Kubernetes A Complete Visual Guide

Sample Book
Chapters 1 to 2

From Foundations to Production Operations

Introduction

This book is designed for readers who are new to Kubernetes and systems engineers who want to organize and deepen their knowledge of the subject, from foundations through practical intermediate and advanced applications.

Rather than asking you to memorize more than you need, the book builds from essential concepts to the intermediate and advanced applications commonly used in day-to-day work, explaining them in enough depth for you to apply them in practice.

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.

Kubernetes is a distributed control system that continually tries to make running infrastructure match declared intent. A production Kubernetes platform succeeds only when that control model is paired with safe change, explicit security boundaries, observable behavior, tested failure handling, and verified recovery. The most useful way to learn Kubernetes is therefore not to memorize YAML, but to follow four questions for every object and workflow: **What state did I request? Which controller or operator is responsible? What evidence shows the current state? What dependency prevents convergence?**

This guide progresses from container and Linux foundations through Kubernetes architecture, workloads, networking, storage, scheduling, and autoscaling, then builds those mechanisms into production cluster design, delivery systems, security governance, observability, troubleshooting, disaster recovery, performance engineering, operators, service meshes, multi-cluster operations, and an integrated capstone.

The examples target **Kubernetes 1.37** and prefer stable APIs. Linux worker nodes are assumed. A local three-node `kind` cluster is used for many exercises because it is reproducible, but local containers do not reproduce cloud load balancers, durable storage, failure domains, managed identity, or production security controls. Product-specific examples are identified as such. Managed Kubernetes services, Linux distributions, network plugins, storage drivers, and cloud integrations differ, so version and platform support must be checked before applying an environment-specific procedure. Helm examples use **Helm 4** behavior; teams operating Helm 3 should validate client, plugin, and chart compatibility.

A small case study develops throughout the guide. It begins as a stateless HTTP service named `web`, then gains health checks, configuration, stable discovery, external routing, traffic restrictions, persistence, placement rules, autoscaling, delivery controls, identity, policy, telemetry, recovery procedures, and multi-cluster considerations.

How to Use This Book

Read the chapters in order when the subject is new to you. The sequence deliberately builds the Linux and container foundation first, then the Kubernetes control model, workload mechanics, networking, storage, scheduling, production operations, security, reliability, and advanced platform patterns. Experienced practitioners can enter at a later chapter, but should revisit an earlier chapter whenever a concept depends on an unfamiliar mechanism.

At the beginning of each chapter, review **What You'll Be Able to Do** before reading the detailed material. It identifies the practical outcomes that the explanations, diagrams, commands, and worked examples are designed to support.

Diagrams clarify relationships, processes, state transitions, security boundaries, and component responsibilities that are difficult to hold in working memory from prose alone. Read the diagram together with the surrounding explanation: the image provides the structural model, while the text supplies mechanisms, constraints, trade-offs, and failure conditions.

Run the code and hands-on examples yourself whenever a suitable lab is available. For every command, confirm the prerequisites, predict the intended change, compare the actual result with the expected result, and use the stated verification step rather than assuming that command completion proves success. Output varies by runtime, operating system, network plugin, storage implementation, and Kubernetes distribution, so verify **conditions and relationships**, not incidental identifiers such as Pod suffixes or IP addresses.

Use each **Chapter Summary** and **Review Questions** section to check both foundational understanding and practical application. A satisfactory answer should explain not only what an object or feature is, but also when to use it, how to verify it, which failure modes matter, and what evidence would distinguish competing causes.

Intermediate and advanced applications in this book build directly on the foundations and are taught within the guide rather than treated as external prerequisites. Work through the practical examples, inspect the resulting objects and telemetry, deliberately test important failure or denial paths, and verify recovery. This is the most reliable way to turn conceptual familiarity into operating competence.

Table of Contents

Introduction
How to Use This Book
Main Text
Part I. Foundations and the Kubernetes Operating Model
- Chapter 1. Kubernetes in One Picture: What It Solves and When It Fits
  - 1.1 The Core Idea: Reconciliation
  - 1.2 What Kubernetes Coordinates
  - 1.3 One Request from Source Code to User
  - 1.4 What Kubernetes Does Not Guarantee
  - 1.5 When Kubernetes Fits
  - 1.6 Choosing the Simplest Sufficient Platform
  - 1.7 Trace Two Reconciliation Events
    - Changing a replica count
    - Losing a node
  - 1.8 Practical Decision Exercise
  - 1.9 Misconceptions That Cause Incidents
- Chapter 2. Container, Linux, and Network Foundations Kubernetes Assumes
  - 2.1 Image, Container, and Process
  - 2.2 Tags Are Names; Digests Identify Content
    - Local inspection example
  - 2.3 Namespaces: Isolation of Views
  - 2.4 cgroups: Accounting and Control
  - 2.5 Capabilities and seccomp
  - 2.6 PID 1 and Signal Handling
    - Observe graceful and forced termination
  - 2.7 Filesystems and Ephemeral Data
  - 2.8 The Network Path Is Layered
    - Independent connectivity checks
  - 2.9 Routing, NAT, and Load Balancing
  - 2.10 TLS Has Endpoints, Not a Magical “On” State
  - 2.11 Foundation Failures That Look Like Kubernetes Failures

Main Text

Part I. Foundations and the Kubernetes Operating Model

Chapter 1. Kubernetes in One Picture: What It Solves and When It Fits

What You'll Be Able to Do

After completing this chapter, you will be able to:

- Describe Kubernetes as a reconciliation system rather than a collection of deployment commands.
- Trace responsibility from application code through Kubernetes mechanisms, platform services, and infrastructure.
- Decide when Kubernetes is justified and when a simpler platform is the safer choice.

Kubernetes is appropriate when a team needs a common, automated control plane for many containerized workloads and is prepared to operate the additional distributed system that provides it. It is not a general-purpose remedy for unreliable applications, unclear ownership, or weak operational practices.

1.1 The Core Idea: Reconciliation

A **declarative system** accepts a description of the result you want rather than a sequence of low-level actions. In Kubernetes, that description is stored as API objects. Controllers repeatedly compare the desired state in those objects with observed reality and take corrective action.

A thermostat is a useful analogy. You set a target temperature rather than issuing instructions such as "run the heater for 47 seconds." The thermostat measures the room, compares measurement with target, and acts until the difference is acceptable. Kubernetes works similarly:

