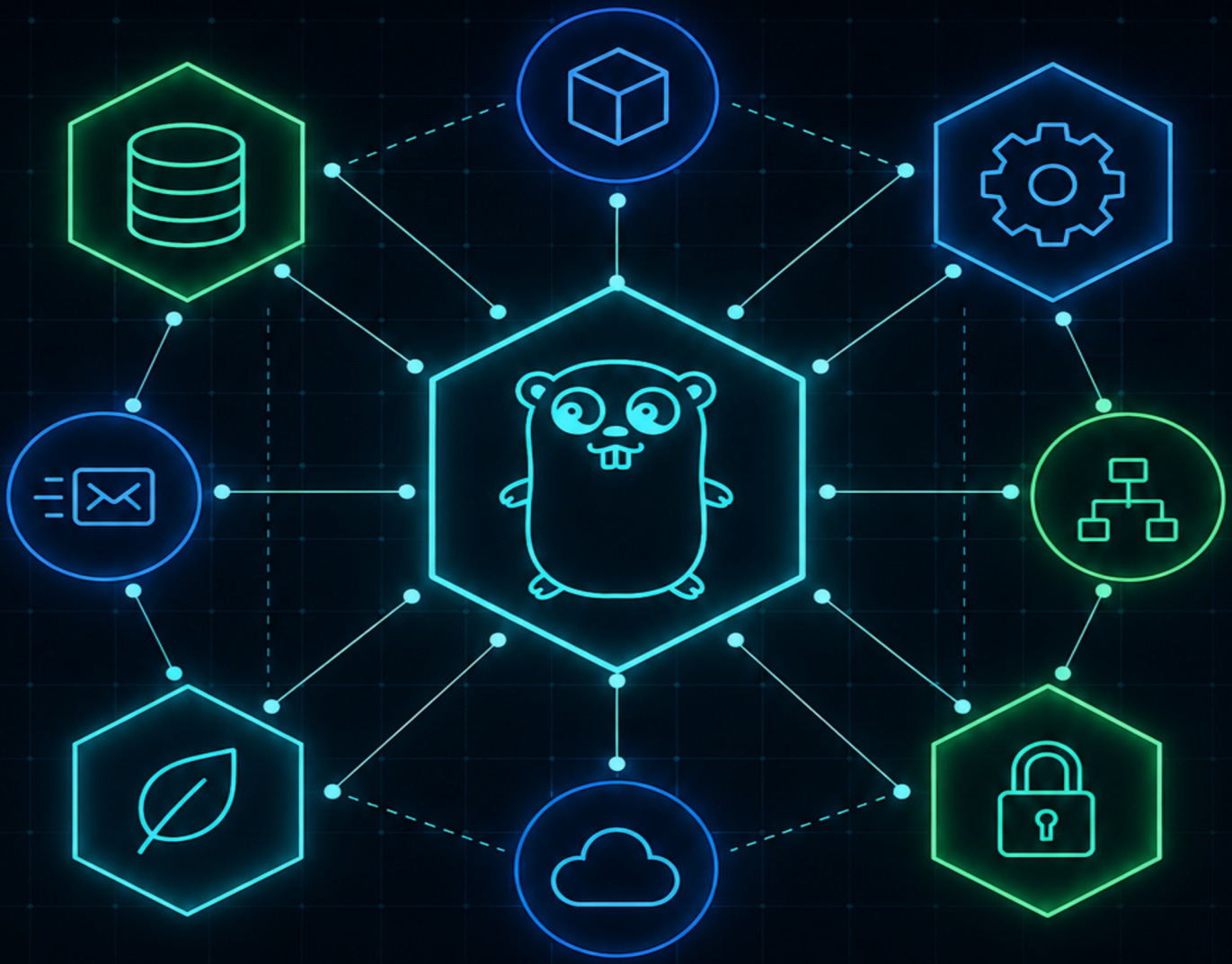
Distributed Systems in Go

A Complete Guide to Building Cloud-Native Distributed Go Systems



Joel Bryan Juliano

Distributed Systems in Go

A Complete Guide to Building Cloud-Native Distributed Go Systems

Joel Bryan Juliano

This book is available at https://leanpub.com/distributed-systems-in-go

This version was published on 2026-05-29

This is a Leanpub book. Leanpub empowers authors and publishers with the Lean Publishing process. Lean Publishing is the act of publishing an in-progress ebook using lightweight tools and many iterations to get reader feedback, pivot until you have the right book and build traction once you do.

Contents

Preface

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/distributed-systems-in-go.

Why Cloud-Native Distributed Systems?

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/distributed-systems-in-go.

Who This Book Is For

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/distributed-systems-in-go.

How to Read This Book

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/distributed-systems-in-go.

About the Code

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/distributed-systems-in-go.

What This Book Is Not

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/distributed-systems-in-go.

Introduction

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/distributed-systems-in-go.

The Problem with Distributed Systems Books

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/distributed-systems-in-go.

The Mental Model

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/distributed-systems-in-go.

The Company SDK

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/distributed-systems-in-go.

The System We Build

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/distributed-systems-in-go.

The Code Style

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/distributed-systems-in-go.

Before You Begin

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/distributed-systems-in-go.

Chapter 1: Why Distributed? The Cost of Coordination

Start with a single binary. One Go program, one process, one machine. It handles HTTP requests, queries a database, and returns results. You can run it, test it, debug it, and deploy it in an afternoon. Everything shares memory. Function calls are free. There is no network.

> This is the best system you will ever build.
>
> So why does anyone build anything more complicated?

The Limits of One

Here is a service that counts page views:

```go
package main

import (
    "fmt"
    "net/http"
    "sync/atomic"
)

var views atomic.Int64

func handler(w http.ResponseWriter, r *http.Request) {
    views.Add(1)
    fmt.Fprintf(w, "Views: %d", views.Load())
}

func main() {
    http.HandleFunc("/", handler)
    http.ListenAndServe(":8080", nil)
}
```

It works. It is fast. It is correct. `atomic.Int64` gives you safe concurrent increments without a lock. You can run this on a laptop and it will handle thousands of requests per second without breaking a sweat.

Now imagine this service is part of a real product. Traffic grows. One machine is not enough. You add a second server behind a load balancer.

The code breaks immediately.

Each server has its own `views` counter. Server 1 counts its requests. Server 2 counts its requests. Neither knows about the other. The number your users see depends on which server they hit. It is wrong by design.

