

SRE A Visual Guide

Sample Book — Chapters 1 to 2

Foundations, Measurement, Operations, Resilience, and Production Readiness

Introduction

This book is designed for readers who are taking their first steps with SRE, 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

- 1. Read the chapters in order.** The sequence builds the background knowledge step by step, beginning with the meaning of reliability and continuing through measurement, operations, resilience, recovery, and production readiness.
- 2. Review “What You’ll Be Able to Do” before each chapter.** These outcomes identify the capabilities the chapter is intended to build and help you focus on the most important ideas.
- 3. Use the diagrams to understand relationships, processes, and structures.** They support the main text by making sequences, boundaries, dependencies, and cause-and-effect relationships easier to inspect.
- 4. Try the code, commands, calculations, and hands-on examples whenever possible.** Reproducing a small example helps turn recognition into practical understanding; use a safe non-production environment for any action that changes a system.
- 5. Use the Chapter Summary and Review Questions to check retention.** Return to the relevant section when an answer is unclear rather than memorizing a phrase without understanding the reasoning.
- 6. Treat advanced-topic overviews as signposts.** You do not need to memorize them at this stage. Their purpose is to help you recognize concepts that may become important as your responsibilities grow.

Table of Contents

Introduction

How to Use This Book

Chapter 1: SRE as a Reliability Engineering Discipline

- Reliability Is a Product Quality
- What SRE Adds to Operations
- The Reliability Feedback Loop
- Reliability Is Shared Ownership
- SRE and Related Disciplines
- Responsibilities Across the Service Lifecycle
 - During design
 - Before launch
 - During normal operation
 - During an incident
- Common SRE Engagement Models
- Joining a Team: The First Reliability Questions
- Misconceptions That Distort the Foundation

Chapter 2: Understanding a Service and How It Fails

- From User Journey to Request Path
- Services, Components, and Dependencies
- Stateful and Stateless Components
- Data Plane and Control Plane
- Ownership Is Part of the Architecture
- The Main Dimensions of User-Visible Failure
- Partial Failure in Distributed Systems
- Failure Domains and Blast Radius
- Building a Useful Service Map
- Tracing Two Failure Scenarios
 - A slow database
 - An unavailable identity provider
- Hidden Coupling and Correlated Failure

Chapter 1: SRE as a Reliability Engineering Discipline

What You'll Be Able to Do

After completing this chapter, you will be able to:

- Explain why SRE treats reliability as a user-visible engineering outcome rather than only an infrastructure condition.
- Describe the SRE feedback loop and distinguish SRE from adjacent operational disciplines.
- Identify the owners, responsibilities, and first questions needed when joining a service team.

The central idea of SRE is that reliability should be engineered as deliberately as functionality. A team should not rely on hope, heroic individual effort, or an ever-growing collection of manual procedures. It should define the service behavior users need, measure that behavior, operate the service with clear responsibilities, and improve the system using evidence.

Reliability Is a Product Quality

A **service** is a system that provides a useful capability to users or to other systems. An order-processing API, a company login service, and an internal reporting portal are all services, even though their users and technical designs differ.

Reliability is the degree to which a service provides the behavior its users need over time. This definition is broader than whether servers are running. **Availability** describes whether users can obtain an acceptable response; **latency** is the time an operation takes; **correctness** is whether its result is accurate; **durability** is whether accepted data survives; and **freshness** is whether information is current enough. A service may be technically “up” and still be unreliable because it is too slow, returns incorrect results, loses data, or serves information that is no longer fresh enough to be useful.

Consider an order API that accepts a customer's purchase:

- If the API refuses every request, it has an availability problem.
- If it responds after 45 seconds, it has a latency problem even if it eventually succeeds.
- If it reports success but creates the wrong order, it has a correctness problem.
- If a confirmed order disappears after a database failure, it has a durability problem.

- If stock information is several hours old, it has a freshness problem.

Chapter 1: SRE as a Reliability Engineering Discipline

1-1 Reliability as Five User-Visible Qualities

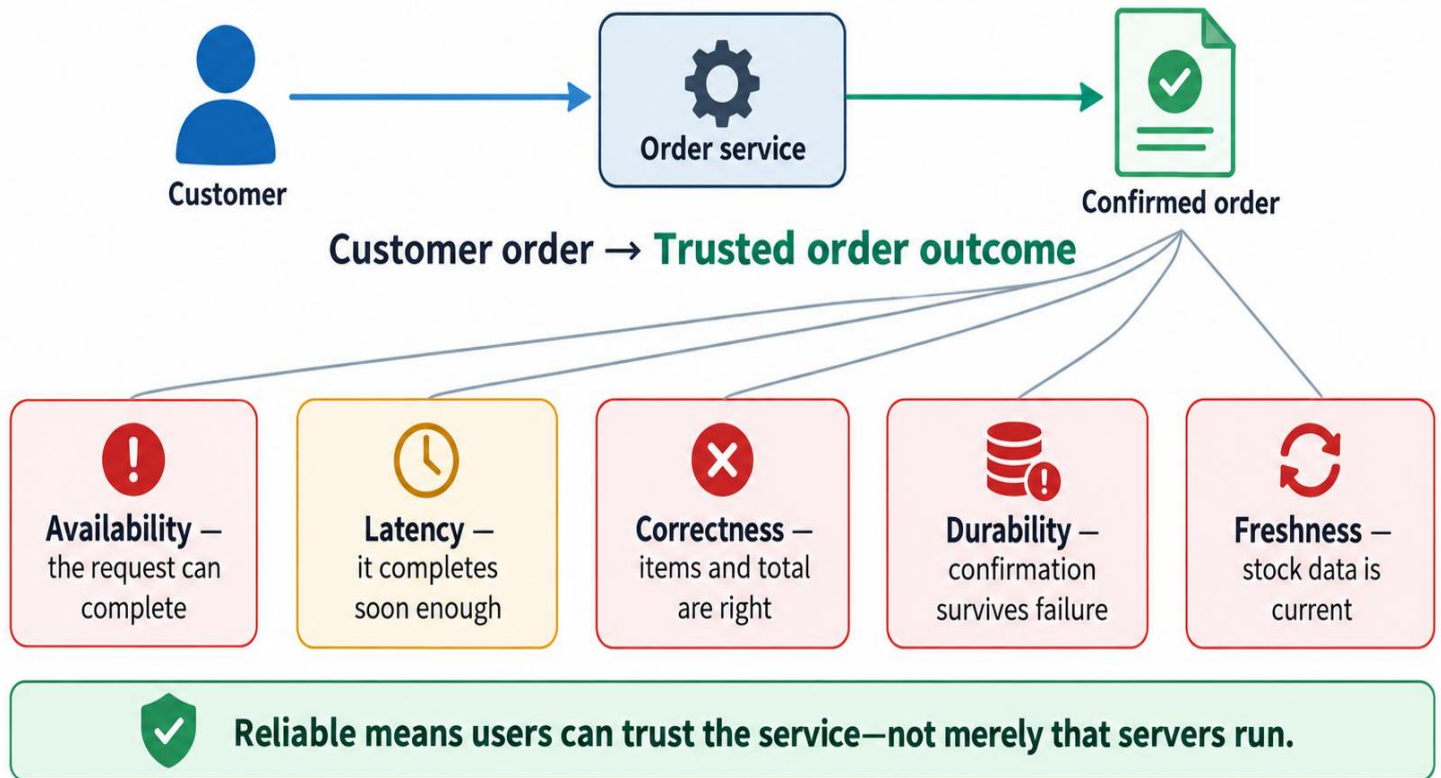


Figure 1-1: Reliability as five user-visible qualities

These are all reliability concerns because they affect whether the user can trust the service.

A useful everyday analogy is a railway. A train service is not reliable merely because the engines start. Passengers care whether trains arrive, whether the journey takes a reasonable time, whether they reach the correct destination, and whether the published schedule reflects reality. In the same way, systems engineers must look beyond machine health and evaluate the complete service experience.

What SRE Adds to Operations

Site Reliability Engineering is the practice of applying software-engineering methods to the operation of services. It treats recurring operational problems as candidates for measurement, redesign, standardization, and carefully controlled automation.

Operations work has always required technical skill. SRE does not imply that traditional operations lacks engineering. The distinction is the explicit operating model: SRE uses measurable reliability objectives and allocates engineering effort to reduce operational risk and repetitive work.

Imagine a team that restarts a failing process by hand several times each week. Manual restarts may be a reasonable short-term mitigation, but they do not scale well. As the number of services, users, and deployments grows, the team must either add people indefinitely or change the system. SRE asks questions such as:

- What condition causes the process to fail?
- How much user impact occurs before the team notices?
- Can the process recover safely without human intervention?
- What safeguard would prevent repeated failure?
- Which change would produce the greatest reduction in risk?

The goal is not to automate every task. The goal is to make recurring operations **predictable, observable, and proportionate to their risk.**

The Reliability Feedback Loop

SRE can be understood as a continuous feedback loop rather than a collection of independent tools.

1. **Define** the service behavior users need.
2. **Measure** that behavior using evidence from real or representative requests.
3. **Operate** the service with clear ownership, monitoring, documentation, and response procedures.
4. **Learn** from incidents, near misses, performance changes, and routine operational work.
5. **Improve** the design, controls, and ways of working, then measure again.

Chapter 1: SRE as a Reliability Engineering Discipline

1-3 The Reliability Feedback Loop



Repeat: improved systems create new evidence and better decisions.

002

Figure 1-2: The reliability feedback loop

This loop resembles maintaining a comfortable room temperature. A thermostat starts with a desired range, observes the current temperature, activates heating or cooling, and checks the result. Without a target, the measurement has no meaning. Without measurement, action becomes guesswork. Without feedback, the system cannot correct itself.

The same logic applies to reliability. A dashboard containing thousands of measurements is not an SRE program if the team has not defined which service behavior matters or what action should follow from the data.

Reliability Is Shared Ownership

SRE is a shared engineering responsibility, even when an organization has a dedicated SRE team. The people who design features, operate infrastructure, define business priorities, protect the service, and support users all influence reliability.