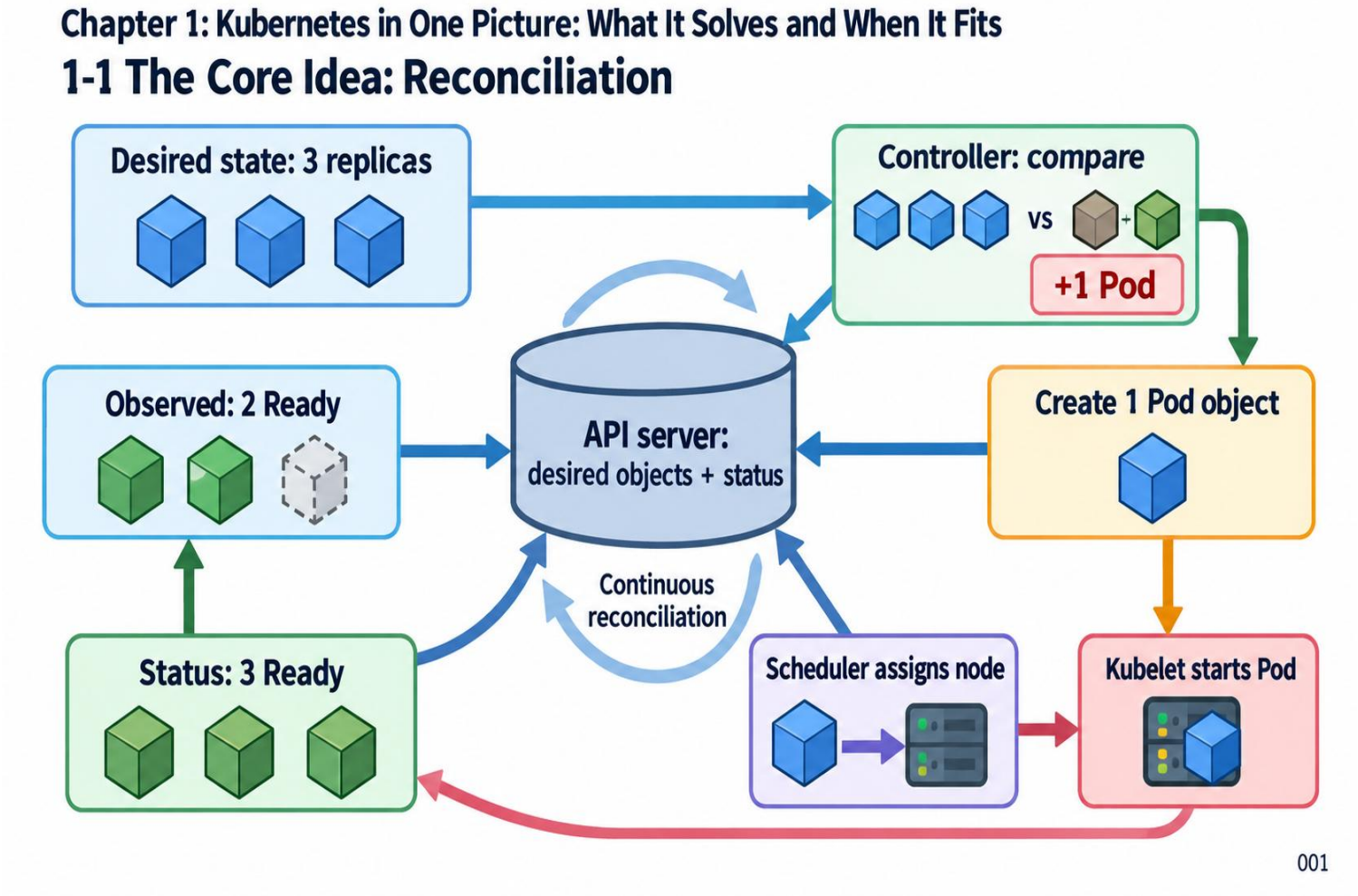


Figure 1-1: The Core Idea: Reconciliation

The loop is **continuous**, not a one-time deployment action. If one replica disappears later, the controller detects the new difference and creates another. This property is often called **self-healing**, but the term has limits: Kubernetes can replace a failed process or Pod; it cannot repair a corrupt business transaction, restore missing data without a recovery design, or decide whether a logically incorrect response is healthy.

1.2 What Kubernetes Coordinates

Kubernetes provides common mechanisms for several recurring infrastructure problems:

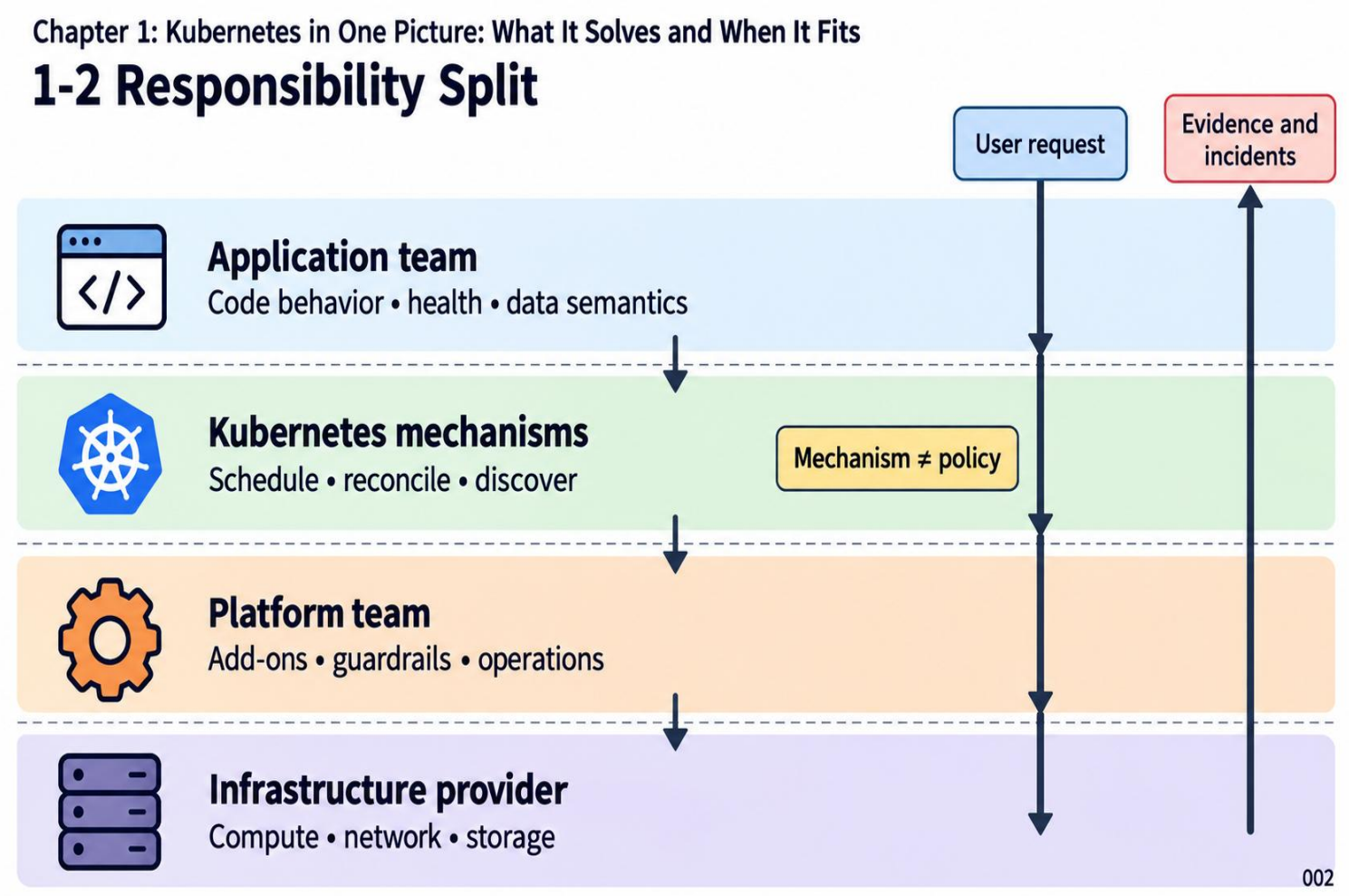


Figure 1-2: Responsibility Split

- **Scheduling:** choosing an eligible node for each Pod based on resources and placement rules.
- **Service discovery:** giving clients a stable name and virtual endpoint while Pods change.
- **Rollout control:** replacing one application version with another while enforcing availability constraints.
- **Failure recovery:** recreating controller-managed Pods and restarting failed containers.
- **Elasticity:** changing replica counts from operator input or measured signals.
- **Configuration distribution:** delivering non-sensitive and sensitive values to workloads.
- **Policy attachment:** expressing rules for identity, admission, network reachability, placement, and resource use.

A **container runtime** starts and stops containers on one machine. A **cluster orchestrator** coordinates workloads across machines. Kubernetes is an orchestrator and API platform; it becomes a broader application platform only after teams select and operate networking, storage, ingress or Gateway, metrics, policy, logging, delivery, and other components.

