Kubernetes Fundamentals

A Complete 12-Chapter Container Orchestration Course

Topics covered:

Why Kubernetes exists & its architecture · Pods & kubectl

Deployments, ReplicaSets & Services · ConfigMaps, Secrets & storage

Ingress & external exposure · Namespaces & resource management

Health checks & self-healing · Capstone: a full multi-tier app

Exercises: 36 hands-on exercises with worked solutions

Format: A4 · Dark-theme code examples · builds directly on the site's own Docker courses

Table of Contents

CHAPTER 1 OF 12

What Kubernetes Is & Why It Exists

Kubernetes Fundamentals

Chapter 1 · What Kubernetes Is & Why It Exists

This course builds directly on this site's own `docker1` / `docker2` courses — container knowledge is assumed throughout. `docker2-8` deliberately gave only an "orientation" toward Kubernetes without teaching it. This chapter is that promised deeper dive, starting from the actual problem Kubernetes exists to solve.

The Problem: Containers at Scale

A single container is easy to manage by hand — `docker run`, or Docker Compose for a handful running together on one host. But what happens with hundreds or thousands of containers spread across many physical machines? Which host runs which container? What happens when a host dies? How do containers find each other across hosts? How do you roll out an update without downtime? How do you scale up or down automatically? None of this is solved by Docker alone — Docker manages containers on a *single* host.

What "Container Orchestration" Actually Means

Container orchestration is automating the deployment, scaling, networking, and lifecycle management of containers across a **cluster** of machines, not just one. An orchestrator makes scheduling decisions (which node runs which workload), handles failures automatically (self-healing), and provides a consistent way to describe "what should be running" — a declarative desired-state model this course returns to repeatedly, starting with Chapter 5.

A Brief History — From Borg to Kubernetes

Google's internal Borg system ran containers at massive internal scale for over a decade before Kubernetes existed. Kubernetes, released in 2014, was Google's open-sourced, redesigned answer — donated to the Cloud Native Computing Foundation (CNCF). The name is Greek for "helmsman" or "pilot," explaining both the ship's-wheel logo and the common abbreviation "K8s" (K, 8 letters, s).

Why Docker Alone Isn't Enough at Scale

Directly closing the loop on `docker2-8`'s deferral: Docker Compose handles multiple containers on *one* host well. It has no concept of multiple physical machines, no automatic failover if a host dies, no built-in rolling-update mechanism across a fleet, and no cluster-aware scheduling at all. These gaps are exactly what Kubernetes exists to close.

What Kubernetes Actually Adds

- Scheduling workloads across many nodes (Chapter 2).
- Automatically restarting or replacing failed containers (Chapters 5, 11).
- Built-in service discovery and load balancing (Chapter 6).
- Declarative rollouts and rollbacks (Chapter 5).
- Horizontal scaling, covered fully in Course 2.
- Secret and configuration management (Chapter 7).
- Storage orchestration across a cluster (Chapter 8).

Kubernetes vs. Docker — Clearing Up a Common Confusion

Kubernetes and Docker are **not** competitors. Kubernetes *orchestrates* containers; Docker (or another container runtime) actually *runs* them. Historically Kubernetes used Docker directly as its runtime; modern Kubernetes uses the Container Runtime Interface (CRI) and commonly runs `containerd` directly instead. Container images built in `docker1`/`docker2` remain fully compatible and reusable regardless — they follow the OCI (Open Container Initiative) image standard, independent of which runtime actually executes them.

> **EVERYTHING YOU LEARNED ABOUT BUILDING IMAGES STILL APPLIES, UNCHANGED**
>
> Kubernetes changes how containers are *run* and *managed* across a cluster — not how they're *built*. Every Dockerfile skill from `docker1`/`docker2` carries forward directly and unchanged into this course.

> **"KUBERNETES DEPRECATED DOCKER" — WHAT ACTUALLY HAPPENED**
>
> Kubernetes removed *dockershim*, an internal compatibility layer that let Kubernetes talk to the Docker daemon specifically, in favor of using the CRI directly — a change to Kubernetes' own internal runtime interface. Docker itself remains a widely used, fully supported tool; container images built with it remain completely usable in a Kubernetes cluster today. The "Docker is deprecated" headlines that circulated overstated a narrower, internal plumbing change into something far bigger than it actually was.

Where Kubernetes Fits in the Cloud & DevOps Landscape

`cloud1-2`'s terminology map already named the managed Kubernetes services — EKS, AKS, GKE. This course stays deliberately provider-agnostic and conceptual first, matching `cloud1`'s own cross-provider framing, before any provider-specific managed-service detail. Worth distinguishing from `cloud1-3`'s VM-level auto-scaling groups too: that's scaling whole virtual machines; Kubernetes orchestrates containers *within* a cluster of machines — related, but a genuinely different layer.

When You Don't Need Kubernetes

An honest counterpoint, matching this site's own recurring convention: a small application running on a single server, or even Docker Compose on one host, doesn't need Kubernetes' complexity. Kubernetes solves problems that only really appear at a certain scale — adopting it prematurely adds real operational overhead without a corresponding benefit. Worth naming plainly rather than treating Kubernetes as something every project automatically needs.

Hands-On Exercises

EXERCISE 1

Explain, in your own words, at least three specific problems that arise when running containers across multiple hosts that Docker Compose alone doesn't solve.

View solution

EXERCISE 2

Explain why "Kubernetes replaced Docker" is a misleading way to describe what actually changed, and state the accurate relationship between the two tools.

View solution

EXERCISE 3

A small startup runs one application on a single VM with light, predictable traffic. Would you recommend adopting Kubernetes? Justify your answer using this chapter's "when you don't need Kubernetes" material.

View solution

CHAPTER 1 QUICK REFERENCE

- Docker manages containers on **one** host; Kubernetes orchestrates them across a **cluster** of hosts
- Gaps Compose can't fill: multi-host failover, cluster-aware scheduling, fleet-wide rolling updates
- Kubernetes (2014) — Google's open-sourced answer to its internal Borg system, donated to the CNCF; "K8s" = K + 8 letters + s
- Kubernetes *orchestrates*, a runtime (containerd, formerly often Docker) *runs* — images stay OCI-compatible regardless
- "Kubernetes deprecated Docker" overstates dockershim's removal — an internal plumbing change, not Docker's obsolescence
- Kubernetes solves scale problems — adopting it for a small, single-server app adds overhead with no real benefit
- **Next chapter:** Kubernetes Architecture — control plane vs. worker nodes, API server, etcd, scheduler, kubelet

CHAPTER 2 OF 12

Kubernetes Architecture

Kubernetes Fundamentals

Chapter 2 · Kubernetes Architecture

Chapter 1 covered why Kubernetes exists. This chapter covers what it's actually made of — the components that together make cluster orchestration possible.

The Big Picture — Control Plane vs. Worker Nodes

A cluster is one or more **control plane** nodes (the "brain," making decisions) plus many **worker nodes** (where actual application containers run). This split matters practically: losing a worker node only loses the workloads running there. Losing the control plane, without redundancy, threatens the cluster's ability to make *new* decisions — but already-running workloads on healthy worker nodes keep running even if the control plane is briefly unavailable, a genuinely important operational nuance.

The Control Plane, Component by Component

- **kube-apiserver** — the front door. Every piece of communication — `kubectl`, other control plane components, external tools — goes through it. It validates and processes requests, then persists resulting state to etcd.
- **etcd** — a distributed key-value store holding the cluster's *entire* state — the single source of truth for "what should exist."
- **kube-scheduler** — watches for newly created pods with no assigned node, and decides which node each should run on, based on resource requirements and constraints (Chapter 10 goes deeper on this).
- **kube-controller-manager** — runs controller processes that continuously watch the cluster's actual state and reconcile it toward the desired state stored in etcd — the concrete mechanism behind Chapter 1's "declarative desired state" promise, explained fully below.

Worker Node Components

- **kubelet** — an agent running on *every* worker node. Talks to the API server, and ensures the containers described for that node are actually running and healthy.

- **kube-proxy** — maintains network rules on each node, enabling communication to and from pods and implementing Kubernetes' Service abstraction (Chapter 6) at the node level.
- **The container runtime** — `containerd`, per Chapter 1's own clarification — actually pulls images and starts/stops containers as instructed by the kubelet.

The Reconciliation Loop — Kubernetes' Core Mechanism

Every controller performs the same basic loop, continuously: **observe** current state → **compare** it against desired state → **act** to reconcile any difference → repeat, forever.

