LIKELY CAUSE	RECOVERY
Build fails	Render build log shows "command not found" or missing dependency	Missing dependency in `package.json`; wrong build command	Add the missing package; correct the build command; redeploy
Application crashes after deploy	500 error; application missing from logs	Wrong start command; missing environment variable; port mismatch	Check Render logs; set the missing variable in the dashboard; fix the start command; redeploy

13 Failure modes and recovery (continued)

FAILURE	SYMPTOM	LIKELY CAUSE	RECOVERY
Secret leaked	API key or database URL found in Git history	`.env` was committed before being added to `.gitignore`	Rotate the key immediately; remove it from the code; treat it as compromised [G-05, S-01]
Deploy breaks a feature	Checkout stopped working after an AI edit	Regression; change was not checked in staging first	`git revert` the offending commit; push; Render redeploys the previous version F-06

13 Failure modes and recovery (continued)

FAILURE	SYMPTOM	LIKELY CAUSE	RECOVERY
Domain does not resolve	Browser shows "server not found" or the wrong site	DNS A/CNAME not configured or not yet propagated	Check DNS settings with your registrar; allow up to 48 hours for propagation
No visibility into errors	You do not know the application is down until a customer reports it	No monitoring or log-review habit	Check Render logs after deploy; schedule a weekly check of the homepage and order form [S-10]

13 Failure modes and recovery (continued)

All scenarios in this table are hypothetical and labeled as such — they illustrate failure patterns, not real case studies.

14 Professional standard — quality threshold before going live

PRO STANDARD

An application is deployable only when all of the following hold:

14 Professional standard — quality threshold before going live (continued)

- **Version-control discipline (Q-06):** Commit before and after AI sessions; meaningful commit messages; `.gitignore` excludes `node_modules`, `.env`, build output, and database files. H-01
- **Staging before production (Q-07):** Verify the release candidate in a staging or preview environment that mirrors production before serving real users. [Class C — reasonable recommendation; H-04 is an evidence gap, so this is presented as the author team's guidance, not an industry standard]
- **Documentation (Q-08):** A README with setup instructions, acceptance criteria, and a deploy checklist. [Constitution §10]
- **Secrets (S-01):** No hardcoded secrets; `.env` in `.gitignore`; production secrets in the platform dashboard only. G-05
- **HTTPS everywhere (S-02):** Production is served over TLS; no credentials or tokens travel over plain HTTP. [A-01; G-01]
- **Tests before deploy (S-08):** Unit and smoke tests pass; no deploy on red. F-01 · F-03
- **Dependencies updated (S-09):** Monitor for vulnerable and outdated components; update regularly. [G-01, A06]
- **Error monitoring (S-10):** You know when the application fails in production; basic error tracking is in place. [G-01, A09]
- **Backups and rollback (S-11):** Regular database backups exist, and you can revert a bad deployment. [Constitution §3 — Class C reasonable recommendation]

15 Hands-on exercise (L3 — Application)

BUILD IT

Task: Deploy your Mifugo Market codebase to Render and confirm it is reachable at a public URL.

Constraints:

- Commit all work with Git before deploying, and push to GitHub first.
- Do not commit `.env` or any secrets.
- Set production environment variables in Render's dashboard, not in the repository.
- Run the test suite locally before deploying; all tests must pass.

Deliverable: A live public URL (Render free subdomain, or a custom domain such as `mifugomarket.co.ke`) showing the deployed Mifugo Market.

Success criteria:

- The URL loads in a browser over HTTPS.
- The order form loads and submits without a server error.
- `.env` is not tracked by Git (`git ls-files | grep .env` returns nothing).
- At least one environment variable is set in Render's dashboard.

Assessment level: L3 (Application — performs a procedure with guidance)

Time estimate: 60–90 minutes.

16 Build challenge (L4 — Analysis) with acceptance criteria

● PUSH FURTHER

Task: Set up a staging preview, intentionally introduce a small breaking change, observe it in the preview, roll back, and document the process. Produce a monthly hosting-cost estimate.

Constraints:

- Do not ask the AI to regenerate the whole application (stated rule; see Chapters 8 and 12).
- Make the smallest possible change to reproduce the issue.
- Test the change in the preview/staging environment before it reaches production.

Acceptance criteria (Given–When–Then):

- **Given** a feature branch with a breaking change pushed to GitHub, **When** Render builds the preview environment, **Then** the preview URL is reachable and the breaking behavior is observable before any production user sees it.
- **Given** the breaking change is merged to `main` and deployed, **When** a user visits the affected route in production, **Then** the error is visible in Render's build or deploy logs.
- **Given** a bad production deployment, **When** the human reverts the offending commit and pushes, **Then** Render redeploys the previous working version and the route returns to normal.
- **Given** the application is deployed, **When** the reader lists all recurring hosting costs, **Then** the estimate includes platform tier, domain cost, and optional add-ons, stated in both USD and KES.

Deliverable: Documented rollback steps (one to two paragraphs), a monthly cost estimate, and a note of the preview URL used.

16 Build challenge (L4 — Analysis) with acceptance criteria (continued)

Assessment level: L4 (Analysis — investigates evidence, diagnoses a failure, plans recovery)

17 Key takeaways

KEY TAKEAWAYS

- Git is your safety net: commit before and after every AI session, and never skip it.
- Deployment is the beginning of operations, not the end — monitoring, backups, and rollback plans matter.
- Secrets live in the platform dashboard, never in Git and never in code.
- Staging (preview) catches bugs before customers see them; treat it as the dress rehearsal before opening night.
- Every deploy follows a checklist: tests pass, secrets excluded, environment variables set, build log reviewed.
- Know your monthly hosting cost and keep your domain renewed.

18 Glossary of new terms

TERMS INTRODUCED IN THIS CHAPTER

18 Glossary of new terms (continued)

TERM	PLAIN-LANGUAGE DEFINITION	EXAMPLE	WHY IT MATTERS	TAG
Commit	A labeled save point in Git with a message describing what changed.	`git commit -m "fix: correct order total calculation"`	Lets you return to a known-good state when something breaks.	
Branch	A separate line of development in Git.	`git checkout -b fix/checkout-bug`	Lets you experiment without disturbing the main code.	
Merge	Combining one branch into another, usually into `main`.	Merging a feature branch after review and tests.	Integrates new work safely into the main application.	
Revert	A new commit that undoes an earlier commit while keeping history.	`git revert abc1234`	Rolls back a bad change without erasing what happened.	
Push / Pull	Sending commits to a remote repository (GitHub) or fetching updates from it.	`git push origin main`	Backs up your work and lets Render redeploy from GitHub.	
`.gitignore`	A file listing folders and files Git must never track.	`node_modules/`, `.env`, `*.db`	Keeps secrets and build clutter out of your repository.	[security]

18 Glossary of new terms (continued)

TERM	PLAIN-LANGUAGE DEFINITION	EXAMPLE	WHY IT MATTERS	TAG
Staging environment	A private, production-like environment for final verification before release.	Render preview URL for a pull request	Catches problems before real users encounter them. [guidance]	[security
Production environment	The live environment serving real users at your public URL.	`https://mifugomarket.co.ke`	Mistakes here affect real people and real data.	[security
Environment variable	A value set outside the code that the app reads at runtime.	`NODE_ENV=production` in Render's dashboard	Separates configuration from code and keeps secrets out of Git.	[security
`.env`	A local file holding development environment variables.	`SESSION_SECRET=dev-only-value`	Convenient locally, but must never be committed to Git.	[security
PaaS	A hosting platform that manages servers, building, and deployment.	Render	Lets you deploy without managing infrastructure yourself.	

18 Glossary of new terms (continued)

TERM	PLAIN-LANGUAGE DEFINITION	EXAMPLE	WHY IT MATTERS	TAG
Build command	The command a platform runs to install dependencies.	`npm install`	Wrong commands mean the application will not start in production.	
Start command	The command a platform runs to start your application.	`node server.js`	Must match how your application actually starts.	
Custom domain	A domain name you own that points to your application.	`mifugomarket.co.ke`	Gives your application a professional, memorable address.	
DNS	Translates a domain name into an IP address.	`mifugomarket.co.ke` → Render's IP address	Lets users reach your application by name.	
A record / CNAME	DNS records pointing a domain to a server or another domain.	CNAME `www` → `xxx.onrender.com`	Connects your custom domain to the hosting platform.	
Rollback	Returning the application to a previous version after a bad deploy.	`git revert` + `git push`	Restores service quickly when a release breaks something.	
Uptime	The percentage of time your application is reachable and working.	99.9% uptime target	Measures reliability for your users.	
Backup	A copy of your database and code stored separately.	Weekly export of `mifugo.db`	Protects against data loss or corruption.	

19 Knowledge-check questions

KNOWLEDGE CHECK

Attempt these before looking at the answers below.

Q1. Where should production secrets (API keys, database URLs, signing keys) be stored: in the source code, in a `.env` file committed to Git, or in the hosting platform's dashboard?

Q2. What does `git revert` do, and why is it safer than manually deleting files after a bad deployment?

Q3. Name one difference between a staging environment and a production environment.

Q4. True or false: If the application works on your laptop, it is ready to deploy to real users without further testing.

Q5. What should `.gitignore` contain for a Node.js application?

[PAGE BREAK — ANSWER KEY]

Answers

A1. In the hosting platform's dashboard (for example, Render's environment-variable settings). Secrets must never appear in source code or be committed to Git. [S-01, G-05]

A2. `git revert` creates a new commit that undoes the changes introduced by an earlier commit, while keeping the full history intact. It is safer than manual deletion because the reversal is a recorded, reviewable change, and you can still see what the broken version contained. `H-01`

A3. Staging is a private, production-like environment used for final verification before release; production is the live environment serving real users. Problems found in staging never reach customers. [Q-07 guidance; H-04 is an evidence gap, presented as recommendation]

19 Knowledge-check questions (continued)

A4. False. Local success does not guarantee production behavior — environment variables, data, and real traffic differ. Always run tests before deploying and smoke test after deploying. [M-16, S-08]

A5. `node_modules/`, `.env`, `.env.local`, database files (`*.db`, `mifugo.db`), build output (`dist/`, `build/`), and `.DS_Store`. [Q-06]

20 Links to earlier and later chapters

WHERE THIS SITS IN THE BOOK

Earlier chapters (prerequisites and context):

- **Chapter 1:** HTTP, client-server model, and status codes — what happens when a visitor opens your URL.
- **Chapter 6:** First Git commit, first PaaS deploy, `.env`, and `.gitignore`.
- **Chapter 10:** The backend and API routes you are now deploying.
- **Chapter 11:** Authentication and HTTPS that must survive the move to production.
- **Chapter 12:** Tests that must pass before every deploy.
- **Chapter 13:** The pre-deploy security checklist to complete before going live.

