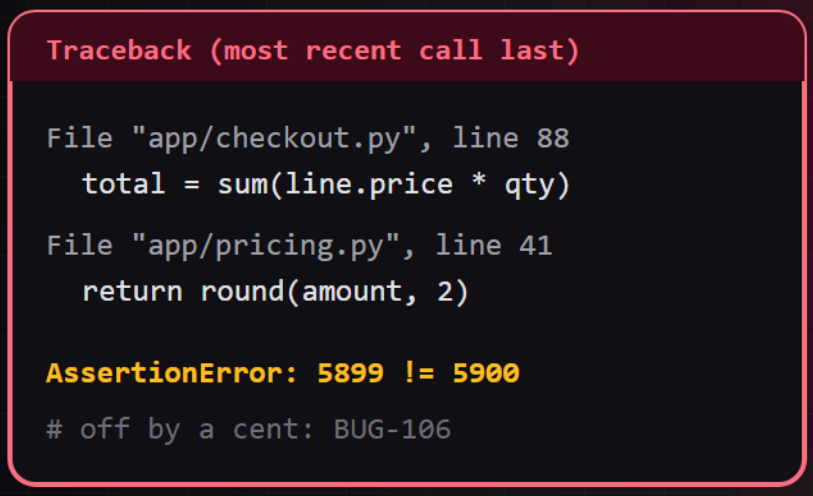
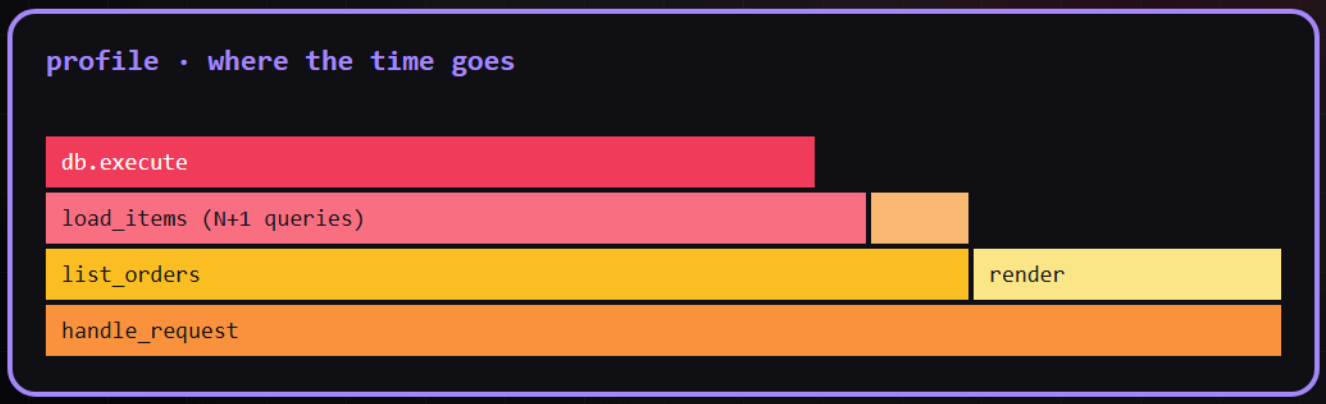






REPRODUCE · OBSERVE · HYPOTHESIZE · EXPERIMENT · VERIFY

A SYSTEMATIC, PRACTICAL GUIDE

Debugging Techniques That Actually Work

Find and fix bugs faster — from local code to production

Debuggers, git bisect, profiling, races, flaky tests, memory leaks, observability, and incident practice

PYTHON · TYPESCRIPT · SQL · ANY LANGUAGE

THENEXBOOKS

Debugging Techniques That Actually Work

A Practical, Systematic Guide to Finding and Fixing Bugs Faster, from Local Code to Production

First Edition

THENEXBOOKS

Debugging Techniques That Actually Work
A Practical, Systematic Guide to Finding and Fixing Bugs Faster, from Local Code to Production

First Edition, 2026

Copyright © 2026 thenexbooks (thenexbooks.gumroad.com). All rights reserved.

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 publisher, except for brief quotations in reviews and certain other non-commercial uses permitted by copyright law. Code examples may be used freely in your own projects.

The information in this book is provided "as is" without warranty of any kind. While every precaution has been taken in its preparation, the publisher assumes no responsibility for errors or omissions, or for damages resulting from the use of the information contained herein. Tools, flags, and defaults change between versions; consult the documentation for the versions you use. Timings, memory figures, and counts in the bug stories are illustrative.

Python is a trademark of the Python Software Foundation. JavaScript is a trademark of Oracle and/or its affiliates. TypeScript, Visual Studio Code, and Windows are trademarks of Microsoft Corporation. Node.js, Chrome, PostgreSQL, Redis, Stripe, Git, and other product names are trademarks of their respective owners. Their use in this book is for identification only and does not imply endorsement.

