GrowthSchool

KUBERNETES

COMPLETE NOTES

LEARN - BUILD - DEPLOY

Clusters, Pods, Services, storage, security and production ops

growthschool.cc | @growthschool.cc

Learning Roadmap

GrowthSchool

Learn Kubernetes in order. Every page has explanation, real kubectl/YAML, a diagram or example, common mistakes, a student lab and a revision snapshot. We deploy an ecommerce "Shop API" throughout.

HOW TO USE THESE NOTES

Practice on kind or minikube - a free local cluster. Type every command; read Events daily. Use pages 30-34 for fast revision before interviews.

1. Why Kubernetes

GrowthSchool

IN SIMPLE TERMS
Kubernetes (also written K8s) is a system that runs and manages your containers for you across many machines - starting them, restarting crashed ones, and scaling them up or down automatically.

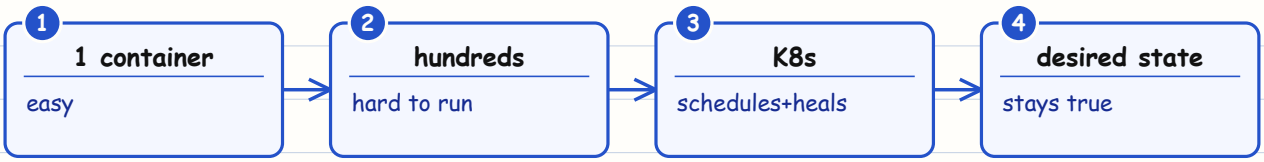
* The Problem It Solves

- One container is easy. Hundreds across many machines need scheduling, healing, networking and safe releases.
- Kubernetes continuously reconciles the cluster toward the desired state you declare in YAML.

* What You Get

- Self-healing, rolling updates/rollbacks, service discovery, load balancing, autoscaling and secrets.
- It is declarative: you describe the result; controllers make it happen and keep it true.

Why Kubernetes



REAL-WORLD EXAMPLE

```
# desired state in one command
kubectl apply -f shop-api.yaml
kubectl get deploy,pods,svc
# controllers keep actual state == desired state
```

COMMON MISTAKE
Reaching for Kubernetes for a single small app. It adds real operational complexity. Use it when you genuinely need scale, resilience and many services - not for a hello-world site.

REVISION SNAPSHOT

ASK YOURSELF	What does "declarative reconciliation" mean in Kubernetes?
DO THIS	Name three problems Kubernetes solves that a single Docker host cannot.

2. Containers Recap

GrowthSchool

IN SIMPLE TERMS
A container is your app packaged with everything it needs to run; an image is the saved template you start containers from. Kubernetes simply runs these containers at scale.

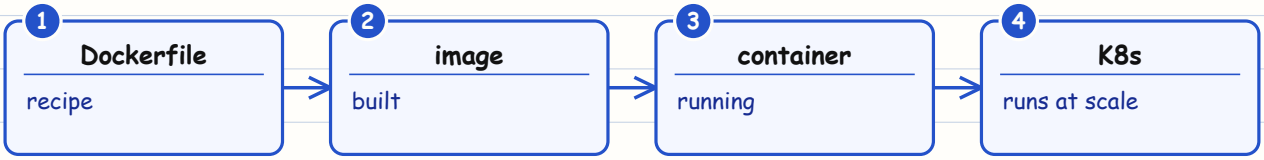
* Image vs Container

- An image is an immutable package (code + runtime + deps). A container is a running instance of it.
- Kubernetes schedules containers (inside Pods); it does not replace Docker knowledge, it builds on it.

* Why It Matters Here

- The same image runs on a laptop, CI and the cluster. Rollbacks are just running an older image tag.
- Never store durable data only in a container filesystem - use persistent storage.

Container to cluster



REAL-WORLD EXAMPLE

```
docker build -t shop-api:1.0 .
docker run -p 8080:8080 shop-api:1.0
# then Kubernetes runs the same image at scale
kubectl create deploy shop-api --image=shop-api:1.0
```

COMMON MISTAKE
Using the latest tag for cluster workloads. It is mutable and ambiguous, making rollbacks and debugging painful. Pin immutable tags like shop-api:1.4.2 or a digest.

REVISION SNAPSHOT

ASK YOURSELF	Why must container images be immutable and versioned for Kubernetes?
DO THIS	Build an image, run it locally, then create a Deployment from it.

3. Cluster Architecture

GrowthSchool

IN SIMPLE TERMS
A Kubernetes cluster is a group of machines working as one: a "control plane" that makes decisions and "worker nodes" that actually run your apps.

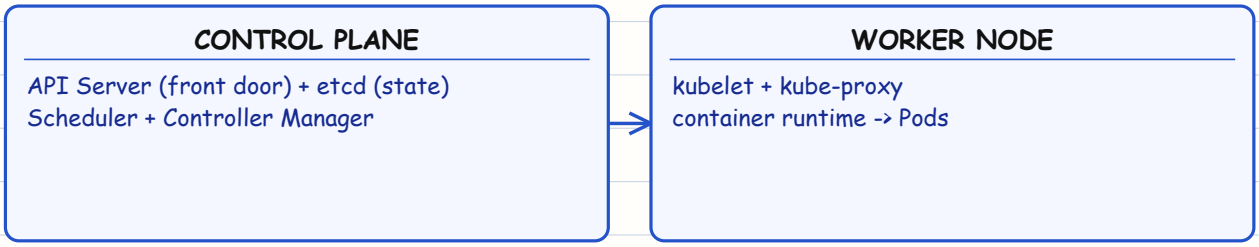
* Two Halves

- The control plane makes decisions; worker nodes run your Pods. Together they form a cluster.
- You talk to the cluster via the API Server using kubectl or CI/CD tooling.

* Desired vs Actual

- You submit desired state; controllers observe actual state and act to close the gap, forever.

Cluster: Control Plane (decides) + Worker Nodes (run)



REAL-WORLD EXAMPLE

```
kubectl cluster-info
kubectl get nodes -o wide
kubectl get componentstatuses   # legacy health view
kubectl get pods -n kube-system
```

REVISION SNAPSHOT

ASK YOURSELF	Which half decides and which half runs your workloads?
DO THIS	List your nodes and the system Pods running in kube-system.

4. Control Plane in Depth

GrowthSchool

IN SIMPLE TERMS
The control plane is the brain of the cluster - the components that decide what should run where and keep everything in the state you asked for.

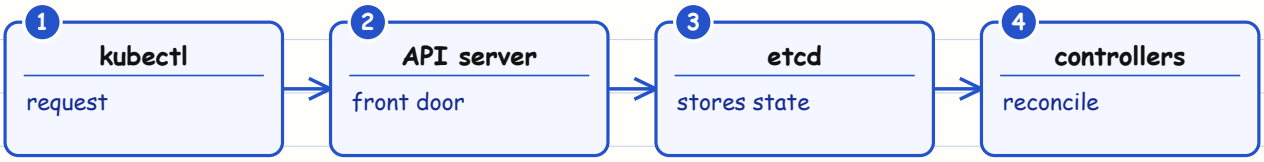
* The Components

- API Server: the front door and only component that talks to etcd. etcd: the key-value store of cluster state.
- Scheduler: picks a node for each Pod. Controller Manager: runs reconciliation loops for objects.

* Cloud Controller

- The cloud controller manager integrates load balancers, nodes and volumes with a cloud provider.
- The control plane does not run your business Pods; it coordinates and decides.

kubectl to running Pod



REAL-WORLD EXAMPLE

```
kubectl get pods -n kube-system
kubectl -n kube-system get pod -l component=kube-apiserver
kubectl describe pod -n kube-system <scheduler-pod>
# etcd holds ALL cluster state - back it up!
```

COMMON MISTAKE
Ignoring etcd backups. etcd is the single source of truth; losing it loses the whole cluster state. On self-managed clusters, snapshot etcd regularly and test restores.