Later chapters (where this is used next):

- **Chapter 15:** Ship-readiness decision, license choice, and honest product claims.
- **Chapter 16:** Capstone — shipping Mifugo Market end-to-end; this chapter's deploy process is the launchpad.
- **Chapter 17:** Troubleshooting tree for failed builds and deploys, plus your next-project plan.

20 Links to earlier and later chapters (continued)

Companion website: Current Render pricing, free-tier limits, Kenyan payment accessibility, and updated deploy steps — flagged "verify current — see companion website" because platform-specific instructions date quickly. [T]

Time estimate: Approximately 230 minutes total (100 minutes reading, 130 minutes hands-on practice).

Evidence notes: Claims in this chapter trace to evidence-register IDs H-01 (Git), H-02 (PaaS deployment), H-03 (environment variables), H-04 (staging vs production — Class D gap, presented as guidance), G-05 (secrets), Q-06–Q-08 (quality practices), S-01/S-02/S-08–S-11 (security practices), F-06 (regression), and R-09 (Kenyan payment accessibility). Time-sensitive entries are flagged "[verify current]."

PART 7

7

Product & Responsibility

Validation, viability, and trust

15 From Prototype to Product: Validation, Viability, and Trust

PART 7 · PRODUCT & RESPONSIBILITY

15

From Prototype to Product: Validation, Viability, and Trust

01 Opening problem — "shipping means I'm done"

● SLOP DETECTOR — THE MISCONCEPTION

The scenario

Wanjiku's Mifugo Market is live at a public URL. Customers can browse produce, place orders, and log in. She has deployed it, tested it, and secured it. She considers the project finished.

But three months later, she has only five regular customers, she is paying for hosting she cannot clearly afford, and she has not confirmed whether anyone actually values the app enough to keep using it. She called her prototype an MVP and shipped it to real users without knowing whether there was demand, without a license on her code, and without an honest description of what the app does and does not promise.

Shipping was not the end. It was the beginning of a different set of questions: Do people want this? Can it sustain itself? Is it safe and honest to offer to real users?

The misconception

01 Opening problem — "shipping means I'm done"

(continued)

MISCONCEPTION	WHY IT IS DANGEROUS	THE CORRECTION
"A prototype that works is ready for customers."	A throwaway prototype lacks security, testing, backups, and monitoring. Shipping it to real users exposes their data and your reputation.	Prototype, MVP, and production are distinct stages with different criteria (see maturity ladder, Section 8). [M-15, R-04]
"Shipping means I'm done."	After launch, you still need users, a business model, a license, and an honest description of what your app promises.	Launch starts validation, viability, and trust work — not the end of it. [Constitution §3]

01 Opening problem — "shipping means I'm done"

(continued) (continued)

MISCONCEPTION	WHY IT IS DANGEROUS	THE CORRECTION
"AI-built code is automatically mine to commercialize."	Ownership of AI-generated code is unsettled and varies by jurisdiction. Assuming ownership can create legal problems later.	Disclose uncertainty; do not assert a rule; consult an attorney for commercial use. [R-03, I-04]

02 Why this topic matters in real software projects

Building an application is only half the work. The other half is deciding whether the application should exist at all — whether real people use it, whether it can sustain itself, and whether it is honest and safe to offer to the public.

Constitution §4 names commercial readiness as a learning outcome: the reader can assess whether their prototype has users, describe one revenue model, and state their IP position. This chapter covers that final stage of the learning arc: Understand → Direct → Build → Protect → Ship → Sustain.

Without this stage, a reader can ship an insecure, unwanted, or legally unclear application and not know it. With it, the reader can make a reasoned go/no-go decision about whether to expose real users to their software.

03 Learning objectives

WHAT YOU WILL BE ABLE TO DO

By the end of this chapter, the reader can:

03 Learning objectives (continued)

- Distinguish demo, prototype, MVP, and production system, and state criteria for each.
- Assess whether Mifugo Market has real users (validation).

03 Learning objectives (continued)

- Describe one revenue model appropriate for a small produce-ordering app.
- State their IP position and choose a license (MIT, Apache 2.0, or proprietary), with a disclaimer that this is not legal advice.

03 Learning objectives (continued)

- Write a responsible claims statement — one honest sentence about what the app does and does not promise.
- Decide, with reasoning, whether the app is ready for real users (ship-readiness).

04 Prerequisites and what to skip

BEFORE YOU START

Prerequisites

Chapters 1–14. You should have a deployed, tested, secured application (Mifugo Market or your own) from Chapters 9–14.

What you can skip

04 Prerequisites and what to skip (continued)

No new code in this chapter. You will write artifacts: a ship-readiness checklist, a license choice, and an honest product description. Jurisdiction-specific legal advice is excluded (Constitution §6) — licensing is covered as awareness only, and you should consult a qualified attorney for your jurisdiction.

05 Plain-English explanation

Demo, prototype, MVP, production: four stages, four standards

These terms are often used interchangeably, but they mean different things:

- **Demo:** A quick demonstration, possibly not functional, used to show an idea. Not for real users.
- **Prototype:** A working model built to learn or communicate. May be incomplete or insecure. Not for real users.
- **MVP (minimum viable product):** The smallest version that delivers real value to real users so you can learn. Built for real users, but limited in scope.
- **Production system:** Software real users depend on, with security, testing, backups, monitoring, and a rollback plan.

Calling a prototype an MVP is dangerous: it leads to shipping insecure software to real users. (The demo/prototype/MVP/production distinction is addressed as labeled guidance — see research gap R-04.)

Validation: do real people want this?

Validation means gathering evidence that real users have the problem you solved and will use your solution. It is not asking friends "do you like it?" It is observing behavior: sign-ups, repeat orders, messages from customers.

05 Plain-English explanation (continued)

For Mifugo Market, validation means: do shop owners in Ruiru and Nairobi actually place produce orders through the app, and do they come back?

Revenue model: how does it sustain itself?

A **revenue model** describes how the application covers its costs (hosting, domain, time). For a small produce-ordering app, options include:

05 Plain-English explanation (continued)

- A small commission on each order.
- A monthly subscription for the shop owner.
- A delivery fee paid by the customer.

The point is not to pick the perfect model. It is to have a plausible one and test whether customers will pay.

IP and licensing: who owns the code?

Intellectual property (IP) is the legal ownership of creative work, including code. Without a license, default copyright law reserves all rights to the author, and others cannot legally use the code I-04 .

Common choices:

- **MIT:** Permissive, simple, allows commercial use and closed-source derivatives; requires only copyright/license notice preservation. Good for small tools and libraries. I-02
- **Apache 2.0:** Permissive with an explicit patent grant; enterprise-friendly; requires stating changes and preserving a NOTICE file. Good for SaaS backends. I-03
- **Proprietary (all rights reserved):** You keep all rights; others cannot use, modify, or distribute without your permission.

Responsible claims: honesty about what your app does

05 Plain-English explanation (continued) (continued)

A **responsible claims statement** is an honest description of what your application does and does not promise. It avoids overstating AI involvement, security, or reliability. It is the opposite of marketing hype.

06 Mental model — and where it breaks down

Analogy: a roadside produce stand growing into a shop

Think of your software like Wanjiku's produce business:

- **Demo:** She shows a neighbor a photo of the produce she plans to sell.
- **Prototype:** She sets up a temporary table by the road and lets a few friends buy from it to see if anyone is interested.
- **MVP:** She opens a small stall with a limited range of produce, real prices, and real customers, to learn what sells.
- **Production:** She runs a permanent shop with scales, receipts, stock records, opening hours, and a plan for when she is sick.

Where the analogy breaks down

Unlike a physical stall, software can be copied instantly, accessed from anywhere, and can fail silently at scale. A produce stand that collapses affects one day's sales; an insecure app that leaks customer data affects many people at once. Software needs stricter standards at the production stage than a physical stall does.

07 Technical explanation with correct terminology

Each term is defined at first use below and listed in the chapter glossary (Section 18).

07 Technical explanation with correct terminology

(continued)

- **Demo** — a quick, possibly non-functional demonstration of an idea. Not for real users.
- **Prototype** — a working model built to learn or communicate. May be incomplete or insecure. Not for real users.
- **MVP (minimum viable product)** — the smallest version that delivers real value to real users so you can learn. [R-04 — labeled guidance]
- **Production system** — software real users depend on, with security, testing, backups, monitoring, and rollback.
- **Validation** — gathering evidence that real users have the problem and will use your solution.
- **Revenue model** — how the application covers its costs (commission, subscription, delivery fee, etc.).
- **Intellectual property (IP)** — the legal ownership of creative work, including code.
- **License** — a legal statement of what others may do with your code (MIT, Apache 2.0, proprietary, etc.).
- **MIT license** — permissive; allows commercial use and closed-source derivatives; requires copyright/license notice preservation. I-02
- **Apache 2.0 license** — permissive with explicit patent grant; requires stating changes and preserving a NOTICE file. I-03
- **Proprietary / all rights reserved** — you keep all rights; others need your permission to use the code. I-04
- **Responsible claims statement** — an honest description of what the app does and does not promise.
- **Terms of service** — the rules users agree to when using your application.
- **Ship-readiness** — a reasoned decision, based on a checklist, about whether to expose real users to the software.

07 Technical explanation with correct terminology

(continued)

- **Go/no-go decision** — the written choice to launch (go) or not launch (no-go), with reasoning.

07 Technical explanation with correct terminology

(continued)

[security] — none directly (product chapter; security is Chapter 13; this chapter's ship-readiness gate references it)

08 Concrete example — Mifugo Market maturity assessment

IN PRACTICE

Wanjiku evaluates Mifugo Market against the maturity ladder:

STAGE	CRITERIA	MIFUGO MARKET STATUS
Demo	Shows the idea works	✓ Landing page exists
Prototype	Working model for learning	✓ Customers can place orders
MVP	Real value for real users, limited scope	! Five regular customers; needs validation
Production	Secure, tested, monitored, maintainable	! Security and tests done; monitoring and backups need confirmation

She writes her ship-readiness checklist:

08 Concrete example — Mifugo Market maturity assessment (continued)

- [ ] Tests pass (Chapter 12)
- [ ] Security checklist complete (Chapter 13)
- [ ] Deployed with HTTPS and environment variables (Chapter 14)
- [ ] Real users confirmed (validation — this chapter)
- [ ] Revenue model identified (this chapter)
- [ ] License chosen and LICENSE file added (this chapter)
- [ ] Honest product description written (this chapter)
- [ ] Go/no-go decision documented (this chapter)

She chooses the MIT license for her repo (permissive, simple, allows others to use it). She writes: "Mifugo Market helps small shops order fresh produce. It is not a substitute for direct supplier relationships, and produce availability is not guaranteed."

She decides: **Go** — with the caveat that she will monitor usage for one month and reassess.