Marigold Pantry, the running example in this book, is fictional, as are all people, customers, and bugs in its stories. Any resemblance to real companies or products is coincidental.

Preface

Every developer debugs. Few are ever taught how. We learn languages, frameworks, and design patterns in courses and books, and then spend a large and very real part of our working lives staring at behavior that makes no sense — learning to debug by trial, error, and the occasional heroic late night.

That is a shame, because debugging is not a mysterious talent. It is a skill with a method, a toolkit, and a set of recurring patterns. The developers who seem to find bugs "by instinct" are usually doing something systematic very quickly: reproducing the problem reliably, reading the evidence carefully, forming a hypothesis they can test, cutting the search space in half, and verifying the fix instead of hoping. Once you can see the method, you can practice it — and get dramatically better.

This book teaches that method and the tools that support it, from the debugger in your editor to `git bisect`, profilers, heap snapshots, packet captures, query plans, distributed traces, and production incident practice. It walks through the bug families that cause most real-world pain — off-by-one errors, stale state, time zones, floating-point money, environment drift, races, flaky tests, leaks, and slow queries — and shows how to recognize each one quickly. We explain *why* each technique works, not just which button to press.

Who This Book Is For

- **Junior developers** who want a reliable process instead of random changes and hope.
- **Experienced developers** who want to get faster at hard bugs: concurrency, memory, performance, and distributed systems.
- **QA engineers** who turn vague reports into precise, reproducible bug reports.
- **SREs and on-call engineers** who debug live systems under pressure and need to do it safely.
- **Tech leads** who want their team to learn from bugs instead of repeating them.

What You Will Learn

- A systematic debugging method you can apply to any bug in any language.
- How to reproduce bugs reliably, read stack traces and logs, and shrink a problem to a minimal example.
- Debuggers, breakpoints, and logging done well; binary search with `git bisect`; and tests as debugging tools.
- How to recognize and fix logic, state, data, time-zone, configuration, and dependency bugs.
- How to tame races, deadlocks, async pitfalls, flaky tests, memory leaks, and performance problems.
- Debugging across networks and services, in the browser, and in the database.
- Observability, incident response, and safe production debugging.

- Using AI assistants effectively without trusting them blindly.
- Writing debuggable code, running blameless postmortems, and building a team debugging playbook.

How This Book Is Organized

Part I — The Debugging Mindset (Chapters 1–5) introduces the method: reproduce, read the evidence, hypothesize, experiment, and shrink the problem. **Part II — Core Tools and Habits** (Chapters 6–10) covers debuggers, logging, bisection, tests, and the human side of debugging. **Part III — Common Bug Families** (Chapters 11–15) teaches you to recognize the bugs you will meet most often. **Part IV — Hard Bugs** (Chapters 16–20) tackles concurrency, nondeterminism, memory, performance, and distributed systems. **Part V — Debugging in Production** (Chapters 21–25) covers observability, incidents, the browser, the database, and AI assistants. **Part VI — Prevention and Learning** (Chapters 26–28) turns debugging experience into better code and better teams, ending with a full case study. The appendices provide a cheat sheet, a tool reference, checklists, and a glossary.

Each chapter includes practice exercises and ends with a **Key Takeaways** box for quick review.

The Running Example: Marigold Pantry

Throughout the book we debug **Marigold Pantry**, a fictional online grocery-box subscription service. Its web app is written in TypeScript and React, its API in Python with FastAPI, and its background worker in Node.js; it stores data in PostgreSQL, uses Redis for caching and queues, and takes payments through Stripe. Rosa works on the API, Kenji on the web app, Amara leads on-call, Lucas is the junior developer who asks the questions you might ask, and Dee runs customer support and files the bug reports.

Marigold Pantry has a catalog of ten memorable bugs — `BUG-101` to `BUG-110` — from Auckland customers seeing the wrong delivery day to rare double charges, a slowly leaking worker, a flaky checkout test, and totals that are off by a single cent. You will meet them one by one in the chapters where their bug family is explained, and Chapter 28 follows the worst of them from first report to lasting fix.

Versions and Conventions

Examples use recent versions of Python, Node.js, TypeScript, PostgreSQL, and common tools. The techniques apply to any language; where a tool differs meaningfully in Java, Go, or other ecosystems, a note says so. Terminal sessions show commands and trimmed output; numbers in bug stories are illustrative.

> **TIP**
>
> A recommended practice or a technique that saves you time.

> **NOTE**
>
> Background information, version details, or a useful aside.

> **WARNING**
>
> A common trap that wastes hours, hides the real cause, or makes production worse.

File captions such as *marigold-api/app/orders.py* tell you which service a snippet belongs to.

