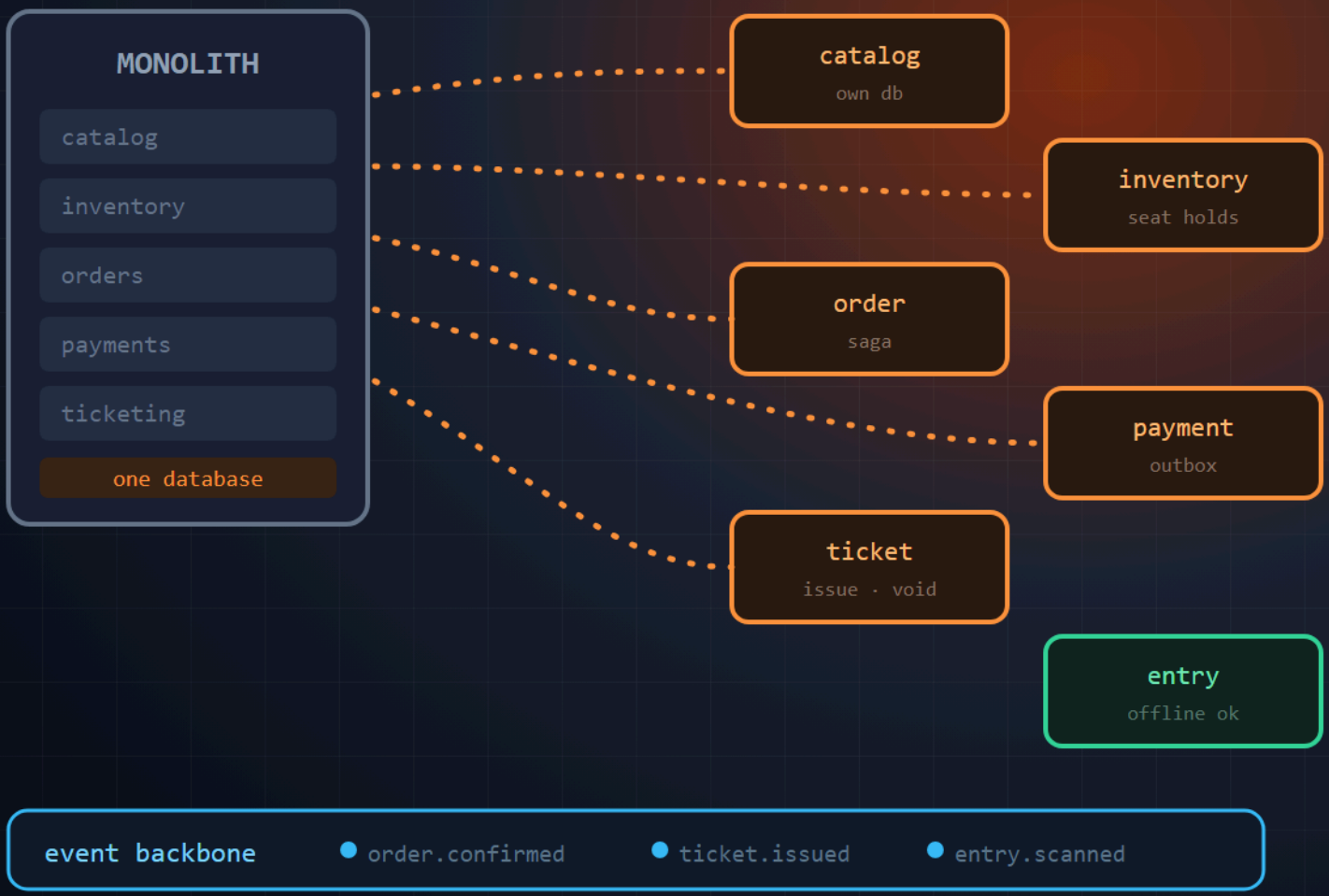


SPLIT · ROUTE · BACKFILL · CUT OVER · DELETE THE OLD PATH

PATTERNS · PITFALLS · MIGRATION STRATEGIES

Microservices in Practice

Decompose a real system without breaking it

28 chapters · 415 pages · one monolith taken apart service by service, from first split to peak traffic

BOUNDARIES · SAGAS · OUTBOX · STRANGLER FIG · SLOs · COST

THENEXBOOKS

Microservices in Practice

Patterns, Pitfalls, and Migration Strategies
Decomposing a real system without breaking it

First Edition

THENEXBOOKS

Microservices in Practice
Patterns, Pitfalls, and Migration Strategies

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. Distributed systems fail in ways no book can fully anticipate; test every pattern against your own workload before depending on it.

