Kubernetes Intermediate/Advanced

A Complete 10-Chapter Production Kubernetes Course

Topics covered:

StatefulSets & DaemonSets/Jobs/CronJobs · Helm · RBAC & Security

Networking Deep Dive & Autoscaling · Observability & Troubleshooting

GitOps & CI/CD · Capstone: a production-ready deployment

Exercises: 30 hands-on exercises with worked solutions

Format: A4 · Dark-theme code examples · builds directly on Kubernetes Fundamentals

Table of Contents

CHAPTER 1 OF 10

StatefulSets

Kubernetes Intermediate/Advanced

Chapter 1 · StatefulSets

Course 1 closed with a deliberate simplification — a Deployment and a single PVC standing in for a proper database. This chapter is the payoff: what a StatefulSet actually adds, and why.

Restating the Problem — Why the Capstone's Database Was a Simplification

A plain Deployment treats all its replicas as interchangeable "cattle" — `k8s1-3`'s own framing. Fine for stateless apps. Genuinely wrong for a database that might need multiple replicas, each with its own distinct identity and its own distinct, consistently-reattached storage.

What a StatefulSet Actually Guarantees

- **Stable, predictable network identity** — each pod gets a predictable, persistent name like `db-0`, `db-1`, `db-2` (not a random suffix, unlike a Deployment's pods), staying the *same* even if that specific pod is recreated or rescheduled.
- **Stable storage** — each replica gets its own PersistentVolumeClaim (`k8s1-8`'s material), and critically, when a pod is recreated, it's automatically **reattached to the same PVC it had before**, not a fresh or randomly assigned one.

This is the core value proposition, stated plainly: identity and storage that survive individual pod replacement, per-replica.

Ordered, Predictable Pod Creation & Deletion

StatefulSet pods are created and deleted in **order** — `db-0` before `db-1` before `db-2`, and deleted in reverse — rather than in parallel or arbitrary order, unlike a Deployment. This matters for stateful systems where, for instance, a "primary" instance needs to exist and be ready before "replica" instances that depend on it start up — a genuine, real requirement for many clustered systems.

Headless Services — Enabling Direct Pod Addressing

A StatefulSet is typically paired with a **headless** Service (`clusterIP: None`). Rather than load-balancing across all matching pods the way a normal Service does (`k8s1-6`'s material), a headless Service instead provides a DNS record for *each individual pod* directly — e.g. `db-0.db-service.namespace.svc.cluster.local` — letting other parts of the system address a *specific* replica by name, not just "any healthy one." Genuinely necessary when replicas aren't interchangeable — writes must go to a specific primary, for instance.

A Concrete StatefulSet YAML, Explained

```yaml
apiVersion: apps/v1
kind: StatefulSet
metadata:
  name: db
spec:
  serviceName: db-service
  replicas: 3
  selector:
    matchLabels:
      app: db
  template:
    metadata:
      labels:
        app: db
    spec:
      containers:
        - name: db
          image: postgres:16
  volumeClaimTemplates:
    - metadata:
        name: data
      spec:
        accessModes: ["ReadWriteOnce"]
        resources:
          requests:
            storage: 10Gi
```

Same `apps/v1` group as Deployment — a genuinely useful reminder that StatefulSet is a sibling controller, not an unrelated concept. `spec.template` and `spec.selector` are the same shape as a Deployment's. The genuinely new piece is `volumeClaimTemplates` — this automatically creates a *unique* PVC per replica (e.g. `data-db-0`, `data-db-1`, `data-db-2`), rather than the single shared PVC the capstone used.

> **A SIBLING OF DEPLOYMENT, NOT A SEPARATE CONCEPT**
>
> Everything about template/selector/labels from `k8s1-5` transfers directly. StatefulSet only adds three things: predictable naming, per-replica storage via `volumeClaimTemplates`, and ordered creation/deletion.

Revisiting the Capstone's Database, Properly

With the StatefulSet above, `db-0` keeps its own name and its own dedicated PVC (`data-db-0`) across any restart or rescheduling. Scaled to 3 replicas, `db-1` and `db-2` each get their own independent PVCs too, rather than sharing or conflicting over the capstone's single shared volume — exactly the gap Course 1's warn-box flagged.

When You Actually Need a StatefulSet

Honest framing, matching this course's own recurring convention: if only *one* replica needs persistent storage, `k8s1-8`'s simple Deployment+single-PVC pattern (exactly what the capstone used) is genuinely fine and simpler. StatefulSets earn their added complexity specifically when *multiple* replicas each need their own distinct identity and storage — a multi-node database cluster where each node needs to know its own identity, a distributed system with leader election, or a message queue cluster with partition ownership tied to specific nodes.

SCALING DOWN DOES NOT DELETE THE REMOVED REPLICAS' PVCS

A deliberate data-safety measure: scaling a StatefulSet from 3 replicas down to 1 leaves `data-db-1` and `data-db-2` intact, untouched, and still billing — echoing `cloud1-9`'s own orphaned-storage cost material. Storage can silently accumulate if these aren't cleaned up manually once genuinely no longer needed.

Hands-On Exercises

EXERCISE 1

Explain the two specific guarantees a StatefulSet provides that a Deployment does not, and explain why a multi-replica database genuinely needs both.

View solution

EXERCISE 2

Explain what a headless Service is and why a StatefulSet typically needs one, specifically contrasting it with a normal Service's load-balancing behavior.

View solution

EXERCISE 3

A team scales their StatefulSet down from 3 replicas to 1, then later scales back up to 3. Explain what happens to the PVCs of the two removed replicas during the scale-down, and what this means for the team's ongoing storage costs if they don't clean up manually.

📄 View solution

CHAPTER 1 QUICK REFERENCE

- **Stable identity** (predictable names like `db-0`) + **stable storage** (reattached to the same PVC) — the two things a Deployment doesn't provide
- Pods created/deleted in strict order — `db-0` first, deleted last
- **Headless Service** — DNS per individual pod, not load-balanced across all of them
- `volumeClaimTemplates` — the StatefulSet-specific field auto-creating a unique PVC per replica
- Same `apps/v1` group and template/selector shape as Deployment — a sibling controller, not a new concept
- Only worth the complexity when multiple replicas each need distinct identity/storage — a single-replica case doesn't need one
- Scaling down leaves removed replicas' PVCs intact and billing — a real, easy-to-miss cost trap
- **Next chapter:** DaemonSets & Jobs/CronJobs — node-level agents, batch and scheduled workloads

CHAPTER 2 OF 10

DaemonSets & Jobs/CronJobs

Kubernetes Intermediate/Advanced

Chapter 2 · DaemonSets & Jobs/CronJobs

Chapter 1 covered StatefulSets for identity/storage-sensitive workloads. This chapter covers two more specialized controllers, each genuinely different from anything covered so far: DaemonSets (one pod per node) and Jobs/CronJobs (run to completion, not forever).

DaemonSets — One Pod Per Node, Guaranteed

A DaemonSet ensures exactly **one** copy of a pod runs on **every** node in the cluster (or a selected subset via node selectors) — genuinely different from a Deployment's "N replicas, scheduler decides where" model. A DaemonSet's desired count is inherently tied to the *number of nodes*, not an arbitrary number you specify. When a new node joins the cluster, the DaemonSet automatically schedules a pod there too — another instance of Chapter 2 (Course 1)'s reconciliation loop, just with desired state defined as "one per node" rather than a fixed number.

What DaemonSets Are Actually Used For

- **Log collection agents** (e.g. Fluentd/Fluent Bit) — needs to run on every node to collect logs from that node's own containers (`k8s2-7` 's upcoming observability chapter builds on this).
- **Monitoring/metrics agents** (e.g. Prometheus's node-exporter) — needs per-node system-level metrics, also feeding into `k8s2-7` .
- **Networking/CNI plugin pods** — some cluster networking implementations run as DaemonSets themselves, tying back to `k8s1-2` 's own architecture material.