You have just met the first hard rule of distributed systems:

Local state does not distribute. The counter that worked perfectly on one machine becomes a liability the moment you have two. Every piece of local state — in-memory caches, connection pools, counters, session data, in-flight rate limits — is a problem waiting for scale to find it.

This is not a flaw in Go. It is not a flaw in your design. It is the nature of distributed systems: state that is consistent inside one process is inconsistent when that process is replicated.

The Eight Things You Cannot Assume

In 1994, Sun Microsystems engineers compiled a list of assumptions that developers habitually make about networks. Every assumption is wrong. They called them the fallacies of distributed computing:

1. The network is reliable.
2. Latency is zero.
3. Bandwidth is infinite.
4. The network is secure.
5. Topology doesn't change.
6. There is one administrator.
7. Transport cost is zero.
8. The network is homogeneous.

These are not edge cases. They are the baseline reality of every distributed system you will build. Let's take them one at a time with the Go distributed systems lens.

The network is reliable. In a year of running a production service, you will see: packet loss on a noisy network segment, a DNS resolution failure, a TCP timeout when a server is overloaded but still accepting connections, a connection refused

because a service is restarting, and a half-open TCP connection that never fires a FIN. Every network call in your Go code needs to account for all of these. The retry logic in Chapter 9 exists because of this fallacy.

Latency is zero. A function call within a process takes nanoseconds. An HTTP call to a service on the same machine takes microseconds (localhost). An HTTP call to a service in the same data center takes milliseconds. An HTTP call across a continent takes tens of milliseconds. If your handler calls five services in sequence, and each takes 10ms, you have added 50ms of irreducible latency before any business logic runs. Fan-out (Chapter 10) exists because of this fallacy.

Bandwidth is infinite. Not a problem for most services — but your analytics pipeline that copies 100GB of data between services every night will discover this eventually.

The network is secure. In 2018, a major cloud provider had a routing misconfiguration that sent traffic to an unexpected path for several minutes. In distributed systems, you cannot assume that a packet that reaches your service came from a legitimate caller. JWT authentication (Chapter 5) and service tokens (Chapter 9) exist because of this fallacy.

Topology doesn't change. Your service starts up knowing the addresses of three downstream services. Six months later, those services have moved to a different namespace, the load balancer's IP has changed, and one of the services has been split into two. Service discovery, health checks, and graceful handling of connection failures all exist because topology is not stable.

There is one administrator. In a system with five services, each owned by a different team, you will have conflicting configuration, incompatible deploy-

ment schedules, and API versioning disagreements. The patterns in Chapter 11 (distributed consistency) exist because there is no single authority to enforce correctness across service boundaries.

Transport cost is zero. Serialization and deserialization of HTTP/JSON requests is not free. At 10,000 requests per second, the CPU cost of JSON parsing is measurable. This is why gRPC with protobuf (Chapter 9) is preferred for high-volume internal APIs — binary encoding is significantly cheaper.

The network is homogeneous. Your Go service talks to a Python analytics service, a Java billing system, and a PostgreSQL database, all over TCP. The protocols, serialization formats, and error conventions are all different. The adapter patterns and schema management in this book exist because the network is not homogeneous.

Understanding these fallacies does not make distributed systems easy. It makes their failures predictable. When your service fails in a new and interesting way, one of these eight things is almost always the root cause.

When You Have To

The reasons to distribute are real. They are just rarer than the industry makes them seem.

You need more capacity than one machine can provide. A database that grows beyond the memory and disk of the largest available instance. A video encoding job that would take 48 hours on a single core. A web service that needs to handle more concurrent connections than one process can sustain.

Note that “more capacity” is a measurable constraint. You can observe: CPU utilization above 80%, memory pressure causing swapping, disk I/O saturation, connection pool exhaustion. When you see these, distribution is justified. When you do not see them, it is premature.

You need fault tolerance. One machine will fail eventually. A hard drive will fail (MTBF of spinning disks is ~3 years). A datacenter will lose power. A cloud availability zone will have an incident. If your service runs on one machine, any of these events takes your service down. Two machines in different availability zones means a single failure is survivable. Three means you can take one down for maintenance without downtime.

You need independent deployability. A team of 200 engineers deploying to the same binary on the same schedule is a coordination nightmare. Every deploy requires synchronizing 200 people. Every bug fix requires a full regression test of the entire system. At some organizational scale, the coordination cost of a shared deployment unit becomes worse than the coordination cost of independent services.

This is a people problem, not a technical one. If you have ten engineers, a monolith may be fine. If you have ten teams of ten, independent deployment becomes necessary.

You need different scaling curves. Your image processing service needs 16 CPU cores and 32GB RAM. Your user profile API needs 2 cores and 2GB RAM. In a monolith, every instance runs everything – you are paying for 16 cores and 32GB to serve profile API requests that need none of it. As separate services, you scale each tier independently and pay only for what each component actually needs.

Notice what is not on this list: "microservices are the industry standard." Or "small services are easier to reason about." Or "it is better separation of concerns." These are opinions, not constraints. The four reasons above are measurable. When you can point to a specific metric – CPU saturation, fault tolerance requirement, team size, cost inefficiency – distribution is justified. When you cannot, it is not.

The Hidden Tax

Every distribution boundary introduces costs. Not all of them are visible in benchmarks. Most show up in incident reports.

Latency math

A function call takes around 5 nanoseconds. An HTTP call to a service on the same host takes around 200 microseconds (200,000 nanoseconds). That is a factor of 40,000.

For a request that calls five services in sequence:

```
Request arrives: 0ms
Call auth service:       +2ms
Call user service:       +2ms
Call order service:      +3ms
Call inventory service:  +2ms
Call notification svc:   +1ms
Response: 10ms total overhead
```

Ten milliseconds of irreducible latency — before a single line of business logic — just from service boundaries. For a user-facing API where your target is 50ms end-to-end, that is 20% of your budget consumed by network hops alone.

In a single-process application, those five function calls take 25 nanoseconds. The difference is not an argument against distribution — it is an argument for being deliberate about when you add a boundary and when you parallelize calls (Chapter 10).