09 Step-by-step demonstration — ship-readiness for your application

STEP BY STEP

Step 1: Classify your application on the maturity ladder

Answer honestly:

- Is this a demo (showing an idea)?
- Is this a prototype (built to learn, not for real users)?
- Is this an MVP (real value for real users, limited scope)?
- Is this a production system (secure, tested, monitored)?

Be honest. A prototype called an MVP is still a prototype.

09 Step-by-step demonstration — ship-readiness for your application (continued)

Step 2: Validate with real users

Choose one validation method:

- **Conversation:** Talk to five real users (not friends) about whether they would use your app and what they currently do instead.
- **Usage data:** If your app is live, check how many people placed orders in the last 30 days and how many returned.
- **Payment intent:** Ask users whether they would pay, and if so, how much.

09 Step-by-step demonstration — ship-readiness for your application (continued)

Record the evidence. If you have no evidence of real users, you are not at MVP yet.

Step 3: Choose a revenue model

Pick one plausible model for your application. For Mifugo Market: a small commission on each order placed through the app. Write one sentence describing it.

Step 4: Choose a license

Decide how others may use your code:

09 Step-by-step demonstration — ship-readiness for your application (continued)

- **MIT** if you want maximum adoption and simplicity.
- **Apache 2.0** if you want patent protection and enterprise friendliness.
- **Proprietary** if you want to keep all rights.

Add a `LICENSE` file to your repository with the chosen license text. [I-01–I-04]

[verify current] — License texts are stable, but which license suits your situation depends on your goals and jurisdiction. This is guidance, not legal advice. Consult a qualified attorney before choosing a license for commercial use. [R-03, I-04]

Step 5: Write a responsible claims statement

Write one honest sentence describing what your app does and does not promise. Example:

"Mifugo Market helps small shops order fresh produce online. It does not guarantee produce availability, delivery times, or prices."

Avoid: "AI-built," "guaranteed secure," "the best," or any claim you cannot support.

Step 6: Make the go/no-go decision

Review your ship-readiness checklist. Write a one-paragraph decision:

- **Go:** you have evidence of users, a plausible revenue model, a license, an honest description, and the security/testing gates are passed.
- **No-go:** you lack evidence of users, or the security/testing gates are not passed, or you are not ready to be responsible for real users' data.

09 Step-by-step demonstration — ship-readiness for your application (continued)

Document your reasoning.

10 AI collaboration example (prompt format P1–P10)

AI COLLABORATION — PROMPT FORMAT P1–P10

PROMPT FORMAT **A reusable prompt you can copy**

P1 **P1 · PROBLEM IT SOLVES**
Draft a license choice rationale, a responsible claims statement, and a ship-readiness justification without overstating capabilities.

P2 **P2 · WHEN TO USE IT**
Chapter 15, when preparing to launch Mifugo Market or your own application.

P3 **P3 · REQUIRED INPUTS**
Your validation evidence, your chosen license, your app's actual capabilities, and your ship-readiness checklist.

10 AI collaboration example (prompt format P1–P10) (continued)

P4 **P4 · THE PROMPT**

```
TEXT
Role: You are a careful product advisor.
Context: Mifugo Market (Chapter 15). A fresh-produce
ordering app for small shops in Nairobi. Live at [URL].
Validation evidence: [SUMMARY]. Chosen license: [MIT /
Apache 2.0 / proprietary].
Task: Draft:
1. A one-paragraph justification for the chosen license.
2. A responsible claims statement (one honest sentence
about what the app does and does not promise).
3. A one-paragraph go/no-go decision with reasoning, based
on the ship-readiness checklist.

Constraints:
- Do not overstate AI involvement, security, or
reliability.
- Do not assert legal ownership of AI-generated code.
- Mark assumptions as [ASSUMPTION].
- Do not invent user statistics.
```

10 AI collaboration example (prompt format P1–P10) (continued)

P5 **P5 · WHAT A GOOD RESPONSE CONTAINS**

License rationale; honest claims statement (limitations included); go/no-go decision with reasoning; assumptions labeled; no invented statistics.

10 AI collaboration example (prompt format P1–P10) (continued) (continued)

P6 **P6 · WHAT THE READER MUST INSPECT**
Confirm no overclaiming; confirm assumptions are flagged; confirm the decision matches your actual evidence; confirm no legal certainty is asserted.

P7 **P7 · WEAK OR INCOMPLETE RESPONSE**
Claims the app is "production-ready" without evidence; asserts IP ownership as fact; invents user numbers; omits limitations.

10 AI collaboration example (prompt format P1–P10) (continued) (continued)

P8 **P8 · FOLLOW-UP WHEN INADEQUATE**
"Remove the claim that the app is secure," "Which of these is an assumption?" "Add the limitation that availability is not guaranteed."

P9 **P9 · SAFETY/SECURITY CONSIDERATIONS**
Do not assert legal conclusions about IP ownership (R-03 — unsettled law). Do not promise security you have not verified (Chapter 13).

10 AI collaboration example (prompt format P1–P10) (continued) (continued)

P10 **P10 · VERIFICATION STEP**
Compare the AI's draft against your actual validation evidence and checklist. Every claim should trace to something you observed or decided.

11 Human decisions and responsibilities (non-delegable judgment)

● YOUR DECISION — NOT THE AI'S

- **Whether to launch.** You decide if real users should use your software; the AI cannot judge ship-readiness.
- **Which license to choose.** You decide based on your goals; the AI can explain options but cannot choose for you.
- **What to promise users.** You write the honest description; the AI may be tempted to overstate.
- **Whether to consult an attorney.** You decide when legal questions (IP, terms, jurisdiction) arise.

[Human decision-point box] Before you announce your application to real users, answer: Do I have evidence of real users? Do I have a plausible revenue model? Is my license chosen and added? Is my product description honest about limitations? Have I passed the security and testing gates? If any answer is no, do not launch yet.

12 Inspect and verify AI output (ship-readiness checklist)

● QUALITY CHECK

Before launching, confirm:

12 Inspect and verify AI output (ship-readiness checklist) (continued)

- ✓ [ ] You can name real users (not friends) who have used the app.
- ✓ [ ] You have a written revenue model (one sentence).
- ✓ [ ] A LICENSE file exists in your repository.
- ✓ [ ] Your product description states what the app does *and* does not promise.
- ✓ [ ] The security checklist from Chapter 13 is complete.
- ✓ [ ] Tests pass from Chapter 12.
- ✓ [ ] The app is deployed with HTTPS and environment variables (Chapter 14).
- ✓ [ ] Your go/no-go decision is written with reasoning.

13 Failure modes and recovery

WHEN IT GOES WRONG

FAILURE	SYMPTOM	LIKELY CAUSE	RECOVERY
Prototype called MVP	Real users on insecure software	Labels used loosely without criteria	Apply the maturity ladder; secure before launching
No evidence of users	Launch, then silence	No validation before launch	Pause promotion; run conversations or usage checks
Overpromising	Marketing claims exceed capabilities	AI-generated or enthusiastic language	Rewrite claims statement honestly; correct public materials
No license	Others cannot legally use your code	No LICENSE file added	Add a license; consult an attorney if commercial
Not knowing when to hire	You are stuck on issues beyond your skill	No plan for external help	Define when to hire a developer vs. continue solo

All scenarios in this section are hypothetical and labeled as such.

14 Professional standard — quality threshold before launch

PRO STANDARD

An application is ready to launch to real users only when:

- **Validation:** Evidence of real users (conversations, sign-ups, or usage data).
- **Revenue model:** A plausible model for covering costs is written down.
- **License:** A LICENSE file exists; the choice is reasoned, not random. [I-01–I-04]
- **Honest description:** Claims state what the app does and does not promise.
- **Security and testing gates:** Chapter 13 checklist complete; Chapter 12 tests pass.
- **Deployment:** HTTPS and environment variables in place (Chapter 14).
- **Go/no-go decision:** Written, with reasoning, based on the above.

These criteria are the author team's recommended practice, informed by licensing literature [I-01–I-04] and product-readiness thinking (Constitution §3, §4; R-04 labeled guidance).

15 Hands-on exercise (L3 — Application)

BUILD IT

Task: Complete the ship-readiness checklist for your application (Mifugo Market or your own), choose a license, and write an honest product description.

Constraints:

15 Hands-on exercise (L3 — Application) (continued)

- No code changes required.
- Base your validation evidence on real observations, not assumptions (label any assumptions).
- Do not assert legal conclusions about IP ownership.

15 Hands-on exercise (L3 — Application) (continued)

Deliverable: A completed ship-readiness checklist, a LICENSE file, and a one-paragraph honest product description.

Success criteria:

15 Hands-on exercise (L3 — Application) (continued)

- Checklist items from Section 12 are all addressed.
- LICENSE file added to the repository.
- Product description mentions at least one limitation.
- Go/no-go decision written with reasoning.

15 Hands-on exercise (L3 — Application) (continued)

Assessment level: L3 (Application — performs a procedure with guidance)

Time estimate: 70–90 minutes.

16 Build challenge (L4 — Analysis) with acceptance criteria

● PUSH FURTHER

Task: Classify your application on the maturity ladder, identify gaps between your current state and production readiness, and document a go/no-go decision.

Constraints:

- Do not ask the AI to regenerate the app.
- Work with your actual deployed application and evidence.
- Document reasoning in writing.

Acceptance criteria (Given–When–Then):

- **Given** the maturity ladder (demo → prototype → MVP → production), **When** you classify your application, **Then** you can state which stage it is in and why, with evidence.
- **Given** the ship-readiness checklist, **When** you review each item, **Then** every item is checked off or a gap is noted with a fix plan.
- **Given** your validation evidence, **When** you assess whether real users want the app, **Then** you can name at least one observed user behavior (sign-up, order, return visit).
- **Given** your license choice, **When** you add the LICENSE file, **Then** the repository contains a license and you can state what it permits others to do.
- **Given** all evidence, **When** you make the go/no-go decision, **Then** the decision is written with reasoning and matches the evidence.

Deliverable: A written maturity classification, gap analysis, license note, and go/no-go decision with reasoning.

Assessment level: L4 (Analysis — investigates evidence, diagnoses gaps, makes a judgment)

16 Build challenge (L4 — Analysis) with acceptance criteria (continued)

Time estimate: 60–75 minutes.

17 Key takeaways

KEY TAKEAWAYS

- Demo, prototype, MVP, and production are different stages with different standards — do not confuse them.
- Shipping is not the end; it is the start of validation, viability, and trust work.
- Validation means evidence from real users, not opinions from friends.
- A revenue model answers how the app covers its costs.
- Choose a license (MIT, Apache 2.0, or proprietary) and add a LICENSE file; this is guidance, not legal advice.
- Write an honest claims statement that states limitations, not just features.
- Make a written go/no-go decision with reasoning before exposing real users.