REVISION SNAPSHOT

ASK YOURSELF	What is the only component that reads/writes etcd directly?
DO THIS	Identify the control-plane Pods and explain each one's job.

5. Worker Nodes in Depth

GrowthSchool

IN SIMPLE TERMS
A worker node is a machine that actually runs your application containers, with small agents that take orders from the control plane.

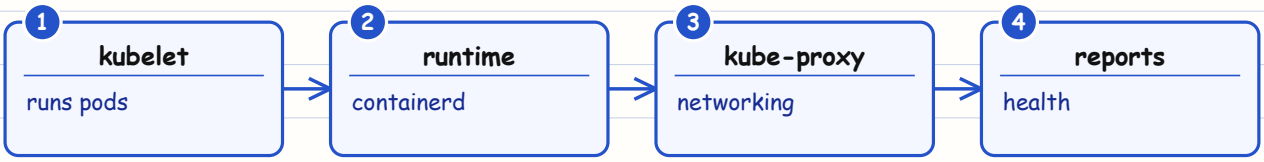
* Node Agents

- kubelet runs on each node, creates Pods assigned to it, and reports health back to the API Server.
- kube-proxy programs network rules so Service traffic reaches the right Pods.

* The Runtime

- A CRI runtime (containerd or CRI-O) actually pulls images and runs containers.
- Unhealthy nodes stop sending heartbeats; the control plane reschedules their Pods when possible.

Inside a worker node



REAL-WORLD EXAMPLE

```
kubectl get nodes
kubectl describe node <node>      # capacity, conditions
kubectl top nodes                 # needs metrics-server
kubectl get pods -o wide          # which node each Pod is on
```

COMMON MISTAKE
Treating nodes as pets. A node can die at any time; design workloads (via controllers) so Pods are recreated elsewhere automatically. Do not store state on the node.

REVISION SNAPSHOT

ASK YOURSELF	What do kubelet and kube-proxy each do on a worker node?
DO THIS	Describe a node and read its capacity, conditions and allocatable resources.

6. kubectl Basics

GrowthSchool

IN SIMPLE TERMS
kubectl is the command-line tool you use to talk to a Kubernetes cluster - to create, view, and change things.

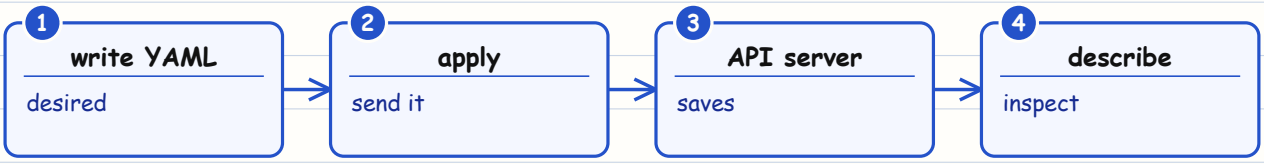
* The Universal Tool

- kubectl talks to the API Server. get lists, describe explains, logs reads output, apply creates/updates.
- Use -n for namespace, -o wide/yaml/json for output, and --dry-run=client to preview YAML.

* Imperative vs Declarative

- Imperative (create, run) is fast for learning; declarative (apply -f) is how you run production.
- Generate YAML with --dry-run=client -o yaml, then edit and apply it.

The kubectl loop



REAL-WORLD EXAMPLE

```
kubectl get pods -A
kubectl create deploy web --image=nginx --dry-run=client -o yaml > web.yaml
kubectl apply -f web.yaml
kubectl describe deploy web
kubectl explain deployment.spec.strategy
```

COMMON MISTAKE
Living on imperative commands in production. They are not reproducible or reviewable. Capture YAML in Git and apply it, so every change has history and can be rolled back.

REVISION SNAPSHOT

ASK YOURSELF	How do you generate starter YAML without creating the object?
DO THIS	Create a deployment YAML with --dry-run, edit it, then apply it.

7. YAML, Namespaces & Metadata

GrowthSchool

IN SIMPLE TERMS
A YAML manifest is a text file where you describe what you want Kubernetes to create; a namespace is a folder-like way to group and separate resources.

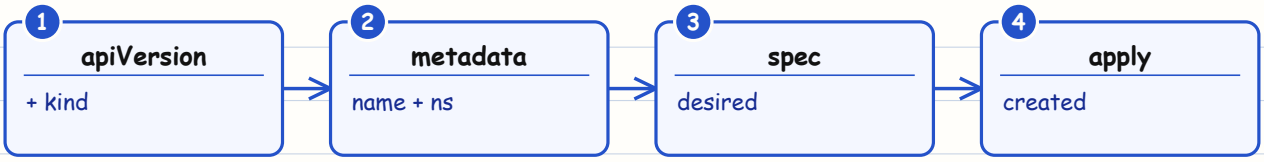
* Manifest Anatomy

- apiVersion selects the API; kind is the object type; metadata identifies it; spec is desired state.
- status is written by Kubernetes - never use it to configure an app.

* Namespaces

- Namespaces separate environments, teams and system components within one cluster.
- A namespace is not a hard security boundary alone - pair it with RBAC, quotas and NetworkPolicy.

Manifest anatomy



REAL-WORLD EXAMPLE

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: shop-api
  namespace: shop
spec:
  replicas: 3
kubectl create namespace shop
```

COMMON MISTAKE
Dumping everything into the default namespace. It gets messy and risky fast. Create purposeful namespaces (shop, monitoring) and set quotas per namespace.

REVISION SNAPSHOT

ASK YOURSELF	What are the four top-level fields of almost every manifest?
DO THIS	Create a namespace and deploy a labelled object into it.

8. Labels, Selectors & Annotations

GrowthSchool

IN SIMPLE TERMS
Labels are simple name tags you stick on objects, and selectors find objects by those tags. This tagging is how Kubernetes links things together.

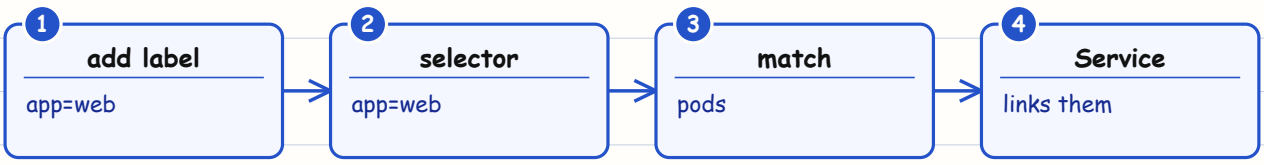
* Labels & Selectors

- Labels are key/value tags (app=shop-api, tier=backend). Selectors query them.
- Services and Deployments find their Pods through label selectors - this is the glue of Kubernetes.

* Annotations

- Annotations hold non-identifying metadata for tools (e.g. ingress config, checksums).
- Use labels for selection/grouping; use annotations for machine-readable extras.

Labels link things



REAL-WORLD EXAMPLE