Development teams affect reliability through application design, testing, dependency choices, and deployment behavior. Systems and operations engineers affect it through infrastructure, runtime configuration, access, monitoring, and recovery procedures. Security teams affect identity, secrets, patching, and abuse resistance. Product and business teams decide which user journeys are most important and how much interruption is acceptable.

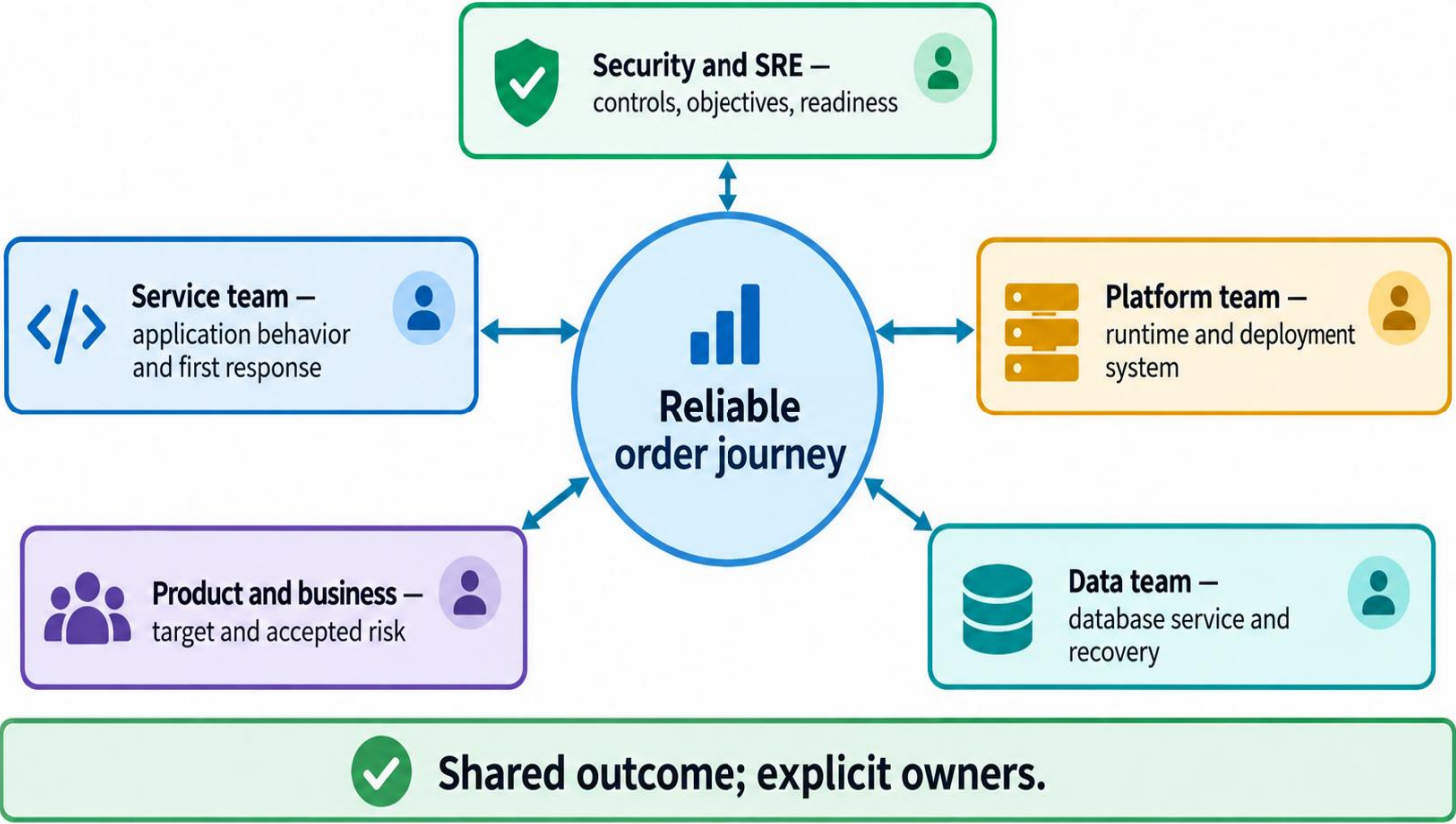
Shared ownership does not mean that everyone is responsible for everything. That usually produces unclear accountability. It means that responsibilities are explicit and coordinated. For example:

- A service team may own application behavior and first response.

- A platform team may own the shared runtime and deployment system.
- A database team may own the managed data service.
- Product leadership may approve the reliability target and accepted risk.
- An SRE team may help define objectives, review readiness, and improve operations.

Chapter 1: SRE as a Reliability Engineering Discipline

1-4 Shared Ownership with Clear Accountability



003

Figure 1-3: Shared ownership without unclear accountability

When an incident occurs, the service owner cannot safely assume that “the infrastructure team will handle it,” and the infrastructure team cannot determine user impact without service context. Reliable operation depends on both.

SRE and Related Disciplines

Organizational labels vary, so the name of a team is less important than its responsibilities and methods. The following comparison shows common emphases rather than strict boundaries.

Discipline	Primary emphasis	Typical contribution to reliability
Traditional operations	Running and supporting production systems	Procedures, infrastructure administration, monitoring, recovery, and continuity
DevOps	Improving collaboration and delivery flow between development and operations	Shared ownership, automation, rapid feedback, and smaller changes
SRE	Managing service reliability through measurable objectives and engineering	Service-level indicators, objectives, error budgets, incident response, and operational improvement
Platform engineering	Building internal products that make delivery and operation easier for other teams	Standard runtime environments, deployment workflows, self-service capabilities, and common controls
Production engineering	Bridging application engineering and production operation	Performance, debugging, production architecture, and operational software; scope varies by organization

DevOps is a broad culture and set of practices, while SRE is a concrete approach to achieving reliable operations within that culture. Platform engineering may provide the common systems that make SRE practices easier to apply. Production engineering often overlaps with SRE, especially in organizations that use different names for similar work.

A common mistake is to rename an operations team “SRE” without changing its authority, objectives, workload, or engineering capacity. The label alone does not create SRE. A team needs time and permission to improve systems, not only a queue of support requests and pages.

Responsibilities Across the Service Lifecycle

SRE work begins before a launch and continues throughout the service's life.

During design

The reliability contribution is to make risks visible early. Engineers clarify the critical user journeys, identify dependencies, consider failure behavior, and challenge assumptions such as “this dependency is always available.” The purpose is not to demand an elaborate architecture. It is to prevent avoidable uncertainty from becoming an emergency later.

Before launch

A **rollback** returns a change to a previously known state. The team confirms ownership, monitoring, support coverage, access, rollback capability, and recovery expectations. A service that has no clear owner or usable operational documentation is not ready merely because its functional tests pass.

During normal operation

Engineers track user-visible reliability, investigate trends, improve alerts, keep documentation accurate, and reduce recurring manual work. Normal operation is when most reliability improvement should occur; waiting for an incident makes every change more urgent and riskier.

During an incident

The priority changes from improvement to harm reduction. Engineers establish coordination, determine user impact, contain the problem, restore acceptable service, and communicate clearly. Deep redesign can wait until the service is stable.

Chapter 1: SRE as a Reliability Engineering Discipline

1-6 Reliability Across the Service Lifecycle

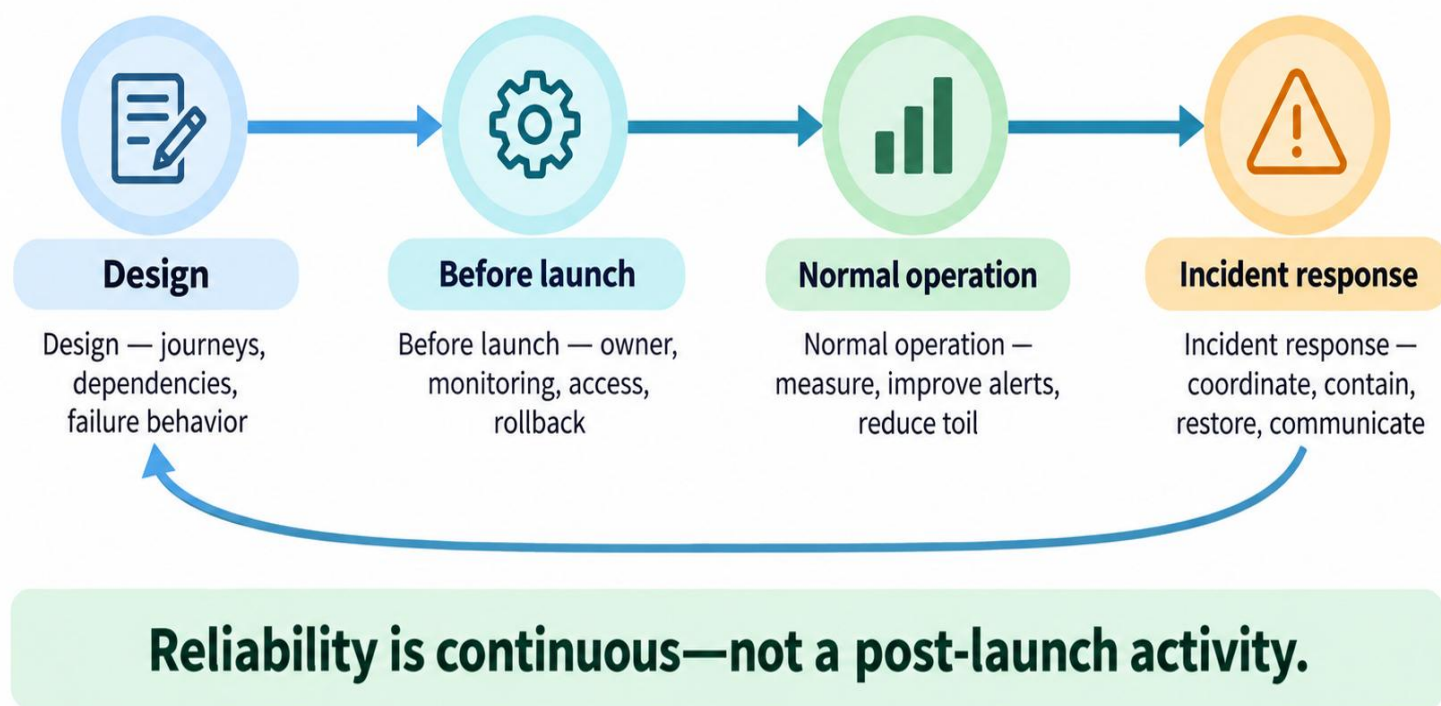


Figure 1-4: Reliability responsibilities across the service lifecycle

Common SRE Engagement Models

Organizations arrange SRE work in several ways. None is universally best.

