

Kubernetes

Complete Interview Q&A

100 Questions | Unique Pastel Per Question

Simple English | Bullet Points | YAML & kubectl Commands

DevOps / SRE / Cloud Engineers — 3 to 5 Years Experience

1. Basics & Architecture	Q1 – Q10
2. Pods & Workloads	Q11 – Q22
3. Services & Networking	Q23 – Q34
4. Configuration & Storage	Q35 – Q44
5. Scheduling & Resource Management	Q45 – Q54
6. Scaling & Availability	Q55 – Q62
7. Security	Q63 – Q70
8. Observability	Q71 – Q75
9. CI/CD & Packaging (Helm)	Q76 – Q80
10. Troubleshooting	Q81 – Q90
11. Scenario-Based Questions	Q91 – Q100

Q01

Basics &
Architecture

What is Kubernetes and why is it used?

- Kubernetes (K8s) is an open-source container orchestration platform originally built by Google.
- It automates deployment, scaling, load balancing, and self-healing of containerised applications.
- Without K8s: you manually restart crashed containers, manage config on each server, scale by hand.
- With K8s: declare what you want (e.g. 3 replicas) — K8s makes it happen and keeps it that way forever.
- Works on any cloud (AWS EKS, GCP GKE, Azure AKS) or on-premises bare metal.
- Key features: automatic bin-packing, horizontal scaling, service discovery, rolling updates, secret management.

■ Example — YAML / Commands

```
# Deploy nginx with 3 replicas – one command
kubectl create deployment nginx --image=nginx:1.25 --replicas=3
# Expose it so traffic can reach it
kubectl expose deployment nginx --port=80 --type=LoadBalancer
# Watch pods start up
kubectl get pods -w
```

■ Key Insight

K8s doesn't just run containers — it watches them 24/7 and fixes anything that drifts from the desired state. That is self-healing.

Q02

Basics &
Architecture

Explain Kubernetes architecture — control plane vs worker nodes.

- Kubernetes has two layers: Control Plane (the brain) and Worker Nodes (the muscle).
- Control Plane decides what should happen. Worker Nodes do the actual container work.
- Control Plane components: kube-apiserver, etcd, kube-scheduler, kube-controller-manager.
- Worker Node components: kubelet, kube-proxy, container runtime (containerd).
- Every kubectl command you run hits the kube-apiserver on the control plane first.
- Worker nodes can keep running even if the control plane temporarily goes down.

■ Example — YAML / Commands

```
# List all nodes and their roles
kubectl get nodes -o wide
# Check control plane component health
kubectl get componentstatuses
# See all system pods (control plane components run here)
kubectl get pods -n kube-system
```

■ Key Insight

Control plane = brain. Worker nodes = hands. Brain goes down → hands keep working but can't take new instructions.

Q03

Basics &
Architecture

What are the core components of the control plane?

- kube-apiserver: the single entry point — every kubectl command, controller, and kubelet talks to it.
- etcd: distributed key-value store — holds ALL cluster state (pods, configs, secrets, RBAC rules).
- kube-scheduler: watches for new pods with no node assigned, picks the best node for them.
- kube-controller-manager: runs all control loops (ReplicaSet, Node, Job, Endpoint controllers).
- cloud-controller-manager: optional, integrates with cloud APIs (AWS, GCP, Azure) for LBs and disks.
- All components talk to etcd only through the kube-apiserver — never directly.

■ Example — YAML / Commands

```
# Check control plane pods
kubectl get pods -n kube-system | grep -E 'api|etcd|scheduler|controller'
# Describe API server pod
kubectl describe pod kube-apiserver-<node> -n kube-system
# Check scheduler logs
kubectl logs -n kube-system kube-scheduler-<node> --tail=50
```

■ Key Insight

Lose kube-apiserver → kubectl stops. Lose etcd without backup → lose ALL cluster state permanently.
Protect both.

Q04

Basics &
Architecture

What is kube-apiserver?

- kube-apiserver is the ONLY entry point to the cluster — everything goes through it.
- Exposes the Kubernetes REST API over HTTPS.
- Three gates every request passes: Authentication (who are you?), Authorization (RBAC check), Admission (policy check).
- After passing all gates, the request is written to etcd (the cluster's database).
- It is stateless — all state lives in etcd, so you can run multiple API server replicas for HA.
- Supports watch API — controllers and kubectl can subscribe to changes and get notified instantly.

■ Example — YAML / Commands

```
# Every kubectl command is just an HTTP call to the API server
kubectl get pods
# = GET https://<api-server>/api/v1/namespaces/default/pods
# Talk to API server directly with raw HTTP
kubectl get --raw /api/v1/namespaces
# List all available API resources
kubectl api-resources
kubectl api-versions
```

■ Key Insight

Protect the API server: private endpoint, IP allowlist, audit logging, disable anonymous auth. It is the crown jewel.

Q05

Basics &
Architecture

What is etcd and how does it store data?

- etcd is a distributed, consistent key-value store — it is the single source of truth for the cluster.
- Stores everything: pod specs, deployments, secrets, ConfigMaps, RBAC rules, service info.
- Uses Raft consensus algorithm — requires majority quorum for any write (3 nodes = tolerates 1 failure).
- Data is stored under `/registry/*` key prefix — e.g. `/registry/pods/default/my-pod`.
- Can get corrupted if disk is full, power loss during write, or storage quota exceeded (default 2 GB).
- ALWAYS take regular backups and TEST your restore — an untested backup is not a backup.

■ Example — YAML / Commands

```
# Backup etcd snapshot
ETCDCTL_API=3 etcdctl snapshot save /backup/etcd-$(date +%F).db \
--endpoints=https://127.0.0.1:2379 \
--cacert=/etc/etcd/ca.crt \
--cert=/etc/etcd/server.crt \
--key=/etc/etcd/server.key
# Verify backup integrity
ETCDCTL_API=3 etcdctl snapshot status /backup/etcd-$(date +%F).db
# Check etcd cluster health
ETCDCTL_API=3 etcdctl endpoint health
```

■ Key Insight

Lose etcd without a backup = cluster state is gone forever. Back up etcd daily and test restores monthly.
Non-negotiable.

Q06

Basics &
Architecture

What is kube-scheduler and how does it work?

- kube-scheduler watches for newly created pods that have no node assigned yet.
- Runs two phases for every unscheduled pod: Filtering then Scoring.
- Filtering: eliminates nodes that fail hard constraints — not enough CPU/memory, wrong taints, wrong affinity.
- Scoring: ranks remaining nodes by priority functions — least requested resources, image locality, spreading.
- Selects the highest scoring node and writes nodeName into the pod spec via kube-apiserver.
- kubelet on that node picks up the pod and starts the containers.

■ Example — YAML / Commands

```
# See why a pod is not scheduled
kubectl describe pod <pending-pod> | grep -A10 Events
# View scheduler logs
kubectl logs -n kube-system kube-scheduler-<node>
# Force a pod to a specific node (bypass scheduler)
spec:
  nodeName: worker-node-1 # hardcoded — use only for debugging
```

■ Key Insight

Scheduler only places new pods. It never moves running pods. Use drain + uncordon to force rescheduling if needed.

Q07

Basics &
Architecture

What is kube-controller-manager?

- Runs multiple controller loops in a single process — each controller watches one resource type.
- All controllers follow the same reconcile loop: observe current state → compare to desired → act to fix diff.
- ReplicaSet Controller: ensures correct pod replica count at all times.
- Node Controller: monitors node heartbeats, marks NotReady after timeout, evicts pods.
- Job Controller: creates pods for Jobs and tracks successful completions.
- Endpoint Controller: keeps the Endpoints object updated as pods start and stop.
- Deployment Controller: manages ReplicaSets, handles rolling updates and revision history.

■ Example — YAML / Commands

```
# See controller manager running in kube-system
kubectl get pods -n kube-system | grep controller-manager

# Check controller manager logs
kubectl logs -n kube-system kube-controller-manager-<node> --tail=50

# Watch ReplicaSet react to a deleted pod
kubectl delete pod <pod-name>
kubectl get pods -w # watch new pod appear immediately
```

■ Key Insight

Every controller is a forever-running reconcile loop. This is what makes Kubernetes self-healing without any human help.

Q08

Basics &
Architecture

What is kubelet?

- kubelet is the node agent — runs on EVERY worker node as a system service.
- Watches kube-apiserver for pods assigned to its node.
- Calls the container runtime (containerd) to start, stop, and manage containers.
- Runs liveness probes (restart on failure) and readiness probes (remove from service on failure).
- Reports node capacity, node status, and pod status back to the control plane every few seconds.
- If kubelet dies, the node goes NotReady and the Node Controller eventually evicts pods to healthy nodes.

■ Example — YAML / Commands

```
# Check kubelet service status on a node
systemctl status kubelet
# Follow kubelet logs live
journalctl -u kubelet -f
# Get last 100 kubelet log lines
journalctl -u kubelet -n 100 --no-pager
# Check kubelet config
cat /var/lib/kubelet/config.yaml
```

■ Key Insight

kubelet is the bridge between the cluster brain and the containers on the node. If kubelet dies, the node goes dark.

Q09

Basics &
Architecture

What is kube-proxy?

- kube-proxy runs on every node as a DaemonSet — it makes Services work at the network level.
- Watches kube-apiserver for Service and Endpoint changes, then updates kernel rules.
- iptables mode (default): creates PREROUTING/OUTPUT iptables rules that DNAT ClusterIP to a pod IP.
- IPVS mode: uses Linux Virtual Server — O(1) lookup, more LB algorithms, better above 1000 services.
- kube-proxy handles IP routing only. DNS resolution is handled by CoreDNS — a separate component.
- If kube-proxy crashes, new services stop routing but existing open connections often continue.

■ Example — YAML / Commands

```
# Verify kube-proxy is running
kubectl get pods -n kube-system | grep kube-proxy

# Check current kube-proxy mode
kubectl get configmap kube-proxy -n kube-system -o yaml | grep mode

# View iptables rules for a service (on node)
iptables -t nat -L KUBE-SERVICES -n

# View IPVS rules (if IPVS mode enabled)
ipvsadm -Ln
```

■ Key Insight

kube-proxy handles IP routing for Services. Network issues with Services? Always check kube-proxy is healthy first.

Q10

Basics &
Architecture

What happens if the control plane node goes down?

- Running workloads (pods) continue working — kubelet and kube-proxy on each worker keep running.
- kubectl stops working immediately — no API server to receive commands.
- No new pods can be scheduled — scheduler is gone.
- Self-healing stops — controllers cannot replace crashed pods.
- Service endpoint updates stop — new pods not added to load balancing.
- Solution: run 3 control plane nodes in HA mode so one failure has zero impact.

■ Example — YAML / Commands

```
# In EKS/GKE: control plane is managed HA — focus on worker node HA instead
# For kubeadm HA: join a second control plane node
kubeadm join <vip>:6443 --token <tok> \
--discovery-token-ca-cert-hash sha256:<hash> \
--control-plane
# Check etcd cluster members
ETCDCTL_API=3 etcdctl member list
# Check all control plane nodes
kubectl get nodes | grep master
```

■ Key Insight

In production: always run 3 control plane nodes. In EKS/GKE, the cloud manages HA for you automatically.

Q11

Pods &
Workloads

What is a Pod?

- A Pod is the smallest deployable unit in Kubernetes — K8s never manages containers directly.
- A Pod wraps one or more containers that share the same network namespace (same IP) and storage volumes.
- Containers inside a Pod talk to each other over localhost — they share one IP address.
- Pods are ephemeral — they can be killed and replaced at any time. Never treat pod IPs as stable.
- A pod that is killed is NOT restarted in place — a new pod with a new IP is created by the controller.
- Always use a Deployment or StatefulSet to manage pods — never create bare pods in production.