Contents

Preface

PART I — THE DEBUGGING MINDSET

1 Why Debugging Is a Skill: A Systematic Method for Finding Bugs

2 Reproduce First: Turning Vague Reports into Reliable Reproductions

3 Reading Errors, Stack Traces, and Logs Carefully

4 Hypotheses and Experiments: Debugging as Applied Science

5 Shrinking the Problem: Isolation, Minimal Examples, and Delta Debugging

PART II — CORE TOOLS AND HABITS

6 Debuggers in Depth: Breakpoints, Stepping, Watches, and Conditional Breakpoints

7 Print Debugging and Logging Done Right

8 Binary Search Everything: git bisect and Divide-and-Conquer

9 Tests as Debugging Tools: Failing Tests, Characterization Tests, and Property-Based Tests

10 The Human Side: Rubber Ducks, Breaks, Pairing, and Asking Good Questions

PART III — COMMON BUG FAMILIES

11 Logic Bugs: Off-by-One Errors, Boundaries, Null and Undefined

12 State Bugs: Mutation, Aliasing, Caching, and Stale Data

13 Data Bugs: Encoding, Floating Point, Money, Dates, and Time Zones

14 Environment and Configuration Bugs: "It Works on My Machine"

15 Dependency and Build Bugs: Versions, Lockfiles, and Toolchains

PART IV — HARD BUGS

16 Concurrency Bugs: Race Conditions, Deadlocks, and Async Pitfalls

17 Heisenbugs and Flaky Tests: Taming Nondeterminism

18 Memory Bugs: Leaks, Bloat, and Heap Analysis

19 Performance Bugs: Profiling CPU, I/O, and Hot Paths

20 Debugging Across Boundaries: HTTP, Networks, and Distributed Systems

PART V — DEBUGGING IN PRODUCTION

21 Observability for Debugging: Structured Logs, Metrics, and Traces

22 Production Incidents: Triage, Mitigation, and Gathering Evidence Safely

23 Frontend and Browser Debugging: DevTools, Network, and Rendering

24 Database and Query Debugging

25 Debugging with AI Assistants: Using Them Well and Verifying Their Answers

PART VI — PREVENTION AND LEARNING

26 Writing Debuggable Code: Assertions, Errors, and Design for Diagnosis

27 Postmortems and Bug Journals: Learning from Every Bug

28 Case Study: Hunting Down Marigold Pantry's Worst Bugs

APPENDICES

A Debugging Cheat Sheet

B Tool and Command Reference

C Debugging Checklists

D Glossary

PART I

The Debugging Mindset

Debugging as a learnable, systematic skill: reproducing bugs reliably, reading errors and logs carefully, testing hypotheses like a scientist, and shrinking problems until the cause has nowhere to hide.

1 · Why Debugging Is a Skill
2 · Reproduce First
3 · Reading Errors and Logs
4 · Hypotheses and Experiments
5 · Shrinking the Problem

CHAPTER 1 · PART I — THE DEBUGGING MINDSET

Why Debugging Is a Skill: A Systematic Method for Finding Bugs

Debugging is not a talent some developers are born with. It is a method you can learn, practice, and improve. This chapter introduces Marigold Pantry, the system we will debug together throughout the book, and the seven-step Debugging Loop that every later chapter builds on.

A Monday Morning Ticket

At 8:52 on a Monday, Dee, the support lead at Marigold Pantry, files a ticket. It is short, as the first report of a bug usually is: *"Customer says her account page shows her plan as $0.00. She's worried she hasn't been paying. Can someone look?"*

Lucas, a junior developer, picks it up. His own test customer shows the correct price. He stares at the pricing code, adds a `print` call, sees the right price again, and starts wondering about old browsers. By lunchtime he has tried five things and learned almost nothing.

Rosa, the backend developer who owns the `api` service, stops by his desk. She does not look at the code first. She asks three questions: "Which customer? What plan are they on? What exactly did they see?" Within fifteen minutes she has a reproduction, a hypothesis, a one-line experiment that confirms it, and a fix with a test. She is not smarter than Lucas, and she does not know the pricing code better. The difference is that Rosa follows a method and Lucas does not have one yet.

This book is about that method. We will return to Dee's ticket later in the chapter and follow Rosa step by step.

Debugging Is a Skill, Not a Gift

Most developers learn debugging by being thrown in the deep end. They pick up a private collection of tricks they cannot fully explain — some good, many superstitions.

The developers who seem to have a gift for debugging usually have three things that can be learned:

- **A process.** They do not jump from idea to idea. They know what step they are on and what the next step should be. When they are stuck, they know which step to go back to.
- **Good observation habits.** They read error messages completely, they look at actual data instead of assuming what it contains, and they notice small discrepancies that others skip over.
- **A toolkit they are fluent with.** Debuggers, logs, `git bisect`, profilers, and tracing are not exotic to them. They reach for the right tool without hesitation because they have practiced with it.

Experience helps, but experience without method produces pattern-matching that fails on unfamiliar bugs. Method gets you to the answer reliably, and this book pairs it with many realistic bugs to accelerate the experience.

