Kubernetes

The Complete Story

Every concept explained through stories, from the simplest Pod to the most advanced cluster patterns.

228 concepts. 25 chapters. One continuous narrative.

— FREE SAMPLE —

Chapters 1–3

Vamsikrishna Vankayala

First Edition — 2026 | Covering Kubernetes v1.30+

Kubernetes: The Complete Story

Free Sample — Chapters 1–3

This is a free sample containing Chapters 1–3.

The full book contains 25 chapters, 228 concepts,
415 pages, exercises, appendices, and a cheatsheet.

Get the full book here

Foreword

> *"The best way to predict the future is to invent it."*
> — Alan Kay

When containers first appeared on the scene, they promised a revolution: lightweight, portable, reproducible software that could run anywhere. And they delivered. But with that revolution came a new class of problems. How do you run thousands of containers across hundreds of machines? How do you handle failures at three in the morning? How do you update a live system without taking it down?

Kubernetes answered those questions. Born from over a decade of experience running production workloads at Google, it became the operating system of the cloud. Today, Kubernetes runs everywhere—from the largest financial institutions to two-person startups, from edge devices to supercomputers. It has become the foundation on which modern infrastructure is built.

But Kubernetes has a reputation. It is powerful, yes. It is also vast, layered, and sometimes bewildering. The official documentation is excellent as a reference, but learning Kubernetes by reading reference material is like learning a language by reading a dictionary. You get the words, but you miss the story.

That is what makes this book different.

Vamsikrishna does something rare: he teaches Kubernetes through narrative. Every concept is introduced because a character needs it. Every abstraction exists because a real problem demands it. You do not learn what a Pod is and then wonder why it matters. You watch a team's application crash, understand the pain, and then discover that Pods are the solution. By the time you reach the advanced chapters—service meshes, operators, multi-cluster federation—you have the full context of *why* these tools exist, not just *how* they work.

This approach matters because Kubernetes is not a single tool. It is an ecosystem, a philosophy, a way of thinking about infrastructure. Understanding the "what" without the "why" leads to cargo-cult configurations copied from Stack Overflow. Understanding the story behind each concept gives you the intuition to make your own decisions.

Whether you are a developer deploying your first container, an operations engineer managing a fleet of clusters, or an architect evaluating Kubernetes for your organization, this book meets you where you are and takes you further than you expected to go. All 228 concepts. All 25 chapters. One continuous narrative from the first container to the most advanced patterns in the ecosystem.

The Kubernetes story is still being written. This book gives you everything you need to be part of it.

March 2026

Contents

How to Read This Book

This is not a reference manual. This is a story.

Every Kubernetes concept in this book is born from a problem. You will meet characters—developers, ops engineers, CTOs—who face real failures. Their servers crash. Their apps go down. Their customers leave.

And each time, a new Kubernetes concept appears. Not because someone invented it for fun, but because *the pain demanded a solution.*

The story is continuous. Each concept's limitation naturally leads to the next concept. By the time you finish, you will not have memorized Kubernetes. You will have *understood* it.

★ How each concept is presented

1. **The Story** — A real-world scenario showing the pain
2. **The Problem** — What exactly goes wrong
3. **The Solution** — The Kubernetes concept that fixes it
4. **Under the Hood** — How it actually works beneath
5. **The YAML** — Concrete configuration examples
6. **The Limitation** — What this concept *cannot* solve (leading to the next concept)

Start from Chapter 1. Do not skip ahead. The story builds on itself.

1 The World Before Kubernetes

You cannot understand where you are going until
you understand where you have been.

— Every grizzled ops engineer, eventually

Before we touch a single Kubernetes manifest, we need to understand *why* it exists. Kubernetes did not fall from the sky. It was forged in the fires of production outages, exploding infrastructure bills, and 3 AM pager alerts.

This is the story of how we got here.

1.1 The Monolith Problem

ShopFast: The Early Days It is 2016. Three friends—**Priya** (backend developer), **Marco** (frontend developer), and **Jin** (the "I do everything else" person)—quit their jobs and start a company called **ShopFast**. An e-commerce platform. They are going to disrupt online shopping.

They build the entire application as a single Java monolith. One codebase. One WAR file. One Tomcat server. The product catalog, shopping cart, payment processing, user authentication, order management, email notifications—all in one application.

And it works. Beautifully.

Priya pushes code. Marco builds the frontend. Jin deploys by copying a WAR file to the server. Total deploy time: 4 minutes. They can ship three times a day. Customers love the product. Investors come knocking.

ShopFast grows. 5 engineers become 15. Then 30. Then 50.

And that is when the nightmare begins.

✕ Death by Monolith

With 50 engineers all committing to the same codebase, everything slows to a crawl:

- **Deploy takes 4 hours.** The build process compiles everything. Tests run for all modules. The deployment pipeline is a bottleneck.
- **One bug takes down everything.** A junior developer introduces a memory leak in the payment module. The entire application crashes. Product catalog, search, user accounts—all gone. For everyone.
- **Scaling is all-or-nothing.** Black Friday is coming. The product catalog gets 100x more traffic than the payment system. But you cannot scale them independently. You scale the entire monolith or nothing.
- **Technology lock-in.** The team wants to use Python for the recommendation engine because the ML libraries are better. Too bad. Everything is Java. The monolith does not care about your preferences.
- **Team collisions.** The cart team and the payment team keep breaking each other's code. Merge conflicts are a daily ritual. Finger-pointing becomes the company sport.

✓ Microservices

Jin reads a blog post about **microservices**. The idea is simple: instead of one giant application, split it into small, independent services. Each service does one thing well. Each service has its own codebase, its own team, its own deploy pipeline.

ShopFast decomposes the monolith into services:

- `catalog-service` — product listings and search
- `cart-service` — shopping cart management
- `payment-service` — payment processing
- `auth-service` — user authentication
- `order-service` — order management
- `email-service` — notifications

Now the payment team can deploy without touching the catalog. The catalog can scale independently during Black Friday. The recommendation engine can be written in Python. Teams are autonomous.

The monolith is dead. Long live microservices.

But now ShopFast has a new question: where do all these services *run*?

1.2 Virtual Machines vs Containers

The Infrastructure Bill Jin's first instinct is straightforward. Each microservice gets its own **Virtual Machine**. One VM for the catalog service. One for the cart. One for payments. Clean separation. Total isolation.

ShopFast spins up six VMs on their cloud provider. Each VM runs Ubuntu, has its own kernel, its own system libraries, its own copy of everything. Jin installs Java on the catalog VM, Python on the recommendation VM. Life is neat and organized.

Then the bill arrives.

Each VM, even the tiny email service that sends maybe 10 emails per hour, is running a full operating system. That is 1–2 GB of RAM just for the OS. The email service itself uses 128 MB. The operating system uses 15x more resources than the actual application.

Multiply that by six services. Then by staging and development environments. ShopFast is suddenly running 24 VMs. The monthly cloud bill goes from $800 to $6,400.

Marco looks at the bill and says, "We are paying rent for 24 apartments when we only need 6 desks."

He is exactly right.

↻ Why VMs Are Heavy

A **Virtual Machine** runs a complete operating system on top of a **hypervisor**. The hypervisor (like VMware ESXi, KVM, or Hyper-V) sits between the hardware and the guest operating systems, virtualizing the CPU, memory, storage, and network for each VM.

The cost of this isolation:

- Each VM boots its own kernel — takes 30–60 seconds to start
- Each VM runs its own OS processes (`systemd`, `sshd`, `cron`, etc.)
- Each VM reserves its own RAM for the OS (typically 512 MB to 2 GB)
- Each VM disk image includes the full OS (several gigabytes)

The isolation is excellent. A crash in one VM cannot touch another. But the overhead is brutal.

✓ A Lighter Way: Containers

What if you could get the isolation of separate environments *without* running separate operating systems?

That is the core idea behind **containers**. Instead of virtualizing the hardware and running a full OS for each workload, containers share the host operating system's kernel. They use kernel-level features to create isolated environments that *feel* like separate machines but are really just walled-off processes on the same OS.

A container starts in milliseconds, not minutes. It uses megabytes, not gigabytes. And you can run dozens of them on a machine that could barely handle four VMs.

Jin hears about this technology called **Docker**. But before understanding Docker, we need to understand what a container actually *is* under the hood.

1.3 What is a Container?

A container is not a magical black box. It is a regular Linux process that has been given three things: **resource limits**, **isolation**, and a **bundled filesystem**. These come from three Linux kernel features that have existed for years.

1.3.1 Namespaces: The Walls

Linux namespaces make a process believe it is alone on the machine. There are several types:

- **PID namespace** — the process sees itself as PID 1. It cannot see processes outside its namespace.
- **Network namespace** — the process gets its own network stack, its own IP address, its own ports.
- **Mount namespace** — the process sees its own filesystem tree, separate from the host.
- **UTS namespace** — the process can have its own hostname.
- **User namespace** — the process can have its own set of user and group IDs.
- **IPC namespace** — isolated inter-process communication resources.

From inside the container, the process believes it is running on its own machine. From outside, it is just another process on the host.

1.3.2 Cgroups: The Limits

Namespaces give isolation, but they do not prevent a container from eating all the CPU or RAM on the host. That is what **cgroups** (control groups) do. Cgroups let you say:

- "This process can use at most 512 MB of RAM."
- "This process can use at most 0.5 CPU cores."
- "This process can do at most 100 disk I/O operations per second."

Without cgroups, one misbehaving container could starve every other container on the host. Cgroups are the bouncer.

1.3.3 Union Filesystems: The Layers

The third piece is the **union filesystem** (like OverlayFS). This is what makes container images so efficient. Instead of each container having a full copy of the OS filesystem, images are built in **layers**.

A base Ubuntu layer might be 70 MB. A Java runtime layer adds 200 MB on top. Your application JAR adds 50 MB. These layers are *shared*. If ten containers all use the same Ubuntu + Java base, that base exists only once on disk. Each container adds only its own thin application layer on top.

1.3.4 The Full Picture

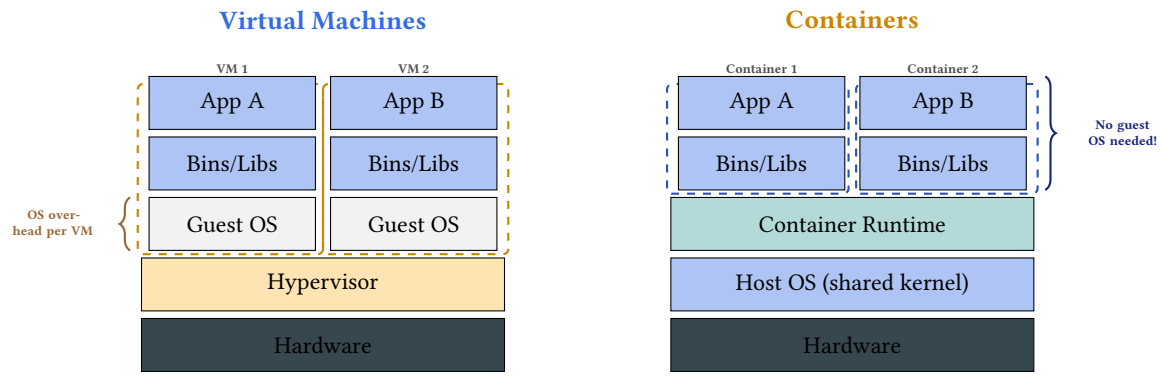


Figure 1.1: Virtual machines vs containers

△ Containers Are Not VMs

A common misconception: containers are "lightweight VMs." They are not. A VM virtualizes hardware and runs a full kernel. A container is a process on the host, isolated by kernel features. This means containers share the host kernel. If the kernel has a vulnerability, all containers are affected. Containers trade some isolation for massive efficiency gains.

▷ Try It Yourself

Run this now to see that a container is just a process on the host:

```
docker run -d --name test-container nginx:alpine
docker top test-container
docker rm -f test-container
```

You should see the Nginx process listed with its PID on the host. This confirms that containers are not VMs — they are isolated processes sharing the host kernel.

1.4 Docker Basics

ShopFast Meets Docker Jin reads the Docker documentation over a weekend. On Monday morning, she walks into the office and says: "I can package each microservice into something that starts in under a second, uses a fraction of the RAM, and runs exactly the same on my laptop, in staging, and in production."

Marco says, "That sounds too good to be true."

Jin opens her terminal and shows them.