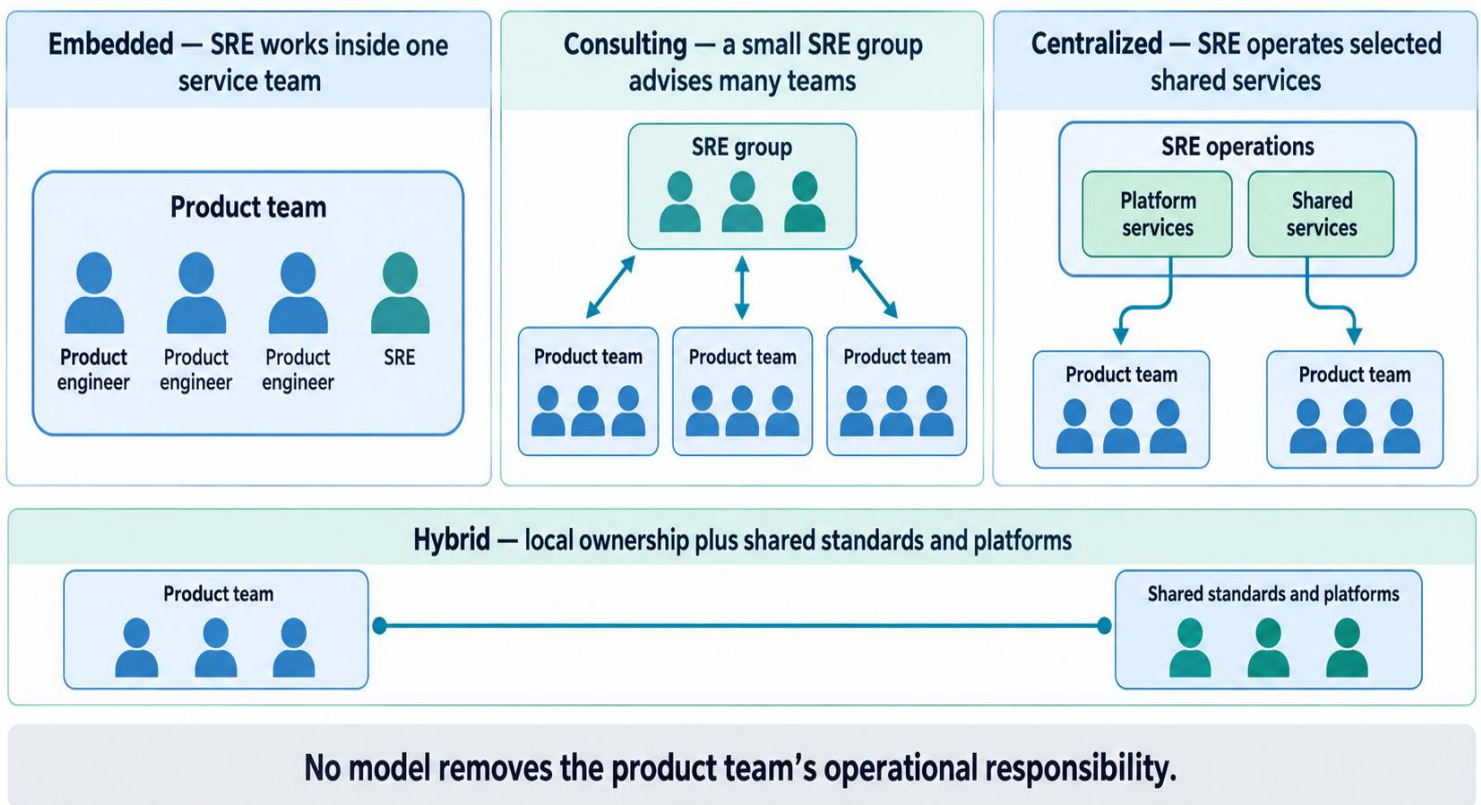
An **embedded model** places an SRE or reliability engineer directly within a product or service team. This creates strong local context but may make standards inconsistent across teams.

A **consulting model** uses a small reliability group to advise many service teams, review designs, and help with difficult incidents. This scales expertise but requires product teams to retain operational ownership.

A **centralized model** places operation of selected services in a dedicated SRE organization. This can build deep operational capability, but handoffs must be managed carefully so the development team does not become disconnected from production behavior.

Chapter 1: SRE as a Reliability Engineering Discipline

1-7 Common SRE Engagement Models



005

Figure 1-5: Embedded, consulting, centralized, and hybrid SRE models

Many organizations use a hybrid. For example, a central group may provide standards and incident coordination while embedded engineers focus on the highest-risk services. The correct choice depends on service criticality, organization size, available skills, and how much context each team needs.

Joining a Team: The First Reliability Questions

Suppose a systems engineer joins the team responsible for a customer-facing order API. The first task should not be to redesign the monitoring system. The engineer first needs a reliable mental model of the service.

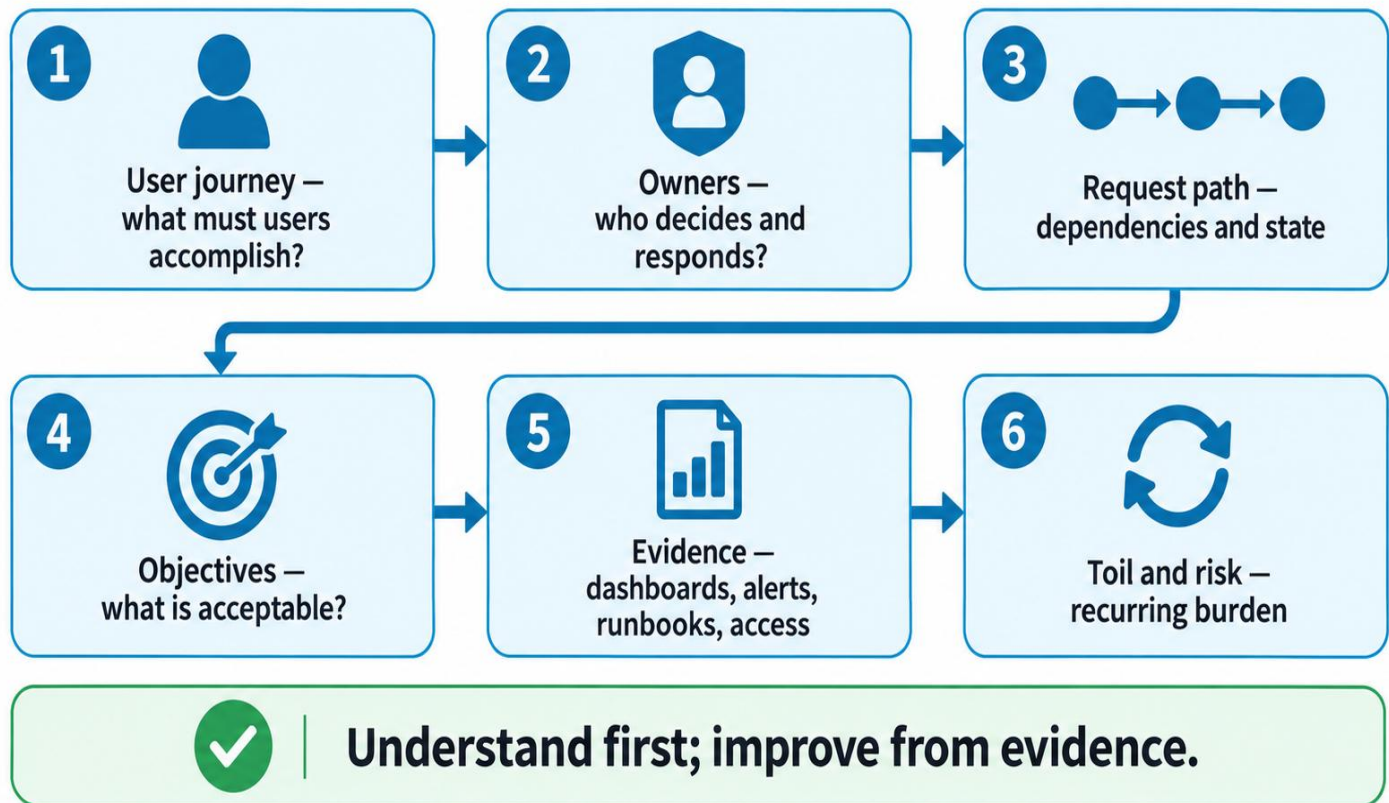
A practical sequence is:

1. Identify the main users and the most important user journey.
2. Find the service owner, support rotation, and escalation path.
3. Trace a normal request from the user to its dependencies and back.
4. Learn what "successful," "too slow," and "unavailable" mean for this service.
5. Review current reliability objectives, recent incidents, and known risks.
6. Locate the main dashboards, alerts, runbooks, deployment history, and recovery procedures.

7. Ask which recurring operational tasks consume the most time or create the most risk.

Chapter 1: SRE as a Reliability Engineering Discipline

1-8 First Reliability Questions When Joining a Team



006

Figure 1-6: First reliability questions when joining a team

For the order API, the most important journey may be “submit a valid order and receive a confirmed order identifier.” That immediately focuses attention on the API, identity service, inventory data, payment dependency, and order database. A less important administrative report should not automatically receive the same operational priority.

Misconceptions That Distort the Foundation

SRE is not simply monitoring. Monitoring supplies evidence, but SRE also defines objectives, manages risk, coordinates response, and changes systems.

SRE is not simply on-call staffing. An on-call rotation without engineering improvement can become a mechanism for repeatedly absorbing the same failures.

SRE is not a promise of zero incidents. Complex systems change, dependencies fail, and unexpected interactions occur. The discipline is designed to limit impact, restore service effectively, and reduce recurrence.

