bench_test.go

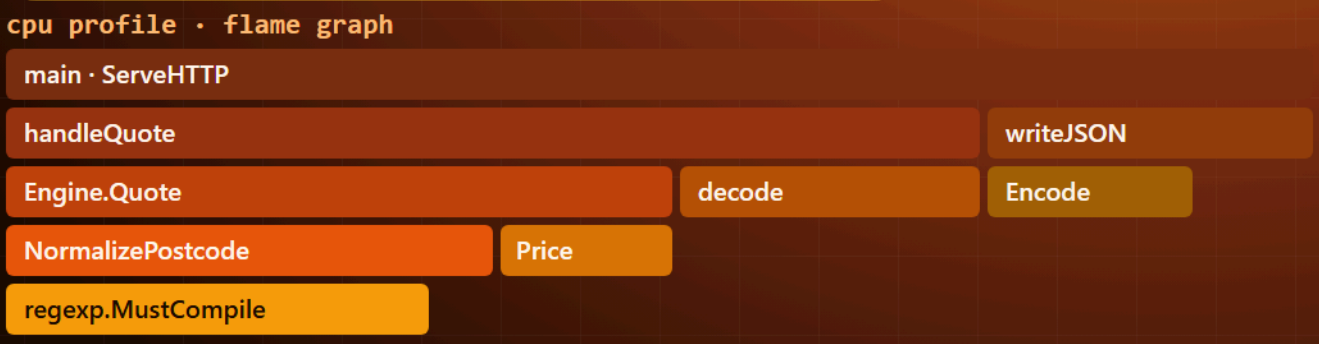
```
func BenchmarkQuote(b *testing.B) {
  e := rates.NewEngine()
  b.ReportAllocs()
  for b.Loop() {
    e.QuoteInto(&q, &req)
  }
}
```

pprof

go tool trace

GOMEMLIMIT

PGO




```
$ go test -bench . -benchmem -count 10
$ go tool pprof -top cpu.out
$ benchstat old.txt new.txt
```

MEASURE · PROFILE · OPTIMIZE

High-Performance Go

Benchmark, profile, and optimize Go for speed, memory, and latency

Go 1.27 · testing.B and b.Loop · benchstat · pprof · traces
escape analysis · GC tuning · PGO · budgets in CI

CPU · MEMORY · CONCURRENCY · I/O · PRODUCTION

THENEXBOOKS

High-Performance Go

Benchmark, Profile, and Optimize Go for Speed, Memory, and Latency
From testing.B and pprof to the garbage collector, PGO, and performance budgets in CI — for Go 1.27

First Edition

THENEXBOOKS

High-Performance Go
Benchmark, Profile, and Optimize Go for Speed, Memory, and Latency

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. Go, its runtime, and the libraries discussed in this book evolve; always consult the official documentation for the versions you use.

About the measurements. Every benchmark result, profile value, and timing printed in this book was measured on our test machine with the exact code shown. Your numbers will differ with your hardware, operating system, Go version, and background load. Compare relative changes on your own machine rather than absolute numbers across machines.

Go and the Go gopher are trademarks of Google LLC. Windows and Visual Studio Code are trademarks of Microsoft Corporation. Linux is a registered trademark of Linus Torvalds. GitHub is a trademark of GitHub, Inc. AMD and Ryzen are trademarks of Advanced Micro Devices, Inc. SQLite is in the public domain; "SQLite" is a registered trademark of Hipp, Wyrick & Company, Inc. Grafana and Pyroscope are trademarks of Grafana Labs. Other product names are trademarks of their respective owners. Their use in this book is for identification only and does not imply endorsement. This book is independent and is not affiliated with, sponsored by, or endorsed by Google or the Go project.

Brisklane, the parcel-shipping company used as the running example in this book, is fictional, as are its people, hubs, customers, carrier partners, and data. Domains such as example.com and brisklane.example are reserved for examples.

Preface

Go has a reputation for being fast, and it earns it: programs compile to native code, the garbage collector is designed for low pauses, and goroutines make concurrency cheap. But "Go is fast" is a statement about the language, not about your program. A service written in Go can still spend most of its time compiling the same regular expression over and over, allocating a fresh buffer for every response, waiting on a single mutex, or opening a new TCP connection for every outbound call. None of that shows up in code review. All of it shows up in latency, in memory, and in the bill for the machines you run on.

This book is about finding and fixing those problems with evidence instead of folklore. We measure first, change one thing, and measure again. We use the tools that ship with Go — `testing.B`, `benchstat`, `pprof`, execution traces, the compiler's own optimization reports, and runtime metrics — and we learn what each of them can and cannot tell us. Every optimization in the book is justified by a measurement, and every measurement is labeled for what it is: one run on one machine.