1.3 One Request from Source Code to User

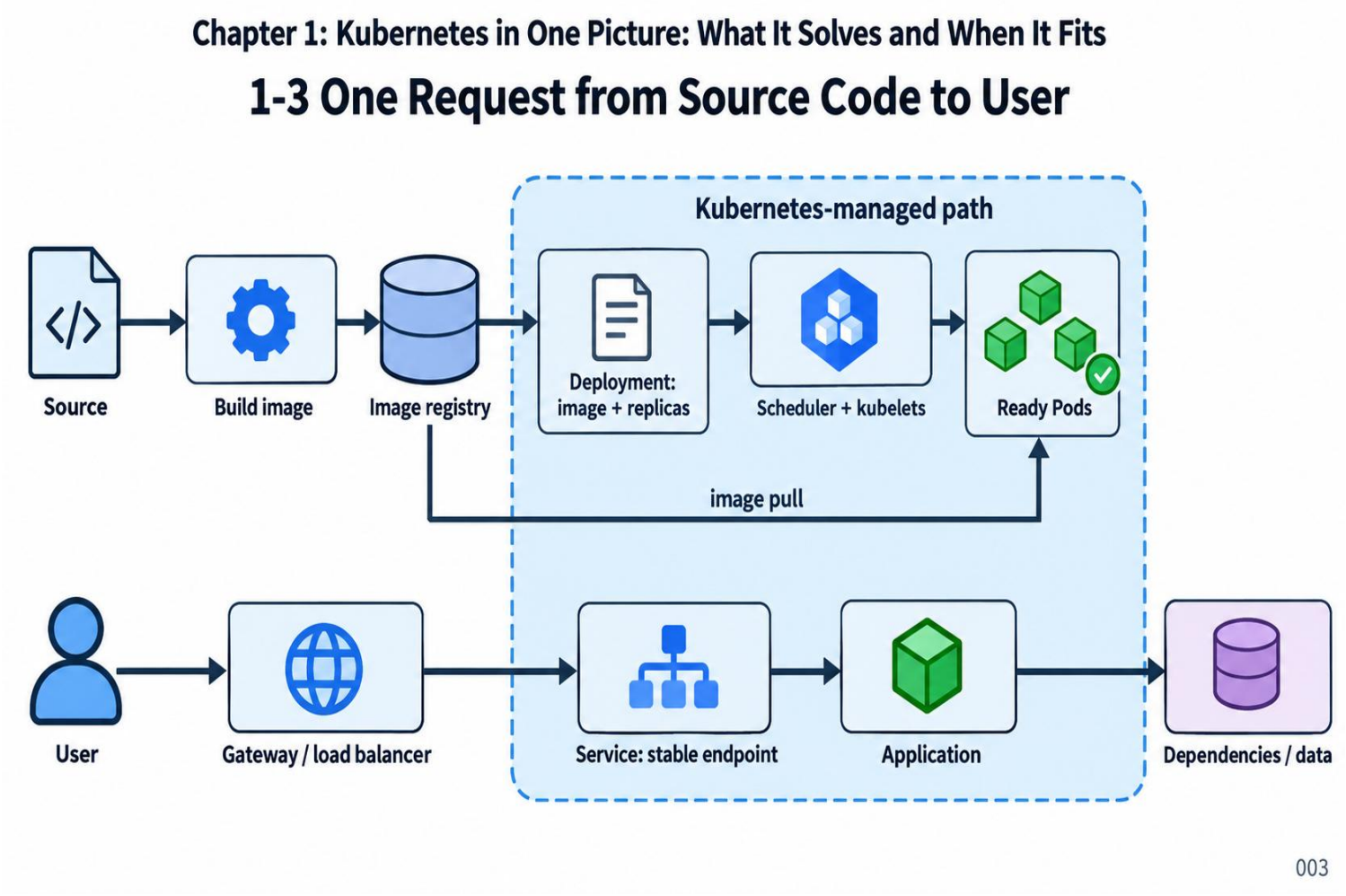


Figure 1-3: One Request from Source Code to User

Kubernetes owns only part of this path. The application team still owns code behavior, health semantics, data migrations, dependency handling, and safe shutdown. A platform team may own cluster services and guardrails. A cloud provider may own portions of the control plane, network, or storage. **Reliability depends on the complete path and the boundaries between owners.**

1.4 What Kubernetes Does Not Guarantee

A running Pod is not proof of a correct service. Kubernetes cannot infer whether an application returned the wrong price, acknowledged work before committing it, or served stale authorization data. It also does not automatically provide:

- Durable application data or application-consistent backups.
- Correct readiness, liveness, or startup checks.
- Safe CPU and memory sizing.
- Least-privilege access for every workload.
- End-to-end encryption merely because external TLS is configured.
- Useful logs, traces, dashboards, alerts, or service-level objectives.
- A tested recovery process.

The platform supplies mechanisms. Engineers must define **meaningful signals, policies, and operating procedures**.

1.5 When Kubernetes Fits

Kubernetes usually becomes reasonable when several of these conditions are true:

- Many services need consistent deployment, discovery, rollout, and policy behavior.

- Teams need portable APIs across multiple infrastructure environments.
- Workloads have variable demand or require automated rescheduling.
- A platform team can provide supported patterns rather than exposing raw complexity to every developer.
- The organization can test upgrades, add-ons, failure modes, and recovery.
- Standardization has more value than the cost of another distributed control plane.

Warning signs that a simpler model may be better include one small service, stable low traffic, no platform ownership, weak container knowledge, highly specialized hardware, or a regulatory environment where the platform cannot be supported and evidenced adequately.