> **THE SINGLE MOST IMPORTANT CONCEPT IN THIS CHAPTER**
>
> This one pattern, repeated across many different controllers for many different resource types, is what makes Kubernetes self-healing and declarative at the same time. Nearly everything covered later in this course — self-healing pods (Chapters 5, 11), rolling updates (Chapter 5), autoscaling (Course 2) — is just this same loop applied to a different kind of resource. Understanding it now pays off repeatedly later.

How a kubectl Command Actually Flows Through the System

```
1. kubectl sends a request to the API SERVER
2. API server validates it and writes the desired state to ETCD
3. SCHEDULER notices an unscheduled pod, assigns it to a node,
   writes that decision back via the API server
4. KUBELET on that node notices (watching the API server) it has
   a new pod assigned
5. Kubelet instructs the CONTAINER RUNTIME to pull the image and
   start the container
6. Kubelet continuously reports the pod's actual status back
   through the API server, closing the loop
```

Every component named earlier in this chapter has a specific, concrete role in that sequence.

High Availability for the Control Plane

Production clusters typically run multiple control plane nodes — particularly multiple etcd instances forming a quorum — specifically to avoid the control plane becoming a single point of failure. Managed Kubernetes services (EKS/AKS/GKE, per `cloud1-2`'s own naming) typically handle this control-plane redundancy for you — a genuinely reassuring point for anyone worried about needing to manage this complexity by hand.

> **ETCD LOSS IS GENUINELY CATASTROPHIC**

etcd holds the cluster's entire state. Losing etcd data without a backup means losing the cluster's complete record of what should exist — a real operational risk worth taking seriously from day one, not an abstract architecture detail.

Hands-On Exercises

EXERCISE 1

If the control plane becomes temporarily unavailable while worker nodes remain healthy, explain what happens to (a) already-running application pods and (b) the ability to schedule new pods — and why these two outcomes differ.

View solution

EXERCISE 2

Walk through, step by step, what happens architecturally from the moment `kubectl apply` is run to create a new pod, to that pod actually running on a node.

View solution

EXERCISE 3

Explain the reconciliation loop concept in your own words, and give an example — not explicitly covered in this chapter — of a scenario where this loop would automatically fix a problem without any human intervention.

View solution

CHAPTER 2 QUICK REFERENCE

- **Control plane** (decisions) vs. **worker nodes** (where containers actually run) — losing the control plane doesn't stop already-running pods
- **API server** (front door) · **etcd** (entire cluster state) · **scheduler** (assigns pods to nodes) · **controller-manager** (runs the reconciliation loops)
- **kubelet** (per-node agent) · **kube-proxy** (network rules/Service abstraction) · **container runtime** (containerd — actually runs containers)
- The **reconciliation loop** — observe, compare, act, repeat — is the mechanism behind nearly everything self-healing or declarative in Kubernetes
- Full request flow: kubectl → API server → etcd → scheduler → kubelet → container runtime → status reported back
- Production clusters run redundant control planes (multi-node etcd quorum); managed services (EKS/AKS/GKE) typically handle this for you

- **Next chapter:** Pods — The Basic Unit — why not just containers directly, single vs. multi-container pods, the sidecar pattern

CHAPTER 3 OF 12

Pods — The Basic Unit

Kubernetes Fundamentals

Chapter 3 · Pods — The Basic Unit

Chapter 2 covered the architecture that schedules and runs workloads. This chapter covers the actual thing that architecture schedules: the **Pod**, Kubernetes' smallest deployable unit.

Why Not Just Containers Directly?

Kubernetes doesn't schedule or manage individual containers directly — it schedules Pods, which wrap one or more containers. This extra layer exists because some workloads genuinely need multiple tightly-coupled containers that must always be scheduled together, share network and storage, and live and die together — a Pod provides exactly that grouping boundary. A Pod is the smallest unit Kubernetes can create, schedule, or scale — there's no such thing as scheduling "half a pod."

What a Pod Actually Provides

- **Shared network namespace** — every container in a pod shares the same IP address and port space, and can reach each other via `localhost`, unlike separate containers needing a real network call to find each other.
- **Shared storage volumes** — containers in a pod can mount the same Volume and share files directly.
- Pods are generally **ephemeral** — not meant to be durable, long-lived entities managed by hand. A pod is created, and if something goes wrong, it's typically *replaced* entirely, not repaired — directly foreshadowing Chapter 5's Deployment material.

Single-Container Pods — The Common Case

The vast majority of real pods run exactly **one** container — the simplest, most common pattern. "One container per pod" is the default mental model; multi-container pods are the exception for specific reasons, not the norm — worth stating plainly to avoid the misconception that pods are usually complex, multi-container arrangements.

Multi-Container Pods & the Sidecar Pattern

When multiple containers *do* belong in one pod, the most common reason is the **sidecar pattern** — a secondary "helper" container supporting the main application container. Genuinely common real examples: a logging/log-shipping sidecar reading logs from a shared volume and forwarding them elsewhere, or a service mesh proxy sidecar (Course 2's own `k8s2-5` covers service meshes properly) intercepting network traffic for the main container.

An **init container** is a related but distinct concept: it runs to *completion* before the main containers start at all — used for setup or prerequisite tasks — rather than running alongside the main container the way a sidecar does.

The Pod Lifecycle

PHASE	MEANING
Pending	Accepted by the API server, but not yet fully scheduled/running
Running	At least one container is running
Succeeded / Failed	Terminal states for pods meant to run to completion, not indefinitely (relevant to Course 2's Jobs material)

A pod's overall phase doesn't tell the whole story — individual containers within it also have their own states worth checking separately, especially useful for troubleshooting (Course 2's `k8s2-8` builds directly on this).

Pods Are Disposable — A Genuinely Important Mental Model Shift

Revisiting Chapter 2's reconciliation loop directly: when a pod fails or its node dies, Kubernetes generally doesn't try to *repair* the existing pod — it creates a brand new one to replace it, typically with a new name and often a new IP address.

PETS VS. CATTLE

A well-known phrase in this space, worth carrying through the rest of the course: shift from a "pet" mental model (a server carefully maintained, patched, and kept running for a long time) to a "cattle" mental model (an instance you don't get attached to, replaceable at any time). Never rely on a specific pod's identity staying stable — Chapter 6's Services exist specifically to solve exactly this problem.

NEVER HARDCODE A POD'S IP ADDRESS

A pod's IP address can change the moment it's replaced — which can happen at any time, for reasons entirely outside your control. An application that hardcodes a specific pod IP will break the first time that pod is recreated, which is exactly the failure this course's Services chapter (Chapter 6) is built to prevent.

Why You Rarely Create Pods Directly

Despite pods being the fundamental unit, you'll rarely create bare Pods directly in real usage — almost always through a higher-level controller like a Deployment (Chapter 5) that manages a pod's full lifecycle, replacement, and scaling for you. This chapter deliberately focused on understanding *what* a pod is and how it behaves; Chapters 4 and 5 build directly on this foundation to show *how* you'd actually create and manage them in practice.

Hands-On Exercises

EXERCISE 1

Explain why Kubernetes schedules Pods rather than individual containers directly, and give a concrete example of when a multi-container pod (the sidecar pattern) would make sense.

View solution

EXERCISE 2

Explain the difference between an init container and a sidecar container — specifically how their timing relative to the main container differs.

View solution

EXERCISE 3

Explain the "pets vs. cattle" mental model shift in your own words, and explain specifically why an application that hardcodes a pod's IP address is likely to break in a real Kubernetes environment.

View solution

CHAPTER 3 QUICK REFERENCE

- Kubernetes schedules **Pods**, not containers directly — the smallest unit it can create/schedule/scale
- A pod gives its containers a shared network namespace (localhost) and shared storage volumes
- One container per pod is the default; multi-container pods (sidecar pattern) are the deliberate exception
- **Init containers** run to completion before the main container starts; **sidecars** run alongside it
- Pod phases: Pending → Running → Succeeded/Failed; individual containers also have their own states
- **Pets vs. cattle** — pods are disposable and replaceable, never repaired in place; never hardcode a pod IP

- Bare pods are rarely created directly — Chapter 5's Deployments manage their full lifecycle instead
- **Next chapter:** kubectl & Your First Deployment — CLI basics, imperative vs. declarative, YAML manifests, namespaces

CHAPTER 4 OF 12

kubectl & Your First Deployment

Kubernetes Fundamentals

Chapter 4 · kubectl & Your First Deployment

Chapter 3 covered what a pod is. This chapter covers the actual tool and workflow used to create and manage resources — pods and everything else — in a real cluster.