■ Example — YAML / Commands

```
# Minimal pod YAML
apiVersion: v1
kind: Pod
metadata:
  name: my-app
spec:
  containers:
  - name: app
    image: nginx:1.25
    ports:
    - containerPort: 80
  resources:
    requests:
      cpu: 100m
      memory: 128Mi
kubectl apply -f pod.yaml
kubectl get pod my-app
```

■ Key Insight

Pods are cattle, not pets. Design apps so any pod can die and be replaced without data loss or downtime.

Q12

Pods &
Workloads

Can a Pod have multiple containers? When and why?

- Yes — a pod can run multiple containers that share network and storage.
- All containers in the pod share one IP. They communicate via localhost on different ports.
- Sidecar pattern: a helper container runs alongside the main app — e.g. Fluent Bit for log shipping.
- Ambassador pattern: a proxy container handles external communication — e.g. Envoy for service mesh.
- Adapter pattern: normalises the output of the main container into a standard format.
- Rule of thumb: use multi-container pods only for tightly coupled helpers. One container per pod 90% of the time.

■ Example — YAML / Commands

```
# Multi-container pod: app + log sidecar
spec:
  volumes:
    - name: shared-logs
      emptyDir: {}
  containers:
    - name: app
      image: my-app:v1
    - name: log-shipper
      image: fluent/fluent-bit:2.1
  volumeMounts:
    - name: shared-logs
      mountPath: /var/log/app
    - name: log-shipper
      image: fluent/fluent-bit:2.1
  volumeMounts:
    - name: shared-logs
      mountPath: /var/log/app
```

■ Key Insight

Containers in a pod live and die together. If one crashes, the pod restarts. Design sidecars to be robust.

Q13

Pods &
Workloads

What are init containers?

- Init containers run BEFORE the main app containers start — they must complete successfully first.
- They run sequentially — if one fails, K8s retries it until it succeeds before moving on.
- Separate image from main containers — can use busybox for simple checks without bloating your app image.
- Use cases: wait for a database to be ready, run DB migrations, download config files, set file permissions.
- If an init container loops forever, the main app never starts — very common cause of pod stuck in Init state.
- Since K8s 1.29 there are native sidecar init containers that start before other containers and keep running.

■ Example — YAML / Commands

```
spec:
  initContainers:
    - name: wait-for-db
      image: busybox:1.35
      command:
        - sh
        - -c
        - until nc -z postgres 5432; do echo waiting for db; sleep 3; done
    - name: run-migrations
      image: my-app:v2
      command: ['python', 'manage.py', 'migrate']
  containers:
    - name: app
      image: my-app:v2
```

■ Key Insight

Init containers are perfect for dependency checks. A stuck init container is always visible in `kubectl describe Events`.

Q14

Pods &
Workloads

What are sidecar containers?

- Sidecar containers run alongside the main container in the same pod, extending its functionality.
- They share the same network (localhost) and can share volumes with the main container.
- Common sidecars: Fluent Bit (log shipping), Envoy (service mesh proxy), Prometheus exporter.
- Key benefit: you enhance the main app without changing its code or image.
- In K8s 1.29+, sidecar containers can be declared as init containers with restartPolicy: Always.
- They start before the main container and are kept alive for the pod's lifetime.

■ Example — YAML / Commands

```
# Istio automatically injects Envoy as a sidecar
# Manual sidecar example: log shipper
containers:
- name: app
  image: my-app:v1
- name: envoy-proxy
  image: envoyproxy/envoy:v1.28
ports:
- containerPort: 9901
args:
- -c
- /etc/envoy/envoy.yaml
# Check sidecar is running
kubectl get pod my-pod -o jsonpath='{.spec.containers[*].name}'
```

■ Key Insight

Service mesh (Istio/Linkerd) works by auto-injecting a sidecar proxy into every pod. Zero code changes needed.

Q15

Pods &
Workloads

What are ephemeral containers?

- Ephemeral containers are temporary containers injected into a RUNNING pod for live debugging.
- They are NOT in the original pod spec and cannot be restarted after they exit.
- Primary use: debug distroless or scratch images that have no shell or debugging tools.
- They share the same network namespace as the target container — you can curl localhost services.
- Use `--target` to share the process namespace of a specific container for full visibility.
- Available since K8s 1.23 stable. No need to restart the pod to debug it.

■ Example — YAML / Commands

```
# Add ephemeral container to a running pod
kubectl debug -it my-pod \
  --image=busybox:1.35 \
  --target=app \
  -- /bin/sh
# Or use nicolaka/netshoot for network debugging
kubectl debug -it my-pod \
  --image=nicolaka/netshoot \
  --target=app
# Check ephemeral containers in a pod
kubectl get pod my-pod -o jsonpath='{.spec.ephemeralContainers}'
```

■ Key Insight

Ephemeral containers are the right tool for debugging minimal images. No pod restart needed. Available in K8s 1.23+.

Q16

Pods &
Workloads

What is a Deployment?

- A Deployment manages a ReplicaSet to ensure a desired number of pod replicas run at all times.
- Adds rolling updates, rollbacks, and revision history on top of a basic ReplicaSet.
- You describe desired state in YAML — the Deployment controller makes it happen and keeps it that way.
- When you update the image, Deployment creates a new ReplicaSet and gradually scales it up.
- Old ReplicaSets stay at 0 replicas for rollback — K8s keeps the last 10 by default.
- Use Deployment for stateless apps. Use StatefulSet for databases and stateful workloads.

■ Example — YAML / Commands

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: my-app
  namespace: production
spec:
  replicas: 3
  selector:
    matchLabels:
      app: my-app
  template:
    metadata:
      labels:
        app: my-app
    spec:
      containers:
        - name: app
          image: my-app:v2
          resources:
            requests:
              cpu: 200m
              memory: 256Mi
            limits:
              cpu: 500m
              memory: 512Mi
  kubectl apply -f deployment.yaml
  kubectl rollout status deployment/my-app
```

■ Key Insight

Always use Deployment, never bare pods in production. Deployment gives you rollout, rollback, and self-healing for free.

Q17

Pods &
Workloads

How does ReplicaSet work?

- ReplicaSet ensures a specified number of pod replicas are running at all times.
- Uses a label selector to identify which pods it owns and manages.
- If a pod is deleted or crashes, ReplicaSet immediately creates a new one to maintain replica count.
- If too many pods exist matching its selector, ReplicaSet deletes the extras.
- You rarely create a ReplicaSet directly — Deployments create and manage ReplicaSets for you.
- Multiple ReplicaSets per Deployment: during rollout, old RS scales down while new RS scales up.

■ Example — YAML / Commands

```
# List ReplicaSets (see old and new during rollout)
kubectl get replicaset
kubectl get rs -o wide
# Describe a ReplicaSet to see events
kubectl describe rs <rs-name>
# Manually scale a ReplicaSet (not recommended — use Deployment)
kubectl scale rs my-rs --replicas=5
# See which Deployment owns a RS
kubectl get rs -o jsonpath='{.items[*].metadata.ownerReferences}'
```

■ Key Insight

Never create bare ReplicaSets — Deployments manage them. Scaling a RS directly is overridden by the Deployment controller.

Q18

Pods &
Workloads

Difference between Deployment and StatefulSet?

- Deployment: for stateless apps. Pods are interchangeable and get random names (my-app-7f8d-xkp2).
- StatefulSet: for stateful apps. Pods get stable, ordered names: my-db-0, my-db-1, my-db-2.
- StatefulSet gives each pod a stable network identity (DNS) and its own persistent storage (PVC).
- Deployment pods start in parallel. StatefulSet pods start in order (0 then 1 then 2).
- StatefulSet pods stop in reverse order (2, 1, 0) for graceful shutdown of clustered DBs.
- Use StatefulSet for: MySQL, PostgreSQL, Kafka, ZooKeeper, Elasticsearch, Redis Cluster.

■ Example — YAML / Commands

```
# Deployment: pod names are random
# nginx-deploy-7d8f9-xkp2n
# nginx-deploy-7d8f9-m3qz1
# StatefulSet: pod names are stable and ordered
# postgres-0
# postgres-1
# postgres-2
# StatefulSet DNS per pod:
# postgres-0.postgres.default.svc.cluster.local
# postgres-1.postgres.default.svc.cluster.local
kubectl get statefulset
kubectl get pods -l app=postgres
```

■ Key Insight

Rule: stateless app = Deployment. Has its own disk or cluster identity = StatefulSet. Use the wrong one and pain follows.

Q19

Pods &
Workloads

What is a DaemonSet?

- DaemonSet ensures exactly ONE pod runs on every node in the cluster.
- When a new node joins, DaemonSet automatically schedules a pod on it. When a node leaves, pod is garbage collected.
- Can use nodeSelector or affinity to run on a subset of nodes instead of all nodes.
- By default, DaemonSet pods also run on control plane nodes if you add the right toleration.
- Common use cases: log collectors (Fluent Bit), node metrics (Node Exporter), CNI plugins (Calico), security agents.
- You cannot scale a DaemonSet replicas manually — count equals number of matching nodes.

■ Example — YAML / Commands

```
apiVersion: apps/v1
kind: DaemonSet
metadata:
  name: fluent-bit
  namespace: logging
spec:
  selector:
    matchLabels:
      app: fluent-bit
  template:
    metadata:
      labels:
        app: fluent-bit
    spec:
      tolerations:
        - key: node-role.kubernetes.io/control-plane
          effect: NoSchedule
      containers:
        - name: fluent-bit
          image: fluent/fluent-bit:2.1
kubectrl get daemonset -n logging
kubectrl get pods -n logging -o wide # one pod per node
```

■ Key Insight

DaemonSets are perfect for infrastructure agents. If a DS pod is Pending, 99% chance it is a taint/toleration mismatch.

Q20

Pods &
Workloads

What is Job vs CronJob?

- Job: runs one or more pods to COMPLETION then stops — for batch tasks, migrations, one-off reports.
- CronJob: creates a Job on a schedule using standard cron syntax — for nightly backups, hourly reports.
- Deployment keeps pods running forever. Job runs until the task is done and exits successfully.
- Job retries on failure up to backoffLimit (default 6). Set restartPolicy: Never or OnFailure.
- Always set ttlSecondsAfterFinished to auto-clean completed Jobs — they accumulate fast otherwise.
- Set activeDeadlineSeconds to kill Jobs that hang forever.

■ Example — YAML / Commands

```
# One-off Job: DB migration
apiVersion: batch/v1
kind: Job
metadata:
  name: db-migration
spec:
  ttlSecondsAfterFinished: 3600
  template:
    spec:
      containers:
        - name: migrate
          image: my-app:v2
          command: ['python', 'manage.py', 'migrate']
          restartPolicy: Never
  ---
# CronJob: nightly backup at 2am
apiVersion: batch/v1
kind: CronJob
spec:
  schedule: '0 2 * * *'
  jobTemplate:
    spec:
      template:
        spec:
          containers:
            - name: backup
              image: backup-tool:v1
              restartPolicy: OnFailure
```

■ Key Insight

Always set ttlSecondsAfterFinished. Without it, old Job pods pile up in the cluster and consume etcd storage.

Q21

Pods &
Workloads

What is a rollout in Deployment?

- A rollout is the process of updating a Deployment to a new version with zero downtime.
- K8s creates a new ReplicaSet with the new image and scales it up while scaling the old one down.
- Rolling strategy (default): maxSurge controls extra pods during update, maxUnavailable controls pods that can be down.
- Rollout only progresses once new pods pass their readiness probe — critical for zero-downtime.
- kubectl rollout status streams the progress in real time.
- Rollout pauses if pods fail to become ready — preventing a bad version from fully rolling out.

■ Example — YAML / Commands