This book contains no benchmarks, latency figures, cost savings, or adoption statistics, because such numbers depend entirely on workload, hardware, and configuration, and are routinely quoted out of context. Where a number appears, it is either a budget chosen for the fictional system used as an example, or a blank for you to fill in from your own measurements.

Java, Kubernetes, Kafka, PostgreSQL, Redis, Docker, Envoy, OpenTelemetry, and other product and project names mentioned in this book are trademarks of their respective owners. Their use is for identification only and does not imply endorsement. This book is independent and is not affiliated with, sponsored by, or endorsed by any of these projects, foundations, or companies.

Gatecrest, the running example in this book, is fictional, as are its venues, promoters, payment provider, and every other organization named in its examples. Any resemblance to real companies, platforms, or events is coincidental.

Preface

Most books about microservices show you a finished architecture. Boxes with clean arrows, each service owning its data, events flowing politely between them. It looks obvious once drawn. What the diagram never shows is the eighteen months of work between the system you have and the system in the picture — or the possibility that you should not have drawn the picture at all.

This book is about that gap. It treats microservices as a trade rather than a destination: you exchange the simplicity of one deployable unit and one transaction for independent deployment, fault isolation, and team autonomy, and you pay for that exchange in network failures, split data, operational surface area, and cognitive load. Sometimes the trade is clearly worth it. Often it is worth it for part of a system and not the rest. Occasionally the honest answer is that a modular monolith would give you almost everything you want for a fraction of the cost, and the second chapter is written to help you reach that conclusion if it is the right one.

What follows is the whole arc: deciding, designing boundaries, communicating between services, owning data, migrating a live system piece by piece, and then operating the result — because a microservice architecture you cannot deploy, observe, and debug is strictly worse than the monolith it replaced.

Who This Book Is For

- **Backend engineers** building or maintaining services who want to understand the patterns rather than copy them.
- **Tech leads and architects** responsible for a decomposition, or for repairing one that stalled.
- **Engineers on a team that already split** and is now living with the consequences.
- **Platform and SRE engineers** who have to make many services observable, deployable, and survivable.
- **Engineering managers** who must sequence a migration and explain its cost honestly.

You should know one backend language and relational databases. No distributed-systems theory is assumed — the book teaches the mechanisms it depends on.

What You Will Learn

- What the architecture genuinely buys and costs, and a decision framework for whether to adopt it at all.
- How to find service boundaries that hold, and how to recognize the ones that will not.
- Synchronous contracts, versioning without flag days, and event-driven design that does not become disguised RPC.
- Data ownership, sagas, the transactional outbox, idempotency, and why exactly-once is an illusion.

- CQRS, projections, and answering queries whose data now lives in six services.
- A complete migration playbook: strangler fig, first extraction, database split, dual running, cutover, and sequencing at program scale.
- Resilience, observability and SLOs, deployment, configuration, testing, and on-call practice.
- Service identity and zero trust, latency budgets and backpressure, and the real operational cost.
- Team topologies, an anti-pattern catalog, and how to merge services back when they never earned their separation.

How This Book Is Organized

Part I — Foundations and Decisions (Chapters 1–5) covers the trade, the decision, boundaries, the starting system, and the target design. **Part II — Communication and Data** (Chapters 6–11) covers synchronous and asynchronous communication, data ownership, sagas and the outbox, consistency and idempotency, and cross-service querying. **Part III — Migration Strategies** (Chapters 12–16) is the migration playbook. **Part IV — Reliability and Operations** (Chapters 17–22) keeps the result alive. **Part V — Security, Performance, and Cost** (Chapters 23–25) covers the three bills that come due afterward. **Part VI — Organization and Endgame** (Chapters 26–28) covers teams, anti-patterns, and a closing case study. Four appendices give a pattern cheat sheet, templates and reference material, a production readiness checklist, and a glossary.

Each chapter ends with a **Key Takeaways** box for quick review.