SRE is not owned only by a specialist team. A specialist can guide and enable reliability work, but the teams making product and architecture decisions must participate in the tradeoffs.

The foundation is therefore straightforward: **reliability is a user-facing property, SRE makes it measurable and manageable, and the work succeeds only when ownership and feedback are explicit.**

Chapter Summary

- Reliability includes availability, latency, correctness, durability, and freshness as experienced by users.
- SRE applies measurement, software engineering, operational discipline, learning, and improvement in a repeating loop.
- Reliability is shared, but effective operation requires explicit accountability and bounded responsibilities.

Review Questions

1. Why can a service be running while still being unreliable for users?
2. How does the Define–Measure–Operate–Learn–Improve loop change reliability work?
3. Which ownership and service questions should a new engineer clarify first?

Chapter 2: Understanding a Service and How It Fails

What You'll Be Able to Do

After completing this chapter, you will be able to:

- Trace a user journey through the technical request path and its dependencies.
- Distinguish stateful and stateless components, data and control planes, and different dependency roles.
- Identify partial failure, failure domains, hidden coupling, and blast radius on a service map.

Reliability work starts with a model of how a user receives value and where that path can break. Without that model, teams tend to monitor individual machines, fix the most visible symptom, and overlook dependencies that actually determine the user experience.

From User Journey to Request Path

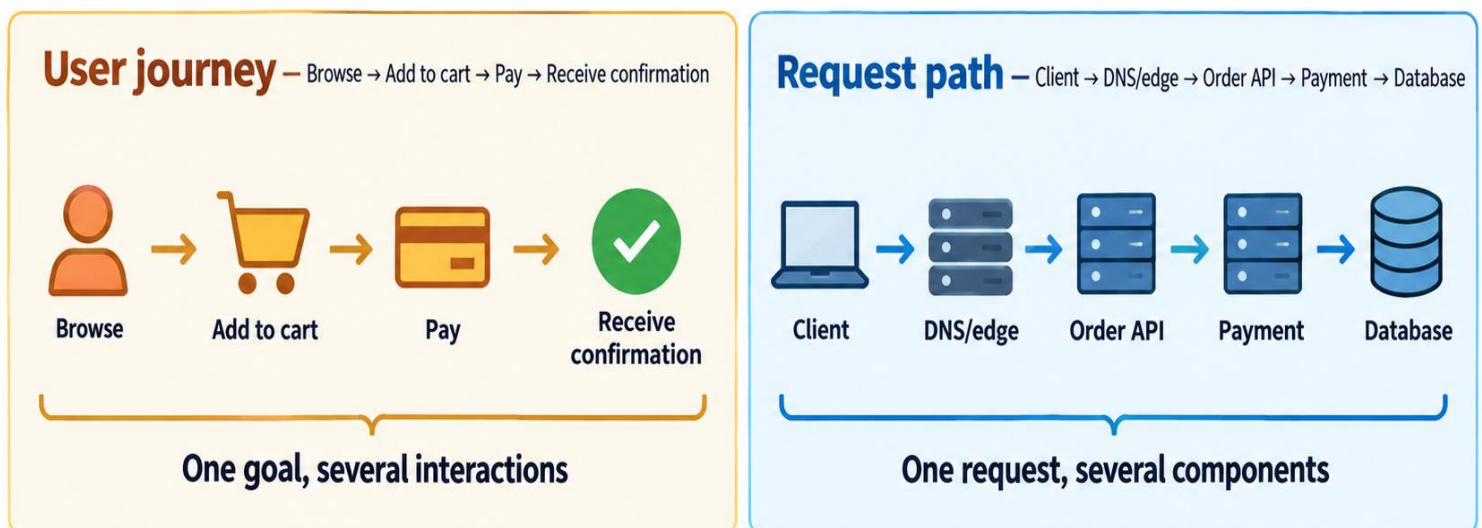
A **user journey** is a sequence of interactions through which a user completes a meaningful goal. “Place an order” is a user journey; it may include signing in, checking stock, submitting payment details, creating the order, and receiving confirmation.

A **request path** is the technical route a particular request follows through the system. One step in the order journey might follow this path:

```
Customer
-> DNS
-> Edge or load balancer
-> Order API
    -> Identity service
    -> Inventory database
    -> Payment provider
    -> Order database
-> Confirmation response
```

Chapter 2: Understanding a Service and How It Fails

2-1 User Journey versus Technical Request Path



Measure the boundary that matches the user outcome.

Figure 2-1: User journey versus technical request path

DNS, or the Domain Name System, translates a service name into the network address a client should contact. An **edge service** is a component near the boundary of a system that may terminate network connections, enforce policy, or route traffic. A **load balancer** distributes requests among service instances.

The diagram is intentionally simple. A real request may also pass through firewalls, proxies, service meshes, queues, caches, and shared platform services. The useful model is not the one with the most boxes. It is the smallest model that explains how value is delivered and where meaningful failure can occur.

A user journey and a request path are related but not identical. The journey may span several requests and include human decisions. A single request path shows the components involved in one technical interaction. SRE uses both views: the journey defines what matters, and the request path helps locate risk and evidence.

Services, Components, and Dependencies

A **component** is a distinct part of a service that performs a technical function, such as an API process, a queue, or a database. A **dependency** is another component or service whose behavior affects the service being examined.

Dependencies may be internal or external. The order API may depend on an identity service managed by another internal team and a payment provider managed by another company. Ownership changes how the team can respond, but it does not remove the dependency from the reliability model.

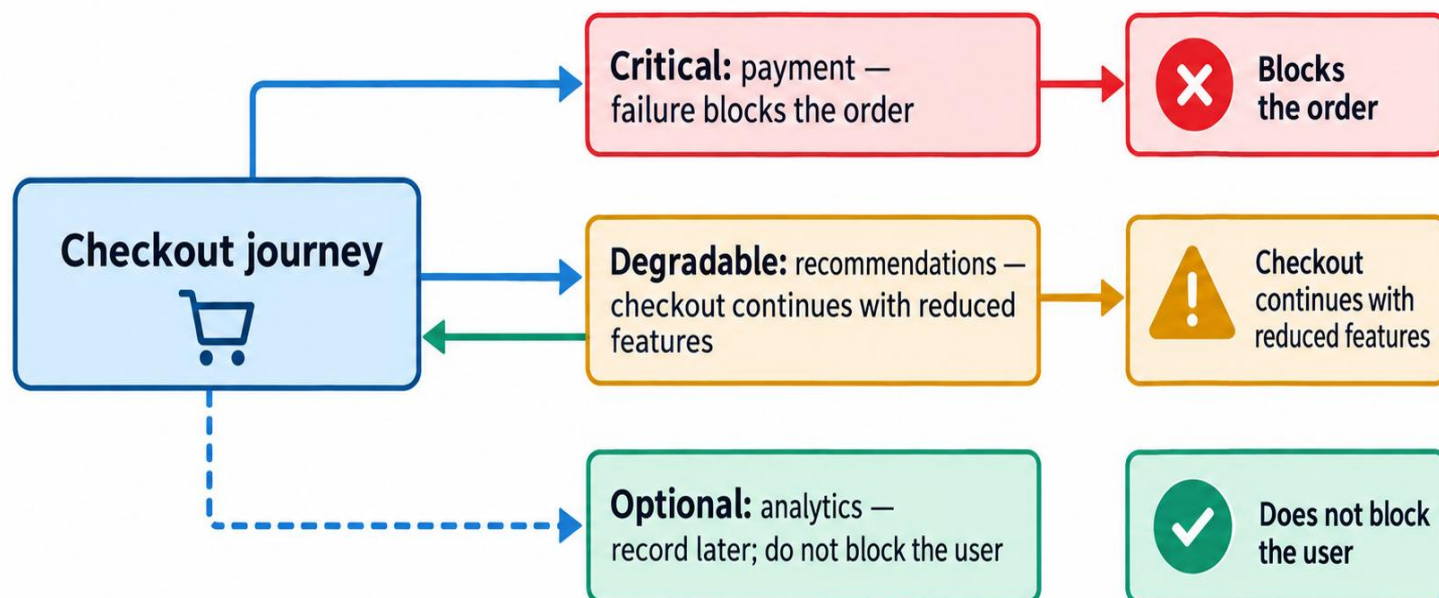
A useful test is: **If this component becomes slow, unavailable, or incorrect, can the user journey still succeed?** The answer helps determine dependency criticality.

Dependencies can be classified in practical terms:

- A **critical dependency** must work for the core journey to succeed.
- A **degradable dependency** can fail while the service provides reduced but acceptable behavior.
- An **optional dependency** supports a secondary feature and should not prevent the core journey.

Chapter 2: Understanding a Service and How It Fails

2-2 Critical, Degradable, and Optional Dependencies



Classification depends on failure behavior, not the dependency's name.

008

Figure 2-2: Critical, degradable, and optional dependencies

For example, payment authorization may be critical to submitting a paid order. A recommendation service may be degradable: the checkout can proceed without product suggestions. An analytics event may be optional if it can be buffered or dropped without affecting the transaction.

This classification should reflect actual behavior, not design intent. A supposedly optional analytics call is effectively critical if the application waits indefinitely for it before returning a response.

Stateful and Stateless Components

A **stateful component** retains information that must survive beyond one request, while a **stateless component** can handle a request without relying on local persistent information from earlier requests.

An order database is stateful because the stored orders must persist. An API instance is often designed to be stateless if any healthy instance can handle the next request using shared storage or request data.