Docker is the tool that made containers accessible to ordinary developers. Before Docker, containers existed (LXC had been around for years), but they were difficult to use. Docker gave us four key abstractions:

1.4.1 Images

A **Docker image** is a read-only template containing everything your application needs to run: the code, the runtime, libraries, environment variables, and configuration files. Images are built in layers using a **Dockerfile**.

1.4.2 Dockerfile

A **Dockerfile** is a recipe for building an image. Here is the Dockerfile for ShopFast's catalog service:

```
# Start from an official Java base image
FROM openjdk:17-slim
```

```
3
4  # Set the working directory inside the container
5  WORKDIR /app
6
7  # Copy the application JAR file
8  COPY target/catalog-service.jar app.jar
9
10 # Expose the port the service listens on
11 EXPOSE 8080
12
13 # Define the command to run the application
14 CMD ["java", "-jar", "app.jar"]
```

Listing 1.1: Dockerfile for catalog-service

Each line creates a layer. The `FROM` line pulls a base image with Java pre-installed. The `COPY` line adds the application. The result is a self-contained, portable package.

1.4.3 Containers

A **container** is a running instance of an image. You can run many containers from the same image, just like you can start many processes from the same binary.

```
1  # Build the image
2  docker build -t shopfast/catalog-service:1.0 .
3
4  # Run a container from the image
5  docker run -d -p 8080:8080 --name catalog \
6    shopfast/catalog-service:1.0
7
8  # Check it is running
9  docker ps
10 # CONTAINER ID   IMAGE                            STATUS
11 # a1b2c3d4e5f6   shopfast/catalog-service:1.0      Up 2 seconds
```

Listing 1.2: Running the catalog service

Two seconds from "start" to "running." Not two minutes. Two seconds.

1.4.4 Registries

A **registry** is a storage and distribution system for Docker images. Think of it as a package repository, but for containers. **Docker Hub** is the default public registry. Companies run private registries (like Amazon ECR, Google GCR, or a self-hosted Harbor) to store proprietary images.

```
1  # Tag the image for your private registry
2  docker tag shopfast/catalog-service:1.0 \
3    registry.shopfast.io/catalog-service:1.0
4
5  # Push it
6  docker push registry.shopfast.io/catalog-service:1.0
```

Listing 1.3: Pushing an image to a registry

Now any server can pull this image and run the exact same container. No more "works on my machine." The image *is* the machine.

✓ The Dev/Prod Parity Promise

Before Docker, production environments were snowflakes. Each server had slightly different library versions, slightly different configurations, slightly different kernel parameters. Bugs that appeared in production but not in development were a constant plague.

Docker eliminates this entire category of bugs. If it runs in the container on your laptop, it will run in the

container in production. The container carries its entire world with it.

The Honeymoon ShopFast dockerizes all six microservices in a week. Deploys go from 4 hours to 4 minutes. Each developer can run the entire stack on their laptop with `docker-compose up`. The staging environment is identical to production. Bugs caused by environment differences disappear.

Priya says, "Docker is the best thing that ever happened to us."

She is right. For now.

The team celebrates. They hire more engineers. They build more services. 12 microservices become 25. Then 47.

And slowly, quietly, a new kind of chaos begins.

1.5 The Orchestration Problem

3 AM, a Tuesday in November Jin's phone buzzes at 3:07 AM. The monitoring system is screaming. The checkout flow is completely broken. Customers are seeing 502 errors.

She logs into the server. The `payment-service` container crashed 40 minutes ago. Nobody restarted it. It is just *gone*. The health check? There is no health check. Jin runs `docker start payment` and goes back to bed.

Thursday. The same week. Traffic spikes because a tech blog featured ShopFast. The `catalog-service` buckles under load. Jin needs more instances. She SSH-es into three different servers, pulls the image, and runs `docker run` three times manually. It takes 20 minutes. By then, the traffic spike is over and customers have left.

Friday. Server `prod-07` has a hardware failure. It hosted 12 containers. All of them are gone. Which 12? Jin does not remember. She checks her spreadsheet. Yes, a spreadsheet. That is how she tracks which containers run on which servers.

She rebuilds everything from the spreadsheet. It takes four hours. Two services were missing from the spreadsheet. Nobody notices until Monday.

✕ Managing Containers at Scale is Chaos

Docker solves the packaging problem. It does *not* solve the operations problem. With 47 microservices running across 20 servers, ShopFast faces:

- **No self-healing.** A container crashes. Nothing restarts it. Customers suffer until a human notices and intervenes.
- **No auto-scaling.** Traffic spikes are handled by a human SSH-ing into servers and manually running containers.
- **No intelligent scheduling.** Jin decides which server gets which container by gut feeling. Some servers are overloaded. Others sit idle.
- **No service discovery.** When `order-service` needs to talk to `payment-service`, it uses a hardcoded IP address. If the payment container moves to a different server, the IP changes. Everything breaks.
- **No rolling updates.** Deploying a new version means stopping the old container and starting the new one. There is a gap. Requests are dropped.
- **No single source of truth.** The state of the infrastructure lives in Jin's head and a spreadsheet. There is no brain. No automation. No orchestration.

Docker gave ShopFast perfect containers. But nobody is *managing* those containers. There is no brain. No automation. No orchestration.

ShopFast does not need a better container. It needs a **container orchestrator**.

1.6 What is Kubernetes?

Meanwhile, at Google While ShopFast struggles with 47 containers, Google has been running *billions* of containers per week internally for over a decade. Their secret? An internal system called Borg.

Borg manages Google's entire infrastructure. Search, Gmail, YouTube, Maps—all running as containers managed by Borg. Engineers do not SSH into servers. They do not manually start processes. They write a configuration file that says, "I want 50 instances of this service, each with 2 GB of RAM," and Borg makes it happen.

In 2014, Google decides to share this wisdom with the world. They do not open-source Borg directly (it is too tightly coupled to Google's internal systems). Instead, they build a clean-room implementation of the same ideas. They call it **Kubernetes**—Greek for "helmsman," the person who steers the ship.

The project is donated to a newly formed foundation: the **Cloud Native Computing Foundation** (CNCF). It is open source from day one.

Kubernetes (often abbreviated as `kubectl`—the 8 stands for the eight letters between the 'K' and the 's') is a **container orchestration platform**. It automates the deployment, scaling, and management of containerized applications.

The core philosophy is radical in its simplicity:

✓ The Declarative Promise

You tell Kubernetes *what* you want. Kubernetes figures out *how* to make it happen.

You do not say: "SSH into server 7, pull image v2.1, stop the old container, start a new one on port 8080."

You say: "I want 3 replicas of my catalog service running version 2.1, each with 512 MB of RAM."

Kubernetes reads your declaration, compares it to the current state of the world, and takes whatever actions are needed to make reality match your desire. If a container crashes, Kubernetes restarts it. If a server dies, Kubernetes moves the containers to a healthy server. If you change the desired version, Kubernetes performs a rolling update.

You declare the *what*. Kubernetes handles the *how*.

This is called the **declarative model**, and it is powered by a continuous **reconciliation loop**:

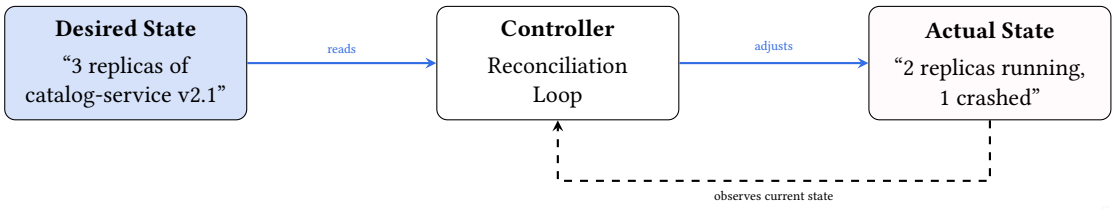


Figure 1.2: Kubernetes reconciliation loop

↻ The Reconciliation Loop

The reconciliation loop is the heartbeat of Kubernetes. It runs continuously for every resource in the cluster:

1. **Observe** — What is the current actual state?
2. **Compare** — Does it match the desired state?
3. **Act** — If not, take action to close the gap.
4. **Repeat** — Go back to step 1. Forever.

This is fundamentally different from imperative scripting. An imperative script says "do these steps." If it

fails halfway through, you are left in an unknown state. The declarative model is self-healing by nature. It does not care how it got into the current state. It only cares about the *gap* between desired and actual, and it will keep trying to close that gap.

Let us see what Kubernetes's declarative model looks like in practice. Here is how ShopFast would declare their catalog service:

```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: catalog-service
spec:
  replicas: 3
  selector:
    matchLabels:
      app: catalog
  template:
    metadata:
      labels:
        app: catalog
    spec:
      containers:
      - name: catalog
        image: registry.shopfast.io/catalog-service:2.1
        resources:
          requests:
            memory: "512Mi"
            cpu: "250m"
          limits:
            memory: "1Gi"
            cpu: "500m"
        ports:
        - containerPort: 8080
```

Listing 1.4: A Kubernetes Deployment manifest (preview)

That is it. Three replicas. Version 2.1. 512 MB of RAM each. Kubernetes takes this file and makes it real. If a container crashes, Kubernetes replaces it. If a server dies, Kubernetes reschedules the containers elsewhere. If you change `2.1` to `2.2` and reapply the file, Kubernetes performs a zero-downtime rolling update.

Jin stares at this YAML file and realizes: this is the brain she has been missing. This is the thing that manages the containers while she sleeps.

1.7 Kubernetes vs Other Orchestrators

Kubernetes was not the only container orchestrator. Several competitors emerged around the same time. But the market has spoken clearly.

Feature	Kubernetes	Docker Swarm	Apache Mesos
Complexity	Steep learning curve	Very simple to start	Very complex
Scaling	Thousands of nodes	Hundreds of nodes	Tens of thousands
Ecosystem	Massive (CNCF)	Limited	Limited
Cloud support	All major clouds (EKS, GKE, AKS)	Minimal	Minimal
Community	Largest in cloud-native	Declining	Declining
Status (2024+)	Industry standard	Effectively deprecated	Niche use only

↻ Why Kubernetes Won

Kubernetes won for three reasons that had nothing to do with technical superiority alone:

1. Google's credibility. When the company that runs the largest infrastructure on Earth says "this is how you should manage containers," people listen. Google's endorsement was priceless.

2. The CNCF and vendor neutrality. By donating Kubernetes to the CNCF, Google ensured it was not "Google's product." AWS, Azure, Red Hat, VMware, and hundreds of other companies could contribute and build on it without fear of vendor lock-in. This created a gravitational pull that no competitor could match.

3. The ecosystem effect. Once Kubernetes became the standard, every tool in the cloud-native world built integrations for it. Service meshes, monitoring systems, CI/CD pipelines, security scanners—they all speak Kubernetes first. Choosing a different orchestrator means giving up this ecosystem.

Docker Swarm was simpler, but simplicity was not enough. Enterprise customers needed the features and ecosystem that Kubernetes offered. Docker Inc. eventually integrated Kubernetes into Docker Desktop and pivoted away from orchestration entirely.

Apache Mesos could scale to tens of thousands of nodes, but it was complex to operate and its community never reached critical mass. Companies like Twitter and Apple used it, but the broader market chose Kubernetes.

HashiCorp Nomad remains a respectable alternative, especially for teams that use other HashiCorp tools (Vault, Consul, Terraform). It is simpler than Kubernetes and can orchestrate non-container workloads. But its ecosystem is a fraction of the size.

★ Should You Always Use Kubernetes?

No. Kubernetes is powerful, but it is also complex. If you have 3 services and a small team, Docker Compose or a simple PaaS (like Heroku or Railway) might be all you need. Kubernetes shines when you have dozens of services, multiple teams, and real scaling requirements. Do not use a battleship to cross a pond.

The Decision ShopFast's CTO calls a meeting. Jin presents the orchestration options. The team debates for an hour. In the end, the decision is unanimous: Kubernetes.

Not because it is trendy. Because they have lived through the pain. They have felt the 3 AM pages. They have managed containers with spreadsheets. They have watched servers die and take services with them.

They do not want a tool. They want a *platform*. Something that manages their containers, heals itself, scales automatically, and lets them sleep through the night.

"Alright," says Jin. "Let us learn Kubernetes. Starting with its architecture."

She opens Chapter 2.