Getting Connected — kubeconfig

`kubectl` needs to know which cluster to talk to and how to authenticate — a **kubeconfig** file (typically `~/.kube/config`) holds this connection info. **Contexts** let you switch between multiple clusters — dev/staging/prod, or different providers entirely, echoing `cloud1-2`'s own cross-provider framing — via `kubectl config get-contexts` and `kubectl config use-context`. A genuinely easy-to-overlook, practical detail worth naming early: running a command against the wrong context by mistake is a real, common gotcha.

kubectl Basics

Revisiting `cloud2-1`'s own console/CLI/API point, specifically for Kubernetes: `kubectl` talks to the API server (Chapter 2's architecture) exactly like any other client. Four commands used constantly:

- `kubectl get pods` — list resources.
- `kubectl describe pod <name>` — detailed information about one resource.
- `kubectl logs <name>` — a container's log output.
- `kubectl delete pod <name>` — remove a resource.

`kubectl get` works across many resource types — pods, deployments, services, nodes, and more, covered as this course goes on.

Imperative vs. Declarative — Two Ways to Work

Imperative commands tell `kubectl` what to do right now — `kubectl run`, `kubectl create deployment` — fast for quick testing, but leaving no reusable record of what was done. The **declarative** approach instead

writes a YAML manifest describing the desired state, then runs `kubectl apply -f file.yaml` — Kubernetes' own reconciliation loop (Chapter 2) takes it from there.

This is exactly `cloud1-11`'s declarative-vs-imperative IaC distinction, applied here to Kubernetes objects instead of general cloud infrastructure. Practical guidance: imperative commands are fine for quick exploration, but real, reusable, version-controllable work (`git1`-`git3`) should be declarative YAML.

Anatomy of a YAML Manifest

```yaml
apiVersion: v1
kind: Pod
metadata:
  name: my-first-pod
  labels:
    app: demo
spec:
  containers:
    - name: web
      image: nginx:latest
      ports:
        - containerPort: 80
```

- **apiVersion** — which version of the Kubernetes API this resource type belongs to.
- **kind** — the resource type (Pod, Deployment, Service, and so on).
- **metadata** — identifying information: name, and **labels** — key-value tags that later chapters use to connect resources together (Chapter 6's Services select pods by label).
- **spec** — the actual desired configuration — containers, images, ports, and everything specific to that resource type.

This structure recurs, largely unchanged in shape, across every YAML example the rest of this course uses.

kubectl apply — Your Main Tool Going Forward

`kubectl apply` creates a resource if it doesn't exist, or updates it if it does, by diffing against the last-applied configuration.

> **THIS IS WHAT MAKES YAML MANIFESTS SAFELY RE-RUNNABLE**
>
> Applying the same manifest repeatedly is safe and **idempotent** — running it twice with no changes produces the same result as running it once, with no errors or duplicate resources. This is exactly the property real infrastructure-as-code tooling needs, and it's the opposite of `cloud1-11`'s manual-fix-causes-drift gotcha — `apply` is specifically designed to reconcile safely, every time.

Namespaces — Organizing a Cluster

A namespace is a logical partition *within* a single cluster — not a separate cluster. Used to separate teams, environments, or projects sharing one cluster. A `default` namespace always exists, but real usage typically creates dedicated ones. Resource names must be unique *within* a namespace, not across the whole cluster. Some resources are cluster-scoped (like Nodes) rather than namespace-scoped — which is why not every `kubectl get` command needs a namespace flag.

RUNNING A COMMAND AGAINST THE WRONG CONTEXT IS A REAL, SERIOUS MISTAKE

Accidentally running a delete or apply command while pointed at production, thinking you're in a dev context, is a genuinely common and potentially serious error. Always confirm the current context (`kubectl config current-context`) before anything destructive, especially in a cluster where production and non-production environments share tooling.

A Practical First Workflow

```
kubectl config current-context          # confirm where you're pointed
kubectl apply -f my-first-pod.yaml       # create it
kubectl get pods                         # confirm it's running
kubectl describe pod my-first-pod        # detail
kubectl logs my-first-pod                # see its output
kubectl delete -f my-first-pod.yaml      # clean up, using the same file
```

A genuinely satisfying full loop for a first real interaction with a cluster.

Hands-On Exercises

EXERCISE 1

Explain the difference between imperative and declarative approaches to creating a Kubernetes resource, and explain which approach is more appropriate for something you intend to keep and reuse, and why.

View solution

EXERCISE 2

Given a YAML manifest's basic structure (`apiVersion` / `kind` / `metadata` / `spec`), explain what each of these four top-level fields is generally responsible for.

View solution

EXERCISE 3

Explain why running `kubectl apply -f manifest.yaml` repeatedly, with no changes to the file, is safe and doesn't cause any problems. What specific property of `apply` makes this true?

View solution

CHAPTER 4 QUICK REFERENCE

- **kubeconfig/contexts** — which cluster you're talking to; always confirm before anything destructive
- Core commands: `get`, `describe`, `logs`, `delete`
- **Imperative** (quick, no record) vs. **declarative** (YAML + `apply`, reusable/version-controllable) — same distinction as `cloud1-11`'s IaC material
- YAML structure: **apiVersion**, **kind**, **metadata** (name/labels), **spec** (desired config) — recurs throughout the course
- `kubectl apply` is **idempotent** — safe to re-run with no changes, unlike manual console-style drift
- **Namespaces** partition one cluster (not separate clusters); names are unique per-namespace, not cluster-wide
- **Next chapter:** Deployments & ReplicaSets — declarative desired state, rolling updates, rollback, self-healing

CHAPTER 5 OF 12

Deployments & ReplicaSets

Kubernetes Fundamentals

Chapter 5 · Deployments & ReplicaSets

Chapters 3 and 4 both pointed here: the controller you'll actually use almost all the time, rather than bare Pods. This chapter covers the Deployment, and the ReplicaSet mechanism underneath it.

The Problem With Bare Pods, Revisited

A bare Pod, once created, isn't automatically replaced if it fails or its node dies — unless something is actively watching it. You could theoretically write your own reconciliation logic by hand; Kubernetes already provides it, via Deployments.

ReplicaSets — Ensuring N Copies Are Always Running

The layer directly underneath Deployments. A ReplicaSet's job is simple and singular: ensure a specified *number* of pod replicas matching a given template are running at all times — a concrete instance of Chapter 2's reconciliation loop (observe how many matching pods exist, compare to the desired count, create or delete pods to reconcile). ReplicaSets use **label selectors** (Chapter 4's labels material) to know which pods belong to them — a genuinely important mechanism that explains a lot of later behavior.

Deployments — Managing ReplicaSets for You

A Deployment sits one layer above a ReplicaSet: you rarely create a ReplicaSet directly — you create a Deployment, and Kubernetes creates and manages the ReplicaSet(s) underneath it automatically. A Deployment's real value is managing the *transition* between different versions of a ReplicaSet — exactly what makes rolling updates and rollback possible.

Rolling Updates — Changing Versions Without Downtime

When a Deployment's pod template changes (a new container image version, for instance), Kubernetes creates a **new** ReplicaSet with the updated template, and gradually scales it up while scaling the *old* ReplicaSet

down — a few pods at a time, never all at once. This is why routine deployment updates generally don't cause downtime: healthy pods are always serving traffic throughout the transition.

Two practical, configurable knobs: **maxSurge** (how many extra pods beyond the desired count can be created during rollout) and **maxUnavailable** (how many can be unavailable at once).

Rollback — Undoing a Bad Deployment

Kubernetes keeps a history of previous ReplicaSets/revisions for a Deployment. If a new rollout turns out to be broken, `kubectl rollout undo` reverts to the previous working ReplicaSet — genuinely fast compared to manually reconstructing the old configuration by hand. `kubectl rollout status` and `kubectl rollout history` are the practical commands for checking on this.

Self-Healing in Action

Tying together Chapter 3's "pods are disposable" material concretely: if a pod managed by a ReplicaSet dies or is deleted, the ReplicaSet's own reconciliation loop notices the actual count has dropped below the desired count, and creates a replacement automatically — the concrete mechanism behind the self-healing Chapter 1 promised. This happens without writing any recovery logic at all, simply by declaring the desired replica count.

A Complete Deployment YAML, Explained

```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: web-deployment
spec:
  replicas: 3
  selector:
    matchLabels:
      app: demo
  template:
    metadata:
      labels:
        app: demo
    spec:
      containers:
        - name: web
          image: nginx:latest
          ports:
            - containerPort: 80
```

SPEC.TEMPLATE IS A POD SPEC

Everything under `spec.template` is structurally the same Pod YAML from Chapter 4, just nested inside a Deployment. Nothing about Pod YAML needs relearning here — `spec.selector.matchLabels` is what ties the Deployment to the ReplicaSet's own label-selector mechanism above, and `spec.replicas` is the desired count that reconciliation loop maintains.