Failure taxonomy

In-process failures are simple: a function panics or returns an error. In a distributed system, a call can fail in dozens of distinguishable ways:

- `connection refused` — the service is down or not yet started
- `i/o timeout` — the service is slow or overloaded, not down
- `context deadline exceeded` — your timeout fired before the service responded
- `EOF` — the connection was closed mid-response (partial failure)
- `TLS handshake failure` — a certificate expired
- HTTP 500 — the service handled your request but its own logic failed
- HTTP 503 — the service is temporarily unavailable (retry-able)
- HTTP 429 — you are sending too many requests (rate limited)

- HTTP 400 — your request was wrong (do not retry)
- Correct HTTP 200 with wrong JSON — the service changed its schema

Each of these requires a different response. Chapter 9 maps these failure modes to retry decisions. Chapter 12 shows how to make them visible in production.

Operational surface area

A monolith has one deployment artifact, one configuration, one log stream, one metrics endpoint. A ten-service system has ten of each. The operational surface area — the number of things that can be wrong at any moment — scales with service count.

This does not mean monoliths are better. It means the operational investment needed to run a distributed system is larger. Observability (Chapter 12), structured logging, and distributed tracing are not optional in a multi-service system. They are the minimum viable investment to make the system operable.

The Struct-to-Service Progression

Here is a mental model that will serve you through this book.

A `struct` in Go is a unit of state and behavior. It has fields (data) and methods (operations). It lives in a process.

A service is also a unit of state and behavior. It has a data store (fields) and an API (methods). It lives on a network.

The patterns are the same. The failure modes are different.

```go
type OrderService struct {
    db     *sql.DB
    events EventPublisher
}

func (s *OrderService) CreateOrder(ctx context.Context, order Order) error {
    if err := s.db.SaveOrder(ctx, order); err != nil {
        return fmt.Errorf("save order: %w", err)
    }
    return s.events.Publish(ctx, "order.created", order)
}
```

You are already looking at a distributed service in miniature. `db` is a downstream dependency that can be slow or unavailable. `events` is a message queue that can be full or unreachable. `ctx` carries the deadline that says how long we are willing to wait before giving up.

The only difference between this struct and a deployed service is that the struct's dependencies are in the same process. Once you deploy this as a service, `db` becomes a database over TCP, and `events` becomes a message broker over TCP, and every call to either can fail in ways that never happen inside a process.

When you look at a microservices architecture diagram and see arrows between boxes, you are looking at a distributed version of this struct: boxes are services, arrows are method calls over a network. The patterns – context propagation, retry, circuit breaker, backpressure – exist because method calls over a network fail in ways that in-process calls do not.

This progression – struct to service, in-process to over-network – is the core mental shift this book is trying to make. When you can see a service as a struct with network-shaped failure modes, distributed systems stop feeling foreign.

Go's Fit

Go was designed with networked services in mind. Not by accident.

The language's concurrency primitives map directly onto distributed systems patterns. A goroutine handling one HTTP request is cancellable via context when the client disconnects. A channel connecting a producer and consumer is buffered or unbuffered – the same backpressure decision you make when choosing between a bounded and unbounded queue. A `select` statement choosing between a result and a timeout is the foundation for every circuit breaker and retry pattern in this book.

These are not metaphors. The worker pool pattern from Chapter 2 is the same code as the Kafka consumer group from Chapter 8. The channel used to coordinate goroutines has the same semantics as the message queue used to coordinate services. The mental model transfers directly.

The standard library is production-ready for service development. `net/http` (Chapter 4) handles HTTP without a framework. `database/sql` handles connection pooling. `context.Context` propagates cancellation and deadlines from the HTTP handler through every downstream call. The type system makes interfaces the natural contract between components, which makes them the natural contract between services.

Go compiles to a static binary with no runtime dependencies. Deployment is copying a file. For a distributed system where you deploy ten services instead of one, this operational simplicity compounds: no JVM to configure, no Python

venv to manage, no node_modules to install. A Docker image for a Go service is typically under 20MB. Under 5MB with scratch or distroless base images.

The Monolith-First Principle

The right default is: do not distribute until you have to.

A well-structured monolith is faster to build, easier to test, and simpler to operate than an equivalent distributed system. It has none of the fallacies, none of the latency tax, and none of the operational surface area.

The strangler fig pattern — gradually extracting services from a working monolith when specific constraints force it — is more reliable than designing microservices from scratch. You will not know which service boundaries are natural until you have shipped features and seen where the seams actually are. Premature service decomposition creates wrong boundaries, and wrong boundaries are expensive to undo.

Signs that a specific piece is ready to extract:

- It has a radically different scaling requirement from the rest of the system
- It needs to be deployed on its own schedule by its own team
- It uses a different technology stack (GPU processing, specialized hardware)
- Its failure should be isolated from the rest of the system

Signs that extraction is premature:

- You want "cleaner code" (that is a refactoring problem, not a distribution problem)
- The team is small (fewer than ~20 engineers)
- You do not yet have distributed tracing and structured logging in place

The patterns in this book apply equally to the monolith and to the distributed system. Context propagation, graceful shutdown, structured logging, circuit breakers — these are good engineering practice in any Go program. Start with them in your monolith. When you extract a service, you will have built the right habits.

What You Will Build

This book builds a distributed system piece by piece. Not a toy. A real system with authentication, a database, a message queue, a service registry, and a capstone that puts everything together.

Every chapter adds one piece and explains how it connects to the rest. By the end, you will have built:

- An HTTP gateway with composable middleware (Chapter 4)
- JWT-based authentication with ECDSA signing (Chapter 5)
- A PostgreSQL-backed data service with zero-downtime migrations (Chapter 6)
- A NATS producer-consumer pair, then Kafka at depth (Chapters 3, 8)
- Service-to-service calls with gRPC, protobuf, retries, and circuit breakers (Chapter 9)

- Parallel fan-out with errgroup (Chapter 10)
- Distributed consistency with sagas and idempotency (Chapter 11)
- Observability: slog + Prometheus + OpenTelemetry (Chapter 12)
- Tests for failure scenarios with testcontainers and chaos patterns (Chapter 13)
- Configuration and secrets management (Chapter 14)
- A complete capstone system: six services, one `docker-compose up` (Chapter 15)

You will understand not just how to build each piece, but why it is built that way — what breaks if you do it differently, and when the simple version is actually the right call.

Q&A

Q: Should I start with a monolith and split it later, or design for distribution from the start?