The common thread: infrastructure-level concerns that are inherently per-node, not per-application.

A Concrete DaemonSet YAML, Explained

```yaml
apiVersion: apps/v1
kind: DaemonSet
metadata:
  name: log-agent
spec:
  selector:
    matchLabels:
      app: log-agent
  template:
    metadata:
      labels:
        app: log-agent
    spec:
      containers:
        - name: fluent-bit
          image: fluent/fluent-bit:latest
```

Same `apps/v1` group as Deployment/StatefulSet — Chapter 1's own "sibling controller" point holds again. Critically: **no `replicas` field at all**, since the count is implicitly "one per matching node," not something specified directly.

Jobs — Run to Completion, Not Forever

A completely different category: Deployments, StatefulSets, and DaemonSets are all about keeping something running indefinitely. A **Job** is for a task that should run once (or a specified number of times) and then *stop* — directly tying back to `k8s1-3`'s own Pod lifecycle material (Succeeded/Failed terminal states) and `k8s1-11`'s `OnFailure` restart policy, which exists specifically for this use case. Genuinely common real uses: a database migration script, a one-off batch data-processing task, sending a batch of emails.

Job Completion & Parallelism

`spec.completions` — how many successful pod completions are needed for the Job to be considered done. `spec.parallelism` — how many pods can run at once working toward that count. A Job processing 100 queue items could run `completions: 100`, `parallelism: 10`, processing 10 items concurrently until all 100 finish — a genuinely different kind of "scaling" concept from Course 2's own upcoming `k8s2-6` autoscaling material, worth not conflating.

CronJobs — Scheduled Jobs

A CronJob is, in a sense, to a Job what a Deployment is to a Pod — it creates Jobs on a recurring schedule, using standard cron syntax. Genuinely common real uses: nightly database backups, periodic cleanup tasks, scheduled report generation. Each scheduled run creates a **new Job object**, which then creates pods per that

Job's own completion/parallelism settings — worth being explicit about the layering: CronJob creates Jobs, Jobs create Pods, three distinct layers.

A Concrete CronJob YAML, Explained

```yaml
apiVersion: batch/v1
kind: CronJob
metadata:
  name: nightly-backup
spec:
  schedule: "0 2 * * *"   # 2am daily, standard cron syntax
  jobTemplate:
    spec:
      template:
        spec:
          containers:
            - name: backup
              image: backup-tool:latest
          restartPolicy: OnFailure
```

Yet another distinct API group — `batch/v1` — reinforcing the recurring "apiVersion differs by kind" lesson from Chapters 5, 6, and 9. Note the genuinely deep nesting: `spec.jobTemplate.spec.template.spec` — a CronJob spec containing a Job spec containing a pod spec, three layers deep, worth reading carefully rather than being intimidated by.

> **CRONJOB → JOB → POD, THREE LAYERS**
>
> Keeping this chain straight matters directly for troubleshooting later (`k8s2-8`): a failed backup could mean the CronJob never triggered, the Job it created failed, or the Pod that Job spawned crashed — three genuinely different places to look.

A Genuinely Common Gotcha — Missed & Overlapping Schedules

What happens if a CronJob's previous run is still executing when the next scheduled time arrives? `spec.concurrencyPolicy` controls this: **Allow** (default, runs can overlap), **Forbid** (skip the new run if the previous is still going), **Replace** (cancel the old run, start the new one). Genuinely important to configure deliberately for something like a database backup — overlapping backup runs could cause real problems.

> **LEAVING CONCURRENCYPOLICY AT ITS DEFAULT FOR A BACKUP JOB IS A REAL RISK**
>
> The default, `Allow`, permits overlapping runs — for a job that genuinely shouldn't run twice concurrently (a backup writing to the same target, for instance), this can cause real data corruption or resource contention. Set `Forbid` explicitly for anything where overlap would be a genuine problem, rather than leaving the default unexamined.

Hands-On Exercises

EXERCISE 1

Explain why a DaemonSet's "desired pod count" isn't something you directly specify the way a Deployment's replica count is, and give one concrete real-world example of a workload that genuinely needs to run on every node.

View solution

EXERCISE 2

Explain the three-layer relationship between a CronJob, a Job, and a Pod — which one creates which.

View solution

EXERCISE 3

A team runs a nightly database backup as a CronJob with the default `concurrencyPolicy`. One night the backup takes much longer than usual and is still running when the next scheduled run begins. Explain what happens by default, why this could be a real problem for a backup job specifically, and what `concurrencyPolicy` setting would prevent it.

View solution

CHAPTER 2 QUICK REFERENCE

- **DaemonSet** — one pod per node, automatically, no `replicas` field; used for log/metrics agents and CNI plugins
- **Job** — runs to completion, not forever; `completions` / `parallelism` control how much work and how much concurrency
- **CronJob** — creates Jobs on a schedule; three layers: CronJob → Job → Pod
- CronJob uses yet another apiVersion (`batch/v1`) — the recurring apiVersion-differs-by-kind pattern continues
- `concurrencyPolicy` (Allow/Forbid/Replace) — leaving it at the default Allow for something like a backup risks real overlap problems
- **Next chapter:** Helm — Package Management for Kubernetes — charts, templating, releases

CHAPTER 3 OF 10

Helm

Kubernetes Intermediate/Advanced

Chapter 3 · Helm — Package Management for Kubernetes

Every chapter so far has piled up more raw YAML — StatefulSets, DaemonSets, Jobs, RBAC objects. This chapter covers the tool that manages all of that at real scale: Helm.

The Problem With Raw YAML at Scale

`k8s1-12`'s capstone alone needed roughly eight separate YAML resources — a namespace, ConfigMap, Secret, three Deployments, three Services, an Ingress, a PVC. Manually managing, versioning, and deploying dozens of interrelated YAML files for a real application becomes genuinely unwieldy. Worse: what if the same application needs deploying to dev, staging, and prod with slightly different values — replica counts, resource limits, image tags? Raw YAML has no built-in templating — that means separate, near-duplicate files per environment, or external tooling.

What Helm Actually Is

A package manager for Kubernetes — conceptually similar to apt/npm/pip, but for Kubernetes applications. A **chart** is a Helm package: a collection of templated YAML manifests plus metadata. Installing a chart with specific configuration values produces a **release** — a named, tracked deployment of that chart into a cluster.

Chart Structure, Explained

```
mychart/
  Chart.yaml          # metadata — name, version, description
  values.yaml         # default configuration values
  templates/           # the actual YAML manifests, with templating syntax
    deployment.yaml
    service.yaml
    ingress.yaml
```

RENDERED HELM OUTPUT IS JUST ORDINARY KUBERNETES YAML

> Genuinely reassuring: the templating syntax in `templates/` is filled in with values from `values.yaml`, and the RESULT is identical in shape to everything Course 1 already taught. Helm doesn't replace that knowledge — it generates it programmatically.

Templating — Turning the Capstone Into a Chart

A fragment of `k8s1-12`'s own Deployment YAML, templated:

```
# templates/deployment.yaml
spec:
  replicas: {{ .Values.replicaCount }}
  template:
    spec:
      containers:
        - name: backend
          image: {{ .Values.image.repository }}:{{ .Values.image.tag }}
```

```
# values.yaml (defaults)
replicaCount: 3
image:
  repository: shop-backend
  tag: "1.0"
```

A separate `values-prod.yaml` overriding just `replicaCount: 10` and a different `tag` lets the exact same chart deploy meaningfully different dev and prod configurations — directly solving the "raw YAML per environment" problem from the intro.