18 Glossary of new terms

TERMS INTRODUCED IN THIS CHAPTER

18 Glossary of new terms (continued)

TERM	PLAIN-LANGUAGE DEFINITION	EXAMPLE	WHY IT MATTERS	TAG
Demo	A quick, possibly non-functional demonstration of an idea.	A screenshot or mockup shown to a friend	Shows the idea; not for real users.	
Prototype	A working model built to learn or communicate; may be incomplete.	Mifugo Market's first order form	Tests an idea; not secure enough for real users.	
MVP	The smallest version delivering real value to real users so you can learn.	Mifugo Market with real orders from five shops	Real users, limited scope; a step toward production.	[R-04]
Production system	Software real users depend on, with security, testing, backups, monitoring.	Mifugo Market after passing all gates	The standard for public-facing software.	
Validation	Evidence that real users have the problem and will use your solution.	Five shop owners placing orders	Confirms demand before you invest more.	
Revenue model	How the application covers its costs.	Commission on each order	Determines whether the app can sustain itself.	
Intellectual property (IP)	Legal ownership of creative work, including code.	Wanjiku owns the Mifugo Market code	Determines who can use, modify, or sell the code.	
License	A legal statement of what others may do with your code.	MIT License file in the repo	Lets others use your code under stated terms.	
MIT license	Permissive license; allows commercial use; requires notice preservation.	MIT LICENSE file	Simple, widely used, good for small tools.	I-02

18 Glossary of new terms (continued)

TERM	PLAIN-LANGUAGE DEFINITION	EXAMPLE	WHY IT MATTERS	TAG
Apache 2.0 license	Permissive with patent grant; requires stating changes.	Apache LICENSE file	Enterprise-friendly; good for backends.	I-03
Proprietary / all rights reserved	You keep all rights; others need permission to use.	No LICENSE file; all rights reserved	Maximum control; least sharing.	I-04
Responsible claims statement	Honest description of what the app does and does not promise.	"Helps shops order produce; availability not guaranteed"	Builds trust; avoids overhyping AI or security.	
Terms of service	Rules users agree to when using your application.	"Users must be 18+"	Sets expectations and limits liability.	
Ship-readiness	A reasoned decision, based on a checklist, about launching to real users.	Passed security + tests + validation	The gate before public launch.	
Go/no-go decision	A written choice to launch or not, with reasoning.	"Go — will monitor for one month"	Documents the judgment behind launching.	

19 Knowledge-check questions

KNOWLEDGE CHECK

Attempt these before looking at the answers below.

Q1. What is the difference between a prototype and an MVP?

Q2. Name one revenue model appropriate for a small produce-ordering app.

Q3. What is the difference between the MIT and Apache 2.0 licenses?

19 Knowledge-check questions (continued)

Q4. True or false: Adding a LICENSE file means you own the copyright to AI-generated code.

Q5. What should a responsible claims statement include?

[PAGE BREAK — ANSWER KEY]

Answers

19 Knowledge-check questions (continued)

A1. A prototype is built to learn or communicate and is not for real users. An MVP is the smallest version that delivers real value to real users so you can learn. [R-04 — labeled guidance]

A2. Any one of: a small commission on each order; a monthly subscription for the shop owner; a delivery fee paid by the customer.

19 Knowledge-check questions (continued)

A3. MIT is permissive and simple, requiring only copyright/license notice preservation. Apache 2.0 is also permissive but adds an explicit patent grant and requires stating changes and preserving a NOTICE file. I-02 · I-03

19 Knowledge-check questions (continued)

A4. False. Ownership of AI-generated code is unsettled and varies by jurisdiction. A LICENSE file states what others may do with your code, but it does not resolve AI ownership questions. Consult an attorney. [R-03, I-04]

19 Knowledge-check questions (continued)

A5. What the app does *and* what it does not promise; it states limitations honestly and avoids overstating AI involvement, security, or reliability.

20 Links to earlier and later chapters

WHERE THIS SITS IN THE BOOK

Earlier chapters (prerequisites):

- **Chapter 13:** Security, Privacy, and Data Protection — the security checklist is part of ship-readiness.
- **Chapter 12:** Testing, Debugging, and Regression Prevention — tests must pass before launch.
- **Chapter 14:** Version Control, Deployment, and Operations — HTTPS and environment variables must be in place.

Later chapters (where this is used next):

- **Chapter 16:** Capstone Project — the go/no-go decision, license, and honest description are required capstone deliverables.
- **Chapter 17:** Your Independent Practice — the revenue model and license choice inform your next project plan.

Research gap notes:

- **R-03:** IP ownership in AI-generated code is unsettled and varies by jurisdiction. This chapter discloses uncertainty and recommends attorney consultation; it does not assert legal rules. I-04
- **R-04:** Demo/prototype/MVP/production definitions have no primary source located. The maturity ladder in Section 8 is presented as labeled practitioner guidance, not an industry standard. [M-15]

20 Links to earlier and later chapters (continued)

Evidence notes: Claims trace to I-01–I-04 (licensing), R-03/R-04 (labeled guidance gaps), Constitution §3 and §4, M-15 (misconception register). Kenya DPA/GDPR awareness from Chapter 13 applies to any user data collected. All scenarios are hypothetical.

Time estimate: Approximately 150 minutes total (80 minutes reading, 70 minutes hands-on practice).

PART 8

8 Capstone

Ship Mifugo Market, then practise alone

16 Capstone Project — Ship Mifugo Market End-to-End

17 Your Independent Practice

PART 8 · CAPSTONE

16

Capstone Project — Ship Mifugo Market End-to-End

01 Opening problem — "I've learned the pieces but can't put them together"

● SLOP DETECTOR — THE MISCONCEPTION

The scenario

Wanjiku has worked through all the chapters. She knows how to validate a problem, write user stories, specify acceptance criteria, prompt an AI, review a diff, build a frontend, connect a backend, add authentication, write tests, run a security checklist, and deploy to Render. But she has never done all of it at once, in order, with real decisions at each step.

She faces a blank repository and does not know where to start. The pieces feel separate: the problem statement from Chapter 3, the acceptance criteria from Chapter 5, the tests from Chapter 12, the security checklist from Chapter 13, the deploy from Chapter 14. She needs to integrate them into one shipped application.

This chapter is that integration. It is the capstone: one project, four weeks, one live URL.

The misconception

MISCONCEPTION	WHY IT IS DANGEROUS	THE CORRECTION
"I can learn each piece separately and assemble them at the end."	Software development is not a collection of independent skills. Decisions in one chapter affect every other: your data model shapes your API, your authorization shapes your tests, your deployment shapes your security.	The capstone forces integration: every decision affects the whole, and you feel the consequences.
"The capstone is just a bigger version of earlier exercises."	Earlier exercises build one feature or one skill. The capstone requires trade-offs: scope, time, security, and cost all compete. You must decide what to cut.	Scope control is part of the project, not an obstacle to it.

02 Why this topic matters in real software projects

Constitution quality criterion #1 states: a reader with no coding background can complete the capstone project and deploy a functioning, authenticated web application to a public URL. This chapter is where that promise is fulfilled.

Building an end-to-end application teaches something no single chapter can: how decisions connect. The database schema you choose in week 2 determines what your tests check in week 3. The authorization rules you set in week 3 determine what your security checklist verifies in week 3. The deployment in week 4 reveals whether your environment variables and build commands actually work.

The capstone is also the evidence of competence. A live URL, a public repository, and a completed rubric show that you can direct AI, review code, test, secure, and deploy — not just read about it.

03 Learning objectives

WHAT YOU WILL BE ABLE TO DO

By the end of this chapter, the reader can:

03 Learning objectives (continued)

- Take Mifugo Market from problem statement to live URL, applying every earlier chapter.
- Write a one-page specification with 3–5 acceptance criteria per core feature.
- Build a responsive, accessible frontend: product catalog, order form, login, owner dashboard.
- Build a backend API: product list, order creation, order retrieval.
- Implement a database: products, orders, users.
- Implement authentication: sign-up/sign-in with password hashing.
- Implement authorization: customers see own orders; owner sees all; object-level checks on `/orders/:id`.
- Write and run a test suite: 3+ unit tests and 1 smoke test, all passing.
- Apply the pre-deploy security checklist.

03 Learning objectives (continued)

- Deploy with environment variables in the platform dashboard (not in repo).
- Set up basic monitoring and a monthly cost estimate.
- Document the project: README with setup instructions, acceptance criteria, and deploy checklist.
- Add a license file and write an honest product description.
- Make a go/no-go ship decision with written reasoning.

04 Prerequisites and what to skip

BEFORE YOU START

Prerequisites

Chapters 1–15. Every prior chapter feeds into the capstone:

- Chapters 3–5: Problem, users, specification, acceptance criteria.
- Chapters 6–8: Git, editor, context file, prompting, diff review.
- Chapters 9–11: Frontend, backend, authentication, authorization.
- Chapters 12–13: Tests, security checklist.
- Chapter 14: Deployment, environment variables, rollback.
- Chapter 15: Ship-readiness, license, honest claims.

What you can skip

Nothing — this chapter integrates everything. But you do not need to invent new concepts; you apply what you already learned.

05 Plain-English explanation

End-to-end delivery means one URL, one repository, one owner

"End-to-end" means you take an idea through every stage — problem, specification, build, test, secure, deploy, operate — and finish with one public URL that real people can use, one repository that contains the code, and one person (you) who understands how it all fits together.

It does not mean the application is perfect. It means it is complete enough to show real users, safe enough to handle their data, and documented enough that someone else could take it over.

The capstone is a guided project, not an exam

You are not alone. This chapter walks you through four weeks of work, with milestones, checklists, and decision points at each stage. At the end, you assess yourself against a rubric. You may revise and resubmit.

06 Mental model — and where it breaks down

Analogy: building a market stall from scratch

The capstone is like building a market stall:

- Week 1: You decide what to sell, who your customers are, and where the stall goes (specification).
- Week 2: You build the stall, set up the shelves, and price the produce (build).
- Week 3: You check the scales work, the produce is fresh, and the cash box locks (test and secure).
- Week 4: You open the stall, put up a sign, and count the takings each evening (deploy and operate).

Where the analogy breaks down

Unlike a market stall, software can reach customers far beyond your neighborhood, can fail at 2 a.m. with no one to notice, and stores data that needs protection. The capstone adds digital guardrails — tests, security, monitoring — that a physical stall does not need.

07 Technical explanation with correct terminology

Each term is defined at first use below and listed in the chapter glossary (Section 18).

07 Technical explanation with correct terminology

(continued)

- **Capstone project** — the integrative final project: ship Mifugo Market end-to-end.
- **End-to-end delivery** — taking an idea through every stage to a live, public application.
- **Ship-readiness decision** — a reasoned go/no-go based on a checklist.
- **Documentation** — README with setup instructions, acceptance criteria, and deploy checklist.
- **Handoff** — making the project understandable to someone else.
- **Post-launch checklist** — tasks after launch: monitor, backups, cost review.