1.6 Choosing the Simplest Sufficient Platform

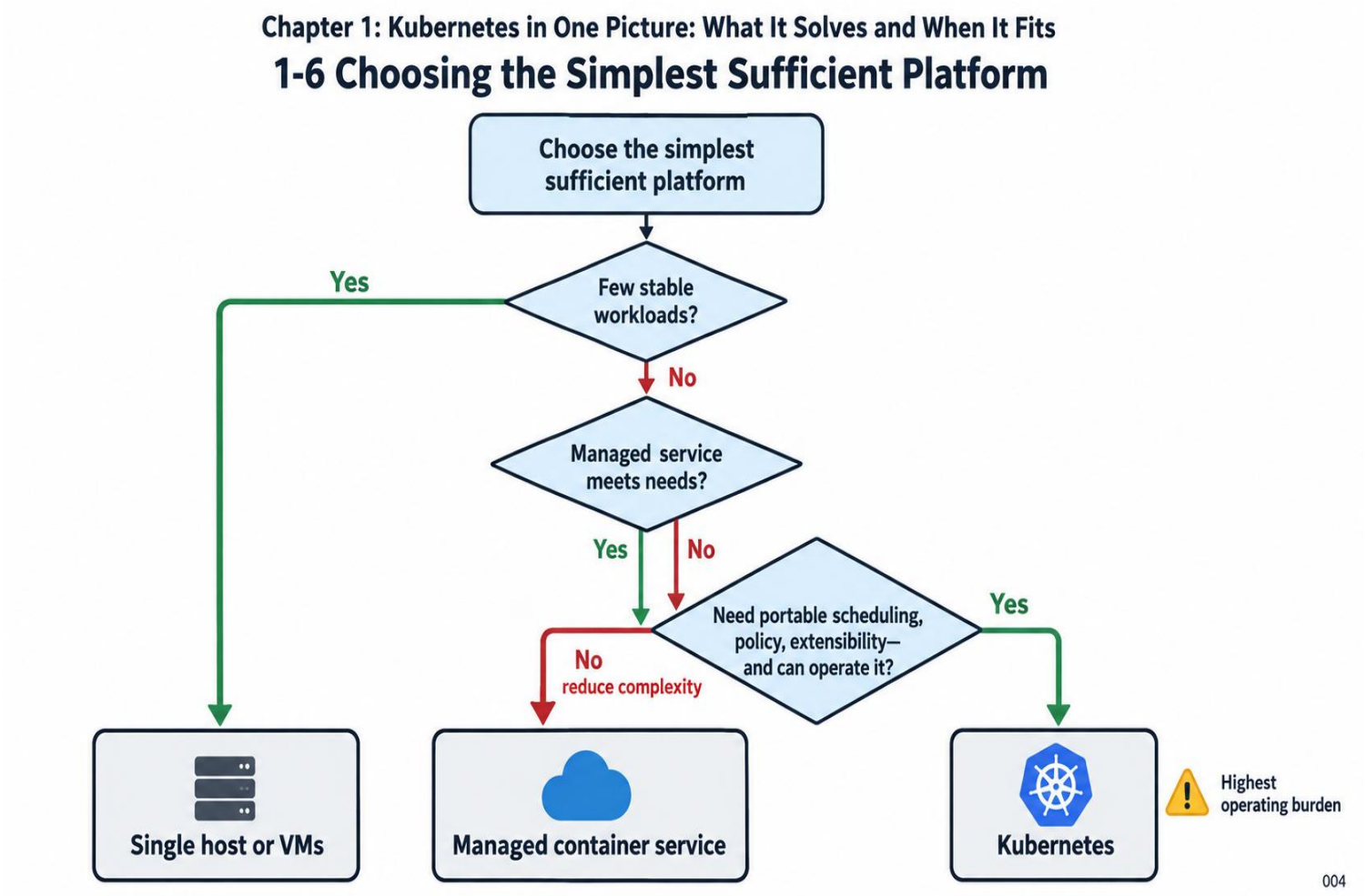


Figure 1-4: Choosing the Simplest Sufficient Platform

Requirement	Single host or virtual machines	Managed container service	Kubernetes
Small number of stable workloads	Often simplest	Good fit	Often unnecessary
Fine-grained scheduling and placement	Manual or external tooling	Product-dependent	Strong, portable model
Extensible APIs and controllers	Limited	Product-dependent	Core design strength
Operational burden	Lowest initially	Shared with provider	Highest unless well managed
Portability	Image and OS level	Provider-specific	API portability, with add-on differences
Complex multi-team policy	Custom tooling	Product-dependent	Rich mechanisms, still requires governance
Fast path to first deployment	Usually fastest	Fast	Slower because platform choices are required

Choose Kubernetes for the control model and ecosystem it enables, not because it is fashionable. A managed container service can be a better engineering decision when it satisfies the same requirements with fewer moving parts.

1.7 Trace Two Reconciliation Events

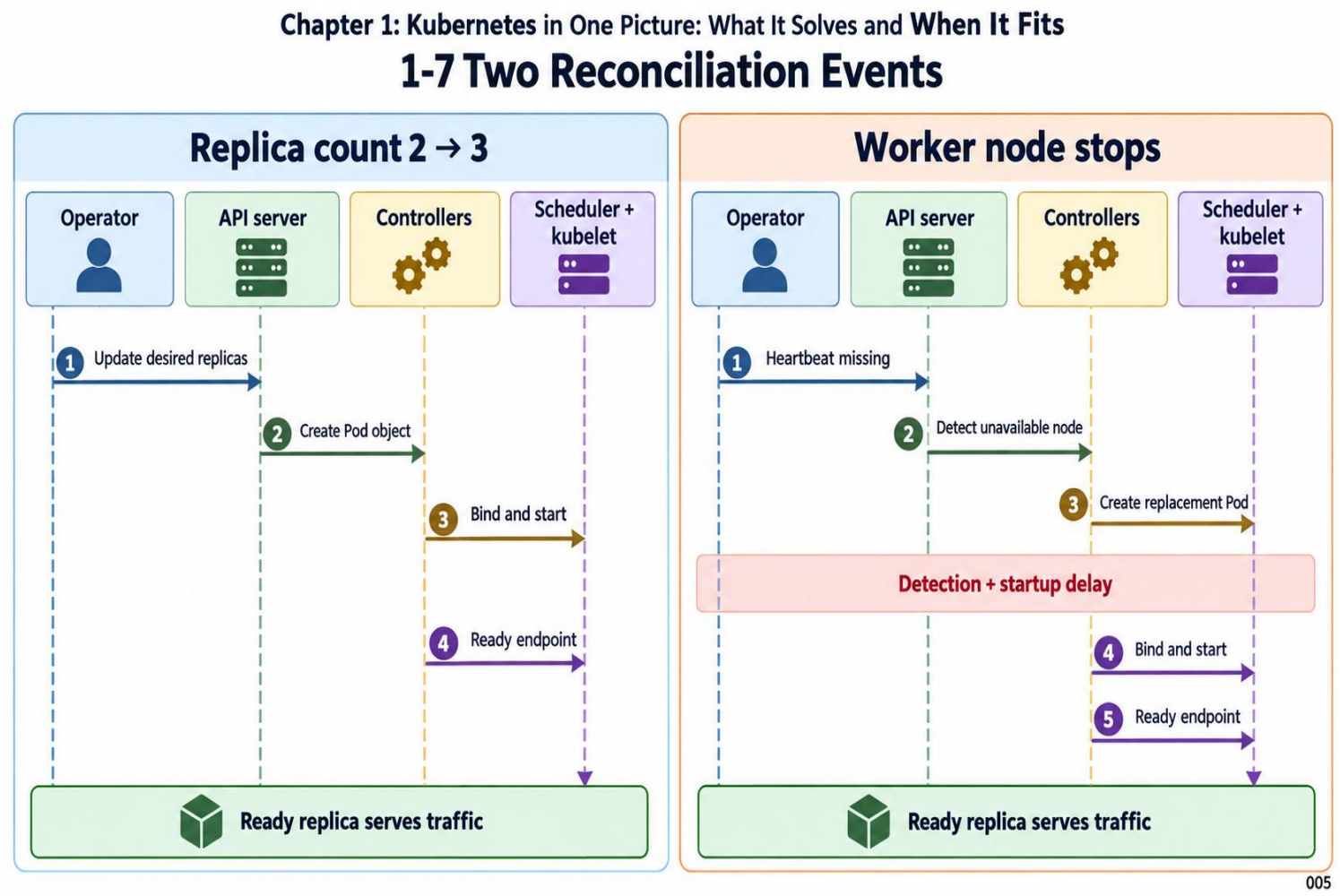


Figure 1-5: Two Reconciliation Events

Changing a replica count

Assume a Deployment currently has two ready replicas and its desired count changes to three.

1. The updated Deployment is accepted by the API server.
2. The Deployment controller changes the desired ReplicaSet state.
3. The ReplicaSet controller creates a Pod object.
4. The scheduler assigns the unscheduled Pod to a node.
5. The node's kubelet asks the runtime to pull the image and start containers.
6. Readiness becomes true after the application can serve.
7. The Service begins using the new endpoint.

The important point is that **no single component performs the whole operation**. Each component reconciles a narrower responsibility.

Losing a node

If a worker node stops reporting health, the control plane eventually marks it unavailable. Existing connections to Pods on that node fail. Controllers create replacements elsewhere when policy and capacity permit. Recovery time therefore includes failure detection, Pod creation, image pull, application startup, and readiness—not just “Kubernetes restart time.”

1.8 Practical Decision Exercise

Consider an internal reporting service with two containers, five users, no 24-hour requirement, and one operator. A virtual machine with a supervised container process may be safer and cheaper. Now add 80 services, ten teams,

controlled rollout requirements, isolated environments, policy enforcement, and variable demand. Kubernetes becomes more attractive because a common control plane replaces many one-off systems.

The deciding question is not “Can Kubernetes run this?” It almost certainly can. The useful question is **“Does the organization gain more from standardized reconciliation than it spends operating and constraining the platform?”**

1.9 Misconceptions That Cause Incidents

“Kubernetes makes the application highly available.” It supplies replication and placement mechanisms. Availability still depends on independent failure domains, capacity, readiness, dependency behavior, and data design.

“A healthy Pod means a healthy user journey.” A probe observes only what it was designed to test. A shallow check can remain green while all real requests fail.

“Kubernetes replaces systems engineering.” It exposes Linux, DNS, routing, TLS, storage, and resource behavior through another control layer. Engineers who understand the lower layers diagnose incidents faster.

Chapter Summary

- Kubernetes continuously compares declared intent with observed state and acts through controllers to reduce the difference.
- The platform coordinates workloads, but application correctness, data safety, and operating discipline remain explicit engineering responsibilities.
- Adoption is justified by recurring orchestration and policy needs, not by container use alone.