APPS/V1, NOT V1 — A GENUINELY COMMON EARLY MISTAKE

Chapter 4's Pod manifest used `apiVersion: v1`. Deployments live under a different API group entirely — `apps/v1`. Using the wrong apiVersion for a given kind is a genuinely common early mistake worth watching for directly.

Scaling — Changing Replica Count

Imperatively: `kubectl scale deployment <name> --replicas=N`. Declaratively (preferred, per Chapter 4's own guidance): edit the YAML's `replicas` field and re-apply. This chapter covers manual scaling; Course 2's `k8s2-6` covers automatic scaling based on live metrics.

Hands-On Exercises

EXERCISE 1

Explain the relationship between a Deployment, a ReplicaSet, and a Pod — specifically which one manages which — and explain why you'd create a Deployment rather than a ReplicaSet or a bare Pod directly.

View solution

EXERCISE 2

Explain how a rolling update avoids downtime, referencing maxSurge/maxUnavailable and the transition between the old and new ReplicaSets.

View solution

EXERCISE 3

A pod managed by a Deployment with `replicas: 3` is accidentally deleted by a user running `kubectl delete pod` directly. Explain exactly what happens next, and why.

View solution

CHAPTER 5 QUICK REFERENCE

- **ReplicaSet** — ensures N pods matching a template exist, via label selectors and the reconciliation loop
- **Deployment** — manages ReplicaSets for you; its real value is managing the transition between versions
- **Rolling updates** — new ReplicaSet scales up while old scales down gradually; maxSurge/maxUnavailable control the pace
- **Rollback** — `kubectl rollout undo` reverts to a previous ReplicaSet revision fast
- **Self-healing** — a deleted/failed pod is automatically replaced, with zero custom recovery code, purely from declaring a desired replica count
- `spec.template` is structurally a full Pod spec — Chapter 4's YAML knowledge transfers directly
- Deployments use `apps/v1`, not `v1` — a common early mistake
- **Next chapter:** Services & Networking Basics — ClusterIP/NodePort/LoadBalancer, service discovery, in-cluster DNS

CHAPTER 6 OF 12

Services & Networking Basics

Kubernetes Fundamentals

Chapter 6 · Services & Networking Basics

Chapter 5's rolling updates and self-healing constantly create and destroy pods, each with a new IP address. This chapter covers the abstraction that solves the resulting problem — directly closing the loop on Chapter 3's warning: never hardcode a pod IP.

The Problem Services Solve, Restated

Pod IPs change constantly — a rolling update, a self-healing replacement, any pod recreation event changes it. An application can't reasonably track and update which specific IPs to talk to every time this happens. What's needed is a **stable** address that always routes to whichever healthy pods currently exist, regardless of their individual identities.

What a Service Actually Is

A Service provides a single, stable IP address and DNS name that automatically routes to a dynamic, changing *set* of pods. It uses the exact same label-selector mechanism as Chapter 5's ReplicaSets — a Service doesn't care about individual pod names or IPs, it continuously watches for pods matching its selector labels and load-balances traffic across whichever currently match and are healthy. This is another instance of the reconciliation loop pattern: the Service's list of backing pods (its **Endpoints**) is kept continuously up to date automatically.

Service Types — ClusterIP, NodePort & LoadBalancer

TYPE	REACHABLE FROM	TYPICAL USE
ClusterIP (default)	Inside the cluster only	Internal service-to-service traffic — e.g. web tier → backend tier
NodePort	Outside, via a static port on every node's own IP	A blunt mechanism, less commonly used directly in production
LoadBalancer	Outside, via a provisioned cloud load balancer	The common way to expose a service to the public internet on a managed provider (`cloud1-5`'s load balancer material)

In-Cluster DNS

Every Service automatically gets a DNS name within the cluster — typically `<service-name>.<namespace>.svc.cluster.local`, or just `<service-name>` from within the same namespace. This means application code can simply use the service name as a hostname, needing no IP address at all — cluster or pod. This is the actual mechanism that closes the loop on Chapter 3's warning: an application should talk to `my-backend-service`, never a pod IP directly.

Endpoints — How a Service Tracks Its Pods

An Endpoints object (or EndpointSlice in modern Kubernetes) is automatically maintained by Kubernetes, listing the current IPs of pods matching a Service's selector. When a pod matching the selector is created or destroyed — Chapter 5's rolling updates and self-healing — the Endpoints list updates automatically and near-instantly, and the Service starts or stops routing traffic to it accordingly. This is genuinely the "glue" making the whole abstraction work — a direct Kubernetes-side parallel to `cloud2-2`'s own load-balancer-health-check material from Cloud Platforms.

A Complete Service YAML, Explained

```yaml
apiVersion: v1
kind: Service
metadata:
  name: web-service
spec:
  selector:
    app: demo
  ports:
    - port: 80
      targetPort: 8080
  type: ClusterIP
```

Services use plain `v1`, not `apps/v1` — a nice, direct contrast to Chapter 5's own apiVersion warning, worth reinforcing rather than assuming it always follows the same pattern. `spec.selector` uses the **same labels** as Chapter 5's Deployment's pod template — that shared label is the connecting thread between the two resources. `port` is what the Service itself listens on; `targetPort` is what the container actually listens on — a genuinely easy point of confusion.

Testing Service Discovery

From any pod in the cluster, `curl http://web-service` just works — resolving via in-cluster DNS and load-balancing across whichever matching pods are currently healthy. A genuinely satisfying, concrete way to see the whole Chapter 3-through-6 arc pay off in one working example.

NEVER HARDCODE A POD IP — NOW FULLY EXPLAINED

Chapter 3 warned against this without yet explaining the fix. Now it's concrete: always connect via a Service's stable DNS name. The Service's Endpoints mechanism transparently handles every pod replacement behind the scenes — the application code never needs to know or care.

PORT VS. TARGETPORT — A GENUINELY COMMON EARLY MISTAKE

Getting these backwards — or forgetting that the container's actual listening port must match `targetPort` exactly — is a frequent, confusing source of "why can't I reach my service" issues. `port` is what clients connect to; `targetPort` must match what the container itself is genuinely listening on.

Hands-On Exercises

EXERCISE 1

Explain why an application should connect to a Service's DNS name rather than a specific pod's IP address, tying directly to Chapter 3's pod disposability material.

View solution

EXERCISE 2

Explain the difference between ClusterIP, NodePort, and LoadBalancer, and recommend which is appropriate for (a) a backend database only the app tier should reach, and (b) a public-facing web frontend on a managed cloud provider.

View solution

EXERCISE 3

A Service's spec defines `port: 80` and `targetPort: 8080`, but the container is actually listening on port 3000. Explain what would go wrong, and how to fix it.

View solution

CHAPTER 6 QUICK REFERENCE

- A **Service** gives a stable IP/DNS name routing to a dynamic, changing set of pods via the same label-selector mechanism as ReplicaSets
- **ClusterIP** (internal only) · **NodePort** (static port on every node) · **LoadBalancer** (provisions a real cloud LB, the common public-facing choice)
- In-cluster DNS lets code use a service name as a hostname — no IP knowledge needed at all
- **Endpoints** — automatically tracks which pod IPs currently back a Service, updated the instant pods come or go
- Services use `v1`, not `apps/v1`; `port` (what clients connect to) vs. `targetPort` (what the container listens on) is a common confusion
- Chapter 3's "never hardcode a pod IP" is now fully explained — always use the Service's DNS name instead
- **Next chapter:** ConfigMaps & Secrets — externalizing configuration, mounting as env vars/volumes

CHAPTER 7 OF 12

ConfigMaps & Secrets

Kubernetes Fundamentals

Chapter 7 · ConfigMaps & Secrets

Chapter 6 covered how pods communicate. This chapter covers how to give pods configuration and sensitive data without baking either into the container image itself — a genuinely important separation-of-concerns practice.

Why Externalize Configuration?

Baking configuration values directly into a container image means rebuilding the image every time a value changes, and makes the same image unusable across different environments (dev/staging/prod) without modification. The twelve-factor app principle states config should live in the environment, not in code or images — Kubernetes provides two first-class objects for exactly this: ConfigMaps and Secrets.

ConfigMaps — Non-Sensitive Configuration

A ConfigMap holds key-value configuration data — feature flags, URLs, non-sensitive settings. Created from literal values, files, or directories via `kubectl`, or declaratively via YAML. Genuinely important to state plainly: a ConfigMap is **not** encrypted or specially protected — a direct contrast with Secrets, covered next.