07 Technical explanation with correct terminology

(continued)

- **Maintenance plan** — how the application will be kept running over time.
- **Milestone** — a dated checkpoint with specific deliverables.

07 Technical explanation with correct terminology

(continued)

[security] — Ship-readiness decision, Documentation (the security checklist and tests are prerequisites for shipping)

08 Concrete example — Mifugo Market capstone roadmap

IN PRACTICE

Week 1 — Specify:

- Refine the problem statement (Chapter 3).
- Confirm the primary persona and user journey (Chapter 4).
- Write the specification with 3–5 acceptance criteria per feature (Chapter 5).
- Deliverable: one-page spec + acceptance criteria.

Week 2 — Build:

- Build the frontend: product catalog, order form, login, owner dashboard (Chapters 9–11).
- Build the backend API: product list, order creation, order retrieval (Chapter 10).
- Implement the database: products, orders, users (Chapters 10–11).
- Deliverable: working application locally.

Week 3 — Test and secure:

- Write 3+ unit tests and 1 smoke test; all pass (Chapter 12).
- Run the pre-deploy security checklist (Chapter 13).
- Diff-review every AI-generated change (Chapter 8).
- Deliverable: passing test suite + completed security checklist.

Week 4 — Deploy and operate:

08 Concrete example — Mifugo Market capstone roadmap (continued)

- Deploy to Render with environment variables in the dashboard (Chapter 14).
- Set up monitoring and a monthly cost estimate.
- Add a LICENSE file and honest product description (Chapter 15).
- Make the go/no-go ship decision.
- Deliverable: live URL + repository + documentation + rubric.

09 Step-by-step demonstration — the four-week capstone

STEP BY STEP

Week 1: Specify

Day 1–2: Refine your problem statement. Confirm who the primary user is and what problem you are solving. Use Chapter 3's format.

Day 3–4: Confirm the persona and user journey. Use Chapter 4's format: five steps, one decision point, two edge cases.

Day 5–7: Write the specification. One page, with goal, users, scope, constraints, 3–5 Given-When-Then acceptance criteria per feature, and an out-of-scope list. Use Chapter 5's format.

Deliverable: One-page specification + acceptance criteria.

Checkpoint: Can you explain, in one paragraph, what Mifugo Market does, for whom, and what is out of scope? If not, revise before building.

Week 2: Build

09 Step-by-step demonstration — the four-week capstone (continued)

Day 8–10: Build the frontend. Product catalog, order inquiry form, login page, owner dashboard. Responsive, mobile-first, accessible (Chapter 9). Review every AI-generated diff (Chapter 8).

Day 11–13: Build the backend API. `GET /api/products`, `POST /api/orders`, `GET /api/orders/:id`. Database: products, orders, users tables (Chapter 10).

Day 14–15: Implement authentication and authorization. Sign-up/sign-in with password hashing. Object-level authorization: customers see own orders; owner sees all; `/orders/:id` returns 403 for wrong user (Chapter 11).

Deliverable: Working application locally, with frontend, backend, database, and auth.

09 Step-by-step demonstration — the four-week capstone (continued)

Checkpoint: Can a customer sign up, sign in, place an order, and see only their own orders? Can the owner see all orders? If not, fix before testing.

Week 3: Test and secure

Day 16–17: Write tests. 3+ unit tests (price calculation, stock check, discount logic) and 1 smoke test (homepage loads, order form submits). All pass (Chapter 12).

Day 18–19: Run the pre-deploy security checklist. No hardcoded secrets. HTTPS on. Authorization on every route. Input validated. Secrets in the platform dashboard (Chapter 13).

09 Step-by-step demonstration — the four-week capstone (continued) (continued)

Day 20–21: Diff-review every remaining AI-generated change. Fix any security-critical issues (Chapter 8).

Deliverable: Passing test suite + completed security checklist.

Checkpoint: Do all tests pass? Is the security checklist complete? If not, do not deploy.

Week 4: Deploy and operate

Day 22–23: Deploy to Render. Connect GitHub, set build/start commands, set environment variables in the dashboard (not in code), confirm HTTPS, smoke test the live URL (Chapter 14).

09 Step-by-step demonstration — the four-week capstone (continued) (continued)

Day 24–25: Set up monitoring and a monthly cost estimate. Know how to roll back if a deploy breaks (Chapter 14).

Day 26–27: Document. README with setup instructions, acceptance criteria, and deploy checklist. Add LICENSE file. Write honest product description (Chapter 15).

Day 28: Make the go/no-go decision. Written, with reasoning. If go, announce the live URL. If no-go, document what must change.

09 Step-by-step demonstration — the four-week capstone (continued)

Deliverable: Live URL + repository + documentation + completed capstone rubric.

Checkpoint: Is the application live at a public URL with HTTPS? Can you point someone to it and have them use it?

10 AI collaboration example (prompt format P1–P10)

AI COLLABORATION — PROMPT FORMAT P1–P10

PROMPT FORMAT **A reusable prompt you can copy**

P1 **P1 · PROBLEM IT SOLVES**
Get AI help for capstone milestones without losing control or introducing slop.

P2 **P2 · WHEN TO USE IT**
Chapter 16, throughout the four-week capstone.

P3 **P3 · REQUIRED INPUTS**
The current milestone, the specification and acceptance criteria from Week 1, the context file (AGENTS.md), and the relevant chapter's checklist.

10 AI collaboration example (prompt format P1–P10) (continued)

P4 **P4 · THE PROMPT**

```
TEXT
Role: You are a careful software builder helping with the
Mifugo Market capstone.
Context: Mifugo Market (Chapter 16). Week [WEEK_NUMBER]
milestone: [MILESTONE_DESCRIPTION]. Specification:
[SPEC_SUMMARY]. Acceptance criteria: [CRITERIA_SUMMARY].
Project conventions: see AGENTS.md.
Task: [TASK_DESCRIPTION].
Constraints:
- Review every diff before accepting (Chapter 8
discipline).
- No hardcoded secrets or API keys.
- Input validated on the server (Chapter 13).
- Authorization enforced on user-data routes (Chapter 11).
- Return unified diffs with file paths.
- Do not suggest scope beyond the acceptance criteria.
```

P5 **P5 · WHAT A GOOD RESPONSE CONTAINS**

Unified diff limited to the milestone task; constraints honored; no secrets; no scope creep; references to AGENTS.md.

10 AI collaboration example (prompt format P1–P10) (continued)

P6 **P6 · WHAT THE READER MUST INSPECT**

Confirm the diff matches the milestone; confirm no secrets; confirm validation and authorization are present; run tests after accepting.

10 AI collaboration example (prompt format P1–P10) (continued) (continued)

P7 **P7 · WEAK OR INCOMPLETE RESPONSE**
Suggests features outside the acceptance criteria; hardcodes a key; skips authorization; suggests whole-app rewrite.

P8 **P8 · FOLLOW-UP WHEN INADEQUATE**
"Limit the diff to [FILE] only," "Add the ownership check," "Remove the hardcoded key," "Stay within the acceptance criteria."

10 AI collaboration example (prompt format P1–P10) (continued) (continued)

P9 **P9 · SAFETY/SECURITY CONSIDERATIONS**
Authentication and authorization are security-critical paths requiring mandatory human review (D-04). Review every line of auth-related changes.

10 AI collaboration example (prompt format P1–P10) (continued) (continued)

P10 **P10 · VERIFICATION STEP**
After accepting, run the test suite; confirm the milestone behavior works; confirm no regression; commit with a meaningful message.

11 Human decisions and responsibilities (non-delegable judgment)

● YOUR DECISION — NOT THE AI'S

- **What is in scope.** You decide what features make the cut; the AI will suggest more.
- **Whether to ship.** You make the go/no-go decision based on evidence, not enthusiasm.
- **What to promise users.** You write the honest description; the AI may overstate.
- **When to ask for help.** You decide when a problem is beyond your skill and whether to hire a developer.
- **What to maintain.** You decide how long to keep the application running and what it costs.

[Human decision-point box] At the end of each week, answer: Did I meet the milestone's acceptance criteria? Are tests passing? Is the security checklist complete? Can I explain every part of the application? If any answer is no, do not proceed to the next week.

12 Inspect and verify AI output (capstone checklist)

● QUALITY CHECK

Before accepting any AI-generated code in the capstone:

- ✓ [ ] The diff was reviewed line by line (Chapter 8).
- ✓ [ ] No secrets or API keys in code or configuration.
- ✓ [ ] Authorization enforced on every user-data route (403 test passes).
- ✓ [ ] Input validated on the server.
- ✓ [ ] Tests pass after the change (no regression).
- ✓ [ ] The change stays within the acceptance criteria — no scope creep.
- ✓ [ ] You can explain what the code does.

13 Failure modes and recovery

WHEN IT GOES WRONG

FAILURE	SYMPTOM	LIKELY CAUSE	RECOVERY
Scope creep	Project grows beyond four weeks; nothing ships	Adding "just one more feature"	Freeze scope; cut features to meet the deadline
Skipping tests/security	Deploy fails or data exposed	Rushing to ship	Pause deploy; run tests and security checklist
No rollback plan	Bad deploy breaks the app; no way back	No Git revert practice	Learn `git revert`; redeploy previous version
No documentation	No one (including you) can rebuild the app	Documentation deferred	Write README before launch, not after
Shipping without go/no-go	Uncertain whether app is ready	No written decision	Complete the rubric; decide with reasoning

All scenarios in this section are hypothetical and labeled as such.

14 Professional standard — quality threshold for the capstone

PRO STANDARD

The capstone passes only when all rubric criteria are met (Section 16):

14 Professional standard — quality threshold for the capstone (continued)

- **Live URL:** Public URL reachable with HTTPS.
- **Authentication:** Sign-up/sign-in works; passwords not plaintext.
- **Authorization:** `/orders/:id` returns 403 for another user's order.
- **Testing:** 3+ unit tests and 1 smoke test, all passing before deploy.
- **Security:** No hardcoded secrets; inputs validated; pre-deploy checklist complete.
- **Accessibility:** Keyboard navigable; contrast ≥4.5:1; visible focus; alt text.

14 Professional standard — quality threshold for the capstone (continued)

- **Deployment:** PaaS deploy; env vars in dashboard (not repo); rollback demonstrated.
- **Documentation:** README explains setup, acceptance criteria, and deploy.
- **Go/no-go:** Written decision with reasoning.

14 Professional standard — quality threshold for the capstone (continued)

You may revise and resubmit. Passing the capstone demonstrates independent competence (Constitution quality criterion #1).

15 Hands-on exercise (L6 — Creation)

BUILD IT

Task: Complete the capstone project: ship Mifugo Market (or your own application) end-to-end over four weeks.