Review Questions

1. What four questions reveal where a reconciliation failure is occurring?
2. Which parts of an end-to-end request remain outside Kubernetes control?
3. What evidence would support choosing Kubernetes over a simpler deployment platform?

Chapter 2. Container, Linux, and Network Foundations Kubernetes Assumes

What You'll Be Able to Do

After completing this chapter, you will be able to:

- Explain how images, containers, and processes relate at runtime.
- Reason about Linux namespaces, cgroups, capabilities, seccomp, signals, and ephemeral filesystems.
- Decompose a failed connection into name resolution, routing, transport, TLS, and application layers.

A Kubernetes workload is still a set of Linux processes using ordinary files, sockets, memory, and CPU. Kubernetes changes how those processes are described and coordinated; it does not suspend operating-system or network behavior.

2.1 Image, Container, and Process

An **OCI image** is a content-addressed package of filesystem layers and metadata that a compatible runtime can use to create a container. A **container** is a running or stopped process environment created from that image with isolation and resource controls. It is not a miniature virtual machine with its own kernel.

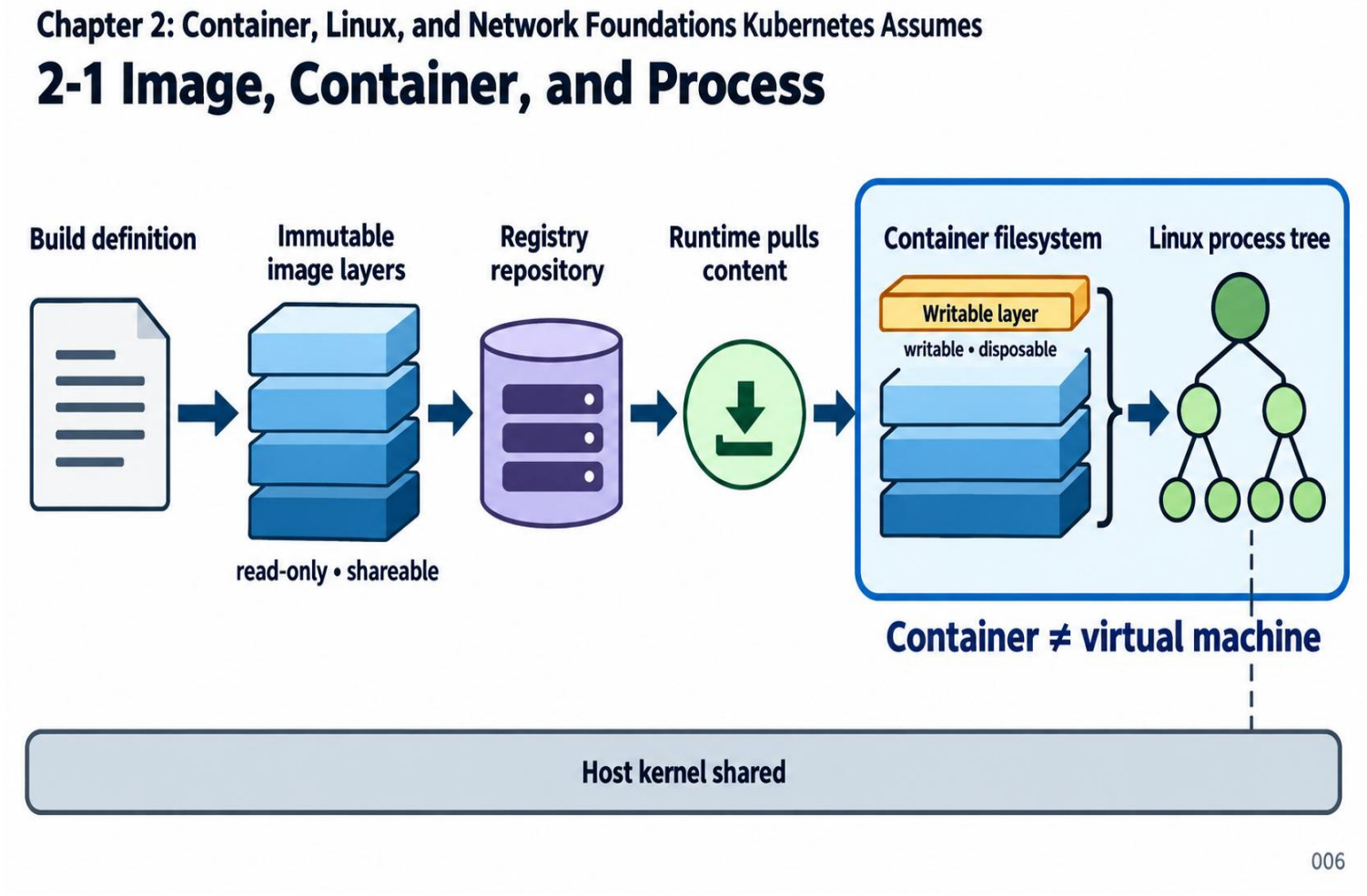


Figure 2-1: Image, Container, and Process

Image layers are read-only and shared where the runtime permits. Each running container receives a writable layer that is normally disposable. Kubernetes can mount volumes into selected paths, but writing elsewhere in the container filesystem does not make the data durable.

2.2 Tags Are Names; Digests Identify Content

A tag such as `app:stable` is a mutable reference that a registry operator can move to different content. A digest such as `app@sha256:...` identifies exact image content.

Use tags for human-readable release organization, but use immutable digests when reproducibility and provenance matter. A practical deployment may record both through an image reference and release metadata.

Local inspection example

This example uses Docker as one local runtime interface. Podman or another supported tool may use different commands.

Prerequisites: a working local container runtime and registry access.

```
docker pull nginx:1.27-alpine
docker image inspect nginx:1.27-alpine \
  --format 'image={{.Id}} digests={{json .RepoDigests}}'
```

The pull downloads missing image content. The inspect command shows the local image ID and any repository digest recorded by the runtime.

Expected result: at least one content identifier is displayed. Verify that rerunning the command gives the same digest until the referenced registry content changes. A mutable tag can later resolve to a different digest, which is precisely why tags alone are not immutable evidence.

2.3 Namespaces: Isolation of Views

A Linux **namespace** isolates a process's view of a resource. Kubernetes and runtimes commonly use namespaces for processes, networking, mounts, hostnames, and interprocess communication.

- A PID namespace gives the container a restricted process view.
- A network namespace gives it interfaces, routes, and sockets.
- A mount namespace gives it a distinct filesystem mount view.
- A UTS namespace isolates hostname and domain name.

Isolation is not the same as a separate kernel. Kernel vulnerabilities and overly broad privileges can cross expected boundaries, which is why container hardening still matters.

2.4 cgroups: Accounting and Control

A Linux **control group**, usually shortened to cgroup, accounts for and constrains resources used by processes. Kubernetes resource requests influence scheduling; limits are implemented through runtime and kernel controls.

CPU and memory fail differently:

- **CPU is compressible.** When a container reaches a CPU limit, it can be throttled and run more slowly.
- **Memory is incompressible.** When memory cannot be reclaimed and a cgroup limit is exceeded, a process may be killed by the kernel's out-of-memory handling.

This difference explains a common incident pattern: a CPU-constrained service becomes slow, while a memory-constrained service restarts with `OOMKilled`.

2.5 Capabilities and seccomp

Linux traditionally grants broad privilege to user ID 0. **Capabilities** split root privilege into narrower units, such as binding low ports or changing network configuration. **seccomp** restricts which system calls a process may make.