▷ Chapter 1 Exercises

1. **Write and build a Dockerfile.** Create a simple Dockerfile that starts from `nginx:alpine`, copies a custom `index.html` into `/usr/share/nginx/html/`, and exposes port 80. Build the image with `docker build -t my-nginx:v1 .` and verify it exists with `docker images | grep my-nginx`.
2. **Run a container and inspect it.** Run your image with `docker run -d -p 8080:80 --name test-nginx my-nginx:v1`. Verify it is serving your page with `curl http://localhost:8080`. Then inspect resource usage with `docker stats test-nginx --no-stream`. Clean up with `docker rm -f test-nginx`.
3. **Tag and push to a registry.** Create a free Docker Hub account (or use a local registry with `docker run -d -p 5000:5000 registry:2`). Tag your image with `docker tag my-nginx:v1 localhost:5000/my-nginx:v1` and push it with `docker push localhost:5000/my-nginx:v1`. Verify the push succeeded by removing the local image and pulling it back: `docker rmi localhost:5000/my-nginx:v1 && docker pull localhost:5000/my-nginx:v1`.
4. **Explore Linux namespaces.** Run a container in the background: `docker run -d --name ns-test nginx:alpine sleep 3600`. From the host, find the container's PID with `docker inspect --format`

`'{{.State.Pid}}' ns-test`. Then compare the process view inside vs outside: run `docker exec ns-test ps aux` (container sees only its own processes) and `ps aux | grep sleep` on the host (host sees it as a regular process).

5. **Observe cgroup resource limits.** Run a container with memory limits: `docker run -d --name cgroup-test --memory=64m --cpus=0.5 nginx:alpine`. Inspect the applied limits with `docker inspect cgroup-test | grep -A5 Memory` and verify the container respects them with `docker stats cgroup-test --no-stream`. Clean up with `docker rm -f cgroup-test`.

Key Concepts

- **Monolith**: A single, large application containing all functionality
- **Microservices**: Decomposing an application into small, independent services
- **Virtual Machine**: Complete OS running on virtualized hardware via a hypervisor
- **Container**: Isolated process sharing the host OS kernel, using namespaces, cgroups, and layered filesystems
- **Namespaces**: Linux kernel feature providing process isolation
- **Cgroups**: Linux kernel feature providing resource limits
- **Union filesystem**: Layered filesystem enabling efficient image sharing
- **Docker**: Platform for building, shipping, and running containers
- **Docker image**: Read-only template containing application and dependencies
- **Dockerfile**: Recipe for building a Docker image
- **Container registry**: Storage system for Docker images
- **Container orchestrator**: System for automating deployment and management of containers
- **Kubernetes**: Container orchestration platform with declarative model
- **Declarative model**: Specify desired state; system figures out how to achieve it
- **Reconciliation loop**: Continuous process of observing, comparing, and acting to match desired state

In the next chapter, we will dive into Kubernetes architecture—the control plane, the data plane, and how all the pieces fit together.

2 The Kingdom's Architecture

> "To understand the castle, you must first understand why every wall, every gate, and every guard exists."
>
> — Ancient Ops Proverb

ShopFast survived. Their containers are running in Kubernetes now. Deployments are smoother. The team sleeps better. But one morning, Priya—the lead platform engineer—gets a question from a new hire:

"So... what actually *is* Kubernetes? Like, what are all the pieces?"

Priya pauses. She realizes most of the team treats Kubernetes like a black box. They `kubectl apply` things and hope for the best. Nobody truly understands the kingdom they live in.

That changes today.

The City Analogy

Think of a Kubernetes cluster as a city. There is a city hall where all the decisions are made. There are workers who build and run things. There is a massive record book that stores every single fact about the city. There are traffic cops, matchmakers, inspectors, and construction crews.

Every component exists for a reason. Every role was born from a real problem. Let us walk through the entire city, one building at a time.

2.1 Cluster

A **Cluster** is the entire city. It is a set of machines—physical or virtual—working together as one unit. From the outside, you do not talk to individual servers. You talk to the cluster. It is a single logical system made of many parts.

Why a Cluster?

Priya remembers the old days at ShopFast. They had twelve servers. Each one was managed individually. SSH into server-7, check the logs. SSH into server-3, restart the app. It was like running twelve separate restaurants instead of one chain. No coordination. No consistency. Pure chaos.

A cluster changed everything. Now those twelve servers are one city. One API. One set of rules. One place to deploy.

What Makes a Cluster?

A Kubernetes cluster consists of two types of machines:

- **Control plane nodes** — The brains. They run the management components.
- **Worker nodes** — The muscle. They run your actual application workloads.

A minimal cluster can be a single machine running everything. A production cluster might have 3 control

plane nodes and hundreds of workers. But it is always *one cluster*. One city.

```
# See the cluster info
kubectl cluster-info

# List all nodes in the cluster
kubectl get nodes -o wide

# Get detailed cluster component status
kubectl get componentstatuses
```

Listing 2.1: Viewing your cluster information

2.2 Nodes

A **Node** is a single machine in the cluster. It can be a physical server in a data center, a virtual machine in the cloud, or even a Raspberry Pi in your garage. Each node is a worker that can run containers.

The Factory Workers

Imagine the cluster-city has factories. Each factory is a node. Some factories are big—128 GB of RAM, 64 CPUs. Some are small. But they all follow the same rules, report to the same city hall, and build whatever they are told to build.

At ShopFast, Priya has 8 nodes. Three are beefy machines for database workloads. Five are smaller ones for stateless web servers. The cluster sees them all and knows their capabilities.

Every node registers itself with the control plane and continuously reports its status: how much CPU it has, how much memory is free, how many Pods it is running, and whether it is healthy.

```
# List all nodes with details
kubectl get nodes -o wide

# Describe a specific node
kubectl describe node worker-01

# Check node resource usage
kubectl top node
```

Listing 2.2: Inspecting a node

Node Conditions

Each node reports several conditions to the control plane:

- `Ready` — The node is healthy and can accept Pods.
- `MemoryPressure` — The node is running low on memory.
- `DiskPressure` — The node is running low on disk space.
- `PIDPressure` — Too many processes on the node.
- `NetworkUnavailable` — The node's network is misconfigured.

If a node stops reporting (the default grace period is 40 seconds), the control plane marks it as `Unknown`. After 5 minutes, it starts evicting Pods from that node.

△ Nodes Are Cattle, Not Pets

In Kubernetes, you should treat nodes as disposable. Do not SSH into nodes and tweak things manually. If a node is sick, drain it, fix it or replace it, and let Kubernetes reschedule the workloads. The moment you start treating nodes as special snowflakes, you lose the entire benefit of orchestration.

▷ Try It Yourself

Run this now on your local cluster to explore its architecture:

```
kubectl cluster-info
kubectl get nodes -o wide
kubectl get pods -n kube-system
```

You should see the cluster endpoint URL, your node(s) with their roles and container runtime, and the system Pods (API server, etcd, scheduler, controller-manager, CoreDNS). This confirms the control plane components you just learned about are real, running processes.

2.3 Control Plane

The **Control Plane** is the mayor's office. It is the brain of the entire cluster. Every decision—where to place a Pod, when to restart a container, how to route traffic—originates here.

The Day City Hall Went Dark

It was a Tuesday. ShopFast had a single control plane node. An engineer accidentally filled the disk with debug logs. The control plane crashed.

The web servers kept running. Existing Pods continued serving traffic. But nothing new could happen. A deployment was stuck mid-rollout. A crashed Pod could not be restarted. Auto-scaling did not trigger even as traffic spiked. New services could not be created.

The workers kept doing their last known task, but nobody could make new decisions. The city was frozen in time.

It took 45 minutes to free disk space and restart the control plane. By then, the damage was done. Three microservices were down because their Pods had crashed and nobody was there to recreate them.

That week, Priya set up three control plane nodes. Never again.

✕ Single Control Plane

A single control plane node is a single point of failure. If it dies:

- Existing workloads continue running (the workers are autonomous).
- No new Pods can be scheduled.
- No failed Pods can be replaced.
- No scaling events can occur.
- No new deployments can happen.
- `kubectl` commands fail because the API server is unreachable.

The cluster becomes a headless body—still alive, but unable to think.

✓ Highly Available Control Plane

In production, run at least 3 control plane nodes. This gives you:

- Redundancy: if one node fails, the others take over.

- etcd quorum: with 3 nodes, you can tolerate 1 failure (you need a majority).
- Load distribution: API requests are balanced across control plane nodes.

Managed Kubernetes services (EKS, GKE, AKS) handle this for you automatically. If you run self-managed Kubernetes, this is your responsibility.

The control plane consists of several components, each with a specific job. Let us meet them one by one.

2.4 kube-apiserver

The **kube-apiserver** is the front desk of the entire cluster. Every single request—from `kubectl`, from the scheduler, from the kubelet, from controllers—goes through the API server. It is the only component that talks directly to etcd.

The Front Desk

Picture a large government building. There is one front desk. Every citizen, every department head, every inspector must check in at the front desk. Want to register a new business? Front desk. Want to look up a building permit? Front desk. Want to file a complaint? Front desk.

Nobody walks directly to the records room. Nobody bypasses the desk. The front desk verifies your identity, checks if you are allowed to make your request, validates the paperwork, and then processes it.

That is the API server. It is the single point of entry for all cluster operations.

When you run `kubectl get pods`, here is what happens:

1. Your request hits the API server.
2. The API server authenticates you (who are you?).
3. The API server authorizes you (are you allowed to list pods?).
4. The API server validates the request.
5. The API server reads from etcd and returns the result.

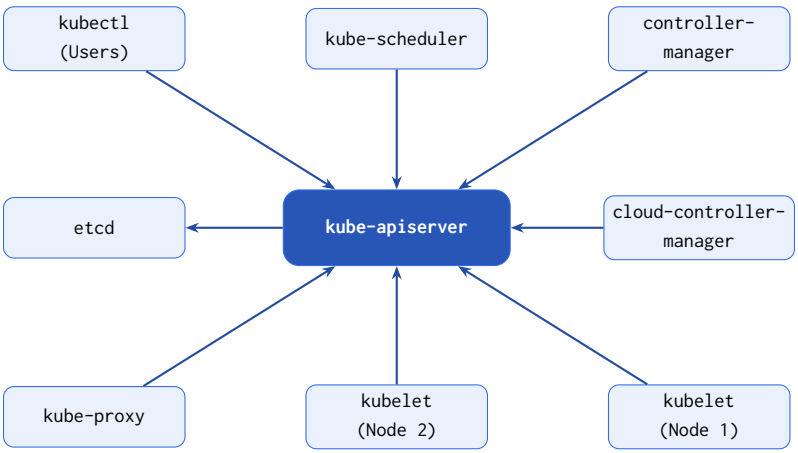
Every interaction follows this pattern. The API server is a RESTful API—everything is a resource, every operation is an HTTP verb (`GET`, `POST`, `PUT`, `DELETE`, `PATCH`).

The API Server Pipeline

Every request passes through a pipeline of stages:

1. **Authentication** — Who is making this request? (certificates, tokens, OIDC)
2. **Authorization** — Is this user allowed? (RBAC, ABAC, Webhook)
3. **Admission Control** — Should we modify or reject this? (Mutating and validating webhooks)
4. **Validation** — Is the object schema correct?
5. **etcd** — Persist the change.

Only after all stages pass does the object get stored. This pipeline is why the API server is the security gatekeeper of the cluster.



Every component communicates through the API server. Only the API server talks to etcd.

Figure 2.1: API server as central hub

★ Talking to the API Server Directly

You can bypass `kubectl` and talk to the API server directly with `curl`:

```
# Start a proxy to the API server
kubectl proxy --port=8001

# Then use curl
curl http://localhost:8001/api/v1/namespaces/default/pods
```

This is useful for debugging and understanding that Kubernetes is, at its core, just a REST API.

▷ Try It Yourself

Run this now to inspect a node in detail:

```
kubectl describe node $(kubectl get nodes -o jsonpath='{.items[0].metadata.name}')
```

Scroll through the output and find the `Conditions`, `Capacity`, and `Allocatable` sections. You should see `Ready: True` and the total CPU/memory available on the node. This confirms how the control plane tracks node health and resources.

2.5 etcd

etcd is the city's record book. It is a distributed key-value store that holds *every single piece of state* in the cluster. Every Pod definition, every Service, every Secret, every ConfigMap—all stored in etcd.

Why Not a Regular Database?

When ShopFast first heard about etcd, a junior developer asked: "Why not just use PostgreSQL?"

Good question. Here is why.

Imagine three mayors sharing one record book. If Mayor A writes something on page 47, Mayors B and C need to see it immediately. If Mayor A suddenly dies mid-sentence, the other two need to agree on what the last valid entry was. If Mayor B goes rogue and tries to write garbage, the other two must outvote him.