Start with a monolith. The strangler fig pattern — gradually extracting services when constraints force it — is more reliable than upfront microservice design. The boundaries that feel natural at design time are usually wrong. You will not know the real seams until you have built features and observed where the complexity actually lives.

Q: How many services is too many?

When the cognitive load of understanding the system exceeds the team's capacity to hold it in their heads. A proxy metric: if a new engineer cannot understand the system's critical paths in their first week, you may have too many

services. This is a team-size and documentation problem masquerading as an architecture problem.

Q: Is Go the right language for distributed systems?

Go is a strong choice. The concurrency model (goroutines, channels, context) aligns directly with distributed systems patterns. The deployment model (static binary, small images) simplifies operations at scale. The standard library covers most service infrastructure without external dependencies. The main alternative for systems requiring predictable sub-millisecond tail latency is Rust — Go's garbage collector introduces periodic pauses that Rust does not have. For most distributed services, Go's characteristics are a better tradeoff between developer productivity and runtime performance.

Q: What's the difference between distributed systems and microservices?

Microservices is one architecture pattern for distributed systems — specifically, a style where services are small, independently deployable, and owned by a single team. Distributed systems is a broader category: it includes client-server architectures, distributed databases, consensus protocols (Raft, Paxos), peer-to-peer systems, and streaming platforms. This book teaches distributed systems patterns; microservices are the most common production context where you will use them.

Q: If distributed systems are more complex, why does the whole industry seem to use them?

They do not, uniformly. Most successful businesses run on architectures far simpler than the industry press suggests. The companies writing about their

microservices architectures are systematically biased — they are large enough and engineering-heavy enough to have solved the operational complexity. The companies that built a monolith and grew to $100M ARR on it are not at conferences talking about their architecture. When you see a conference talk about distributed systems at scale, weight it by the speaker's team size and engineering investment.

Before we build anything distributed, we need to understand the concurrency model that makes Go uniquely suited to it. The goroutines, channels, and context primitives in the next chapter are not just language features — they are a mental model for reasoning about services. Chapter 2 builds that model from the ground up.

Chapter 2: Concurrency as Distributed Systems in Miniature

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/distributed-systems-in-go.

Goroutines

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/distributed-systems-in-go.

The lifecycle problem

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/distributed-systems-in-go.

Channels

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/distributed-systems-in-go.

Directional channels

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/distributed-systems-in-go.

Buffered vs unbuffered

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/distributed-systems-in-go.

Closing channels

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/distributed-systems-in-go.

select

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/distributed-systems-in-go.

Timeout with select

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/distributed-systems-in-go.

Non-blocking operations

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/distributed-systems-in-go.

sync.WaitGroup

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/distributed-systems-in-go.

context.Context

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/distributed-systems-in-go.

The cancellation tree

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/distributed-systems-in-go.

WithCancel

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/distributed-systems-in-go.

WithTimeout and WithDeadline

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/distributed-systems-in-go.

Checking cancellation

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/distributed-systems-in-go.

Context values

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/distributed-systems-in-go.

Putting It Together: A Worker Pool

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/distributed-systems-in-go.

Graceful Shutdown

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/distributed-systems-in-go.

The Concurrency Mental Model

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/distributed-systems-in-go.

Q&A

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/distributed-systems-in-go.

What Goes in the SDK

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/distributed-systems-in-go.

Chapter 3: Two Services, One Queue

Imagine a courier company. You drop a parcel at the counter. The courier picks it up and delivers it. You do not wait at the counter for delivery to complete. The courier does not wait for you to be home before accepting the parcel. The two operations – dropping off and delivering – are decoupled in time. The post office is the intermediary.

A message queue works the same way. One service produces a message. Another service consumes it. Neither needs to know the other exists. Neither needs to be running at the same moment. The queue is the intermediary.

This is the core pattern of asynchronous distributed systems. In this chapter you will build it: two Go programs – a producer and a consumer – that communicate through a message queue. Then you will kill one of them and watch the other keep running. That moment is the first win.

Why a Queue and Not an HTTP Call?

Before we write code, it is worth being clear on when to use a queue versus a direct HTTP call between services.

Use HTTP when: you need a response immediately, you want to know if the operation succeeded before moving on, or the downstream service must process

the request synchronously as part of the caller's workflow. An order service checking whether a product is in stock before confirming an order is an HTTP call – the answer must come back before the response goes out.

Use a queue when: the work can happen later, failure in the downstream service should not fail the upstream service, you need to process the same event in multiple places, or you need to smooth out bursts of load. An order service notifying a fulfilment service, an email service, and an analytics pipeline about a new order is a queue – the order does not depend on any of those notifications completing.

The wrong choice here creates real problems. Direct HTTP coupling between services means a slow consumer brings down the producer. A queue decouples them: the producer keeps working even if the consumer is down, the messages accumulate, and the consumer catches up when it recovers.

NATS for a First Example

We will use NATS for this chapter. Kafka is the right tool for durable, high-throughput event streams – and we will use it in Chapter 8. But Kafka requires a broker cluster and Zookeeper or KRaft for a first example, which is more infrastructure than this chapter needs.

NATS is a lightweight, high-performance messaging system. A single NATS server binary, no external dependencies. For local development: `docker run -p 4222:4222 nats:latest`. For production: NATS JetStream adds persistence and consumer acknowledgements.

We will use NATS Core (no persistence) here because the architecture is clearer without durability. Understanding the non-persistent case first makes the persistent case (Chapter 8) more meaningful.

The Interfaces

Before we touch NATS, define the interfaces. This is a habit worth building: write the interface before the implementation. The interface is the design; the implementation is the detail.

```go
// producer/publisher.go
package publisher

import "context"

type Publisher interface {
    Publish(ctx context.Context, subject string, data []byte) error
    Close() error
}
```

```go
// consumer/subscriber.go
package subscriber

import "context"

type Message struct {
    Subject string
    Data    []byte
}

type Subscriber interface {
    Subscribe(ctx context.Context, subject string) (<-chan Message, error)
    Close() error
}
```

These two interfaces are the contract between the application and the messaging layer. The application calls `Publish` to send. It receives from the channel returned by `Subscribe`. It never touches NATS directly.

This matters for testing: you can write a mock publisher and subscriber that work entirely in-process, without a running NATS server. We will use this in Chapter 13.