Constraints:

- Follow the four-week milestone structure (specify, build, test/secure, deploy/operate).
- Do not ask the AI to regenerate the whole application at any point.
- Review every AI-generated diff before accepting.
- Make a written go/no-go decision before announcing the live URL.

Deliverable: Live public URL, public repository, documentation (README + LICENSE + honest description), and completed capstone rubric (self-assessed).

Success criteria: All nine rubric criteria met (Section 16).

Assessment level: L6 (Creation — builds independently from requirements)

Time estimate: 4–6 weeks part-time.

16 Build challenge / capstone rubric (L6 — Creation)

PUSH FURTHER

Task: Self-assess your capstone against the pass/fail rubric. If you pass, you are eligible for optional external review by a technical reviewer. If you fail any criterion, revise and resubmit.

Capstone rubric:

16 Build challenge / capstone rubric (L6 — Creation)

(continued)

Live URL	Application reachable at a public URL (custom domain or free subdomain) with HTTPS.	Not reachable, or HTTP only, or requires local install.
Authentication	Sign-up/sign-in works; passwords not stored in plaintext.	No auth, or plaintext passwords, or auth bypassable.
Authorization	`/orders/:id` returns 403 when the order belongs to another user.	Any user can view any order by changing the URL.
Testing	3+ unit tests and 1 smoke test, all passing before deploy.	No tests, or tests failing at deploy time.
Security	No hardcoded secrets; inputs validated; pre-deploy checklist completed.	Hardcoded API key/password found; no input validation; checklist skipped.
Accessibility	Keyboard navigable; contrast ≥ 4.5:1 for body text; visible focus; alt text on images.	Keyboard traps; unreadable contrast; no focus indicator.
Deployment	Deployed via PaaS; env vars set in platform dashboard (not in repo); rollback demonstrated at least once.	`.env` committed; no rollback; deploy not repeatable.
Documentation	README explains setup, acceptance criteria, and how to deploy.	No README, or README does not allow a new person to run the app.
Go/no-go decision	Written decision to ship (or not) with reasoning based on the checklist.	Shipped without any readiness assessment.

Capstone outcome: Pass = all criteria met. You may revise and resubmit.

16 Build challenge / capstone rubric (L6 — Creation)

(continued)

Deliverable: Completed rubric with your self-assessment, plus evidence for each criterion (URL, test output, security checklist, README, go/no-go statement).

Assessment level: L6 (Creation — builds independently from requirements; evaluates own work against a standard)

Time estimate: Included in the four-week capstone timeline (final days).

17 Key takeaways

KEY TAKEAWAYS

- The capstone integrates everything: specification, build, test, secure, deploy, document, decide.
- Every decision affects the whole: data model shapes API, authorization shapes tests, deployment shapes security.
- Review every diff; run tests before every deploy; complete the security checklist before launch.
- The go/no-go decision is yours — written, with reasoning, based on evidence.
- Passing the capstone rubric demonstrates independent competence.
- You may revise and resubmit; the capstone is a learning milestone, not a one-shot test.

18 Glossary of new terms

TERMS INTRODUCED IN THIS CHAPTER

TERM	PLAIN-LANGUAGE DEFINITION	EXAMPLE	WHY IT MATTERS	TAG
Capstone project	The integrative final project: ship Mifugo Market end-to-end.	Four weeks, one live URL	Proves you can apply everything you learned.	
End-to-end delivery	Taking an idea through every stage to a live, public application.	Problem → spec → build → test → secure → deploy	Shows how decisions connect across the whole project.	
Ship-readiness decision	A reasoned go/no-go based on a checklist.	"Go — will monitor for one month"	The judgment call before exposing real users.	
Documentation	README with setup instructions, acceptance criteria, and deploy checklist.	A file that lets someone else run your app	Makes the project maintainable and handoff-ready.	
Handoff	Making the project understandable to someone else.	A new developer reads your README and runs the app	Your code outlives your personal involvement.	
Post-launch checklist	Tasks after launch: monitor, backups, cost review.	Weekly check that the homepage loads	Operations begins after deployment.	
Maintenance plan	How the application will be kept running over time.	Monthly cost review; quarterly dependency updates	Keeps the application alive after launch.	
Milestone	A dated checkpoint with specific deliverables.	Week 1: specification complete	Breaks the capstone into manageable pieces.	

19 Knowledge-check questions

KNOWLEDGE CHECK

Attempt these before looking at the answers below.

Q1. What four stages does the capstone follow, in order?

Q2. What is the go/no-go decision, and when is it made?

Q3. Name three capstone rubric criteria.

Q4. True or false: You may ask the AI to regenerate the whole application during the capstone.

Q5. What happens if you fail a capstone rubric criterion?

[PAGE BREAK — ANSWER KEY]

Answers

A1. Week 1: Specify (problem, persona, journey, specification, acceptance criteria). Week 2: Build (frontend, backend, database, auth). Week 3: Test and secure (tests, security checklist, diff review). Week 4: Deploy and operate (deploy, monitoring, documentation, go/no-go decision).

A2. A written choice to launch (go) or not launch (no-go), with reasoning based on the ship-readiness checklist. Made at the end of Week 4, before announcing the live URL.

A3. Any three of: live URL with HTTPS, authentication works, authorization enforced (403 test), 3+ unit tests + 1 smoke test passing, no hardcoded secrets, accessibility basics, PaaS deploy with env vars in dashboard, README documentation, written go/no-go decision.

A4. False. You review every diff and make the smallest changes possible; you never ask the AI to regenerate the whole application.

A5. You revise and resubmit. The capstone is a learning milestone, not a one-shot test.

20 Links to earlier and later chapters

WHERE THIS SITS IN THE BOOK

Earlier chapters (all are prerequisites):

- **Chapters 3–5:** Problem, users, specification, acceptance criteria (Week 1).
- **Chapters 6–8:** Git, editor, context file, prompting, diff review (used throughout).
- **Chapters 9–11:** Frontend, backend, authentication, authorization (Week 2).
- **Chapters 12–13:** Tests, security checklist (Week 3).
- **Chapter 14:** Deployment, environment variables, rollback (Week 4).
- **Chapter 15:** Ship-readiness, license, honest claims (Week 4).

Later chapters (where this is used next):

- **Chapter 17:** Your Independent Practice — the capstone is the bridge to independent practice; your next project plan builds on what you learned here.

Evidence notes: Claims trace to all prior evidence register IDs; Constitution §4 (project learning outcome), §13 (quality criterion #1 — reader deploys a functioning, authenticated app to a public URL), §10 (practical learning requirements). All scenarios are hypothetical.

Time estimate: 4–6 weeks part-time (capstone project, not a single sitting).

PART 8 · CAPSTONE

17 Your Independent Practice

01 Opening problem — "the book ends here"

SLOP DETECTOR — THE MISCONCEPTION

The scenario

Wanjiku has shipped Mifugo Market. It is live at a public URL, customers are placing orders, and she has passed the capstone rubric. She has learned how to validate problems, write specifications, prompt AI, review diffs, build frontends and backends, add authentication, write tests, run security checklists, and deploy.

But now she is on her own. A new idea comes to her: a delivery scheduling tool for the same Ruiru shop owners. She does not know where to start, how to keep her dependencies updated, what to do when a build fails six months from now, or when to hire a developer instead of continuing solo.

She has the skills, but she does not yet have the habits of independent practice.

The misconception

01 Opening problem — "the book ends here" (continued)

MISCONCEPTION	WHY IT IS DANGEROUS	THE CORRECTION
"The book ends here."	Finishing the capstone is not the end of learning; it is the beginning of independent practice. Without a plan for maintenance, updates, and troubleshooting, the shipped application will decay.	The book launches independent practice; it does not conclude it.
"I will remember how everything works."	Six months from now, you will have forgotten why you made certain decisions, where secrets live, or how to roll back. Documentation and checklists exist because memory fades.	Document as you go; use checklists; keep a context file.

01 Opening problem — "the book ends here" (continued)

(continued)

MISCONCEPTION	WHY IT IS DANGEROUS	THE CORRECTION
"Every new idea should be shipped."	Shipping every idea without validation leads to a graveyard of unused applications and wasted time.	Validate each new problem before building, just as you did in Chapter 3.

02 Why this topic matters in real software projects

Software is never truly finished. Applications need dependencies updated, domains renewed, monitoring watched, and backups verified. The reader who can build an application but cannot maintain it will eventually lose it.

Constitution §10 requires practical learning requirements: checklists, templates, prompt libraries, and troubleshooting references. Constitution §13's durability criterion states that the companion website keeps tool-version guidance current for at least 24 months after publication, because tool-specific instructions date quickly.

This chapter is your bridge to independent practice: a prompt library, inspection checklists, a troubleshooting tree, a maintenance plan, and a framework for your next project — including when to hire a developer.

03 Learning objectives

WHAT YOU WILL BE ABLE TO DO

By the end of this chapter, the reader can:

03 Learning objectives (continued)

- Use the prompt library and inspection checklists for future projects.
- Troubleshoot common failures (build fails, deploy fails, tests red, auth broken).
- Navigate the companion website for current tool versions.
- Plan their next project independently (problem statement, stack choice, timeline, checklists).

03 Learning objectives (continued)

- Identify when to hire a developer versus continue solo.

04 Prerequisites and what to skip

BEFORE YOU START

Prerequisites

Chapters 1–16. You should have shipped a capstone application and understand the full lifecycle.

What you can skip

No new code in this chapter. You will write plans, checklists, and decision frameworks. Tool-specific instructions are flagged "verify current — see companion website" because they date quickly.

05 Plain-English explanation

Independent practice means owning the whole lifecycle

Independent practice means you can start a new idea, validate it, build it, deploy it, operate it, and decide whether to continue — without this book guiding every step. It means knowing where to find current information when tools change, and knowing when you are out of your depth.

A prompt library saves time and prevents slop

A **prompt library** is a collection of reusable, structured prompts (P1–P10 format from Chapter 7) for common tasks: adding validation, creating an endpoint, writing tests, deploying, reviewing a diff. Instead of starting from a blank chat each time, you adapt a proven prompt to your new project.

Inspection checklists catch problems before users do

An **inspection checklist** is a pass/fail list you run before accepting AI-generated code or before deploying. The capstone rubric (Chapter 16), the security checklist (Chapter 13), and the frontend checklist (Chapter 9) are examples. Keep them and reuse them.

A troubleshooting tree turns panic into process

When something breaks, a **troubleshooting tree** gives you a sequence of evidence-gathering steps: read the error message, check the logs, reproduce the issue, form a hypothesis, test a minimal change, fix or escalate. It prevents the "ask AI to regenerate everything" failure mode.

A maintenance window keeps the application alive

Schedule regular time to: check monitoring, renew the domain, review hosting costs, update dependencies, and verify backups. A neglected application becomes a security risk.