Kubernetes can configure both through a container security context. Detailed governance belongs later in the full guide, but the foundational rule is immediate: **do not assume “inside a container” is a security boundary strong enough to justify unnecessary privilege.**

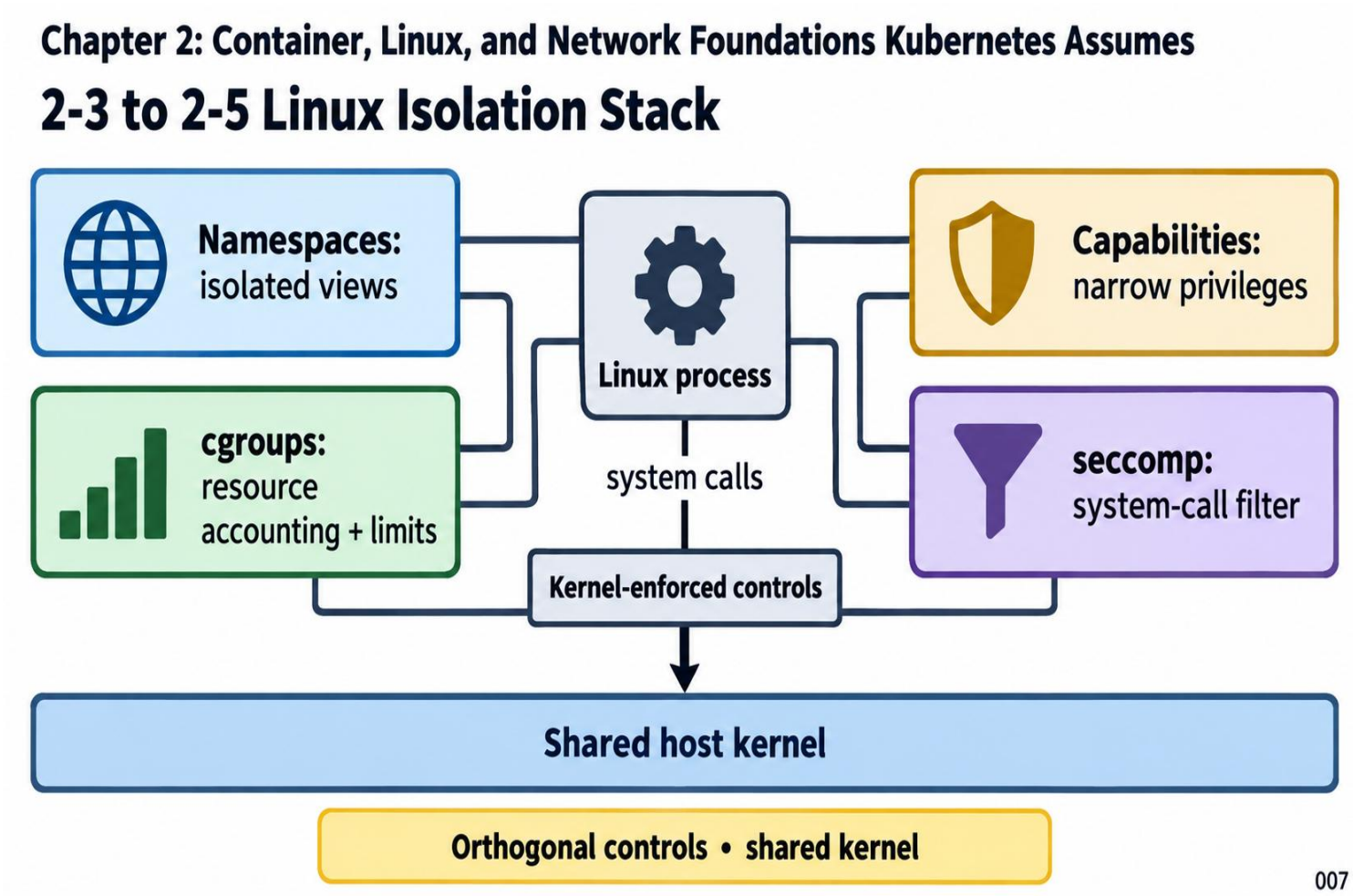


Figure 2-2: Linux Isolation Stack

2.6 PID 1 and Signal Handling

The first process in a container becomes PID 1 inside its PID namespace. PID 1 has special responsibilities, including reaping orphaned child processes, and many applications or shell wrappers do not handle these responsibilities correctly.

The difference between these container commands is operationally important:

```
# Shell form: PID 1 is commonly a shell.
CMD my-server --port 8080

# Exec form: the server becomes PID 1 directly.
CMD ["my-server", "--port", "8080"]
```

With shell form, a termination signal may reach the shell but not the server unless the shell forwards it. With exec form, the server receives the signal directly. An entrypoint script should normally finish with `exec "$@"` so the application replaces the shell.

2-6 PID 1 Signal Delivery

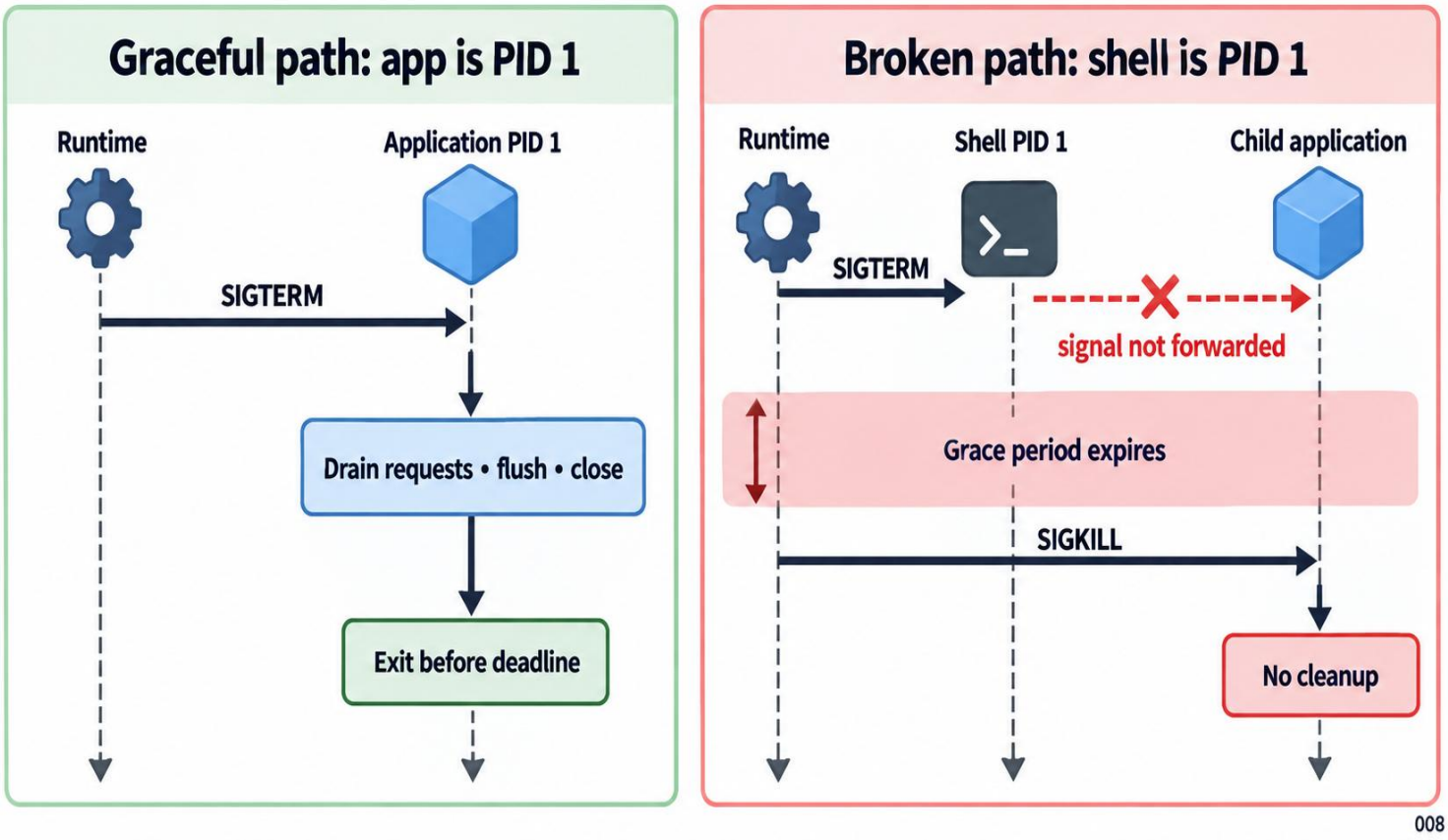


Figure 2-3: PID 1 Signal Delivery

Observe graceful and forced termination

Prerequisites: the nginx image from the previous example.

```
docker run --name signal-demo -d -p 18080:80 nginx:1.27-alpine
docker top signal-demo
docker stop --time 10 signal-demo
docker inspect signal-demo --format 'status={{.State.Status}} exit={{.State.ExitCode}}'
docker rm signal-demo
```

`docker stop` sends the runtime's configured termination signal and waits before forcing termination. Nginx should exit cleanly within the timeout.