Secrets — Sensitive Configuration

Structurally very similar to ConfigMaps, but intended for sensitive data — passwords, API keys, tokens — directly echoing `crypto1-11`'s key management material and `pipelines1-5`'s "never commit credentials" rule, now applied at the Kubernetes object level. Secrets are **base64-encoded** by default, **not encrypted** — a critical, commonly misunderstood point worth its own dedicated section.

The Base64 Misconception — A Dedicated Warning

BASE64 ≠ ENCRYPTION

Base64-encoding a value and storing it in a Secret does **not** make it secure by itself. Anyone with read access to the Secret object — or to etcd directly, per Chapter 2's own "etcd holds the entire cluster state" point — can trivially

decode it back to plaintext. Base64 is an encoding, not encryption; it provides zero real confidentiality on its own.

Genuine protection requires additional measures: encryption at rest for etcd (`crypto1-5/6`, `dbsec1-5`'s encryption-at-rest material), RBAC restricting who can read Secret objects at all (Course 2's own `k8s2-4` covers this properly), and ideally an external secrets manager (`cloud1-11`'s own KMS/secrets-manager material) for genuinely sensitive production secrets, rather than relying on raw Kubernetes Secrets alone.

Consuming ConfigMaps & Secrets — Two Methods

METHOD	BEHAVIOR
Environment variables	Injected at container startup; a running pod does **not** pick up a later change, since env vars are only read once
Mounted volumes	Appears as files inside the container; updates automatically (with some delay) if the underlying ConfigMap/Secret changes, without restarting the pod

A GENUINELY PRACTICAL DISTINCTION WHEN DESIGNING CONFIG CONSUMPTION

If an application needs to pick up config changes without a restart, mounted volumes are the right choice. If simplicity matters more and a restart-on-change is acceptable (or even desired, tied to a deliberate redeploy), environment variables are fine.

A Concrete Example — ConfigMap as Environment Variables

```
# ConfigMap
apiVersion: v1
kind: ConfigMap
metadata:
  name: app-config
data:
  LOG_LEVEL: "debug"

# Referenced in a Deployment's pod spec (Ch.5's own structure)
containers:
  - name: web
    image: myapp:latest
    envFrom:
      - configMapRef:
          name: app-config
```

A Concrete Example — Secret as a Mounted Volume

```
volumes:
  - name: db-secret-volume
    secret:
      secretName: db-credentials
containers:
  - name: web
    volumeMounts:
      - name: db-secret-volume
        mountPath: /etc/secrets
```

The application reads the database password from a *file* at `/etc/secrets` rather than an environment variable — genuinely a more common production pattern for secrets specifically, since environment variables can leak more readily than file contents (process listings, crash dumps, and logging tools frequently capture env vars, less often a file's contents).

A Note on GitOps & Secrets

Since Secrets are only base64-encoded, committing raw Secret YAML files to a git repository is genuinely risky, for exactly the same reason `pipelines1-5` flags committing any credential to source control. Tools like Sealed Secrets or External Secrets Operator exist specifically to solve this — keeping genuinely encrypted secret material out of git while still supporting a GitOps workflow, a topic Course 2's `k8s2-9` builds on directly.

Hands-On Exercises

EXERCISE 1

Explain why base64 encoding a Secret's value does not provide real security on its own, and name at least two additional measures that would provide genuine protection.

View solution

EXERCISE 2

Explain the practical difference between consuming a ConfigMap as an environment variable versus as a mounted volume, specifically regarding what happens when the ConfigMap's value changes after the pod is already running.

View solution

EXERCISE 3

Explain why committing a raw Kubernetes Secret YAML file to a git repository is risky, and name a category of tool that exists specifically to solve this problem for GitOps workflows.

View solution

CHAPTER 7 QUICK REFERENCE

- **ConfigMaps** — non-sensitive config, not encrypted or protected
- **Secrets** — for sensitive data, but only base64-encoded, **not** encrypted; needs etcd encryption at rest + RBAC + ideally an external secrets manager for real protection
- **Env vars** — simple, but read once at startup; a running pod won't pick up a later change
- **Mounted volumes** — update automatically without a restart; also the safer default for secrets specifically (less prone to leaking than env vars)
- Never commit a raw Secret YAML to git — Sealed Secrets/External Secrets Operator exist to solve this for GitOps (`k8s2-9`)
- **Next chapter:** Storage in Kubernetes — Volumes, PersistentVolumes/PersistentVolumeClaims, StorageClasses

CHAPTER 8 OF 12

Storage in Kubernetes

Kubernetes Fundamentals

Chapter 8 · Storage in Kubernetes

Chapter 7 covered configuration and secrets. This chapter covers persistent *data* — what happens when a pod needs to store data that survives beyond its own disposable lifecycle (Chapter 3's pets-vs-cattle material).

The Problem — Container Filesystems Are Ephemeral Too

A container's own filesystem is ephemeral by default — when the container stops or is replaced, anything written locally is gone. This is exactly Chapter 3's pod-disposability problem again, but for *data* rather than the pod's own identity or IP — the same underlying theme, recurring at a different layer.

Volumes — Pod-Level Storage

A basic Volume attaches to a **pod**, not an individual container — it can be shared between multiple containers in the same pod, echoing Chapter 3's shared-storage point. An `emptyDir` volume's lifetime is tied to the pod's own lifetime — it survives container restarts *within* the same pod, but **not** pod deletion or replacement. Useful for temporary scratch space or inter-container file sharing — not for data that must outlive the pod itself.

> **EMPTYDIR SURVIVING A RESTART DOESN'T MEAN IT'S PERSISTENT**
>
> An emptyDir volume surviving a container restart within the same pod can create a false impression of durability. It does **not** survive the pod itself being deleted or replaced — a genuinely common point of confusion worth being deliberate about.

PersistentVolumes (PV) — Cluster-Level Storage Resources

A PersistentVolume represents an actual piece of storage in the cluster — backed by cloud block storage (`cloud1-4`'s own EBS/Managed Disks/Persistent Disk material, directly reused here), NFS, or another backend. A PV's lifetime is **independent** of any specific pod — it exists as a cluster resource on its own, genuinely solving the persistence problem.

PersistentVolumeClaims (PVC) — Requesting Storage

A PVC is how a pod actually requests storage — specifying how much space and what access mode it needs, without needing to know the underlying storage details. Kubernetes matches (or "binds") a PVC to a suitable available PV automatically.

> **A REPEATED KUBERNETES PATTERN WORTH RECOGNIZING**
>
> This mirrors Chapter 6's own Service abstraction exactly — a stable, decoupled interface (the PVC, like a Service) hiding a dynamic underlying implementation (the PV, like a pod). Recognizing this same "claim/interface vs. dynamic backing resource" pattern recurring across different Kubernetes concepts makes each new one faster to understand.

StorageClasses — Dynamic Provisioning

Rather than a cluster administrator manually pre-creating PVs ahead of time, a StorageClass defines *how* storage should be dynamically provisioned on demand when a PVC requests it — which cloud storage type or tier to use (directly tying to `cloud1-4`'s own storage-tier material). When a PVC references a StorageClass, Kubernetes automatically creates a matching PV behind the scenes — the common, practical pattern on managed cloud Kubernetes (EKS/AKS/GKE), rather than manual PV pre-provisioning.

Access Modes

MODE	MEANING
ReadWriteOnce (RWO)	Mounted read-write by a single node at a time — the common case for block storage (e.g. a database)
ReadOnlyMany (ROX)	Many nodes can mount read-only simultaneously
ReadWriteMany (RWX)	Many nodes can mount read-write simultaneously — needs a backend that actually supports it (e.g. NFS, not typical block storage)

Genuinely practical to know ahead of Course 2's `k8s2-1` StatefulSets chapter, where shared-storage scenarios come up directly.

Putting It Together — A Concrete PVC + Pod Example

```yaml
# PersistentVolumeClaim
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
  name: data-pvc
spec:
  accessModes:
    - ReadWriteOnce
  storageClassName: standard
  resources:
    requests:
      storage: 10Gi

# Referenced in a pod spec (Ch.5's structure, Ch.7's mounting pattern)
volumes:
  - name: app-data
    persistentVolumeClaim:
      claimName: data-pvc
containers:
  - name: app
    volumeMounts:
      - name: app-data
        mountPath: /data
```

This is genuinely the same mounting mechanism as Chapter 7's Secret volume — just a different volume source underneath.