```
# Trigger a rollout by updating the image
kubectl set image deployment/my-app app=my-app:v3
# Watch rollout progress
kubectl rollout status deployment/my-app
# See rollout history (revisions)
kubectl rollout history deployment/my-app
# Configure rolling update strategy in YAML
strategy:
  type: RollingUpdate
  rollingUpdate:
    maxSurge: 1 # 1 extra pod above desired
    maxUnavailable: 0 # zero downtime
```

■ Key Insight

maxUnavailable: 0 + a readiness probe = true zero-downtime rollout. Without the probe, traffic hits unhealthy pods.

Q22

Pods &
Workloads

How does rollback work in Deployment?

- K8s keeps old ReplicaSets with 0 replicas — these are your rollback points.
- Rolling back just scales up the previous ReplicaSet and scales down the current one — very fast.
- By default stores last 10 revisions (revisionHistoryLimit). Increase for more rollback options.
- `kubectl rollout undo` rolls back to the previous revision in seconds.
- You can rollback to a specific revision using `--to-revision` flag.
- After rollback, check pod status and logs to confirm the previous version is healthy.

■ Example — YAML / Commands

```
# Quick rollback to previous version
kubectl rollout undo deployment/my-app

# Check revision history first
kubectl rollout history deployment/my-app

# Rollback to a specific revision
kubectl rollout undo deployment/my-app --to-revision=3

# Verify rollback succeeded
kubectl rollout status deployment/my-app
kubectl get pods -l app=my-app
kubectl describe deployment my-app | grep Image
```

■ Key Insight

Rollback is instant because K8s just re-scales an old ReplicaSet. Keep `revisionHistoryLimit` at 10+ in production.

Q23

Services &
Networking

What is a Kubernetes Service?

- A Service provides a stable, permanent IP and DNS name for accessing a group of pods.
- Pods are ephemeral — they get new IPs when recreated. A Service gives you a fixed endpoint.
- Service routes traffic to pods using label selectors — any pod with matching labels gets traffic.
- ClusterDNS: every service gets a DNS record — <name>.<namespace>.svc.cluster.local.
- Services also load balance traffic across all healthy, ready pods behind them.
- Without a Service, pods can only be reached by their changing IP address — not practical.

■ Example — YAML / Commands

```
apiVersion: v1
kind: Service
metadata:
  name: my-app-svc
  namespace: production
spec:
  selector:
    app: my-app # matches pods with this label
  ports:
    - port: 80 # port the service listens on
      targetPort: 8080 # port your container listens on
# Verify endpoints are populated
kubectl get endpoints my-app-svc
# Test service from inside cluster
kubectl run test --image=busybox --rm -it -- wget -O- http://my-app-svc
```

■ Key Insight

If `kubectl get endpoints` shows no addresses, the Service selector does not match any pod label. That is the #1 Service bug.

Q24

Services &
Networking

What are the types of Kubernetes Services?

- ClusterIP (default): internal-only virtual IP. Only reachable inside the cluster.
- NodePort: exposes service on a static port (30000-32767) on every node's external IP.
- LoadBalancer: provisions a cloud load balancer (AWS ELB, GCP LB). External traffic entry point.
- ExternalName: maps a service to an external DNS name (CNAME). No proxying, just DNS.
- Headless (clusterIP: None): no virtual IP. DNS returns individual pod IPs directly.
- Production best practice: ClusterIP for all internal services + one Ingress Controller for external HTTP/HTTPS.

■ Example — YAML / Commands

```
# ClusterIP: internal only (default)
spec:
  type: ClusterIP
# NodePort: accessible via <NodeIP>:31000
spec:
  type: NodePort
ports:
  - port: 80
  nodePort: 31000
# LoadBalancer: cloud LB created automatically
spec:
  type: LoadBalancer
# Headless: DNS returns pod IPs
spec:
  clusterIP: None
```

■ Key Insight

Use ClusterIP + Ingress in production. LoadBalancer per service gets expensive fast — one Ingress handles all of them.

Q25

Services &
Networking

What is ClusterIP?

- ClusterIP is the default Service type — it gets a virtual IP that is only reachable inside the cluster.
- The IP comes from the cluster's Service CIDR range (e.g. 10.96.0.0/12).
- kube-proxy creates iptables/IPVS rules on every node to forward ClusterIP traffic to pod IPs.
- DNS: <service>.<namespace>.svc.cluster.local always resolves to the ClusterIP.
- Perfect for service-to-service communication where external access is not needed.
- ClusterIP is completely firewalled from the outside — cannot be accessed without kubectl proxy or port-forward.

■ Example — YAML / Commands

```
# Create ClusterIP service
kubectl expose deployment my-app --port=80 --target-port=8080
# Get the ClusterIP
kubectl get svc my-app -o jsonpath='{.spec.clusterIP}'
# Test from inside the cluster
kubectl exec -it my-pod -- curl http://my-app:80
# Port-forward for local testing (not production)
kubectl port-forward svc/my-app 8080:80
# Now access http://localhost:8080 from your laptop
```

■ Key Insight

ClusterIP + Ingress is the production pattern. ClusterIP alone gives you fast, free internal load balancing between services.

Q26

Services &
Networking

What is NodePort?

- NodePort opens a static port on EVERY node's IP — external traffic hits <NodeIP>:<NodePort>.
- Port range: 30000-32767 (default). Can be specified or auto-assigned by K8s.
- Traffic flow: external → NodeIP:NodePort → Service ClusterIP → Pod.
- Useful for on-prem clusters or quick testing where you don't have a cloud load balancer.
- Not recommended for production: no health check on the node IP, exposes all nodes, harder to manage.
- Every LoadBalancer service is also a NodePort service underneath.

■ Example — YAML / Commands

```
apiVersion: v1
kind: Service
metadata:
  name: my-app-nodeport
spec:
  type: NodePort
  selector:
    app: my-app
  ports:
    - port: 80 # ClusterIP port
      targetPort: 8080 # Pod port
      nodePort: 31080 # Node port (omit for auto-assign)
# Access from outside (replace with actual node IP)
curl http://<node-external-ip>:31080
# Get assigned NodePort
kubectl get svc my-app-nodeport -o jsonpath='{.spec.ports[0].nodePort}'
```

■ Key Insight

NodePort is fine for dev/testing. In production use Ingress (for HTTP) or LoadBalancer (for TCP). Avoid NodePort at scale.

Q27

Services &
Networking

What is LoadBalancer service?

- LoadBalancer extends NodePort — it also provisions a cloud load balancer automatically.
- Cloud LB gets a public IP/DNS and forwards traffic to the NodePort on each worker node.
- Works out-of-box on AWS (ELB/NLB), GCP, Azure. Needs MetalLB for on-prem bare metal.
- The LB IP is shown in `kubectl get svc` EXTERNAL-IP column after provisioning (can take 1-2 min).
- Cost: each LoadBalancer service = one cloud LB. At scale this gets expensive fast.
- Better pattern: one LoadBalancer for the Ingress Controller, then Ingress rules for all services.

■ Example — YAML / Commands

```
# Create a LoadBalancer service
kubectl expose deployment my-app \
  --type=LoadBalancer --port=80 --target-port=8080
# Watch for external IP to be provisioned
kubectl get svc my-app -w
# AWS: annotate for NLB instead of CLB
metadata:
  annotations:
    service.beta.kubernetes.io/aws-load-balancer-type: nlb
# AWS: internal LB (not internet-facing)
service.beta.kubernetes.io/aws-load-balancer-internal: 'true'
```

■ Key Insight

One LoadBalancer for the Ingress Controller + Ingress rules for all apps = massive cost saving vs one LB per service.

Q28

Services &
Networking

What is a Headless Service?

- Headless Service has clusterIP: None — no virtual IP is assigned to the service.
- DNS query for a Headless Service returns ALL pod IPs directly instead of a single service IP.
- Essential for StatefulSets: each pod gets its own stable DNS entry —
 <pod>.<svc>.<ns>.svc.cluster.local.
- The client decides how to use the pod IPs — e.g. DB driver connects to primary, not a random pod.
- Also used when the app handles its own service discovery or load balancing.
- Without a Headless Service, StatefulSet pods cannot address each other by stable DNS name.

■ Example — YAML / Commands

```
# Headless Service
apiVersion: v1
kind: Service
metadata:
  name: postgres
spec:
  clusterIP: None # This makes it headless
  selector:
    app: postgres
  ports:
    - port: 5432
# DNS entries created (one per pod):
# postgres-0.postgres.default.svc.cluster.local
# postgres-1.postgres.default.svc.cluster.local
# Verify DNS resolution from inside cluster
kubectl exec -it my-pod -- nslookup postgres-0.postgres
```

■ Key Insight

Headless Service + StatefulSet = the foundation for running databases in K8s. Stable DNS per pod is everything.

Q29

Services &
Networking

What is Ingress?

- Ingress is an API object that manages external HTTP/HTTPS access to multiple Services.
- Provides host-based routing (api.example.com vs app.example.com) and path-based routing (/api, /web).
- Requires an Ingress Controller to be installed — the Ingress resource alone does nothing without it.
- Common controllers: NGINX Ingress, AWS ALB Controller, Traefik, HAProxy.
- Supports TLS termination — you configure one SSL cert at the Ingress, not on every pod.
- Cost win: one cloud LB for all apps via Ingress vs one cloud LB per service.

■ Example — YAML / Commands

```
apiVersion: networking.k8s.io/v1
kind: Ingress
metadata:
  name: app-ingress
  annotations:
    nginx.ingress.kubernetes.io/rewrite-target: /
spec:
  tls:
    - hosts: [app.example.com]
      secretName: tls-cert
  rules:
    - host: app.example.com
      http:
        paths:
          - path: /api
            pathType: Prefix
            backend:
              service:
                name: api-svc
                port:
                  number: 80
          - path: /
            pathType: Prefix
            backend:
              service:
                name: frontend-svc
                port:
                  number: 80
```

■ Key Insight

Ingress = one LB for all HTTP/HTTPS apps. Service LoadBalancer = one LB per app. Use Ingress to save money at scale.

Q30

Services &
Networking

Difference between Ingress and Service?

- Service (L4): routes TCP/UDP traffic to pods. No URL routing, no hostname routing, no TLS termination.
- Ingress (L7): routes HTTP/HTTPS traffic based on hostname and URL path. Handles TLS termination.
- A Service exposes ONE app and requires one cloud LB per LoadBalancer-type service.
- An Ingress sits in front of many Services — one cloud LB handles routing to all of them.
- Ingress supports redirects, rewrites, rate limiting, authentication — Service does not.
- In production: ClusterIP for all backend services + Ingress Controller for external access.

■ Example — YAML / Commands

```
# Without Ingress: 3 LoadBalancer services = 3 cloud LBs = $$$$  
kubectl get svc | grep LoadBalancer  
# With Ingress: 1 LB for the controller, unlimited services behind it  
kubectl get svc -n ingress-nginx # only one external LB  
# Debug ingress routing  
kubectl describe ingress my-ingress  
kubectl get endpoints api-svc  
kubectl logs -n ingress-nginx deploy/ingress-nginx-controller | tail -20
```

■ Key Insight

Use Service for stable pod access inside the cluster. Use Ingress for external HTTP/HTTPS access to multiple apps.

Q31

Services &
Networking

How does kube-proxy work internally?

- kube-proxy runs on every node as a DaemonSet and manages network rules for Services.
- It watches the API server for Service and Endpoint changes and updates kernel rules in real time.
- iptables mode (default): adds PREROUTING iptables rules that DNAT ClusterIP → pod IP randomly.
- IPVS mode: uses Linux Virtual Server kernel module — faster, more LB algorithms, O(1) rule lookup.
- Every node has identical rules — traffic is local, no extra hops to a centralised proxy.
- kube-proxy does NOT handle DNS. DNS is CoreDNS. Two separate systems.

■ Example — YAML / Commands

