

Microservices A Visual Guide

**Essential Architecture, Communication, Data, Reliability, Deployment, and Operations for
Beginners and New Systems Engineers**

Reiner Braun

Sample Book — Chapters 1 to 2

Introduction

This book is designed for readers who are taking their first steps with microservices, including systems engineers who want to build the right foundation from the start.

Rather than asking you to memorize more than you need, the book focuses on the essential concepts that will support your future study and day-to-day work. More advanced topics are included only when they are likely to become useful to many readers, and even then they are introduced only far enough to explain their purpose, context, and overall role.

Processes, relationships, and structures that can be difficult to picture from text alone are supported by diagrams. By moving between explanations, visuals, and examples, you can build a clearer and more practical understanding of the subject.

Product names, service names, logos, icons, and other identifying marks mentioned in this book are trademarks or registered trademarks of their respective owners. This book is not an official publication, certified training resource, or endorsed learning material of any company or organization.

How to Use This Book

- Read the chapters in order the first time through. Each chapter builds on terminology, assumptions, and operating practices introduced earlier, so this sequence develops the required background step by step.
- Before reading a chapter, review its **What You'll Be Able to Do** section. These outcomes identify the knowledge and practical judgments that deserve the most attention.
- Use the diagrams to understand relationships, processes, structures, boundaries, and changes over time. They support the explanation in the text rather than replace it.
- Try the code, commands, and hands-on examples yourself whenever your environment permits. Observing the request, output, failure, or recovery behavior is more valuable than only reading the example.
- At the end of each chapter, use the **Chapter Summary** and **Review Questions** to confirm that you understand and can recall the essential fundamentals.
- Treat advanced-topic overviews as signposts rather than material to memorize now. Return to them when a project creates the specific problem they are intended to solve.

Table of Contents

[Introduction](#)

[How to Use This Book](#)

[Part I: Build the Correct Mental Model](#)

[Chapter 1: The Big Picture—What Microservices Are](#)

[What You'll Be Able to Do](#)

[Begin with the architectural trade-off](#)

[One customer action can cross several services](#)

[Microservices compared with related structures](#)

[Why teams adopt microservices](#)

[The costs appear immediately](#)

[Conditions that support success](#)

[When a monolith is the better choice](#)

[The systems engineer's role](#)

[The first misconception to remove](#)

[Chapter Summary](#)

[Review Questions](#)

[Chapter 2: Finding Effective Service Boundaries](#)

[What You'll Be Able to Do](#)

[Boundaries determine whether independence is real](#)

[Cohesion and coupling](#)

[Deriving candidate boundaries from real work](#)

[A step-by-step decomposition of the example](#)

[Why technical-layer services usually preserve coupling](#)

[Service, team, contract, and operation should align](#)

[Map dependencies before approving a split](#)

[Warning signs of a poor boundary](#)

[Granularity is an outcome, not a quota](#)

[Avoid the “one noun, one service” mistake](#)

[Chapter Summary](#)

[Review Questions](#)

Part I: Build the Correct Mental Model

Chapter 1: The Big Picture—What Microservices Are

What You'll Be Able to Do

After completing this chapter, you will be able to:

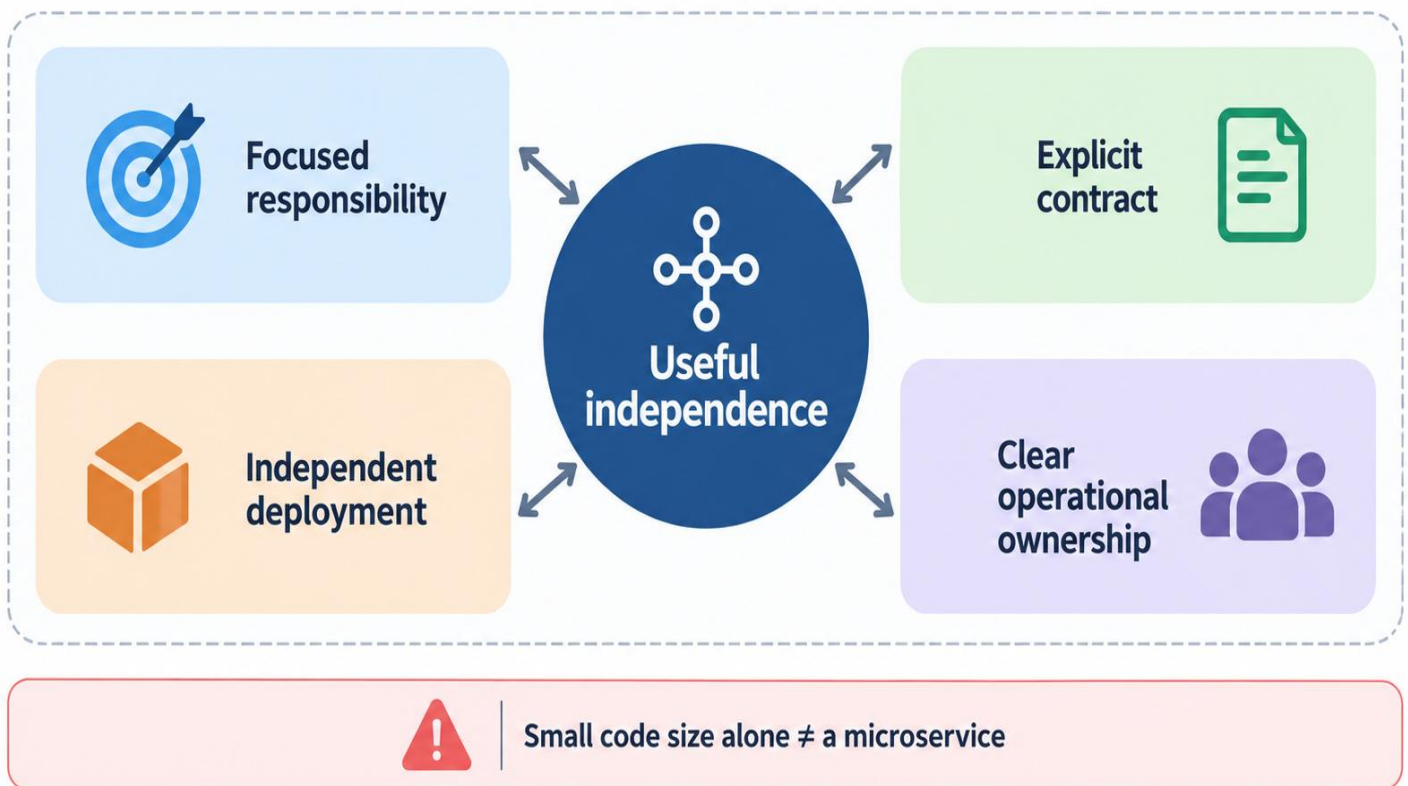
- Define a **service**, **microservice**, **monolith**, and **modular monolith** in practical terms.
- Explain why independent change and deployment are the central benefits of microservices.
- Identify the distributed-system and operational costs introduced by service boundaries.
- Judge when a monolith is the safer starting point and when microservices may be justified.

Begin with the architectural trade-off

A microservice architecture is successful when it creates useful independence. **A service should be independently changeable, deployable, and operable within a clearly owned responsibility.** Small code size alone does not create that independence.

Chapter 1: The Big Picture—What Microservices Are

1-1 Useful Independence



001

Figure 1-1: Useful independence: responsibility, contract, deployment, and operation

A **service** is a running software component that provides a defined capability through an interface. A **microservice** is a service with a focused business responsibility, an explicit contract, clear ownership, and the ability to be deployed without requiring a simultaneous deployment of the whole system. A **contract** is the agreed request, response, message, and error behavior on which other components may depend.

An **application programming interface**, or API, is a defined set of operations through which software components interact. An **endpoint** is a particular network-accessible operation or address exposed by an API.

The word *micro* can be misleading. There is no correct number of source files, endpoints, database tables, or developers for a microservice. A service is too large when unrelated responsibilities prevent independent change. It is too small when ordinary business work requires constant coordination across many services.

Four properties matter more than size:

- **Independent deployment:** The service can be released without rebuilding and redeploying unrelated parts of the system.
- **Explicit contracts:** Other components depend on documented APIs or messages rather than internal code or tables.
- **Focused responsibility:** The service owns a coherent business capability rather than an arbitrary technical slice.
- **Clear ownership:** A team is accountable for the service's code, operation, data, and lifecycle.

These properties reinforce one another. Independent deployment is difficult when two services share internal tables. Clear ownership is weak when several teams must approve every change. A focused responsibility is not useful if its API changes unpredictably.

One customer action can cross several services

Consider a customer submitting an order. From the customer's perspective, this is one action. Inside a microservice system, it may become a sequence of network and data operations:

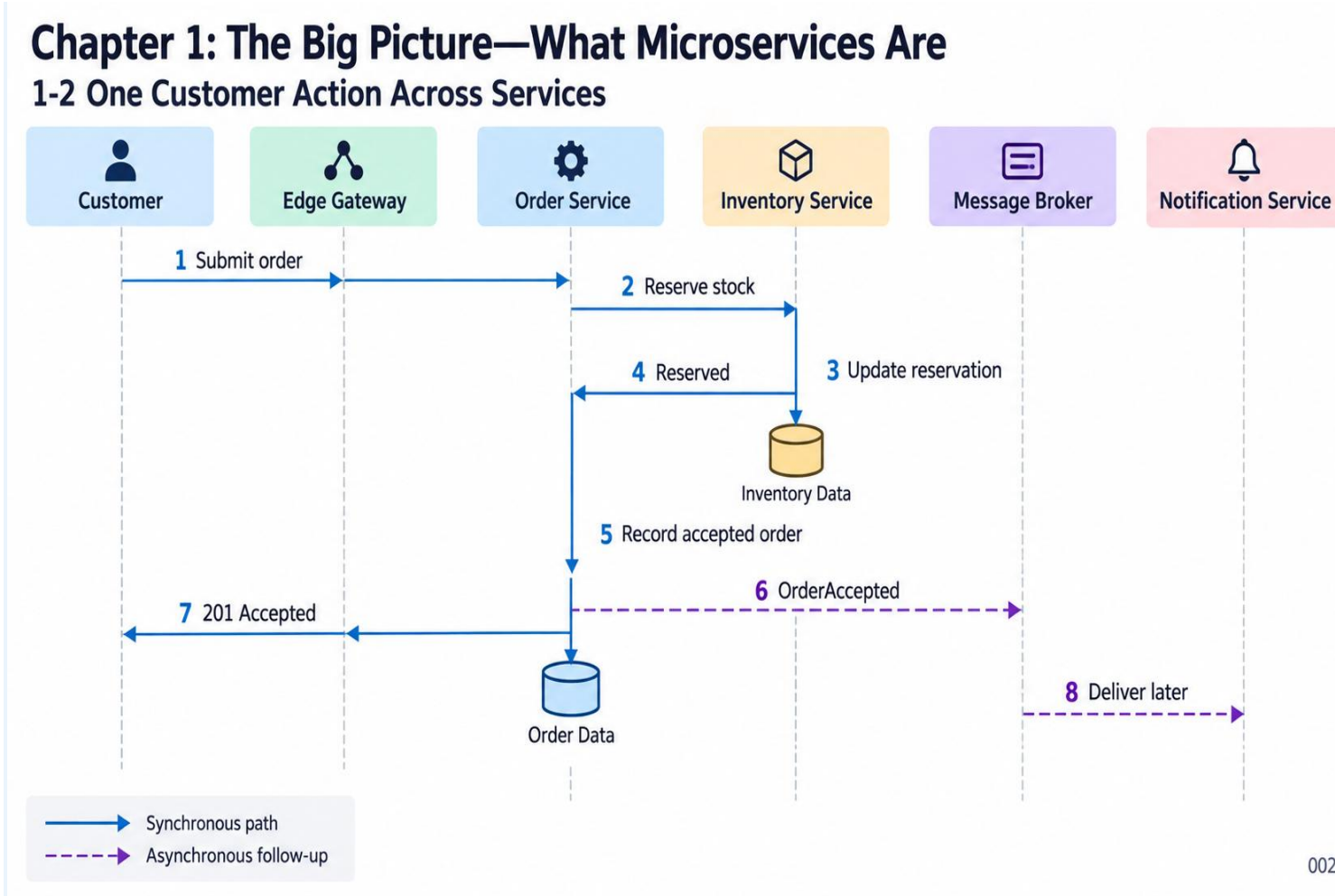


Figure 1-2: One customer action across Order, Inventory, broker, and Notification

A possible flow is:

1. The customer sends an order request to the edge gateway.
2. The gateway routes the request to an available Order service instance, which is one running copy of the service.

3. The Order service validates the request and asks the Inventory service to reserve stock.
4. The Inventory service updates the inventory it owns and returns a result.
5. The Order service records the accepted order in its data store, a system that persists data beyond one process lifetime.
6. The Order service publishes an event, a message reporting that the order was accepted.
7. The Notification service later receives the event and sends a confirmation.

This flow demonstrates both the value and the cost of microservices. Inventory can be changed and scaled separately from Notification, and a temporary notification delay does not need to block order acceptance. At the same time, the flow now depends on routing, network connectivity, compatible contracts, data consistency, message delivery, and failure handling.

Microservices compared with related structures

A **distributed system** is a system whose cooperating components run in separate processes or machines and communicate through a network. The following comparison focuses on the dimensions that most affect systems engineering. The boundaries are not absolute; organizations sometimes use these terms differently.

Structure	Deployment unit	Communication	Data ownership	Typical strength	Typical risk
Traditional monolith	One main application	Mostly in-process calls	Common application data model	Simple deployment and local transactions	Changes and scaling become coupled as the application grows
Modular monolith	One application with enforced internal modules	In-process calls through module interfaces	Logically separated, often in one database	Strong boundaries without distributed-system overhead	Weak discipline can allow modules to become entangled
Service-oriented architecture	Multiple services, often integrated through shared enterprise infrastructure	Network calls and messaging	Varies by implementation	Reuse and integration across large organizations	Central integration layers can become bottlenecks or sources of coupling
Microservices	Many independently deployable services	Explicit network APIs and messages	Each service controls its data	Independent delivery, scaling, and ownership	High operational and distributed-system complexity

A **monolith** is an application deployed as one primary unit. Monolith does not mean badly designed. A modular monolith can have excellent internal boundaries and may offer the best balance for a product that does not yet require independent deployment.

A **modular monolith** is one deployable application whose internal modules have explicit responsibilities and controlled dependencies. It is often a strong starting architecture because local function calls and database transactions remain simple while the code is still organized around business capabilities.

Service-oriented architecture, often shortened to SOA, is a broader family of designs in which network-accessible services support organizational capabilities. Microservices share some ideas with SOA, but usually emphasize decentralized ownership, small autonomous teams, independent deployment, and lightweight service-specific choices.

Why teams adopt microservices

Independent release cadence

In a large monolith, a small change may enter a shared release train with many unrelated changes. Testing and approval expand because the entire application is one deployment unit. With well-bounded services, the Notification team can release a template change without redeploying Inventory.

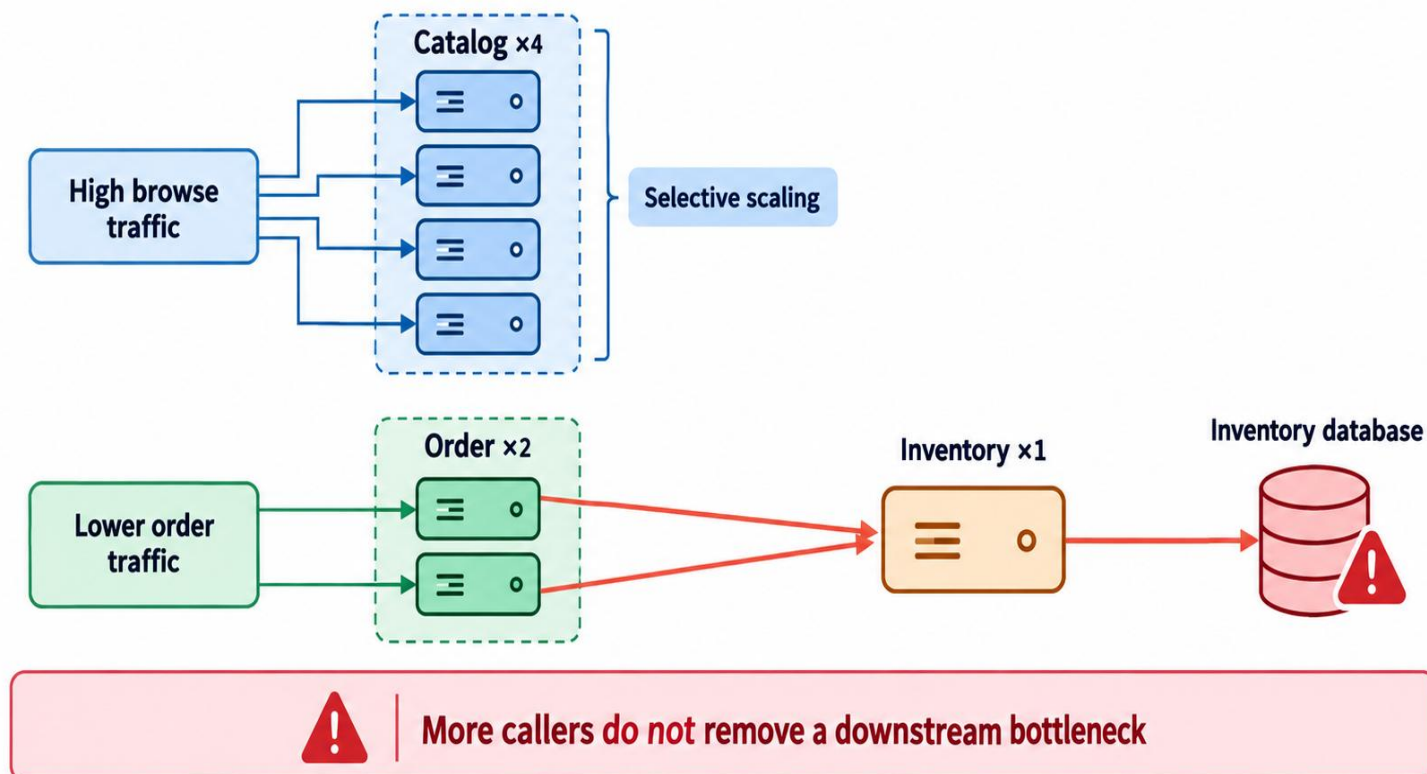
This benefit is real only when the service is actually compatible with its consumers and does not require coordinated releases. Separate repositories and pipelines do not help if all services must still be deployed together.

Selective scaling

Different capabilities may have different load patterns. Product browsing may receive thousands of reads for every order submission. A separate Catalog service can scale for read traffic without increasing the number of Inventory or Notification instances.