When to Actually Use Persistent Storage

Matching this course's own honest "when you don't need X" convention from Chapter 1: stateless applications — most web and API servers — generally don't need persistent volumes at all. They should store state in an external database or object store (`cloud1-4`'s object storage material) rather than local pod storage. Persistent volumes are genuinely needed for stateful workloads specifically — databases, message queues with local state — directly foreshadowing Course 2's `k8s2-1` StatefulSets chapter.

Hands-On Exercises

EXERCISE 1

Explain the difference between an emptyDir Volume and a PersistentVolume, specifically regarding what pod-lifecycle events each one survives.

View solution

EXERCISE 2

Explain the relationship between a PersistentVolumeClaim and a PersistentVolume, and explain the structural parallel to Chapter 6's Service/pod relationship.

View solution

EXERCISE 3

A team is deploying a stateless REST API that doesn't store any of its own data locally — it reads and writes to an external managed database. Should they configure a PersistentVolume for their pods? Justify your answer using this chapter's "when to actually use persistent storage" material.

View solution

CHAPTER 8 QUICK REFERENCE

- Container filesystems are ephemeral — the same disposability problem as Chapter 3, now applied to data
- **emptyDir Volume** — tied to pod lifetime, survives container restarts, NOT pod replacement
- **PersistentVolume (PV)** — a real cluster storage resource, independent of any specific pod
- **PersistentVolumeClaim (PVC)** — a pod's request for storage, matched/bound to a PV — the same decoupled-claim pattern as Chapter 6's Services
- **StorageClass** — defines dynamic provisioning; the practical default on managed Kubernetes
- **RWO** (single node) vs. **ROX** (many, read-only) vs. **RWX** (many, read-write, needs a backend like NFS)
- Stateless apps generally don't need persistent storage at all — use an external database/object store instead
- **Next chapter:** Ingress & Exposing Services Externally — Ingress controllers, path/host-based routing

CHAPTER 9 OF 12

Ingress & Exposing Services Externally

Kubernetes Fundamentals

Chapter 9 · Ingress & Exposing Services Externally

Chapter 6 covered exposing a single Service externally. This chapter covers a genuinely common real need Chapter 6 didn't fully solve: exposing *multiple* services through a single entry point, with smarter routing than a plain LoadBalancer Service provides.

The Limitation of LoadBalancer Services, Revisited

A LoadBalancer Service provisions **one** external cloud load balancer **per** Service. Ten services each needing external exposure means potentially ten separate cloud load balancers — each with its own cost (`cloud1-9`'s own cost material) and its own external IP. A plain LoadBalancer Service also operates at Layer 4 (`cloud1-5`'s own L4/L7 distinction) — it has no understanding of HTTP paths or hostnames at all, just raw TCP/UDP forwarding. A genuinely common real need it can't solve: routing `app.example.com/api` to one service and `app.example.com/web` to another, through one shared entry point.

What Ingress Actually Is

An Ingress is a set of rules describing how external HTTP(S) traffic should be routed to internal Services, based on hostname and/or URL path.

AN INGRESS RESOURCE BY ITSELF DOES NOTHING

Critically, an Ingress requires an **Ingress controller** actually running in the cluster to read and implement those rules. Creating Ingress YAML with no controller installed accomplishes nothing at all — a genuinely common early confusion worth stating directly. Common controllers: the NGINX Ingress Controller, and cloud-provider-specific ones (AWS Load Balancer Controller, GCP's own, Azure's own). These often integrate with the provider's own LoadBalancer mechanism (Chapter 6) to get traffic into the cluster in the first place, then the controller's own Layer 7 logic takes over routing from there.

Host-Based & Path-Based Routing

- **Host-based** — `api.example.com` routes to `api-service`; `app.example.com` routes to a different `frontend-service` — both sharing the same external IP/load balancer.

- **Path-based** — `example.com/api/*` routes to `api-service`; `example.com/*` (everything else) routes to `frontend-service`.

Both can be combined. This directly closes the loop on the "10 services, 10 load balancers" cost problem — one Ingress, one external entry point, routing to many internal Services by rule.

A Concrete Ingress YAML, Explained

```yaml
apiVersion: networking.k8s.io/v1
kind: Ingress
metadata:
  name: app-ingress
spec:
  rules:
    - host: example.com
      http:
        paths:
          - path: /api
            pathType: Prefix
            backend:
              service:
                name: api-service
                port:
                  number: 80
          - path: /
            pathType: Prefix
            backend:
              service:
                name: frontend-service
                port:
                  number: 80
```

Yet another distinct API group — `networking.k8s.io/v1` — reinforcing Chapter 5's own point that apiVersion varies by kind. Every backend referenced is an ordinary Chapter 6 Service — Ingress routes *to* Services, it doesn't replace them.

TLS Termination at the Ingress

An Ingress can also handle TLS/HTTPS termination centrally — one place to manage certificates (`https1`'s TLS material, `crypto1`'s certificate chain material) for many backend services, rather than configuring TLS separately in every individual service. A genuine, practical reduction in operational complexity beyond just routing.

Ingress vs. Service LoadBalancer — When to Use Which

NEED	RIGHT CHOICE
A single service, or Layer 4/non-HTTP traffic (raw TCP, a database)	LoadBalancer Service directly
Multiple HTTP(S) services sharing one entry point, host/path routing, centralized TLS	Ingress

Genuinely common in practice: many real clusters use **both** — an Ingress for the bulk of HTTP services, and a small number of standalone LoadBalancer Services for anything that doesn't fit the HTTP routing model.

10 SERVICES = 10 LOAD BALANCERS = A REAL, MULTIPLIED COST

Directly echoing `cloud1-9`'s cost material: exposing every HTTP service individually via its own LoadBalancer Service multiplies real infrastructure cost unnecessarily when a single Ingress could route all of them through one shared entry point instead.

Hands-On Exercises

EXERCISE 1

Explain why creating an Ingress YAML resource alone, with no Ingress controller installed in the cluster, accomplishes nothing. What's actually missing?

View solution

EXERCISE 2

Five different HTTP microservices all need to be reachable from the internet under the same domain, on different paths. Explain why using five separate LoadBalancer Services would be a worse choice than a single Ingress, referencing both cost and Layer 4 vs. Layer 7 capability.

View solution

EXERCISE 3

Explain the benefit of TLS termination at the Ingress level rather than configuring TLS separately in each backend service.

View solution

CHAPTER 9 QUICK REFERENCE

- LoadBalancer Services are one-per-service and Layer 4 only — no HTTP host/path awareness
- An **Ingress** defines HTTP(S) routing rules — but does nothing without an **Ingress controller** actually running
- Host-based and path-based routing, combinable, all through one shared external entry point
- Ingress uses yet another apiVersion (`networking.k8s.io/v1`); backends are ordinary Services (Ch.6)
- Centralized TLS termination at the Ingress reduces real operational complexity
- Single/non-HTTP service → LoadBalancer directly; many HTTP services sharing an entry point → Ingress; real clusters often use both
- **Next chapter:** Namespaces & Resource Management — organizing a cluster, resource requests/limits, basic quotas

CHAPTER 10 OF 12

Namespaces & Resource Management

Kubernetes Fundamentals

Chapter 10 · Namespaces & Resource Management

Chapter 9 covered getting traffic into the cluster. This chapter turns inward: how a cluster's own resources — CPU, memory, and organizational structure — are shared fairly across many workloads sitting on the same physical hardware.

Namespaces, Properly Explained

Revisiting Chapter 4's brief mention in full: a namespace is a logical partition *within* one cluster — not a separate cluster. Used to separate teams, environments, or projects sharing infrastructure. Dev/staging/prod as three namespaces within one cluster is common, though many organizations use fully separate clusters per environment instead — both patterns genuinely exist in practice, echoing `cloud1-1`'s own honest hybrid/multi-environment framing. Resource names must be unique *within* a namespace, not cluster-wide. RBAC (Course 2's `k8s2-4` covers this fully) is commonly scoped per namespace, letting different teams have appropriately restricted access to only their own namespace's resources — a genuinely practical reason namespaces matter beyond simple tidiness.

Cluster-Scoped vs. Namespace-Scoped Resources

Most resources covered so far — Pods, Deployments, Services, ConfigMaps, Secrets, PVCs — are namespace-scoped. Some are cluster-scoped instead: Nodes, PersistentVolumes themselves (distinct from PVCs), and Namespaces themselves. Genuinely useful to know which is which: `kubectl get pods` needs a namespace context; `kubectl get nodes` doesn't, and never will, regardless of any namespace flag.