The Running Example: Gatecrest

Throughout the book we work on **Gatecrest**, a fictional live-events ticketing and venue operations platform. Promoters list events, fans buy tickets in a rush the moment an on-sale opens, staff scan tickets at the gate, and finance reconciles the money afterward.

We chose it because it refuses to let any hard problem stay abstract. Seat inventory has brutal contention — thousands of people want the same few hundred seats in the same few seconds. Checkout spans inventory, payment, ticketing, and notification, with no shared transaction and real money involved. Gate scanning has to keep working when venue connectivity drops, which forces an honest conversation about availability and consistency. And none of it starts clean: Gatecrest begins as a single Java monolith with one PostgreSQL database, a table five modules read, and a deploy that takes the whole platform down for a few minutes. We decompose that system across the book, and by Chapter 28 we put the result through a stadium-scale on-sale that does not go entirely to plan.

Conventions and a Note on Numbers

Code examples are Java 21 with Spring Boot for most services, Go for the venue-edge service, TypeScript for the promoter portal, and PostgreSQL for storage, with Kafka as the event backbone and Kubernetes for deployment. The patterns are not specific to those choices; where a technology detail

matters, we say so and point you at its documentation rather than guessing at defaults.

You will find no performance benchmarks, cost savings, or industry statistics in this book. Those figures depend on workload, hardware, and configuration to the point where quoting them is misleading. Where a number appears, it is Gatecrest's own budget or a blank for you to fill in from your own measurements.

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

> **NOTE**
> Background information, a clarification, or a useful aside.

> **WARNING**
> A common mistake or a behavior that causes outages, data loss, or a migration that never finishes.

Code appears in `monospace`, diagrams are drawn as ASCII art so they survive any reader, and long lines are wrapped to fit the page. Money is always an integer in minor units with a currency code — never a floating-point number.

Contents

Preface

PART I — FOUNDATIONS AND DECISIONS

1 What Microservices Actually Buy You — and What They Cost

2 Is This the Right Move? Deciding Before You Split

3 Service Boundaries: Domains, Capabilities, and Coupling

4 The Gatecrest Monolith: The System We Will Decompose

5 Designing the Target Architecture

PART II — COMMUNICATION AND DATA

6 Synchronous Communication: APIs, Contracts, and Versioning

7 Asynchronous Messaging: Events, Commands, and Brokers

8 Data Ownership: One Database per Service

9 Distributed Transactions: Sagas and the Transactional Outbox

10 Consistency, Idempotency, and the Exactly-Once Illusion

11 Querying Across Services: CQRS, Projections, and Search

PART III — MIGRATION STRATEGIES

12 The Strangler Fig: Routing Traffic Away from the Monolith

13 Extracting the First Service Safely

14 Splitting the Database

15 Dual Running, Backfills, and Cutover

16 Migration at Scale: Sequencing, Teams, and Timelines

PART IV — RELIABILITY AND OPERATIONS

17 Resilience: Timeouts, Retries, Bulkheads, and Circuit Breakers

18 Observability: Logs, Metrics, Traces, and SLOs

19 Deployment: Containers, Kubernetes, and Progressive Delivery

20 Configuration, Secrets, and Environments

21 Testing Microservices: Contracts, Components, and Chaos

22 Running in Production: On-Call, Incidents, and Postmortems

PART V — SECURITY, PERFORMANCE, AND COST

23 Security: Identity, Service-to-Service Auth, and Zero Trust

24 Performance: Latency Budgets, Caching, and Backpressure

25 Cost and Efficiency: What Microservices Really Spend

PART VI — ORGANIZATION AND ENDGAME

26 Teams, Ownership, and Conway's Law in Practice

27 Anti-Patterns and When to Merge Services Back

28 Case Study: Gatecrest Through a Peak On-Sale

APPENDICES

A Pattern Cheat Sheet

B Templates and Reference

C Production Readiness Checklist

D Glossary

PART I

Foundations and Decisions

What the architecture actually buys and what it costs, how to decide honestly whether to split at all, how to find boundaries that hold, a full tour of the Gatecrest monolith we will decompose, and the target architecture we are aiming at.

1 · What Microservices Actually Buy You

2 · Is This the Right Move?

3 · Service Boundaries

4 · The Gatecrest Monolith

5 · Designing the Target Architecture

CHAPTER 1 · PART I — FOUNDATIONS AND DECISIONS

What Microservices Actually Buy You — and What They Cost

Microservices are a trade, not an upgrade. This chapter defines the style precisely, lists what it genuinely buys you, states the bill just as plainly, introduces the distributed monolith you are most likely to build by accident, and starts the story of Gatecrest with an on-sale that the monolith barely survived.