Chapter 1: The Big Picture—What Microservices Are

1-4-2 Selective Scaling by Capability



003

Figure 1-3: Selective scaling by capability

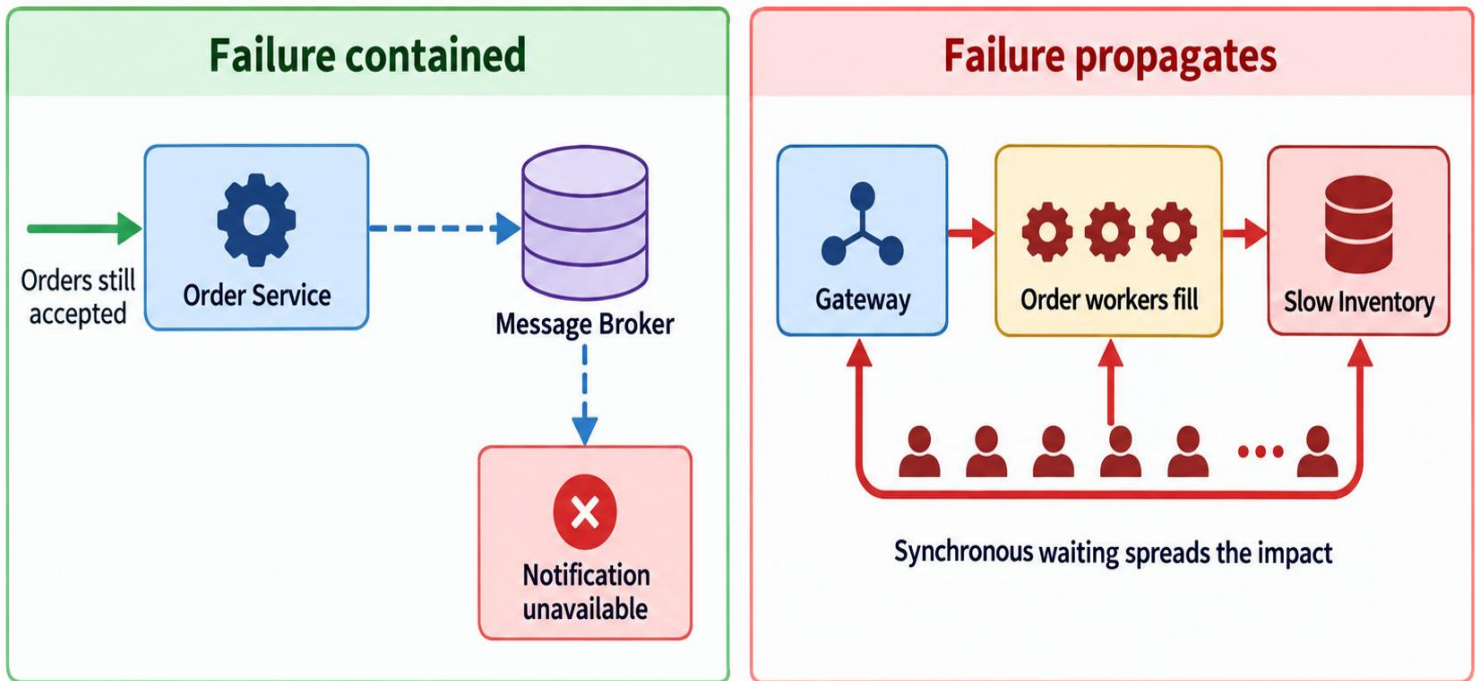
Selective scaling does not remove downstream limits. Adding Order service instances will not help if every instance is blocked by the same saturated database.

Fault isolation

A failure in one service can sometimes be contained. If Notification is unavailable, the system may continue accepting orders and deliver confirmations later. This is **fault isolation**: limiting the area affected by a failure.

Chapter 1: The Big Picture—What Microservices Are

1-4-3 Fault Isolation vs Failure Propagation



004

Figure 1-4: Fault isolation versus failure propagation

Isolation must be designed. A long synchronous chain can allow one slow service to consume resources throughout the system. Merely placing code in separate processes does not guarantee resilience.

Team autonomy

A team can own one service from design through operation, make local implementation decisions, and release on its own schedule. This can reduce communication overhead when the service boundary matches an organizational responsibility.

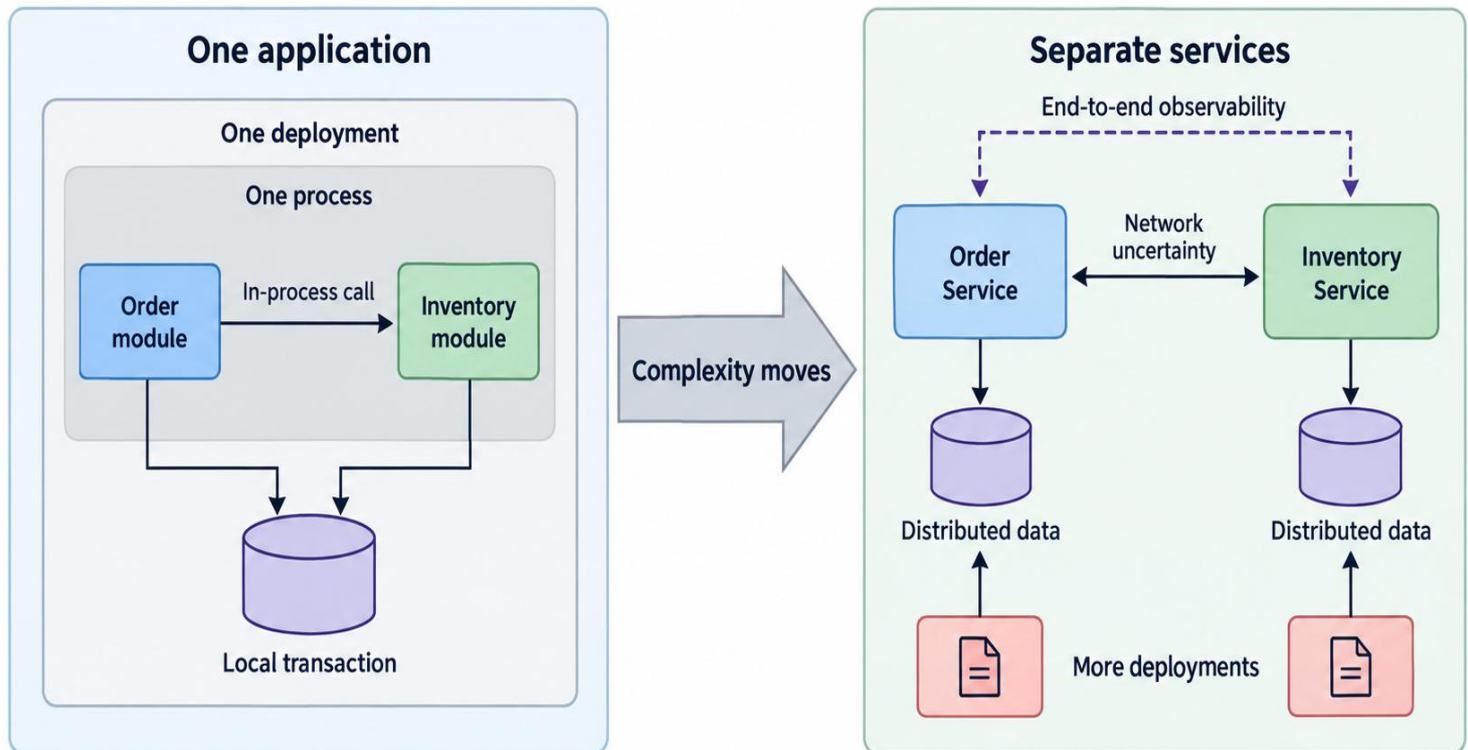
Autonomy is not the absence of standards. Teams still need common approaches for identity, deployment, telemetry, secrets, and incident response. Without them, every service becomes a separate operational puzzle.

The costs appear immediately

Microservices move some complexity rather than eliminating it. In a monolith, a call between modules may be an in-memory function call that either returns or raises an error. In a distributed system, the equivalent call may be delayed, duplicated, rejected, partially completed, or completed after the caller has stopped waiting.

Chapter 1: The Big Picture—What Microservices Are

1-5 Complexity Moves to Distributed Boundaries



005

Figure 1-5: Complexity moves from in-process calls to distributed boundaries

The main costs are interconnected:

- **Network uncertainty:** Requests have latency and can fail independently of either application process.
- **Distributed data:** One business workflow may update several data stores that cannot participate in one ordinary transaction.
- **More deployments:** Each service needs a build, configuration, rollout, health model, and recovery procedure.
- **Harder testing:** A complete workflow may depend on multiple versions, contracts, and infrastructure components.
- **Greater operational demand:** Engineers need enough visibility to trace a symptom across processes and dependencies.
- **Coordination at boundaries:** Independent teams still need compatible APIs, message schemas, and shared platform standards.

An everyday analogy is useful. A monolith is like one well-organized workshop: tools and materials are close together, communication is immediate, and one manager can see the whole job. Microservices are like a network of specialist workshops. Each can improve and schedule its work independently, but every handoff needs a clear order, reliable transportation, tracking, and a plan for delays.

Conditions that support success

Microservices are more likely to provide value when several conditions are present:

- The system is large enough that independent release or scaling solves a measurable problem.
- Business responsibilities can be separated into stable, understandable capabilities.
- Teams have clear end-to-end ownership.
- Builds, tests, deployments, and rollbacks are substantially automated.
- The runtime platform provides dependable networking, configuration, and workload management.
- Engineers can diagnose behavior across service boundaries.
- API and data changes are managed for compatibility.

Not every condition must be perfect before the first service exists. However, weak automation and unclear ownership make each additional service increasingly expensive.

When a monolith is the better choice

A monolith or modular monolith is usually the safer starting point when:

Chapter 1: The Big Picture—What Microservices Are 1-7 Choose a Monolith or Microservices

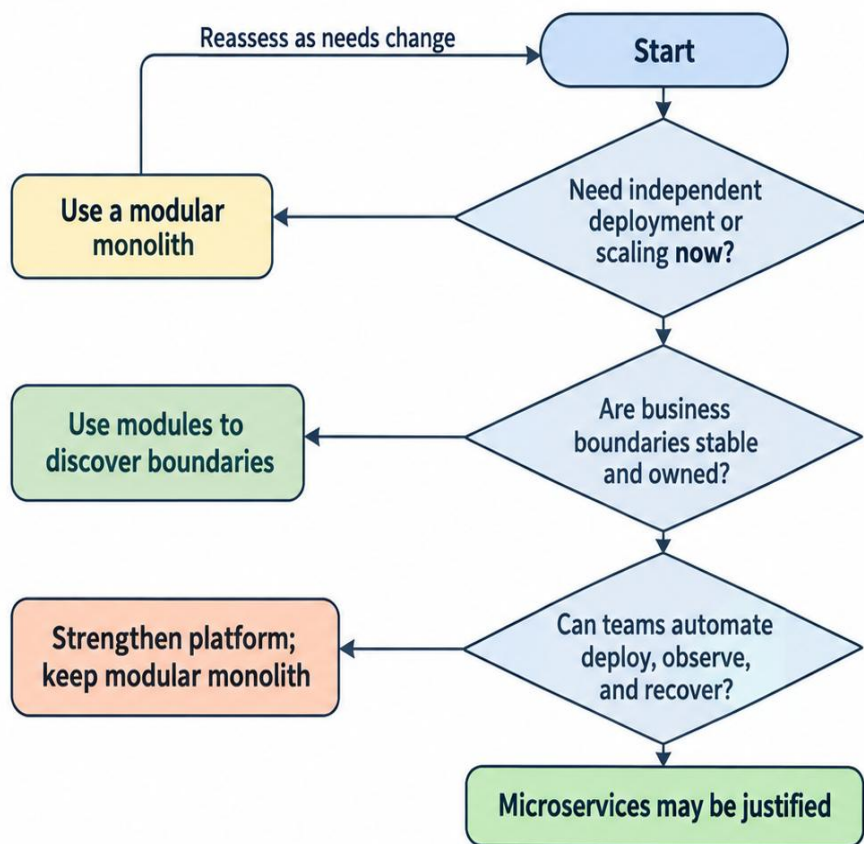


Figure 1-6: When to prefer a monolith or microservices

- One small team owns the entire product.
- The product and its domain boundaries are still changing rapidly.
- The expected load can be handled by one application and data store.
- Independent deployment does not solve a current delivery problem.
- The organization does not yet have reliable deployment and operating practices.
- Strong transactions across the whole workflow are central and difficult to separate.