A regular database was not built for this. etcd uses the **Raft consensus protocol**. Every write must be agreed upon by a majority of etcd nodes before it is committed. This means:

- With 3 etcd nodes, you can lose 1 and keep running.

- With 5 etcd nodes, you can lose 2 and keep running.
- You always need a majority (quorum) to write.

This is why control plane nodes come in odd numbers. Three, five, seven. Never two, never four.

✕ What Happens if etcd Data Is Lost?

If etcd data is lost and there are no backups, *everything is lost.* Not the running containers—they will keep running until they crash. But the *desired state* is gone. Kubernetes will not know what should be running. It will not know your Deployments, your Services, your Secrets. The cluster becomes an amnesiac.

This is not a theoretical risk. It has happened to real companies. etcd backup is not optional. It is survival.

```
# Snapshot etcd data (run on a control plane node)
ETCDCTL_API=3 etcdctl snapshot save /backup/etcd-snapshot.db \
  --endpoints=https://127.0.0.1:2379 \
  --cacert=/etc/kubernetes/pki/etcd/ca.crt \
  --cert=/etc/kubernetes/pki/etcd/server.crt \
  --key=/etc/kubernetes/pki/etcd/server.key

# Verify the snapshot
ETCDCTL_API=3 etcdctl snapshot status /backup/etcd-snapshot.db \
  --write -table
```

Listing 2.3: Backing up etcd

△ etcd Performance

etcd is sensitive to disk latency. It writes ahead logs to disk on every commit. If your disk is slow, etcd is slow, and the entire cluster feels it. In production:

- Use SSD storage for etcd.
- Dedicate etcd nodes (do not run heavy workloads alongside etcd).
- Monitor etcd latency—if commit duration exceeds 100ms, investigate immediately.

↻ The Raft Consensus Protocol

Raft works by electing a leader among etcd nodes. All writes go through the leader. The leader replicates writes to followers. A write is committed only when a majority acknowledges it.

If the leader dies, followers hold an election. The candidate with the most up-to-date log wins. The new leader resumes operations. This entire process takes milliseconds.

This is why etcd can guarantee consistency even when nodes fail. It is not magic. It is math.

2.6 kube-scheduler

The **kube-scheduler** is the matchmaker. When a new Pod is created and has no node assigned to it, the scheduler's job is to find the perfect home for it.

The Housing Agent

Picture a city with a housing crisis. New residents arrive daily. Each one has requirements: "I need at least two bedrooms," "I must be near the hospital," "I cannot live next to the nightclub."

The housing agent—the scheduler—looks at every available apartment (node), checks which ones meet the requirements, scores them based on how good a fit they are, and assigns the best one.

At ShopFast, a new order-processing Pod needs 2 CPU cores and 4 GB of RAM. The scheduler checks all 8 nodes:

- Node 1: Only 1 CPU free. *Filtered out.*
- Node 2: Has 4 CPUs free and 8 GB RAM. *Score: 85.*
- Node 3: Has 2 CPUs free and 4 GB RAM. *Score: 60.*
- Node 4: Tainted for database-only workloads. *Filtered out.*

Node 2 wins. The Pod is assigned to Node 2. The kubelet on Node 2 picks it up and starts the containers.

The scheduler works in two phases:

1. **Filtering** — Eliminate nodes that cannot run the Pod. Reasons include: not enough resources, node selectors do not match, taints that the Pod does not tolerate, affinity rules that exclude the node.
2. **Scoring** — Rank the remaining nodes. Factors include: how much free resource remains after placing the Pod (spreading vs. packing), affinity preferences, topology spread constraints.

```yaml
apiVersion: v1
kind: Pod
metadata:
  name: order-processor
spec:
  containers:
  - name: processor
    image: shopfast/order-processor:v2
    resources:
      requests:
        cpu: "2"
        memory: "4Gi"
      limits:
        cpu: "4"
        memory: "8Gi"
  nodeSelector:
    disktype: ssd
  tolerations:
  - key: "dedicated"
    operator: "Equal"
    value: "high-memory"
    effect: "NoSchedule"
  affinity:
    podAntiAffinity:
      requiredDuringSchedulingIgnoredDuringExecution:
      - labelSelector:
          matchExpressions:
          - key: app
            operator: In
            values:
            - order-processor
        topologyKey: "kubernetes.io/hostname"
```

Listing 2.4: Pod with scheduling constraints

What Happens When No Node Fits?

If the scheduler cannot find any node that passes the filtering phase, the Pod stays in `Pending` state. It will sit there—indefinitely—until a node becomes available. This is one of the most common issues in Kubernetes. When you see a Pod stuck in `Pending`, the first thing to check is `kubectl describe pod <name>` and look at the Events section. The scheduler will tell you exactly why it cannot place the Pod:

```
0/8 nodes are available: 3 Insufficient cpu, 2 Insufficient memory,
3 node(s) had taint that the pod didn't tolerate.
```

2.7 kube-controller-manager

The **kube-controller-manager** is the army of bureaucrats. It runs multiple controllers, each watching a specific resource type and ensuring that the *desired state* matches the *actual state*.

The Building Inspectors

Imagine a city where every building has a blueprint on file. The building inspectors walk the streets all day, every day. They compare each building to its blueprint.

"This blueprint says there should be 3 coffee shops on Main Street. I only see 2. One must have closed. I will build a new one."

"This blueprint says the parking garage should have 5 levels. Someone added a 6th. I will tear it down."

That is what controllers do. They run in an infinite loop:

1. Observe the current state.
2. Compare it to the desired state.
3. Take action to reconcile the difference.

This is the **reconciliation loop**—the beating heart of Kubernetes.

The kube-controller-manager runs dozens of controllers in a single binary. The most important ones:

- **ReplicaSet Controller** — Ensures the right number of Pod replicas are running. If a Pod dies, it creates a new one. If there are too many, it kills the extras.
- **Deployment Controller** — Manages rollouts and rollbacks of Deployments. It creates and manages ReplicaSets.
- **Node Controller** — Monitors node health. If a node stops responding, the controller waits (grace period), then marks it as unreachable, then evicts its Pods.
- **Job Controller** — Ensures that Jobs run to completion. If a Job's Pod fails, the controller creates a new Pod to retry.
- **EndpointSlice Controller** — Populates EndpointSlice objects (linking Services to Pods).
- **ServiceAccount Controller** — Creates default ServiceAccounts for new namespaces.
- **Namespace Controller** — Cleans up resources when a namespace is deleted.

↻ The Watch Mechanism

Controllers do not poll. They *watch*. The API server supports a streaming watch protocol. When a controller starts, it says: "Tell me about every Pod that changes." The API server pushes events to the controller in real time.

This is why Kubernetes reacts so quickly. Controllers do not check every 30 seconds. They are notified instantly when something changes. The watch mechanism, combined with an informer cache, makes this both fast and efficient.

★ Seeing Controllers in Action

Want to see the reconciliation loop in action? Try this:

```
# Create a Deployment with 3 replicas
kubectl create deployment nginx --image=nginx --replicas=3

# Watch the Pods
kubectl get pods -w

# Now delete one Pod manually
kubectl delete pod <pod-name>

```

```
# Watch the ReplicaSet controller immediately create
# a replacement. That is the reconciliation loop.
```

2.8 cloud-controller-manager

The **cloud-controller-manager** is the embassy. It connects Kubernetes to the outside world—specifically, to cloud provider APIs.

The Embassy

ShopFast runs on AWS. When they create a Kubernetes Service of type `LoadBalancer`, someone needs to call the AWS API and actually create an Elastic Load Balancer. When they create a PersistentVolume, someone needs to provision an EBS volume. When a node is terminated in AWS, someone needs to update the Kubernetes node list.

That "someone" is the cloud-controller-manager. It is the ambassador between Kubernetes and the cloud provider. It speaks both languages.

The cloud-controller-manager runs cloud-specific controllers:

- **Node Controller** — Detects when a cloud VM is terminated and removes the corresponding Node object from Kubernetes.
- **Route Controller** — Configures network routes in the cloud so that Pods on different nodes can communicate.
- **Service Controller** — Creates, updates, and deletes cloud load balancers when Services of type `LoadBalancer` are created.

```yaml
apiVersion: v1
kind: Service
metadata:
  name: shopfast-web
  annotations:
    service.beta.kubernetes.io/aws-load-balancer-type: nlb
spec:
  type: LoadBalancer
  selector:
    app: shopfast-web
  ports:
  - port: 443
    targetPort: 8443
    protocol: TCP
```

Listing 2.5: A Service that triggers the cloud-controller-manager

Why a Separate Binary?

Before Kubernetes 1.6, cloud logic was embedded in the kube-controller-manager. This meant every cloud provider's code was compiled into the core Kubernetes binary. Adding support for a new cloud meant modifying Kubernetes itself.

The cloud-controller-manager was extracted so that cloud providers can develop and release their controllers independently. If you run Kubernetes on bare metal, you simply do not run a cloud-controller-manager. Clean separation.

2.9 kubelet

The **kubelet** is the foreman on each node. It is an agent that runs on every worker node, takes orders from the API server, and ensures that the containers described in Pod specs are actually running.

The Site Manager

Each construction site in the city has a site manager. The city hall does not build anything directly. It sends blueprints to site managers, and they do the actual work.

Every morning, the site manager checks: "What am I supposed to be building? Is it still standing? Does it match the blueprint?" If a wall has collapsed, the site manager rebuilds it. If a new building order arrives, the site manager starts construction.

That is the kubelet. It runs on every node. It watches for Pods assigned to its node, starts containers through the container runtime, monitors their health, and reports back to the API server.

The kubelet's responsibilities:

1. **Pod lifecycle management** — Start, stop, and restart containers as defined in Pod specs.
2. **Health checks** — Execute liveness probes, readiness probes, and startup probes. Kill and restart containers that fail liveness checks. Remove Pods from Service endpoints if they fail readiness checks.
3. **Resource monitoring** — Report CPU, memory, and disk usage to the API server via the Metrics API.
4. **Volume mounting** — Mount ConfigMaps, Secrets, PersistentVolumes, and other volume types into containers.
5. **Node status reporting** — Regularly send heartbeats and node conditions to the API server.

```yaml
apiVersion: v1
kind: Pod
metadata:
  name: shopfast-api
spec:
  containers:
  - name: api
    image: shopfast/api:v3
    ports:
    - containerPort: 8080
    livenessProbe:
      httpGet:
        path: /healthz
        port: 8080
      initialDelaySeconds: 10
      periodSeconds: 5
    readinessProbe:
      httpGet:
        path: /ready
        port: 8080
      initialDelaySeconds: 5
      periodSeconds: 3
    resources:
      requests:
        cpu: "500m"
        memory: "256Mi"
```

Listing 2.6: A Pod with health probes (managed by kubelet)

△ The kubelet Is Not a Pod

Unlike most Kubernetes components, the kubelet does *not* run as a Pod. It runs as a system service (usually via `systemd`) directly on the node's operating system. Think about it: if the kubelet ran inside a container, who would manage that container? It is the one component that must run outside of Kubernetes to bootstrap everything else.

★ Debugging kubelet Issues

If a node is misbehaving, check the kubelet logs:

```
# On the node itself
journalctl -u kubelet -f

# From your workstation, check node conditions
kubectl describe node <node-name>

# Look for conditions like:
#   Ready: False
#   MemoryPressure: True
#   DiskPressure: True
```

2.10 kube-proxy

The **kube-proxy** is the traffic cop on each node. It maintains network rules that allow Pods to communicate with each other and with the outside world through Services.

The Traffic Cop

ShopFast has a `payment-service` running on 3 Pods spread across 3 nodes. When the `checkout-service` needs to call the payment service, it does not know (or care) which Pod to talk to. It just calls `payment-service:8080`.

Someone needs to intercept that call and route it to one of the 3 actual Pod IPs. That someone is kube-proxy. It sits on every node, watches the API server for Service and Endpoint changes, and updates the node's network rules accordingly.

kube-proxy can operate in three modes:

- **iptables mode** (default) — Programs iptables rules to redirect traffic. Fast and reliable. Each Service gets a set of rules that randomly distribute traffic to backend Pods.
- **IPVS mode** — Uses Linux IPVS (IP Virtual Server) for load balancing. Better performance with large numbers of Services. Supports multiple load-balancing algorithms (round-robin, least connections, etc.).
- **nftables mode** — Uses nftables, the successor to iptables. Available from Kubernetes 1.29+. Better performance and scalability than iptables.