We work on a real code base. Brisklane's quote service starts slow but correct, with a catalog of fourteen planted performance problems, each one typical of what we find in production Go code. Chapter by chapter we find them with the right tool, explain why they cost what they cost, fix them, and prove the fix. By the end you will have a repeatable method, not a list of tricks.

Who This Book Is For

- **Go developers** who build services, command-line tools, or libraries and need them to be faster, leaner, or more predictable under load.
- **Backend and platform engineers** responsible for latency targets, memory limits in containers, and cloud costs.
- **Site reliability engineers** who want to profile production services safely and catch regressions before users do.
- **Tech leads** who want their team to argue about performance with benchmarks and profiles rather than opinions.

You should be comfortable writing Go: packages, structs, interfaces, error handling, goroutines, channels, and tests. We do not teach the language itself, and we explain every runtime and compiler concept before we rely on it.

What You Will Learn

- How to write benchmarks that measure what you think they measure, with `testing.B` and `b.Loop`, and how to compare runs honestly with `benchstat`.
- How to read CPU, heap, allocation, block, mutex, goroutine, and goroutine-leak profiles with `go tool pprof`, and how to use execution traces and the flight recorder.

- What the compiler does for you: inlining, escape analysis, bounds-check elimination, and devirtualization, verified with `-gcflags=-m`.
- How to reduce allocations in real code: strings and bytes, slices and maps, struct layout, interfaces and generics, and `sync.Pool`.
- How the garbage collector works and how to tune it with `GOGC` and `GOMEMLIMIT`; how to use profile-guided optimization; and how the scheduler and `GOMAXPROCS` affect throughput.
- How to make concurrent code scale, and how to make I/O, HTTP, JSON, and database access cheaper.
- How to load-test a service, reason about percentiles and tail latency, run continuous profiling, and enforce performance budgets in CI.

How This Book Is Organized

Part I — Measure Before You Optimize (Chapters 1–5) sets up the method: benchmarks, statistics, and CPU and memory profiles. **Part II — Seeing Inside Your Program** (Chapters 6–10) adds the profiles for waiting and leaks, execution traces, the compiler's optimization reports, memory layout, and profiling of live services. **Part III — Allocation-Lean Code** (Chapters 11–15) turns measurements into leaner code. **Part IV — The Runtime: GC, PGO, and Concurrency** (Chapters 16–20) explains the garbage collector, memory tuning, profile-guided optimization, the scheduler, and scalable concurrency. **Part V — I/O, Networking, Encoding, and Data** (Chapters 21–24) covers files, HTTP, JSON, and databases. **Part VI — Performance in Production** (Chapters 25–28) covers load testing, continuous profiling, performance budgets in CI, and a complete before-and-after review. The appendices provide a cheat sheet, templates and command references, a production checklist, and a glossary.

Each chapter contains "Try It Yourself" exercises and ends with a **Key Takeaways** box for quick review.

The Running Example: Brisklane

Brisklane is a fictional parcel-shipping company. Online shops call its quote API at checkout to show shipping prices, and its sorting hubs report a steady stream of tracking scans. Odette leads the backend team and owns `quoted`, the quote service; Bram is the site reliability engineer who watches latency and memory in production; and Ilse is the data engineer who owns the scan pipeline and the shipments database. The reference project, `brisklane`, contains the quote engine, the scan parser, a SQLite-backed store, a carrier-partner client, the HTTP server, and a load generator — all deliberately slow but correct. Each planted problem has an ID from PERF-01 to PERF-14, and the chapter that fixes it says so.

Versions and Conventions

Examples target **Go 1.27**, the current release at the time of writing, and were run with Go 1.27.1. Most techniques apply to any recent Go release; where a feature needs a specific version, we say so. The reference project uses one third-party module, the pure-Go SQLite driver `modernc.org/sqlite`, plus `benchstat` from `golang.org/x/perf` as a tool.

Measurements. Benchmark and profile output in this book is real output from our test machine (Go 1.27.1, `windows/amd64`, an AMD Ryzen 9 5950X with 16 cores and 32 hardware threads). Your numbers will differ. What should carry over is the direction and rough size of a change, which is why we prefer `benchstat` comparisons to single numbers.

> **TIP**
> A recommended practice or a technique that makes your work easier.

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

> **WARNING**
> A common mistake or something that can mislead your measurements or hurt production.

Code appears in `monospace`. Long lines are wrapped to fit the page. File names above code blocks, such as `brisklane/internal/rates/engine.go`, tell you where the code lives. Commands are shown for a terminal and work the same on Windows, macOS, and Linux unless noted.