Helm Commands — The Basic Workflow

- `helm install <release-name> <chart>` — deploy a new release.
- `helm upgrade` — apply changes to an existing release, often triggering `k8s1-5`'s own rolling-update mechanism underneath.
- `helm rollback` — revert to a previous release revision, a direct parallel to `k8s1-5`'s `kubectl rollout undo`, tracked at the whole-chart/release level instead of a single Deployment.
- `helm uninstall` — remove a release and everything it created.

Helm Repositories

Charts can be packaged and shared via repositories, conceptually similar to a package registry. Many popular applications — databases, monitoring stacks — already have official or community Helm charts available, meaning common infrastructure components often don't need to be written from scratch at all — a genuinely practical, time-saving point.

When Helm Is Worth the Extra Layer

Honest framing, matching this course's own recurring convention: for a single, simple application with one environment, raw YAML plus `kubectl apply` (Course 1's own approach) is perfectly fine and arguably simpler to understand. Helm earns its added complexity specifically once there are multiple environments needing different configuration, or a genuinely complex multi-resource application worth packaging and reusing — exactly the "8 YAML files, multiple environments" scenario the intro raised.

Helm & GitOps

Helm charts, being just files, version-control naturally (`git1` - `git3`), and are commonly the actual deployment artifact a GitOps pipeline applies — a topic Course 2's own `k8s2-9` covers directly.

HELM UPGRADE WITH A PARTIAL VALUES FILE CAN SILENTLY RESET OTHER VALUES

A genuinely real gotcha: running `helm upgrade` with only a new, small values file (say, just an updated image tag) — without including previously-set custom values like a production replica count — can reset those OTHER values back to the chart's defaults, unless `--reuse-values` is used or the full, correct values file(s) are passed every time. A real, common source of "why did my production config just change" incidents.

Hands-On Exercises

EXERCISE 1

Explain what problem Helm's templating specifically solves that raw YAML manifests (Course 1's own approach) cannot solve on their own, using the dev/staging/prod scenario.

View solution

EXERCISE 2

Explain the relationship between a Helm "chart" and a Helm "release." Are they the same thing, and if not, how do they differ?

View solution

EXERCISE 3

A team runs `helm upgrade` on their production release using only a new, smaller values file that specifies an updated image tag, without including their previously-set custom replica count. Explain what might happen to their replica count as a result, and how this could cause a real incident.

📄 View solution

CHAPTER 3 QUICK REFERENCE

- **Chart** = the package (Chart.yaml + values.yaml + templates/); **release** = a named, tracked instance of a chart deployed with specific values
- Templated YAML renders down to ordinary Kubernetes manifests — nothing from Course 1 becomes obsolete
- Different values files let the same chart deploy meaningfully different configs per environment
- `install` / `upgrade` / `rollback` / `uninstall` — rollback is `rollout undo`, tracked at the release level
- Repositories provide pre-built charts for common infrastructure — often no need to write YAML from scratch
- Helm earns its complexity with multiple environments or genuinely complex apps — a single simple app is fine with raw YAML
- A partial values file on `helm upgrade` can silently reset other config back to defaults — a real incident source
- **Next chapter:** Networking Deep Dive — CNI plugins, Network Policies, a service mesh overview

CHAPTER 4 OF 10

RBAC & Security

Kubernetes Intermediate/Advanced

Chapter 4 · RBAC & Security

This chapter closes two loops left open earlier in this course: `k8s1-7`'s promise that "who can read Secret objects at all" would be covered properly here, and `k8s1-10`'s brief mention of per-namespace access control. It also revisits `cloud1-6`'s IAM material — from Cloud Platforms — now applied specifically inside a Kubernetes cluster.

Why Kubernetes Needs Its Own Access Control

Revisiting `cloud1-6`'s AuthN-vs-AuthZ split directly: getting *into* a cluster (authentication — typically via the cloud provider's own IAM feeding into `k8s1-4`'s kubeconfig) is a different question from what you're allowed to *do* once you're in (authorization). Kubernetes' own **RBAC** (Role-Based Access Control) system handles authorization specifically, entirely separate from whatever authentication mechanism sits in front of it.

The Building Blocks — Roles, ClusterRoles, RoleBindings, ClusterRoleBindings

- **Role** — a set of permissions (verbs like `get` / `list` / `create` / `delete` on specific resource types), scoped to *one namespace* (`k8s1-10`'s namespace material).
- **ClusterRole** — the same idea, but cluster-scoped — for cluster-scoped resources (`k8s1-10`'s own Nodes example) or reused across multiple namespaces.
- **RoleBinding** — grants a Role's permissions to a specific user/group/ServiceAccount, within one namespace.
- **ClusterRoleBinding** — grants cluster-wide.

Genuinely important to be careful about: a Role/RoleBinding pair is namespace-scoped even if a Role with the exact same name exists in another namespace — they're completely separate objects.

ServiceAccounts — Identity for Pods, Not People

Humans authenticate via the cluster's own auth mechanism, often tied to cloud provider IAM. But pods and applications sometimes need their *own* identity to interact with the Kubernetes API directly — a controller or

operator that needs to list/watch other resources, for instance. A **ServiceAccount** is exactly this: an identity for a pod or process, not a person.

> **EVERY POD GETS A DEFAULT SERVICEACCOUNT AUTOMATICALLY**
>
> An easy-to-overlook default with real security implications: if none is specified, a pod is automatically assigned the namespace's `default` ServiceAccount. It typically has minimal but non-zero permissions — worth being deliberate about rather than assuming it's harmless.

A Concrete RBAC Example, Explained

```yaml
apiVersion: rbac.authorization.k8s.io/v1
kind: Role
metadata:
  name: pod-reader
  namespace: shop-app
rules:
  - apiGroups: [""]
    resources: ["pods"]
    verbs: ["get", "list"]
---
apiVersion: rbac.authorization.k8s.io/v1
kind: RoleBinding
metadata:
  name: monitoring-can-read-pods
  namespace: shop-app
subjects:
  - kind: ServiceAccount
    name: monitoring-sa
roleRef:
  kind: Role
  name: pod-reader
```

A read-only Role, a ServiceAccount for a monitoring tool, and a RoleBinding connecting them — exactly the kind of setup `k8s2-2`'s own log-agent/monitoring DaemonSet example would genuinely need to query the Kubernetes API for pod information.

The Principle of Least Privilege, Applied to Kubernetes

Directly reusing `dbsec1-3`/`cloud1-6`'s least-privilege material: never grant `cluster-admin` or broad wildcard permissions (`"*"` on `"*"`) when a narrowly-scoped Role would do.

> **"CLUSTER-ADMIN FOR NOW, WE'LL TIGHTEN IT LATER" — A REAL, COMMON ANTI-PATTERN**

> Granting every ServiceAccount cluster-admin during development, to avoid dealing with permissions while iterating, then never actually tightening it before production — exactly the same anti-pattern `cloud1-6` flagged for cloud IAM, recurring here at the Kubernetes RBAC layer.

Closing the Loop on Chapter 1-7's Secrets Warning

`k8s1-7` stated plainly that Secrets are only base64-encoded, not encrypted, and that "RBAC restricting who can read Secret objects at all" was part of the real protection — deferred to this chapter. Concretely: a Role can specifically restrict `get` / `list` access on Secret resources. Even though a Secret's value is trivially decodable once retrieved, properly scoped RBAC ensures far fewer identities can actually retrieve and decode it in the first place — closing that loop directly.

Pod Security Standards, Briefly

A different, complementary layer: Pod Security Standards (replacing the older, now-removed PodSecurityPolicy) define baseline security postures a pod's own spec must comply with — disallowing running as root, disallowing privileged containers — enforced at the namespace level via labels.

Genuinely distinct from RBAC: RBAC controls *who* can create or modify resources. Pod Security Standards control *what* a pod is allowed to actually configure or run as, regardless of who created it — two complementary layers worth not conflating.