```
kubectl label pod shop-api-x tier=backend
kubectl get pods -l app=shop-api,tier=backend
kubectl get pods -l "env in (prod,staging)"
kubectl annotate deploy shop-api note="owned by payments team"
```

COMMON MISTAKE
Inconsistent labels (App vs app, api vs shop-api). Selectors then silently match nothing. Agree a label convention and apply it everywhere.

REVISION SNAPSHOT

ASK YOURSELF	How do Services connect to the right Pods?
DO THIS	Select Pods by two labels at once, then by a set-based selector.

9. Pods

GrowthSchool

IN SIMPLE TERMS
A Pod is the smallest thing Kubernetes runs - a wrapper around one (or a few) containers that share a network address and storage.

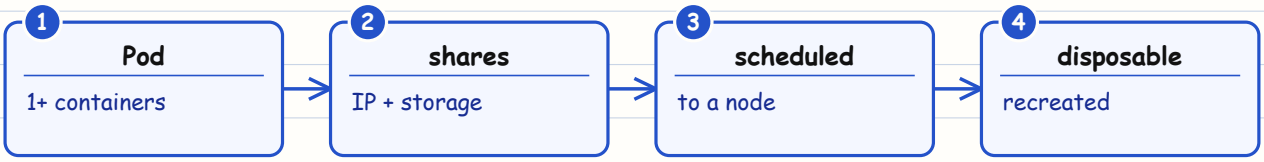
* The Smallest Unit

- A Pod wraps one or more containers that share an IP, localhost and volumes.
- Usually one app container per Pod; Pods are the unit Kubernetes schedules and heals.

* Pods Are Disposable

- Never depend on a Pod name or IP for stable traffic - they change as Pods are recreated.
- A controller (Deployment) should own production Pods so failures are recreated automatically.

A Pod



REAL-WORLD EXAMPLE

```
kubectl run demo --image=nginx
kubectl get pods -o wide
kubectl describe pod demo
kubectl delete pod demo    # a bare Pod is NOT recreated
```

COMMON MISTAKE
Running production apps as bare Pods. If the node dies, the Pod is gone for good. Always use a controller like Deployment so Pods self-heal.

REVISION SNAPSHOT

ASK YOURSELF	Why should you never rely on a Pod's IP for client traffic?
DO THIS	Create a bare Pod, delete it, and confirm nothing recreates it.

10. Multi-Container Pods

GrowthSchool

IN SIMPLE TERMS
A multi-container Pod holds a main app container plus small helper containers that support it and share the same Pod.

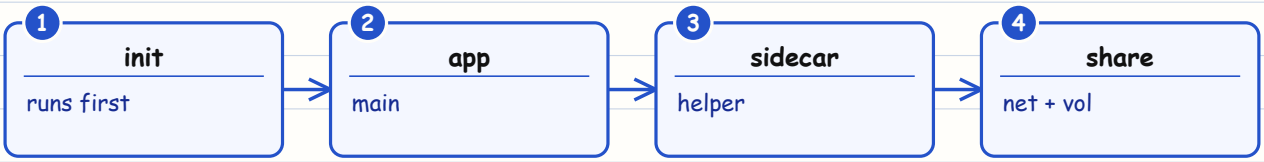
* Patterns

- Sidecar: a helper beside the app (log shipper, proxy). Init container: runs to completion first.
- Ambassador: a local proxy for external services. All containers in a Pod share network + volumes.

* Do Not Overuse

- Do not co-locate unrelated microservices in one Pod - they cannot scale or release independently.
- Use separate Deployments + a Service for communication between distinct apps.

Multi-container Pod



REAL-WORLD EXAMPLE

```
# init container waits for the DB, then the app starts
initContainers:
- name: wait-db
  image: busybox
  command: ["sh","-c","until nc -z db 5432; do sleep 2; done"]
```

COMMON MISTAKE
Cramming app + database + cron into one Pod. You lose independent scaling, restarts and clean logs. One primary concern per Pod; add sidecars only for a clear supporting purpose.

REVISION SNAPSHOT

ASK YOURSELF	When is a sidecar appropriate vs a separate Deployment?
DO THIS	Add an init container that waits for a dependency before the app starts.

11. ReplicaSet & Deployment

GrowthSchool

IN SIMPLE TERMS
A Deployment is a manager that keeps the right number of identical Pods running and updates them safely; a ReplicaSet is the piece that maintains the count.

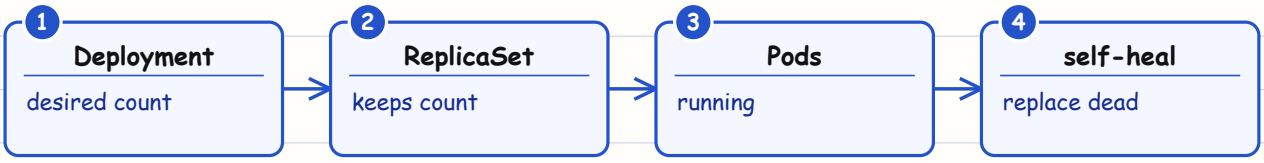
* ReplicaSet

- Keeps a specified number of matching Pod replicas alive; recreates any that die.
- You rarely create these directly - Deployments manage them for you.

* Deployment

- Declaratively manages stateless apps: rolling updates, pause/resume and revision history for rollback.
- Use it for web APIs, frontends and workers whose replicas are interchangeable.

Deployment manages Pods



REAL-WORLD EXAMPLE

```
kubectl create deploy shop-api --image=shop-api:1.0 --replicas=3
kubectl get rs,deploy
kubectl scale deploy/shop-api --replicas=5
kubectl rollout history deploy/shop-api
```

COMMON MISTAKE
Editing Pods created by a Deployment directly. The controller reverts your change on the next reconcile. Change the Deployment template instead - it owns the Pods.

REVISION SNAPSHOT

ASK YOURSELF	What does a Deployment add on top of a ReplicaSet?
DO THIS	Create a 3-replica Deployment, scale it, and view its rollout history.

12. Deployment YAML Deep Dive

GrowthSchool

IN SIMPLE TERMS
The Deployment YAML is the detailed recipe telling Kubernetes which image to run, how many copies, and how to update them.

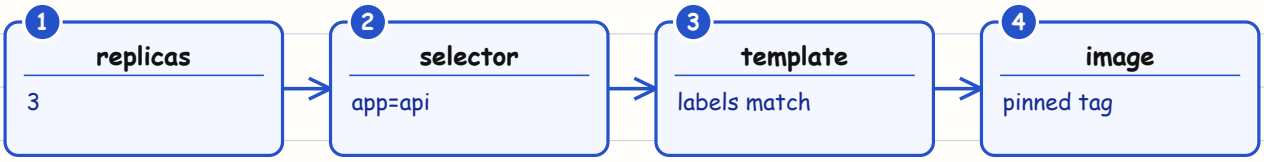
* Selectors Must Match

- spec.selector.matchLabels must equal template.metadata.labels or the Deployment cannot own its Pods.
- Set replicas for a baseline; let HPA adjust when metrics require.

* Safe Defaults

- Use immutable image tags, resource requests/limits, readiness probes and a rolling update strategy.
- Configure imagePullPolicy and pull secrets correctly for private images.

Deployment spec parts



REAL-WORLD EXAMPLE

```
spec:
  replicas: 3
  selector:
    matchLabels: { app: shop-api }
  template:
    metadata: { labels: { app: shop-api } }
    spec:
      containers: [{ name: api, image: shop-api:1.4.2 }]
```

COMMON MISTAKE
A selector that does not match the template labels. The Deployment then adopts no Pods (or the
API rejects it). Keep selector.matchLabels and template labels identical.

REVISION SNAPSHOT

ASK YOURSELF	Why must selector.matchLabels match the template labels exactly?
DO THIS	Write a Deployment spec and deliberately mismatch labels to see the error.

13. Rolling Update & Rollback

GrowthSchool

IN SIMPLE TERMS
A rolling update replaces old Pods with new ones gradually so users see no downtime; a rollback returns to the previous working version.

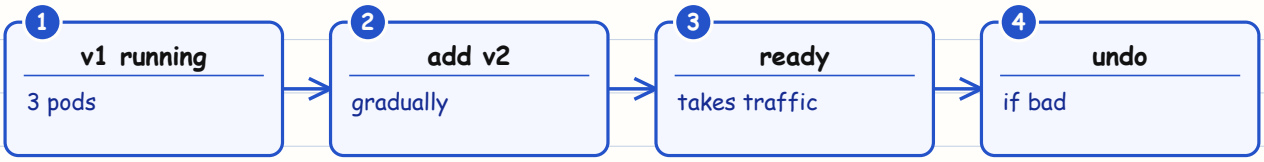
* Rolling Update

- New Pods are added and old ones removed gradually; readiness gates when new Pods take traffic.
- maxSurge is extra temporary capacity; maxUnavailable is how many may be down during the update.

* Rollback

- If a release is unhealthy, inspect events + logs, then roll back to a known-good revision fast.
- Rollback is quick only if the old image and config still exist.

Rolling update



REAL-WORLD EXAMPLE