```
# Verify kube-proxy mode
kubectl get configmap kube-proxy -n kube-system -o yaml | grep mode
# View iptables rules for all services (on the node)
iptables -t nat -L KUBE-SERVICES -n --line-numbers
# Switch to IPVS mode (edit configmap then restart kube-proxy)
kubectl edit configmap kube-proxy -n kube-system
# Change mode: '' to mode: 'ipvs'
# View IPVS virtual servers
ipvsadm -Ln
```

■ Key Insight

IPVS mode is better for clusters with 1000+ services. iptables becomes slow — rule count grows linearly with services.

Q32

Services &
Networking

What is CNI?

- CNI (Container Network Interface) is the plugin system that handles pod networking in Kubernetes.
- K8s itself has no built-in networking — it calls a CNI plugin when pods start and stop.
- CNI plugin responsibilities: assign IP to pod, create network interface (veth pair), set up routes.
- Calico: BGP routing, full NetworkPolicy support. Most common in production.
- Cilium: eBPF-based, highest performance and observability. Growing fast for production use.
- Flannel: simple VXLAN overlay, easy to set up, but limited NetworkPolicy support.
- Without a CNI plugin, pods cannot communicate — cluster is completely broken.

■ Example — YAML / Commands

```
# Install Calico CNI
kubectl apply -f https://docs.projectcalico.org/manifests/calico.yaml
# Verify CNI pods are healthy
kubectl get pods -n kube-system | grep calico
# Check pod IPs are assigned (CNI working)
kubectl get pods -o wide
# NetworkPolicy only works with Calico/Cilium — NOT Flannel
kubectl get networkpolicies -A
# Debug CNI issues on node
ls /etc/cni/net.d/
cat /etc/cni/net.d/10-calico.conflist
```

■ Key Insight

Always confirm your CNI supports NetworkPolicy before relying on it for security. Flannel does NOT. Calico and Cilium do.

Q33

Services &
Networking

How does DNS work inside Kubernetes?

- K8s runs CoreDNS as the cluster DNS server — all pods are configured to use it via `/etc/resolv.conf`.
- Every Service gets a DNS entry automatically: `<svc>.<namespace>.svc.cluster.local`.
- StatefulSet pods get per-pod DNS: `<pod>.<svc>.<namespace>.svc.cluster.local`.
- Same-namespace access: just use the service name — `curl http://my-svc`.
- Cross-namespace access: use full name — `curl http://my-svc.other-ns.svc.cluster.local`.
- DNS lookup chain: pod → CoreDNS → resolves to ClusterIP → kube-proxy routes to pod IP.

■ Example — YAML / Commands

```
# Debug DNS from inside a pod
kubectl exec -it my-pod -- nslookup my-svc
kubectl exec -it my-pod -- nslookup my-svc.prod.svc.cluster.local
# Check resolv.conf inside pod
kubectl exec -it my-pod -- cat /etc/resolv.conf
# Check CoreDNS is running
kubectl get pods -n kube-system -l k8s-app=kube-dns
# Run one-shot DNS debug pod
kubectl run dns-test --image=busybox --rm -it \
--restart=Never -- nslookup kubernetes.default
```

■ Key Insight

Always use FQDN for cross-namespace calls. Short names only resolve in the same namespace — #1 DNS bug in production.

Q34

Services &
Networking

How do Pods communicate across nodes?

- K8s networking rule: every pod can reach every other pod directly — no NAT, no port mapping.
- Each pod gets a unique cluster-wide IP from the pod CIDR range (e.g. 10.244.0.0/16).
- CNI plugin creates the overlay or underlay network so pod IPs are routable across all nodes.
- Flannel: wraps pod traffic in VXLAN (UDP) packets — tunnel between nodes. Easy but adds overhead.
- Calico: advertises pod CIDRs via BGP routing — no overlay, native routing, much lower latency.
- Cilium: uses eBPF for kernel-level routing — fastest option with best observability.

■ Example — YAML / Commands

```
# Check pod IPs across different nodes
kubectl get pods -o wide -A

# Test cross-node pod connectivity
kubectl exec -it pod-on-node1 -- ping <ip-of-pod-on-node2>
kubectl exec -it pod-on-node1 -- curl http://<ip-of-pod-on-node2>:8080

# Check pod CIDR on a node
kubectl get node <node> -o jsonpath='{.spec.podCIDR}'

# Trace route between pods
kubectl exec -it my-pod -- traceroute <target-pod-ip>
```

■ Key Insight

Pod-to-pod communication works the same way whether pods are on the same node or different nodes. CNI handles all of it.

Q35

Configuration &
Storage

What are ConfigMaps?

- ConfigMap stores non-sensitive configuration data as key-value pairs or files.
- Decouples config from the container image — change config without rebuilding and redeploying the image.
- Can store: simple key-value pairs, entire config files (nginx.conf, application.yaml), feature flags.
- Inject into pods as: environment variables, command-line arguments, or volume-mounted files.
- Changing a ConfigMap does NOT automatically update running pods using env vars — requires pod restart.
- Volume-mounted ConfigMaps auto-update inside running pods after ~1 minute (but app must re-read the file).

■ Example — YAML / Commands

```
apiVersion: v1
kind: ConfigMap
metadata:
  name: app-config
data:
  APP_ENV: production
  LOG_LEVEL: info
  config.yaml: |
    database:
      host: postgres-svc
      port: 5432
    cache:
      ttl: 300
# Use as env vars
envFrom:
  - configMapRef:
      name: app-config
# Use as mounted file
volumes:
  - name: config
    configMap:
      name: app-config
```

■ Key Insight

ConfigMap change + `kubectl rollout restart deployment/my-app` = safest way to apply config changes to all pods.

Q36

Configuration &
Storage

What are Secrets?

- Secrets store sensitive data: passwords, API keys, TLS certificates, tokens.
- Stored base64-encoded in etcd — base64 is NOT encryption. Enable encryption at rest for real security.
- Types: Opaque (generic), kubernetes.io/dockerconfigjson (image pull secret), kubernetes.io/tls (TLS certs).
- Like ConfigMaps but: not shown in `kubectl describe` by default, can be encrypted at rest, RBAC restricted.
- Inject as env vars or volume-mounted files — volume mounts are safer (less likely to appear in logs).
- For production: use External Secrets Operator + AWS Secrets Manager or HashiCorp Vault.

■ Example — YAML / Commands

```
# Create secret from CLI
kubectl create secret generic db-secret \
--from-literal=DB_PASSWORD=mysecretpass \
--from-literal=DB_USER=admin
# Use in pod as env var
env:
- name: DB_PASSWORD
valueFrom:
secretKeyRef:
name: db-secret
key: DB_PASSWORD
# Image pull secret for private registry
kubectl create secret docker-registry my-registry \
--docker-server=myrepo.io \
--docker-username=user \
--docker-password=pass
```

■ Key Insight

Secrets in plain K8s are just base64. Use External Secrets Operator + Vault or AWS SM for real secret management.

Q37

Configuration &
Storage

Difference between ConfigMap and Secret?

- ConfigMap: for non-sensitive data — feature flags, URLs, log levels, config files. Stored as plain text.
- Secret: for sensitive data — passwords, tokens, certs. Stored base64-encoded in etcd.
- Secrets are redacted in kubectl describe output. ConfigMap values are visible.
- Secrets can be encrypted at rest using EncryptionConfiguration. ConfigMaps cannot.
- Both can be used as env vars or volume mounts. The API is nearly identical.
- RBAC tip: restrict Secret read access to only the pods/humans that need them. ConfigMaps are usually open.

■ Example — YAML / Commands

```
# ConfigMap: visible in plain text
kubectl get configmap app-config -o yaml
# Secret: value is base64 encoded (not encrypted!)
kubectl get secret db-secret -o yaml
# Decode a secret value
kubectl get secret db-secret \
-o jsonpath='{.data.DB_PASSWORD}' | base64 --decode
# Enable encryption at rest (add to API server)
# --encryption-provider-config=/etc/kubernetes/encryption.yaml
# Who can read secrets? (audit RBAC)
kubectl auth can-i get secrets --as=dev-user -n prod
```

■ Key Insight

Golden rule: ConfigMap = config. Secret = credentials. Never put passwords in ConfigMaps. Never commit secrets to git.

Q38

Configuration &
Storage

How to inject environment variables into Pods?

- Method 1 — Direct value: hardcode in env: - name: KEY value: VALUE (for static non-sensitive values).
- Method 2 — From ConfigMap key: use valueFrom.configMapKeyRef to pull a specific key.
- Method 3 — From Secret key: use valueFrom.secretKeyRef to pull a specific secret key.
- Method 4 — Entire ConfigMap as env: use envFrom.configMapRef to inject all keys as env vars.
- Method 5 — Entire Secret as env: use envFrom.secretRef to inject all secret keys as env vars.
- Downward API: inject pod's own metadata — name, namespace, node, resource limits as env vars.

■ Example — YAML / Commands

```
containers:
- name: app
image: my-app:v1
env:
- name: APP_ENV
  value: production # direct value
- name: DB_HOST
  valueFrom:
    configMapKeyRef:
      name: app-config
      key: DB_HOST # from ConfigMap
- name: DB_PASS
  valueFrom:
    secretKeyRef:
      name: db-secret
      key: password # from Secret
- name: POD_NAME
  valueFrom:
    fieldRef:
      fieldPath: metadata.name # Downward API
envFrom:
- configMapRef:
  name: feature-flags # all CM keys as env
```

■ Key Insight

Env var changes require pod restart. Volume-mounted ConfigMaps auto-update. Prefer mounts for frequently changing config.

Q39

Configuration &
Storage

What is Persistent Volume (PV)?

- A PV is a piece of storage in the cluster — provisioned by an admin or dynamically by a StorageClass.
- It is a cluster-scoped resource with its own lifecycle independent of any pod.
- Backed by real storage: AWS EBS, GCP PD, Azure Disk, NFS, Ceph, iSCSI, local disk.
- Access modes: ReadWriteOnce (RWO — one node), ReadOnlyMany (ROX), ReadWriteMany (RWX — needs NFS/EFS).
- Reclaim policy: Delete (PV deleted when PVC deleted), Retain (PV kept for manual recovery).
- Phase states: Available → Bound (when a PVC claims it) → Released → Deleted.

■ Example — YAML / Commands

```
# Manual PV (rarely needed – use dynamic provisioning instead)
apiVersion: v1
kind: PersistentVolume
metadata:
  name: my-pv
spec:
  capacity:
    storage: 20Gi
  accessModes:
    - ReadWriteOnce
  persistentVolumeReclaimPolicy: Retain
  storageClassName: fast-ssd
  awsElasticBlockStore:
    volumeID: vol-0123456789
    fsType: ext4
# Check PV status
kubectl get pv
kubectl describe pv my-pv
```

■ Key Insight

Never create PVs manually in cloud environments — use StorageClass with dynamic provisioning. Manual PVs are admin burden.

Q40

Configuration &
Storage

What is Persistent Volume Claim (PVC)?

- A PVC is a request for storage by a user or pod — like ordering a disk from a catalog.
- Namespace-scoped: a PVC belongs to a namespace, a PV is cluster-scoped.
- User specifies: size (10Gi), access mode (RWO), and optionally a StorageClass.
- K8s finds a matching PV and binds the PVC to it. The pod then uses the PVC as a volume.
- Once bound, the PVC and PV are locked together — no other pod can claim the same PV (for RWO).
- StatefulSet uses volumeClaimTemplates to create one PVC per pod automatically.

■ Example — YAML / Commands