Hands-On Exercises

EXERCISE 1

Explain the difference between a Role and a ClusterRole, and between a RoleBinding and a ClusterRoleBinding — specifically regarding scope.

View solution

EXERCISE 2

Explain what a ServiceAccount is for, and why an application/pod needs one distinct from a human user's own cluster credentials.

View solution

EXERCISE 3

Explain how properly scoped RBAC closes the security gap left open by Chapter 1-7's "Secrets are only base64-encoded, not encrypted" warning. What specifically does RBAC restrict that base64 encoding alone doesn't?

View solution

CHAPTER 4 QUICK REFERENCE

- Authentication (getting in) vs. RBAC/authorization (what you can do) — the same split as `cloud1-6`, now inside Kubernetes itself
- **Role/RoleBinding** — namespace-scoped; **ClusterRole/ClusterRoleBinding** — cluster-wide
- **ServiceAccount** — identity for pods, not people; every pod gets a default one automatically, worth being deliberate about
- Never grant cluster-admin as a permanent "development shortcut" — the same anti-pattern as broad cloud IAM policies
- Scoped RBAC on Secrets restricts *who can even retrieve* a Secret — the missing piece base64 encoding alone never provided
- **Pod Security Standards** — complementary to RBAC; controls what a pod is allowed to run as, not who can create it
- **Next chapter:** Networking Deep Dive — CNI plugins, Network Policies, a service mesh overview

CHAPTER 5 OF 10

Networking Deep Dive

Kubernetes Intermediate/Advanced

Chapter 5 · Networking Deep Dive

Chapter 4 covered who can call the Kubernetes API. This chapter covers a different layer entirely: network-level traffic control *within* the cluster, going deeper than `k8s1-6`'s basic Services material.

CNI — The Container Network Interface

Kubernetes itself doesn't implement pod networking directly — it delegates this to a **CNI** (Container Network Interface) plugin, a pluggable standard interface, much like Chapter 1's own CRI/containerd pluggability point. Popular implementations: Calico, Cilium, Flannel — each with different capabilities and performance characteristics. Worth knowing directly: some CNI plugins also implement Network Policies (below), while simpler ones (basic Flannel, for instance) don't — genuinely practical to know when troubleshooting later, since "why doesn't my NetworkPolicy do anything" can sometimes simply mean the CNI plugin in use doesn't support it at all.

The Default — Flat, Unrestricted Pod Networking

An important baseline fact: by default, **every** pod in a Kubernetes cluster can communicate with **every other** pod, across any namespace, with no restrictions at all. Genuinely surprising coming from a more locked-down mental model. This flat networking model is exactly what makes Chapter 6's Services/DNS work simply — but it also means a compromised pod can, by default, reach anything else in the cluster, a real security consideration worth naming directly.

Network Policies — Restricting Pod-to-Pod Traffic

A **NetworkPolicy** defines allow rules for traffic to/from pods matching a label selector (Chapter 5, Course 1's own label mechanism, reused again). By default, with no NetworkPolicy at all, everything is allowed.

DENY-BY-DEFAULT, BUT ONLY ONCE SELECTED

> The instant *any* NetworkPolicy selects a given pod, that pod's traffic switches to deny-by-default, except for what's explicitly allowed — a genuinely important, easy-to-get-wrong-the-first-time behavior. Applying policies incrementally means thinking carefully about exactly what each newly-selected pod actually needs allowed.

A Concrete NetworkPolicy Example

```yaml
apiVersion: networking.k8s.io/v1
kind: NetworkPolicy
metadata:
  name: db-allow-backend-only
  namespace: shop-app
spec:
  podSelector:
    matchLabels:
      app: db
  policyTypes: [Ingress]
  ingress:
    - from:
        - podSelector:
            matchLabels:
              app: backend
```

Restricting `k8s1-12`'s own capstone database to accept traffic *only* from pods labeled `app: backend` — closing a real security gap the capstone's own architecture left implicitly open, despite Chapter 6's Service already existing. This is genuinely the Kubernetes-internal equivalent of `cloud1-5`'s own private-subnet analogy — defense in depth at the pod level, rather than the VPC/subnet level.

Service Meshes — A Brief Overview

Foreshadowed since Chapter 3, Course 1's own sidecar material: a service mesh — Istio and Linkerd being the most common — adds a layer of infrastructure specifically for managing service-to-service communication, typically implemented via a sidecar proxy (Chapter 3's own pattern, now with a concrete real use case) injected into every pod, intercepting all network traffic. What it adds beyond NetworkPolicies:

- Automatic **mutual TLS (mTLS)** between services (`crypto1`'s own TLS/certificate material).
- Fine-grained traffic routing and canary deployments.
- Detailed per-service traffic metrics and observability (`k8s2-7`'s upcoming chapter builds on this).
- Retry and circuit-breaking logic.

Genuinely powerful — and genuinely adding real operational complexity.

When You Need a Service Mesh vs. When NetworkPolicies Are Enough

Honest framing matching this course's own recurring convention (`k8s1-1`, `k8s1-8`): most clusters, even production ones, genuinely don't need a full service mesh. NetworkPolicies alone solve basic traffic restriction. A service mesh earns its complexity specifically at real microservices scale, when mTLS, fine-grained routing, or detailed per-service metrics become genuinely necessary — not adopted just because the name is well known.

FORGETTING DNS IN A NETWORKPOLICY IS A REAL, CONFUSING TRAP

Applying a NetworkPolicy that selects pods but forgetting an explicit rule allowing DNS traffic (typically to `kube-system`, port 53) can break that pod's ability to resolve *any* DNS name at all — including Chapter 6's own Service names. A genuinely common, confusing troubleshooting scenario, since the symptom (broken DNS resolution) looks nothing like "a networking policy problem" at first glance.

Hands-On Exercises

EXERCISE 1

Explain the default pod networking behavior in Kubernetes before any NetworkPolicy is applied, and explain precisely what changes the moment a NetworkPolicy selects a given pod.

View solution

EXERCISE 2

Explain what a CNI plugin is responsible for, and why "my NetworkPolicy isn't working" can sometimes have nothing to do with the policy YAML itself being wrong.

View solution

EXERCISE 3

A team applies a NetworkPolicy to their backend pods allowing only traffic from their frontend pods. Afterward, the backend pods can no longer resolve any Service DNS names at all, even though the policy only mentions frontend traffic. Explain what's likely missing from their NetworkPolicy and why.

View solution

CHAPTER 5 QUICK REFERENCE

- **CNI** — a pluggable interface Kubernetes delegates pod networking to; not all CNI plugins support NetworkPolicies

- Default pod networking is **flat and unrestricted** — any pod can reach any other pod, cluster-wide, until a NetworkPolicy says otherwise
- The moment a NetworkPolicy selects a pod, that pod becomes **deny-by-default** except for explicitly allowed traffic
- NetworkPolicies are the pod-level equivalent of `cloud1-5`'s private-subnet pattern — defense in depth
- **Service mesh** (Istio/Linkerd) — sidecar-based mTLS, fine-grained routing, and observability, genuinely powerful but adds real complexity
- Most clusters don't need a full mesh — NetworkPolicies alone are enough until real microservices-scale needs emerge
- Forgetting an explicit DNS-allow rule in a NetworkPolicy silently breaks all Service name resolution — a real, confusing trap
- **Next chapter:** Autoscaling — Horizontal Pod Autoscaler, Vertical Pod Autoscaler, Cluster Autoscaler

CHAPTER 6 OF 10

Autoscaling

Kubernetes Intermediate/Advanced

Chapter 6 · Autoscaling

Chapter 5 (Course 1) named manual scaling and pointed here for the automatic version. This chapter closes that loop, and revisits `cloud1-3`'s own VM-level auto-scaling material at the Kubernetes/pod level.