```
kubectl set image deploy/shop-api api=shop-api:1.5.0
kubectl rollout status deploy/shop-api
kubectl rollout undo deploy/shop-api
kubectl rollout undo deploy/shop-api --to-revision=3
```

COMMON MISTAKE
Declaring a release "done" before rollout status completes. Watch it finish and check readiness and error rates before moving on - a half-rolled release can serve broken Pods.

REVISION SNAPSHOT

ASK YOURSELF	What do maxSurge and maxUnavailable control?
DO THIS	Roll a Deployment to a new tag, watch status, then roll it back.

14. Pod Lifecycle & Phases

GrowthSchool

IN SIMPLE TERMS
A Pod lifecycle is the set of stages a Pod passes through - from waiting to be placed, to running, to finishing or failing.

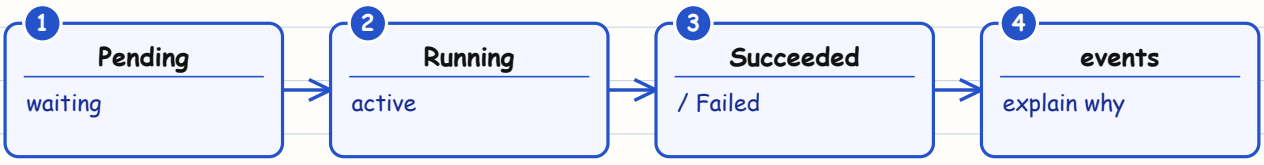
* Phases

- Pending: accepted, waiting to schedule or pull. Running: bound to a node, containers running.
- Succeeded/Failed: containers finished. Unknown: the node cannot report status.

* Restarts

- restartPolicy (Always for Deployments) plus backoff governs container restarts.
- CrashLoopBackOff is Kubernetes backing off between repeated crash restarts - read the logs.

Pod phases



REAL-WORLD EXAMPLE

```
kubectl get pods
kubectl describe pod <pod>     # Events explain Pending
kubectl logs <pod> --previous  # last crashed container
kubectl get events --sort-by=.lastTimestamp
```

COMMON MISTAKE
Deleting a CrashLoopBackOff Pod over and over. The controller recreates the same broken config.
Read logs + events and fix the root cause instead.

REVISION SNAPSHOT

ASK YOURSELF	What does CrashLoopBackOff actually indicate?
DO THIS	Force a Pending Pod (huge request) and diagnose it from Events.

15. Probes

GrowthSchool

IN SIMPLE TERMS
A probe is a small health check Kubernetes runs on your container to decide if it is alive and ready to receive traffic.

* Three Probes

- Liveness: is it alive? Failure restarts the container. Readiness: can it serve now? Failure removes it from endpoints.
- Startup: protects slow-booting apps so liveness/readiness only begin after startup succeeds.

* Use Them Right

- Readiness prevents traffic hitting unready Pods (during warm-up, dependency waits or draining).
- A too-aggressive liveness probe causes needless restart loops - give apps time to start.

Three Probes Guard Availability

REAL-WORLD EXAMPLE

```
readinessProbe:
  httpGet: { path: /ready, port: 8080 }
  initialDelaySeconds: 5
livenessProbe:
  httpGet: { path: /health, port: 8080 }
  periodSeconds: 10
```

COMMON MISTAKE
Pointing liveness at a deep dependency (e.g. the database). If the DB blips, Kubernetes kills healthy app Pods. Keep liveness shallow; use readiness for dependency checks.

REVISION SNAPSHOT

ASK YOURSELF	What is the difference in effect between liveness and readiness failure?
DO THIS	Add readiness + liveness probes to the Shop API and watch endpoints update.

16. Resources, Limits & QoS

GrowthSchool

IN SIMPLE TERMS
Requests are the resources a Pod is guaranteed; limits are the most it may use. QoS is how Kubernetes ranks Pods when resources run short.

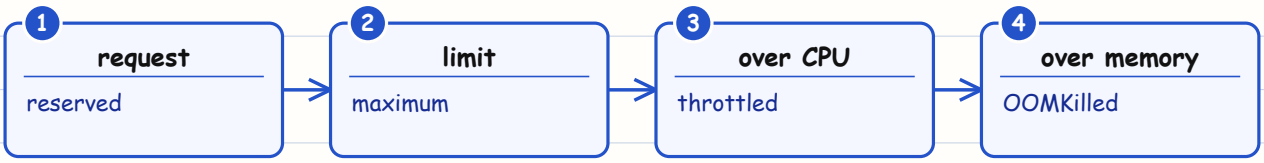
* Requests vs Limits

- A request is what the scheduler reserves; a limit caps usage. CPU over-limit throttles; memory over-limit OOM-kills.
- Requests decide placement; without them, a busy Pod can land on an already-crowded node.

* QoS Classes

- Guaranteed (requests==limits) > Burstable (some set) > BestEffort (none). Higher QoS survives pressure better.

Requests vs limits



REAL-WORLD EXAMPLE

```
resources:
  requests: { cpu: 500m, memory: 768Mi }
  limits:   { cpu: "1",  memory: 1Gi }
kubectl top pods    # needs metrics-server
```

COMMON MISTAKE
Setting a memory limit below real usage, causing constant OOMKills and restarts. Measure normal
and peak usage first, then set requests near normal and limits with headroom.

REVISION SNAPSHOT

ASK YOURSELF	What happens when a Pod exceeds its CPU limit vs its memory limit?
DO THIS	Set requests/limits on the Shop API and read its QoS class.

17. Autoscaling

GrowthSchool

IN SIMPLE TERMS
Autoscaling means Kubernetes automatically adds or removes Pods (and even machines) as demand rises and falls.

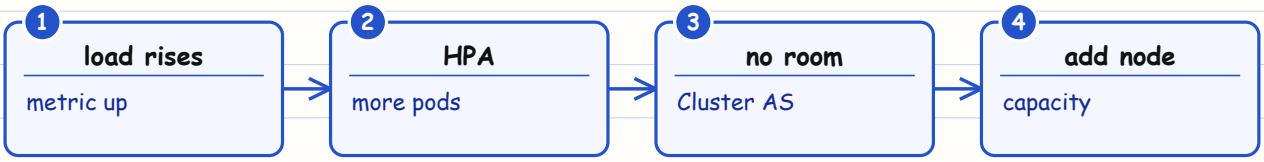
* Horizontal Pod Autoscaler

- HPA changes replica count from CPU/memory/custom metrics; it needs metrics-server for basic metrics.
- It works best when replicas are stateless and each has a meaningful resource request.

* Cluster Autoscaler

- Adds nodes when Pods cannot schedule and removes underused nodes when safe.
- It cannot fix impossible requests or bad node selectors - only capacity shortages.

Two kinds of scaling



REAL-WORLD EXAMPLE