Why guessing feels productive and is not

Lucas's morning is worth examining because almost everyone debugs that way sometimes. He was busy the whole time. Each change felt like progress. But look at what actually happened: he never reproduced the problem, so none of his experiments could possibly have told him anything. His local test customer was on a different plan than the real customer. Every "fix" he tried was tested against a situation where the bug did not occur, so every result looked like success or like nothing at all.

This style is sometimes called **shotgun debugging**: change things more or less at random and see whether the symptom goes away. It has three serious problems. First, it is slow, because the space of possible changes is enormous and most of them are irrelevant. Second, when it does appear to work, you often do not know why, so you cannot be sure the bug is really gone rather than just hidden. Third, it leaves debris behind — extra null checks, retries, and sleeps that were added "just in case" and that make the next bug harder to find.

The alternative is a sequence of small, deliberate experiments, each designed to answer one question.

A Vocabulary for Bugs

The word "bug" is doing a lot of work in everyday conversation. It can mean a line of code with a mistake in it, a wrong value sitting in memory, or a customer seeing the wrong thing on a screen. Separating those meanings makes debugging clearer, because they usually live in different places.

Term	Meaning	Marigold example
Defect	The mistake in the code or configuration	A wrong key in a price table
Infection	Wrong state produced when the defect runs	A price of 0 cents in memory
Propagation	Wrong state flowing through the program	0 cents serialized into JSON
Failure	The visible symptom someone notices	"$0.00" on the account page

The practical lesson is that **the failure is rarely where the defect is**. The customer saw the problem in the web frontend. The defect was in the Python API. Between them was a chain of correct code faithfully passing along a wrong value. Debugging, in this view, is the work of walking backward along that chain from the failure to the defect. Each step of the method in this chapter is a way of moving along the chain more efficiently.

You will also hear the word **root cause**. We use it carefully in this book. A bug often has several contributing causes: the defect itself, a missing test that would have caught it, a silent default that hid it, and a missing alert that let it reach customers. Fixing only the defect is necessary but frequently not

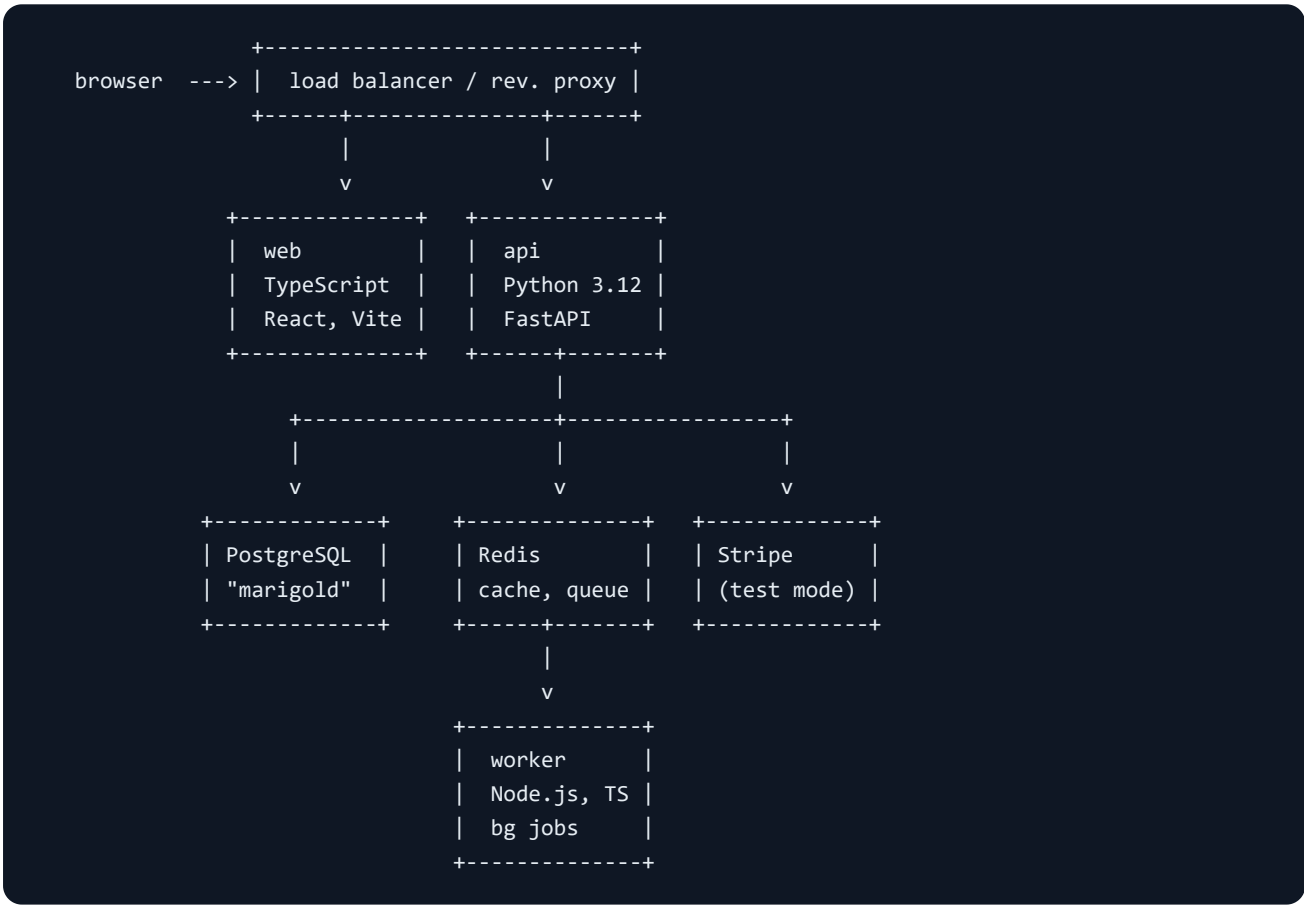
sufficient. Chapter 27 goes deeper into how teams analyze causes without blame.