A Definition Narrow Enough to Be Useful

Most arguments about microservices are really arguments about definitions. So we will fix one and use it for the rest of the book. A microservices architecture is a system built from services where each service is:

- **Independently deployable.** You can release it to production on its own, at a time of your choosing, without coordinating a release of anything else.
- **Owned by one team.** A single team decides what goes into it, reviews its changes, and carries its pager.
- **In sole possession of its data.** No other service reads or writes its tables. Data leaves only through an interface the owning team controls.

All three matter, and the third is the one people quietly drop. A system of forty deployable units that all read the same `orders` table is not a microservices architecture; it is a monolith whose call stack now crosses a network. The data rule is what makes the deployment rule true, because as long as two components share a schema, a migration in one is a release coordination problem for the other.

Notice what the definition does not mention. It says nothing about size. "Micro" is the most misleading syllable in our industry: it has caused more damage than any other single word in this field, because it suggests that smaller is better and that lines of code are the unit of design. They are not. A service that owns ticket issuance for a ticketing platform may be twenty thousand lines and be exactly the right size, because it owns one coherent responsibility and one set of tables. A service that owns "sending one kind of email" may be two hundred lines and be the wrong size, because nothing about it can change independently — every new notification requires changes in it and in whoever triggers it.

The definition also says nothing about containers, Kubernetes, service meshes, or message brokers. Those are common implementation choices, and we will use them in this book, but you can build a microservices architecture without any of them and you can run a monolith on all of them. Confusing

the platform with the architecture is how teams end up having spent a year on infrastructure and still having one release train.

> **NOTE**
>
> Throughout the book, "service" means a unit that satisfies all three properties above. When we mean a deployable that does not own its data, we say "component" or, more bluntly, "part of the distributed monolith".

What You Genuinely Buy

There are real benefits here. They are narrower than the marketing suggests, but they are not imaginary, and each of them is the answer to a specific pain that some teams genuinely have.

Independent deployment cadence

This is the benefit. Everything else on the list is secondary, and if you are not going to get this one, you should probably not do the rest of this book's work at all.

In a single deployable owned by several teams, every release is a joint release. The queue is shared, so a broken change from one team blocks everyone behind it. The rollback is shared, so reverting one team's mistake reverts three other teams' work. The release window is shared, so the most conservative team's risk appetite becomes everyone's. The effect is not mainly technical — it is that each team's ability to ship becomes a function of other teams' calendars, and that dependency grows roughly with the number of teams.

Split ownership along service lines and that coupling dissolves. The catalog team ships an autocomplete fix on a Tuesday afternoon while the payments team is mid-way through a risky refactor, and neither has to know about the other. That is the prize. It is worth a great deal to an organization of ten teams and almost nothing to an organization of one.

Fault isolation, when you actually build it

In a monolith, a memory leak in report generation eventually takes down checkout, because they share a heap. A connection pool exhausted by a slow analytics query starves the request path, because they share a pool. A thread pool saturated by a slow third-party call stops serving everything, because threads are a global resource.

Separate processes remove those specific shared-fate mechanisms. A crash in a reporting service does not touch the checkout process. But — and this qualifier is doing enormous work — separate processes create a new shared fate through the call graph. If checkout calls a fraud-scoring service synchronously with no timeout, then fraud scoring being slow is checkout being slow, and fraud scoring being down is checkout being down. You have converted a shared-heap problem into a shared-latency problem, and made it harder to see.

Fault isolation is not a property you get from splitting. It is a property you get from timeouts, bounded concurrency, fallbacks, and honest thinking about which dependencies are critical. Chapter 17 is entirely about earning it. For now, hold the claim in its accurate form: splitting makes fault isolation *possible*; it does not make it *true*.

Targeted scaling

A monolith scales as one unit. To get more capacity for the one component under load, you run more copies of everything — including the components that were not busy, each carrying its full memory footprint and its share of the database connections.