```
# PVC requesting 10Gi from gp3 StorageClass
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
  name: db-data
  namespace: production
spec:
  accessModes:
    - ReadWriteOnce
  storageClassName: gp3
  resources:
    requests:
      storage: 10Gi
# Check PVC binding status
kubectl get pvc -n production
# STATUS should be Bound – Pending means no matching PV
kubectl describe pvc db-data
```

■ Key Insight

PVC stuck in Pending: StorageClass doesn't exist, or provisioner has no permission, or wrong access mode.

Q41

Configuration &
Storage

Difference between PV and PVC?

- PV: the actual storage resource — cluster-scoped, created by admin or dynamic provisioner.
- PVC: a claim/request for storage — namespace-scoped, created by developers or StatefulSets.
- Analogy: PV is a physical disk drive in a server rack. PVC is the ticket to mount that drive.
- K8s binds a PVC to a suitable PV based on: requested size, access mode, and StorageClass.
- A PV can only be bound to ONE PVC at a time (for RWO mode).
- Relationship is permanent until the PVC is deleted — even if the pod is deleted, PVC and PV stay.

■ Example — YAML / Commands

```
# See PVs and their binding status
kubectl get pv

# CLAIM column shows which PVC is bound
# See PVCs and their status
kubectl get pvc -A

# STATUS: Bound = good, Pending = problem
# Dynamic provisioning: PVC triggers auto-creation of PV
# 1. Create PVC (below)
# 2. StorageClass provisions a real disk in AWS/GCP
# 3. PV object appears automatically
# 4. PVC binds to the new PV
kubectl describe pvc db-data | grep 'Volume:'
kubectl get pv $(kubectl get pvc db-data -o jsonpath='{.spec.volumeName}')
```

■ Key Insight

Always use dynamic provisioning. Create a PVC and let StorageClass handle the PV. Never manually create PVs in the cloud.

Q42

Configuration &
Storage

What is StorageClass?

- StorageClass defines the type of storage — disk tier, provisioner, replication, reclaim policy.
- Enables dynamic provisioning: a PV is automatically created when a PVC is submitted.
- Each cloud has its own provisioner: ebs.csi.aws.com (AWS), pd.csi.storage.gke.io (GCP).
- Parameters control disk options: type (gp2, gp3, io1 for AWS EBS), encrypted, throughput.
- reclaimPolicy: Delete (disk auto-deleted on PVC delete) or Retain (disk kept for recovery).
- volumeBindingMode: WaitForFirstConsumer — waits for a pod to be scheduled before provisioning disk in the right AZ.

■ Example — YAML / Commands

```
# AWS EBS gp3 StorageClass
apiVersion: storage.k8s.io/v1
kind: StorageClass
metadata:
  name: fast-ssd
annotations:
  storageclass.kubernetes.io/is-default-class: 'true'
provisioner: ebs.csi.aws.com
parameters:
  type: gp3
  encrypted: 'true'
  reclaimPolicy: Delete
  volumeBindingMode: WaitForFirstConsumer
# List available StorageClasses
kubectl get storageclass
kubectl get sc
```

■ Key Insight

Set volumeBindingMode: WaitForFirstConsumer — prevents EBS volumes being created in the wrong AZ and causing Pending PVCs.

Q43

Configuration &
Storage

What is dynamic provisioning?

- Dynamic provisioning auto-creates PVs on-demand when a PVC is submitted — no admin pre-work.
- Before dynamic provisioning: admins had to manually create PVs in advance. Not scalable.
- The StorageClass provisioner (e.g. AWS EBS CSI Driver) calls the cloud API to create the actual disk.
- Flow: PVC created → StorageClass provisioner creates real disk → PV object created → PVC binds to PV.
- reclaimPolicy: Delete = disk is deleted when PVC is deleted. Retain = disk survives for manual recovery.
- Always use dynamic provisioning in cloud environments. Static PVs are for on-prem with NFS/Ceph.

■ Example — YAML / Commands

```
# Enable dynamic provisioning: just set the storageClassName
# and submit a PVC – no admin action needed
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
  name: auto-pvc
spec:
  storageClassName: fast-ssd # triggers dynamic provisioning
  accessModes: [ReadWriteOnce]
resources:
  requests:
    storage: 50Gi
# Watch the PV appear automatically
kubectl get pvc auto-pvc -w
kubectl get pv # new PV appears after a few seconds
```

■ Key Insight

Dynamic provisioning is cloud-native storage. Set a good default StorageClass and developers never think about disks again.

Q44

Configuration &
Storage

How do you mount storage in Pods?

- Declare volumes in `spec.volumes` (what storage to use), mount them in `spec.containers.volumeMounts` (where).
- Volume types: `persistentVolumeClaim` (PVC), `configMap`, `secret`, `emptyDir` (temp), `hostPath` (node disk).
- `emptyDir`: created when pod starts, deleted when pod dies. Good for temp files, caches, shared memory.
- `hostPath`: mounts a path from the host node — avoid in production, breaks pod portability.
- Multiple containers in a pod can mount the same volume — used for sidecars to read app logs.
- `subPath`: mount a single file or subdirectory from a volume instead of the whole volume.

■ Example — YAML / Commands

```
spec:
  volumes:
    - name: data
      persistentVolumeClaim:
        claimName: db-data
    - name: config
      configMap:
        name: app-config
    - name: tmp
      emptyDir: {}
  containers:
    - name: app
      image: my-app:v1
      volumeMounts:
        - name: data
          mountPath: /var/data
        - name: config
          mountPath: /etc/app
          readOnly: true
        - name: tmp
          mountPath: /tmp/cache
```

■ Key Insight

Always mark `ConfigMap/Secret` volume mounts as `readOnly: true`. No reason for apps to write to config files.

Q45

Scheduling &
Resource
Management

What are resource requests and limits?

- Request: minimum resources guaranteed to the container. Scheduler uses this to pick a node.
- Limit: maximum resources the container can use. Enforced at runtime by the Linux kernel.
- CPU limit: container is CPU-throttled (slowed down) if it exceeds the limit. NOT killed.
- Memory limit: container is OOMKilled (killed immediately) if it exceeds the limit.
- Without requests: scheduler cannot make good decisions — pods land on overloaded nodes.
- Without limits: one runaway pod can starve all other pods on the same node.

■ Example — YAML / Commands

```
containers:
- name: app
resources:
requests:
cpu: 250m # 0.25 core — guaranteed
memory: 256Mi # 256MB guaranteed
limits:
cpu: 500m # throttled above 0.5 core
memory: 512Mi # OOMKilled above 512MB
# Check actual resource usage
kubectl top pod my-pod
kubectl top node
# See allocated resources on a node
kubectl describe node <node> | grep -A10 'Allocated resources'
```

■ Key Insight

Always set both requests and limits. Pods without limits are time bombs. Use VPA recommendations to rightsize them.

Q46

Scheduling &
Resource
Management

What is QoS class in Kubernetes?

- K8s assigns a QoS (Quality of Service) class to every pod based on resource configuration.
- Guaranteed: ALL containers have requests == limits for both CPU and memory. Highest priority. Last to be evicted.
- Burstable: at least one container has requests set but not equal to limits. Medium priority.
- BestEffort: NO containers have requests or limits set. Lowest priority. First to be evicted under pressure.
- When a node runs out of memory, K8s evicts BestEffort pods first, then Burstable, then Guaranteed last.
- Check QoS class: `kubectl get pod <n> -o jsonpath='{.status.qosClass}'`

■ Example — YAML / Commands

```
# Guaranteed QoS: requests == limits
resources:
  requests:
    cpu: 500m
    memory: 512Mi
  limits:
    cpu: 500m # same as request
    memory: 512Mi # same as request
# Check QoS class of a pod
kubectl get pod my-pod -o jsonpath='{.status.qosClass}'
# Find all BestEffort pods (high eviction risk)
kubectl get pods -A -o json | \
jq '.items[] | select(.status.qosClass=="BestEffort") | .metadata.name'
```

■ Key Insight

For critical production pods: set requests == limits to get Guaranteed QoS. They survive memory pressure events.

Q47

Scheduling &
Resource
Management

What are taints and tolerations?

- Taint: applied to a node — repels pods that don't explicitly tolerate it.
- Toleration: applied to a pod — allows it to be scheduled on a tainted node.
- Three taint effects: NoSchedule (no new pods), PreferNoSchedule (try not to), NoExecute (evict existing pods).
- Use taints to dedicate nodes: GPU nodes for ML, spot instances for batch, system nodes for infra pods.
- Adding NoExecute taint to a node evicts all existing pods that don't tolerate it.
- Control plane nodes have a taint by default — that's why your app pods don't land there.

■ Example — YAML / Commands

```
# Taint a node for GPU workloads only
kubectl taint nodes gpu-node-1 dedicated=gpu:NoSchedule
# Remove a taint (add - at end)
kubectl taint nodes gpu-node-1 dedicated=gpu:NoSchedule-
# Pod with matching toleration
spec:
  tolerations:
    - key: dedicated
      operator: Equal
      value: gpu
      effect: NoSchedule
# View taints on all nodes
kubectl describe nodes | grep -A2 Taints
```

■ Key Insight

Taint + toleration + nodeAffinity together = fine-grained node targeting. Use all three for dedicated GPU/batch nodes.

Q48

Scheduling &
Resource
Management

What is nodeSelector?

- nodeSelector is the simplest node targeting method — schedules pods only on nodes with matching labels.
- The pod will only be scheduled on nodes that have ALL the labels specified in nodeSelector.
- It is a hard constraint — if no matching node exists, the pod stays Pending forever.
- Use case: target SSD nodes for database pods, specific AZ for data locality.
- Limitation: only supports exact label match — use nodeAffinity for more complex expressions.
- Label your nodes descriptively: disktype=ssd, zone=us-east-1a, instance-type=memory-optimised.

■ Example — YAML / Commands

```
# Label a node
kubectl label nodes node-1 disktype=ssd
kubectl label nodes node-1 zone=us-east-1a
# List node labels
kubectl get nodes --show-labels
# Use nodeSelector in pod spec
spec:
  nodeSelector:
    disktype: ssd
    zone: us-east-1a
# If no node matches, pod stays Pending
# kubectl describe pod shows: 0/3 nodes available: 3 node(s) didn't match node selector
```

■ Key Insight

nodeSelector is a hard constraint. If label changes on all nodes, all pods using that selector go Pending. Use carefully.

Q49

Scheduling &
Resource
Management

What is node affinity?

- Node affinity is the more expressive version of nodeSelector — supports operators and soft preferences.
- `requiredDuringSchedulingIgnoredDuringExecution`: HARD rule — pod won't schedule if not met.
- `preferredDuringSchedulingIgnoredDuringExecution`: SOFT rule — tries to satisfy but schedules anyway.
- Supports operators: In, NotIn, Exists, DoesNotExist, Gt (greater than), Lt (less than).
- `IgnoredDuringExecution` means: if a node label changes after the pod is running, the pod is NOT evicted.
- Combine required + preferred: hard constraint on region, soft preference for AZ.

■ Example — YAML / Commands

```
affinity:
nodeAffinity:
  requiredDuringSchedulingIgnoredDuringExecution:
    nodeSelectorTerms:
      - matchExpressions:
          - key: topology.kubernetes.io/zone
            operator: In
            values:
              - us-east-1a
              - us-east-1b
    preferredDuringSchedulingIgnoredDuringExecution:
      - weight: 80
        preference:
          matchExpressions:
            - key: disktype
              operator: In
              values: [ssd]
```

■ Key Insight

Use required for hard zone constraints, preferred for soft disk preferences. Combine both for production placement control.

Q50

Scheduling &
Resource
Management

What is pod affinity and anti-affinity?

- Pod affinity: schedule a pod ON nodes where specific other pods are running (co-location).
- Pod anti-affinity: schedule a pod AWAY FROM nodes where specific pods are running (spreading).
- topologyKey defines the scope: `kubernetes.io/hostname` (per node) or `topology.kubernetes.io/zone` (per AZ).
- Use anti-affinity to spread replicas across nodes/AZs — prevents all replicas dying on one node failure.
- required anti-affinity: pod will not schedule if a matching pod is on the node — hard guarantee.
- preferred anti-affinity: tries to spread but schedules anyway if no other option.

■ Example — YAML / Commands