To contrast forced termination, start it again and use:

```
docker kill signal-demo
```

A forced kill gives the process no opportunity to drain requests, flush buffers, or close files safely. Kubernetes follows the same essential distinction during Pod termination: graceful signal first, force after the grace period.

2.7 Filesystems and Ephemeral Data

A container's writable layer is tied to that container. A restarted container in the same Pod may receive a fresh writable layer; a replacement Pod certainly does. Common ephemeral locations include:

- The container writable layer.
- `emptyDir` volumes, which live for the Pod's lifetime.
- Container logs stored on the node.

Persistent data requires an explicit storage design. Even a mounted persistent volume is not automatically backed up, replicated, or application-consistent.

2.8 The Network Path Is Layered

A failed HTTP request can result from different layers that happen to present similar symptoms.

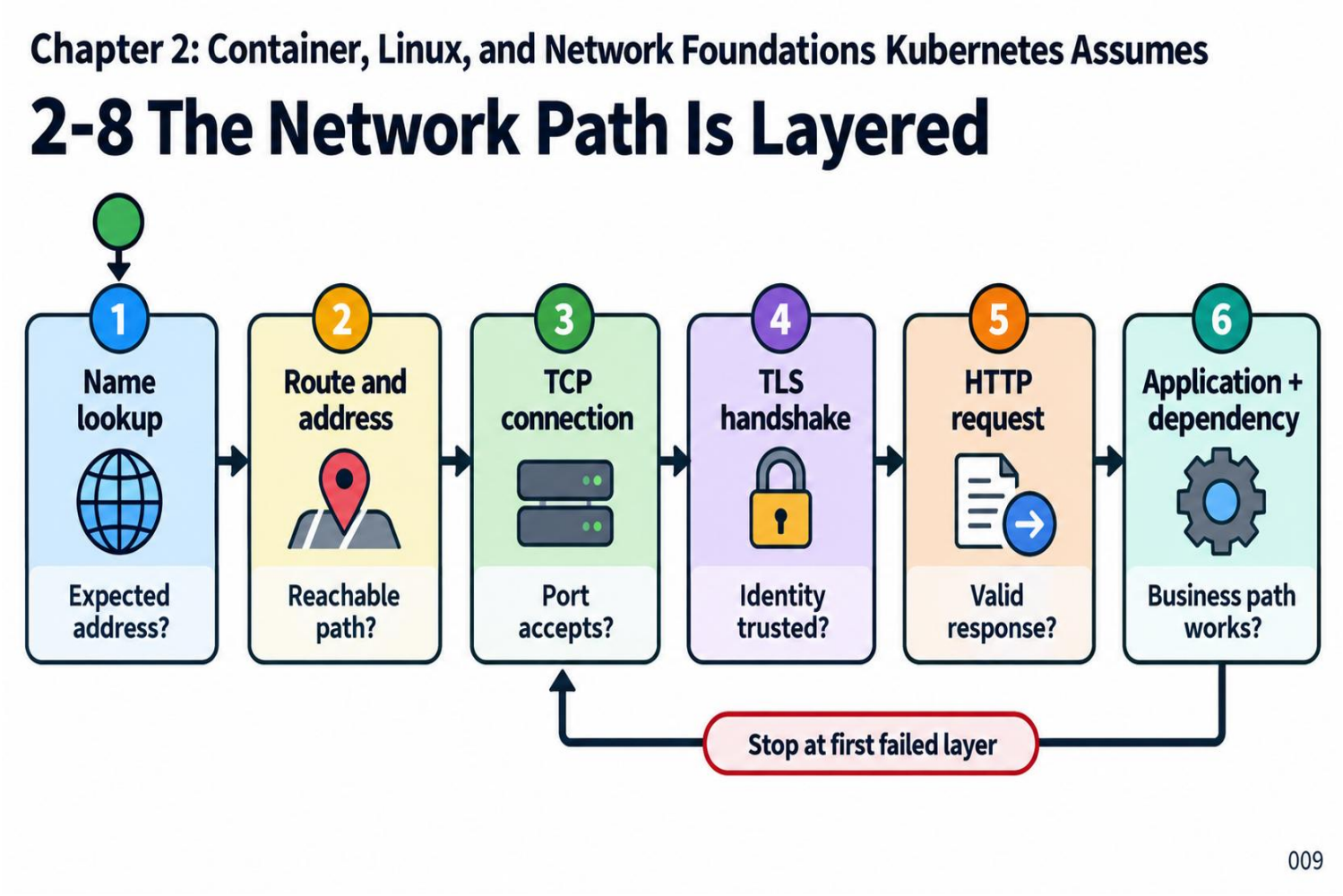


Figure 2-4: The Network Path Is Layered

Diagnose in that order. Do not investigate an HTTP route before proving that DNS resolved the expected address and TCP connected to the intended port.

Independent connectivity checks

Prerequisites: `getent`, `curl`, and OpenSSL on the workstation; replace the host with one you are authorized to test.

```
getent hosts example.com
curl -v --connect-timeout 5 https://example.com/
openssl s_client -connect example.com:443 -servername example.com </dev/null
```

- `getent` verifies name resolution through the system resolver.
- `curl -v` exposes address selection, TCP connection, TLS, and HTTP details.
- `openssl s_client` isolates the TLS handshake and certificate chain.

Expected result: the name resolves, a TCP connection opens, the certificate name matches, and an HTTP response arrives. A failure at one layer should redirect investigation to that layer rather than to Kubernetes generally.

2.9 Routing, NAT, and Load Balancing

A route chooses the next hop for an IP packet. **Network address translation**, or NAT, rewrites addresses or ports. A load balancer selects one backend among several. Kubernetes networking uses all three, depending on the implementation.