Contents

Preface

PART I — MEASURE BEFORE YOU OPTIMIZE

1 Performance Is a Feature: Speed, Memory, Latency, and the Brisklane Baseline

2 Benchmarking Correctly: testing.B, b.Loop, and the Traps That Lie to You

3 Comparing Results Honestly: benchstat, Variance, and a Quiet Machine

4 CPU Profiling with pprof: Finding Where the Time Goes

5 Memory Profiling: Heap and Allocation Profiles

PART II — SEEING INSIDE YOUR PROGRAM

6 Block, Mutex, and Goroutine Profiles: Waiting, Contention, and Leaks

7 Execution Traces: go tool trace, Tasks, Regions, and the Flight Recorder

8 Inside the Compiler: Inlining, Escape Analysis, and Bounds-Check Elimination

9 Memory Layout and Allocation Costs: Stacks, the Heap, Pointers, and Padding

10 Profiling Live Services: net/http/pprof, Profile Labels, and Safe Exposure

PART III — ALLOCATION-LEAN CODE

11 Reducing Allocations: Preallocation, Reuse, Append-Style APIs, and Hidden Copies

12 Strings and Bytes: Building, Converting, Parsing, and Interning

13 Slices and Maps: Growth, Capacity, Clearing, Sorting, and Iteration Costs

14 Interfaces, Generics, and Function Values: What Abstraction Costs

15 sync.Pool and Object Reuse: When Pooling Pays and When It Hurts

PART IV — THE RUNTIME: GC, PGO, AND CONCURRENCY

16 How the Go Garbage Collector Works: Pacing, Green Tea, and gctrace

17 Tuning Memory: GOGC, GOMEMLIMIT, and Bounded Caches

18 Profile-Guided Optimization: Building with default.pgo

19 The Scheduler, GOMAXPROCS, and the Real Cost of Goroutines

20 Concurrency Performance: Contention, Atomics, Sharding, and False Sharing

PART V — I/O, NETWORKING, ENCODING, AND DATA

21 Fast I/O: Buffering, bufio, io.Copy, and System Call Costs

22 HTTP and Network Performance: Connection Reuse, Timeouts, and Server Tuning

23 JSON and Encoding Costs: encoding/json, json/v2, Streaming, and Alternatives

24 Database Access Performance: Pools, Prepared Statements, Batching, and N+1 Queries

PART VI — PERFORMANCE IN PRODUCTION

25 Latency Under Load: Load Testing, Percentiles, and Tail Latency

26 Continuous Profiling and Runtime Metrics in Production

27 Performance Budgets in CI: Benchmark Gates, Allocation Budgets, and Regressions

28 Case Study: The Brisklane Tuning Review, from Baseline to Budget

APPENDICES

A High-Performance Go Cheat Sheet

B Templates and Command Reference

C Performance Production Checklist

D Glossary

PART I

Measure Before You Optimize

What "fast" means for a Go service — throughput, latency, memory — and the measuring habits everything else depends on: benchmarks that do not lie, honest comparisons with benchstat, and CPU and memory profiles that point at the real hot spots in Brisklane's quote engine.

1 · Performance Is a Feature

2 · Benchmarking Correctly

3 · Comparing Results with benchstat

4 · CPU Profiling with pprof

5 · Memory Profiling

CHAPTER 1 · PART I — MEASURE BEFORE YOU OPTIMIZE

Performance Is a Feature: Speed, Memory, Latency, and the Brisklane Baseline

Fast software is not an accident and not a personality trait: it is a feature with requirements, tests, and owners. In this chapter we define what "performance" means in numbers you can measure, meet the Brisklane team and the quote service we will tune for the rest of the book, and take our first honest baseline.

What We Mean by Performance

Ask five engineers whether a service is "fast" and you will get five different answers, because the word hides at least five separate quantities. Before we change a single line of code, we want names for them, because each one is measured with a different tool and each one is improved by different changes. A fix that helps one can easily hurt another.

Throughput

Throughput is work completed per unit of time: requests per second for a web service, megabytes per second for a parser, rows per second for a batch job. Throughput is what capacity planning cares about. If one server handles a certain number of quotes per second, the number of servers you need follows from the traffic you expect. Throughput is usually measured with the system saturated, or close to it, which is exactly the state in which other qualities, like latency, look their worst.

Latency and percentiles

Latency is how long one operation takes from the caller's point of view. For an online shop calling a shipping-price API during checkout, latency is the number the customer feels. The important word is *distribution*: a service does not have "a latency", it has thousands of them per second, and they are never all the same. Most requests are quick; a few are slow because they hit a garbage collection, a cold cache, a lock held by somebody else, or a slow disk.