Revisiting Manual Scaling

`k8s1-5`'s approach: `kubectl scale deployment --replicas=N`, or editing the YAML directly — a fixed, manually-chosen number. Genuinely fine for predictable load; doesn't respond to real-time demand changes automatically.

Horizontal Pod Autoscaler (HPA) — Scaling Pod Count

HPA automatically adjusts a Deployment's (or StatefulSet's) replica count based on observed metrics — most commonly CPU/memory utilization. Critically, HPA's calculations are based on the percentage of *requested* resources actually being used (`k8s1-10`'s own requests/limits material) — exactly why setting requests accurately matters even more once autoscaling is involved. This is another instance of Chapter 2 (Course 1)'s reconciliation loop: observe the current metric value, compare against target, adjust replica count to reconcile. HPA can also scale on custom metrics (e.g. requests-per-second) via metrics adapters, worth knowing exists without deep detail here.

A Concrete HPA YAML, Explained

```yaml
apiVersion: autoscaling/v2
kind: HorizontalPodAutoscaler
metadata:
  name: backend-hpa
spec:
  scaleTargetRef:
    apiVersion: apps/v1
    kind: Deployment
    name: backend
  minReplicas: 3
  maxReplicas: 15
  metrics:
    - type: Resource
      resource:
        name: cpu
        target:
          type: Utilization
          averageUtilization: 70
```

Yet another distinct API group — `autoscaling/v2`. `scaleTargetRef` points at the Deployment being scaled, reusing Chapter 5's own structure. `maxReplicas` is a genuinely important safety bound — always cap it, tying directly to `cloud1-9`'s cost-consciousness material, since unbounded autoscaling under a traffic spike or a misbehaving metric could scale to a genuinely expensive pod count.

Vertical Pod Autoscaler (VPA) — Scaling Pod Size

A genuinely different, complementary approach: rather than adding *more* pods (HPA), VPA adjusts the resource requests/limits (`k8s1-10`'s material) of *existing* pods, recommending or automatically applying more appropriate CPU/memory values based on observed actual usage over time — directly solving Chapter 10's "requests set too low/too high" sizing problem automatically rather than requiring manual tuning.

> **VPA'S DEFAULT MODE RESTARTS PODS TO APPLY NEW VALUES**
>
> Since `k8s1-10` established that resource requests are only read at pod creation/scheduling time, VPA's default "Auto" update mode genuinely restarts pods to apply new resource values — a real operational trade-off worth knowing about directly, not a silent, cost-free adjustment.

HPA + VPA Together — A Genuine Conflict Worth Knowing About

Running HPA (based on CPU/memory utilization) and VPA (which changes the resource *requests* those percentages are calculated against) targeting the *same* workload simultaneously can create confusing, conflicting behavior — VPA changing requests changes what "70% utilization" even means for HPA's own calculation. Worth being deliberate about this combination rather than combining the two carelessly.

Cluster Autoscaler — Scaling the Nodes Themselves

A genuinely different layer from HPA/VPA — both of those scale *within* the existing set of nodes. Cluster Autoscaler instead adds or removes actual **nodes** (VMs, directly tying to `cloud1-3`'s own VM-level auto-scaling groups material), based on whether currently-pending pods can't be scheduled due to insufficient capacity, or whether nodes sit significantly underutilized. HPA/VPA operate on pods within a fixed node set; Cluster Autoscaler operates on the nodes themselves — two different scaling problems that work together: HPA adds pods → if no node has room, those pods stay **Pending** (`k8s1-3`'s pod phase material) → Cluster Autoscaler notices the pending pods and adds a new node → the pods get scheduled.

The Full Autoscaling Picture — All Three Layers Together

```
1. Traffic spikes
2. HPA adds more pod replicas in response to rising CPU
3. If existing nodes lack capacity for the new pods -> they become PENDING
4. Cluster Autoscaler notices the Pending pods, provisions a new node
5. The Pending pods get scheduled onto it
   (meanwhile, VPA, if configured, separately and gradually right-sizes
    each pod's own resource requests based on longer-term usage trends)
```

A genuinely satisfying "all three working together" picture, closing the loop on manual scaling, resource requests, and node-level provisioning all at once.

> **ACCURATE RESOURCE REQUESTS NOW MATTER EVEN MORE**
>
> `k8s1-10`'s "set requests accurately" point becomes doubly important with autoscaling in play — both HPA's percentage-based calculations and VPA's own recommendations directly depend on requests being meaningful in the first place.

Hands-On Exercises

EXERCISE 1

Explain what specific problem HPA solves that Chapter 5's manual `kubectl scale` doesn't, and explain why setting resource requests (Chapter 10) accurately matters more once HPA is in use, not less.

View solution

EXERCISE 2

Explain the difference between what HPA scales and what VPA scales, and explain why running both against the same workload's CPU-based metrics simultaneously can cause confusing behavior.

View solution

EXERCISE 3

A traffic spike causes HPA to want to scale a Deployment from 5 to 20 replicas, but only enough node capacity exists for 12 more pods. Explain what happens to the remaining pods HPA tried to create, and what would need to happen next for all 20 to actually become Running.

View solution

CHAPTER 6 QUICK REFERENCE

- **HPA** — scales pod COUNT, based on utilization percentage of requested resources; always set `maxReplicas` as a cost safeguard
- **VPA** — scales pod SIZE (requests/limits); default mode restarts pods to apply changes
- Running HPA and VPA on the same CPU-based metric simultaneously can conflict — VPA changes the very baseline HPA's percentage is calculated against
- **Cluster Autoscaler** — scales NODES themselves, triggered by Pending pods lacking capacity or significantly underutilized nodes
- Full picture: HPA adds pods → Pending if no room → Cluster Autoscaler adds a node → pods scheduled; VPA right-sizes independently over time
- Accurate resource requests (Ch.10) matter even more once autoscaling depends on them
- **Next chapter:** Observability in Kubernetes — built-in tooling limitations, Prometheus/Grafana integration

CHAPTER 7 OF 10

Observability

Kubernetes Intermediate/Advanced

Chapter 7 · Observability in Kubernetes

Chapter 6's autoscaling itself depends on metrics. This chapter covers how to actually see what's happening inside a cluster — revisiting `cloud1-8`'s metrics/logs/traces framework, now specifically at the Kubernetes layer.

Revisiting Cloud1-8's Three Pillars, for Kubernetes Specifically

Metrics, logs, traces — `cloud1-8`'s own framework, from Cloud Platforms. This chapter covers how each pillar actually works *at* the Kubernetes layer, distinct from the cloud-provider-level monitoring that chapter covered.

kubectl's Own Built-In Observability — First-Line Tools

- `kubectl logs` — a specific container's stdout/stderr; the `--previous` flag retrieves a crashed container's *last* logs before it restarted, genuinely important since a freshly-restarted container's live logs won't show what caused the previous crash at all.
- `kubectl describe` — a resource's full state, including recent Events — often the fastest way to see *why* a pod is stuck: a failed scheduling reason, an image pull failure.
- `kubectl get events` — a cluster-wide or namespace-scoped event stream, useful for correlating *when* something happened across multiple resources — directly foreshadowing Course 2's own `k8s2-8` troubleshooting workflow.

--PREVIOUS IS EASY TO FORGET EXACTLY WHEN IT MATTERS MOST

Right after a crash, the natural instinct is to check `kubectl logs` immediately — but a container that already restarted is showing its NEW logs, not the ones from the crash. `--previous` is specifically what surfaces the actual crash-causing output.

The Limitation of Built-In Tools

Genuinely great for a single resource, right now — but they don't retain history long-term (pod logs disappear when the pod is deleted), don't aggregate across many pods/services easily, and provide no dashboards,

alerting, or trend analysis. Exactly the gap dedicated observability tooling exists to fill — the same log-aggregation problem `cloud1-8` already covered at the cloud-provider level, recurring here specifically inside Kubernetes.