```
kubectl autoscale deploy/shop-api --cpu-percent=60 --min=3 --max=20
kubectl get hpa
kubectl describe hpa shop-api
# HPA scales Pods; Cluster Autoscaler scales Nodes
```

COMMON MISTAKE
Expecting HPA to help a Pod with no resource requests. With no request, CPU utilisation is undefined and HPA cannot decide. Always set requests before enabling HPA.

REVISION SNAPSHOT

ASK YOURSELF	What does HPA scale vs what does the Cluster Autoscaler scale?
DO THIS	Add an HPA to the Shop API and load-test to watch it scale.

18. Services

GrowthSchool

IN SIMPLE TERMS
A Service gives a stable address and name to a group of Pods, so other apps can reach them even as Pods come and go.

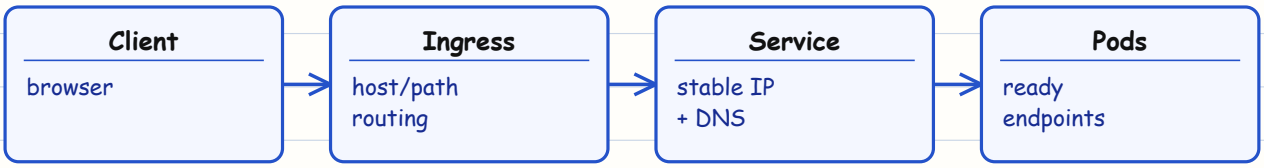
* Why Services

- Pods are ephemeral; a Service gives a stable virtual IP and DNS name to a changing set of Pods.
- A label selector picks the target Pods; traffic is load-balanced across ready endpoints.

* Service Types

- ClusterIP (internal, default), NodePort (a port on every node), LoadBalancer (cloud LB),
- ExternalName (DNS alias). Most in-cluster traffic uses ClusterIP.

Traffic Path: Ingress -> Service -> ready Pods



REAL-WORLD EXAMPLE

```
kubectl expose deploy/shop-api --port=80 --target-port=8080
kubectl get svc,endpointslices
kubectl port-forward svc/shop-api 8080:80
curl http://localhost:8080/health
```

COMMON MISTAKE
Hard-coding a Pod IP in a client instead of using the Service name. Pod IPs change; the Service name/IP is stable. Always talk to Services.

REVISION SNAPSHOT

ASK YOURSELF	What stable things does a Service provide over raw Pods?
DO THIS	Expose the Shop API with a Service and reach it via port-forward.

19. DNS & Service Discovery

GrowthSchool

IN SIMPLE TERMS
Cluster DNS lets Pods find each other by an easy name instead of by ever-changing IP addresses.

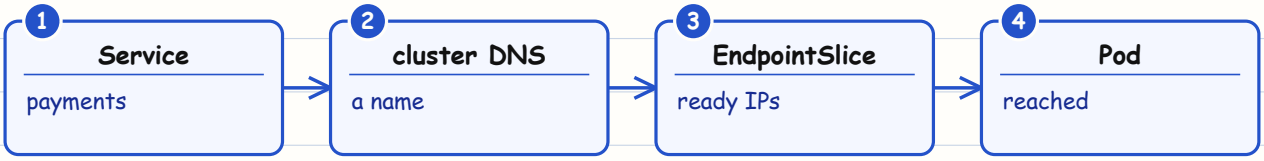
* Cluster DNS

- A Service payments in namespace shop resolves as payments, payments.shop, or payments.shop.svc.cluster.local.
- Apps should use these DNS names, never Pod IPs.

* Endpoints

- A selector builds EndpointSlices of ready Pod IPs. No endpoints usually means a bad selector or failed readiness.
- A headless Service (clusterIP: None) returns Pod addresses directly - common with StatefulSets.

Service discovery



REAL-WORLD EXAMPLE

```
kubectl get svc,endpoints,endpointslices
kubectl run netcheck --rm -it --image=busybox -- sh
# inside: nslookup shop-api.shop.svc.cluster.local
# inside: wget -qO- http://shop-api.shop/health
```

COMMON MISTAKE
Debugging "connection refused" without checking endpoints. If a Service has no endpoints, the selector or readiness is wrong - fix that before blaming the network.

REVISION SNAPSHOT

ASK YOURSELF	What does an empty EndpointSlice usually tell you?
DO THIS	From a debug Pod, resolve a Service by DNS and curl it.

20. Ingress & Gateway

GrowthSchool

IN SIMPLE TERMS
An Ingress is a smart HTTP entry point that routes web requests from the outside world to the right Service by hostname and URL path.

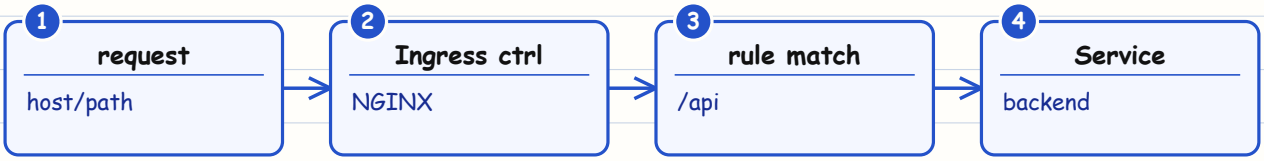
* Ingress

- Ingress routes external HTTP/HTTPS by host and path to Services. It needs an Ingress Controller (NGINX, Traefik).
- Example: shop.example.com/api -> shop-api Service; / -> frontend Service.

* TLS & Gateway API

- Terminate HTTPS at the Ingress with an automatic certificate manager; renew certs automatically.
- Gateway API is the newer, more expressive successor - learn it after mastering Ingress.

Ingress routing



REAL-WORLD EXAMPLE

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

COMMON MISTAKE
Creating an Ingress resource with no Ingress Controller installed. Nothing routes traffic and you get confusing 404s. Install/confirm a controller first.

REVISION SNAPSHOT

ASK YOURSELF	What extra component must exist for Ingress to work?
DO THIS	Route two paths to two Services with a single Ingress resource.

21. ConfigMaps & Secrets

GrowthSchool

IN SIMPLE TERMS
A ConfigMap stores plain settings and a Secret stores sensitive values, so configuration lives outside your container image.

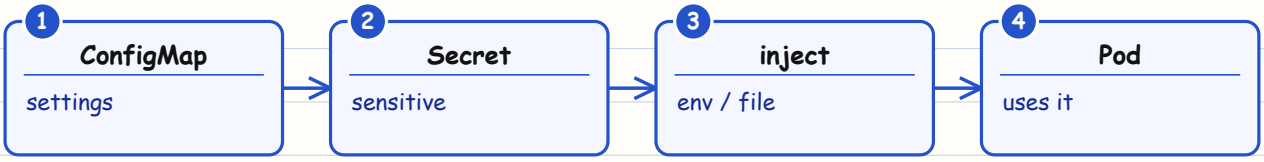
* ConfigMap

- Stores non-sensitive config: URLs, flags, log levels, config files. Inject as env vars or mounted files.
- Reload behaviour depends on the app - env changes usually need a Pod restart.

* Secret

- Stores sensitive values. base64 is encoding, not encryption. Protect with RBAC + encryption at rest.
- Avoid printing secrets in logs; prefer an external secrets manager for production.

Config vs Secret



REAL-WORLD EXAMPLE

```
kubectl create configmap shop-config --from-literal=MODE=production
kubectl create secret generic db-creds --from-literal=password=CHANGE-ME
# mount as env: envFrom: [{ configMapRef: { name: shop-config } }]
```

COMMON MISTAKE
Treating a Secret as secure just because it is base64. Anyone with get access can decode it. Lock down RBAC, enable encryption at rest, and keep Secrets out of Git.

REVISION SNAPSHOT

ASK YOURSELF	Why is base64 in a Secret not a security control?
DO THIS	Inject a ConfigMap as env and a Secret as a mounted file, then verify.

22. Volumes, PV, PVC & StorageClass

GrowthSchool

IN SIMPLE TERMS
A Volume gives a Pod storage; a PersistentVolumeClaim requests durable storage that survives Pod restarts, provisioned through a StorageClass.

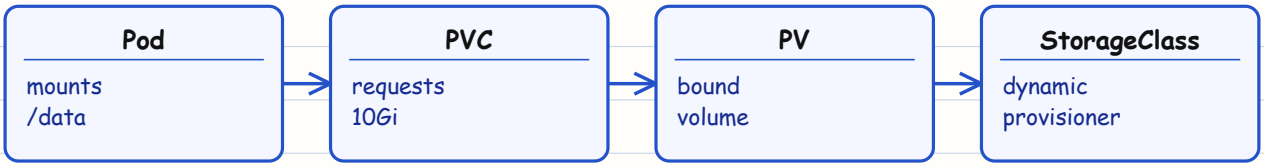
* The Chain

- A Volume attaches storage to a Pod. A PersistentVolume (PV) is a cluster storage resource.
- A PersistentVolumeClaim (PVC) requests storage; a StorageClass provisions PVs dynamically.

* Design Choice

- Keep state out of the container layer. Pick a StorageClass by latency, durability and access mode.
- ReadWriteOnce mounts to one node; some classes support ReadWriteMany.

Storage: Pod -> PVC -> PV (via StorageClass)



REAL-WORLD EXAMPLE