That is why we describe latency with **percentiles**. The p50 (median) is the value half the requests beat. The p99 is the value 99% of requests beat, so one request in a hundred is slower. The p99 matters more than it seems: a checkout page that calls three backend services, each with a 1% chance of a slow answer, shows a slow page much more often than 1% of the time, and a busy shop makes many such calls every second. The mean (average) hides all of this, because a few very slow requests and many fast ones can average out to a comfortable-looking number. Chapter 25 is devoted to measuring latency under load correctly; for now, remember that we always report latency as percentiles and never as a single average.

Memory

Memory has two faces in Go. The first is the *footprint*: how much RAM the process holds, which decides whether it fits in its container and how many instances share a machine. The second is *allocation rate*: how many bytes and objects per second the program asks the runtime for. Go has a garbage collector, so every allocation is a small debt the collector must later pay with CPU time. A program can have a small footprint and a huge allocation rate, and it will spend a surprising share of its CPU in the collector. Benchmarks report allocations per operation for exactly this reason, and we will treat `allocs/op` as one of the most useful numbers in the book.

CPU

CPU time is how much processor work an operation needs, regardless of how long the caller waits. A request can take 10 ms of wall-clock time while using only a fraction of a millisecond of CPU, because it spent the rest waiting for a database. Conversely, a parallel computation can burn several CPU-seconds in one wall-clock second. CPU profiles (Chapter 4) show where CPU time goes; execution traces (Chapter 7) show where wall-clock time goes, including the waiting.

Cost

Finally, **cost**: the money spent on machines, the energy they use, and the engineering time spent making and keeping them fast. Cost ties the other four together. Halving CPU per request may let you run fewer instances; halving memory may let you pick a cheaper instance shape; and a clever optimization that nobody on the team can maintain has a cost too. We will keep asking whether a change is worth what it costs to keep.

> **NOTE**
>
> These quantities trade against each other. Caching trades memory for CPU and latency. Batching trades latency for throughput. Running more goroutines can raise throughput and also raise tail latency through contention. When someone says a change made the service "faster", the first useful question is: faster by which measure, and what got worse?

Where Time Goes: Reasoning Before Measuring

Profiles tell us where a program spends its time, but a little arithmetic tells us whether a fix *can* matter before we write it. The reasoning is usually called Amdahl's law. If a part of the work takes a fraction p of the total, and we make that part s times faster, the whole thing becomes

```
new time = (1 - p) + p / s      (as a fraction of the old time)
speedup  = 1 / ((1 - p) + p / s)
```

The consequence is sobering. If a function accounts for 10% of a request, making it infinitely fast saves at most 10%. Making it twice as fast saves 5%. No amount of cleverness changes that ceiling; only finding a bigger p does.

Here is a real example from the Brisklane service you are about to meet. We profiled the benchmark that sends the standard quote request through the whole HTTP handler (Chapter 4 explains the tools; the run used one CPU to keep the picture simple). On our test machine, with your numbers sure to differ, the cumulative shares were roughly: JSON decoding of the request 40%, JSON encoding of the response 22%, building the test request 17%, and building a cache key 11%. The quote engine itself did not even appear, because this particular benchmark answers from a cache after the first request.

Amdahl's law now tells us something useful. If we made the response encoding infinitely fast, the best possible speedup of this benchmark would be 1 / (1 − 0.22), about 1.28×. If we made decoding infinitely fast, the ceiling would be 1 / (1 − 0.40), about 1.67×. And no change to the pricing code could speed up this benchmark at all, because it does not run the pricing code. A team that "optimizes the engine" on the strength of a hunch would do real work and see nothing. That is the most common way performance work fails: not with wrong code, but with the right code in the wrong place.

> **TIP**
>
> Before starting an optimization, write down the share of time or allocations the target accounts for, and the best case your change can achieve. If the ceiling is a 3% gain on a path nobody waits for, there is probably something better to do this week.

When Not to Optimize

Performance is a feature, and like every feature it competes for time with other features. Some situations clearly call for leaving the code alone:

- **There is no requirement.** If nobody can say how fast the code needs to be, nobody can say when you are done. Start by writing a goal (we do that at the end of this chapter).
- **The code is not on a hot path.** A startup routine that runs once, an admin page used twice a day, or a nightly report that finishes in two minutes with a six-hour window are rarely worth tuning.
- **You have not measured.** Intuition about performance is unreliable even for experienced engineers, and Go's compiler and runtime change between releases. Code that was slow in one release may be fine in the next.
- **The change costs clarity you cannot afford.** Hand-written parsers, pooled objects, and unsafe tricks have a maintenance price. We will use some of them, but only when a measurement says they pay.