How iptables Mode Works

When you create a Service, kube-proxy programs iptables rules like this:

1. Traffic destined for the Service's ClusterIP is intercepted.
2. iptables randomly selects one of the backend Pod IPs (using the `statistic` module with probability-based rules).
3. The packet's destination is rewritten (DNAT) to the selected Pod IP.
4. The packet is forwarded to the Pod.

This all happens in the kernel. There is no user-space proxy. It is extremely fast. The downside? iptables rules do not scale well beyond thousands of Services—that is where IPVS shines.

```
# View kube-proxy configuration
kubectl get configmap kube-proxy -n kube-system -o yaml

# Check current iptables rules (on a node)
iptables -t nat -L KUBE-SERVICES -n | head -20

# For IPVS mode, check virtual servers
ipvsadm -Ln
```

Listing 2.7: Checking kube-proxy mode

2.11 Container Runtime

The **Container Runtime** is the actual construction crew. While Kubernetes decides *what* to build and *where* to build it, the container runtime does the actual building. It pulls images, creates containers, starts processes, and manages the low-level lifecycle.

The Construction Crew

The kubelet is the site manager. It has the blueprints. But it does not swing the hammer. It tells the construction crew: "Build this container from this image, with these environment variables, on this network, with this volume mounted." The crew does the work.

Kubernetes does not care *which* crew you use, as long as they speak the same language. That language is the **Container Runtime Interface (CRI)**.

The most common container runtimes:

- **containerd** — The industry standard. Extracted from Docker. Used by most cloud providers (EKS, GKE, AKS). Lightweight, stable, battle-tested.
- **CRI-O** — Built specifically for Kubernetes. Created by Red Hat. The default in OpenShift. Minimal—it does exactly what Kubernetes needs, nothing more.

The CRI Interface

The **Container Runtime Interface (CRI)** is a gRPC-based API that the kubelet uses to communicate with the container runtime. It defines two services:

- **RuntimeService** — Manages the lifecycle of containers and Pods (create, start, stop, remove, exec, attach).
- **ImageService** — Manages container images (pull, list, remove, image status).

Before CRI existed, Kubernetes had Docker hardcoded into the kubelet. When people wanted to use other runtimes, they had to modify Kubernetes source code. CRI solved this by creating a clean abstraction layer. Now any runtime that implements CRI can work with Kubernetes.

This is why Docker was "removed" in Kubernetes 1.24. Docker itself does not implement CRI. Kubernetes used a shim called `dockershim` to translate between CRI and Docker. When `dockershim` was removed, users switched to containerd—which was always the actual runtime inside Docker anyway.

```
# See which runtime each node uses
kubectl get nodes -o wide
# Look at the CONTAINER-RUNTIME column

# On the node itself, check containerd
crictl info
crictl ps    # List running containers
crictl images # List pulled images
```

Listing 2.8: Checking the container runtime on a node

★ containerd vs CRI-O

Both are excellent choices. If you do not have a strong preference:

- Use **containerd** if you are on a managed cloud service or want the widest community support.
- Use **CRI-O** if you are running OpenShift or want the most minimal runtime possible.

Either way, your Pods will behave identically.

2.12 The Complete Architecture

Now let us put it all together. Here is the complete Kubernetes cluster architecture—every component, every connection.

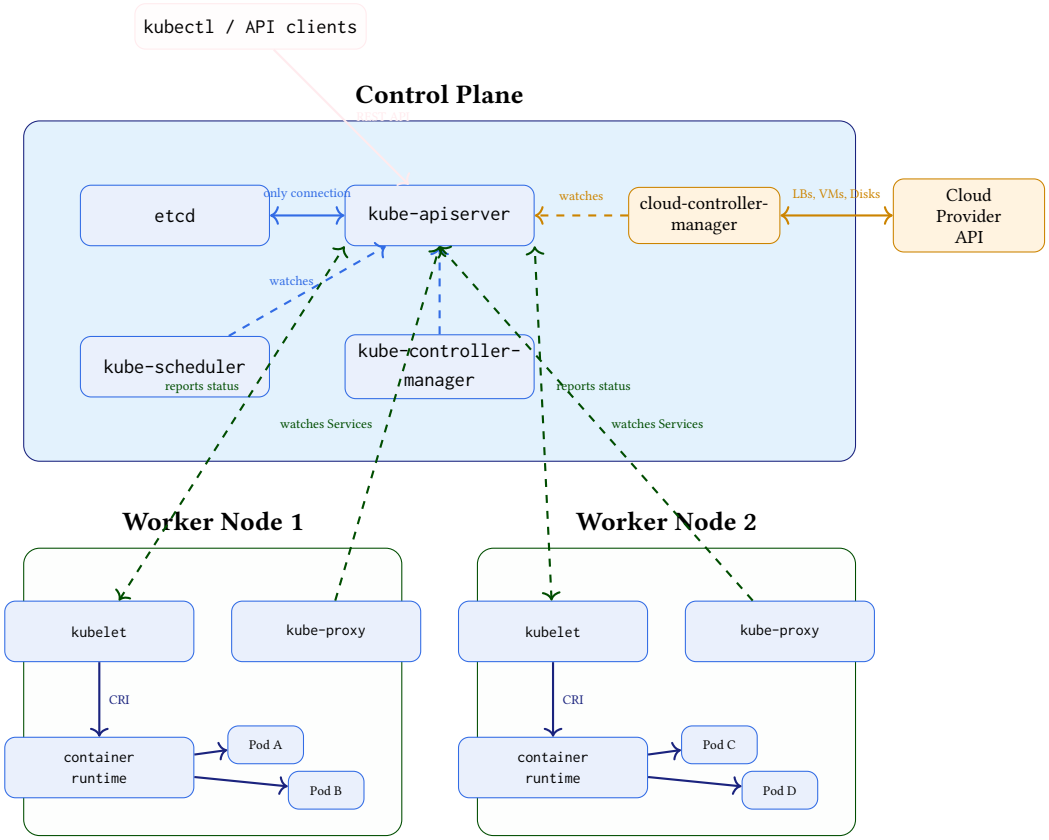


Figure 2.2: Complete Kubernetes cluster architecture

Reading the Architecture

Follow the flow:

1. A developer runs `kubectl apply -f deployment.yaml`.
2. The request hits the **kube-apiserver**, which authenticates, authorizes, validates, and stores the Deployment in **etcd**.
3. The **Deployment controller** (in kube-controller-manager) notices the new Deployment and creates a ReplicaSet.
4. The **ReplicaSet controller** notices the new ReplicaSet and creates Pod objects.
5. The **kube-scheduler** notices the unscheduled Pods and assigns each one to a node.
6. The **kubelet** on each assigned node notices the new Pod and tells the **container runtime** to pull the image and start the containers.
7. The **kube-proxy** on each node updates network rules so the Pods are reachable via Services.
8. If a **LoadBalancer** Service is involved, the **cloud-controller-manager** calls the cloud API to provision an external load balancer.

Every step is asynchronous. Every component watches the API server and reacts to changes. There is no central orchestrator barking orders. There is only desired state, actual state, and a swarm of controllers working to make them match.

★ The One Rule to Remember

In Kubernetes, the API server is the **single source of truth**. If a component needs information, it asks the API server. If a component wants to change something, it tells the API server. If you want to debug anything, start at the API server.

Everything else is a consequence of this design.

Priya finishes her whiteboard session. The new hire stares at the diagram.

"So Kubernetes is not one thing," he says. "It is a bunch of independent components all watching and reacting."

"Exactly," Priya says. "It is a city that runs itself. You just tell it what you want, and the bureaucrats make it happen."

He pauses. "What if I want to tell it something more specific? Like, I want 3 copies of my app, and I want them to restart if they crash?"

Priya smiles. "That is what we will cover next. The workloads."

▷ Chapter 2 Exercises

1. **Inspect your cluster.** Run `kubectl cluster-info` and `kubectl get nodes -o wide`. Identify the Kubernetes version, the node names, their roles (control-plane vs worker), the container runtime in use, and the internal IP addresses. If you do not have a cluster yet, create one with `minikube start` or `kind create cluster`.
2. **Explore control plane Pods.** List all Pods in the `kube-system` namespace with `kubectl get pods -n kube-system`. Identify which Pods correspond to the API server, etcd, the scheduler, and the controller manager. Pick one and run `kubectl describe pod <pod-name> -n kube-system` to examine its container image, resource requests, and liveness probes.
3. **Check node conditions and capacity.** Run `kubectl describe node <node-name>` on one of your nodes. Find the `Conditions` section and verify that `Ready` is `True` and that `MemoryPressure`, `DiskPressure`, and `PIDPressure` are all `False`. Then find the `Capacity` and `Allocatable` sections and note how much CPU and memory are available for scheduling.
4. **Talk to the API server directly.** In one terminal, start `kubectl proxy --port=8001`. In another terminal, use `curl http://localhost:8001/api/v1/namespaces/default/pods` to list Pods via the REST API. Then try `curl http://localhost:8001/api/v1/nodes` to list nodes. Observe that the responses are the same JSON objects that `kubectl` parses for you.
5. **Observe the reconciliation loop.** Create a Deployment with `kubectl create deployment nginx-test --image=nginx --replicas=2`. Watch the Pods with `kubectl get pods -w`. Delete one Pod manually with `kubectl delete pod <pod-name>`. Observe the ReplicaSet controller immediately creating a replacement. Clean up with `kubectl delete deployment nginx-test`.

3 The Citizens

> "You don't run containers in Kubernetes. You run Pods. That distinction will save your life one day."
>
> — Ravi, after his first production incident

Kubernetes has a cluster. It has nodes. It has a control plane that watches everything. But none of that matters until you put *something* on it. Something that actually does work.

This chapter is about the citizens of your cluster. The workloads. The things that run your code, serve your traffic, process your jobs, and store your data.

These are the 14 core workload objects. Every single one was born from pain.

3.1 Pods — The Atomic Unit

Ravi's First Deploy

Ravi has been running containers on his laptop for months. Docker is his best friend. He has a Node.js API server packaged into a neat little image: `ravi/api-server:v1`.

Now his company has a Kubernetes cluster. His manager says: "Deploy your app to the cluster."

Ravi thinks: "Easy. I'll just tell Kubernetes to run my container."

He tries. And immediately learns something surprising.

Kubernetes does not run containers.

✕ The Problem

Ravi tries to think in terms of containers. But Kubernetes does not schedule containers. It schedules **Pods**. A Pod is the smallest deployable unit in Kubernetes — not a container.

Why? Why add this extra layer?

✓ The Solution

A **Pod** is a wrapper around one or more containers that share the same network namespace and storage volumes. Every container in a Pod gets the same IP address, can talk to other containers in the Pod via `localhost`, and can mount the same volumes.

Think of a Pod as an apartment. The containers inside are roommates. They share the same address, the same front door, and the same kitchen. But each roommate has their own room (process space).

Here is the simplest possible Pod manifest:

```
apiVersion: v1
kind: Pod
metadata:
  name: api-server
```

```
  labels:
    app: api-server
spec:
  containers:
  - name: api
    image: ravi/api-server:v1
    ports:
    - containerPort: 3000
```

Listing 3.1: A minimal Pod

You can also create a Pod imperatively:

```
# Quick way to run a Pod
kubectl run api-server --image=ravi/api-server:v1 --port=3000

# See it running
kubectl get pods

# Get details
kubectl describe pod api-server
```

How a Pod Gets Its IP

When a Pod is scheduled onto a node, the container runtime creates a **network namespace** for it. The CNI (Container Network Interface) plugin assigns a unique IP address from the Pod CIDR range. Every container in the Pod shares this namespace. That is why they all have the same IP and can reach each other on `localhost`.

From the outside, a Pod is one IP address. From the inside, it is one or more processes sharing a network stack.

★ Pro Tip

A Pod with one container is the most common pattern. Multi-container Pods exist, but you should always ask: "Do these containers *need* to share network and storage?" If not, they belong in separate Pods.

▷ Try It Yourself

Run this now to create your first Pod and inspect it:

```
kubectl run my-first-pod --image=nginx:alpine --port=80
kubectl get pods -o wide
kubectl describe pod my-first-pod
kubectl delete pod my-first-pod
```

You should see the Pod transition to `Running` status with an assigned IP address and node. The `describe` output shows the container state, events (Scheduled, Pulling, Started), and the Pod IP. This confirms that Kubernetes wraps your container in a Pod with its own network identity.