Chapter 2: Understanding a Service and How It Fails

2-3 Stateful versus Stateless Components

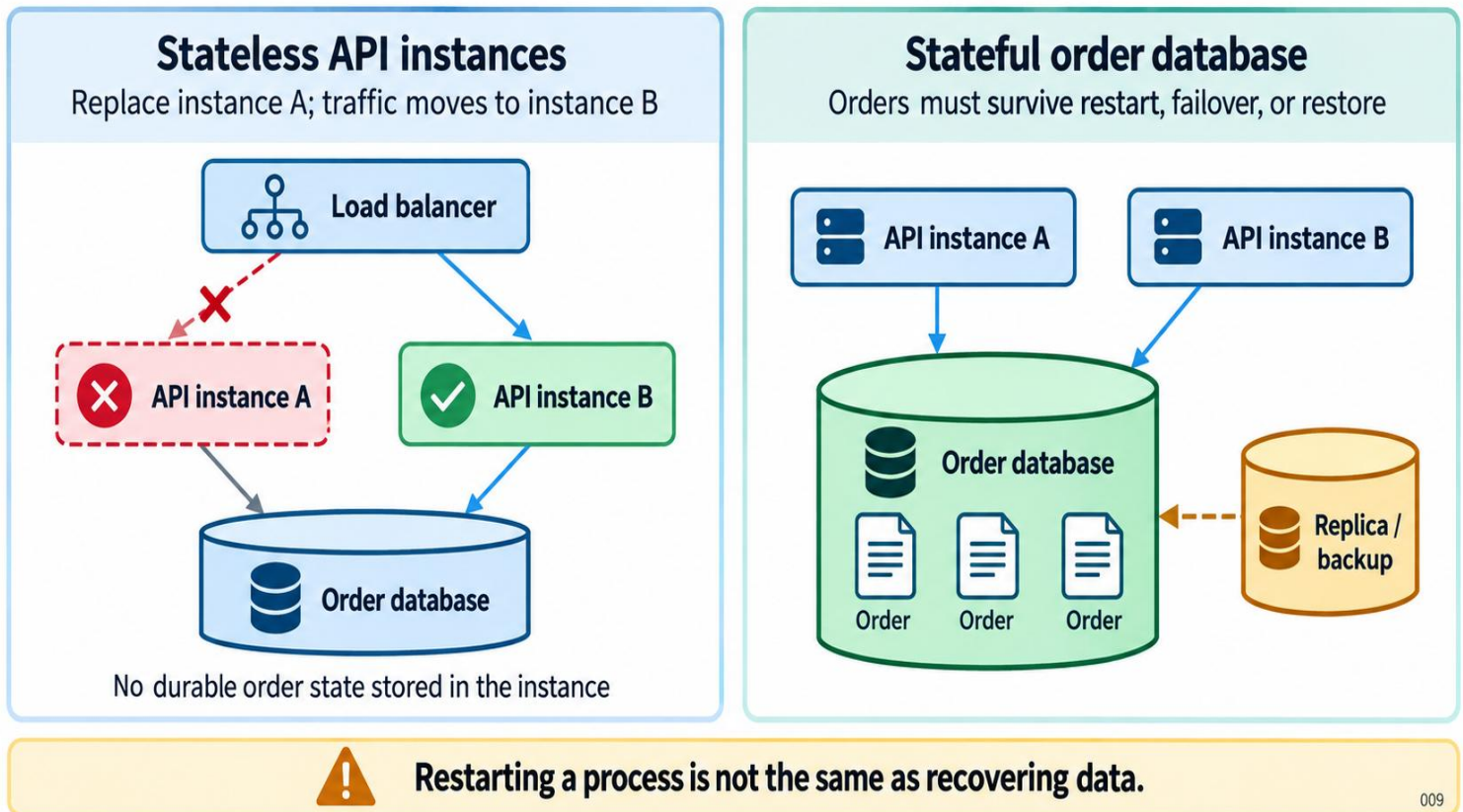


Figure 2-3: Stateful versus stateless components

Stateless components are usually easier to replace or scale horizontally, but “stateless” does not mean “independent.” An API instance may have no persistent local data yet remain completely dependent on a stateful database, identity system, or configuration service.

State changes the reliability questions:

- Can data be lost or corrupted?
- Can another instance take over safely?
- What happens when two writers disagree?
- How is state restored after failure?
- Does recovery preserve the user's confirmed action?

At this stage, the important distinction is that restarting a stateless worker and recovering a stateful data store are fundamentally different operations. The latter requires stronger attention to integrity and persistence.

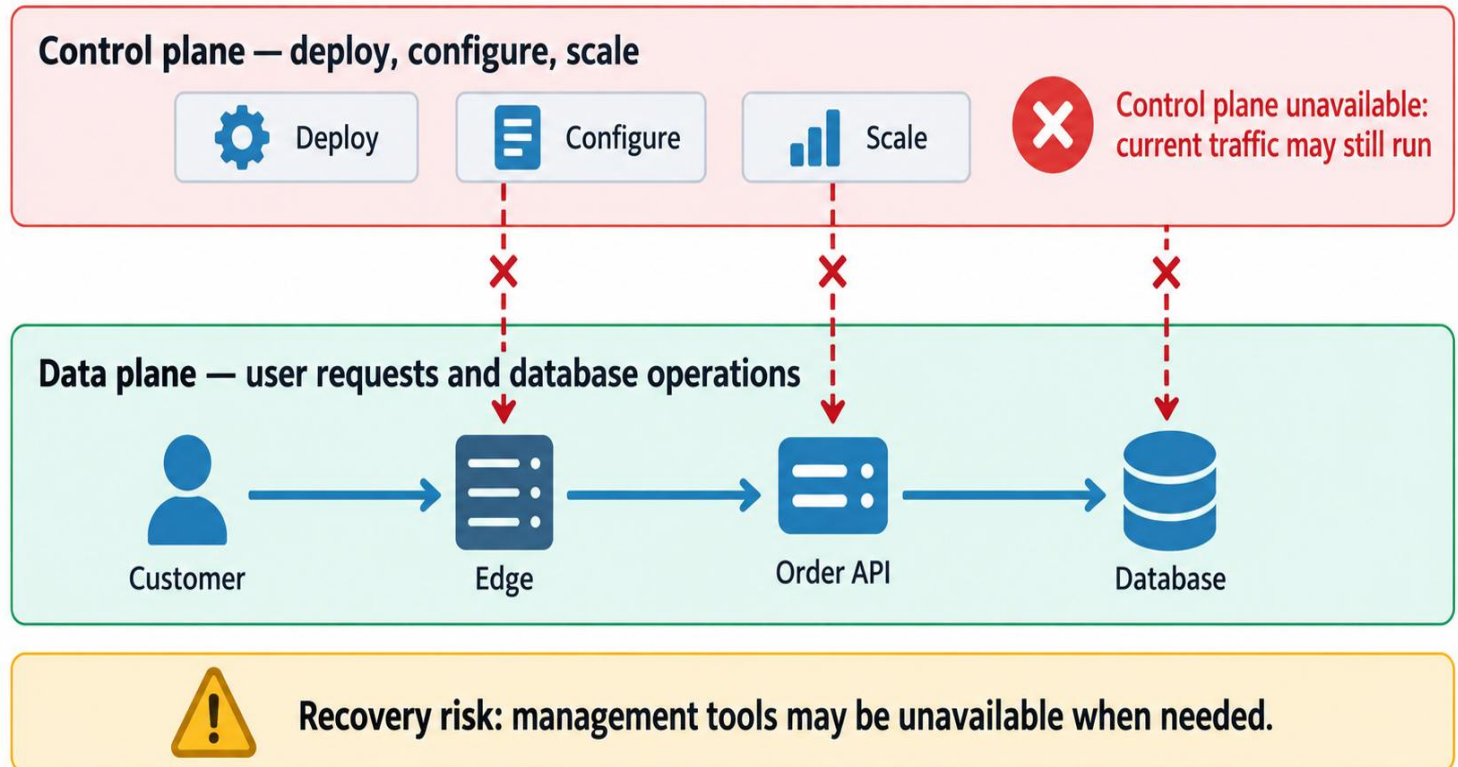
Data Plane and Control Plane

The **data plane** is the part of a system that handles normal user or workload traffic. The **control plane** is the part that configures, coordinates, or manages how the data plane operates.

For an order service, API requests and database operations are data-plane activity. A deployment controller, configuration service, or administrative console may be part of the control plane.

Chapter 2: Understanding a Service and How It Fails

2-4 Data Plane and Control Plane



010

Figure 2-4: Data plane and control plane

This distinction matters because the planes can fail differently. A control-plane outage may leave current API traffic working while preventing deployments, scaling, or configuration changes. A data-plane outage immediately harms user requests even if management tools remain healthy.

During an incident, engineers sometimes discover that the control plane needed for recovery depends on the same failing infrastructure as the service. This is a form of hidden coupling. The lesson is not that every control plane must be independent in every system. It is that recovery assumptions should be visible and tested against likely failures.

Ownership Is Part of the Architecture

A service map that shows components but not owners is incomplete. **Service ownership identifies the team or role accountable for operating and changing a service.** Dependency ownership identifies who can diagnose and act on a supporting component.