> **NOTE**
>
> You may have heard that the word "bug" comes from a moth found in a relay of the Harvard Mark II computer in 1947. The logbook page with the taped-in moth is a well-known and genuine artifact, but engineers were already using "bug" to mean a technical fault well before then — the logbook entry itself jokes about the "first actual case of bug being found." It is a good story about the word, not its origin.

Meet Marigold Pantry

Throughout this book we debug one fictional system so that the examples connect into a single story. **Marigold Pantry** sells weekly grocery boxes by subscription in several regions. Customers pick a plan, choose a delivery slot, and receive a box every week unless they skip or pause. It is a small business with a modern, fairly typical architecture — which means it has every classic family of bugs.

The services



- `web` — the customer-facing single-page app, written in TypeScript with React and built with Vite. Repository: `marigold-web`.

- `api` — the HTTP API in Python 3.12 with FastAPI and SQLAlchemy. It owns business rules: pricing, subscriptions, orders, and payments. Repository: `marigold-api`.
- `worker` — a Node.js and TypeScript service that runs background jobs from a Redis-backed queue: building weekly orders, sending emails, syncing payment status. Repository: `marigold-worker`.
- **PostgreSQL** holds the `marigold` database; **Redis** serves as both a cache and the job queue; payments go through **Stripe in test mode** in every environment you will see in this book.

Everything runs as containers behind a reverse proxy and load balancer. Observability comes from structured JSON logs, metrics, and OpenTelemetry traces, which we cover properly in Chapter 21.

The data

The core tables are small and familiar:

```
customers      (id, email, region, timezone)
subscriptions  (id, customer_id, plan, status, next_delivery_date)
orders         (id, customer_id, total_cents, currency, status, created_at)
order_items    (id, order_id, product_id, quantity, unit_price_cents)
delivery_slots (id, region, starts_at timestamptz, ends_at)
payments       (id, order_id, stripe_payment_intent_id, amount_cents, status)
```

Three sample customers appear again and again. **1001 Hana Sato** (`hana@example.com`) lives in region `wellington` with time zone `Pacific/Auckland`. **1002 Omar Haddad** is in region `london` with `Europe/London`. **1003 Grace Liu** is in region `vancouver` with `America/Vancouver`. The plans are `classic` at 59.00, `family` at 89.00, and `veggie` at 54.00. Notice that these customers span time zones on both sides of UTC. That is deliberate, and it will matter a great deal.

The people

- **Rosa** — backend developer who owns `api`. Methodical, keeps notes, and has a habit of asking "what exactly did you see?"
- **Kenji** — frontend developer who owns `web`. Lives in browser DevTools and is suspicious of anything involving dates.
- **Amara** — SRE and on-call lead. Thinks in terms of blast radius, mitigation, and evidence. Runs the incident calls.
- **Lucas** — junior developer. He asks the questions many readers will be thinking, and he improves visibly over the book.
- **Dee** — support lead. She talks to customers and files the bug reports, and she has strong opinions about tickets that get closed as "cannot reproduce."

The Bug Catalog: A Map of the Book

Marigold Pantry has a backlog of bugs that we will investigate in depth in later chapters. Each has an ID that stays the same everywhere in the book. You do not need to understand them yet; this table is a map, so that when a chapter mentions `BUG-104` you know where it fits.