Metrics Server — The Baseline

> **HPA SILENTLY DOES NOTHING WITHOUT METRICS SERVER**
>
> A genuinely easy-to-miss prerequisite: the Metrics Server is what actually powers `kubectl top nodes` / `kubectl top pods`, and it's what HPA (Chapter 6) actually queries for CPU/memory data. Without it installed, resource-based HPA simply doesn't work at all — directly the same pattern as `k8s1-9`'s "Ingress does nothing without a controller" — a foundational piece of infrastructure worth checking explicitly rather than assumed present.

Prometheus — The De Facto Standard for Kubernetes Metrics

Prometheus is a metrics collection and storage system that **pulls** (scrapes) metrics from configured targets at regular intervals, rather than applications pushing metrics to it. Applications commonly expose a `/metrics` HTTP endpoint in a specific text format; Prometheus scrapes it periodically. Node-level metrics exporters (like node-exporter) commonly run as DaemonSets — directly reusing `k8s2-2`'s own DaemonSet material, one exporter instance per node, for exactly the reason that chapter explained.

Grafana — Visualizing What Prometheus Collects

Prometheus stores and queries metrics, but its own native UI is minimal. Grafana is the dashboarding/visualization layer commonly paired with it — Prometheus as the data source, Grafana as the presentation layer. Genuinely worth being clear these are two separate tools with distinct jobs, not one combined product, since that's a common point of confusion for newcomers.

Logs at Scale — Beyond kubectl logs

Since `kubectl logs` only shows one pod at a time and loses history once a pod is gone, real clusters typically run a log-aggregation pipeline — commonly a DaemonSet-based log-shipping agent (Fluentd/Fluent Bit, `k8s2-2`'s own concrete DaemonSet example, now with its actual real-world purpose fully explained) collecting every node's container logs and forwarding them to a centralized store (Elasticsearch, Loki, or a cloud provider's own logging service per `cloud1-8`'s own material), where they persist beyond any individual pod's lifetime and can be searched across the whole cluster.

This Course's Own Scope, Honestly

Matching this course's recurring convention: this chapter is deliberately an *orientation* to Kubernetes observability, not a deep Prometheus/Grafana course — this site's own bucket list has a separate, still-outstanding Observability course topic for that depth. This chapter's job is making sure the Kubernetes-specific pieces (Metrics Server as an HPA prerequisite, DaemonSet-based collection patterns, the built-in-vs-dedicated-tooling gap) are understood — exactly what a dedicated Prometheus/Grafana course would otherwise need to re-explain from scratch in a Kubernetes context anyway.

Hands-On Exercises

EXERCISE 1

A pod crashed and was automatically restarted by its restart policy (Chapter 11, Course 1). A user runs `kubectl logs` and sees only a few seconds of output, nothing explaining the crash. Explain what's likely happening and what command/flag would actually show the crash's cause.

View solution

EXERCISE 2

Explain why Prometheus and Grafana are described as two separate tools with distinct jobs rather than one combined product, and what each one is actually responsible for.

View solution

EXERCISE 3

An HPA is configured correctly with reasonable CPU thresholds, but it never scales the Deployment at all, even under heavy real load. Using this chapter's material, what's a genuinely likely root cause worth checking first, and why?

View solution

CHAPTER 7 QUICK REFERENCE

- `kubectl logs --previous`, `describe`, `get events` — the immediate, first-line built-in tools
- Built-in tools don't retain history, aggregate across pods, or provide dashboards/alerting — the gap dedicated tooling fills
- **Metrics Server** — a genuinely easy-to-miss prerequisite; HPA does nothing without it, same pattern as a controller-less Ingress

- **Prometheus** — pulls/scrapes metrics from `/metrics` endpoints; node exporters commonly run as DaemonSets (`k8s2-2`)
- **Grafana** — a separate visualization layer over Prometheus's data, not a combined product
- Log aggregation (Fluentd/Fluent Bit as a DaemonSet → centralized store) solves what `kubectl logs` can't: history and cross-pod search
- This chapter is an orientation, not a full observability course — a separate, deeper course remains bucket-listed
- **Next chapter:** Troubleshooting Kubernetes — CrashLoopBackOff, ImagePullBackOff, and other common failure patterns

CHAPTER 8 OF 10

Troubleshooting

Kubernetes Intermediate/Advanced

Chapter 8 · Troubleshooting Kubernetes

Chapter 7 built the tools. This chapter catalogs the actual failure patterns you'll run into repeatedly — the Kubernetes-specific counterpart to `cloud2-5`'s own failure-mode catalog from Cloud Platforms.

A Troubleshooting Starting Discipline

Reusing Chapter 7's own tools as a fixed sequence for almost every pod-level problem: `kubectl get pods` first — what state is it actually in? — then `kubectl describe` for the specific reason/Events, then `kubectl logs` (and `--previous`, per Chapter 7) for the actual application output.

CrashLoopBackOff — The Container Keeps Dying

The container starts, exits, and Kubernetes keeps restarting it per Chapter 11's restart policy, with an increasing backoff delay between attempts — hence the name. Genuinely important: this **isn't** a special error state — it's the normal, expected behavior of the restart policy responding to a container that keeps failing. The root cause is almost always in the *application itself*, not Kubernetes. Check `kubectl logs --previous` (Chapter 7) first. Common real causes: an application config error, a Secret/ConfigMap (Chapter 7, Course 1) not actually mounted correctly, or the application failing a startup dependency check.

ImagePullBackOff / ErrImagePull — Kubernetes Can't Get the Image

The kubelet (Chapter 2, Course 1's architecture) can't successfully pull the specified image. Common real causes: a typo in the image name/tag, the image genuinely doesn't exist, the image is in a private registry and the pod lacks correct pull credentials — genuinely tying to Chapter 4's ServiceAccount material, since image pull secrets are attached via the ServiceAccount — or, less commonly, a registry rate limit. `kubectl describe pod` shows the specific pull error message in its Events, usually explaining exactly which of these applies.

Pending Pods — Never Even Started

A pod stuck in Pending (Chapter 3's own pod phase material) means the scheduler (Chapter 2) hasn't been able to place it on any node. Common real causes: insufficient cluster resource capacity for the pod's own requests (Chapter 10) — directly the same scenario Chapter 6's autoscaling chapter walked through, minus Cluster Autoscaler actually being available to fix it — a node selector/affinity rule no current node satisfies, or a PersistentVolumeClaim (Chapter 8) that can't be bound to any available PersistentVolume. `kubectl describe pod` again shows the specific scheduling failure reason.

Readiness Probe Failures — Running, But Never Actually Serving Traffic

A pod stuck showing `0/1 Ready` despite being in the Running phase means its readiness probe (Chapter 11) is failing continuously — the pod is technically alive, but Kubernetes never adds it to a Service's Endpoints (Chapter 6), so it never receives real traffic. Common causes: the readiness probe's path/port is misconfigured — Chapter 6's own `port` / `targetPort` confusion recurring here — or the application genuinely never finishes its own startup dependency (can't reach the database, for instance). Worth naming a genuinely non-obvious cross-chapter connection: a NetworkPolicy (Chapter 5) blocking database access could be the *actual* underlying cause of a readiness probe that looks purely application-side.

DNS Resolution Failures Inside the Cluster

Revisiting Chapter 5's own DNS/NetworkPolicy gotcha as a cataloged, diagnosable symptom rather than just a preventive warning: a pod that can't resolve a Service name (Chapter 6) at all — check whether a NetworkPolicy is blocking egress DNS traffic first, then confirm the target Service actually exists and has matching Endpoints in the first place, rather than assuming DNS itself is broken.