When to hire a developer

05 Plain-English explanation (continued)

You do not have to do everything yourself. Hire a developer when: the application needs features beyond your skill (complex integrations, real-time features), when security or compliance demands specialist review, or when your time is worth more than the developer's fee. Knowing when to hire is part of independent practice.

06 Mental model — and where it breaks down

Analogy: owning a shop versus building one

Building Mifugo Market was like constructing a shop: you chose the location, built the shelves, installed the lock, and put up the sign. Independent practice is like running the shop: you restock, open and close daily, handle customer complaints, fix the lock when it sticks, and decide whether to expand.

Where the analogy breaks down

Unlike a physical shop, software can break at any hour without a customer present, security threats arrive from anywhere in the world automatically, and dependencies can become vulnerable overnight. Independent practice requires scheduled vigilance, not just daily presence.

07 Technical explanation with correct terminology

Each term is defined at first use below and listed in the chapter glossary (Section 18).

07 Technical explanation with correct terminology

(continued)

- **Independent practice** — building, deploying, and operating software on your own after completing the book.
- **Prompt library** — a collection of reusable, structured prompts (P1–P10) for common tasks.
- **Inspection checklist** — a pass/fail list run before accepting AI-generated code or deploying.
- **Troubleshooting tree** — a sequence of evidence-gathering steps for diagnosing failures.

07 Technical explanation with correct terminology

(continued)

- **Maintenance window** — scheduled time to update dependencies, check monitoring, renew domains, and verify backups.
- **When-to-hire decision** — a judgment about when a project needs a professional developer.
- **Companion website** — the book's website with current tool versions, pricing, and "verify current" notes.
- **Dependency update** — keeping libraries and frameworks current to avoid known vulnerabilities.

07 Technical explanation with correct terminology

(continued)

- **Rollback** — returning the application to a previous version after a bad deploy.
- **Next project plan** — a documented plan for your next application: problem statement, stack, timeline, checklists.

07 Technical explanation with correct terminology

(continued)

[security] — Dependency update, Rollback (security and reliability practices)

08 Concrete example — Wanjiku's next project

IN PRACTICE

Wanjiku's next idea: a delivery scheduling tool for Ruiru shop owners. She applies independent practice:

1. **Validate first** (Chapter 3): Talk to five shop owners about delivery problems before writing code.
2. **Prompt library** (Chapter 7): Reuse her structured prompts for validation, backend, frontend, and tests.
3. **Inspection checklists** (Chapters 8, 13, 16): Run the diff-review, security, and capstone checklists.
4. **Troubleshooting tree** (this chapter): When the build fails, follow the tree instead of panicking.
5. **Companion website** (this chapter): Check current Render pricing and Kenyan payment options before choosing a stack.
6. **Next project plan** (this chapter): Write problem statement, stack choice, timeline, and checklists before starting.
7. **When-to-hire decision**: If she needs real-time delivery tracking (GPS, maps API), she will consider hiring a developer.

09 Step-by-step demonstration — troubleshooting a failed deploy

STEP BY STEP

When your deploy fails, follow this tree:

Step 1: Read the error message.

09 Step-by-step demonstration — troubleshooting a failed deploy (continued)

- Build log shows "command not found"? → Missing dependency or wrong build command.
- Deploy log shows "missing environment variable"? → Set the variable in the platform dashboard.

09 Step-by-step demonstration — troubleshooting a failed deploy (continued)

- Application starts but returns 500 errors? → Check application logs for the root cause.

09 Step-by-step demonstration — troubleshooting a failed deploy (continued)

Step 2: Reproduce locally.

09 Step-by-step demonstration — troubleshooting a failed deploy (continued)

- Run the application locally with the same environment variables. Does it fail the same way?
- If yes, the problem is in your code or configuration, not the platform.

09 Step-by-step demonstration — troubleshooting a failed deploy (continued)

Step 3: Form a hypothesis.

09 Step-by-step demonstration — troubleshooting a failed deploy (continued)

- "The build fails because `npm install` cannot find a package listed in `package.json`."
- "The deploy fails because `DATABASE_URL` is not set in Render's dashboard."

09 Step-by-step demonstration — troubleshooting a failed deploy (continued)

Step 4: Test a minimal change.

- Add the missing package; set the missing variable; redeploy.
- Make one change at a time; do not rewrite everything.

09 Step-by-step demonstration — troubleshooting a failed deploy (continued)

Step 5: Fix or escalate.

09 Step-by-step demonstration — troubleshooting a failed deploy (continued)

- If the minimal change fixes it, commit and redeploy.
- If you cannot identify the cause after two attempts, check the platform's status page, documentation, or companion website. Consider hiring a developer if it is security-critical.

[verify current] — Platform-specific error messages, dashboard layouts, and status page URLs change. Check the companion website and your platform's current documentation. [tool_comparison_framework.md, T]

Diagram — Troubleshooting decision tree

Alt text: A decision tree for diagnosing a failed deploy. Start: "Deploy failed." Branch 1: "Build log shows an error?" Yes → "Read the error message; fix dependency or build command; redeploy." No → Branch 2: "Application starts but errors in production?" Yes → "Check application logs; reproduce locally; fix code; redeploy." No → Branch 3: "Feature works locally but not live?" Yes → "Check environment variables in platform dashboard; set missing variables; redeploy." No → "Check platform status page; if platform is healthy and issue persists after two fix attempts, escalate or hire a developer."

09 Step-by-step demonstration — troubleshooting a failed deploy (continued)

```text
TROUBLESHOOTING TREE — Failed Deploy

START: Deploy failed
       |
```

09 Step-by-step demonstration — troubleshooting a failed deploy (continued) (continued)

```text
      v
[Build log shows an error?]
      | Yes
      v
```

09 Step-by-step demonstration — troubleshooting a failed deploy (continued) (continued)

```text
[Read error message → fix dependency/build command → redeploy]
      |
     No
      |
```

09 Step-by-step demonstration — troubleshooting a failed deploy (continued) (continued)

```text
      v
[Application starts but returns errors?]
      | Yes
      v
```

09 Step-by-step demonstration — troubleshooting a failed deploy (continued) (continued)

```text
[Check application logs → reproduce locally → fix code →
redeploy]
      |
     No
      |
```

09 Step-by-step demonstration — troubleshooting a failed deploy (continued) (continued)

```text
      v
[Feature works locally but not live?]
      | Yes
      v
```

09 Step-by-step demonstration — troubleshooting a failed deploy (continued) (continued)

```text
[Check platform dashboard env vars → set missing variables →
redeploy]
      |
     No
```

09 Step-by-step demonstration — troubleshooting a failed deploy (continued) (continued)

```text
      |
      v
[Platform status page healthy?]
      | No
```

09 Step-by-step demonstration — troubleshooting a failed deploy (continued) (continued)

```text
      v
[Wait for platform recovery; redeploy when healthy]
      |
```

09 Step-by-step demonstration — troubleshooting a failed deploy (continued)

```text
     Yes
      |
      v
[After two failed fix attempts → escalate to platform support or
hire a developer]
```

10 AI collaboration example (prompt format P1–P10)

AI COLLABORATION — PROMPT FORMAT P1–P10

PROMPT FORMAT **A reusable prompt you can copy**

P1 **P1 · PROBLEM IT SOLVES**

Get AI help for a new project without repeating past mistakes or losing control.

P2 **P2 · WHEN TO USE IT**

Chapter 17, when starting your next independent project.

P3 **P3 · REQUIRED INPUTS**

Your problem statement, your chosen stack, your prompt library, and your inspection checklists.

P4 **P4 · THE PROMPT**

TEXT

```
Role: You are a careful software builder helping with a new
project.
Context: New project description: [PROJECT_DESCRIPTION].
Stack: [STACK_CHOICE]. Problem statement:
[PROBLEM_STATEMENT].
Task: [TASK_DESCRIPTION for this milestone].
Constraints:
- Review every diff before accepting (Chapter 8
discipline).
- No hardcoded secrets or API keys.
- Input validated on the server (Chapter 13).
- Authorization enforced on user-data routes (Chapter 11).
- Stay within the acceptance criteria — no scope creep.
- Return unified diffs with file paths.
- Reference my prompt library and inspection checklists.
```

P5 **P5 · WHAT A GOOD RESPONSE CONTAINS**

Unified diff limited to the task; constraints honored; no secrets; no scope creep; references to your checklists.

10 AI collaboration example (prompt format P1–P10) (continued)

P6 **P6 · WHAT THE READER MUST INSPECT**
Confirm the diff matches the task; confirm no secrets; confirm validation and authorization are present; run tests after accepting.

P7 **P7 · WEAK OR INCOMPLETE RESPONSE**
Suggests features outside the scope; hardcodes a key; skips authorization; suggests whole-app rewrite.

P8 **P8 · FOLLOW-UP WHEN INADEQUATE**
"Limit the diff to [FILE] only," "Add the ownership check," "Remove the hardcoded key," "Stay within the acceptance criteria."

P9 **P9 · SAFETY/SECURITY CONSIDERATIONS**
Authentication and authorization are security-critical paths requiring mandatory human review (D-04). Review every line of auth-related changes.

P10 **P10 · VERIFICATION STEP**
After accepting, run the test suite; confirm the milestone behavior works; confirm no regression; commit with a meaningful message.

11 Human decisions and responsibilities (non-delegable judgment)

● YOUR DECISION — NOT THE AI'S

11 Human decisions and responsibilities (non-delegable judgment) (continued)

- **What to build next.** You choose the problem; the AI cannot validate demand for you.
- **Which tools to use.** You choose based on your needs, budget, and regional access; the AI can suggest but not decide. Check the companion website for current options.
- **When to hire a developer.** You judge when a project is beyond your skill or when security/compliance needs specialist review.

11 Human decisions and responsibilities (non-delegable judgment) (continued)

- **How to maintain what you ship.** You set the maintenance window, renew domains, and review costs.
- **Whether to continue or stop.** You decide if an application is worth sustaining.

11 Human decisions and responsibilities (non-delegable judgment) (continued)

[Human decision-point box] Before starting your next project, answer: Have I validated the problem? Do I have a prompt library and checklists ready? Do I know where to find current tool versions? Have I decided when I will hire a developer? If any answer is no, prepare first.

12 Inspect and verify AI output (ongoing practice checklist)

QUALITY CHECK

For every future project:

- ✓ [ ] Prompt library prompts are structured (P1–P10) and reviewed before use.
- ✓ [ ] Inspection checklists (diff, security, frontend, capstone) are run before accepting code or deploying.
- ✓ [ ] Troubleshooting tree is followed when something breaks; no whole-app regeneration.
- ✓ [ ] Companion website is checked for current tool versions before relying on any platform-specific step.
- ✓ [ ] Maintenance window is scheduled; dependencies are updated; domains are renewed.