```
# Anti-affinity: spread replicas across different nodes
affinity:
  podAntiAffinity:
    requiredDuringSchedulingIgnoredDuringExecution:
      - labelSelector:
          matchLabels:
            app: my-app
        topologyKey: kubernetes.io/hostname
# Result: no two my-app pods land on the same node
# Great for HA – node failure kills at most 1 replica
# Preferred anti-affinity: spread across AZs (soft)
preferredDuringSchedulingIgnoredDuringExecution:
  - weight: 100
    podAffinityTerm:
      labelSelector:
        matchLabels:
          app: my-app
      topologyKey: topology.kubernetes.io/zone
```

■ Key Insight

Always add pod anti-affinity to production Deployments. Without it, all 3 replicas can land on one node and die together.

Q51

Scheduling &
Resource
Management

How does the scheduler select a node?

- Step 1 — Filtering: eliminate nodes that fail hard constraints. Results in a list of feasible nodes.
- Filters include: resource fit (CPU/memory), taint tolerations, node affinity, hostPort conflicts, volume topology.
- Step 2 — Scoring: rank feasible nodes using priority functions. Each function returns a score 0-100.
- Scoring functions: LeastAllocated (prefer nodes with most free resources), ImageLocality, NodeAffinity weight.
- Step 3 — Binding: scheduler writes the selected nodeName to the pod spec via kube-apiserver.
- kubelet on the winning node picks up the pod and starts the containers.

■ Example — YAML / Commands

```
# Why did the scheduler pick this node?
kubectl get events --field-selector reason=Scheduled | grep my-pod

# See scheduling failures
kubectl describe pod <pending-pod> | grep -A20 Events

# Check scheduler logs for decisions
kubectl logs -n kube-system kube-scheduler-<node> | grep my-pod

# Temporarily prevent scheduling on a node (cordon)
kubectl cordon node-2

# Allow scheduling again
kubectl uncordon node-2
```

■ Key Insight

The scheduler only makes decisions once per pod. After placement, it never moves running pods unless evicted or drained.

Q52

Scheduling &
Resource
Management

Why do Pods remain in Pending state?

- Insufficient CPU or memory: no node has enough resources for the pod's requests.
- Taint without matching toleration: all nodes are tainted and the pod has no matching toleration.
- Node selector or affinity mismatch: pod requires labels no current node has.
- PVC not bound: pod requires storage that hasn't been provisioned yet.
- All nodes are NotReady: cluster-wide problem — check node status.
- ResourceQuota exceeded: namespace quota full — pod would exceed namespace limit.
- PodSecurity Admission blocking: pod spec violates the namespace security policy.

■ Example — YAML / Commands

```
# FIRST: read the scheduling failure reason
kubectl describe pod <pending-pod>
# Scroll to Events section – it says exactly why
# Check node capacity
kubectl describe nodes | grep -A6 'Allocated resources'
# Check namespace quota
kubectl describe quota -n my-namespace
# Check if nodes have taints
kubectl get nodes -o custom-columns=NAME:.metadata.name,TAINTS:.spec.taints
# Check PVC status
kubectl get pvc -n my-namespace
```

■ Key Insight

`kubectl describe pod <pending-pod>` Events section always tells you WHY. Read it first before guessing.

Q53

Scheduling &
Resource
Management

What is Pod Disruption Budget?

- PDB sets the minimum number of pods that must be available during voluntary disruptions.
- Voluntary disruptions: node drain, cluster upgrade, rolling update, manual pod deletion.
- minAvailable: at least N pods must remain running. Can be an absolute number or percentage.
- maxUnavailable: at most N pods can be simultaneously down. Also absolute or percentage.
- Cluster operations (drain/upgrade) wait for PDB constraints before evicting pods.
- PDB protects against accidental complete outage during maintenance. Essential for stateful apps.

■ Example — YAML / Commands

```
# PDB: always keep at least 2 pods running
apiVersion: policy/v1
kind: PodDisruptionBudget
metadata:
  name: my-app-pdb
  namespace: production
spec:
  minAvailable: 2
  selector:
    matchLabels:
      app: my-app
# Or using maxUnavailable
spec:
  maxUnavailable: 1
  selector:
    matchLabels:
      app: my-app
# Check PDB status
kubectl get pdb -n production
kubectl describe pdb my-app-pdb
```

■ Key Insight

Always create PDBs for production Deployments. Without them, a node drain can take all replicas offline simultaneously.

Q54

Scheduling &
Resource
Management

What is priority class?

- PriorityClass assigns an integer priority to pods — higher value = higher scheduling priority.
- When cluster is under resource pressure, higher-priority pods preempt (evict) lower-priority ones.
- Built-in system classes: system-cluster-critical (2000000000), system-node-critical (2000001000).
- Custom classes: define your own tiers — critical (1000000), high (100000), default (0).
- Preemption: scheduler evicts lower-priority pods to make room for high-priority ones.
- Use carefully — high priority classes can cause churn by continuously evicting other workloads.

■ Example — YAML / Commands

```
# Create a PriorityClass
apiVersion: scheduling.k8s.io/v1
kind: PriorityClass
metadata:
  name: critical-apps
  value: 1000000
  globalDefault: false
  description: 'For critical production services'
---
# Use in pod spec
spec:
  priorityClassName: critical-apps
# List all priority classes
kubectl get priorityclass
# Check pod priority
kubectl get pod my-pod -o jsonpath='{.spec.priority}'
```

■ Key Insight

Assign high priority only to truly critical services. Over-using priority classes causes eviction storms in clusters.

Q55

Scaling &
Availability

What is HPA?

- HPA (Horizontal Pod Autoscaler) automatically scales the number of pod replicas based on metrics.
- Supported metrics: CPU utilisation, memory utilisation, custom metrics (via Prometheus Adapter), external metrics.
- Requires metrics-server installed for default CPU/memory metrics. No metrics-server = HPA won't work.
- Pods MUST have resource requests set — HPA calculates utilisation as actual/request percentage.
- Scale-up is fast (default 15s sync). Scale-down is slow (5 minute stabilisation) to prevent flapping.
- minReplicas and maxReplicas bound the scaling range — HPA never goes outside these limits.

■ Example — YAML / Commands

```
# Quick HPA via CLI
kubectl autoscale deployment my-app \
  --cpu-percent=70 --min=2 --max=20

# Or via YAML
apiVersion: autoscaling/v2
kind: HorizontalPodAutoscaler
metadata:
  name: my-app-hpa
spec:
  scaleTargetRef:
    apiVersion: apps/v1
    kind: Deployment
    name: my-app
  minReplicas: 2
  maxReplicas: 20
  metrics:
  - type: Resource
    resource:
      name: cpu
      target:
        type: Utilization
        averageUtilization: 70
kubectl get hpa --watch
```

■ Key Insight

HPA without resource requests is broken — it cannot calculate utilisation %. Always set requests before enabling HPA.

Q56

Scaling &
Availability

How does HPA work internally?

- HPA controller runs a sync loop every 15 seconds (configurable with `--horizontal-pod-autoscaler-sync-period`).
- Fetches current metrics from Metrics API (metrics-server) or Custom Metrics API (Prometheus Adapter).
- Calculates desired replicas: $\text{ceil}(\text{currentReplicas} * \text{currentMetric} / \text{targetMetric})$.
- Example: 4 replicas at 90% CPU, target 70%: $\text{ceil}(4 * 90/70) = \text{ceil}(5.14) = 6$ replicas.
- Scale-up: applies immediately (with `maxReplicas` cap).
- Scale-down: applies after stabilisation window (default 5 min) to prevent constant up-down flapping.

■ Example — YAML / Commands

```
# Watch HPA make decisions in real time
kubectl get hpa my-app-hpa -w
# See current HPA metrics and targets
kubectl describe hpa my-app-hpa
# HPA logs (check if metrics are being fetched)
kubectl logs -n kube-system deploy/metrics-server
# Load test to trigger HPA scale-up
kubectl run load-gen --image=busybox --rm -it \
-- /bin/sh -c 'while true; do wget -q -O- http://my-app; done'
# Metrics server must be installed
kubectl top pods
```

■ Key Insight

HPA needs both metrics-server and resource requests. Missing either one and HPA shows Unknown in `kubectl get hpa`.

Q57

Scaling &
Availability

What is VPA?

- VPA (Vertical Pod Autoscaler) automatically adjusts CPU/memory requests and limits based on actual usage.
- Analyses historical and real-time usage, recommends or applies better resource values.
- Three modes: Off (recommend only, no changes), Initial (set on pod creation only), Auto (recreate pods).
- In Auto mode, VPA kills pods and recreates them with new resource values — brief disruption.
- Best used for: right-sizing resources when you don't know correct values, batch workloads.
- Do NOT use HPA and VPA (CPU/memory) together on the same Deployment — they conflict with each other.

■ Example — YAML / Commands

```
# Install VPA
git clone https://github.com/kubernetes/autoscaler.git
./autoscaler/vertical-pod-autoscaler/hack/vpa-up.sh
# VPA in recommendation mode (safe start)
apiVersion: autoscaling.k8s.io/v1
kind: VerticalPodAutoscaler
metadata:
  name: my-app-vpa
spec:
  targetRef:
    apiVersion: apps/v1
    kind: Deployment
    name: my-app
  updatePolicy:
    updateMode: 'Off' # recommend only
# Read recommendations
kubectl describe vpa my-app-vpa
```

■ Key Insight

Start VPA in Off mode for 1-2 weeks, read recommendations, then set those as static requests. Use VPA data, not guesses.

Q58

Scaling &
Availability

Difference between HPA and VPA?

- HPA: scales OUT — adds more pod replicas. Better for stateless apps with variable traffic.
- VPA: scales UP — gives each pod more CPU/memory. Better for right-sizing resource requests.
- HPA reacts fast to traffic spikes. VPA reacts slowly based on historical usage patterns.
- HPA needs metrics-server and resource requests. VPA needs VPA add-on installed.
- Using both together on CPU/memory = conflict. HPA changes replica count, VPA changes per-pod resources.
- Safe combination: HPA on CPU (scale out), VPA on memory only (no CPU, just memory right-sizing).

■ Example — YAML / Commands

```
# HPA: scales replicas
kubectl get hpa

# Shows: REPLICAS column changes
# VPA: shows recommended resources
kubectl describe vpa my-app-vpa

# Shows: Lower Bound, Target, Upper Bound, Uncapped Target
# Safe combo: HPA for CPU, VPA for memory only
# In VPA YAML, exclude CPU from recommendations:
resourcePolicy:
containerPolicies:
- containerName: app
controlledResources: ['memory'] # VPA manages memory only
kubectl get hpa,vpa -n production
```

■ Key Insight

HPA = horizontal (more pods). VPA = vertical (bigger pods). They serve different purposes. Combine carefully.

Q59

Scaling &
Availability

What is Cluster Autoscaler?

- Cluster Autoscaler (CA) automatically adds or removes nodes based on pod demand.
- Scale UP: triggered when pods are Pending because no node has enough resources for them.
- Scale DOWN: triggered when a node has been underutilised for 10+ minutes and all its pods can fit elsewhere.
- Works with cloud provider node groups: AWS Auto Scaling Groups, GCP Managed Instance Groups, Azure VMSS.
- CA respects PodDisruptionBudgets during scale-down — won't evict pods if PDB is violated.
- CA + HPA = fully elastic cluster: HPA scales pods, CA scales nodes. The optimal production setup.

■ Example — YAML / Commands