RBAC "Forbidden" Errors — A Different Category Entirely

An application or `kubectl` command failing with a "Forbidden" error is **not** a networking or scheduling problem at all — it means the identity making the request (a ServiceAccount, Chapter 4's material) lacks the necessary Role/RoleBinding permissions for the specific action attempted. Genuinely worth recognizing the error message itself as the fastest signal for which category actually applies — "Forbidden" points straight at RBAC, not at networking, scheduling, or probes.

A Genuinely Useful Troubleshooting Decision Tree

```
1. kubectl get pods  -- what state is it in?

   PENDING?              -> check scheduling/resources/PVC binding
   CRASHLOOPBACKOFF?      -> logs --previous, app config/secrets
   IMAGEPULLBACKOFF?      -> image name, registry credentials
   RUNNING, 0/1 READY?    -> readiness probe config + its own dependencies
   RUNNING, READY, still unreachable? -> DNS / NetworkPolicy / Service Endpoints
   "FORBIDDEN" ERROR SPECIFICALLY?    -> go straight to RBAC, skip the rest
```

THE POD STATE/ERROR MESSAGE ITSELF IS THE FASTEST TRIAGE SIGNAL

Directly paralleling `cloud2-2`'s own failure-type distinction from Cloud Platforms: reading exactly what Kubernetes reports — the specific status or error text — before assuming which category of problem applies is this chapter's single most useful takeaway.

ASSUMING THE WRONG CATEGORY WASTES REAL TIME

Chasing a networking explanation for what's actually an RBAC "Forbidden" error (or vice versa) burns real investigation time. The specific status/error Kubernetes reports is usually a strong, specific signal — read it carefully before deciding which chapter's material actually applies.

Hands-On Exercises

EXERCISE 1

A pod shows STATUS "CrashLoopBackOff" in `kubectl get pods`. Explain what this status actually means mechanically, tying to Chapter 11's restart policy material, and what the first command to run should be.

View solution

EXERCISE 2

A pod is stuck in "Pending" and never starts. List at least two genuinely different root causes this chapter names, and explain how `kubectl describe pod` would help distinguish between them.

View solution

EXERCISE 3

A pod is Running and shows 1/1 Ready, but an application trying to reach it via its Service name gets no response at all. Using this chapter's material, name two genuinely different categories of cause worth checking, tying each to a specific earlier chapter.

📄 View solution

CHAPTER 8 QUICK REFERENCE

- Discipline: `get pods` (state) → `describe` (reason/Events) → `logs --previous` (application output)
- **CrashLoopBackOff** — normal restart-policy behavior for a repeatedly-failing container; root cause is almost always the app itself
- **ImagePullBackOff** — bad image name/tag, missing registry credentials (ServiceAccount-attached), or a rate limit
- **Pending** — the scheduler can't place it: insufficient resources, an unsatisfiable node selector, or an unbound PVC
- **0/1 Ready** — a failing readiness probe; misconfigured port, or a genuine dependency failure (sometimes a NetworkPolicy)
- **DNS failures** — check NetworkPolicy egress rules first, then confirm the Service/Endpoints actually exist
- **"Forbidden"** — always RBAC, never networking/scheduling/probes
- The pod state/error message itself is the fastest triage signal — read it before guessing a category
- **Next chapter:** GitOps & CI/CD for Kubernetes — declarative infra as a natural fit for git-based workflows

CHAPTER 9 OF 10

GitOps & CI/CD

Kubernetes Intermediate/Advanced

Chapter 9 · GitOps & CI/CD for Kubernetes

Chapter 8 covered fixing things once they're already deployed. This chapter covers how deployments get into the cluster in the first place, in a disciplined, reviewable way — building on `cloud1-11`'s IaC material and this site's own `pipelines1` CI/CD course.

Revisiting Cloud1-11's Declarative-vs-Imperative Material

`k8s1-4`'s own point: "real, reusable, version-controllable work should be declarative YAML." This chapter takes that all the way — not just writing YAML declaratively, but making **git itself** the actual source of truth for what's deployed, rather than a human running `kubectl` by hand.

What GitOps Actually Means

A git repository holds the desired state of the cluster — the same YAML/Helm charts this entire course has been writing — as the single source of truth. A dedicated **GitOps controller** running *inside* the cluster continuously watches that repository and automatically reconciles the cluster's actual state to match it.

> **THE RECONCILIATION LOOP, ONE MORE TIME**
> This is, again, Chapter 2 (Course 1)'s reconciliation loop pattern — just with the "desired state" source now being a git repository rather than what's already stored in etcd directly.

GitOps vs. Traditional CI/CD Push-Based Deployment

	TRADITIONAL CI/CD (PUSH)	GITOPS (PULL)
Who initiates deployment?	An external pipeline (`pipelines1`) runs `kubectl apply` / `helm upgrade` against the cluster	A controller inside the cluster pulls changes from git itself
Where do deployment credentials live?	In the external CI/CD system	Nowhere external — the cluster reaches out, nothing reaches in

A genuinely meaningful difference: broad deployment credentials living in an external CI system is a real, larger attack surface than a cluster-internal controller that only ever pulls — directly echoing `k8s2-4`'s own least-privilege material.

ArgoCD & Flux — The Two Common GitOps Controllers

Both watch a git repository (or repositories) and continuously reconcile cluster state to match. **ArgoCD** offers a genuinely useful web UI showing exactly what's in git vs. what's actually deployed, and any **drift** between them — directly tying to `cloud1-11`'s own configuration-drift material. GitOps controllers make drift immediately visible, and can optionally auto-correct it back to match git — genuinely resolving `cloud1-11`'s own "manual fix gets silently overwritten" gotcha, by making that overwriting the actual *intended* behavior rather than a surprise. **Flux** offers similar core capability with different tooling/ecosystem choices, often used more as a Kubernetes-CLI/GitOps-toolkit-native option.

The Full GitOps Workflow

```
1. A developer changes a Helm chart's values.yaml (k8s2-3) in git
2. Opened as a pull request, reviewed (git2-7's own code review process)
3. Merged
4. The GitOps controller notices the change in git
5. It automatically applies the change to the cluster
   -- nobody runs kubectl or helm manually at all
```

Genuinely closing the loop on Chapter 11 (Course 1)'s own "config as code, reviewed like any other code" aspiration.

Secrets in a GitOps World — Revisiting Chapter 1-7's Own Note

`k8s1-7` flagged this chapter directly: "committing raw Secret YAML to git is genuinely risky... tools like Sealed Secrets or External Secrets Operator exist specifically to solve this." Now explained properly: Sealed Secrets encrypts a Secret's value *client-side*, before it's ever committed to git, producing a SealedSecret object that's genuinely safe to commit — only the cluster's own controller, holding the matching private key, can decrypt it back into a real Secret. This resolves the tension between GitOps' own core principle ("everything should be in git") and Chapter 7's own warning ("Secrets shouldn't be committed in a readable form") simultaneously.

When GitOps Is Worth the Setup

Honest framing, matching this course's own recurring convention: for a single-cluster, small-team setup, a straightforward CI/CD pipeline running `kubectl apply` / `helm upgrade` directly (`pipelines1`'s own approach) is genuinely simpler and perfectly reasonable. GitOps earns its setup complexity specifically once

there are multiple clusters/environments needing consistent, auditable, driftless deployment, or a genuine desire to eliminate external systems holding direct cluster deployment credentials.

This Course's Closing Thread — One Pattern, All the Way Through

Worth naming directly, this close to the end of the course: the reconciliation loop (Chapter 2, Course 1) has now appeared as ReplicaSets (Chapter 5), Service Endpoints (Chapter 6), liveness probes (Chapter 11), DaemonSets (`k8s2-2`), HPA (`k8s2-6`), and now a GitOps controller reconciling an *entire cluster* against a git repository — the same one idea, at every scale from a single container's health to a whole cluster's complete configuration.