The Producer

The producer is a service that generates events. In our example it generates order events — a simple stand-in for any real domain event.

```
// cmd/producer/main.go
package main

import (
    "context"
    "encoding/json"
    "log/slog"
    "os"
    "os/signal"
    "syscall"
    "time"

    "github.com/nats-io/nats.go"
)

type OrderEvent struct {
    ID        string    `json:"id"`
    ProductID string    `json:"product_id"`
    Quantity  int       `json:"quantity"`
    CreatedAt time.Time `json:"created_at"`
}

func main() {
    logger := slog.New(slog.NewJSONHandler(os.Stdout, nil))

    natsURL := os.Getenv("NATS_URL")
    if natsURL == "" {
        natsURL = nats.DefaultURL
    }

    nc, err := nats.Connect(natsURL)
    if err != nil {
        logger.Error("failed to connect to NATS", "error", err)
        os.Exit(1)
    }
    defer nc.Close()
```

```
37
38      logger.Info("producer started", "nats_url", natsURL)
39
40      ctx, stop := signal.NotifyContext(context.Background(),
41           os.Interrupt,
42           syscall.SIGTERM)
43      defer stop()
44
45      ticker := time.NewTicker(2 * time.Second)
46      defer ticker.Stop()
47
48      orderNum := 1
49      for {
50          select {
51          case <-ctx.Done():
52              logger.Info("producer shutting down")
53              return
54          case <-ticker.C:
55              event := OrderEvent{
56                  ID:        fmt.Sprintf("order-%04d", orderNum),
57                  ProductID: "prod-001",
58                  Quantity:  orderNum % 5 + 1,
59                  CreatedAt: time.Now().UTC(),
60              }
61
62              data, err := json.Marshal(event)
63              if err != nil {
64                  logger.Error("failed to marshal event", "error", err)
65                  continue
66              }
67
68              if err := nc.Publish("orders.created", data); err != nil {
69                  logger.Error("failed to publish", "error", err)
70                  continue
71              }
72
73              logger.Info("published order",
74                   "order_id",
75                   event.ID,
76                   "quantity",
77                   event.Quantity)
78              orderNum++
79          }
80      }
81  }
```

Walk through the structure:

- `log/slog` with JSON output — structured logging from the start. Every field is key-value, not a formatted string. This matters when you are searching logs across multiple services.

- NATS URL from environment — no hardcoded addresses. The service does not know whether it is running locally or in a cluster.
- `signal.NotifyContext` — graceful shutdown. The producer publishes until interrupted, then exits cleanly.
- `ticker` + `select` — the production loop. Every two seconds, publish one event. The `ctx.Done()` case in the same `select` ensures the loop stops immediately when the signal arrives, without waiting for the next tick.

The Consumer

The consumer receives events and processes them. In a real system it might write to a database, call another service, or trigger a workflow. Here it logs what it receives.

```
// cmd/consumer/main.go
package main

import (
    "context"
    "encoding/json"
    "log/slog"
    "os"
    "os/signal"
    "syscall"
    "time"

    "github.com/nats-io/nats.go"
)

type OrderEvent struct {
    ID        string    `json:"id"`
    ProductID string    `json:"product_id"`
    Quantity  int       `json:"quantity"`
    CreatedAt time.Time `json:"created_at"`
}

func main() {
    logger := slog.New(slog.NewJSONHandler(os.Stdout, nil))

```

```
    natsURL := os.Getenv("NATS_URL")
    if natsURL == "" {
        natsURL = nats.DefaultURL
    }

    nc, err := nats.Connect(natsURL)
    if err != nil {
        logger.Error("failed to connect to NATS", "error", err)
        os.Exit(1)
    }
    defer nc.Close()

    ctx, stop := signal.NotifyContext(context.Background(),
         os.Interrupt,
         syscall.SIGTERM)
    defer stop()

    msgs := make(chan *nats.Msg, 64)

    sub, err := nc.ChanSubscribe("orders.created", msgs)
    if err != nil {
        logger.Error("failed to subscribe", "error", err)
        os.Exit(1)
    }
    defer sub.Unsubscribe()

    logger.Info("consumer started, waiting for orders")

    for {
        select {
        case <-ctx.Done():
            logger.Info("consumer shutting down")
            return
        case msg, ok := <-msgs:
            if !ok {
                return
            }

            var event OrderEvent
            if err := json.Unmarshal(msg.Data, &event); err != nil {
                logger.Error("failed to unmarshal event", "error", err)
                continue
            }

            logger.Info("received order",
                "order_id", event.ID,
                "product_id", event.ProductID,
                "quantity", event.Quantity,
                "age_ms", time.Since(event.CreatedAt).Milliseconds(),
            )
        }
    }
}
```

The consumer structure mirrors the producer:

- Same graceful shutdown pattern via `signal.NotifyContext`.
- `ChanSubscribe` returns a channel — NATS delivers messages by sending to it. The consumer's `select` reads from this channel alongside the cancellation signal.
- `age_ms` in the log: how many milliseconds old is this event by the time we process it? This is a useful metric in production — if the consumer falls behind, this number grows.

Running It

Create a `go.mod`:

```
module github.com/yourname/orders

go 1.22

require github.com/nats-io/nats.go v1.36.0
```

Start NATS:

```
docker run -p 4222:4222 --name nats nats:latest
```

In one terminal, start the consumer:

```
go run ./cmd/consumer
```

In another terminal, start the producer:

```
go run ./cmd/producer
```

You will see the producer publishing every two seconds:

```
{"time":"...","level":"INFO","msg":"published order","order_id":"order-0001","quantity":2}
{"time":"...","level":"INFO","msg":"published order","order_id":"order-0002","quantity":3}
```

And the consumer receiving them:

```
{"time":"...","level":"INFO","msg":"received
↪  order","order_id":"order-0001","product_id":"prod-001","quantity":2,"age_ms":1}
```

Now stop the consumer with Ctrl-C. The producer keeps running. Messages are being published — but there is nobody to receive them. In NATS Core (no persistence), those messages are dropped. We will address this in Chapter 8 with Kafka's durable log.

Restart the consumer. It starts receiving again from the current moment — it does not see the messages that were published while it was down.

Now stop the producer. The consumer keeps running, waiting. Start the producer again. The consumer picks up immediately. The two services have no awareness of each other's lifecycle. They communicate through the queue, and the queue is the only thing they depend on in common.