3.2 Multi-Container Pods — Roommates with Roles

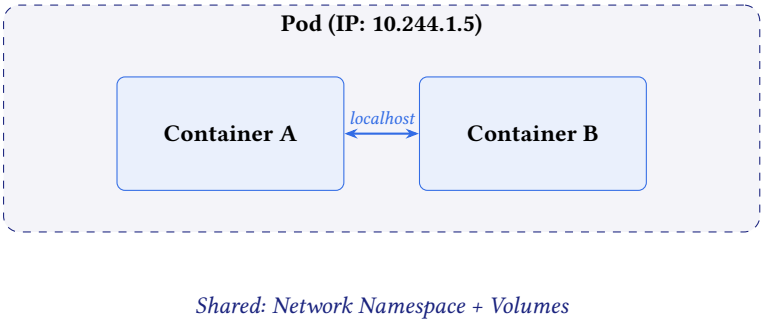


Figure 3.1: Pod with shared network namespace

But Ravi has a question. His Pod is running. It has one container. Why would he ever need more than one?

The Log Problem

Ravi's API server writes logs to a file inside its container at `/var/log/app.log`. The ops team wants those logs shipped to Elasticsearch. Ravi has two options:

1. Modify his application code to ship logs directly. But that mixes business logic with infrastructure concerns.
2. Run a separate log shipper alongside his app.

He picks option 2. But the log shipper needs access to the same log file. If it runs in a separate Pod, it cannot see the file.

Solution: run both containers in the *same* Pod, sharing a volume.

There are three canonical **multi-container Pod patterns**:

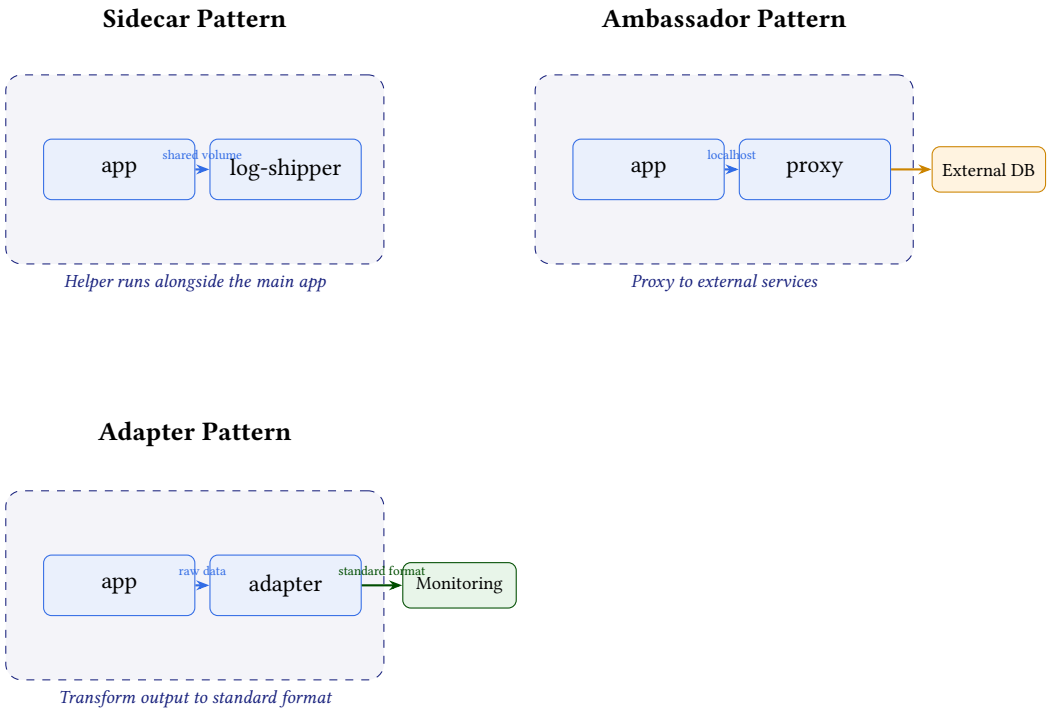


Figure 3.2: Multi-container Pod patterns

3.2.1 Sidecar Pattern

A **sidecar** container extends the main container without modifying it. Classic examples: log shippers (Filebeat, Fluentd), service mesh proxies (Envoy in Istio), and TLS termination sidecars.

```yaml
apiVersion: v1
kind: Pod
metadata:
  name: app-with-logging
spec:
  containers:
  - name: app
    image: ravi/api-server:v1
    volumeMounts:
    - name: logs
      mountPath: /var/log/app
  - name: log-shipper
    image: fluent/fluent-bit:latest
    volumeMounts:
    - name: logs
      mountPath: /var/log/app
      readOnly: true
  volumes:
  - name: logs
    emptyDir: {}
```

Listing 3.2: Pod with sidecar

3.2.2 Ambassador Pattern

An **ambassador** container proxies network traffic from the main container to external services. The app talks to `localhost`, and the ambassador handles connection pooling, retries, or protocol translation to the real backend.

3.2.3 Adapter Pattern

An **adapter** container transforms the output of the main container into a format that external systems expect. For example, converting a proprietary metrics format into Prometheus exposition format.

> **⚠ Warning**
>
> Multi-container Pods are powerful but add complexity. Every container in a Pod is scheduled together, lives together, and dies together. If your sidecar crashes, it affects the Pod. Use this pattern only when containers genuinely need to share resources.

3.3 Init Containers — The Opening Checklist

> **The Restaurant That Opened Too Early**
>
> Ravi's API server needs a database. He deploys both to Kubernetes. The API Pod starts in 2 seconds. The database Pod takes 30 seconds.
>
> For 28 seconds, the API crashes in a loop trying to connect to a database that does not exist yet.
>
> It is like a restaurant that seats customers before the kitchen is ready. The food is not coming. The customers are angry. The waiter keeps walking to the kitchen and coming back empty-handed.
>
> What Ravi needs is a checklist. Before the restaurant opens: is the kitchen ready? Are the tables set? Is the chef present? Only then: open the doors.

✓ The Solution

Init containers run before any app containers start. They run sequentially, one at a time, and each must complete successfully before the next one begins. Only after all init containers succeed do the regular containers start.

```yaml
apiVersion: v1
kind: Pod
metadata:
  name: api-server
spec:
  initContainers:
  - name: wait-for-db
    image: busybox:1.36
    command:
    - sh
    - -c
    - |
      until nc -z postgres-service 5432; do
        echo "Waiting for database..."
        sleep 2
      done
      echo "Database is ready!"
  - name: run-migrations
    image: ravi/api-server:v1
    command: ["node", "migrate.js"]
  containers:
  - name: api
    image: ravi/api-server:v1
    ports:
    - containerPort: 3000
```

Listing 3.3: Pod with init container

↻ Init Container Rules

- Init containers always run to **completion** — they are not long-running.
- They run **sequentially** — init container 2 does not start until init container 1 succeeds.
- If an init container fails, Kubernetes restarts the *entire* Pod (subject to the restart policy).
- Init containers can use different images from the app containers. Need `curl` to check a service? Use a `busybox` image even if your app is distroless.
- Init containers do *not* support readiness probes — they are not meant to serve traffic.

★ Pro Tip

Common init container use cases: waiting for a dependency, populating a shared volume with config files, running database migrations, and downloading secrets from a vault.

3.4 Ephemeral Containers — The Emergency Toolkit

The Mystery of the Distroless Pod

It is 3 AM. Ravi gets paged. A production Pod is returning HTTP 500 errors. He needs to debug it.

Ravi tries:

```
kubectl exec -it api-server -- /bin/sh
```

The response: `OCI runtime exec failed: exec failed: unable to start container process: exec: "/bin/sh": stat /bin/sh: no such file or directory`

The image is *distroless*. No shell. No `curl`. No `ps`. No debugging tools whatsoever. The container has nothing but the application binary.

Ravi is locked out of his own running application.

✕ The Problem

Distroless and minimal images are a security best practice — smaller attack surface. But when something goes wrong, you have no tools to investigate. You cannot add a shell to a running container. You cannot modify a Pod spec after creation.

✓ The Solution

Ephemeral containers are temporary containers injected into a running Pod for debugging. They are never restarted, they do not have ports, and they are not part of the Pod spec. They exist only to let you investigate.

```
# Inject a debug container into the running Pod
kubectl debug -it api-server \
  --image=busybox:1.36 \
  --target=api

# Now you have a shell with tools, sharing
# the process namespace of the 'api' container
```

Listing 3.4: Injecting an ephemeral container

↻ Under the Hood

The `-target=api` flag is crucial. It tells Kubernetes to share the process namespace with the `api` container. This means your ephemeral container can see the processes running in the target container using `ps aux`. Without this flag, you get an isolated view.

Ephemeral containers were promoted to stable in Kubernetes 1.25. They cannot be removed once added — they exist until the Pod is deleted.

△ Warning

Ephemeral containers are for debugging, not for running production workloads. They do not support resource limits, probes, or lifecycle hooks. They are a firefighting tool, not an architecture pattern.

3.5 Pod Lifecycle — From Birth to Death

Every Pod goes through a lifecycle. Understanding these phases is the difference between staring blankly at `kubectl get pods` and knowing exactly what went wrong.

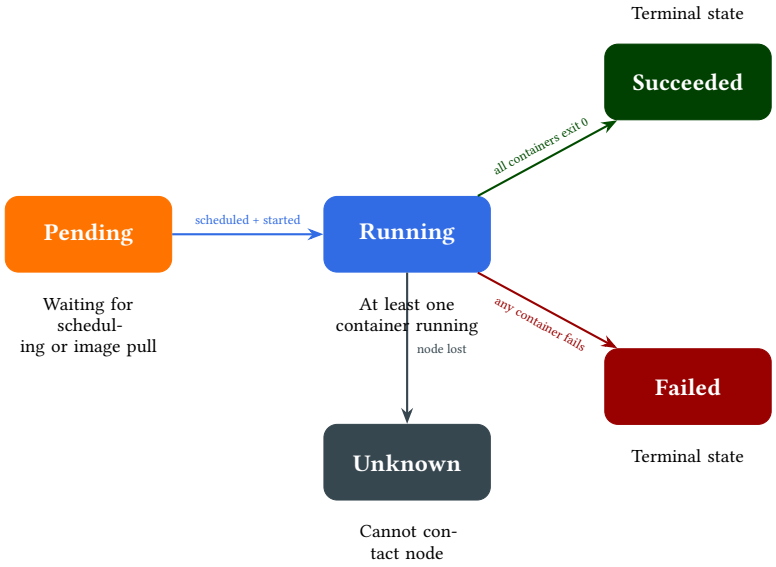


Figure 3.3: Pod lifecycle phases

3.5.1 The Five Phases

Pending The Pod has been accepted by the cluster, but one or more containers are not yet running. This includes time spent waiting for scheduling, downloading images, and running init containers.

Running At least one container is running, or is in the process of starting or restarting.

Succeeded All containers terminated with exit code 0 and will not be restarted. This is the normal end state for Jobs.

Failed All containers have terminated, and at least one exited with a non-zero code or was killed by the system.

Unknown The Pod's state cannot be determined, usually because the kubelet on the node has stopped reporting. This often means the node itself is down.

★ Pro Tip

The most common debugging question: "Why is my Pod stuck in Pending?" Three usual causes: no node has enough resources (CPU/memory), the image cannot be pulled (wrong name, no credentials), or a required PersistentVolume is not available.

3.6 Container States — Inside the Pod

While the Pod has a *phase*, each container inside it has a *state*. These are different things.

Three Roommates, Three Situations

Picture a Pod as an apartment with three roommates. The landlord (kubelet) checks on the apartment (Pod). The apartment is "Running." But inside:

- Roommate A is awake and working (**Running**).
- Roommate B is waiting for a package to arrive before they can start (**Waiting**).
- Roommate C went home for the weekend and left a note (**Terminated**).

The apartment is occupied. But the roommates are in different states.

The three container states:

- **Waiting** — The container is not yet running. It might be pulling its image, or waiting for init containers to complete. The `reason` field tells you why (e.g., `ContainerCreating`, `CrashLoopBackOff`, `ImagePullBackOff`).

- **Running** — The container is executing without issues. The `startedAt` field tells you when it started.
- **Terminated** — The container ran and exited. The `exitCode`, `reason`, and `finishedAt` fields tell you what happened.

Check container states with:

```
kubectl describe pod api-server

# Look for the "State" field under each container:
#   State:       Running
#     Started:   Mon, 15 Jan 2026 10:30:00 +0000
# # or:
#   State:       Waiting
#     Reason:    CrashLoopBackOff
```

⚠ CrashLoopBackOff

This is the most dreaded container state. It means the container starts, crashes, Kubernetes restarts it, it crashes again, and Kubernetes starts backing off — waiting longer each time before restarting (10s, 20s, 40s, up to 5 minutes). Check `kubectl logs <pod>` to find out *why* the container is crashing.