Why Resource Requests & Limits Matter

Multiple pods from different teams and applications share the same physical node's finite CPU and memory. Without any constraints, one poorly-behaved or unexpectedly busy pod could consume all available resources on a node, starving every other pod scheduled there — a genuinely real, common operational problem.

Requests — What the Scheduler Uses

A **request** is the amount of CPU/memory a container is guaranteed to get, and specifically what Chapter 2's scheduler uses when deciding which node has enough available capacity to place a pod on. Setting requests accurately matters: too low and the scheduler might overpack a node; too high and capacity — and cost — is wasted unnecessarily, echoing `cloud1-9`'s own oversized-compute cost material, just at the container level rather than the whole-VM level.

Limits — The Hard Ceiling

A **limit** is the maximum a container is allowed to use, enforced by the container runtime. Exceeding a **memory** limit results in the container being OOM-killed — the same OOM-kill pattern briefly named in Cloud Platforms' own `cloud2-5`, explained properly here at the Kubernetes level. Exceeding a **CPU** limit doesn't kill the container — it's throttled instead.

> **WHY CPU THROTTLES BUT MEMORY KILLS**
>
> CPU is a *compressible* resource — a container can simply be given less CPU time and keep running, just slower. Memory is *not* compressible — there's no graceful way to "un-allocate" memory a process is already using, so exceeding a memory limit forces the runtime to kill the container outright. Same category of constraint, genuinely different enforcement, worth remembering clearly since it explains very different observed behavior for what can otherwise look like "the same kind of problem."

A Concrete Example — Requests & Limits in a Pod Spec

```
containers:
  - name: web
    image: myapp:latest
    resources:
      requests:
        cpu: "250m"
        memory: "128Mi"
      limits:
        cpu: "500m"
        memory: "256Mi"
```

CPU is measured in millicores — `"500m"` means half a CPU core. Memory uses `Mi`/`Gi` (mebibytes/gibibytes). The notation itself is a genuinely common source of confusion for newcomers, worth confirming explicitly rather than assuming.

Quality of Service (QoS) Classes, Briefly

CLASS	HOW IT'S SET	EVICTION PRIORITY
Guaranteed	Requests = limits, for every resource	Highest priority, least likely evicted
Burstable	Requests set, limits higher or unset	The common middle case
BestEffort	No requests/limits set at all	Lowest priority, first evicted under pressure

Worth knowing this classification exists — it explains why some pods get evicted before others during real resource pressure, genuinely relevant troubleshooting knowledge for Course 2's `k8s2-8`.

> **NEVER DEPLOY TO PRODUCTION WITH NO REQUESTS/LIMITS AT ALL**
>
> A pod with no requests or limits set lands in the BestEffort QoS class — no scheduling guarantees, and the first to be evicted under any resource pressure at all. A genuinely real operational risk worth avoiding deliberately, not accidentally.

ResourceQuotas — Limiting a Namespace as a Whole

A ResourceQuota object caps the *total* resource consumption (or object count) allowed within an entire namespace, regardless of individual pod-level requests/limits — a genuinely practical administrative tool for a multi-team cluster, preventing one team's namespace from consuming resources needed by others. Together with per-namespace RBAC, namespaces + quotas + RBAC are how a shared cluster stays fair and secure across multiple teams.

Hands-On Exercises

EXERCISE 1

Explain the difference between a resource request and a resource limit, specifically regarding which one the scheduler uses and which one the runtime enforces during actual execution.

View solution

EXERCISE 2

Explain why exceeding a memory limit results in a container being killed, while exceeding a CPU limit results in throttling instead. What property of each resource type explains this difference?

View solution

EXERCISE 3

Three pods with different QoS classes (Guaranteed, Burstable, BestEffort) are all scheduled on a node that's running low on resources. Explain which one is likely to be evicted first, and why.

View solution

CHAPTER 10 QUICK REFERENCE

- **Namespaces** — partition one cluster; per-namespace RBAC is a genuinely practical reason they matter
- Most resources are namespace-scoped; Nodes/PVs/Namespaces themselves are cluster-scoped
- **Requests** — what the scheduler uses to place pods; **limits** — the hard ceiling the runtime enforces
- CPU limit exceeded → **throttled** (compressible); memory limit exceeded → **OOM-killed** (not compressible)
- CPU in millicores (`500m` = half a core); memory in `Mi` / `Gi`
- **QoS classes** — Guaranteed (safest) → Burstable (common) → BestEffort (no requests/limits, evicted first) — never deploy production BestEffort
- **ResourceQuota** — caps total consumption per namespace, independent of individual pod settings
- **Next chapter:** Health Checks & Self-Healing — liveness/readiness/startup probes, restart policies

CHAPTER 11 OF 12

Health Checks & Self-Healing

Kubernetes Fundamentals

Chapter 11 · Health Checks & Self-Healing

Chapter 5's self-healing and Chapter 2's reconciliation loop both assume Kubernetes can actually detect a problem. This chapter covers exactly how.

The Problem — "Running" Doesn't Mean "Healthy"

Revisiting Chapter 3's pod phase material: a container can sit in the "Running" phase while being completely unresponsive, deadlocked, or stuck in an infinite loop. The process technically hasn't crashed, so Kubernetes' default behavior — restarting only on an actual process exit — wouldn't catch this at all. "The process is running" and "the application is working correctly" are not the same claim.

Probes — How Kubernetes Actually Checks

- **Liveness** — "is this container still alive and functioning, or should it be restarted?"
- **Readiness** — "is this container ready to receive traffic right now?" (directly tied to Chapter 6's Service/Endpoints material.)
- **Startup** — "has this container finished its possibly slow startup process yet?" — used to prevent liveness probes from prematurely killing a container still legitimately starting up.

Liveness Probes — Restart on Failure

A failed liveness probe causes Kubernetes to **kill and restart the container**. This is what actually triggers self-healing at the *container* level — genuinely distinct from Chapter 5's pod-replacement-level self-healing via the ReplicaSet. A liveness probe failure restarts a container within the same pod; a pod being deleted entirely is a separate, higher-level event handled by the ReplicaSet.

Readiness Probes — Removing From Service Without Killing

Sometimes a container is perfectly alive and functioning, but temporarily not ready to serve traffic — still loading a large dataset at startup, or briefly overloaded and needing a moment to catch up. Killing and

restarting it in this case would be actively counterproductive. A readiness probe instead just temporarily removes the pod from the Service's Endpoints (Chapter 6) until it reports ready again — no restart involved at all.

> **A GENUINELY COMMON MISTAKE: USING ONLY A LIVENESS PROBE**
>
> Using a liveness probe alone for a signal that's really a readiness concern causes unnecessary restarts for a problem that never actually needed one — a temporarily-busy-but-healthy container simply needed to be pulled out of rotation for a moment, not killed.

Startup Probes — Handling Slow-Starting Applications

Some applications — heavy initialization work like loading a large ML model or warming a cache — take much longer to become ready than a typical liveness probe's failure threshold tolerates, causing the liveness probe to repeatedly kill the container before it ever finishes starting. A genuinely frustrating, common real problem. A startup probe defers liveness (and readiness) probing entirely until it itself succeeds once, giving slow-starting applications the room they genuinely need.

Probe Mechanics — HTTP, TCP & Exec

- **HTTP GET** — the most common; checks for a successful response code from a specified path.
- **TCP socket** — simply checks whether a port accepts a connection; useful for non-HTTP services.
- **Exec** — runs a command inside the container and checks its exit code; most flexible, also the most expensive and slowest of the three.

Configurable timing parameters — `initialDelaySeconds`, `periodSeconds`, `failureThreshold` — are genuinely practical knobs worth knowing exist, even without exhaustive detail on each here.

A Concrete Example — Liveness & Readiness Together

```
  containers:
    - name: web
      image: myapp:latest
      livenessProbe:
        httpGet:
          path: /healthz
          port: 8080
        initialDelaySeconds: 10
        periodSeconds: 15
      readinessProbe:
        httpGet:
          path: /ready
          port: 8080
        periodSeconds: 5
```

A genuinely common real-world pattern: an application exposes two separate health endpoints with distinct meanings, matching the liveness-vs-readiness distinction explained above.

Restart Policies

`spec.restartPolicy` controls what happens when a container exits: **Always** (default, appropriate for long-running services like Deployments), **OnFailure** (only restarts on a non-zero exit — relevant for Course 2's own Jobs material), **Never**. This is a *pod-level* setting, genuinely distinct from the per-container liveness-probe-triggered restarts covered above — two related but different restart mechanisms worth not conflating.