Equally, some situations call for acting early. A latency target in a contract, a memory limit on a container, a cloud bill growing faster than traffic, or a hot path that every customer request crosses are all good reasons to invest. Brisklane has several of these, which is why it makes a good running example.

The Loop: Measure, Find, Fix, Verify

Every chapter in this book follows the same four steps, and we recommend you follow them in your own work:

1. **Measure.** Establish a baseline with a benchmark, a load test, or production metrics. Without a baseline there is no "before", so there can be no "after".
2. **Find.** Use a profiler, a trace, compiler diagnostics, or runtime metrics to locate the cost. Never guess where the time goes when a tool can tell you.
3. **Fix.** Make the smallest change that addresses the cause the tool pointed at. Understand *why* it was slow; the mechanism is what lets you avoid the same mistake elsewhere.
4. **Verify.** Measure again, the same way, on the same machine, and compare with statistics (Chapter 3). Keep the improvement from regressing with a test or a budget (Chapter 27).

The loop sounds obvious, yet each step is easy to skip. People skip *measure* because the problem "is obviously the database". They skip *find* because they already know a trick they want to try. They skip *verify* because the change "can only help". The rest of Part I exists to make each step cheap enough that you will not want to skip it.

Meet Brisklane

Brisklane is a fictional parcel-shipping company. Online shops call its **quote API** during checkout to show shipping prices; that call is on the shop's critical path, so every millisecond Brisklane spends is a millisecond the shopper waits. Meanwhile, eight sorting hubs report a steady stream of **tracking scans** (picked up, arrived, departed, out for delivery, delivered, exception), which a data pipeline parses and aggregates.

Three people own different parts of the system, and you will see their names throughout the book:

- **Odette** is the backend lead. She owns `quoted`, the quote HTTP service, and the quote engine behind it.
- **Bram** is the SRE. He cares about production latency and memory, profiling in production, and the performance checks in CI.
- **Ilse** is the data engineer. She owns the scan pipeline and the shipments database.

The hot path is simple to draw. A shop sends a JSON request with an origin postcode, a destination postcode, a service level, and one or more parcels. `quoted` decodes the request, checks a cache, prices the request with the rate engine if needed, and writes a JSON quote back.

```
shop checkout ──POST /v1/quotes──> quoted
                                   | decodeQuoteRequest   (JSON -> rates.Request)
                                   | quote cache lookup
                                   | rates.Engine.Quote   (postcodes, zone, bands,
                                   |                       table, surcharge rules)
                                   | writeJSON            (rates.Quote -> JSON)
                                   | Stats.Record         (per-route counters)
shop checkout <──── 200 + JSON ────┘
```

The worked example: a $15.68 quote

One request appears in almost every chapter, so it is worth computing by hand once. A shop in postcode 30412 ships a 1,200 g box measuring 30 × 20 × 10 cm to postcode 91783 with the standard service:

Request body (the canonical example)

```
{"origin":"30412","destination":"91783","service":"standard",
 "parcels":[{"weight_g":1200,"length_cm":30,"width_cm":20,"height_cm":10}]}
```

- **Zone.** The zone depends on the first digit of each postcode: |3 − 9| = 6, and 6 / 3 + 1 = **3**.
- **Billable weight.** Brisklane charges the larger of the actual weight and the volumetric weight (length × width × height / 5): 30 × 20 × 10 / 5 = 1,200 g, so the billable weight is **1,200 g**, which falls into weight band **2** (up to 2,000 g).
- **Base price.** Each service has a tariff: price = Base + (zone − 1) × ZoneStep + band × BandStep. Standard is 495 / 150 / 200 cents, so 495 + 2 × 150 + 2 × 200 = **1,195** cents.
- **Remote area.** Destinations starting with 9 pay a **250**-cent surcharge.
- **Fuel.** A fuel surcharge of 8.5% (850 basis points) applies to the subtotal of 1,445 cents, rounded half up to the cent: (1445 × 850 + 5000) / 10000 = **123** cents.