3.7 Restart Policies — When Failure Happens

To Restart or Not to Restart

Ravi has two workloads:

1. His API server, which should run forever. If it crashes, restart it immediately.
2. A database migration script, which should run exactly once. If it succeeds, do not restart it. If it fails, maybe try once more, then stop.

These two workloads have fundamentally different restart needs. A single restart strategy cannot serve both.

✓ The Solution

Kubernetes Pods support three **restart policies**:

- `Always` (default) — Always restart containers when they exit, regardless of the exit code. This is the default and is used by Deployments.
- `OnFailure` — Restart a container if it exits with a non-zero exit code. Used by Jobs.
- `Never` — Never restart. Once a container exits, it stays dead. Used for one-shot diagnostic Pods.

```
apiVersion: v1
kind: Pod
metadata:
  name: migration-job
spec:
  restartPolicy: OnFailure
  containers:
  - name: migrate
    image: ravi/api-server:v1
    command: ["node", "migrate.js"]
```

Listing 3.5: Pod with restart policy

↻ Under the Hood

The restart policy applies to *all* containers in the Pod. You cannot set different policies for different containers in the same Pod. The kubelet on the node handles restarts locally — it uses an exponential backoff (10s, 20s, 40s, ... up to 5 minutes) to prevent a broken container from consuming resources by crashing in a tight loop.

Now Ravi has a Pod running his API server. It works. But there is a problem. A big one.

3.8 ReplicaSets — Surviving the Inevitable

The Pod That Died Alone

Tuesday, 2:17 PM. Ravi's API server Pod is running on Node 3. Node 3 runs out of memory. The kernel OOM-kills Ravi's container. The Pod is gone.

Nobody notices for 20 minutes. Customers start complaining. Ravi scrambles to recreate the Pod manually.

His manager asks: "Why was there only one copy? And why didn't anything bring it back automatically?"

Ravi has no answer.

A Pod, by itself, is mortal. It is a single point of failure. When it dies, nothing recreates it. It is just... gone.

✕ The Problem

A bare Pod is a fragile thing. The node can crash. The container can OOM. The process can panic. Kubernetes does not automatically recreate Pods that are not managed by a controller. You need something that *watches* and ensures the desired number of copies always exist.

✓ The Solution

A **ReplicaSet** ensures that a specified number of identical Pod replicas are running at all times. If a Pod dies, the ReplicaSet creates a new one. If there are too many Pods (e.g., after a node comes back), it deletes the extras.

It is a self-healing mechanism. You declare "I want 3 copies," and the ReplicaSet controller makes it so — forever.

```yaml
apiVersion: apps/v1
kind: ReplicaSet
metadata:
  name: api-server
spec:
  replicas: 3
  selector:
    matchLabels:
      app: api-server
  template:
    metadata:
      labels:
        app: api-server
    spec:
      containers:
      - name: api
        image: ravi/api-server:v1
        ports:
        - containerPort: 3000
```

Listing 3.6: A ReplicaSet

How ReplicaSets Work

The ReplicaSet controller runs as part of the `kube-controller-manager`. It watches the API server for Pods matching its `selector`. The logic is beautifully simple:

1. Count the Pods that match my label selector.
2. If count < desired replicas: create new Pods from the template.
3. If count > desired replicas: delete excess Pods.
4. If count = desired replicas: do nothing.

The `selector` is what links the ReplicaSet to its Pods. The selector must match the labels in the Pod template. If they do not match, the API server will reject the manifest.

Do Not Create ReplicaSets Directly

You will almost never create a ReplicaSet by hand. Instead, you create a **Deployment**, which creates and manages ReplicaSets for you. The Deployment adds rolling updates and rollback capabilities on top of the ReplicaSet's self-healing.

Think of it this way: ReplicaSets keep your Pods alive. Deployments keep your ReplicaSets sane.

3.9 Deployments — The Art of Zero-Downtime Updates

Ravi's Worst Update

Ravi's API server v1 is running with 3 replicas via a ReplicaSet. Everything is stable. Then comes v2 with a critical bug fix.

Ravi updates the ReplicaSet's Pod template to use `ravi/api-server:v2`. But a ReplicaSet does not do rolling updates. It does not recreate existing Pods when the template changes. The old Pods keep running v1.

So Ravi deletes all three Pods manually. For 15 seconds, there are zero running Pods. Every request returns a 503. His phone rings.

"We need a way to update without downtime," his manager says. "And if the new version is broken, we need to go back to the old one. Instantly."

The Solution

A **Deployment** is a higher-level object that manages ReplicaSets. When you update a Deployment's Pod template, it creates a *new* ReplicaSet with the updated template and gradually scales it up while scaling the old ReplicaSet down. This is a **rolling update**.

If the new version is broken, you can **rollback** to the previous ReplicaSet with a single command.

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: api-server
spec:
  replicas: 3
  selector:
    matchLabels:
      app: api-server
  strategy:
    type: RollingUpdate
    rollingUpdate:
      maxSurge: 1
      maxUnavailable: 0
  template:
```

```
    metadata:
      labels:
        app: api-server
    spec:
      containers:
      - name: api
        image: ravi/api-server:v1
        ports:
        - containerPort: 3000
```

Listing 3.7: A Deployment

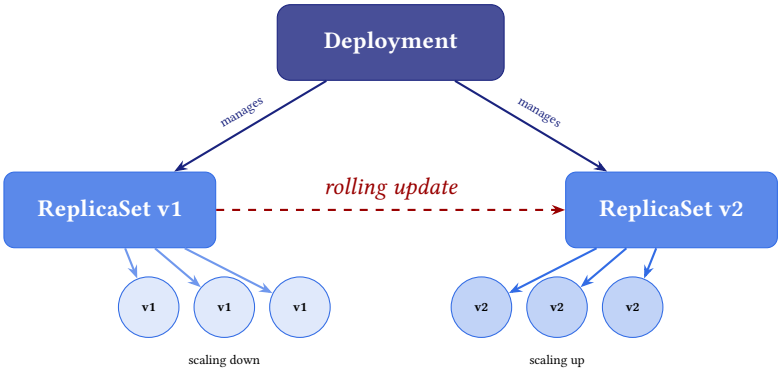


Figure 3.4: Deployment managing rolling update

Key operations:

```
# Update the image (triggers rolling update)
kubectl set image deployment/api-server \
  api=ravi/api-server:v2

# Watch the rollout
kubectl rollout status deployment/api-server

# View rollout history
kubectl rollout history deployment/api-server

# Rollback to previous version
kubectl rollout undo deployment/api-server

# Rollback to a specific revision
kubectl rollout undo deployment/api-server --to-revision=2

# Scale up
kubectl scale deployment/api-server --replicas=5
```

Rolling Update Strategy

The two key parameters are:

- `maxSurge` — How many extra Pods can exist above the desired count during the update. A value of `1` means Kubernetes can have 4 Pods during a 3-replica update.
- `maxUnavailable` — How many Pods can be unavailable during the update. A value of `0` means all 3 replicas must be running at all times.

Setting `maxSurge: 1` and `maxUnavailable: 0` gives you the safest update strategy: Kubernetes creates one new Pod, waits for it to be ready, then terminates one old Pod. Repeat until done.

The alternative strategy is `Recreate`: kill all old Pods first, then create all new Pods. This causes downtime but is useful when the old and new versions cannot coexist (e.g., incompatible database schemas).

★ Pro Tip

Always set `maxUnavailable: 0` for user-facing services. You do not want your available capacity to dip during updates. The trade-off is that updates take slightly longer.

▷ Try It Yourself

Run this now to create a Deployment and scale it:

```
kubectl create deployment web-test --image=nginx:alpine --replicas=2
kubectl get pods -l app=web-test
kubectl scale deployment web-test --replicas=5
kubectl get pods -l app=web-test
kubectl delete deployment web-test
```

You should see the Deployment start with 2 Pods, then scale up to 5 within seconds. This confirms that Deployments manage ReplicaSets, which in turn maintain the desired number of Pod replicas automatically.

3.10 StatefulSets — Identity Matters

The Database That Lost Its Name

Ravi's team needs to run a PostgreSQL cluster: one primary and two read replicas. They try a Deployment with 3 replicas.

Problem 1: The Pods get random names like `postgres-7b9d4-xk2lf`. When a Pod dies and is recreated, it gets a *new* random name. The replicas cannot find the primary because its name keeps changing.

Problem 2: The replicas need to start *after* the primary. A Deployment starts all Pods simultaneously.

Problem 3: Each Pod needs its own persistent storage. If `postgres-0` dies and comes back, it needs the *same* disk, not a new empty one.

Deployments treat Pods as interchangeable cattle. But databases are pets. They have names. They have state. They have identity.

✓ The Solution

A **StatefulSet** provides three guarantees that Deployments do not:

1. **Stable network identities**: Pods are named `<name>-0`, `<name>-1`, `<name>-2`. If `postgres-0` dies, the replacement is also called `postgres-0`.
2. **Ordered startup and shutdown**: Pods are created in order (0, 1, 2) and terminated in reverse order (2, 1, 0).
3. **Stable persistent storage**: Each Pod gets its own PersistentVolumeClaim, and the claim sticks with the Pod identity — not the Pod instance.

A StatefulSet requires a **headless Service** (a Service with `clusterIP: None`) to provide DNS entries for each Pod:

```
apiVersion: v1
kind: Service
metadata:
  name: postgres
spec:
  clusterIP: None
  selector:
    app: postgres
  ports:
  - port: 5432
```

Listing 3.8: Headless Service for StatefulSet

```yaml
apiVersion: apps/v1
kind: StatefulSet
metadata:
  name: postgres
spec:
  serviceName: postgres
  replicas: 3
  selector:
    matchLabels:
      app: postgres
  template:
    metadata:
      labels:
        app: postgres
    spec:
      containers:
      - name: postgres
        image: postgres:16
        ports:
        - containerPort: 5432
        volumeMounts:
        - name: data
          mountPath: /var/lib/postgresql/data
  volumeClaimTemplates:
  - metadata:
      name: data
    spec:
      accessModes: ["ReadWriteOnce"]
      resources:
        requests:
          storage: 10Gi
```

Listing 3.9: A StatefulSet

StatefulSet DNS

With the headless Service `postgres` and StatefulSet `postgres` in namespace `default`, each Pod gets a DNS entry:

- `postgres-0.postgres.default.svc.cluster.local`
- `postgres-1.postgres.default.svc.cluster.local`
- `postgres-2.postgres.default.svc.cluster.local`

These DNS names are *stable*. Even if `postgres-0` is killed and recreated on a different node, its DNS name stays the same. This is how replicas find the primary.

The `volumeClaimTemplates` field creates a separate PVC for each Pod. The PVC is named `data-postgres-0`, `data-postgres-1`, etc. When `postgres-0` is recreated, it reattaches to `data-postgres-0` — the same disk with the same data.

Warning

Deleting a StatefulSet does **not** delete the PVCs. This is intentional — you do not want to accidentally destroy your data. You must delete PVCs manually if you want to reclaim the storage.

★ Pro Tip

Use StatefulSets for: databases (PostgreSQL, MySQL, MongoDB), message queues (Kafka, RabbitMQ), distributed storage (Elasticsearch, Cassandra), and any workload where Pod identity matters.

Use Deployments for everything else.

3.11 DaemonSets — One Per Node, No Exceptions

The Blind Spots

The ops team asks Ravi: "Can you deploy our log collector to every node in the cluster?"

Ravi thinks: "Easy. I'll use a Deployment with 5 replicas — one for each of our 5 nodes."

But then the team adds a 6th node. No log collector runs on it. Logs from that node are lost.

Then a 7th node. Same problem.

Ravi keeps manually adjusting the replica count. This is madness.

What he needs is not "run N copies." It is "run one copy on *every* node, automatically, including nodes that join later."

✓ The Solution

A **DaemonSet** ensures that every node (or a selected subset of nodes) runs exactly one copy of a Pod. When a new node joins the cluster, the DaemonSet automatically schedules a Pod on it. When a node is removed, the Pod is garbage collected.

You do not specify a replica count. The cluster size *is* the replica count.