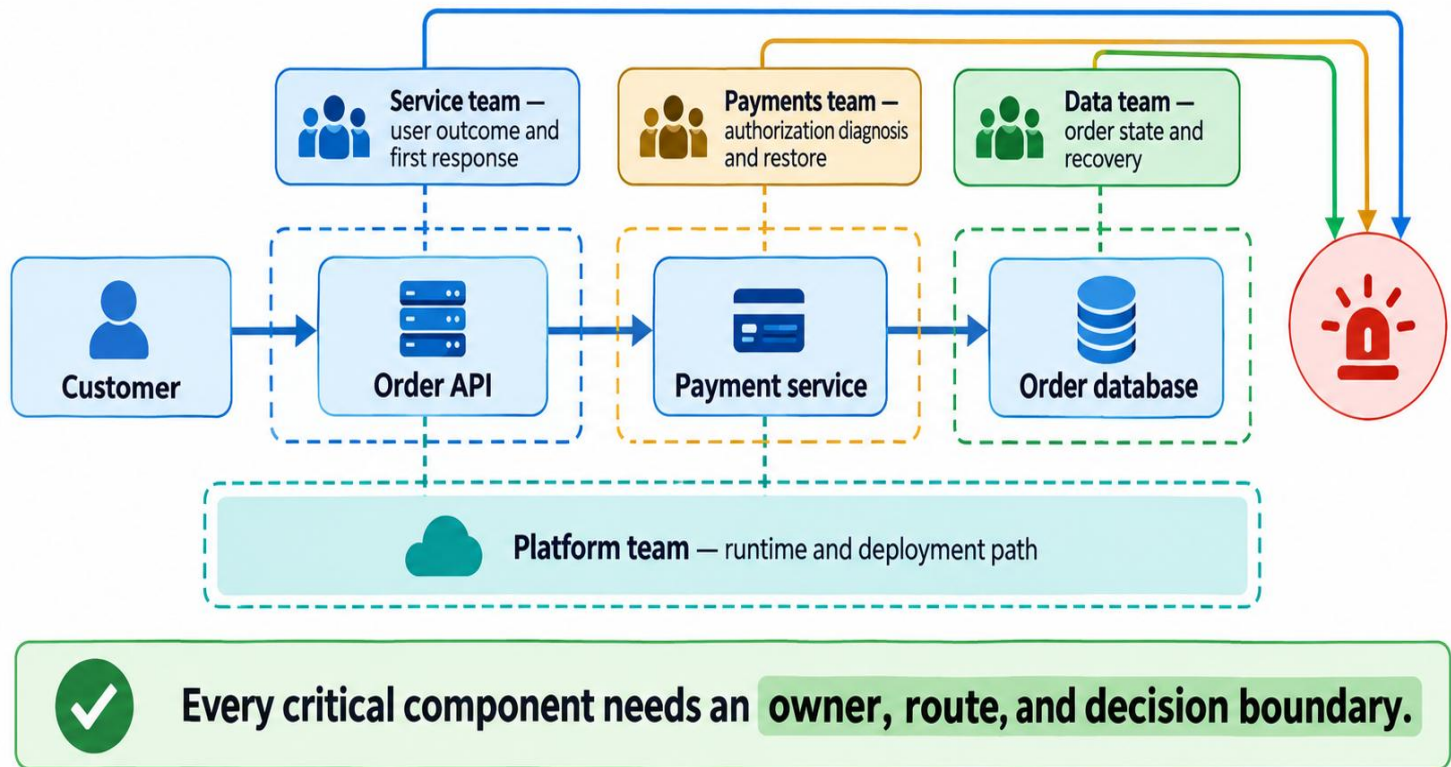
For each critical component, a beginner should be able to find:

- the owning team;
- the current support or escalation route;
- the service documentation;
- the relevant dashboard or status source;

- the expected support boundary.

Chapter 2: Understanding a Service and How It Fails

2-5 Ownership as Part of the Architecture



011

Figure 2-5: Ownership as part of the architecture

A **fallback** is an alternate path or reduced behavior used when the normal path is unavailable. An ownership boundary is not a wall. If the identity service fails, the order team still owns communication about the order journey and may need to implement a safe fallback. The identity team owns diagnosis and restoration of its service. Clear boundaries allow work to proceed in parallel without abandoning the end-to-end user outcome.

The Main Dimensions of User-Visible Failure

Reliability is multi-dimensional. The following dimensions are commonly useful because they describe distinct ways a service can fail from the user's perspective.

Dimension	Question from the user's perspective	Order API example
Availability	Can I use the service when I need it?	The API accepts valid order requests.
Latency	Does the service respond soon enough?	Confirmation arrives within an acceptable time.
Correctness	Is the result accurate?	The order contains the requested items and total.
Durability	Does confirmed information remain intact?	A confirmed order is not lost after a failure.
Freshness	Is the information recent enough to be useful?	Stock status reflects recent inventory changes.

Availability is the proportion of valid attempts for which the service provides an acceptable response. **Latency** is the time taken to complete an operation. **Correctness** describes whether the output or effect matches the intended result. **Durability** describes whether accepted data survives expected failures. **Freshness** describes how current the information is relative to the user's need.

These dimensions interact. A service might preserve availability by returning cached stock data, but the data may be stale. That can be an acceptable tradeoff for product browsing and an unacceptable one for final order confirmation. Reliability must therefore be connected to a specific journey and decision.

Partial Failure in Distributed Systems

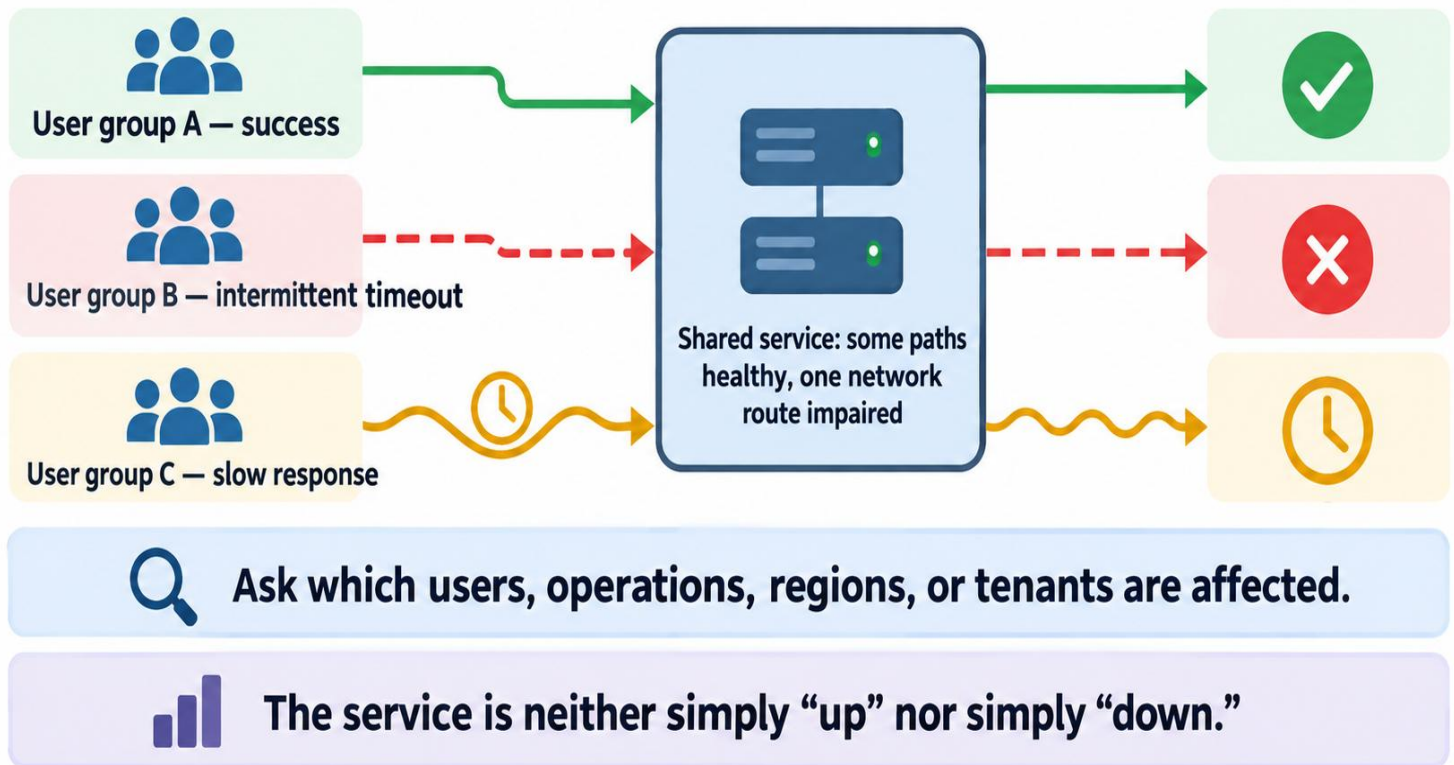
A **distributed system** is a system whose components communicate across process or network boundaries. A **partial failure** occurs when some components or communications fail while others continue to work.

Partial failure is difficult because there may be no single, obvious system state. The API process may be healthy, the database may be accepting connections, and one network route between them may be dropping half the packets. Some customers may succeed while others fail. Retries may hide the issue for a time and then increase load.

An everyday analogy is a group conversation over several phone calls. One person can hear the speaker but cannot be heard. Another receives messages several seconds late. The group has not failed completely, but coordination is unreliable. Distributed services behave similarly: each component observes only part of the situation.

Chapter 2: Understanding a Service and How It Fails

2-7 Partial Failure in a Distributed System



012

Figure 2-6: Partial failure in a distributed system

The practical response is to avoid binary thinking. “The service is up” and “the service is down” are often insufficient. Engineers should ask:

- Which users, operations, regions, or **tenants**—distinct customer or account groups sharing the service—are affected?
- Is the problem complete, intermittent, or slow?
- Which path differs between successful and failed requests?
- Are retries changing the apparent impact?
- Is data still correct and durable despite request failures?

Failure Domains and Blast Radius

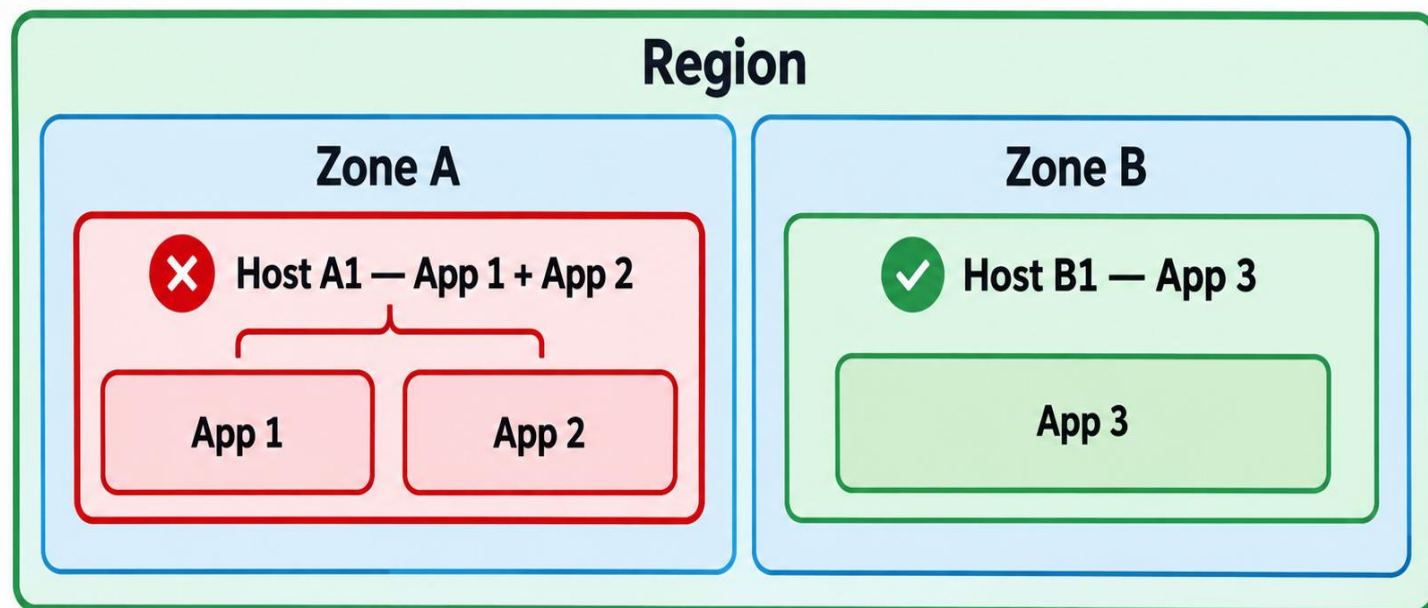
A **failure domain** is a set of components likely to fail together because they share a resource or boundary. Examples include a server rack, power supply, network zone, cloud region, database cluster, administrative account, or deployment group.