The total is **1,568 cents, or $15.68**, with three lines: base 1195, remote 250, fuel 123. Money is always integer cents in an `int64`, and the currency is always `"USD"`. Here is the service computing the same thing. We started `quoted` and sent the request with `curl.exe` (on Windows, plain `curl` in PowerShell 5.1 is an alias for `Invoke-WebRequest`, so we call the real curl explicitly):

Terminal

```
go run ./cmd/quoted -addr localhost:8088
# in a second terminal, with the body above saved as canonical.json:
curl.exe -s -X POST -H "Content-Type: application/json" `
  --data-binary "@canonical.json" http://localhost:8088/v1/quotes
```

Output (our test machine, wrapped for the page)

```
{"service":"standard","zone":3,"billable_g":1200,
 "lines":[{"kind":"base","amount_cents":1195},{"kind":"remote","amount_cents":250},
 {"kind":"fuel","amount_cents":123}],"total_cents":1568,"currency":"USD"}
```

Posting the same body to `/v1/quotes/all` prices every service and returns them cheapest first: economy 1242, standard 1568, and express 2544 cents. These numbers are deterministic; if your copy of the project prints anything else, a change broke the pricing, and no speedup is worth that.

A Tour of the Reference Project

The book ships with a reference project, `brisklane`, that implements this system in plain Go with one dependency (a pure-Go SQLite driver). It is deliberately slow in fourteen specific places, and correct everywhere. The layout is small enough to hold in your head:

```
brisklane/
  go.mod                 module example.com/brisklane (go 1.27.0)
  cmd/quoted/            the HTTP service (-addr, default localhost:8088; -db)
  cmd/loadgen/           closed-loop load generator (-url, -n 20000, -c 16)
  cmd/gendata/           writes scan files and seeds the SQLite database
  cmd/scanagg/           reads a scan file and prints per-hub statistics
  internal/rates/        services, postcodes, zones, the price table, the engine
  internal/scan/         scan-line parsing, aggregation, file reading
  internal/store/        SQLite store: shipments and scans
  internal/carrier/      client for a partner's label API (plus a fake partner)
  internal/server/       HTTP handlers, quote cache, JSON helpers, stats
  internal/gen/          deterministic data generators (seed 2026)
```

Everything that produces data uses a fixed seed (`gen.Seed` is 2026), so your generated requests, scan files, and database are byte-for-byte identical to ours. That matters more than it may seem: benchmarks are only comparable when they process the same input. For example, `gen.Requests(1000, 2026)` produces 1,000 varied but valid quote requests whose totals sum to exactly 3,701,251 cents, and a test checks that sum.

The first thing to do with any project you intend to optimize is to run its tests, because an optimization that breaks behavior is a bug with good marketing.

Terminal

```
cd brisklane
go test ./...
```

Output (our test machine)

```
?       example.com/brisklane/cmd/gendata       [no test files]
?       example.com/brisklane/cmd/loadgen       [no test files]
?       example.com/brisklane/cmd/quoted        [no test files]
?       example.com/brisklane/cmd/scanagg       [no test files]
ok      example.com/brisklane/internal/carrier  0.048s
?       example.com/brisklane/internal/gen      [no test files]
ok      example.com/brisklane/internal/rates    0.069s
ok      example.com/brisklane/internal/scan     1.692s
ok      example.com/brisklane/internal/server   0.042s
ok      example.com/brisklane/internal/store    0.356s
```

Five packages pass. The tests pin down the worked example, a second multi-parcel example (5,089 cents), the 1,000-request sum, the scan-file sizes, and the database seed. Whenever we change code in later chapters, these tests are the safety net that tells us the change was an optimization and not a behavior change.

The PERF Catalog: The Book's Roadmap

The fourteen slow spots in `brisklane` are not random. Each is a mistake we have seen in real Go services, and each is marked in the source with a `// PERF-NN:` comment naming the chapter that finds and fixes it. Together they form the roadmap of the book:

Id	Planted problem	Found in	Fixed in
PERF-01	`NormalizePostcode` compiles a regexp on every call	Ch 4	Ch 4
PERF-02	`Table.Price` builds a `fmt.Sprintf` map key per lookup	Ch 5	Ch 5
PERF-03	`Quote` returns a pointer and grows lines from empty; the request escapes	Ch 8	Ch 8
PERF-04	`ParseLine` uses Split, ToUpper and Sscanf	Ch 5, 12	Ch 12
PERF-05	`Aggregate` has no size hints and uses `sort.Slice`	Ch 13	Ch 13
PERF-06	`scan.Event` field order wastes 16 bytes (80 instead of 64)	Ch 9	Ch 9
PERF-07	`writeJSON` allocates a buffer and encoder per response	Ch 5, 15	Ch 15
PERF-08	Unbounded quote cache with `%+v` keys	Ch 16	Ch 17 (key: Ch 11)
PERF-09	One global mutex in `Stats.Record`	Ch 6	Ch 20
PERF-10	`scan.ReadFile` reads one byte per system call	Ch 7, 21	Ch 21
PERF-11	Partner client creates a `Transport` per call	Ch 6	Ch 22
PERF-12	Request JSON decoded into `map[string]any`	Ch 4, 5	Ch 23
PERF-13	Per-row commits, an N+1 query, a missing index	Ch 24	Ch 24
PERF-14	`QuoteAll` fan-out leaks goroutines	Ch 6	Ch 6, 20

Notice the "Found in" column. We never fix a problem before a tool has shown it to us, even though the comments in the code give the game away. That discipline is the point of the exercise: in your own code there are no `PERF` comments, only tools. Chapter 28 brings everything together in a before-and-

after review.