Loose Coupling in Practice

What you just observed has a name: loose coupling. The producer does not import the consumer's package. The consumer does not import the producer's. They share nothing except:

1. A running NATS server
2. The subject name `"orders.created"`
3. The shape of the `OrderEvent` struct (the schema)

This is the minimum coupling necessary for two services to communicate. Any additional coupling — a shared database, a direct HTTP dependency, a shared configuration file — is a coupling you can remove.

The subject name and the event schema are the explicit contract. They deserve the same attention you would give a public API. When you need to change the schema, version it: `"orders.created.v2"`, not `"orders.created"`. Consumers of the old subject continue working until you migrate them. This is the event versioning problem, and it is a real one — we will return to it in Chapter 11.

Error Handling and Reconnection

Production services lose connections. NATS servers restart. Networks partition. What happens to your producer and consumer when the NATS connection drops?

The `nats.go` library handles reconnection automatically. By default, it attempts to reconnect indefinitely, with configurable backoff. You can configure this at connection time:

```
nc, err := nats.Connect(natsURL,
    nats.MaxReconnects(-1),
    nats.ReconnectWait(2*time.Second),
    nats.DisconnectErrHandler(func(nc *nats.Conn, err error) {
        logger.Warn("disconnected from NATS", "error", err)
    }),
    nats.ReconnectHandler(func(nc *nats.Conn) {
        logger.Info("reconnected to NATS", "url", nc.ConnectedUrl())
    }),
)
```

The disconnect handler and reconnect handler give you visibility into connection events — critical for debugging production incidents where services appear healthy but messages stop flowing.

One thing NATS Core does not give you during a disconnect: the messages published while you were disconnected. They are gone. If your system cannot tolerate message loss, use NATS JetStream (persistent) or Kafka (Chapter 8). For notifications, analytics, and fire-and-forget events, NATS Core's at-most-once delivery is acceptable.

The Complete Picture

Two services, one queue. Here is what makes this architecture powerful:

Independent deployment. You can deploy a new version of the consumer without touching the producer. The producer publishes to the same subject. The consumer reads from it. No coordination required.

Independent scaling. If the consumer is processing orders slower than the producer creates them, you start a second consumer. Both connect to the same NATS subject. NATS delivers each message to one of them (queue group

subscription). You have doubled throughput without changing either service's code.

Independent failure. The producer crashes. The consumer waits for more messages. The producer restarts. The consumer continues. Neither failure is the other's problem.

These three properties — independent deployment, scaling, and failure — are what asynchronous messaging buys you. They come at a cost: you can no longer reason about your system's state at a single point in time. Is the consumer up to date? Has it processed the last message? You do not know without additional instrumentation. That tradeoff is real, and it is part of what Chapter 11 (distributed consistency) is about.

For now: you have built two services that talk. Kill one. The other keeps running. That is the win.

What Goes in the SDK

The Publisher and Subscriber interfaces from this chapter are the contract that every messaging implementation in the system must satisfy. Extract them into `platform/messaging`:

```
platform/messaging/
  interfaces.go   — Publisher, Subscriber, Message types
  nats/
    publisher.go  — NATS publisher implementing Publisher
    subscriber.go — NATS subscriber implementing Subscriber
```

The interfaces are the key export:

```go
// platform/messaging/interfaces.go
package messaging

import "context"

type Publisher interface {
    Publish(ctx context.Context, subject string, data []byte) error
    Close() error
}

type Subscriber interface {
    Subscribe(ctx context.Context, subject string) (<-chan Message, error)
    Close() error
}

type Message struct {
    Subject string
    Data    []byte
    Reply   string
    Ack     func() error // no-op in NATS Core; commits offset in Kafka
}
```

Service code depends on these interfaces, never on `nats.Conn` directly. A handler that publishes events receives a `Publisher`:

```
func createOrderHandler(store OrderStore
     pub messaging.Publisher) http.HandlerFunc {
    return func(w http.ResponseWriter, r *http.Request) {
        // ... create order ...
        event, _ := json.Marshal(OrderCreated{ID: order.ID})
        pub.Publish(r.Context(), "orders.created", event)
        // ...
    }
}
```

When we introduce Kafka in Chapter 8, the Kafka producer implements the same `Publisher` interface. The handler code does not change. Tests use a `FakePublisher` that records published messages in memory – no NATS or Kafka required.

This interface-first approach is the single most important design decision in the platform SDK. It decouples business logic from infrastructure choices.

Q: What is a "subject" in NATS?

A subject is a named destination for messages, similar to a topic in Kafka or an exchange in RabbitMQ. Subjects support wildcards: `orders.*` matches `orders.created` and `orders.cancelled`. `orders.>` matches any number of tokens. This makes subject hierarchies a natural way to organize events.

Q: What is a queue group?

A queue group is a group of subscribers that share the same queue group name. NATS delivers each message to exactly one subscriber in the group. This is the pattern for horizontal scaling of consumers: all instances subscribe with the same queue group name, and each message goes to one instance.

```
sub, err := nc.QueueSubscribe("orders.created", "fulfilment-workers", handler)
```

Q: Should the producer and consumer share the `OrderEvent` struct?

In a single repository (monorepo), yes — a shared `events` package containing event types is practical. In separate repositories, no — each service defines its own struct and validates the incoming data independently. Shared types create compile-time coupling that defeats the purpose of separate deployments. JSON schemas or Protocol Buffers are the right way to define cross-service contracts without source-level coupling.

Q: What happens if the consumer processes a message and then crashes before finishing?

In NATS Core, the message is gone. The consumer processes it (or starts to), crashes, and restarts cold with no memory of the in-flight message. If your processing must be exactly-once, you need either NATS JetStream's acknowledgement model or Kafka's offset management — both covered in Chapter 8. Design your consumers to be idempotent where possible: processing the same message twice should have the same effect as processing it once.

Q: How does this relate to the goroutine model from Chapter 2?

Directly. The NATS channel (`msgs`) in the consumer is exactly the same as the buffered channel in the worker pool from Chapter 2. The `select` loop is the same. The graceful shutdown via `ctx.Done()` is the same. The only difference is that the messages come from a network socket instead of an in-process goroutine. The mental model is identical.

A message queue decouples services in time. But most user-facing services need synchronous HTTP responses. The next chapter builds the HTTP layer — not with a framework, but from `net/http` and composable middleware, the same pattern that underlies every Go HTTP framework.