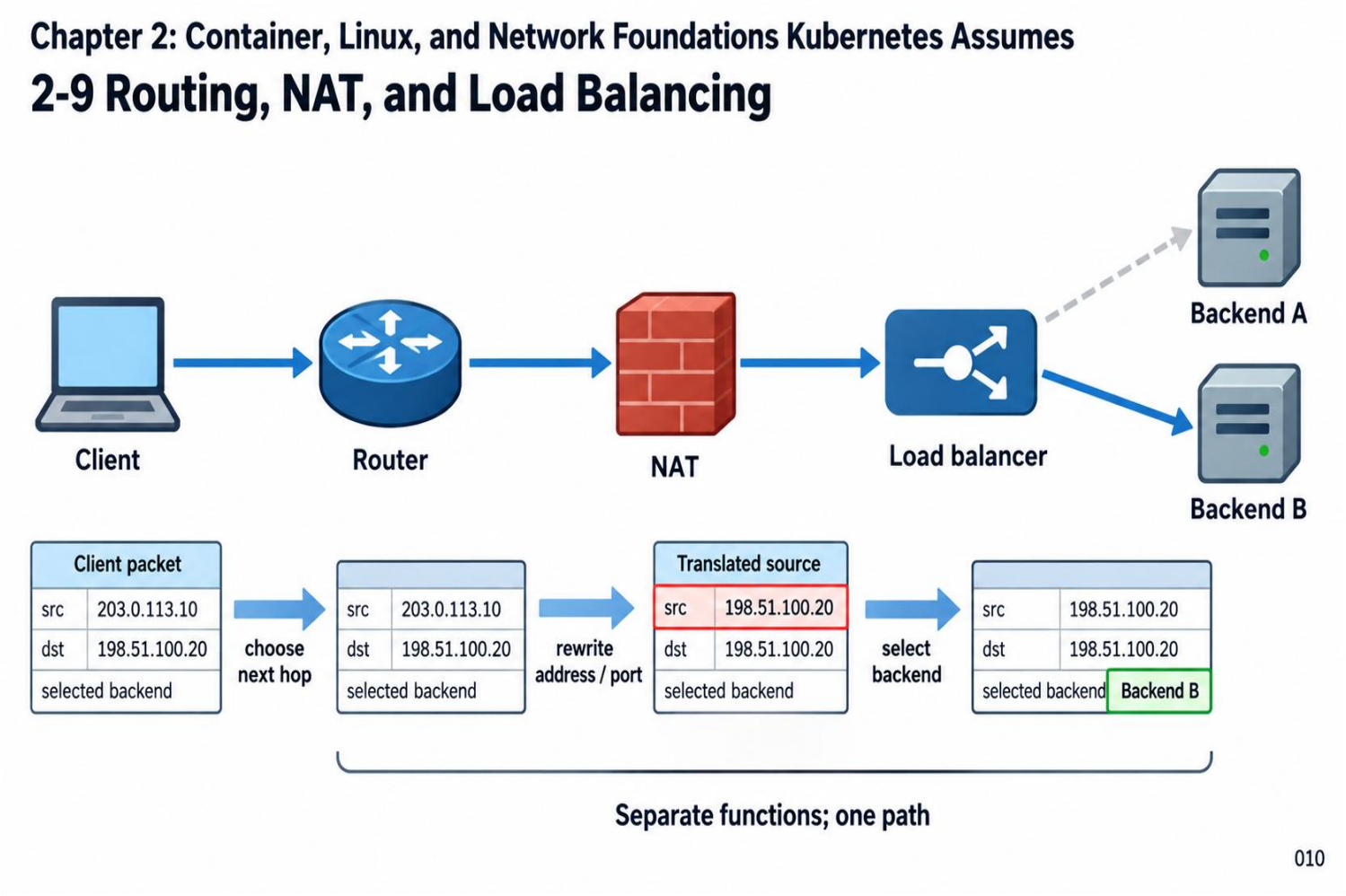


Figure 2-5: Routing, NAT, and Load Balancing

Source addresses can change as traffic passes through proxies or NAT. Applications that depend on client IP must understand whether the relevant component preserves the address, forwards it in a trusted header, or replaces it.

2.10 TLS Has Endpoints, Not a Magical “On” State

Transport Layer Security, or TLS, encrypts and authenticates a connection between two endpoints. If an external proxy terminates TLS and forwards plain HTTP to a Pod, only the client-to-proxy hop is encrypted. Re-encryption requires a second TLS connection from proxy to backend.

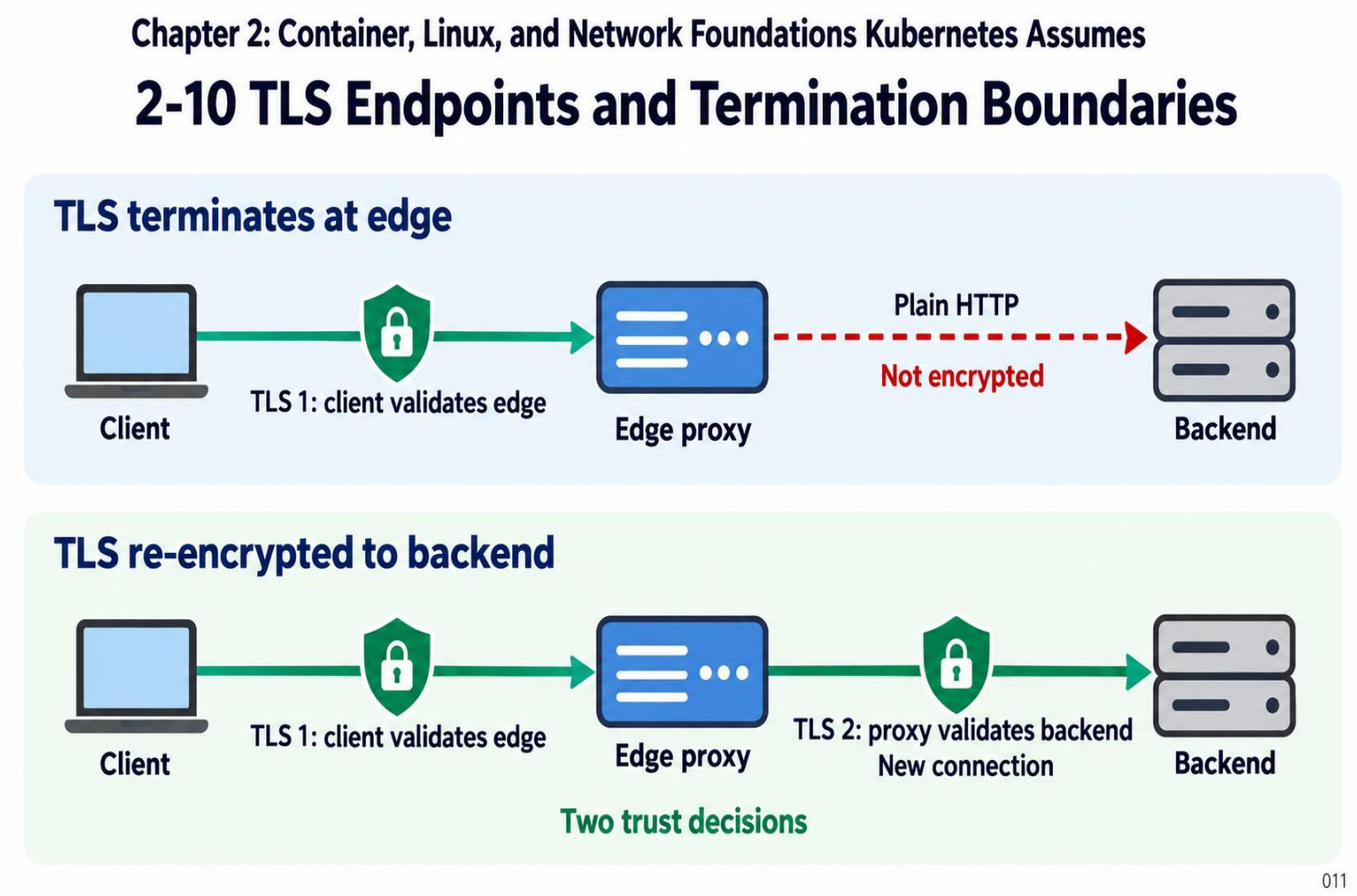


Figure 2-6: TLS Endpoints and Termination Boundaries

Certificate correctness includes the chain of trust, validity dates, key use, and hostname matching. A certificate being present is not proof that clients can validate it.

2.11 Foundation Failures That Look Like Kubernetes Failures

- An application ignores `SIGTERM`, so rollout Pods remain terminating until forced.
- A shell wrapper fails to forward signals.
- A process writes required state into its container layer and loses it on replacement.
- A mutable tag points to unexpected content.
- DNS resolves an old or wrong address.
- A TLS certificate covers a different hostname.
- Memory usage exceeds a cgroup limit and the kernel kills the process.
- CPU throttling raises latency, causing readiness or upstream timeouts.

The operational lesson is **separate orchestration symptoms from process, filesystem, network, and resource causes**.

Chapter Summary

- A container is an isolated process environment created from an image; it is not a lightweight virtual machine.
- Linux isolation and resource-control mechanisms determine the security, shutdown, and performance behavior Kubernetes exposes.

- Network and TLS troubleshooting is faster when each layer and termination boundary is tested independently.

Review Questions

1. Why can a container appear healthy while its application cannot terminate gracefully?
2. How do cgroups differ from namespaces?
3. Which checks separate a DNS failure from a routing, TCP, TLS, or application failure?