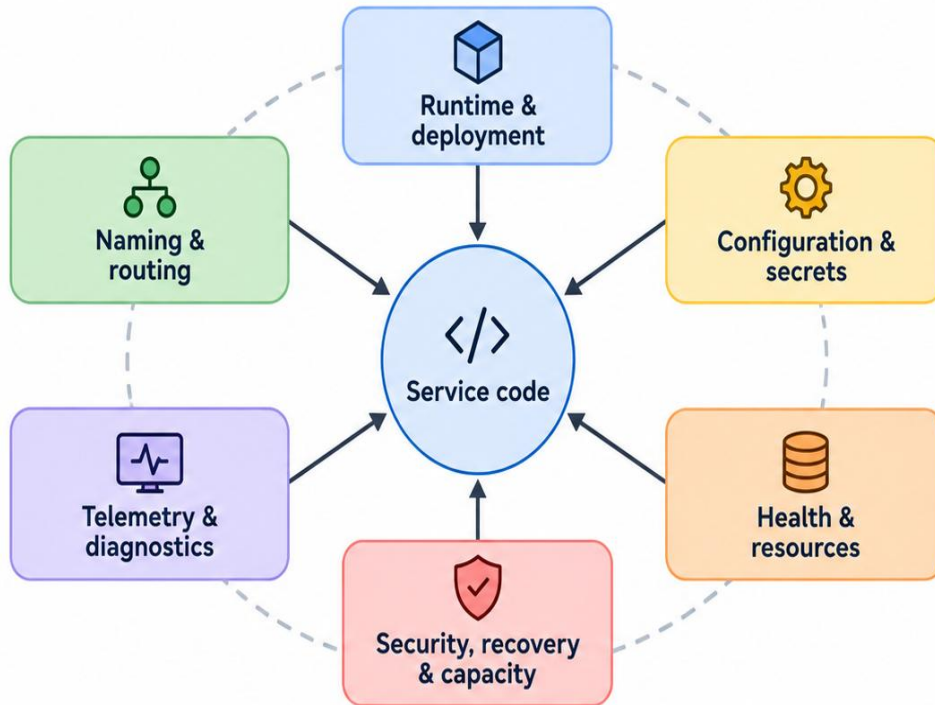
Starting with a modular monolith does not close the door to microservices. It can help a team discover stable boundaries before paying the cost of remote communication. Extracting a well-understood module later is usually easier than merging many premature services.

The systems engineer's role

Application developers often focus on business behavior inside a service. Systems engineers must also understand everything around that code. Typical responsibilities include:

Chapter 1: The Big Picture—What Microservices Are

1-8 Systems Engineer Responsibilities



Can this service change and operate independently without making the system unsafe?

007

Figure 1-7: Systems engineer responsibilities around a service

- Providing reliable runtime environments and deployment mechanisms.
- Designing service naming, routing, and connectivity.
- Managing configuration and secret delivery.
- Defining health checks and resource limits.
- Supporting failure containment and recovery.
- Establishing operational signals and diagnostic workflows.
- Enforcing baseline security and access controls.
- Participating in incident response and capacity planning.

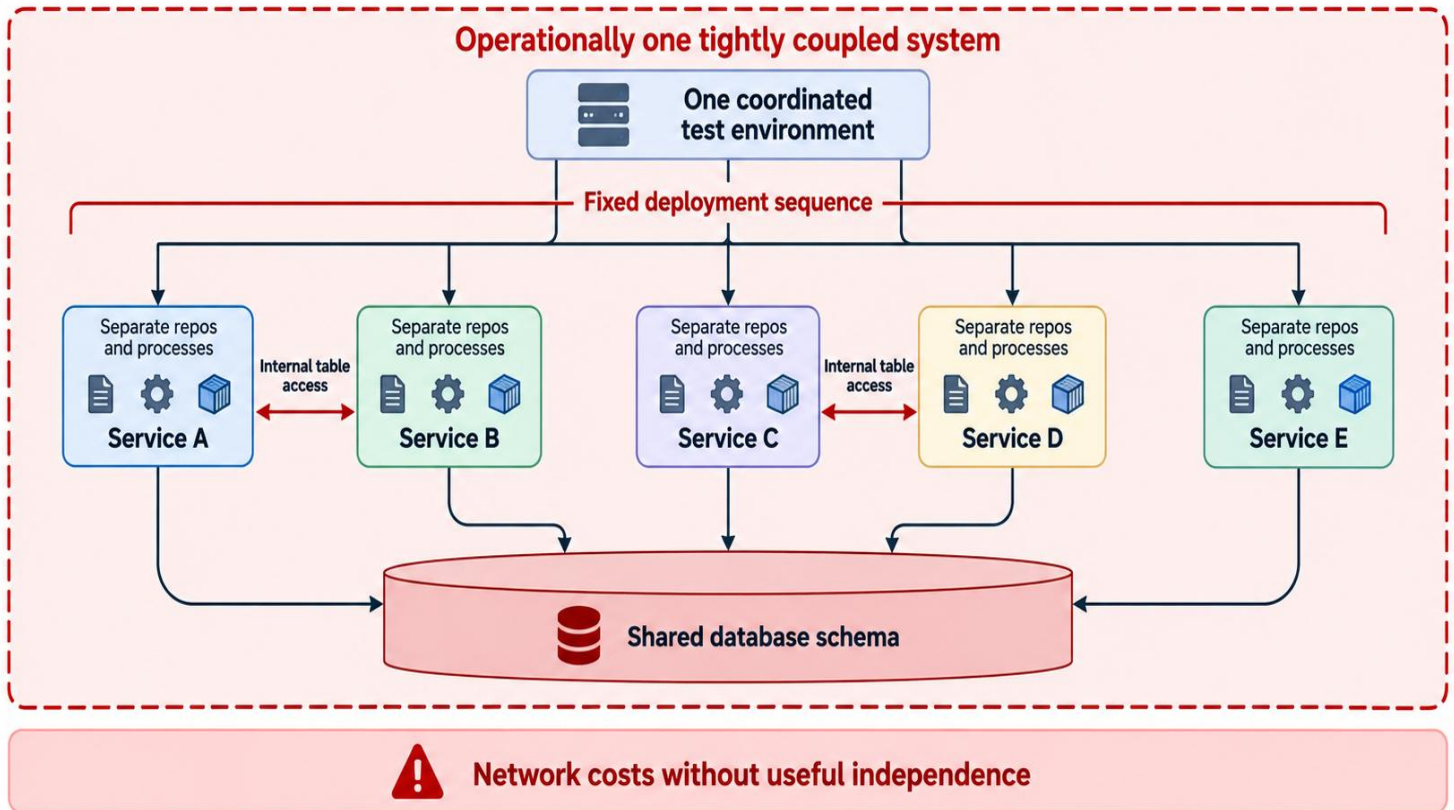
The exact role varies by organization, but the underlying question remains the same: **Can this service be changed and operated independently without making the wider system unsafe?**

The first misconception to remove

Splitting an application into repositories or processes does not automatically create microservices. Suppose five services share one database schema, use each other's internal tables, require one coordinated test environment, and must be deployed in a fixed sequence. They may be physically separate, but operationally they are one tightly coupled system. This is often called a **distributed monolith**: a system that has the deployment and network costs of distribution without meaningful independence.

Chapter 1: The Big Picture—What Microservices Are

1-9 Distributed Monolith Anti-Pattern



008

Figure 1-8: Distributed monolith anti-pattern

A useful architecture review begins with evidence of independence, not a service count.

Chapter Summary

- **Microservices are defined by useful independence, not by small code size.**
- Independent deployment depends on focused responsibility, explicit contracts, clear ownership, and operational capability.
- Microservices can support independent releases, selective scaling, fault isolation, and team autonomy, but they add network, data, testing, and operating complexity.
- A well-structured modular monolith is often the better choice when the need for independent change is limited or the required operating foundations are not ready.

Review Questions

1. Why does splitting code into several repositories not automatically create a microservice architecture?
2. Which benefits can justify accepting the additional cost of distribution?
3. What conditions make a modular monolith a stronger starting point?