ID	Name	Symptom	Chapters
BUG-101	Wrong delivery day	Auckland customers see Tuesday slots labelled Monday	13, 23, 28
BUG-102	Double charge	Rare duplicate payments after a double-click on Pay	16, 20, 28
BUG-103	Worker memory climbs	Worker is OOM-killed about every two days	18, 28
BUG-104	Order history is slow	Seconds-long loads for long-time customers	19, 24
BUG-105	Flaky checkout test	A discount test fails occasionally in CI	9, 17
BUG-106	Off by a cent	Some totals differ from Stripe by 0.01	13, 28
BUG-107	Works on my machine	Password-reset emails fail only in staging	14
BUG-108	Missing last page	Admin order search drops the final partial page	11
BUG-109	Intermittent 502s	A small share of requests fail at the balancer	20, 22
BUG-110	The regression	Plan upgrades broke for paused subscriptions	8

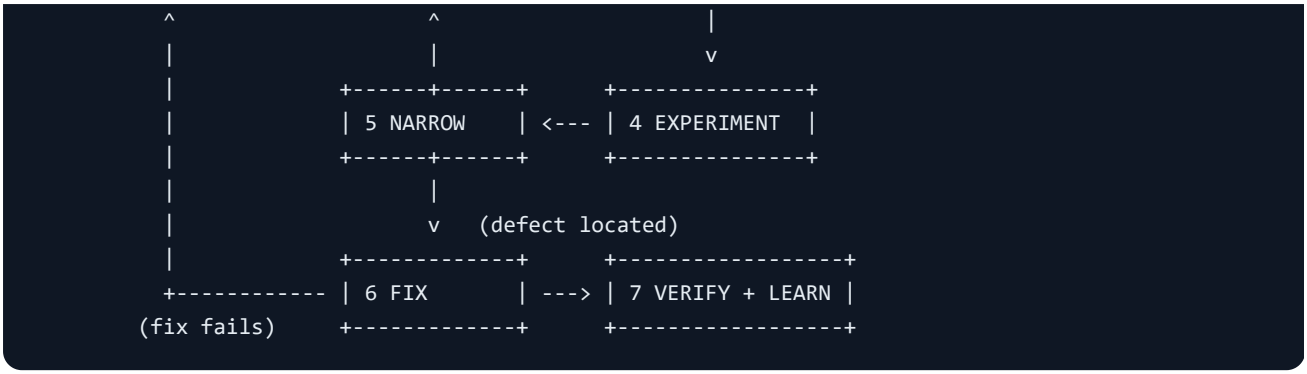
Look at the spread. There are data bugs (dates and money), concurrency bugs (double charges), resource bugs (memory), performance bugs (slow queries), nondeterminism (a flaky test), configuration bugs (staging-only failures), plain logic bugs (an off-by-one), network bugs (502s), and a regression. That variety is the point: a good method works across all of them, and the specialized techniques in Parts III through V plug into the same method. Chapter 28 brings the hardest of them together into a single case study.

> **TIP**
>
> When a story in this book mentions timings, memory sizes, or counts for a catalog bug, those numbers are illustrative. They are there to make the story concrete, not to describe a real system. When you debug your own system, the numbers you measure are the only ones that count.

The Debugging Loop

Here is the method at the center of this book. We call it the **Debugging Loop**, and it has seven steps. Every chapter that follows either deepens one of these steps or shows how to apply the whole loop to a particular kind of bug.





1. **Reproduce.** Make the failure happen on demand, as close to the original conditions as you need and as small and fast as you can. A bug you cannot reproduce is a bug you cannot confirm you have fixed. (Chapter 2.)
2. **Observe.** Collect and read the evidence carefully: the full error message, the stack trace, the logs, the actual data, and the difference between expected and actual behavior. Describe what is happening before you try to explain it. (Chapter 3.)
3. **Hypothesize.** Propose specific, testable explanations. A good hypothesis says where the defect is and predicts something you have not yet seen. (Chapter 4.)
4. **Experiment.** Design the cheapest experiment that could prove the hypothesis wrong, predict its result, run it, and compare. Change one thing at a time. (Chapter 4.)
5. **Narrow.** Use each result to shrink the search space: fewer lines, fewer commits, fewer inputs, fewer services. Loop back through Observe, Hypothesize, and Experiment until the defect is pinned down. (Chapters 5 and 8.)
6. **Fix.** Change the defect, not the symptom. Prefer the smallest correct change, and understand why it works before you commit it.
7. **Verify and Learn.** Confirm the original reproduction now passes, keep it as a regression test, check for the same defect elsewhere, and record what you learned so the next person (often future you) benefits. (Chapters 9 and 27.)

The loop is drawn as a loop for a reason. Real debugging goes around steps 2 through 5 many times. An experiment disproves your favorite hypothesis, so you go back to observing with new eyes. A fix fails verification, so you go back to reproduction and ask whether you were reproducing the real bug at all. The steps are not a waterfall; they are a way of always knowing where you are and what kind of move comes next.

Why this order

Each step exists because skipping it causes a specific, common failure.

Skip Reproduce and you cannot tell whether an experiment worked — Lucas's problem exactly. Skip Observe and you theorize from assumptions that the first line of the stack trace had already ruled out. Skip Hypothesize and your experiments have no question behind them; "I added logging everywhere" is not an experiment. Skip Experiment and you go from a plausible story straight to a fix, though many

bugs have several plausible stories and only one is true. Skip Narrow and you stare at the whole system hoping for insight. Skip Fix discipline and you patch the symptom with a swallowed exception or a retry. Skip Verify and Learn and the same bug returns in three months, with nobody remembering how it was solved.