This becomes a real argument when parts of your system have genuinely different load shapes. Gatecrest is a clean example. Seat inventory during an on-sale is a burst of write contention on a small number of rows, concentrated in minutes. Gate scanning is a burst of short, read-heavy, latency-sensitive requests on the evening of the event, from a network you do not control. Reporting is a long, heavy, read-only scan that nobody is waiting on. Those three want different instance sizes, different concurrency limits, different database configurations, and in the case of the gate, a different physical location.

If all your components have the same load shape and scale together, this benefit is worth very little, and you should be honest about that when you make the case.

Technology choice per service

Separate deployables can be written in different languages and can use different data stores. The honest framing is that this is occasionally decisive and usually a liability.

It is decisive when a component has a requirement the house language serves badly. Gatecrest's entry scanning runs on a small box in a venue equipment cupboard, must start fast, must run in a few hundred megabytes, and must keep admitting people when the venue's uplink drops. A small Go binary with an embedded local store fits that shape much better than a JVM service that assumes a reachable primary database. That is a real reason, tied to a real constraint.

It is a liability when it is exercised because it is available. Every additional language is another set of build tooling, another dependency-vulnerability process, another observability integration, another idiom for new joiners to learn, and another on-call rotation that cannot easily be covered by the neighboring team. Most healthy organizations we would describe as doing this well have one primary language, one sanctioned alternative for a specific class of problem, and a written reason for each exception.

Team autonomy at scale

The last benefit is organizational, and it is the reason microservices spread in the first place. Beyond some number of engineers in one codebase, coordination cost stops being annoying and starts being the dominant cost. Every change needs to be understood by people who did not write the surrounding

code. Merge conflicts become a scheduling problem. The build slows down because the build does everything. Ownership blurs, and code with blurred ownership decays.

Service boundaries give you a place to put an ownership line that is enforced by the runtime rather than by a wiki page. Inside the boundary, a team can refactor freely, choose its own patterns, and move at its own speed. Across the boundary, it must honor a contract. That is a much cleaner arrangement than "please do not edit our package", and it is the arrangement that makes a large organization able to ship in parallel.

> **TIP**
>
> When you write the case for a split, write the benefits in this order: deployment independence, then team autonomy, then scaling, then isolation, then technology. If your case rests mainly on the last two, look harder — those are usually solvable without a network hop.

What You Pay

Now the bill. None of these costs is a reason not to proceed; all of them are reasons to proceed deliberately. The teams that regret the move are almost always the ones who were told about the benefits and discovered the costs afterward.

Every call can now fail halfway

In one process, a method call either returns a value or throws. Those are the outcomes. Over a network there is a third outcome, and it is the one that generates incidents: the call may have succeeded on the other side while you never learned about it. A timeout does not mean "did not happen". It means "unknown".

Look at what a single line of code becomes. Here is the in-process version, from the Gatecrest monolith:

gatecrest-core/src/main/java/com/gatecrest/orders/CheckoutService.java

```
// In-process: two outcomes. Returns, or throws. Same transaction.
PaymentResult result = paymentService.capture(order.id(), order.totalMinor());
if (result.status() == CAPTURED) {
    order.confirm();
}
```

And here is the same intent across a service boundary, written honestly:

order-service/src/main/java/com/gatecrest/order/PaymentClient.java

```
// Across a network: three outcomes, and the third one needs a plan.
CaptureResponse response;
try {
    response = paymentClient.capture(
        new CaptureRequest(order.id(), order.totalMinor(), idempotencyKey));
```

```
} catch (TimeoutException e) {
    // We do NOT know whether the money moved. Do not retry blindly.
    // Record intent, reconcile later, keep the order PENDING.
    captureAttempts.recordUnknown(order.id(), idempotencyKey);
    throw new CaptureOutcomeUnknownException(order.id(), e);
} catch (ServiceUnavailableException e) {
    // Probably did not happen, but "probably" is not "certainly".
    captureAttempts.recordFailed(order.id(), idempotencyKey);
    throw new CaptureFailedException(order.id(), e);
}
```

The business logic did not get more complicated. The failure handling did, and it multiplied by the number of boundaries a request crosses. Two consequences follow immediately and stay with us for the whole book. First, every call needs a timeout, because the default in most clients is to wait longer than your caller is willing to wait. Second, every operation that can be retried must be idempotent, because a retry after an unknown outcome is a duplicate by default — and a duplicate in payments is a second charge. Chapter 10 is devoted to that problem.