4. Which systems-engineering responsibilities become more important as the number of services grows?

Chapter 2: Finding Effective Service Boundaries

What You'll Be Able to Do

After completing this chapter, you will be able to:

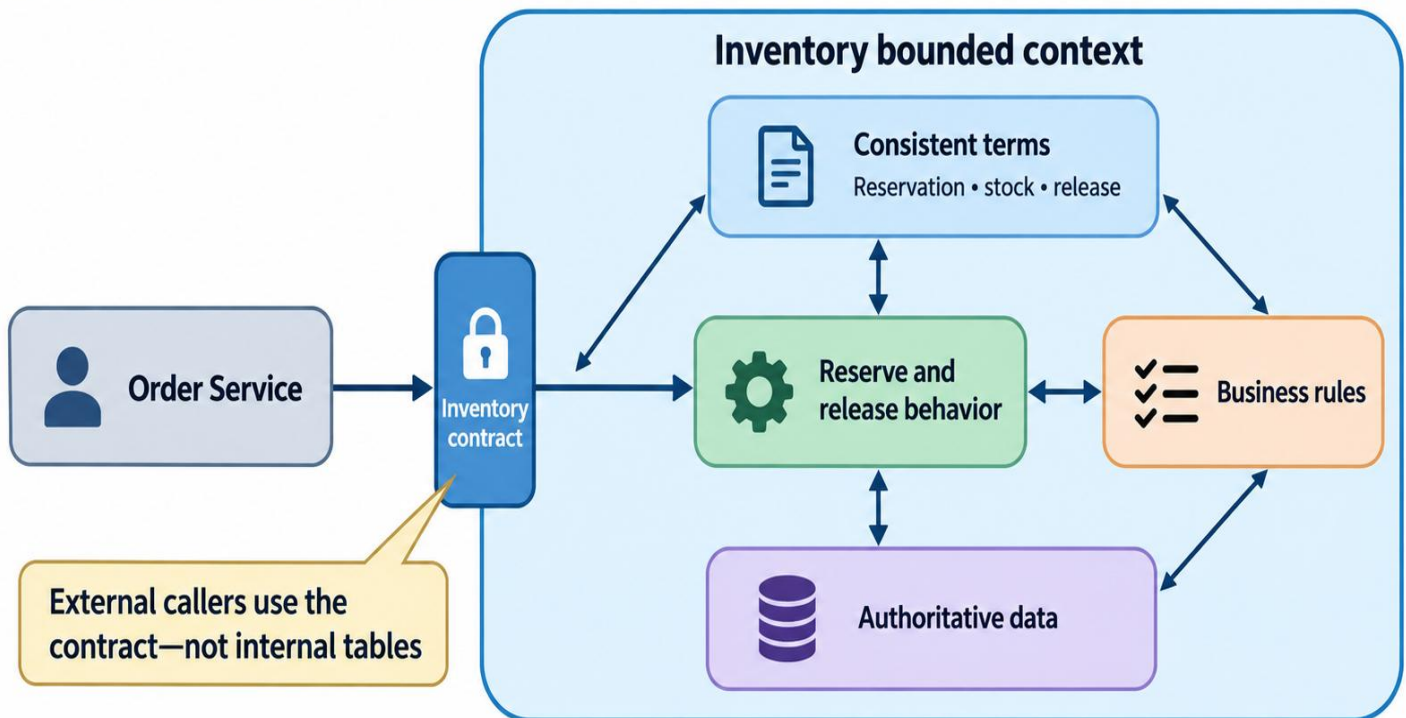
- Use business capabilities and bounded contexts to identify candidate service boundaries.
- Explain **cohesion** and **coupling** and apply them to a service design.
- Recognize boundaries that create excessive synchronous calls, shared data, or coordinated releases.
- Connect each service boundary to an accountable team and a clear source of data authority.

Boundaries determine whether independence is real

A good service boundary contains behavior and data that change for related business reasons. It reduces the number of ordinary changes that cross team and deployment boundaries. **The objective is high internal cohesion and low external coupling, not the smallest possible service.**

Chapter 2: Finding Effective Service Boundaries

2-1 Bounded Context



009

Figure 2-1: Bounded context containing consistent terms, rules, behavior, and data

A **business capability** is a unit of work that an organization recognizes, such as managing orders, reserving inventory, or sending customer notifications. Business capabilities are often better starting points for services than technical layers because they combine the rules and data needed to produce one business result.

A **bounded context** is a boundary within which business terms and rules have one consistent meaning. For example, “available quantity” may mean stock physically present in a warehouse within Inventory, while Order may care about

quantity that can still be promised to a customer. The terms are related, but each context owns its own rules and model.

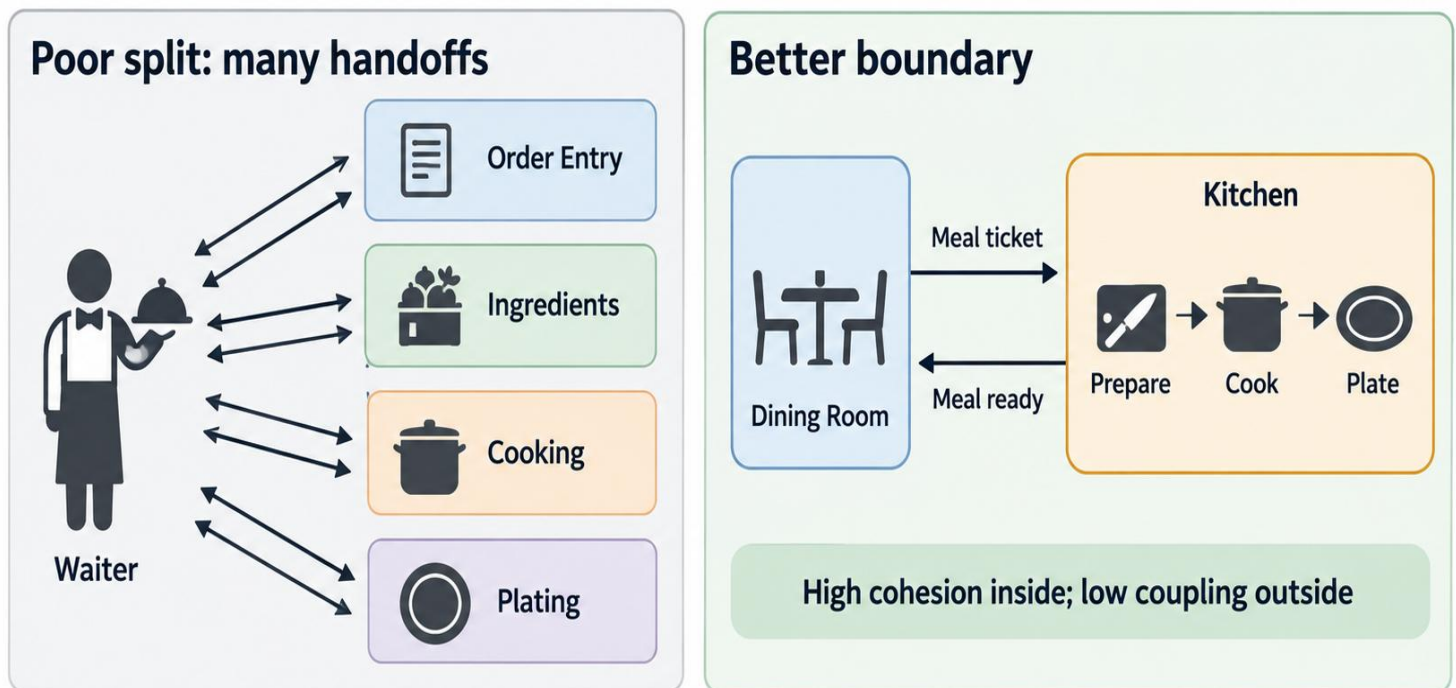
Cohesion and coupling

Cohesion describes how strongly the responsibilities inside one component belong together. **Coupling** describes how strongly one component depends on the details or timing of another.

Think of a restaurant. The kitchen groups equipment and people needed to prepare food; those responsibilities are cohesive. The dining room communicates orders and receives dishes through a defined handoff. If servers must enter the kitchen and operate specific appliances for every order, the two areas are tightly coupled.

Chapter 2: Finding Effective Service Boundaries

2-2 High Cohesion and Low Coupling



010

Figure 2-2: High cohesion and low coupling using the restaurant handoff analogy

The same distinction applies to services:

- Order creation, order status, and order cancellation often have high cohesion because they share rules and lifecycle data.
- Notification template rendering and delivery channel selection may belong together but need only a small event from Order.
- A service that asks another service for five small pieces of data during every request is probably coupled to the other service's internal model.

High cohesion does not mean a service has only one function. It means its functions support one coherent responsibility.

Deriving candidate boundaries from real work

A practical boundary exercise starts with behavior rather than a list of nouns. Consider a simple commerce workflow:

1. A customer finds a product.
2. The customer submits an order.
3. Stock is reserved.
4. The order is confirmed.
5. A notification is sent.

Chapter 2: Finding Effective Service Boundaries

2-3 Four Evidence Lenses

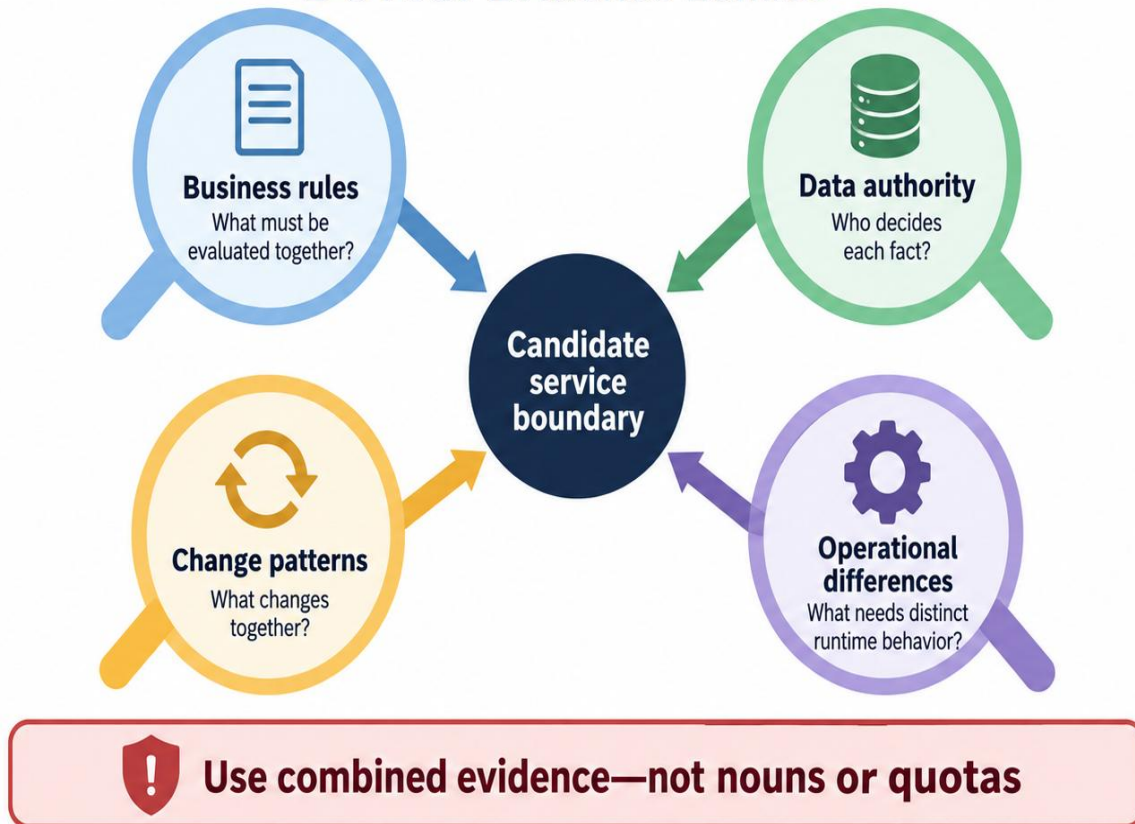


Figure 2-3: Four evidence lenses for discovering service boundaries

Now examine four kinds of evidence.