```yaml
apiVersion: apps/v1
kind: DaemonSet
metadata:
  name: fluentd
  namespace: kube-system
spec:
  selector:
    matchLabels:
      app: fluentd
  template:
    metadata:
      labels:
        app: fluentd
    spec:
      tolerations:
      - key: node-role.kubernetes.io/control-plane
        effect: NoSchedule
      containers:
      - name: fluentd
        image: fluent/fluentd:v1.16
        volumeMounts:
        - name: varlog
          mountPath: /var/log
        - name: containers
          mountPath: /var/lib/docker/containers
          readOnly: true
      volumes:
      - name: varlog
        hostPath:
          path: /var/log
      - name: containers
        hostPath:
```

```
33            path: /var/lib/docker/containers
```

Listing 3.10: A DaemonSet

↻ Under the Hood

DaemonSets bypass the normal scheduler by default. The DaemonSet controller adds a `nodeName` field to each Pod, which tells the scheduler exactly which node the Pod must run on.

If you want the DaemonSet to run only on specific nodes, use a `nodeSelector` or `nodeAffinity` in the Pod template.

Common DaemonSet workloads:

- Log collectors: Fluentd, Fluent Bit, Filebeat
- Monitoring agents: Prometheus Node Exporter, Datadog Agent
- Network plugins: Calico, Cilium, Weave Net
- Storage daemons: Ceph, GlusterFS

★ Pro Tip

Notice the `tolerations` field. By default, control plane nodes have a taint that prevents regular Pods from being scheduled there. If you want your DaemonSet to run on control plane nodes too (e.g., for monitoring), you need to add a matching toleration.

3.12 Jobs — Run to Completion

The Migration That Wouldn't Stop

Ravi needs to run a database migration. He creates a Deployment with 1 replica running the migration script.

The script runs. It succeeds. It exits with code 0.

Kubernetes sees the container exit and restarts it — because the Deployment's restart policy is `Always`. The migration runs a second time. And a third. And a fourth. Every time it finishes, Kubernetes "helpfully" brings it back.

Ravi has accidentally created an infinite migration loop.

✕ The Problem

Deployments are designed for long-running services that should *never* stop. But some work is finite: migrations, batch processing, report generation, data exports. These tasks should run once (or a fixed number of times) and then stop.

✓ The Solution

A **Job** creates one or more Pods and ensures that a specified number of them successfully terminate. Once the required number of completions is reached, the Job is considered complete.

Unlike Deployments, Jobs do not restart Pods that exit successfully.

```
1 apiVersion: batch/v1
2 kind: Job
3 metadata:
4   name: db-migration
5 spec:
6   completions: 1
```

```
  parallelism: 1
  backoffLimit: 3
  activeDeadlineSeconds: 600
  template:
    spec:
      restartPolicy: OnFailure
      containers:
      - name: migrate
        image: ravi/api-server:v1
        command: ["node", "migrate.js"]
```

Listing 3.11: A Job

↻ Job Parameters

- `completions` — How many Pods must succeed. Default is 1. Set to 10 if you need 10 successful runs.
- `parallelism` — How many Pods can run simultaneously. Default is 1. Set to 3 to run 3 Pods at a time.
- `backoffLimit` — How many times a failed Pod is retried before the Job is considered failed. Default is 6.
- `activeDeadlineSeconds` — Maximum time the Job can run before being terminated. A safety net for stuck Jobs.
- `ttlSecondsAfterFinished` — Auto-delete the Job (and its Pods) this many seconds after completion. Keeps your cluster clean.

With `completions: 10` and `parallelism: 3`, Kubernetes runs 3 Pods at a time, creating new ones as each finishes, until 10 total have succeeded.

```
# Check Job status
kubectl get jobs

# See the Pods created by the Job
kubectl get pods --selector=job-name=db-migration

# View logs
kubectl logs job/db-migration
```

△ Warning

Jobs require `restartPolicy` to be either `OnFailure` or `Never`. You cannot use `Always` — that would defeat the purpose. If your Pod template has `restartPolicy: Always`, the API server will reject the Job.

3.13 CronJobs — Jobs on a Schedule

The Midnight Report

The finance team wants a daily report generated at midnight. Every night. Forever.

Ravi could set up a cron job on a server. But servers fail. Engineers forget to monitor them. If the server goes down, no report.

He could write a Deployment that sleeps until midnight, runs the report, then sleeps again. But that wastes resources for 23 hours and 59 minutes every day.

What he needs is Kubernetes to create a Job automatically on a schedule — like cron, but managed by the cluster.

✓ The Solution

A **CronJob** creates Jobs on a cron schedule. At the scheduled time, it creates a new Job object, which in turn creates a Pod that runs to completion.

It is a Job factory with a timer.

```yaml
apiVersion: batch/v1
kind: CronJob
metadata:
  name: nightly-report
spec:
  schedule: "0 0 * * *"
  concurrencyPolicy: Forbid
  successfulJobsHistoryLimit: 3
  failedJobsHistoryLimit: 3
  startingDeadlineSeconds: 300
  jobTemplate:
    spec:
      template:
        spec:
          restartPolicy: OnFailure
          containers:
          - name: report
            image: ravi/report-generator:v1
            command: ["python", "generate_report.py"]
```

Listing 3.12: A CronJob

↻ CronJob Fields Explained

- `schedule` — Standard cron format: minute, hour, day-of-month, month, day-of-week. `"0 0 * * *"` means midnight every day. `"*/15 * * * *"` means every 15 minutes.
- `concurrencyPolicy` — What happens if the previous Job is still running when the next one is scheduled:
  - `Allow` (default) — Let them overlap. Multiple Jobs can run simultaneously.
  - `Forbid` — Skip the new Job if the previous one is still running.
  - `Replace` — Kill the old Job and start the new one.
- `startingDeadlineSeconds` — If the scheduled time passes and the Job has not started within this many seconds (e.g., because the cluster was down), skip it rather than running it late.
- `successfulJobsHistoryLimit` and `failedJobsHistoryLimit` — How many completed and failed Jobs to keep for debugging. Old Jobs beyond this limit are garbage collected.

★ Pro Tip

Use `concurrencyPolicy: Forbid` for Jobs that must not overlap, such as database maintenance or report generation that reads mutable data. Use `Allow` for idempotent tasks that can safely run in parallel.

```bash
# List CronJobs
kubectl get cronjobs

# See Jobs created by a CronJob
kubectl get jobs --selector=job-name=nightly-report

# Manually trigger a CronJob (useful for testing)
kubectl create job --from=cronjob/nightly-report manual-test
```

3.14 ReplicationController — The Ancestor

A Brief History Lesson

Before ReplicaSets, there was the **ReplicationController**. It did the same thing — ensure N Pods are running. But it had a critical limitation: its label selector only supported equality-based matching (`app = api-server`).

ReplicaSets added set-based selectors (`app in (api-server, web-server)`), which are far more flexible. ReplicaSets also became the foundation for Deployments.

The ReplicationController is deprecated. It exists only for backward compatibility with very old clusters. Do not use it.

⚠ Warning

If you encounter `kind: ReplicationController` in existing YAML files, migrate to `kind: Deployment`. There is no feature that a ReplicationController provides that a Deployment does not. The migration is straightforward — change the `kind`, add a `selector` with `matchLabels`, and you are done.

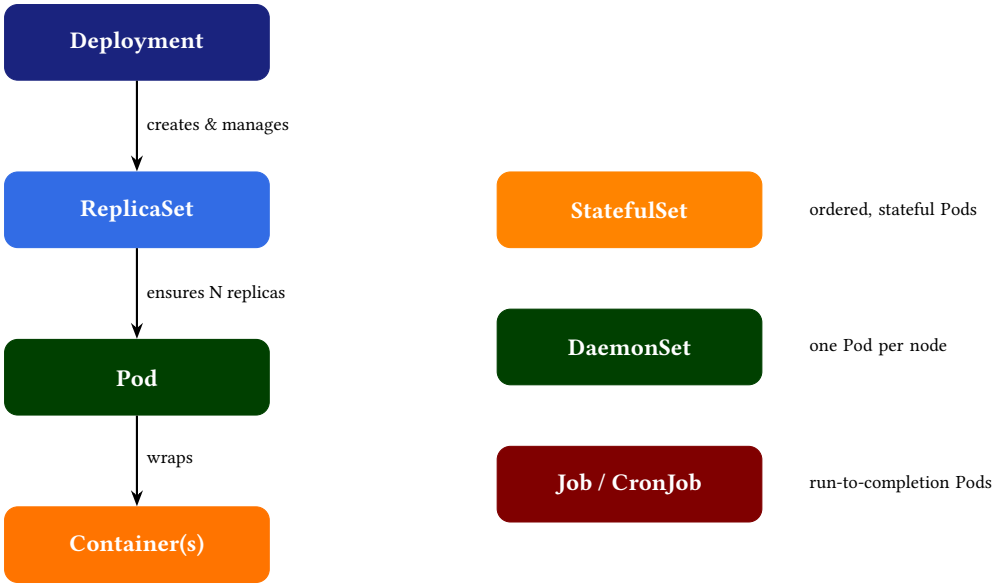


Figure 3.5: Workload resource hierarchy

Chapter Summary

In this chapter, you met the citizens of Kubernetes:

- **Pods** are the atomic unit — one or more containers sharing network and storage.
- **Multi-container Pods** use sidecar, ambassador, and adapter patterns.
- **Init containers** run setup tasks before the main containers start.
- **Ephemeral containers** let you debug running Pods that have no shell.
- **Pod lifecycle** moves through Pending, Running, Succeeded, Failed, or Unknown.
- **Container states** are Waiting, Running, or Terminated.
- **Restart policies** control whether crashed containers come back.
- **ReplicaSets** ensure N Pods are always running.
- **Deployments** add rolling updates and rollbacks on top of ReplicaSets.
- **StatefulSets** provide stable identities, ordered operations, and persistent storage.

- **DaemonSets** run exactly one Pod per node.
- **Jobs** run tasks to completion.
- **CronJobs** run Jobs on a schedule.
- **ReplicationControllers** are deprecated — use Deployments instead.

These objects cover the vast majority of what runs in any Kubernetes cluster. But having citizens is not enough. They need to talk to each other. That is the story of the next chapter.

▷ Chapter 3 Exercises

1. **Create a Pod and inspect it.** Create a Pod imperatively with `kubectl run my-pod --image=nginx:alpine --port=80`. Verify it is running with `kubectl get pods`. Inspect its details with `kubectl describe pod my-pod` and check the Pod IP, node assignment, and container state. View its logs with `kubectl logs my-pod`. Clean up with `kubectl delete pod my-pod`.
2. **Build a multi-container Pod.** Create a YAML manifest for a Pod named `multi-pod` with two containers: (1) an `nginx:alpine` container serving on port 80, and (2) a `busybox:1.36` sidecar that runs `sh -c "while true; do date >> /usr/share/nginx/html/date.txt; sleep 5; done"`. Both containers should share an `emptyDir` volume mounted at `/usr/share/nginx/html`. Apply it, then verify the sidecar is writing by running `kubectl exec multi-pod -c nginx -- cat /usr/share/nginx/html/date.txt`.
3. **Use an init container.** Create a Pod with an init container that uses `busybox:1.36` and runs `sh -c "echo 'Init complete' > /work/status.txt"`. Mount an `emptyDir` volume at `/work` in both the init container and the main container (use `nginx:alpine`). After the Pod starts, verify the init container ran first by executing `kubectl exec <pod-name> -- cat /work/status.txt`. The output should show `Init complete`.
4. **Deploy, scale, and update a Deployment.** Create a Deployment with `kubectl create deployment web --image=nginx:1.24 --replicas=3`. Verify three Pods are running. Scale to 5 replicas with `kubectl scale deployment web --replicas=5`. Then update the image with `kubectl set image deployment/web nginx=nginx:1.25`. Watch the rolling update with `kubectl rollout status deployment/web`. Finally, roll back with `kubectl rollout undo deployment/web` and confirm Pods are running the original image. Clean up with `kubectl delete deployment web`.
5. **Run a Job to completion.** Create a Job that runs `busybox:1.36` with the command `sh -c "echo 'Job done at' $(date); sleep 5"`. Set `backoffLimit: 2` and `restartPolicy: OnFailure`. Apply it and watch with `kubectl get jobs -w`. Once it shows `COMPLETIONS 1/1`, check the output with `kubectl logs job/<job-name>`. Verify the Pod is in `Completed` state with `kubectl get pods`. Clean up with `kubectl delete job <job-name>`.

You've read 3 of 25 chapters.

The story continues with:

+ Exercises, Cheatsheet, Glossary of 228 concepts