Data is spread out, and there is no global transaction

This is the cost that surprises people who expected the hard part to be infrastructure. In the Gatecrest monolith, reserving a seat, creating an order, and capturing a payment happen in one PostgreSQL transaction. Either all of it is durable or none of it is. That guarantee is free, it is total, and it is doing more work in your codebase than you probably realize.

Split inventory, orders, and payments into three services with three databases and it is gone. There is no `COMMIT` that spans them. What replaces it is a sequence of local transactions with compensating actions when a later step fails — a saga — plus a reliable way of publishing the fact that a local transaction happened, which is the transactional outbox. Both are covered in Chapter 9. Both work. Both are considerably more code and considerably more thinking than `@Transactional`.

Two-phase commit across services exists and is usually the wrong trade: it converts independent failure into shared failure, holds locks across the network, and makes the availability of every participant a precondition for any of them making progress. We will explain the mechanism in Chapter 9 so you can recognize when someone proposes it, and then not do it.

Querying gets harder in a way that is less dramatic and more pervasive. A promoter's dashboard wants event details, tickets sold, revenue, and refund count on one screen. In the monolith that is one SQL statement with three joins. Across services it is either a fan-out of calls whose latency is the sum of the slowest path, or a maintained read model fed by events, which is faster and introduces a lag you must name and defend. Chapter 11 works through both.

Debugging becomes an act of correlation

In a monolith, a failure gives you a stack trace. It names the file and the line. You attach a debugger to one process and step through the whole request.

In a distributed system, a failure gives you a 500 at the edge and a research project. The cause is in one of six processes, in a message that was consumed four seconds later, or in a retry storm that started somewhere upstream. You cannot step through the request because there is no single place where the request exists.

What replaces the stack trace is a trace: a correlation identifier propagated on every call and every message, with spans recorded by each participant and stitched together afterward. That works well, and Chapter 18 shows how to build it. But note the shape of this cost. Tracing is not something you add when you need it; it is something that must already be there when the incident happens, because the incident will not reproduce. You are paying for instrumentation you will use rarely and cannot do without.

Operational surface area multiplies

Count what one deployable needs: a build pipeline, a container image, a deployment manifest, a health check, a readiness probe, a resource request, a dashboard, an alert policy, a log stream, a secret set, a database, a migration process, a rollback procedure, and a named owner. Now multiply by the number of services and add the things that only exist because there are many: a service registry or DNS convention, a gateway, a message broker with its topics and partitions and consumer groups and retention policy, a schema registry, a certificate rotation mechanism, and a distributed tracing backend.

The multiplication is not linear in effort if you have good platform automation, which is exactly why the prerequisite chapters in this book exist. But it is never free, and it lands on people. Somebody is on call for each of those services. Somebody upgrades the broker. Somebody notices that one service's dependencies are eight months stale.

A hard dependency on deployment and observability maturity

These two are not costs so much as entry requirements, and they are strict. If deploying one service takes a manual runbook and a coordinated window, then ten services means ten runbooks and you have made everything worse. If you cannot answer "which service caused this" from your telemetry, then every incident becomes a group call where six teams each say their dashboards look fine.

The uncomfortable implication: microservices amplify the quality of your engineering practices in both directions. Good automated tests, one-command deploys, and real observability make a distributed system tractable. Their absence makes it a system nobody can safely change. Chapter 2 turns this into a checklist you can honestly score.

> **WARNING**
>
> The most expensive version of this project is the one where the split happens before the deployment automation. You end up with the coordination cost of a monolith, the failure modes of a distributed system, and no way to release either quickly. If you take one sequencing rule from this book, take that one.

The Distributed Monolith

Now the failure mode. It is not a rare pathology; it is the default outcome of a split done without attention to boundaries, and most teams who describe their microservices migration as painful are describing this.

A distributed monolith is a system that has been physically divided but not logically decoupled. The symptoms are specific and easy to check for:

- Services share a database, or one service reads another's tables directly.
- Releases go out together, because a change in one requires a matching change in another.
- One user action produces a long synchronous chain of calls, so the slowest link sets the latency and any link being down means the action fails.
- A single feature routinely requires pull requests in four repositories.
- There is a shared library containing the domain model, and bumping its version is a cross-team event.
- Local development requires running most of the system.