Business rules

Which rules must be evaluated together?

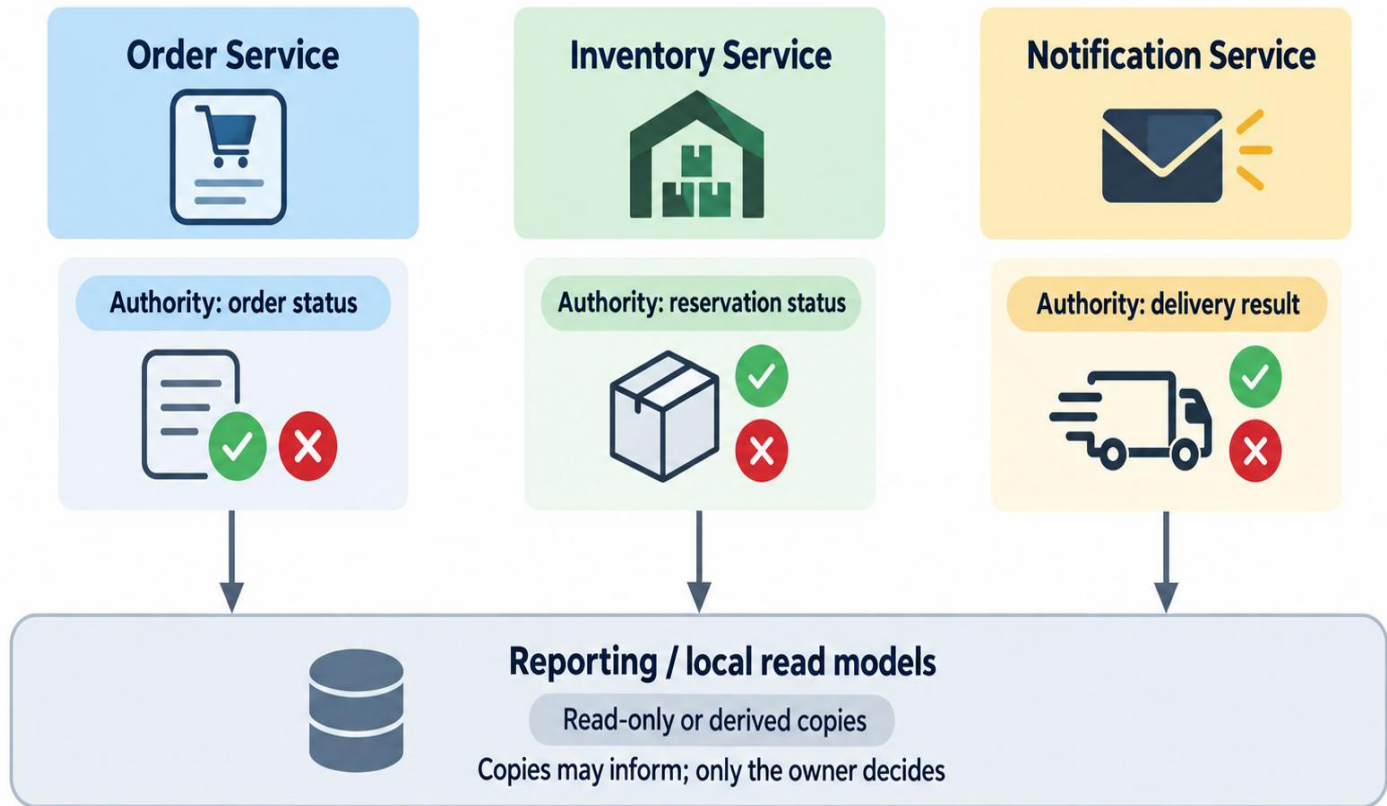
Order may own rules such as whether an order can be cancelled and which states an order may enter. Inventory may own reservation limits, warehouse allocation, and stock release. Notification may own channel preferences and message templates.

Rules that require the same data and usually change together are candidates for one boundary.

Which component is the authoritative source for each fact?

Chapter 2: Finding Effective Service Boundaries

2-3-2 One Authoritative Owner per Fact



012

Figure 2-4: One authoritative owner for each business fact

Order should decide whether an order is accepted or cancelled. Inventory should decide whether stock is reserved. Notification should decide whether a delivery attempt succeeded. Other services may retain copies for their own use, but they should not silently redefine the authoritative fact.

Change patterns

Which files, rules, and teams tend to change together?

Version-control history and work tickets often reveal hidden boundaries. If changes to stock allocation repeatedly touch order pricing code, the existing design may mix two responsibilities. If notification changes are frequent and independent, Notification may be a useful service candidate.

Operational differences

Which parts need distinct scaling, availability, or release behavior?

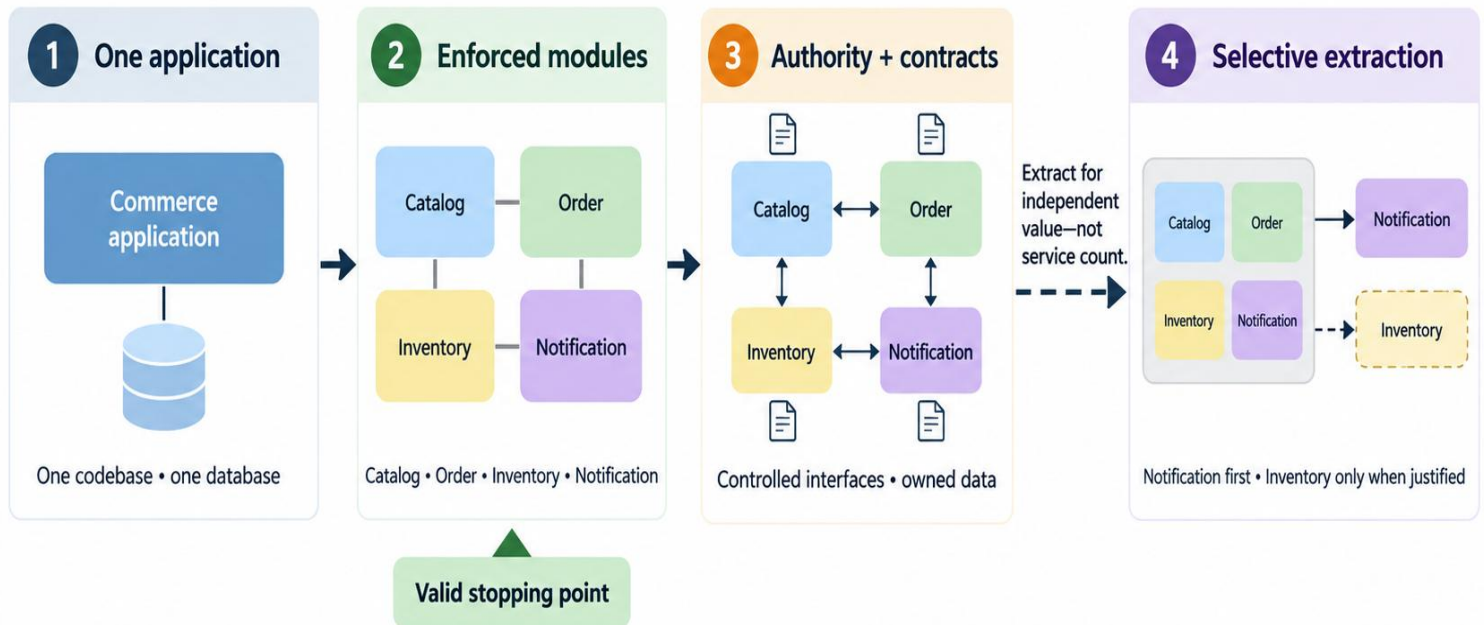
Catalog reads may need aggressive scaling. Notification can often tolerate delayed processing. Order acceptance may need tighter latency and availability. Operational differences can strengthen a boundary, although they should not be the only reason to create one.

A step-by-step decomposition of the example

Assume the system begins as one commerce application with one codebase and one database.

Chapter 2: Finding Effective Service Boundaries

2-4 Evolve Boundaries in Stages



013

Figure 2-5: Progression from one application to modules and then selected services

Step 1: Establish logical modules

Before creating network boundaries, organize the application into modules such as Catalog, Order, Inventory, and Notification. Each module exposes a small interface and does not access another module's private implementation directly.

This step tests whether the proposed responsibilities are understandable. If the team cannot maintain the boundary inside one process, turning it into a network boundary will usually make the problem harder.

Step 2: Assign data authority

Identify which module may create and update each data set:

- Catalog owns product descriptions and visible product information.
- Order owns order lifecycle and customer intent to purchase.
- Inventory owns stock counts and reservations.
- Notification owns delivery attempts and communication preferences relevant to delivery.

