DataShelf Hub

BACKEND ENGINEERING

Concurrency in Practice

Threads, async and lock-free patterns for services under real load

● FREE SAMPLE CHAPTER

Omar El Alaoui
Data & systems engineer

PDF · CH. 01

ABOUT THIS SAMPLE

A free first chapter

What follows is Chapter 1 of *Concurrency in Practice*, reproduced in full and provided at no cost by DataShelf Hub. It is unabridged — nothing has been trimmed or summarised. If it's useful to you, the remaining eight chapters continue exactly where this one leaves off.

This sample is for your personal use while you decide whether the book is right for you. Please don't republish or redistribute it — share a link to the book's page instead.

TITLE	Concurrency in Practice
AUTHOR	Omar El Alaoui
SHELF	Backend Engineering
FORMAT	PDF, DRM-free, 318 pages
THIS SAMPLE	Chapter 1 of 9 — 15 pages
PUBLISHER	DataShelf Hub · datashelfhub.com

WHAT'S IN THE FULL BOOK

Table of contents

9 chapters · 27 sections

Continued on the next page →

TABLE OF CONTENTS, CONTINUED

Chapters 5–9

CHAPTER ONE

01

Why Concurrency Bites

Why code that works perfectly alone falls apart the moment two threads touch the same memory — and the mental model that keeps it from happening to you.

The bug report says "happens maybe once a day, can't reproduce it." You stare at the code for an hour. It's correct — obviously correct, the kind of correct where you can trace every line and see exactly what it does. And it is correct, in isolation. What it isn't is correct when a second thread runs the same function at the same moment, reads the same variable half a microsecond before the first thread finishes writing it, and carries on with a value that was never meant to exist.

This is what makes concurrency different from every other kind of bug. A logic error is wrong every time you run it. A concurrency error is right almost every time you run it — right in tests, right in staging, right for the first six months in production — and then wrong exactly once, under exactly the interleaving of instructions your test suite never happened to produce. You can't fix what you can't reliably reproduce, which is why this chapter starts not with tools but with a way of thinking that catches the bug before it ships.

1.1 Shared State and the Illusion of Order

Single-threaded code gives you a comforting illusion: that your program executes in the order you wrote it, one statement finishing before the next begins. Inside one thread, that's true. The moment a second thread enters the picture, it stops being true for anything those threads touch together — and it was never true at the hardware level to begin with.

Modern CPUs and compilers reorder instructions constantly, for performance, as long as the reordering doesn't change the outcome for a single thread observing its own operations. Nothing in that guarantee protects a second thread watching the same memory. Without explicit synchronisation, thread B is allowed to observe thread A's writes in a different order than thread A made them — not as a bug, but as specification.

> *"Your program has no order of operations except the one you enforce. Everything else is the compiler and the CPU doing you a favour you didn't ask for and can't rely on."*

Consider two threads sharing a plain boolean `ready` and a plain integer `value`. Thread A sets `value = 42` then `ready = true`. Thread B waits for `ready` to become true, then reads `value`. On paper this looks safe. Without a memory barrier, nothing stops the CPU or compiler from making `ready` visible to thread B before `value` is, and B reads a stale default instead of 42 — a bug that will pass every test you write on a machine with a forgiving memory model and appear only under specific load, on specific hardware.

This is why "it worked when I tested it" carries almost no weight in concurrent code. Testing single-threaded logic exhaustively is hard but tractable — you can enumerate inputs. Testing concurrent logic exhaustively means enumerating interleavings, and the number of possible interleavings of even a handful of operations across two threads grows fast enough that "I ran it and it was fine" tells you almost nothing about the interleaving that will actually occur in production, at 3am, under load you didn't test at.

WHAT "THREAD-SAFE" ACTUALLY HAS TO PROMISE

Correctness	the same result regardless of how operations from different threads interleave
Visibility	a write by one thread is guaranteed observable by another, in order
Progress	no thread can be starved or deadlocked by the synchronisation itself

Most code that "happens to work" under concurrent access satisfies correctness by accident and fails silently on visibility — it reads a value that isn't stale often enough for anyone to notice, until a change in hardware, JIT behaviour, or load pattern makes the race window wide enough to hit. The fix is never "be more careful." It's designing shared state out of the hot path wherever you can, and reaching for explicit, well-understood synchronisation — covered from Chapter 3 onward — everywhere you can't.

Keep this one idea from the section: shared mutable state is not dangerous because programmers are careless. It's dangerous because the guarantees you assume from single-threaded reasoning simply don't apply, and nothing about the code's appearance warns you when you've silently stepped outside them.