```
kubectl get sc,pv,pvc
# PVC requests 10Gi; a StorageClass provisions a PV
kubectl describe pvc shop-data
# mount: volumeMounts + volumes.persistentVolumeClaim
```

COMMON MISTAKE
Storing database files in the Pod filesystem. On reschedule the data is gone. Bind a PVC so data survives Pod restarts and rescheduling.

REVISION SNAPSHOT

ASK YOURSELF	What is the difference between a PV and a PVC?
DO THIS	Create a PVC, mount it in a Pod, recreate the Pod, and verify data persists.

23. StatefulSets

GrowthSchool

IN SIMPLE TERMS
A StatefulSet runs apps that need a stable identity and their own storage, like databases, giving each Pod a fixed name and disk.

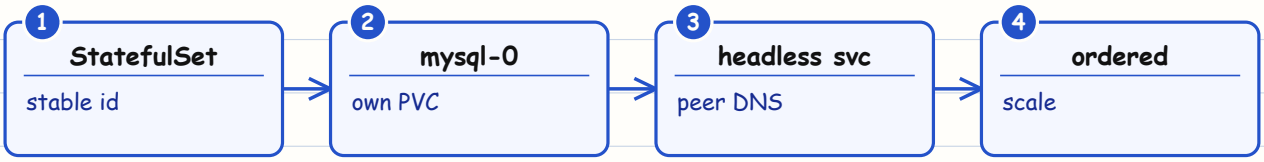
* When It Fits

- For databases, queues and clustered systems needing stable identities (mysql-0, mysql-1) and per-Pod storage.
- Each replica can get its own PVC; scaling and deletion follow a controlled order.

* Reality Check

- Kubernetes runs a database but does not remove its operational burden: backups, replication, upgrades, restore tests.
- For many teams, a managed database is the better default unless you have strong reasons to self-host.

StatefulSet identity



REAL-WORLD EXAMPLE

```
kubectl get statefulset,pvc
# pair with a headless Service for stable peer DNS:
# clusterIP: None -> mysql-0.mysql, mysql-1.mysql
kubectl delete pod mysql-0   # recreated with same name + PVC
```

COMMON MISTAKE
Using a Deployment for a stateful database expecting stable names/storage. Deployments give interchangeable Pods. Use a StatefulSet + headless Service for stable identity.

REVISION SNAPSHOT

ASK YOURSELF	What does a StatefulSet guarantee that a Deployment does not?
DO THIS	Deploy a small StatefulSet and observe stable Pod names and PVCs.

24. DaemonSets

GrowthSchool

IN SIMPLE TERMS
A DaemonSet makes sure one copy of a Pod runs on every node - handy for agents like log or monitoring collectors.

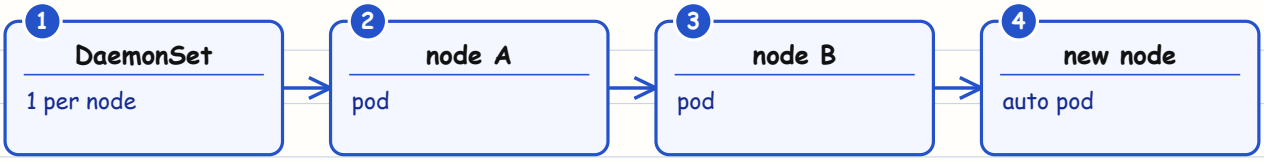
* One Pod Per Node

- A DaemonSet runs a copy of a Pod on every (or selected) node - ideal for node-level agents.
- Used for log collectors, monitoring agents and CNI/network components.

* Scheduling

- As nodes join, the DaemonSet adds a Pod; as nodes leave, its Pod is removed.
- Use node selectors/tolerations to target specific nodes (e.g. only GPU nodes).

DaemonSet coverage



REAL-WORLD EXAMPLE

```
kubectl get daemonset -A
# e.g. a log shipper on every node
kubectl describe ds -n logging fluent-bit
kubectl get pods -o wide -l app=fluent-bit
```

COMMON MISTAKE
Using a Deployment with nodeAffinity to fake "one per node". A DaemonSet is the correct, self-
maintaining primitive for per-node agents.

REVISION SNAPSHOT

ASK YOURSELF	When do you choose a DaemonSet over a Deployment?
DO THIS	Identify the DaemonSets in kube-system and explain what each does.

25. Jobs & CronJobs

GrowthSchool

IN SIMPLE TERMS
A Job runs a task once until it finishes; a CronJob runs Jobs automatically on a schedule.

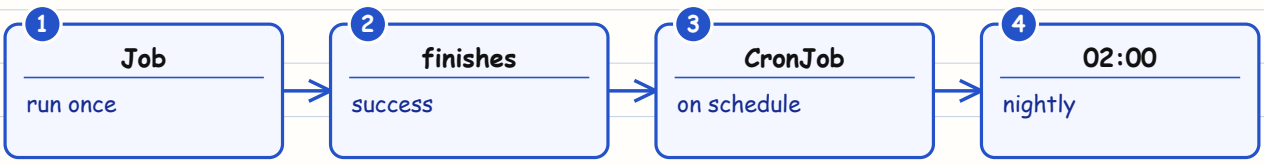
* Job

- Runs one-off work to successful completion: a migration, report or batch import.
- Use backoffLimit and activeDeadlineSeconds so failures do not retry forever.

* CronJob

- Creates Jobs on a schedule (cron syntax) - great for nightly exports and cleanup.
- Set concurrencyPolicy to avoid overlapping runs when a previous Job is slow.

Jobs and CronJobs



REAL-WORLD EXAMPLE

```
schedule: "0 2 * * *"          # daily 02:00
concurrencyPolicy: Forbid
successfulJobsHistoryLimit: 3
kubectl create job --from=cronjob/backup manual-backup
```

COMMON MISTAKE
Letting a CronJob overlap itself because a run is slow, doubling load or corrupting data. Set concurrencyPolicy: Forbid (or Replace) deliberately.

REVISION SNAPSHOT

ASK YOURSELF	How do you stop CronJob runs from overlapping?
DO THIS	Schedule a daily backup Job and trigger a manual run from it.

26. RBAC & ServiceAccounts

GrowthSchool

IN SIMPLE TERMS
RBAC controls who is allowed to do what in the cluster; a ServiceAccount is the identity an app uses to talk to the Kubernetes API.

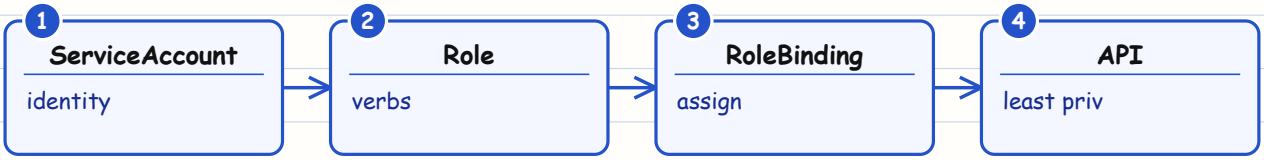
* Least Privilege

- A Role grants verbs on namespaced resources; a ClusterRole covers cluster-wide ones.
- A RoleBinding/ClusterRoleBinding assigns those to a user, group or ServiceAccount.

* Pod Identity

- Give each app its own ServiceAccount; disable token automount if it needs no API access.
- Never grant applications cluster-admin. Start read-only on exactly the resources needed.

Least-privilege RBAC



REAL-WORLD EXAMPLE