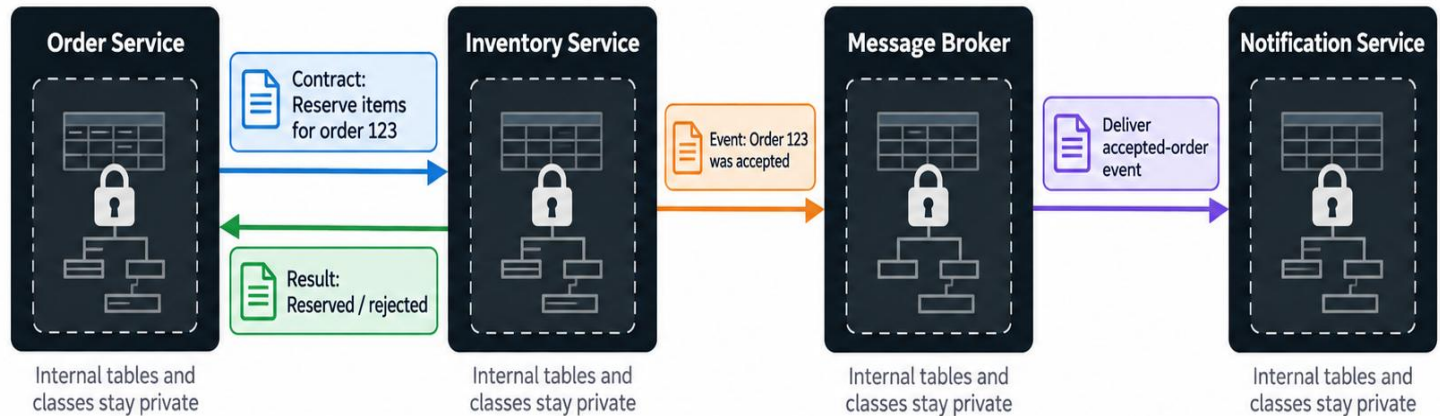
A shared reporting view may read data from several sources, but it is not the authority for those sources.

Step 3: Define the necessary interactions

Order needs to know whether Inventory can reserve stock. Notification needs to know when an order has been accepted. Those needs can be expressed as contracts:

Chapter 2: Finding Effective Service Boundaries

2-4-3 Necessary Interactions as Contracts



014

Figure 2-6: Necessary interactions expressed as contracts

The contract states what one capability needs from another without exposing internal tables or classes.

Step 4: Look for independent value

Extracting Notification may reduce release coordination because message templates and provider integrations can change separately. Extracting Inventory may support independent scaling and specialized ownership, but it also introduces a critical remote call into order processing. The sequence of extraction should reflect value and risk, not a desire to maximize service count.

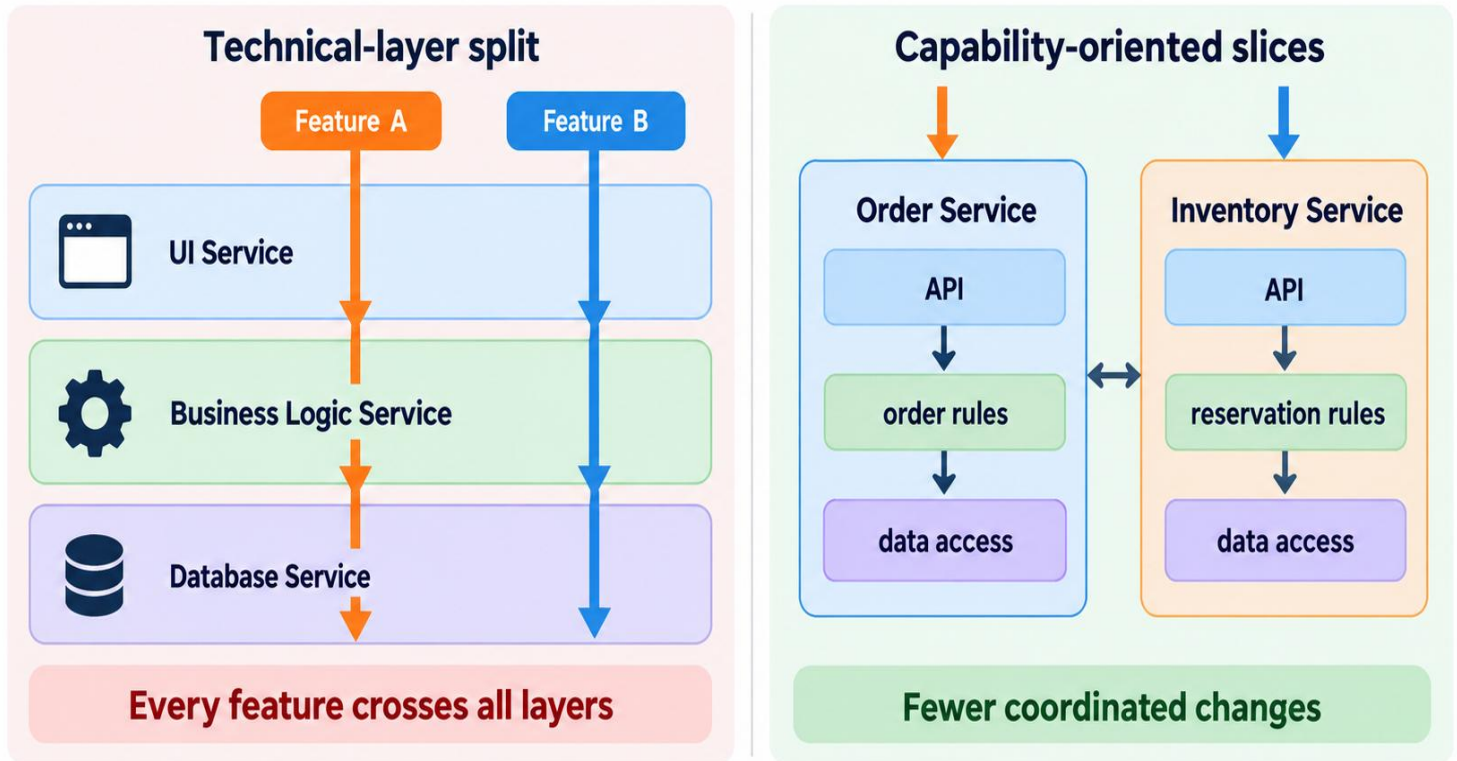
Why technical-layer services usually preserve coupling

A tempting decomposition is:

UI Service -> Business Logic Service -> Database Service

Chapter 2: Finding Effective Service Boundaries

2-5 Technical Layers vs Capability Slices



015

Figure 2-7: Technical-layer services versus capability-oriented vertical slices

This design divides the system by technical layer rather than business capability. Almost every feature still requires changes across all three services. The “Database service” becomes a generic remote data-access layer, and the Business Logic service often remains a large central component.

The result has more network calls and deployments but little independent change. A capability-oriented design is more likely to keep a complete slice of behavior together:

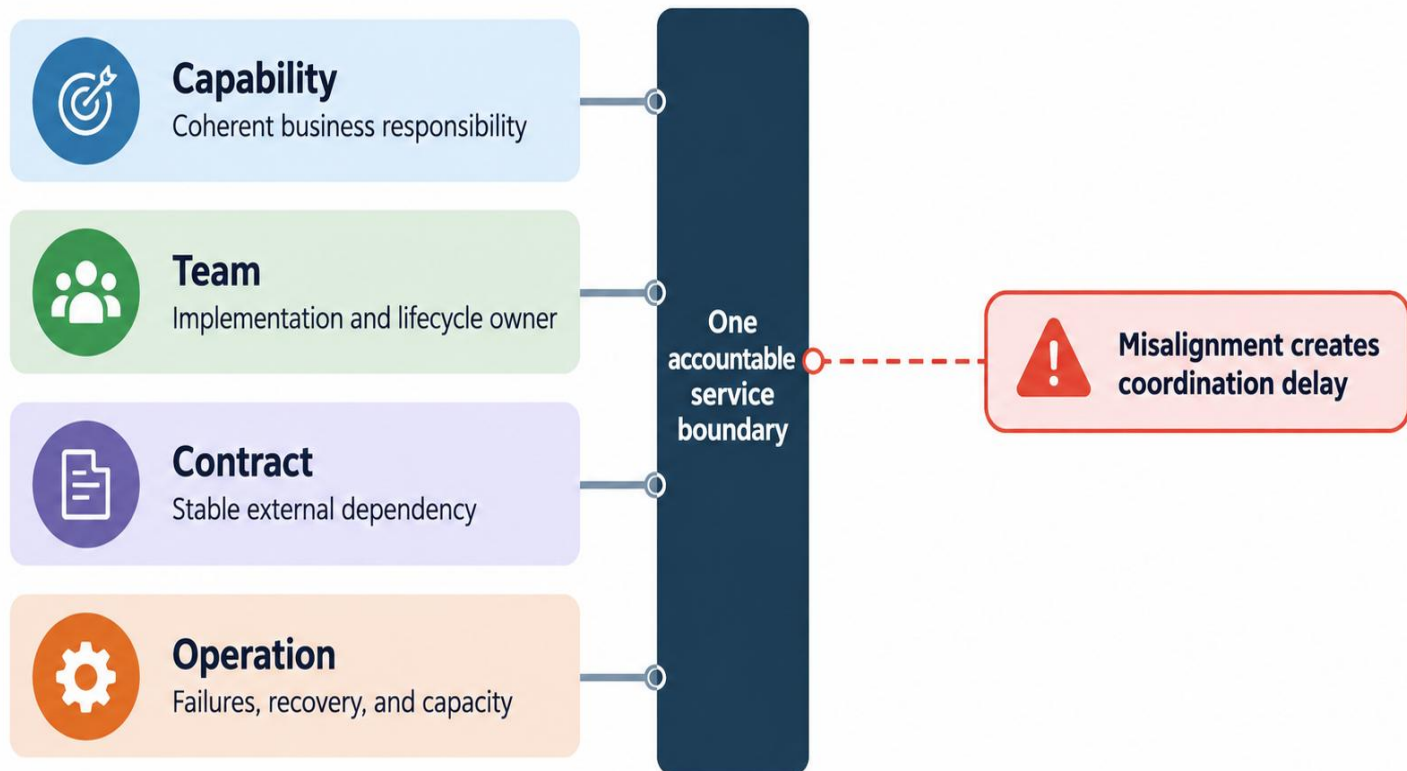
Order Service: API + order rules + order data access
Inventory Service: API + reservation rules + inventory data access

User interfaces and shared infrastructure still exist, but service boundaries are not based only on those layers.

Service, team, contract, and operation should align

A durable boundary has four aligned forms of ownership:

Chapter 2: Finding Effective Service Boundaries 2-6 Align Capability, Team, Contract, and Operation



016

Figure 2-8: Alignment of capability, team, contract, and operation

- The service owns a business capability.
- A team owns the service's implementation and lifecycle.
- The service exposes a contract that others can depend on.
- The team owns the service's operational behavior, including failures and recovery.

Misalignment creates delays. If one team owns code but another team must perform every deployment, local changes are not truly autonomous. If two teams may update the same tables, data authority is unclear. If no team owns after-hours failures, the service is not operationally complete.