```
# Install Cluster Autoscaler on EKS (via Helm)
helm repo add autoscaler https://kubernetes.github.io/autoscaler
helm install cluster-autoscaler autoscaler/cluster-autoscaler \
--set autoDiscovery.clusterName=my-cluster \
--set awsRegion=us-east-1 \
-n kube-system
# Check CA decisions in logs
kubectl logs -n kube-system \
deploy/cluster-autoscaler | grep -E 'scale|node'
# Annotate node group for CA discovery (AWS)
# Tag the ASG with:
# k8s.io/cluster-autoscaler/enabled: 'true'
# k8s.io/cluster-autoscaler/<cluster-name>: owned
```

■ Key Insight

CA scale-up is fast (1-2 min for new node). Scale-down takes 10+ min to avoid flapping. Size your min replicas accordingly.

Q60

Scaling &
Availability

How do you scale manually?

- `kubectl scale`: directly sets replica count on a Deployment, ReplicaSet, or StatefulSet.
- `kubectl edit`: opens the resource in your editor — change replicas and save.
- `kubectl patch`: programmatic way to change a specific field — useful in scripts and CI/CD.
- If HPA is enabled, manual scaling is overridden by HPA on the next sync cycle (15s).
- To manual scale with HPA: update HPA `minReplicas` instead, or delete the HPA first.
- For node scaling: `aws eks update-nodegroup-config` or `kubectl-node-management` tool.

■ Example — YAML / Commands

```
# Scale Deployment to 5 replicas
kubectl scale deployment my-app --replicas=5
# Scale StatefulSet
kubectl scale statefulset my-db --replicas=3
# Edit directly
kubectl edit deployment my-app
# Change spec.replicas and save
# Patch (good for scripts)
kubectl patch deployment my-app \
-p '{"spec":{"replicas":5}}'
# Scale EKS node group
aws eks update-nodegroup-config \
--cluster-name my-cluster \
--nodegroup-name workers \
--scaling-config desiredSize=6
```

■ Key Insight

If HPA is enabled, manual `kubectl scale` is temporary — HPA will override it in 15 seconds. Update HPA `min` instead.

Q61

Scaling &
Availability

What is a rolling update?

- Rolling update replaces old pods with new ones gradually — traffic is never completely interrupted.
- Old pods keep receiving traffic until new pods pass their readiness probe and are marked Ready.
- maxSurge: how many extra pods can exist ABOVE the desired replica count during update.
- maxUnavailable: how many pods can be BELOW the desired count during update.
- For zero downtime: maxSurge: 1, maxUnavailable: 0 — always at least desired number of ready pods.
- If new pods fail readiness checks, the rollout stalls — old pods keep running, new version doesn't propagate.

■ Example — YAML / Commands

```
spec:
  strategy:
    type: RollingUpdate
  rollingUpdate:
    maxSurge: 1 # allow 1 extra pod above desired
    maxUnavailable: 0 # never reduce below desired
  template:
    spec:
      containers:
        - name: app
          image: my-app:v3
      readinessProbe:
        httpGet:
          path: /health
          port: 8080
        initialDelaySeconds: 10
        periodSeconds: 5
    # Trigger the rolling update
  kubectl set image deployment/my-app app=my-app:v3
  kubectl rollout status deployment/my-app
```

■ Key Insight

A stalled rollout is a safety net — bad version can't progress. Fix the issue and the rollout automatically resumes.

Q62

Scaling &
Availability

What is zero-downtime deployment?

- Zero-downtime means users experience no errors or interruptions during a deployment update.
- Requires: maxUnavailable: 0 so pods are never terminated before replacements are ready.
- Requires: readinessProbe so K8s knows when a new pod is actually ready to serve traffic.
- Requires: preStop lifecycle hook with a sleep — lets in-flight requests complete before pod is killed.
- Requires: terminationGracePeriodSeconds long enough for the app to finish its shutdown gracefully.
- Requires: PodDisruptionBudget to prevent maintenance operations from breaking availability.

■ Example — YAML / Commands

```
spec:
  strategy:
    type: RollingUpdate
    rollingUpdate:
      maxSurge: 1
      maxUnavailable: 0
  template:
    spec:
      terminationGracePeriodSeconds: 60
      containers:
        - name: app
          image: my-app:v3
          lifecycle:
            preStop:
              exec:
                command: ['/bin/sh', '-c', 'sleep 5']
          readinessProbe:
            httpGet:
              path: /health
              port: 8080
            initialDelaySeconds: 10
            failureThreshold: 3
```

■ Key Insight

Zero-downtime = maxUnavailable:0 + readinessProbe + preStop hook. All three together. Missing one and you'll get 502s.

Q63

Security

What is RBAC?

- RBAC (Role-Based Access Control) controls who can do what with which resources in the cluster.
- Key objects: Role/ClusterRole (define allowed actions), RoleBinding/ClusterRoleBinding (who gets them).
- Verbs: get, list, watch, create, update, patch, delete, deletecollection, exec.
- Deny by default: if there is no RBAC rule allowing an action, it is denied.
- RBAC applies to: human users, service accounts (pod identities), external tools (Helm, ArgoCD).
- Audit regularly: over-permissive service accounts are the #1 Kubernetes attack vector.

■ Example — YAML / Commands

```
# Check what permissions you have
kubectl auth can-i list pods
kubectl auth can-i delete nodes
# Check permissions as another user
kubectl auth can-i list secrets \
--as=dev-user -n production
# Check SA permissions
kubectl auth can-i get pods \
--as=system:serviceaccount:default:my-sa
# List all ClusterRoleBindings
kubectl get clusterrolebindings
# Find who has cluster-admin
kubectl get clusterrolebindings -o json | \
jq '.items[] | select(.roleRef.name=="cluster-admin")'
```

■ Key Insight

Least privilege: every user and service account gets the minimum permissions needed. Audit RBAC quarterly.

Q64

Security

Difference between Role and ClusterRole?

- Role: namespace-scoped — grants access to resources within ONE namespace only.
- ClusterRole: cluster-scoped — grants access across ALL namespaces or to cluster-level resources.
- Cluster-level resources (nodes, PVs, namespaces) can ONLY be granted via ClusterRole.
- RoleBinding: binds a Role or ClusterRole to a subject within ONE namespace.
- ClusterRoleBinding: binds a ClusterRole to a subject across ALL namespaces.
- Common pattern: one ClusterRole with read access, then RoleBindings per namespace for scoped access.

■ Example — YAML / Commands

```
# Role: read pods in 'dev' namespace only
apiVersion: rbac.authorization.k8s.io/v1
kind: Role
metadata:
  name: pod-reader
  namespace: dev
rules:
- apiGroups: ['']
  resources: ['pods', 'pods/log']
  verbs: ['get', 'list', 'watch']
---
apiVersion: rbac.authorization.k8s.io/v1
kind: RoleBinding
metadata:
  name: dev-pod-reader
  namespace: dev
subjects:
- kind: User
  name: alice
  roleRef:
    kind: Role
    name: pod-reader
  apiGroup: rbac.authorization.k8s.io
```

■ Key Insight

If you need to grant access to nodes, PVs, or all namespaces — use ClusterRole. For team namespace access — use Role.

Q65

Security

What is a Service Account?

- Service Account is an identity for pods to authenticate to the Kubernetes API server.
- Every namespace has a 'default' service account — pods use it unless you specify another.
- SA credentials are auto-mounted at `/var/run/secrets/kubernetes.io/serviceaccount/token` inside the pod.
- The token is a JWT — kube-apiserver validates it. RBAC then checks what the SA is allowed to do.
- K8s 1.22+: tokens are short-lived (1 hour) and auto-rotated via the TokenRequest API.
- IRSA (in EKS): annotate a SA with an IAM role ARN — pods get AWS credentials without any stored keys.

■ Example — YAML / Commands

```
# Create a dedicated service account
kubectl create serviceaccount my-app-sa -n production
# Use it in a deployment
spec:
  serviceAccountName: my-app-sa
  automountServiceAccountToken: false # disable if not using K8s API
# IRSA: link SA to AWS IAM role
kubectl annotate serviceaccount my-app-sa \
  eks.amazonaws.com/role-arn=arn:aws:iam::123:role/MyAppRole \
  -n production
# Check what SA a pod is using
kubectl get pod my-pod -o jsonpath='{.spec.serviceAccountName}'
```

■ Key Insight

Always create dedicated SAs per app with minimal RBAC. Disable `automountServiceAccountToken` if the app doesn't call the K8s API.

Q66

Security

How do Pods authenticate to the API server?

- Pods use their Service Account JWT token — auto-mounted at `/var/run/secrets/kubernetes.io/serviceaccount/`.
- The token is presented as a Bearer token in the Authorization HTTP header to the API server.
- kube-apiserver validates the token using its signing keys. Then RBAC checks if the action is allowed.
- From K8s 1.22+: tokens are bound to specific pod, namespace, and SA — expire after 1 hour, auto-rotated.
- External tools (kubectl, Helm, ArgoCD) use kubeconfig with user certificates or cloud provider tokens.
- For AWS: IRSA links a SA to an IAM role — pods get AWS SDK credentials without storing any access keys.

■ Example — YAML / Commands

```
# Inspect the mounted token inside a pod
kubectl exec -it my-pod -- \
cat /var/run/secrets/kubernetes.io/serviceaccount/token
# Decode the JWT to see claims
TOKEN=$(kubectl exec my-pod -- \
cat /var/run/secrets/kubernetes.io/serviceaccount/token)
echo $TOKEN | cut -d. -f2 | base64 -d | python3 -m json.tool
# Check token expiry
kubectl get pod my-pod -o yaml | grep expirationSeconds
# Manually create a token for a SA (debugging)
kubectl create token my-sa -n production --duration=1h
```

■ Key Insight

Short-lived tokens (1h) are much safer than old static long-lived tokens. K8s 1.22+ does this automatically.

Q67

Security

What are Network Policies?

- NetworkPolicy controls which pods can communicate with which other pods and external endpoints.
- Default: all traffic is allowed. Apply a NetworkPolicy to start restricting ingress and/or egress.
- Requires a CNI that enforces them — Calico or Cilium. Flannel does NOT enforce NetworkPolicies.
- podSelector selects which pods the policy applies to. Empty podSelector = applies to all pods in namespace.
- Best practice: start with a default deny-all policy, then whitelist only necessary traffic flows.
- Policy is namespace-scoped — you need policies in each namespace.

■ Example — YAML / Commands

```
# Default deny-all for a namespace
apiVersion: networking.k8s.io/v1
kind: NetworkPolicy
metadata:
  name: deny-all
  namespace: production
spec:
  podSelector: {}
  policyTypes: [Ingress, Egress]
---
# Allow frontend to reach backend on port 8080
kind: NetworkPolicy
metadata:
  name: allow-frontend-to-backend
  namespace: production
spec:
  podSelector:
    matchLabels:
      app: backend
  ingress:
    - from:
      - podSelector:
          matchLabels:
            app: frontend
      ports:
        - port: 8080
```

■ Key Insight

Apply deny-all first, then whitelist. If you apply only allow policies without deny-all, everything else stays open.

Q68

Security

How do you secure Secrets?

- Default K8s Secrets are base64 only — NOT encrypted. Anyone with etcd access can decode them.
- Enable Encryption at Rest: configure EncryptionConfiguration on the API server for AES-GCM or AES-CBC.
- Use External Secrets Operator (ESO) + AWS Secrets Manager or HashiCorp Vault for real secret management.
- ESO syncs secrets from external vaults into K8s Secrets automatically — no manual handling.
- RBAC: restrict who can get/list Secrets. Developers should not need to read production credentials.
- Never commit secrets to git. Use git-crypt, SOPS, or Sealed Secrets for GitOps workflows.

■ Example — YAML / Commands