1.2 Latency Versus Throughput

Before reaching for concurrency, it's worth being precise about which problem you're actually solving, because the two goals people lump together as "make it faster" pull in different directions. Latency is how long one request takes, start to finish. Throughput is how many requests the system completes per second. Concurrency can improve either — but rarely both with the same technique, and optimising for one can quietly make the other worse.

Adding threads to parallelise a single slow request can cut its latency, at the cost of consuming more resources per request — which reduces how many requests you can run concurrently, hurting throughput under load. Conversely, techniques that maximise throughput, like batching requests together or queuing work behind a fixed-size pool, routinely add latency to any individual request sitting in that queue.

THE SAME SYSTEM, TWO HONEST QUESTIONS

Latency-bound	"How fast can one user's request finish?" — favours parallelising within a request
Throughput-bound	"How many requests per second can we sustain?" — favours efficient queuing and batching

A batch analytics pipeline is almost always throughput-bound: nobody is waiting on any one row, so the right move is maximising total work done per second, even if that means an individual unit of work waits in a queue for a while. An interactive API endpoint is usually latency-bound: a user is waiting, right now, for this one response, and system-wide throughput matters only insofar as it keeps that one request from queuing behind others.

Naming which one you're optimising for, before you write any concurrent code, prevents a specific and common mistake: adding parallelism to a request handler because "concurrency makes things faster," when the actual bottleneck was queue depth under load — a throughput problem that more threads per request makes measurably worse, not better.

RULE OF THUMB

If you can't say in one sentence whether you're optimising latency or throughput, you're not ready to choose a concurrency strategy yet — you're ready to go measure which one is actually the problem.

1.3 A Mental Model for the Book

Every chapter from here treats concurrency as a series of trade-offs between three things: correctness, latency, and throughput — never a free upgrade you bolt onto working code. The book's running position is that most concurrency bugs come from skipping the step of naming, explicitly, what a piece of shared state is allowed to guarantee before you let two threads near it.

Concretely, that means asking four questions of any concurrent design, before implementation: what state is actually shared, what visibility guarantee each reader needs, what happens if two writers race, and what the cost of the synchronisation itself is — because a lock that makes your code correct but serialises every request through it has just traded a concurrency bug for a throughput ceiling.

Those four questions recur, in different clothes, in every remaining chapter: as the choice between threads and async runtimes in Chapter 2, as the real cost of a mutex in Chapter 3, as the conditions under which lock-free structures earn their complexity in Chapter 4, and all the way through to the checklist for running concurrent code in production in Chapter 9.

None of this is abstract for its own sake. The reason to build this mental model before touching a mutex is that concurrency bugs are expensive precisely because they're rare and environment-dependent — the ones that reach production are the ones that passed every review and every test run right up until the interleaving that broke them actually occurred.

A little discipline up front — naming the shared state, naming the guarantee, naming the cost — turns most of those bugs into design decisions you made on purpose instead of races you discover from an incident channel. That discipline is what the rest of this book is teaching, one concurrency primitive at a time.

Let's start with the most basic decision every concurrent system has to make: whether to run work on threads, on an async runtime, or on some mix of the two — and what each one is actually good at.

§ Chapter Summary

Eight things worth carrying into Chapter 2.

BEFORE YOU MOVE ON

01 Single-threaded reasoning about order doesn't survive contact with a second thread — the compiler and CPU only guarantee order within one thread's own view.

02 A concurrency bug can pass every test and still be wrong; "it worked when I tried it" proves almost nothing.

03 Latency and throughput are different goals that often trade off against each other — know which one you're solving before reaching for threads.

04 Name the shared state, the visibility guarantee, and the cost of synchronisation before writing concurrent code, not after debugging it.

05 A lock that makes code correct but serialises every request has traded a concurrency bug for a throughput ceiling — that's a decision, not a fix.

BEFORE CHAPTER 2

Find one piece of shared mutable state in a service you maintain — a cache, a counter, a connection pool. Write down, in one sentence each, what guarantee its readers actually need and what currently enforces it. If the honest answer to the second part is "nothing, it's just usually fine," that's the state worth revisiting before Chapter 2.

END OF SAMPLE

Eight chapters still to go.

Chapter 1 builds the mental model; the rest of the book puts it to work — choosing between threads and async runtimes, the real cost of a mutex, when lock-free patterns earn their complexity, and how to debug a race that only happens once a day in production.