GITOPS ISN'T A SECURITY FIX ON ITS OWN

The GitOps controller's own identity still needs the same RBAC discipline (`k8s2-4`) as anything else — least privilege applies to the controller's own permissions too, not just to human or application ServiceAccounts.

Hands-On Exercises

EXERCISE 1

Explain the key security-model difference between traditional push-based CI/CD deployment and pull-based GitOps, specifically regarding where cluster deployment credentials live.

View solution

EXERCISE 2

Explain how Sealed Secrets resolves the tension between GitOps wanting everything in git and Chapter 1-7's warning against committing raw Secrets.

View solution

EXERCISE 3

Identify at least four different places across this entire course (both Course 1 and Course 2) where the reconciliation loop pattern from Chapter 2 has appeared in a genuinely different guise, including this chapter's own GitOps controller example.

View solution

CHAPTER 9 QUICK REFERENCE

- **GitOps** — git as the source of truth; an in-cluster controller pulls and reconciles, another reconciliation-loop instance
- **Push** (external CI/CD holds deployment credentials) vs. **pull** (nothing external ever needs them) — a genuine security-model difference
- **ArgoCD** (drift visibility/UI) and **Flux** — the two common controllers
- Full workflow: PR → review (`git2-7`) → merge → controller notices → auto-applies, no manual `kubectl`/`helm`
- **Sealed Secrets** — client-side encryption before commit, resolving the git-everything vs. no-raw-Secrets tension from `k8s1-7`
- Worth the setup at multi-cluster/multi-environment scale; a single-cluster small team is fine with plain CI/CD
- GitOps controllers still need RBAC discipline (`k8s2-4`) applied to their own identity
- **Next chapter:** Capstone — Deploying a Production-Ready Application, combining Helm, RBAC, autoscaling, monitoring, and health checks into one realistic deployment

CHAPTER 10 OF 10

Capstone: Deploying a Production-Ready Application

Kubernetes Intermediate/Advanced

Chapter 10 · Capstone — Deploying a Production-Ready Application

Course 1's own capstone (`k8s1-12`) closed with an honest list of what was deliberately left out. This chapter resolves that list, one item at a time, using everything Course 2 has covered.

Where We Left Off

`k8s1-12`'s shop-app: a namespace, ConfigMap/Secret, three Deployments (frontend, backend, and a simplified database), two ClusterIP Services, an Ingress, resource limits, and probes. Its own closing section named exactly what this chapter now delivers: a proper StatefulSet, RBAC, autoscaling, observability, troubleshooting readiness, Helm packaging, and GitOps deployment.

Step 1 — Packaging as a Helm Chart

`k8s2-3`: the capstone's raw YAML becomes a proper chart — `Chart.yaml`, a `values.yaml` with environment-specific overrides, and a `templates/` directory holding every resource from here on.

Step 2 — A Proper StatefulSet for the Database

`k8s2-1`: the simplified Deployment+PVC database is replaced with a real StatefulSet, using `volumeClaimTemplates` and a headless Service — directly resolving `k8s1-12`'s own warn-box and its own Exercise 2 about this exact simplification.

Step 3 — Locking Down the Database With a NetworkPolicy

`k8s2-5`: the exact NetworkPolicy example from that chapter, applied here for real — restricting database access to only backend pods, closing a security gap that existed silently through the entire Course 1 capstone until now.

Step 4 — RBAC for the Monitoring Stack

`k8s2-4` / `k8s2-7`: a dedicated ServiceAccount, Role, and RoleBinding for a monitoring tool needing read-only pod access — the exact worked example from Chapter 4, now actually deployed as part of the real app.

Step 5 — Autoscaling the Backend

`k8s2-6`: an HPA targeting the backend, with a sensible `maxReplicas` cap — directly applying that chapter's own cost-consciousness warning.

Step 6 — Observability

`k8s2-7`: Prometheus scraping the backend's `/metrics` endpoint, node-exporter running as a DaemonSet (`k8s2-2`'s own concrete example, now with full context), Grafana dashboards on top. An honest note, matching Chapter 7's own framing: full observability setup is beyond this capstone's own scope — this integrates the pieces, it doesn't teach Prometheus/Grafana from scratch.

Step 7 — Deploying via GitOps

`k8s2-9`: the chart lives in git. A PR merges a values change; ArgoCD notices and applies it — no manual `kubectl` or `helm` at all. This closes the loop on the entire deployment workflow, not just the application's own configuration.

Step 8 — If Something Breaks

`k8s2-8`: the troubleshooting decision tree still applies exactly as written — `get pods`, `describe`, `logs --previous`, and reading the specific error/state as the fastest triage signal, regardless of how much additional infrastructure this capstone has layered on top.

The Full Picture — Every Course 2 Chapter's Contribution

STEP	CHAPTER
1	Ch.3 — Helm packaging
2	Ch.1 — StatefulSets
3	Ch.5 — NetworkPolicy
4	Ch.4 — RBAC
5	Ch.6 — Autoscaling

STEP	CHAPTER
6	Ch.7 — Observability
7	Ch.9 — GitOps
8	Ch.8 — Troubleshooting

What's Still Out of Scope, Honestly

Matching this course's own recurring convention: even this "production-ready" capstone doesn't cover everything a real production system would need — proper multi-region/disaster-recovery architecture, a full incident response process (`cloud2-6`'s own material from Cloud Platforms), cost optimization at genuine scale (`cloud1-9` / `cloud2-7`), and a genuinely mature CI pipeline with automated testing before the GitOps merge ever happens. Named honestly, rather than implying this capstone is "done" in any absolute sense.

PRODUCTION-READINESS IS A SPECTRUM, NOT A CHECKBOX

Every item resolved in this chapter closes a real gap — but "production-ready" is an ongoing practice, not a state you reach once and never revisit.

Closing the Full Kubernetes Track

From Chapter 1 (Course 1)'s own "why does orchestration even need to exist" all the way to a GitOps-deployed, autoscaled, RBAC-secured, network-policy-restricted, StatefulSet-backed application — the reconciliation loop, named in Chapter 2, has been the throughline the entire way, appearing at every single layer covered across both courses.

ONE EXAMPLE, EVOLVED ACROSS AN ENTIRE TRACK

Revisiting and evolving a single running application across 22 chapters, rather than building something new every time, is a deliberate choice — it reinforces retention far more than isolated examples would, and it's exactly how a real application actually grows in practice: incrementally, not all at once.

Hands-On Exercises

EXERCISE 1

Identify which specific limitation from `k8s1-12`'s own "what's deliberately out of scope" list each of the following Course 2 additions resolves: (a) the StatefulSet, (b) the NetworkPolicy, (c) the HPA.

View solution

EXERCISE 2

Explain why deploying this production-ready version via GitOps (Chapter 9) is considered a meaningfully different security posture than simply running `helm upgrade` manually from a laptop with cluster-admin credentials.

View solution

EXERCISE 3

Across the entire Kubernetes track (both courses), name the single concept/pattern that recurs most often, and explain in your own words why understanding it deeply matters more than memorizing any single YAML snippet.

View solution

CHAPTER 10 QUICK REFERENCE — COURSE & TRACK COMPLETE

- Course 1's capstone limitations, resolved one by one: Helm (Ch.3), StatefulSet (Ch.1), NetworkPolicy (Ch.5), RBAC (Ch.4), HPA (Ch.6), observability (Ch.7), GitOps (Ch.9), troubleshooting readiness (Ch.8)
- Production-readiness remains a spectrum — multi-region, incident response, cost optimization at scale, and mature CI testing are still honestly out of scope
- The reconciliation loop (Ch.2, Course 1) has appeared at every layer, across both courses, right up through this chapter's own GitOps deployment
- **Course 2 complete** — Kubernetes Intermediate/Advanced, 10 chapters
- **Full Kubernetes track complete** — Fundamentals + Intermediate/Advanced, 22 chapters across 2 courses