The loop at different scales

The loop is not only for week-long investigations. For a typo that fails a unit test, you may go around it in thirty seconds: the test reproduces, the assertion message is the observation, the hypothesis is obvious, the fix is immediate, and rerunning the test is verification. Nothing about the method requires ceremony. What it requires is that you *could* say which step you are on. When a bug turns out to be harder than it looked, the loop is already there to scale up with you.

In production, the loop gains a step zero: **stabilize**. If customers are being charged twice right now, stopping the damage comes before understanding it. Amara's incident process in Chapter 22 handles mitigation first and then runs the Debugging Loop on evidence gathered safely. Mitigation and debugging are different activities, and mixing them up is a classic source of confusion during incidents.

> **WARNING**
>
> The most expensive shortcut in debugging is jumping from a plausible explanation directly to a fix without an experiment. It feels efficient. But if the explanation was wrong, the "fix" changes behavior for no reason, the bug remains, and now there is a second unexplained change in the code for the next person to puzzle over. Always ask: what result would prove this idea wrong, and have I checked it?

The Debugging Notebook

One habit supports every step of the loop: writing things down as you go. We call this the **debugging notebook**. It can be a text file, a section of the ticket, or a scratch document — the format matters much less than the habit. Rosa keeps a plain Markdown file per investigation. A minimal entry looks like this:

notes/2026-09-14-plan-price-zero.md

```
# Plan price shows $0.00 (ticket from Dee)

## Symptom
Account page shows plan price 0.00 for customer 1003. Expected 54.00.

## Reproduction
GET /customers/1003/subscription -> "price_cents": 0   (local + staging)
GET /customers/1002/subscription -> "price_cents": 5900 (classic, correct)

## Log
09:10  H1: web formats price wrong        -> API already returns 0. Rejected.
```

```
09:14  H2: price lookup misses plan name  -> plan = "veggie", table has
       "vegetarian". Confirmed in REPL.
09:20  Fix: key renamed + unknown plan raises. Test added.

## Open questions
- Why did 0 not fail loudly? (silent .get default)
- Any other lookups keyed by old plan names?
```

The notebook does several jobs at once. It keeps you honest about which hypotheses you have actually tested versus merely considered. It stops you from repeating an experiment you already ran an hour ago. It makes handing the bug to a colleague trivial, because the context is written down instead of trapped in your head. And after the bug is fixed, it becomes the raw material for a ticket comment, a commit message, or a postmortem. Chapter 27 turns the notebook into a longer-lived bug journal.

Walking the Loop: The $0.00 Plan

Let us return to Dee's ticket and follow Rosa through the loop in detail. The bug is small, which makes it a good first example: every step is visible without the noise of a truly hard problem.

Step 1: Reproduce

Rosa's first move is to find out exactly who saw what. Dee checks the ticket: the customer is Grace Liu, customer 1003, and she is on the `veggie` plan. Lucas's test customer was on `classic`. That single detail explains why Lucas could not see the bug. Rosa reproduces at the lowest level she can, calling the API directly against her local stack, which has the same seed data as staging:

```
$ curl -s localhost:8000/customers/1003/subscription | python -m json.tool
{
    "customer_id": 1003,
    "plan": "veggie",
    "status": "active",
    "price_cents": 0,
    "next_delivery_date": "2026-09-17"
}

$ curl -s localhost:8000/customers/1002/subscription | python -m json.tool
{
    "customer_id": 1002,
    "plan": "classic",
    "status": "active",
    "price_cents": 5900,
    "next_delivery_date": "2026-09-16"
}
```

Now she has a reproduction that takes under a second, does not require a browser, and shows both a failing case and a passing case. That contrast is valuable. Whatever differs between customer 1003 and customer 1002 is where the bug lives.

Step 2: Observe

The observation is precise: the API returns `price_cents: 0` for a `veggie` subscription and the correct value for a `classic` one. The web app is not involved — it simply displays what it receives. Rosa writes that down, because it eliminates an entire service from the investigation. She also notes what she does *not* see: there is no error in the API logs for the request. Something returned zero without complaint.

Step 3: Hypothesize

With the evidence in hand, Rosa lists candidate explanations:

- **H1:** the price is computed correctly but a discount or promotion reduces it to zero.
- **H2:** the price lookup does not recognize the plan name and falls back to zero.
- **H3:** the database row for this subscription has an unexpected plan value, such as trailing whitespace.

Each hypothesis makes a different prediction. If H1 is true, the undiscounted price should be 5400 somewhere along the way. If H2 is true, the lookup function itself returns 0 for the string `"veggie"`. If H3 is true, the plan value in the database will not equal `"veggie"` exactly, even though it prints that way.