```
kubectl create serviceaccount shop-api -n shop
kubectl create role pod-reader --verb=get,list --resource=pods -n shop
kubectl create rolebinding shop-api-read \
  --role=pod-reader --serviceaccount=shop:shop-api -n shop
kubectl auth can-i list pods --as=system:serviceaccount:shop:shop-api -n shop
```

COMMON MISTAKE
Binding cluster-admin to an app "to make it work". A compromise then owns the cluster. Grant the
minimum verbs/resources/namespace and expand only when proven necessary.

REVISION SNAPSHOT

ASK YOURSELF	How do you test what a ServiceAccount is allowed to do?
DO THIS	Create an SA + Role that can only list Pods in one namespace and verify.

27. Network Policies

GrowthSchool

IN SIMPLE TERMS
A NetworkPolicy is a firewall for Pods - it decides which Pods are allowed to talk to which.

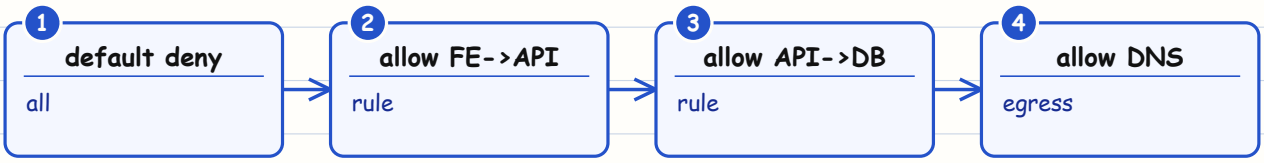
* Default Is Open

- Without policies, most CNIs allow all Pod-to-Pod traffic. A NetworkPolicy defines allowed ingress/egress.
- Enforcement needs a CNI that supports NetworkPolicy (Calico, Cilium, etc.).

* Zero-Trust Pattern

- Start with default-deny in a namespace, then allow only: frontend->api, api->db, and DNS egress.
- Select traffic by labels, not IPs - a missing DNS rule can break all name lookups.

Zero-trust network



REAL-WORLD EXAMPLE

```
# default deny all ingress in a namespace
spec:
  podSelector: {}
  policyTypes: [Ingress]
# then add the smallest allow rules needed
```

COMMON MISTAKE
Applying a default-deny egress policy but forgetting to allow DNS (port 53). Every outbound hostname lookup then fails. Always allow DNS egress explicitly.

REVISION SNAPSHOT

ASK YOURSELF	What is the safe order: deny first or allow first?
DO THIS	Apply default-deny, then add one rule allowing frontend to reach the API.

28. Pod Security & Image Safety

GrowthSchool

IN SIMPLE TERMS
Pod security and image safety are the settings and habits that stop containers running as root or shipping with known vulnerabilities.

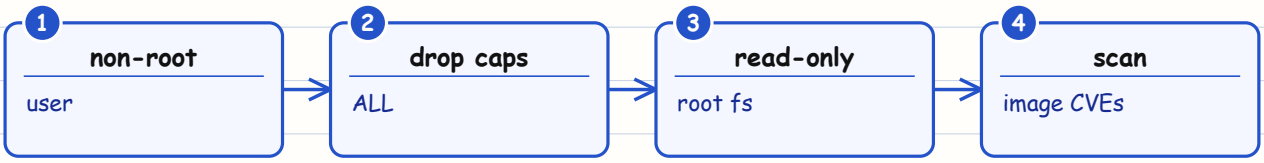
* Secure Runtime

- Run as non-root, disallow privilege escalation, drop Linux capabilities, use a read-only root FS.
- Avoid privileged containers and hostPath unless formally reviewed.

* Secure Supply Chain

- Use minimal trusted base images, scan for CVEs, sign images, and pin tags/digests.
- Never bake secrets or cloud keys into an image - anyone with the image can extract them.

Harden the Pod



REAL-WORLD EXAMPLE

```
securityContext:
  runAsNonRoot: true
  allowPrivilegeEscalation: false
  capabilities: { drop: ["ALL"] }
  readOnlyRootFilesystem: true
```

COMMON MISTAKE
Running every container as root by default. If the app is exploited, root in the container is a much bigger problem. Set runAsNonRoot and drop capabilities.

REVISION SNAPSHOT

ASK YOURSELF	Name four securityContext settings that harden a Pod.
DO THIS	Add a hardened securityContext to the Shop API and confirm it still runs.

29. Observability

GrowthSchool

IN SIMPLE TERMS
Observability means being able to see what your system is doing through its logs, metrics, and traces.

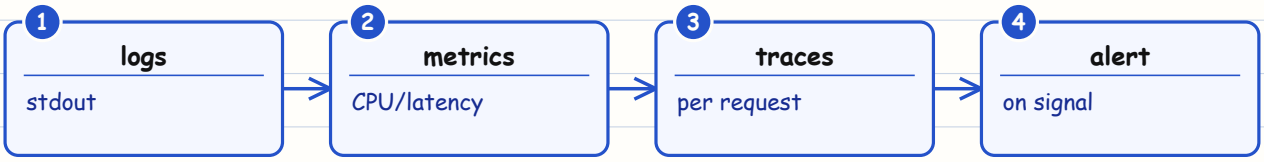
* Logs, Metrics, Traces

- Logs: write to stdout/stderr; a node collector ships them centrally. Use structured JSON + request IDs.
- Metrics: CPU, latency, error rate, saturation over time (Prometheus-style). Traces: one request across services.

* The Point

- You cannot operate what you cannot see. Every production change needs an observable success signal.

Three signals



REAL-WORLD EXAMPLE

```
kubectl logs -f deploy/shop-api
kubectl top pods    # needs metrics-server
kubectl get events --sort-by=.lastTimestamp
# dashboards + alerts on rate, errors, latency (RED method)
```

COMMON MISTAKE
Logging to a file inside the container. On restart the logs vanish and no collector sees them. Log to stdout/stderr so the platform captures everything centrally.

REVISION SNAPSHOT

ASK YOURSELF	What are the three pillars of observability?
DO THIS	Tail a Deployment's logs and read cluster Events during a rollout.

30. Troubleshooting Playbook

GrowthSchool

IN SIMPLE TERMS
Troubleshooting is the step-by-step way to find why a Pod is not working, using describe, logs, and events.

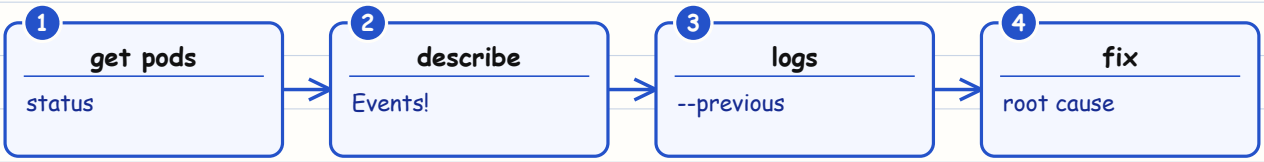
* Symptom -> Cause

- CrashLoopBackOff: read logs (+ --previous), check command/env/missing Secret/exit code/liveness.
- Pending: describe events for insufficient CPU/mem, taints, node selector, PVC binding or quota.

* More

- ImagePullBackOff: wrong image/tag, unreachable registry, missing imagePullSecrets.
- OOMKilled: memory limit too low or a leak - check usage and raise limits or fix the leak.

Debug a Pod



REAL-WORLD EXAMPLE