Here is the shape, with Gatecrest's checkout drawn badly on purpose:

```
DISTRIBUTED MONOLITH — one action, one synchronous chain

  client
    |
    v
  order-svc --> inventory-svc --> catalog-svc --> venue-svc
    |               |
    |               +--> (reads orders table directly)
    v
  payment-svc --> fraud-svc --> buyer-svc

  Availability of checkout = availability of ALL seven.
  Latency of checkout      = sum of ALL seven.
  Release of checkout      = release of ALL seven, in order.


SERVICES — one action, one owner, others informed

  client
    |
    v
  order-svc --(reserve, timeout 400ms)--> inventory-svc
    |
    +--(OrderPlaced event)--> [broker] --> ticket-svc
                                       --> notification-svc
                                       --> reporting-svc

  Checkout needs order-svc and inventory-svc to be up.
  The three consumers can be down and catch up later.
```

The difference between those two pictures is not how many boxes there are. It is how much of the system has to be working, and has to be released, for one thing to happen. That is the metric worth watching, and it is set almost entirely by where you drew the boundaries and how you chose to communicate across them — which is why Chapter 3 is the most consequential chapter in Part I.

The particularly cruel property of a distributed monolith is that it is strictly worse than the monolith it replaced. You gave up transactions, stack traces, and single-process deployment, and in exchange you got network latency, partial failure, and a longer release train. Recognizing this state early is worth a lot, because the fix is usually to merge services back — a legitimate move that Chapter 27 treats as an engineering decision rather than an admission of failure.

A Spectrum, Not a Binary

"Monolith or microservices" is a false choice that has cost the industry an enormous amount of unnecessary work. There are at least four positions, and the middle two are where most systems should live.

Style	Deploy unit	Data	Good when
Monolith	One	One schema, shared freely	Small team, early product, unclear domain
Modular monolith	One	One schema, enforced module ownership	Most teams, most of the time
Service-oriented	A few coarse services	A store per service	Clear, stable, high-level seams
Microservices	Many	A store per service	Many teams shipping in parallel

The modular monolith deserves particular attention because it gets you most of the design benefit at none of the distributed cost. You keep one deployable and one database, and you enforce module boundaries in the build: each module has a public interface, other modules may only use that interface, and a build rule fails the compile when someone reaches around it. Tables are owned by exactly one module by convention, and enforced by review or by schema permissions. Modules communicate through in-process events where they would later communicate through a broker.

What that buys you is the hard part of the work — finding the boundaries — done and validated while mistakes are still cheap. Moving a class between packages is a refactor you can do in an afternoon. Moving a table between services is a migration project. Doing the boundary work first, in a modular monolith, means that when you do extract a service you are extracting along a seam you have already lived with.

It is also the position you should be arguing for in most rooms where someone has proposed microservices. Chapter 2 gives you the questions that decide between them.

> **NOTE**
>
> A modular monolith is not a stepping stone you are obliged to step off. Plenty of systems should stop there permanently. Treating it as a temporary state is how teams talk themselves into a split they do not need.

Gatecrest, and the On-Sale That Nearly Didn't Happen

Every chapter in this book works on one system, so we introduce it now.

Gatecrest is a live-events ticketing and venue operations platform. Promoters list events through a portal. Fans buy tickets — usually calmly, occasionally in a violent rush when a popular on-sale opens. Venue staff scan tickets at gates using handheld Android devices. Finance reconciles money, refunds, and promoter payouts afterward. Partner promoters call a public API to manage their own inventory.

The system is `gatecrest-core`: a Java 21 and Spring Boot modular monolith with one PostgreSQL database named `gatecrest`. It has packages for `catalog`, `inventory`, `orders`, `payments`, `ticketing`, `entry`, `notifications`, `promoters`, and `reporting`. The module boundaries are real in the source tree and thoroughly ignored in the database, where everything joins to everything. There is a React promoter portal and the Android gate scanner app. Chapter 4 gives the full tour, with schema and code.

It is, by most measures, a good monolith. The tests are decent. The domain model is coherent. A single developer can run the whole thing locally with one command. Nobody involved is embarrassed by it, and that is important to the story — this is not a rescue mission from a legacy disaster, it is a competent system reaching the edge of what its shape can do.

The moment that starts the book is an on-sale for a large outdoor festival, and here is what the on-call engineer watched happen.