Step 4: Experiment

The cheapest decisive experiment is to call the pricing function directly in a Python REPL. It takes seconds and it distinguishes H2 from the others immediately.

```
$ python
>>> from app.pricing import plan_price_cents, PLAN_PRICES_CENTS
>>> plan_price_cents("veggie")
0
>>> plan_price_cents("classic")
5900
>>> PLAN_PRICES_CENTS
{'classic': 5900, 'family': 8900, 'vegetarian': 5400}
```

Prediction confirmed: the function returns zero for `"veggie"`, and the table has no `"veggie"` key. Rosa also runs a quick query to rule out H3 — `SELECT plan, length(plan) FROM subscriptions WHERE customer_id = 1003;` returns `veggie` and `6`. The stored value is exactly what it appears to be.

Step 5: Narrow

The defect is now down to a single function:

marigold-api/app/pricing.py (before)

```
PLAN_PRICES_CENTS = {
    "classic": 5900,
    "family": 8900,
    "vegetarian": 5400,
```

```
}


def plan_price_cents(plan: str) -> int:
    return PLAN_PRICES_CENTS.get(plan, 0)
```

A quick `git log -S vegetarian` shows the history. Several weeks earlier, a database migration renamed the plan from `vegetarian` to `veggie` to match the marketing site. The migration updated the `subscriptions` table and the web app's labels, but nobody updated this dictionary. Because `.get(plan, 0)` quietly returns zero for an unknown key, nothing crashed. Existing `veggie` customers were still billed correctly, it turns out, because the billing job reads prices from a different, correctly migrated table — which is why nobody noticed until a customer looked at the account page. That second price source is itself worth a note in the notebook.

Step 6: Fix

A symptom fix would add `"veggie": 5400` to the dictionary and stop there. Rosa does add the key, but she also fixes the thing that let the bug hide: the silent default. An unknown plan is a programming error, and it should fail loudly.

marigold-api/app/pricing.py (after)

```
PLAN_PRICES_CENTS = {
    "classic": 5900,
    "family": 8900,
    "veggie": 5400,
}


class UnknownPlanError(ValueError):
    pass


def plan_price_cents(plan: str) -> int:
    try:
        return PLAN_PRICES_CENTS[plan]
    except KeyError:
        raise UnknownPlanError(f"no price configured for plan {plan!r}") from None
```

Had the code been written this way originally, the migration would have produced an error with the plan name in it on the first request, probably in a test run long before any customer saw it. Chapter 26 explores this idea at length: code that fails loudly and specifically is dramatically easier to debug than code that limps along with a plausible wrong value.

Step 7: Verify and Learn

Verification starts with the original reproduction. The `curl` for customer 1003 now returns 5400. Then Rosa turns the reproduction into permanent tests:

marigold-api/tests/test_pricing.py

```
import pytest

from app.pricing import PLAN_PRICES_CENTS, UnknownPlanError, plan_price_cents
from app.models import PLAN_NAMES  # the enum used by the subscriptions table


@pytest.mark.parametrize("plan", sorted(PLAN_NAMES))
def test_every_plan_has_a_price(plan):
    assert plan_price_cents(plan) > 0


def test_unknown_plan_raises():
    with pytest.raises(UnknownPlanError):
        plan_price_cents("vegetarian")


def test_price_table_has_no_stale_plans():
    assert set(PLAN_PRICES_CENTS) == set(PLAN_NAMES)
```

The first test would have failed the moment the plan was renamed. The third catches the reverse mistake — a stale price for a plan that no longer exists. Then comes the "learn" part: Rosa searches the codebase for other references to `vegetarian` and finds one more, in an email template in `marigold-worker`. She adds a line to the notebook, opens a small follow-up ticket, and writes a commit message that explains the *why*, not just the *what*. Finally, she tells Dee the customer was never undercharged — the important answer from the customer's point of view.

Lucas, watching, asks whether that was a lot of steps for a one-line bug. Rosa points out that it took fifteen minutes and that the steps are what found the second stale reference. "The bug was one line," she says. "The reason it reached a customer was three things."

Habits That Break the Loop

Knowing the steps is half the battle. The other half is noticing when you have slipped out of them. These are the habits we see most often in real teams, including experienced ones.

Fixing before understanding

The code looks suspicious, so you change it, and the symptom goes away. You have not established that the change fixed the defect rather than moving it. A classic example: a race condition "fixed" by adding a `sleep(0.5)`, which makes the race less likely on your machine and just as likely under production load. If you cannot explain in one or two sentences why a fix works, you are not done.

Trusting assumptions over evidence

"That function can't return null." "The cache is definitely invalidated on update." "Staging has the same config as production." Each of these is a hypothesis dressed up as a fact. The most productive debugging question is often *"How do I know that?"* When the answer is "I'm pretty sure," go and look.