Our First Baseline: Benchmarks

A **benchmark** in Go is a function in a `_test.go` file that the `go test` command calls repeatedly while timing it. Chapter 2 takes benchmarks apart in detail; here we only run the ones that ship with the project, to have a first set of numbers. This is the benchmark for the quote engine:

brisklane/internal/rates/bench_test.go

```
func BenchmarkQuote(b *testing.B) {
    e := NewEngine()
    req := canonicalRequest()
    b.ReportAllocs()
    for b.Loop() {
        if _, err := e.Quote(req); err != nil {
            b.Fatal(err)
        }
    }
}
```

The setup (an engine and the worked-example request) runs once; the body of the `for b.Loop()` loop runs as many times as the framework needs for a stable measurement. Run every benchmark in the rates package and skip the ordinary tests:

Terminal

```
go test -run '^$' -bench . -benchmem ./internal/rates
```

Output (our test machine; whitespace trimmed, our extra experiment benchmarks removed)

```
goos: windows
goarch: amd64
pkg: example.com/brisklane/internal/rates
cpu: AMD Ryzen 9 5950X 16-Core Processor
BenchmarkNormalizePostcode-32     73466     17453 ns/op     6068 B/op    91 allocs/op
BenchmarkTablePrice-32          4187650     296.8 ns/op       16 B/op     1 allocs/op
BenchmarkQuote-32                 39219     30977 ns/op    12502 B/op   188 allocs/op
BenchmarkQuoteAll-32               9150    159410 ns/op    38075 B/op   569 allocs/op
BenchmarkQuoteMix-32              36669     37035 ns/op    12504 B/op   188 allocs/op
PASS
ok      example.com/brisklane/internal/rates    10.083s
```

These numbers were measured on our test machine; your numbers will differ, and so will ours on the next run (Chapter 3 shows by how much). The `-run '^$'` flag is a regular expression that matches no test name, so only benchmarks run. In PowerShell the single quotes keep `$` literal; in `cmd.exe` drop the quotes.

How to read a benchmark line

Each line has the same columns:

Column	Example	Meaning
Name and suffix	`BenchmarkQuote-32`	Benchmark name, then `-N` = GOMAXPROCS during the run (32 here; omitted when it is 1)
Iterations	`39219`	How many times the loop body ran in the final measurement
ns/op	`30977 ns/op`	Wall-clock nanoseconds per iteration (total time / iterations)
B/op	`12502 B/op`	Bytes allocated on the heap per iteration
allocs/op	`188 allocs/op`	Heap allocations per iteration

The iteration count is not a quality measure; the framework simply runs a cheap operation more times to fill its default one-second budget. The two memory columns appear because the benchmark calls `b.ReportAllocs()` or because we passed `-benchmem`. They count heap allocations only: stack allocations are free from the garbage collector's point of view and do not show up.

Already the baseline tells a story. Pricing one parcel took about 31 microseconds and **188 heap allocations** on this machine. That is a lot of work for arithmetic on a handful of integers. `NormalizePostcode` alone needs 91 allocations per call, and `Quote` calls it twice (origin and destination): 182 of the 188 allocations come from checking two five-digit postcodes. We will not fix that yet; Chapter 4 will find it with a profiler, which is the honest way. But you can see why measuring first is so valuable: the expensive part of the quote engine is not pricing at all.

`BenchmarkQuoteMix` cycles through the 1,000 generated requests rather than repeating one, so it exercises multi-parcel requests, padded postcodes, and all three services. Its allocation count is the same 188, which tells us the cost is dominated by per-request overhead rather than per-parcel work. Chapter 2 explains why realistic inputs matter and how to cycle through them without measuring the cycling.

Our First Baseline: A Load Test

Benchmarks measure a function in isolation. Users experience the whole service: HTTP parsing, the network stack, scheduling, garbage collection, and contention between concurrent requests. The project includes a small load generator, `loadgen`, that sends the generated request mix to a running `quoted` from a fixed number of concurrent workers and reports throughput and latency percentiles.