Chapter 4: HTTP Services and Middleware Chains

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/distributed-systems-in-go.

net/http in Go 1.22

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/distributed-systems-in-go.

The Handler Interface

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/distributed-systems-in-go.

Building a Middleware Chain

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/distributed-systems-in-go.

The Essential Middleware

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/distributed-systems-in-go.

Recovery

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/distributed-systems-in-go.

Request ID

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/distributed-systems-in-go.

Logging

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/distributed-systems-in-go.

Body Close

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/distributed-systems-in-go.

Request Timeout

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/distributed-systems-in-go.

CORS

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/distributed-systems-in-go.

Structured Error Responses

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/distributed-systems-in-go.

JSON Request Binding

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/distributed-systems-in-go.

Health Check Patterns

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/distributed-systems-in-go.

The Complete Service

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/distributed-systems-in-go.

Testing HTTP Handlers

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/distributed-systems-in-go.

What Frameworks Add

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/distributed-systems-in-go.

Q&A

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/distributed-systems-in-go.

What Goes in the SDK

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/distributed-systems-in-go.

Chapter 5: Authentication and JWT Across Services

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/distributed-systems-in-go.

What a JWT Is

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/distributed-systems-in-go.

HMAC vs RSA vs ECDSA

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/distributed-systems-in-go.

Generating and Validating Tokens with ECDSA

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/distributed-systems-in-go.

The JWKS Endpoint

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/distributed-systems-in-go.

Auth Middleware

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/distributed-systems-in-go.

Scope-Based Authorization

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/distributed-systems-in-go.

The Auth Service

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/distributed-systems-in-go.

Pluggable Authentication Agents

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/distributed-systems-in-go.

Token Propagation Between Services

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/distributed-systems-in-go.

Adding Auth to the Service Chain

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/distributed-systems-in-go.

Testing Auth

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/distributed-systems-in-go.

Q&A

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/distributed-systems-in-go.

What Goes in the SDK

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/distributed-systems-in-go.

Chapter 6: Database Access and Migrations

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/distributed-systems-in-go.

pgx: PostgreSQL Done Right

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/distributed-systems-in-go.

Connection pool sizing

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/distributed-systems-in-go.

PgBouncer for connection pooling at scale

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/distributed-systems-in-go.

The Repository Pattern

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/distributed-systems-in-go.

Transactions

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/distributed-systems-in-go.

Savepoints

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/distributed-systems-in-go.

sqlc: Type-Safe SQL Without an ORM

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/distributed-systems-in-go.

Schema Migrations

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/distributed-systems-in-go.

Zero-Downtime Migrations

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/distributed-systems-in-go.

One Database Per Service

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/distributed-systems-in-go.

Observability for Database Operations

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/distributed-systems-in-go.

Q&A

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/distributed-systems-in-go.

What Goes in the SDK

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/distributed-systems-in-go.

Chapter 7: Context Propagation — The Thread Connecting Everything

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/distributed-systems-in-go.

How the Context Tree Fails

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/distributed-systems-in-go.

The Context Discipline

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/distributed-systems-in-go.

Deadline Propagation

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/distributed-systems-in-go.

Narrowing Deadlines

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/distributed-systems-in-go.

Context Values in Practice

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/distributed-systems-in-go.

The Context Anti-Patterns

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/distributed-systems-in-go.

Context across goroutine trees

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/distributed-systems-in-go.

Detaching from the parent context

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/distributed-systems-in-go.

Connecting the Layers

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/distributed-systems-in-go.

Q&A

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/distributed-systems-in-go.

What Goes in the SDK

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/distributed-systems-in-go.

Chapter 8: Message Queues in Depth — Kafka, Backpressure, and Consumer Groups

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/distributed-systems-in-go.

How Kafka Works

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/distributed-systems-in-go.

Message keys and partition assignment

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/distributed-systems-in-go.

Writing a Kafka Producer

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/distributed-systems-in-go.

Writing a Kafka Consumer

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/distributed-systems-in-go.

Consumer rebalancing

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/distributed-systems-in-go.

Backpressure

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/distributed-systems-in-go.

Scale out consumers

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/distributed-systems-in-go.

Process in batches

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/distributed-systems-in-go.

Dead-letter queues

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/distributed-systems-in-go.

Consumer Groups in Action

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/distributed-systems-in-go.

Dynamic Topic Subscriptions

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/distributed-systems-in-go.

Message Schema Management

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/distributed-systems-in-go.

Q&A

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/distributed-systems-in-go.

What Goes in the SDK

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/distributed-systems-in-go.

Chapter 9: Service-to-Service Calls — gRPC, Protobuf, Retries, and Circuit Breakers

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/distributed-systems-in-go.

Why gRPC for Internal Communication

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/distributed-systems-in-go.

Defining a Service with Protocol Buffers

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/distributed-systems-in-go.

Generating Go Code with buf

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/distributed-systems-in-go.

Implementing the gRPC Server

Running the gRPC Server

The gRPC Client with Interceptors

Service Token Authentication

The Dual API Pattern: Management vs Consumption

Timeouts and Retries

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/distributed-systems-in-go.

HTTP timeouts

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/distributed-systems-in-go.

Retries with exponential backoff

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/distributed-systems-in-go.

The Circuit Breaker

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/distributed-systems-in-go.

Choosing: HTTP vs gRPC

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/distributed-systems-in-go.

Q&A

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/distributed-systems-in-go.

What Goes in the SDK

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/distributed-systems-in-go.

Chapter 10: Orchestration — Parallel Flows and Fan-Out

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/distributed-systems-in-go.

The Sequential Problem

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/distributed-systems-in-go.

Fan-Out with errgroup

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/distributed-systems-in-go.

Concurrency safety

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/distributed-systems-in-go.

SetLimit: capping concurrency

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/distributed-systems-in-go.

Partial Failure Handling

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/distributed-systems-in-go.

Pipeline Composition

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/distributed-systems-in-go.

Fan-Out to Dynamic Targets

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/distributed-systems-in-go.

Collecting results concurrently

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/distributed-systems-in-go.

Timeout Budget Across a Multi-Step Flow

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/distributed-systems-in-go.

Measuring the Impact

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/distributed-systems-in-go.

What Goes in the SDK

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/distributed-systems-in-go.