A **single point of failure** is a component whose failure can stop the required service behavior because no effective alternative path exists. A single database instance may be an obvious example. A less obvious example is one shared identity service used by every operational tool and recovery workflow.

Blast radius is the scope of users, requests, data, or components affected by a failure or change. Two designs may have the same probability of failure but very different consequences. If a bad configuration is deployed to all customers at once, the blast radius is large. If it is first applied to one small group, the same defect can be detected with less harm.

Chapter 2: Understanding a Service and How It Fails

2-8 Failure Domains and Blast Radius



**Host failure affects App 1 and App 2;
App 3 continues.**

**App copies are not independent
when they share a failure domain.**

013

Figure 2-7: Failure domains and blast radius

Failure-domain analysis asks whether apparently separate components share something that can fail at once. Two application instances do not provide useful redundancy if they run on the same host. Two regional systems may still share one global configuration store. Separate services may depend on one common network path or credential authority.

Building a Useful Service Map

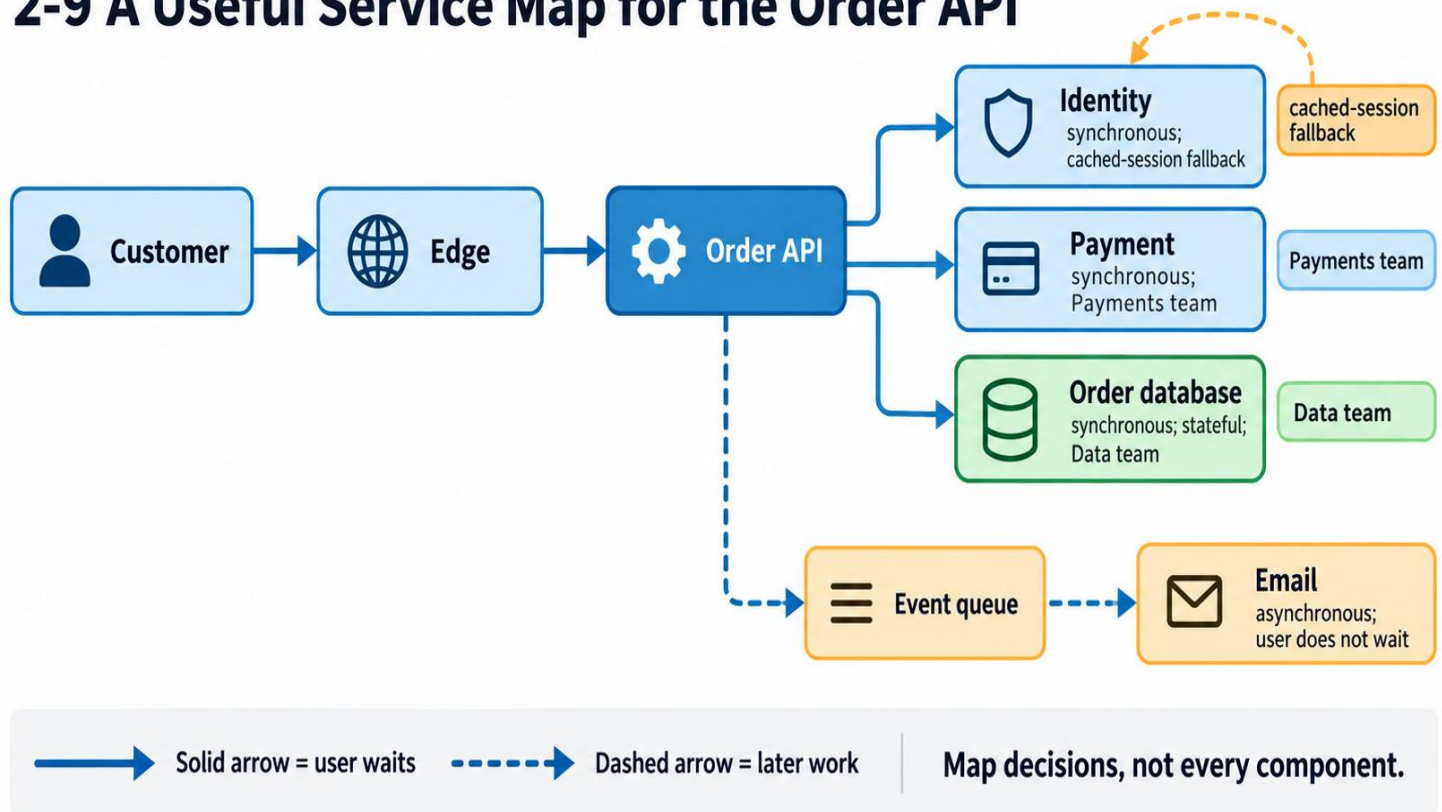
A service map is valuable when it supports decisions, not when it becomes a perfect inventory diagram. A practical mapping process is:

1. Write the user goal in one sentence.
2. Identify the entry point for that journey.
3. Trace the normal path through components and dependencies.
4. Mark which calls are synchronous, meaning the user waits for them, and which are asynchronous, meaning work can continue later.
5. Add owners for critical components.
6. Mark stateful components and data stores.
7. Mark known fallback paths and what behavior they preserve.
8. Group components by shared failure domain.

9. Record important uncertainties as questions rather than inventing answers.

Chapter 2: Understanding a Service and How It Fails

2-9 A Useful Service Map for the Order API



014

Figure 2-8: A useful service map for the order API

A **synchronous call** must complete before the current operation can continue. An **asynchronous operation** is accepted for processing without requiring all work to finish before the user receives the initial response.

For the order API, payment authorization is likely synchronous because the user needs to know whether payment was accepted. Sending a confirmation email may be asynchronous because the order can be confirmed before the email is delivered. If email delivery is incorrectly placed in the synchronous request path, a mail-provider outage can prevent order creation. The map makes that coupling visible.

Tracing Two Failure Scenarios

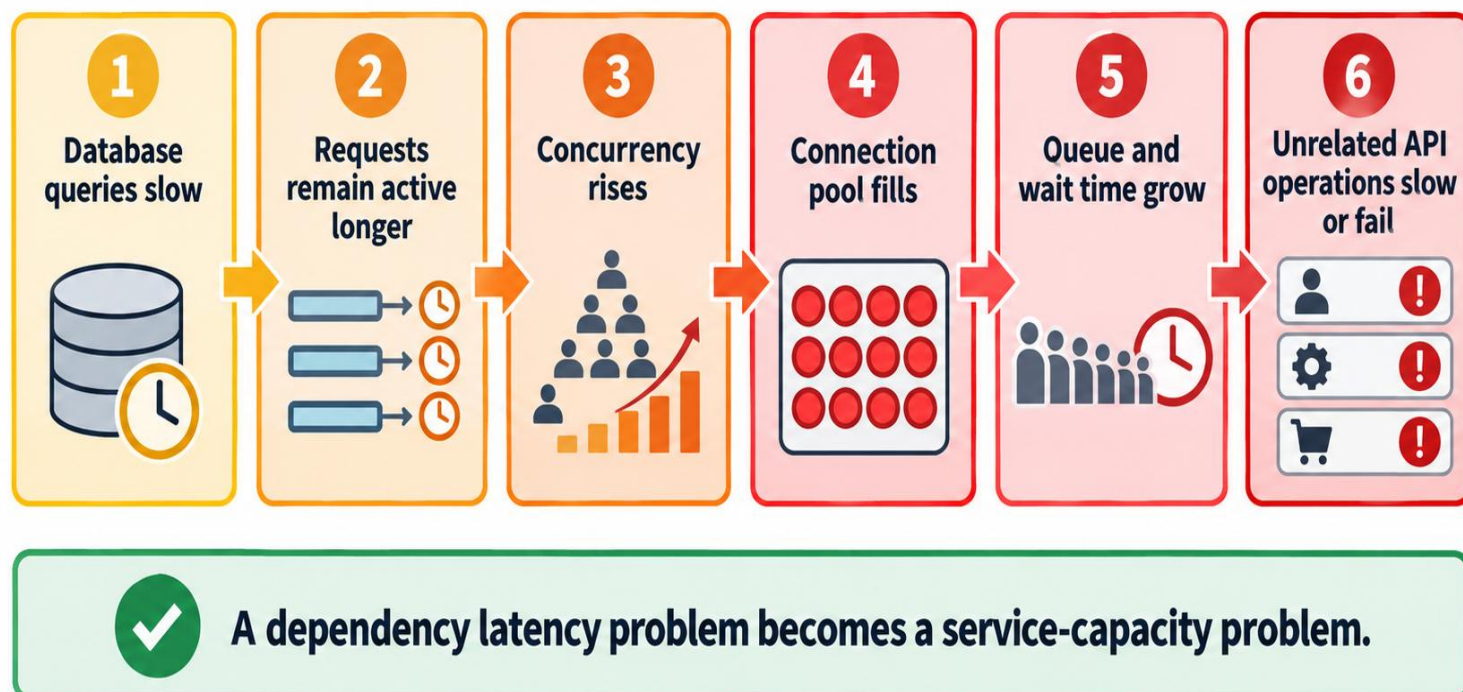
A slow database

A **connection pool** is a reusable, limited set of database connections maintained by an application. Suppose the order database becomes slow but does not reject connections. API instances continue accepting requests and wait longer for

queries. Request concurrency rises because each request remains active. Connection pools fill, queues grow, and eventually unrelated API operations also slow down.