Terminal

```
go build -o quoted.exe ./cmd/quoted
.\quoted.exe -addr localhost:8088
# in a second terminal:
go run ./cmd/loadgen -url http://localhost:8088/v1/quotes -n 20000 -c 16
```

Output (our test machine, first run after starting quoted)

```
requests=20000 errors=0 concurrency=16 elapsed=1.073s
throughput=18642 req/s
latency p50=539.2µs p90=1.7472ms p99=3.9874ms max=31.0094ms
```

Then we ran exactly the same command a second time, without restarting the service:

Output (our test machine, second run)

```
requests=20000 errors=0 concurrency=16 elapsed=745ms
throughput=26851 req/s
latency p50=517.1µs p90=1.0438ms p99=3.216ms max=40.0829ms
```

Same code, same machine, same requests, and throughput rose by about 44%. Why? `loadgen` uses the same seed every time, so the second run sends the same 20,000 requests, and `quoted` keeps every quote it computes in an unbounded cache (that is PERF-08). On the first run nearly every request was a cache miss that ran the slow engine; on the second run all of them were hits. Neither run is "wrong", but they measure different things, and quoting either one without saying which would mislead. Notice also that the maximum latency got *worse* while everything else got better: a single slow request is a poor basis for any conclusion.

> **WARNING**
>
> Load-test results depend on state you cannot see in the command line: warm or cold caches, the size of the heap, what else the machine is doing, and whether the client and server share CPUs (here they did, on one machine). Always record how a number was produced, restart the service between runs you intend to compare, and never compare a warm run with a cold one. Also note that `loadgen` is a closed-loop generator: each worker waits for a response before sending the next request, which hides queueing delays. Chapter 25 adds an open-loop mode and explains why that matters for tail latency.

The service also counts requests per route. After the first load test, `GET /v1/stats` reported `"POST /v1/quotes 200":{"count":20001,...}`: the 20,000 load-test requests plus our one manual `curl` request. That is a handy sanity check that every request in the load test really reached the handler and succeeded.

Setting Goals

A baseline is only useful next to a goal. Brisklane's team wrote down their own targets, and we will use them throughout the book:

- The quote endpoint must answer with **p99 ≤ 5 ms at 2,000 requests per second** on their production servers.

- `quoted` runs in containers with a **512 MiB** memory limit; after Chapter 17 it also runs with `GOMEMLIMIT=400MiB`, following the team's rule of setting the soft limit at about 80% of the container limit.
- From Chapter 27 on, CI enforces **allocation budgets** for the hottest functions, so the gains we make cannot silently disappear.

These are a fictional team's choices for their product, not guidance from the Go project or industry thresholds. A payment API, a game server, and a batch pipeline would choose very different numbers. What matters is the *shape* of a good goal: it names a percentile (not an average), a load level (because latency at idle is meaningless), and the environment where it is checked.

How does our baseline compare? Our closed-loop test reported a p99 of about 4 ms at far more than 2,000 requests per second, which might suggest the target is already met. It is not that simple. The test ran on a 16-core desktop rather than a production server, client and server competed for the same CPUs, the cache made the second run look better than real traffic would, and a closed-loop generator under-reports tail latency when the server stalls. Bram's rule is that a goal is checked under conditions that resemble production, so we will revisit the target with better tools in Chapter 25 and in the case study of Chapter 28. For now we have something more modest and more valuable: numbers we produced ourselves, with a record of how we produced them.

> **NOTE**
>
> The book's test machine is described once in the Preface. All measured numbers in this book come from runs on that machine with the exact code shown, and many of those runs happened while other jobs were using the CPU. Treat absolute times as illustrative and focus on relative changes and allocation counts, which Chapter 3 shows are far more stable.

A Note on Tools and Versions

Everything in this book uses the standard Go toolchain: `go test` for benchmarks, `go tool pprof` for profiles, `go tool trace` for execution traces, and compiler diagnostics through `-gcflags`. The one extra tool is `benchstat` from the Go project's `golang.org/x/perf` module, which we run with `go run` at a pinned version so nothing needs installing (Chapter 3). We verified all behavior against Go 1.27 (specifically Go 1.27.1). Go's compiler and runtime improve with every release, so a number that holds today may change after an upgrade; that is one more reason to measure rather than memorize.

Try It Yourself

1. Run `go test ./...` in your copy of `brisklane`, then run the rates benchmarks twice in a row. Which columns changed between the runs, and which stayed identical?
2. Start `quoted`, post the canonical body to `/v1/quotes/all`, and check that the totals are 1242, 1568, and 2544 cents. Then change the destination to `21783` and predict the new standard total by hand before you send it.