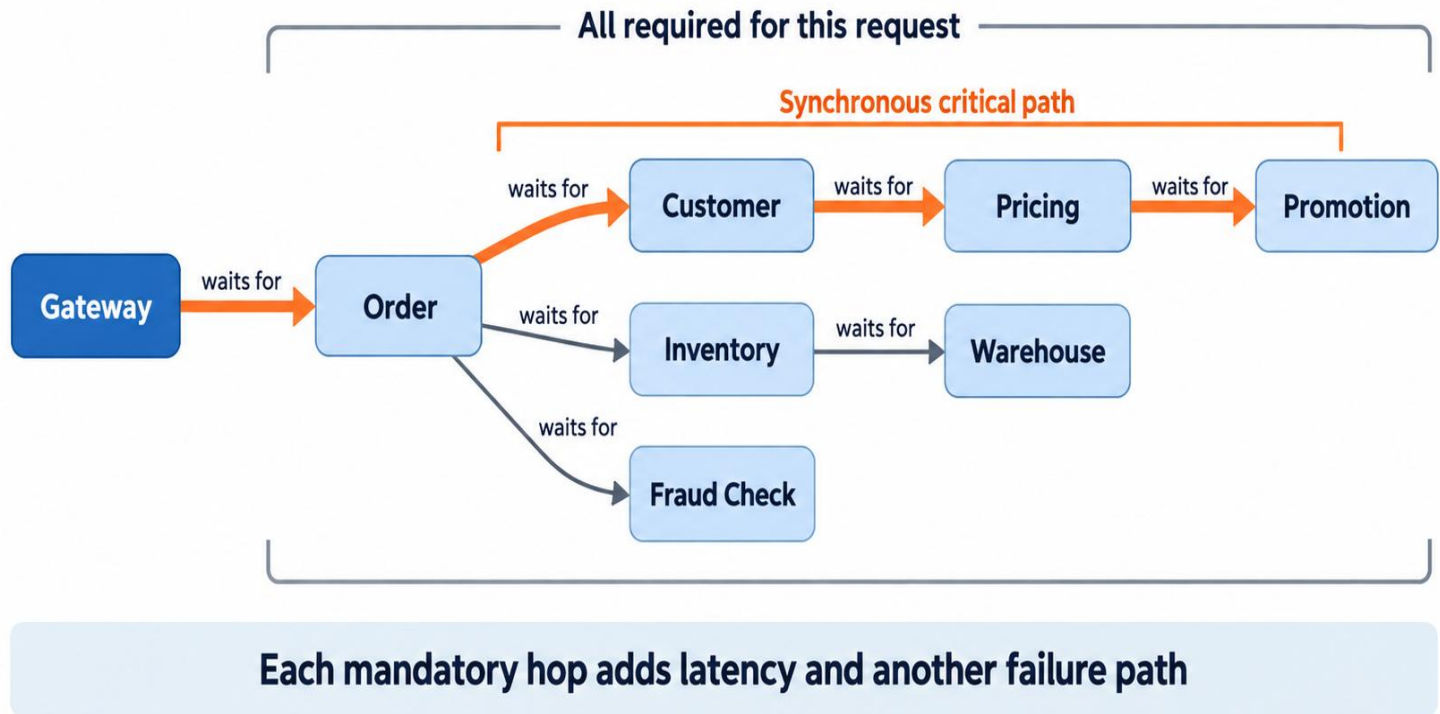
Team boundaries do not need to map one-to-one to services. One team may own several related services. The important point is that each service has one accountable owner and that the team's responsibility is manageable.

Map dependencies before approving a split

A **dependency map** shows which components call or rely on which others. Even a simple directed diagram can reveal dangerous structures:

Chapter 2: Finding Effective Service Boundaries

2-7 Dependency Map and Critical Path



017

Figure 2-9: Dependency map and synchronous critical path

If Order must wait for every dependency, the user request succeeds only when the entire chain is responsive. Each additional synchronous dependency adds latency and another failure path.

Review dependencies with questions such as:

- Does this call need an immediate answer?
- Is the requested information owned by the right service?
- Could the caller use a local copy that is updated by events?
- Does one user request repeatedly cross the same boundary?
- Would a routine feature require coordinated changes in several services?

A call is not automatically wrong because it is synchronous. The concern is whether the chain reflects a real business dependency and whether its failure behavior is acceptable.

Warning signs of a poor boundary

Several recurring symptoms indicate that services are not independently useful:

- **Shared release schedules:** Services must be deployed in a fixed group or sequence for ordinary changes.
- **Frequent cross-service changes:** One feature routinely modifies several service contracts and implementations.
- **Chatty interfaces:** A caller makes many small remote calls to assemble one operation.

- **Shared internal tables:** Several services read or write data structures that one service is supposed to own.
- **Cyclic dependencies:** Service A calls B, B calls C, and C calls A.
- **Unclear ownership:** Incidents and schema changes require negotiation because no team is authoritative.
- **Duplicated rules:** Two services independently decide the same business fact and can disagree.

One symptom may be acceptable temporarily. A cluster of them usually means the boundary should be reconsidered.

Granularity is an outcome, not a quota

There is no rule that one service must fit in one developer's head, contain a particular number of endpoints, or be rewritten within a fixed time. Those heuristics can stimulate discussion, but they are not architectural criteria.

Use practical tests instead:

- Can the owning team explain the service's responsibility in one clear paragraph?
- Do most changes stay inside the service?
- Can the service be deployed without coordinating unrelated releases?
- Does it own the data needed to enforce its rules?
- Is its interface smaller and more stable than its internal implementation?
- Can its failure be contained or handled deliberately?

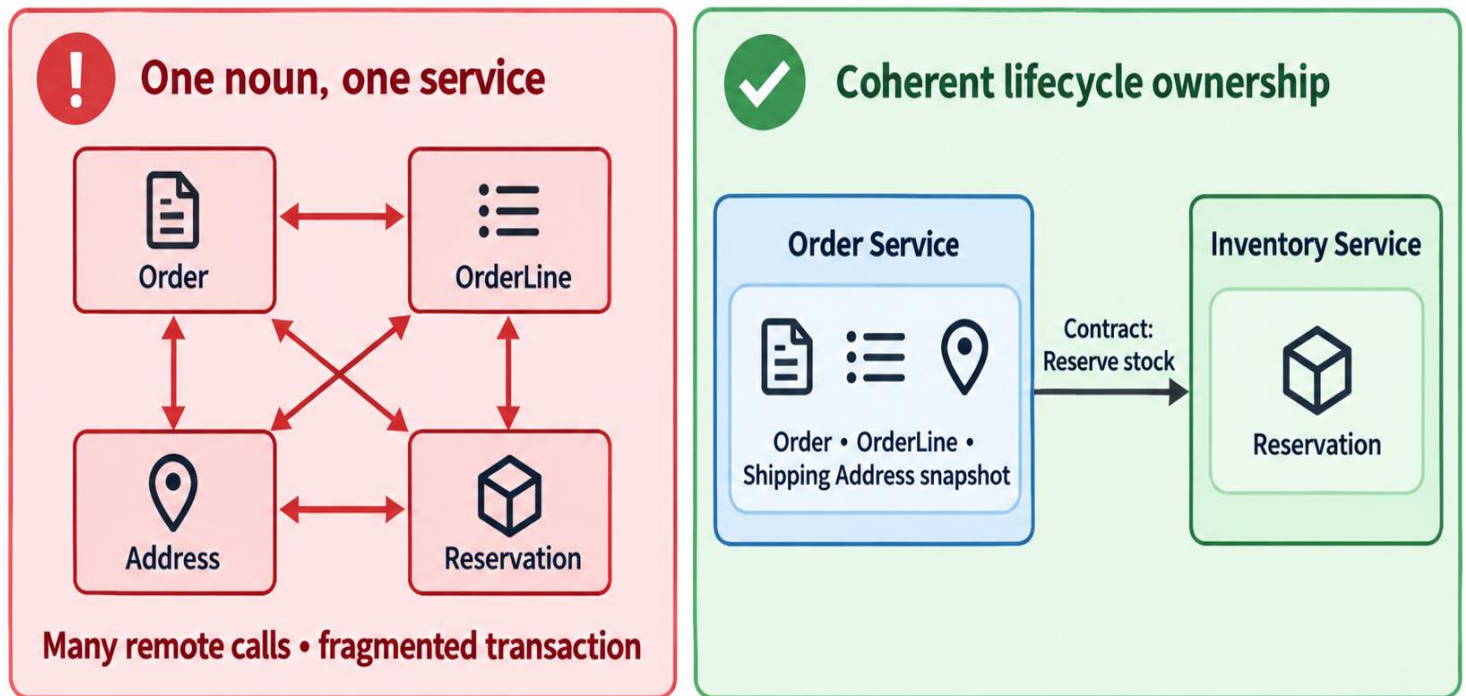
A larger service that passes these tests may be healthier than ten tiny services that fail them.

Avoid the “one noun, one service” mistake

Domain models contain many nouns: Order, OrderLine, Address, Product, Warehouse, Reservation, Customer, and more. Creating one service for every noun leads to constant remote calls and fragmented transactions.

Chapter 2: Finding Effective Service Boundaries

2-10 Avoid One Noun, One Service



018

Figure 2-10: “One noun, one service” fragmentation versus coherent lifecycle ownership

Order and OrderLine usually belong in the same lifecycle. A shipping address captured for an order may be part of the Order service’s record even when a separate Customer service also stores a customer’s current address. The correct boundary follows responsibility and change, not grammatical form.

A boundary is valuable when it reduces coordination without breaking apart behavior that must remain coherent. That principle is more reliable than any target service size.

Chapter Summary

- **A good boundary groups behavior and data that change for related business reasons.**
- High cohesion keeps related rules together; low coupling reduces routine coordination across services.
- Service size is an outcome of responsibility and dependency patterns, not a target number of endpoints or files.
- Frequent cross-service changes, chatty APIs, shared tables, and synchronized deployments are signs of a distributed monolith.

Review Questions

1. How does a business capability differ from a technical layer?
2. What evidence suggests that two responsibilities belong in the same service?
3. Why is creating one service for every domain noun usually harmful?

4. Which warning signs indicate that an existing service boundary should be reconsidered?