```
kubectl get pods -A
kubectl describe pod <pod>       # Events first!
kubectl logs <pod> --previous
kubectl get events --sort-by=.lastTimestamp
kubectl top pod <pod>
```

COMMON MISTAKE
Skipping kubectl describe and its Events. The answer to Pending, ImagePull and scheduling problems is almost always right there in the Events list.

REVISION SNAPSHOT

ASK YOURSELF	What is the first command you run for a misbehaving Pod?
DO THIS	Reproduce ImagePullBackOff (bad tag) and diagnose it from Events.

31. GitOps & Release Workflow

GrowthSchool

IN SIMPLE TERMS
GitOps means your cluster's desired state lives in Git, and an automated tool keeps the real cluster matching it.

* The GitOps Idea

- Store reviewed manifests (or Helm/Kustomize) in Git. CI builds + scans an image; CD reconciles the cluster.
- Git is the single source of truth; avoid imperative production changes that cannot be reproduced.

* Release Checklist

- Tests pass; image scanned; requests/limits + probes set; dashboards/alerts ready; rollback validated.
- Use progressive delivery (canary, blue/green) when the change is risky.

GitOps Release Flow (Shop API)

REAL-WORLD EXAMPLE

```
git commit -m "deploy shop-api:1.5.0"   # change manifest
# CI builds+scans image, CD applies desired state
kubectl rollout status deploy/shop-api
kubectl rollout undo deploy/shop-api     # fast rollback
```

COMMON MISTAKE
Hand-editing production with kubectl edit under pressure. It drifts from Git and cannot be reproduced. Change Git; let CD reconcile - even during incidents where possible.

REVISION SNAPSHOT

ASK YOURSELF	Why is Git the source of truth in GitOps?
DO THIS	Describe the path from a merged PR to a running change in the cluster.

32. Capstone: Deploy the Shop API

GrowthSchool

IN SIMPLE TERMS
This capstone ties everything together by deploying the full Shop API on Kubernetes, end to end.

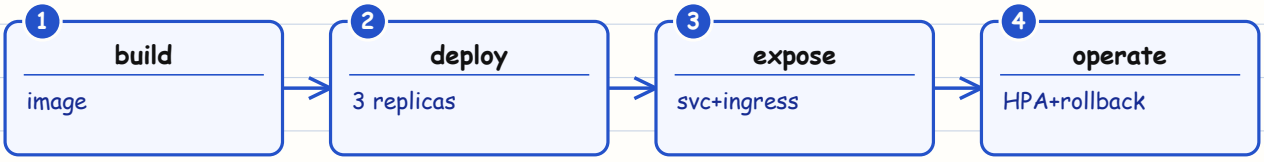
* Build & Deploy

- Containerise the API with /health and /ready endpoints and an immutable tag.
- Deploy 3 replicas with requests/limits, readiness/liveness probes and a rolling strategy.

* Expose & Secure

- Add a ClusterIP Service + Ingress host. Put config in a ConfigMap and the DB password in a Secret.
- Add an HPA on CPU and a default-deny NetworkPolicy allowing only required traffic.

Capstone steps



REAL-WORLD EXAMPLE

```
kubectl apply -f shop-api/       # deploy, svc, ingress, hpa
kubectl rollout status deploy/shop-api
kubectl get hpa,ingress,networkpolicy -n shop
kubectl port-forward svc/shop-api 8080:80
```

STUDENT LAB
Deliverables: 3 healthy replicas, working Service + Ingress, ConfigMap + Secret in use, an HPA, a NetworkPolicy, and a practised bad-release-then-rollback in a safe namespace.

REVISION SNAPSHOT

ASK YOURSELF	Does your capstone cover deploy, expose, scale, secure and rollback?
DO THIS	Complete every deliverable, then deliberately break a release and recover it.

33. kubectl Cheat Sheet

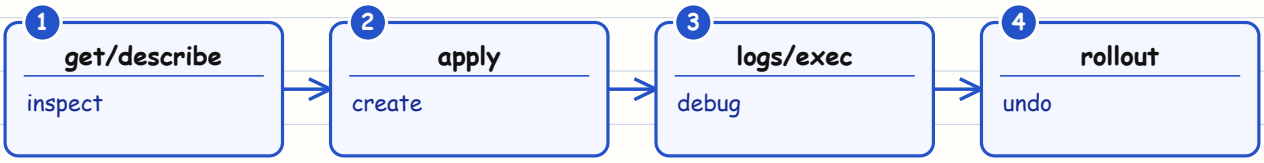
GrowthSchool

IN SIMPLE TERMS
This is a quick-reference list of the kubectl commands you will reach for every day.

* Daily Drivers

- get/describe/logs/exec, apply -f, rollout status/undo, scale, port-forward, top, get events.

Everyday kubectl



DEBUG LOOP
get -> describe (Events!) -> logs (--previous) -> exec -> events. This sequence solves the vast majority of Pod problems.

ONE-PAGE COMMAND REFERENCE

```
kubectl get pods -A              kubectl apply -f app.yaml
kubectl describe pod <p>         kubectl delete -f app.yaml
kubectl logs <p> -c <c>          kubectl rollout status deploy/x
kubectl logs <p> --previous      kubectl rollout undo deploy/x
kubectl exec -it <p> -- sh       kubectl scale deploy/x --replicas=5
kubectl get svc,endpointslices   kubectl port-forward svc/x 8080:80
kubectl top pods/nodes           kubectl get events --sort-by=.lastTimestamp
kubectl explain <kind>.spec      kubectl auth can-i <verb> <res>
```

COMMON MISTAKE
Forgetting -n / -A and looking in the wrong namespace, then concluding "it is not there". Always confirm the namespace; use -A to search everywhere.

REVISION SNAPSHOT

ASK YOURSELF	What is the standard get -> describe -> logs debug loop?
DO THIS	Recall ten kubectl commands from memory grouped by purpose.

34. Interview & Revision Notes

GrowthSchool

> **IN SIMPLE TERMS**
> This page gathers the key ideas and common interview questions so you can revise quickly.

* Explain Clearly

- Pod vs Deployment vs Service vs Ingress; PV vs PVC; ConfigMap vs Secret; requests vs limits.
- Why readiness matters, why Pod IPs are unstable, and how reconciliation works.

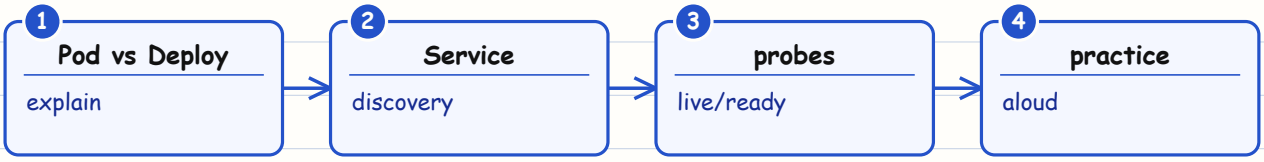
* Common Questions

- What happens when a Pod dies? How does a Service find Pods? How do you debug CrashLoopBackOff?
- How does HPA decide to scale? What is the difference between liveness and readiness?

* Your Next Step

- Build the capstone on kind/minikube, read Events daily, and learn by diagnosing real failures.

Revise out loud



> **REAL-WORLD EXAMPLE**

```
# 30-second self test
kubectl get deploy,rs,pods    # who owns whom?
kubectl describe pod <p>      # can you read the Events?
# Pod vs Deployment? PV vs PVC? liveness vs readiness?
```

> **REVISION SNAPSHOT**
>
> **ASK YOURSELF** Can you explain Pod vs Deployment vs Service in 30 seconds?
>
> **DO THIS** Answer every "Common Question" above out loud without notes.