Fans queued in the web front end for several minutes before the gates opened. At the moment of the on-sale, traffic arrived as a near-vertical step. The seat-hold mechanism — a row in `seat_holds` plus an `UPDATE` on an allocation counter — began serializing, which is what row locks are supposed to do. Hold requests started queuing behind each other. Because every hold is taken inside the same transaction that will later create the order, those transactions got longer, so they held their locks for longer, so the queue grew faster than it drained. Latency climbed.

Then the amplification. The checkout transaction also calls the payment provider, `NORTHGATE-PAY`, and the provider — dealing with the same spike — got slower. Now database transactions were being held open across a slow external HTTP call. The connection pool filled. Because the pool is shared by the entire monolith, everything that needed a connection began to fail: the event catalog stopped rendering, so fans who were not even buying started reloading, which added load. The promoter portal

timed out, so promoters started calling support. And the nightly reporting job, which had been scheduled without much thought, was still finishing its run against the primary database, consuming buffer cache and connections that checkout needed.

Nothing crashed. That is worth noting, because it is typical: the system did not fall over, it degraded everywhere at once, which is harder to diagnose and harder to explain. Tickets did sell. A number of fans saw errors and retried, and some of those retries created duplicate holds that expired later, so inventory looked wrong for an hour. Two orders were charged twice, because a retry hit the provider after a timeout. The festival sold out. Everyone went home tired, and the postmortem contained the sentence that this book is an answer to: *we cannot scale checkout without scaling reporting, and we cannot fix the hold mechanism without redeploying the entire platform.*

Read that sentence carefully, because it is a good reason and it is a specific one. It is not "the monolith is messy". It is two named couplings, each blocking a concrete thing the business needs. That is the standard of evidence Chapter 2 will hold every split to.

> **TIP**
>
> When you write your own version of that sentence, force it into the form "we cannot do X without also doing Y". If you cannot fill in both blanks with something concrete, you do not yet have a reason to split — you have a feeling.

What This Book Will and Will Not Do

The plan for the rest of the book, so you know what you are buying.

Part I decides. Chapter 2 tries to talk you out of this, seriously, with a framework and a prerequisite checklist. Chapter 3 covers boundaries, which is where most of the value and most of the risk live. Chapter 4 gives you the Gatecrest monolith in detail — packages, DDL, the real checkout code — because every later chapter transforms something in it. Chapter 5 designs the target architecture.

Part II builds the mechanisms: synchronous APIs and versioning, asynchronous messaging, data ownership, sagas and the outbox, idempotency, and cross-service querying.

Part III is the migration: the strangler fig, extracting the first service, splitting the database, dual running and cutover, and sequencing the whole program across teams.

Part IV keeps it alive: resilience, observability, deployment, configuration, testing, and on-call.

Part V covers security, performance, and cost. **Part VI** covers teams and ownership, the anti-patterns, when to merge services back, and a closing case study that takes Gatecrest through a peak on-sale in its new shape.

Two things this book will not do. It will not tell you that microservices are the right answer — for a meaningful share of readers, the correct outcome of Chapter 2 is a modular monolith and a much shorter project. And it will not give you numbers we cannot stand behind. Where scale matters, we

describe the mechanism and tell you what to measure in your own system, because a latency figure from someone else's architecture is not evidence about yours.

What it will do is walk one realistic system from a monolith to a set of services, showing the code, the schema, the failure modes, and the decisions — including the ones that turn out to be wrong and get reversed.

Key Takeaways

- A service is independently deployable, owned by one team, and sole owner of its data. Drop the third property and the first becomes a fiction.
- "Micro" is about independence, not size. Lines of code are the wrong unit of design.
- The primary benefit is independent deployment cadence; team autonomy, targeted scaling, fault isolation, and technology choice follow behind it.
- Splitting makes fault isolation possible, not automatic. Without timeouts and bounded concurrency you have traded shared heap for shared latency.
- The costs are partial failure, no global transaction, correlation-based debugging, multiplied operational surface, and a hard prerequisite on deployment and observability maturity.
- The distributed monolith is the usual failure mode and is strictly worse than the monolith it replaced. Check for shared tables, lockstep releases, and long synchronous chains.
- Monolith, modular monolith, service-oriented, and microservices are a spectrum. The modular monolith gets the design benefit without the network and is the right answer for many teams.
- Gatecrest's on-sale gave two concrete reasons to split: checkout cannot scale without reporting, and the hold mechanism cannot change without redeploying everything.