Q&A

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/distributed-systems-in-go.

Chapter 11: Distributed Consistency in Practice

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/distributed-systems-in-go.

The CAP Theorem, Applied

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/distributed-systems-in-go.

Eventual Consistency

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/distributed-systems-in-go.

Idempotency Keys

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/distributed-systems-in-go.

Optimistic Locking

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/distributed-systems-in-go.

The Saga Pattern

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/distributed-systems-in-go.

Event-Driven Consistency

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/distributed-systems-in-go.

Detecting Inconsistency

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/distributed-systems-in-go.

The Outbox Pattern

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/distributed-systems-in-go.

Reconciliation in Practice

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/distributed-systems-in-go.

What Goes in the SDK

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/distributed-systems-in-go.

Q&A

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/distributed-systems-in-go.

Chapter 12: Observability — Logging, Metrics, and Tracing

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/distributed-systems-in-go.

Structured Logging with slog

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/distributed-systems-in-go.

Setting up slog

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/distributed-systems-in-go.

Log levels and when to use them

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/distributed-systems-in-go.

Context-aware logging

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/distributed-systems-in-go.

Multi-transport log aggregation

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/distributed-systems-in-go.

Metrics with Prometheus

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/distributed-systems-in-go.

The three metric types

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/distributed-systems-in-go.

Recording metrics in middleware

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/distributed-systems-in-go.

Business metrics alongside technical metrics

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/distributed-systems-in-go.

Distributed Tracing with OpenTelemetry

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/distributed-systems-in-go.

Setting up the trace provider

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/distributed-systems-in-go.

Automatic HTTP instrumentation

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/distributed-systems-in-go.

Manual spans for database and queue operations

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/distributed-systems-in-go.

Connecting the three pillars

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/distributed-systems-in-go.

Alerting: When Metrics Become Alarms

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/distributed-systems-in-go.

Q&A

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/distributed-systems-in-go.

What Goes in the SDK

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/distributed-systems-in-go.

Chapter 13: Testing Distributed Systems

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/distributed-systems-in-go.

The Testing Pyramid for Distributed Systems

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/distributed-systems-in-go.

Testing with Real Databases: testcontainers

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/distributed-systems-in-go.

Sharing containers across tests

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/distributed-systems-in-go.

The -race Flag

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/distributed-systems-in-go.

Testing with Fake Implementations

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/distributed-systems-in-go.

Testing Idempotency

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/distributed-systems-in-go.

Chaos Testing: Kill a Dependency Mid-Test

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/distributed-systems-in-go.

goleak: Finding Goroutine Leaks

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/distributed-systems-in-go.

Table-Driven Tests for Edge Cases

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/distributed-systems-in-go.

Q&A

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/distributed-systems-in-go.

What Goes in the SDK

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/distributed-systems-in-go.

Chapter 14: Configuration and Secrets

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/distributed-systems-in-go.

The Twelve-Factor Principle

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/distributed-systems-in-go.

Configuration With Struct Tags

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/distributed-systems-in-go.

Viper for Complex Configuration

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/distributed-systems-in-go.

Configuration Validation at Startup

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/distributed-systems-in-go.

The startup sequence

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/distributed-systems-in-go.

Secrets Management

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/distributed-systems-in-go.

Conditional Secret Exposure

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/distributed-systems-in-go.

Live Reload via Signal Handler

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/distributed-systems-in-go.

Configuration in a Multi-Service System

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/distributed-systems-in-go.

Configuration Testing

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/distributed-systems-in-go.

Q&A

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/distributed-systems-in-go.

What Goes in the SDK

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/distributed-systems-in-go.

Chapter 15: Capstone — One Complete System

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/distributed-systems-in-go.

The Design

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/distributed-systems-in-go.

The Data Flows

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/distributed-systems-in-go.

The Repository Structure

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/distributed-systems-in-go.

Implementing the Core: The Order Service

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/distributed-systems-in-go.

The docker-compose

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/distributed-systems-in-go.

The API Gateway

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/distributed-systems-in-go.

Service Discovery

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/distributed-systems-in-go.

Two-Phase Service Startup

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/distributed-systems-in-go.

Deploying to Kubernetes

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/distributed-systems-in-go.

The cloud-native contract

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/distributed-systems-in-go.

Tracing a Request

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/distributed-systems-in-go.

Running It

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/distributed-systems-in-go.

The Integration Test

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/distributed-systems-in-go.

What You Have Built

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/distributed-systems-in-go.

Production Hardening Checklist

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/distributed-systems-in-go.

Q&A

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/distributed-systems-in-go.

Appendix A: Go Concurrency Quick Reference

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/distributed-systems-in-go.

Goroutines

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/distributed-systems-in-go.

Channels

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/distributed-systems-in-go.

select

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/distributed-systems-in-go.

context.Context

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/distributed-systems-in-go.

sync.WaitGroup

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/distributed-systems-in-go.

sync.Mutex / sync.RWMutex

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/distributed-systems-in-go.

sync/atomic

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/distributed-systems-in-go.

errgroup

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/distributed-systems-in-go.

signal.NotifyContext (graceful shutdown)

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/distributed-systems-in-go.

Common Patterns

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/distributed-systems-in-go.

Appendix B: File and Object Storage Abstraction

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/distributed-systems-in-go.

The Storage Interface

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/distributed-systems-in-go.

The S3 Adapter

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/distributed-systems-in-go.

The Local Filesystem Adapter

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/distributed-systems-in-go.

Wiring It Up

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/distributed-systems-in-go.

The Platform SDK Package

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/distributed-systems-in-go.

When to Reach for Afero

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/distributed-systems-in-go.

Appendix C: CLI Tooling for Operations

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/distributed-systems-in-go.

The CLI Structure

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/distributed-systems-in-go.

The Migration Runner

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/distributed-systems-in-go.

The Health Checker

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/distributed-systems-in-go.

Admin Commands

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/distributed-systems-in-go.

Reconciliation

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/distributed-systems-in-go.

Testing CLI Commands

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/distributed-systems-in-go.

What Goes in the SDK

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/distributed-systems-in-go.

Beyond CLI: Operational TUIs

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/distributed-systems-in-go.

Glossary

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/distributed-systems-in-go.

Acknowledgements

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/distributed-systems-in-go.

Credits

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/distributed-systems-in-go.

Further Reading

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/distributed-systems-in-go.