13 Failure modes and recovery

WHEN IT GOES WRONG

13 Failure modes and recovery (continued)

FAILURE	SYMPTOM	LIKELY CAUSE	RECOVERY
Dependencies outdated	Security warning; build fails	No update habit	Schedule monthly dependency review; update and test
No backup	Data lost after failure	No backup routine	Set up automated database backups; test restore
No monitoring	You learn of outage from users	No monitoring setup	Set up error tracking; schedule weekly uptime check
Shipping every idea	Multiple unfinished applications	No validation habit	Validate each problem before building (Chapter 3)
Not knowing when to hire	Stuck on complex feature for weeks	Pride or budget concerns	Define when-to-hire criteria; budget for developer help

13 Failure modes and recovery (continued)

All scenarios in this section are hypothetical and labeled as such.

14 Professional standard — quality threshold for independent practice

● PRO STANDARD

You are practicing independently when:

14 Professional standard — quality threshold for independent practice (continued)

- You validate problems before building (Chapter 3).
- You use structured prompts and inspection checklists (Chapters 7, 8, 13).
- You troubleshoot using evidence, not guesswork (this chapter).
- You maintain what you ship: updates, backups, monitoring, cost review (this chapter).
- You know when to hire a developer (this chapter).

14 Professional standard — quality threshold for independent practice (continued)

- You check the companion website for current tool versions before following platform-specific steps (T).

15 Hands-on exercise (L3 — Application)

BUILD IT

Task: Build your personal prompt library and inspection checklists, and complete a troubleshooting tree exercise.

Constraints:

- No new code required.
- Use your capstone project as the reference application.
- Document your prompts and checklists in a file you can reuse.

Deliverable: A prompt library file (3–5 structured prompts), an inspection checklist, and a completed troubleshooting tree exercise.

15 Hands-on exercise (L3 — Application) (continued)

Success criteria:

- Prompt library includes at least 3 P1–P10 prompts for common tasks.
- Inspection checklist covers diff review, security, and deploy readiness.
- Troubleshooting tree exercise documents one failure scenario and its resolution steps.

Assessment level: L3 (Application — performs a procedure with guidance)

Time estimate: 60–75 minutes.

16 Build challenge (L4 — Analysis) with acceptance criteria

● PUSH FURTHER

Task: Create your next project plan and identify when you would hire a developer.

Constraints:

- Use a real idea you are considering, or the delivery scheduling tool example.
- Base the plan on validation evidence, not assumptions (label assumptions).
- Do not start building until the plan is complete.

Acceptance criteria (Given–When–Then):

16 Build challenge (L4 — Analysis) with acceptance criteria (continued)

- **Given** a new project idea, **When** you complete the next project plan, **Then** it includes a problem statement, stack choice, timeline, and checklists.
- **Given** the plan, **When** you identify features beyond your skill, **Then** you state when you would hire a developer and why.
- **Given** the companion website, **When** you check current tool versions, **Then** you record the URLs and dates you checked (platform-specific steps date quickly).
- **Given** your capstone application, **When** you review its maintenance needs, **Then** you have a maintenance window schedule (updates, backups, monitoring, cost review).

Deliverable: A next project plan document plus a when-to-hire note.

16 Build challenge (L4 — Analysis) with acceptance criteria (continued)

Assessment level: L4 (Analysis — investigates evidence, diagnoses gaps, plans a project)

Time estimate: 45–60 minutes.

17 Key takeaways

KEY TAKEAWAYS

- The book ends; independent practice begins. You now own the full lifecycle.
- Reuse your prompt library and inspection checklists for every project.
- Troubleshoot with evidence: read the error, reproduce locally, form a hypothesis, test minimally, fix or escalate.
- Maintain what you ship: updates, backups, monitoring, domains, costs.
- Know when to hire a developer — it is a sign of good judgment, not failure.
- Check the companion website for current tool versions; platform-specific steps date quickly.

18 Glossary of new terms

TERMS INTRODUCED IN THIS CHAPTER

18 Glossary of new terms (continued)

TERM	PLAIN-LANGUAGE DEFINITION	EXAMPLE	WHY IT MATTERS	TAG
Independent practice	Building, deploying, and operating software on your own after the book.	Starting your next project without chapter-by-chapter guidance	The book's goal: you can continue alone.	
Prompt library	A collection of reusable, structured prompts (P1–P10) for common tasks.	A prompt for adding validation, reused across projects	Saves time; prevents starting from a blank chat.	

18 Glossary of new terms (continued) (continued)

TERM	PLAIN-LANGUAGE DEFINITION	EXAMPLE	WHY IT MATTERS	TAG
Inspection checklist	A pass/fail list run before accepting AI-generated code or deploying.	The capstone rubric, security checklist, frontend checklist	Catches problems before users see them.	
Troubleshooting tree	A sequence of evidence-gathering steps for diagnosing failures.	Build failed → read error → reproduce → fix → redeploy	Turns panic into a repeatable process.	

18 Glossary of new terms (continued) (continued)

TERM	PLAIN-LANGUAGE DEFINITION	EXAMPLE	WHY IT MATTERS	TAG
Maintenance window	Scheduled time to update dependencies, check monitoring, renew domains.	First Sunday of each month	Keeps the application alive and secure.	
When-to-hire decision	A judgment about when a project needs a professional developer.	Hiring for GPS delivery tracking beyond your skill	Prevents getting stuck; uses budget wisely.	

18 Glossary of new terms (continued) (continued)

TERM	PLAIN-LANGUAGE DEFINITION	EXAMPLE	WHY IT MATTERS	TAG
Companion website	The book's website with current tool versions, pricing, and notes.	tool-versions page checked before deploying	Platform-specific instructions date quickly.	

18 Glossary of new terms (continued)

TERM	PLAIN-LANGUAGE DEFINITION	EXAMPLE	WHY IT MATTERS	TAG
Dependency update	Keeping libraries and frameworks current to avoid vulnerabilities.	Updating npm packages monthly	Reduces security risk from outdated components.	
Rollback	Returning the application to a previous version after a bad deploy.	`git revert` + redeploy previous version	Recovers quickly from a broken release.	
Next project plan	A documented plan for your next application.	Problem statement, stack, timeline, checklists	Prevents repeating past mistakes.	

19 Knowledge-check questions

KNOWLEDGE CHECK

Attempt these before looking at the answers below.

Q1. What is a prompt library, and why is it useful?

Q2. Name three steps in the troubleshooting tree for a failed deploy.

Q3. What is a maintenance window, and what does it include?

Q4. Name one sign that it is time to hire a developer.

Q5. Why should you check the companion website before following platform-specific steps?

[PAGE BREAK — ANSWER KEY]

Answers

A1. A prompt library is a collection of reusable, structured prompts (P1–P10 format). It saves time and prevents starting each AI interaction from scratch.

19 Knowledge-check questions (continued)

A2. Any three of: read the error message; reproduce the issue locally; form a hypothesis; test a minimal change; fix or escalate; check the platform status page.

A3. Scheduled time to update dependencies, check monitoring, renew domains, and verify backups. It keeps the application alive and secure after launch.

A4. Any one of: features beyond your skill (complex integrations, real-time features), security or compliance needs specialist review, or your time is worth more than the developer's fee.

A5. Platform-specific instructions (pricing, dashboard layout, deploy steps) date quickly. The companion website maintains current versions for 24 months after publication.

20 Links to earlier and later chapters

WHERE THIS SITS IN THE BOOK

Earlier chapters (all are prerequisites):

- **Chapters 1–16:** Every chapter feeds into independent practice — you now have the full lifecycle.
- **Chapter 7:** Prompting format (P1–P10) used in your prompt library.
- **Chapter 8:** Diff review used in your inspection checklists.
- **Chapter 13:** Security checklist used before deploying.
- **Chapter 16:** Capstone rubric used to assess your shipped application.

Later chapters: None — this is the final chapter. It launches your independent practice.

Companion website: Current tool versions, pricing, Kenyan payment accessibility, and updated deploy steps — flagged "verify current — see companion website" because platform-specific instructions date quickly. [T]

20 Links to earlier and later chapters (continued)

Evidence notes: Claims trace to Constitution §10 (practical learning requirements), §13 (durability criterion), tool_comparison_framework.md (tool selection and regional notes), pre_publication_checklist.md (verification gate). All scenarios are hypothetical.

Time estimate: Approximately 120 minutes total (60 minutes reading, 60 minutes hands-on practice).

APPENDIX A

Glossary of Core Terms

Every specialist term is defined here in plain English, with the US spelling noted where tools use it.

Project: STOP CREATING SLOP: Vibe Code Like a Pro **Purpose:** Terms defined at first use in the book are collected here for the reader and for consistency across chapters. **Source of definitions:** Evidence register and research report (see `evidence_register.md`, `research_report.md`).

APPENDIX B

Evidence Register

Every bracketed citation in this book resolves to an entry below. Classes indicate source strength: A primary official documentation, B recognised standard, C reputable secondary, D flagged gap.

Project: STOP CREATING SLOP: Vibe Code Like a Pro **Status:** Research foundation — Stage 1 (per Book Constitution) **Access date:** 2026-10-24 **Purpose:** Every material claim in the book must trace to an entry here. Entries are classified by evidence strength and flagged for time-sensitivity.

APPENDIX C

Standards and Security References

The external standards this book teaches against, with the controls mapped to the chapters that apply them.

Project: STOP CREATING SLOP: Vibe Code Like a Pro **Purpose:** The authoritative list of standards, regulations, and security practices the book teaches. Each entry includes the source, evidence strength, and where it appears in the book. **Companion:** evidence_register.md, research_report.md

ABOUT THE AUTHOR

Antony Kariuki

Antony Kariuki writes for the people who are now expected to build software without being trained as software engineers: founders, students, operators, and professionals who have real problems and no formal computer-science background.

This book exists because AI made code cheap and judgement scarce. Its argument is simple: the tool has changed, the responsibility has not. You still decide what to build, what "working" means, and what you are willing to put in front of another person.

The running example, Mifugo Market, is a fictional fresh-produce ordering platform for small shops and households in Nairobi. Any resemblance to a real business is coincidental.

COLOPHON

Interior designed with the SPARK & FLOW v2 system: 6 × 9 inch trim, Deep Charcoal (#121216) and Warm Paper (#FBF9F5) grounds, Sunrise Gold (#F2A104), Electric Cyan (#00D4FF), Creative Magenta (#D946EF), Vitality Green (#10B981) and Ember Orange (#F59E0B) accents. Body text Plus Jakarta Sans 10.5/16 pt; display Outfit; code JetBrains Mono 8.1/11.6 pt.

Typeset from Markdown sources through a reproducible HTML/CSS pipeline and rendered with Chromium. Every figure, table, code block, prompt, and exercise in this edition is generated from the same committed source files listed in the release manifest.

First edition, version 1.0 · built 2026–10–24 · Chromium 153 / Playwright (HTML to PDF)