Chapter 2: Understanding a Service and How It Fails

2-10A Slow Database Cascade to Service-Wide Saturation



015

Figure 2-9: Slow database cascading into service-wide saturation

The initial failure is not “the API is down.” It is a dependency latency problem that spreads through **resource saturation**, a state in which demand reaches or exceeds a resource's effective capacity. A host-level CPU alert may not detect it early. A user-visible latency measure and database query evidence are more useful.

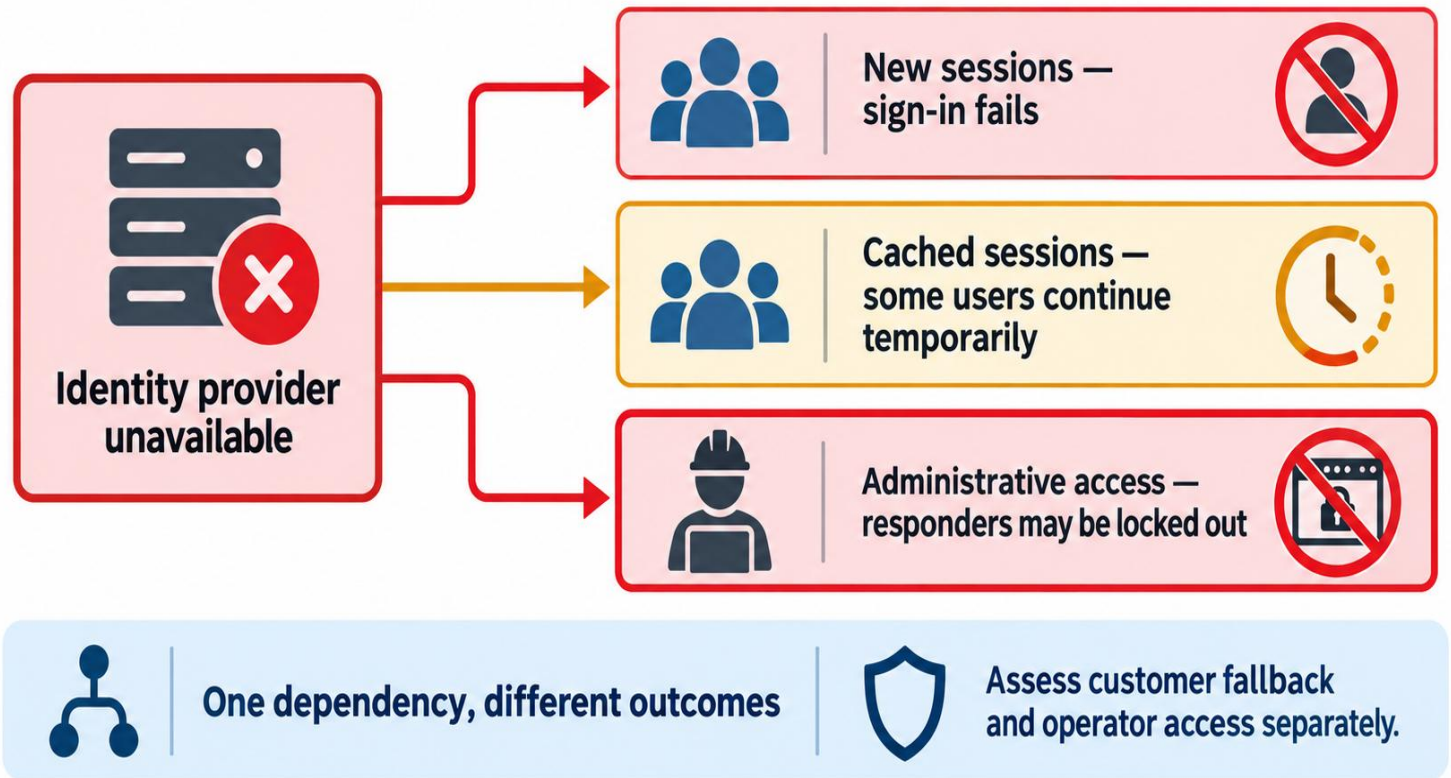
An unavailable identity provider

Suppose the identity provider is unavailable. New sessions cannot be established, but customers with recently validated sessions may still be able to place orders if cached credentials remain valid. Administrative access may fail at the same time if it uses the same provider.

This is partial failure with two distinct effects: customer authentication and operator access. The service map should reveal both. A fallback that allows anonymous checkout might reduce customer impact, but it changes fraud and security risk, so it cannot be treated as a purely technical decision.

Chapter 2: Understanding a Service and How It Fails

2-10B Identity Outage: Customer and Responder Impact



016

Figure 2-10: Identity outage affecting customers and responders differently

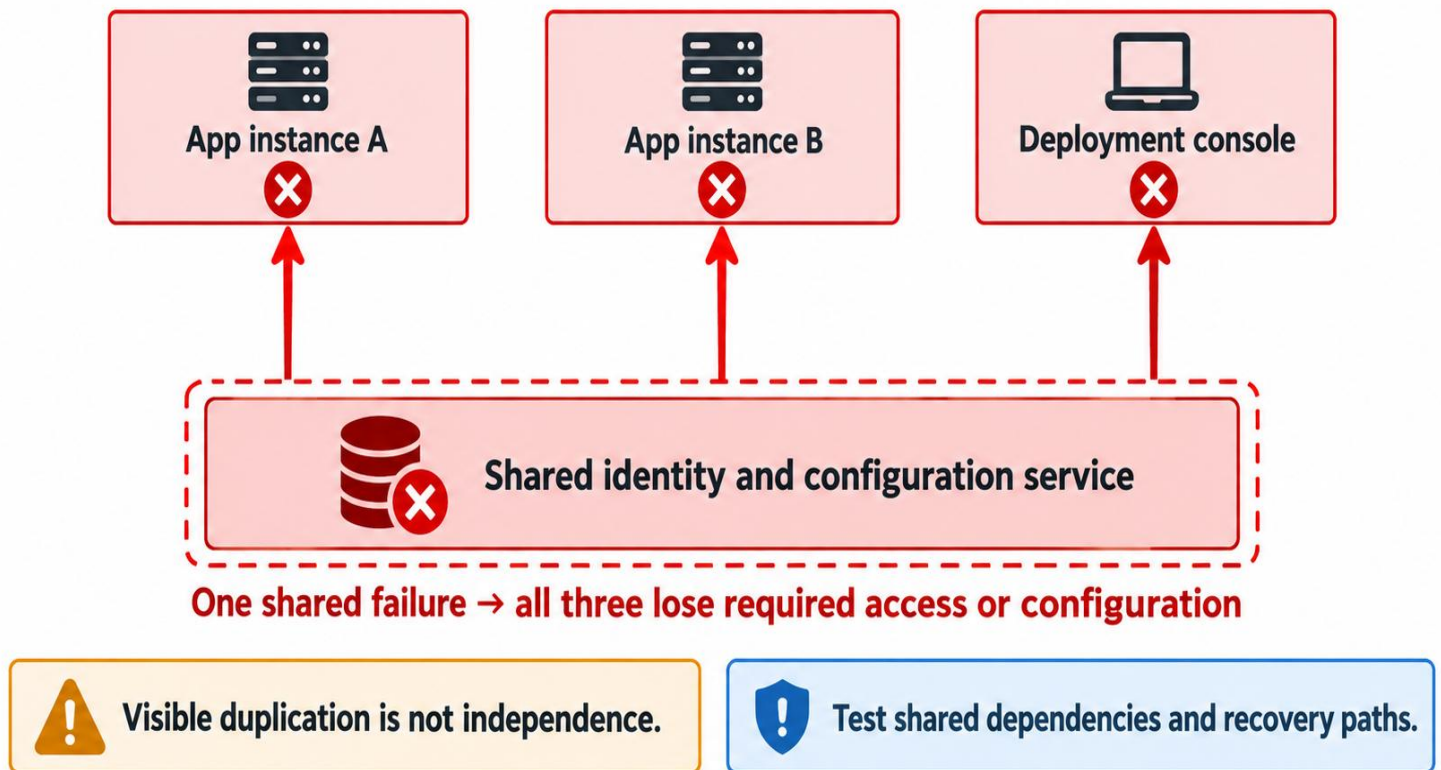
Hidden Coupling and Correlated Failure

Hidden coupling is a dependency relationship that is not obvious from the intended design. It often appears through shared configuration, libraries, credentials, networks, deployment systems, or operational procedures.

Correlated failure occurs when multiple components fail together because they share a cause. Redundant application instances can all fail after receiving the same invalid configuration. Independent services can all become unavailable when a common identity provider fails.

Chapter 2: Understanding a Service and How It Fails

2-11 Hidden Coupling and Correlated Failure



017

Figure 2-11: Hidden coupling and correlated failure

A beginner does not need formal graph analysis to find important coupling. Ask concrete questions:

- Which components are deployed together?
- Which credentials or control services do they share?
- Which teams, accounts, networks, or data stores are common?
- Can the fallback be activated when the primary path has failed?
- Has the fallback been used recently enough to trust it?

The chapter's main result is a working model: **start with the user journey, trace the technical path, attach ownership, and examine how partial and shared failures change the user outcome.** This model becomes the basis for risk decisions and reliability measurements.

Chapter Summary

- Reliability analysis begins with the user journey and follows the complete path required to produce the result.
- Components and dependencies differ in state, criticality, ownership, and failure behavior.
- A useful service map exposes fallback paths, shared failure domains, and places where local failure can spread.

Review Questions

1. How is a user journey different from a technical request path?
2. Why can a partial failure be harder to diagnose than a complete outage?
3. What information must a service map show to support reliability decisions?