How This All Ties Together — The Full Self-Healing Picture

The reconciliation loop (Chapter 2) is the general mechanism. ReplicaSets (Chapter 5) use it to maintain pod *count*. Liveness probes (this chapter) tell Kubernetes when a specific *container* needs restarting. Readiness probes (this chapter) tell Services (Chapter 6) which pods should currently receive traffic. Together, this is what makes Kubernetes self-healing in the full, practical sense Chapter 1 promised — a satisfying convergence of nearly everything this course has covered so far, ahead of Chapter 12's own capstone.

> **READINESS VS. LIVENESS — THE MOST COMMONLY CONFUSED PAIR IN THIS CHAPTER**
>
> Liveness asks "should this be restarted?" Readiness asks "should this currently receive traffic?" Being deliberate about which question a given health signal actually answers avoids both unnecessary restarts and traffic sent to a genuinely broken container.

Hands-On Exercises

EXERCISE 1

Explain the difference between a liveness probe failing and a readiness probe failing — specifically, what action Kubernetes takes in each case.

View solution

EXERCISE 2

Explain why a startup probe is needed for some applications even though liveness and readiness probes already exist. What specific problem does it solve that the other two don't?

View solution

EXERCISE 3

An application is occasionally slow to respond to database queries under heavy load, but is not actually broken or deadlocked. A liveness probe with a very short timeout keeps killing and restarting the container during these slow periods, making the problem worse. Explain what's misconfigured here and how it should be fixed.

View solution

CHAPTER 11 QUICK REFERENCE

- "Running" ≠ "healthy" — a deadlocked container can stay in the Running phase indefinitely without probes
- **Liveness** (restart the container) vs. **readiness** (remove from Service Endpoints, no restart) vs. **startup** (defers the other two until initial startup finishes)
- Probe mechanics: HTTP GET (most common), TCP socket, Exec (most flexible, slowest)
- `restartPolicy` (Always/OnFailure/Never) is pod-level, distinct from container-level liveness-triggered restarts
- Full self-healing picture: reconciliation loop (Ch.2) → ReplicaSets maintain count (Ch.5) → liveness restarts a container → readiness controls Service traffic (Ch.6)
- A liveness probe that's too aggressive can kill a genuinely healthy, just-temporarily-slow application — tune thresholds deliberately
- **Next chapter:** Capstone — A Small Multi-Tier App, combining everything from this course

CHAPTER 12 OF 12

Capstone: A Small Multi-Tier App

Kubernetes Fundamentals

Chapter 12 · Capstone — A Small Multi-Tier App

Every prior chapter contributes one piece to this capstone: a real, deployable frontend + backend + database application, built entirely from concepts already covered.

The Application

Three tiers: a **frontend** (a web UI), a **backend** (a small API service), and a **database** — the frontend talks to the backend, the backend talks to the database, and only the frontend is exposed to the outside world.

Step 1 — Namespace

```
kubectl create namespace shop-app
```

Chapter 10's own material: a dedicated namespace for this app, rather than everything landing in `default`.

Step 2 — ConfigMap & Secret for the Backend

Chapter 7's material, applied directly: `LOG_LEVEL` as a ConfigMap value, the database password as a Secret.

```
apiVersion: v1
kind: Secret
metadata:
  name: db-credentials
  namespace: shop-app
stringData:
  password: changeme
```

Step 3 — Database

A Deployment (Chapter 5) with a single replica, backed by a PVC (Chapter 8) for actual data persistence, with the password mounted as a Secret volume (Chapter 7), reachable internally via a ClusterIP Service (Chapter 6):

```yaml
apiVersion: v1
kind: Service
metadata:
  name: db-service
  namespace: shop-app
spec:
  selector:
    app: db
  ports:
    - port: 5432
      targetPort: 5432
  type: ClusterIP
```

A DELIBERATE SIMPLIFICATION, REVISITED BELOW

A real production database would more properly use a **StatefulSet**, covered in Course 2's `k8s2-1`. This capstone deliberately stays within Course 1's own toolset — the multi-tier *architecture* is the point here, not the deepest-correct storage pattern.

Step 4 — Backend

A multi-replica Deployment, consuming the ConfigMap and Secret (Chapter 7), with liveness/readiness probes (Chapter 11) and resource requests/limits (Chapter 10), exposed internally via another ClusterIP Service:

```yaml
spec:
  replicas: 3
  template:
    spec:
      containers:
        - name: api
          image: shop-backend:latest
          envFrom:
            - configMapRef: {name: app-config}
          resources:
            requests: {cpu: "250m", memory: "128Mi"}
            limits: {cpu: "500m", memory: "256Mi"}
          livenessProbe:
            httpGet: {path: /healthz, port: 8080}
          readinessProbe:
            httpGet: {path: /ready, port: 8080}
```

Step 5 — Frontend & Ingress

A frontend Deployment and Service, with an Ingress (Chapter 9) routing `/api` to the backend and everything else to the frontend — the same host/path routing pattern from Chapter 9's own example:

```
spec:
  rules:
    - host: shop.example.com
      http:
        paths:
          - path: /api
            pathType: Prefix
            backend: {service: {name: backend-service, port: {number: 80}}}
          - path: /
            pathType: Prefix
            backend: {service: {name: frontend-service, port: {number: 80}}}
```

Putting It All Together — Where Every Piece Came From

PIECE	CHAPTER
Namespace	Ch.10 — organizing a cluster
ConfigMap / Secret	Ch.7 — externalized configuration
Deployments (all 3 tiers)	Ch.5 — declarative desired state, self-healing
PVC for the database	Ch.8 — persistent storage
ClusterIP Services (db, backend)	Ch.6 — stable internal addressing
Ingress (frontend + /api routing)	Ch.9 — external exposure, host/path routing
Resource requests/limits	Ch.10 — fair scheduling and QoS
Liveness/readiness probes	Ch.11 — real self-healing and traffic control

Verifying It Works

```
kubectl get pods -n shop-app
kubectl get svc -n shop-app
kubectl get ingress -n shop-app
curl http://shop.example.com/        # frontend
curl http://shop.example.com/api     # backend, via Ingress path routing
```

Chapter 4's own practical workflow, applied to a genuinely complete application rather than a single pod.

What's Deliberately Out of Scope — Where Course 2 Picks Up

- A proper **StatefulSet** for the database, instead of this capstone's simplified Deployment+PVC (`k8s2-1`).
- **RBAC** restricting who can access this namespace's resources (`k8s2-4`).
- **Autoscaling** the backend based on real load, rather than a fixed replica count (`k8s2-6`).
- Real **observability** — Prometheus/Grafana integration (`k8s2-7`).
- **Troubleshooting** this exact app when something actually goes wrong (`k8s2-8`).
- Packaging all of this as a **Helm chart** instead of raw YAML (`k8s2-3`).
- Deploying it via **GitOps** rather than manual `kubectl apply` (`k8s2-9`).

COURSE 1'S OWN THROUGHLINE, ONE MORE TIME

Nearly every piece of this capstone is, underneath, the same reconciliation loop (Chapter 2) applied to a different resource type — Deployments maintaining pod count, Services maintaining Endpoint lists, liveness probes maintaining container health. Recognizing that one repeated pattern is worth more than memorizing any single YAML snippet.

Hands-On Exercises

EXERCISE 1

Identify which chapter of this course each of the following pieces of the capstone app came from: (a) the database's Secret-mounted password, (b) the backend's liveness/readiness probes, (c) the Ingress routing rules.

View solution

EXERCISE 2

Explain why this capstone uses a Deployment+PVC for the database rather than a StatefulSet, and what specific problem a StatefulSet would solve that this simplified setup doesn't fully address.

View solution

EXERCISE 3

Across all of Course 1, name at least three instances of the same underlying reconciliation-loop pattern (Chapter 2) showing up in different guises, and explain what they have in common.

📄 View solution

CHAPTER 12 QUICK REFERENCE — COURSE 1 COMPLETE

- A full 3-tier app assembled entirely from Chapters 1-11's own concepts — namespace, ConfigMap/Secret, Deployments, PVC, Services, Ingress, resource limits, probes
- The database's Deployment+PVC is a deliberate simplification — Course 2's StatefulSets chapter covers the proper pattern
- Every piece of this capstone is, underneath, the same reconciliation loop applied to a different resource type
- **Course 1 complete** — Kubernetes Fundamentals, 12 chapters, from "what is Kubernetes" to a working multi-tier deployment
- **Course 2 next:** Kubernetes Intermediate/Advanced — StatefulSets, Helm, RBAC, autoscaling, observability, troubleshooting, GitOps