```
# Install External Secrets Operator
helm install external-secrets external-secrets/external-secrets -n external-secrets

# SecretStore: point to AWS Secrets Manager
apiVersion: external-secrets.io/v1beta1
kind: SecretStore
metadata:
  name: aws-sm
spec:
  provider:
    aws:
      service: SecretsManager
      region: us-east-1
---
# ExternalSecret: pull and sync a secret
apiVersion: external-secrets.io/v1beta1
kind: ExternalSecret
metadata:
  name: db-creds
spec:
  secretStoreRef:
    name: aws-sm
  target:
    name: db-secret
  data:
    - secretKey: DB_PASSWORD
  remoteRef:
    key: prod/db/password
```

■ Key Insight

ESO + AWS Secrets Manager = secrets never stored in git, auto-rotated, RBAC controlled. The production standard.

Q69

Security

What is Pod Security Policy / Pod Security Standards?

- PSP (PodSecurityPolicy) was deprecated in K8s 1.21 and removed in 1.25 — do not use it.
- Pod Security Admission (PSA) is the replacement — built into K8s 1.25+, no extra install needed.
- Three security levels: Privileged (no restrictions), Baseline (prevents known privilege escalation), Restricted (hardened).
- Applied per namespace via labels — each namespace can have enforce, audit, or warn modes.
- enforce: pod is rejected if it violates the policy. audit: logged but allowed. warn: warning shown.
- Start with baseline, move to restricted for internet-facing workloads.

■ Example — YAML / Commands

```
# Label namespace to enforce restricted policy
kubectl label namespace production \
pod-security.kubernetes.io/enforce=restricted \
pod-security.kubernetes.io/enforce-version=latest
# Warn on baseline violations in dev namespace
kubectl label namespace dev \
pod-security.kubernetes.io/warn=baseline
# Test if a pod would pass a policy
kubectl apply --dry-run=server -f pod.yaml
# Check what policy a namespace has
kubectl get namespace production -o yaml | grep pod-security
```

■ Key Insight

Move all production namespaces to baseline minimum. Apply restricted for internet-facing services. No privileged containers ever.

Q70

Security

How do you secure etcd?

- Enable TLS: etcd must only accept TLS connections. Configure --cert-file, --key-file, --trusted-ca-file.
- Mutual TLS: enable --client-cert-auth so etcd rejects clients without a valid certificate.
- Restrict access: firewall etcd port (2379) — only kube-apiserver should reach etcd. No other services.
- Encryption at rest: use EncryptionConfiguration on the API server — AES-GCM for etcd Secrets.
- Regular backups: snapshot daily, store in an encrypted S3 bucket, test restores monthly.
- Dedicated nodes: run etcd on separate nodes, not co-located with application workloads.

■ Example — YAML / Commands

```
# Enable etcd encryption at rest (API server flag)
--encryption-provider-config=/etc/kubernetes/encryption.yaml
# encryption.yaml
apiVersion: apiserver.config.k8s.io/v1
kind: EncryptionConfiguration
resources:
- resources: [secrets]
providers:
- aescbc:
keys:
- name: key1
secret: <base64-32-byte-key>
- identity: {}
# Verify secrets are encrypted in etcd
ETCDCTL_API=3 etcdctl get /registry/secrets/default/my-secret \
--print-value-only | hexdump -C | head
# Should show garbled (encrypted) data
```

■ Key Insight

Etcd contains ALL cluster secrets. Encrypt it at rest. Firewall it. Back it up. These are non-negotiable in production.

Q71

Observability

How do you monitor Kubernetes?

- Prometheus + Grafana is the production standard — kube-prometheus-stack Helm chart installs everything.
- metrics-server: lightweight in-memory metrics for kubectl top and HPA. Not for long-term storage.
- kube-state-metrics: exports cluster state metrics — pod restarts, deployment replicas, PVC status.
- Node Exporter: OS-level metrics per node — CPU, memory, disk, network, filesystem.
- Alertmanager: routes alerts to PagerDuty (P1), Slack (P2/P3), email. Define SLO-based alerts.
- Grafana dashboards: import K8s community dashboards (ID 315, 6417) for instant cluster visibility.

■ Example — YAML / Commands

```
# Install full observability stack
helm repo add prometheus-community \
https://prometheus-community.github.io/helm-charts
helm install monitoring \
prometheus-community/kube-prometheus-stack \
-n monitoring --create-namespace \
-f monitoring-values.yaml
# Quick checks
kubectl top nodes
kubectl top pods -A --sort-by=cpu
kubectl top pods -A --sort-by=memory
# Access Grafana (port-forward)
kubectl port-forward -n monitoring \
svc/monitoring-grafana 3000:80
```

■ Key Insight

Install monitoring before going live — not after the first incident. Grafana + Prometheus takes 10 min with Helm.

Q72

Observability

How do you collect logs?

- `kubecttl logs`: good for quick pod-level debugging. Not persistent — lost when pod is deleted.
- `Fluent Bit` (`DaemonSet`): lightweight log forwarder on every node. Collects from `/var/log/containers/`.
- Forward to: `Elasticsearch` + `Kibana` (`ELK`), `Grafana Loki` (lighter weight), `CloudWatch`, `Splunk`.
- `Loki` + `Grafana`: log aggregation without indexing — much cheaper than `ELK` at scale. Growing fast.
- Structured logging (`JSON`): makes log filtering and querying much more powerful.
- Never rely on `kubecttl logs` in production — pods are ephemeral, logs disappear when pods die.

■ Example — YAML / Commands

```
# Quick pod log commands
kubecttl logs my-pod
kubecttl logs my-pod -c my-container # multi-container pod
kubecttl logs my-pod --previous # last crashed container
kubecttl logs my-pod -f --tail=100 # follow last 100 lines
kubecttl logs -l app=my-app --all-containers=true

# Install Loki logging stack
helm repo add grafana https://grafana.github.io/helm-charts
helm install loki grafana/loki-stack \
-n monitoring \
--set promtail.enabled=true \
--set grafana.enabled=false
```

■ Key Insight

Use `kubecttl logs` for debugging, not monitoring. In production, logs must go to a centralised system — pods die, logs shouldn't.

Q73

Observability

What is metrics-server?

- metrics-server is a cluster-wide aggregator of resource usage metrics — CPU and memory only.
- Collects metrics from kubelet's Summary API on each node every 60 seconds.
- Stores metrics in memory only — no persistence. Use Prometheus for historical data.
- Required for: kubectl top nodes, kubectl top pods, HPA to work.
- If metrics-server is not installed: kubectl top returns error and HPA shows Unknown.
- Lightweight: single deployment, minimal resources. Should be installed in every cluster.

■ Example — YAML / Commands

```
# Install metrics-server
kubectl apply -f
https://github.com/kubernetes-sigs/metrics-server/releases/latest/download/components.yaml
# Verify it is running
kubectl get deployment metrics-server -n kube-system
# Use it
kubectl top nodes
kubectl top pods
kubectl top pods -A --sort-by=memory
# If in private cluster (no public API), add this flag:
# --kubelet-insecure-tls
# Edit the deployment args:
kubectl edit deployment metrics-server -n kube-system
```

■ Key Insight

metrics-server is the minimum monitoring requirement. Install it first. Without it, kubectl top and HPA are both broken.

Q74

Observability

What monitoring tools have you used — Prometheus, Grafana, others?

- Prometheus: pull-based metrics collection with TSDB storage. PromQL for queries and alerting.
- Grafana: dashboards and visualisation. Connects to Prometheus, Loki, Tempo, CloudWatch.
- Alertmanager: handles Prometheus alerts — routes to Slack, PagerDuty, email by severity.
- Loki: log aggregation (like Prometheus but for logs). Cheap — indexes labels not log content.
- Jaeger/Tempo: distributed tracing — finds slow spans across microservices.
- Datadog / New Relic: all-in-one SaaS alternatives. Great out-of-box, expensive at scale.

■ Example — YAML / Commands

```
# PromQL: useful production queries
# CPU throttling ratio (should be near 0)
rate(container_cpu_cfs_throttled_seconds_total[5m])
/ rate(container_cpu_cfs_periods_total[5m])
# Memory usage vs limit
container_memory_working_set_bytes
/ container_spec_memory_limit_bytes
# Pod restart rate
rate(kube_pod_container_status_restarts_total[1h]) > 0
# Nodes with >90% memory
node_memory_MemAvailable_bytes / node_memory_MemTotal_bytes < 0.1
```

■ Key Insight

PromQL is the skill that separates senior DevOps from mid-level. Know the 5-10 most important cluster queries cold.

Q75

Observability

How do you debug cluster-level issues?

- Start broad, then narrow: cluster → node → pod → container → application.
- Check node health first: `kubectl get nodes`. NotReady nodes cause widespread pod failures.
- Check system pods: `kubectl get pods -n kube-system`. CoreDNS down = all DNS fails. kube-proxy down = Service issues.
- Check events: `kubectl get events -A --sort-by='.lastTimestamp'` — see the most recent cluster-wide events.
- Check resource pressure: `kubectl top nodes`. Full nodes cause evictions and scheduling failures.
- Use describe: `kubectl describe node/pod/deployment` — Events section always tells you what is wrong.

■ Example — YAML / Commands

```
# Golden debugging sequence
kubectl get nodes -o wide
kubectl get pods -A | grep -v Running
kubectl get events -A --sort-by='.lastTimestamp' | tail -30
kubectl top nodes
kubectl top pods -A --sort-by=cpu | head -15
# Describe a NotReady node
kubectl describe node <node>
# Check: Conditions, Capacity, Allocatable, Events
# Check system pod logs
kubectl logs -n kube-system <coredns-pod> --tail=50
kubectl logs -n kube-system <kube-proxy-pod> --tail=50
# Check kubelet on node (SSH required)
journalctl -u kubelet -n 100 --no-pager
```

■ Key Insight

Start with `kubectl get nodes` and `kubectl get events`. 80% of cluster issues are visible in these two commands.

Q76

CI/CD &
Packaging
(Helm)

What is Helm?

- Helm is the package manager for Kubernetes — like apt for Ubuntu or npm for Node.js.
- Bundles K8s manifests (Deployment, Service, Ingress, etc.) into reusable, versioned packages called Charts.
- Manages the full lifecycle: install, upgrade, rollback, uninstall of applications.
- Helm uses Go templating — values.yaml provides defaults, and you override at deploy time.
- A Helm Release is one installed instance of a chart in the cluster.
- Helm keeps release history — you can rollback to any previous revision in seconds.

■ Example — YAML / Commands

```
# Install Helm (if not installed)
curl https://raw.githubusercontent.com/helm/helm/main/scripts/get-helm-3 | bash
# Add a chart repository
helm repo add bitnami https://charts.bitnami.com/bitnami
helm repo update
# Search for charts
helm search repo nginx
# Install a release
helm install my-nginx bitnami/nginx \
--namespace production \
--create-namespace
# List all releases
helm list -A
```

■ Key Insight

Helm is the de facto K8s deployment tool. If your team isn't using Helm, you're manually managing YAML chaos.

Q77

CI/CD &
Packaging
(Helm)

What are Helm charts?

- A Helm chart is a directory of files that describe a K8s application — like a package/module.
- Chart.yaml: chart metadata — name, version, appVersion, description.
- values.yaml: default configuration values that users can override at install time.
- templates/: K8s YAML templates using Go template syntax with {{ .Values.xxx }} placeholders.
- _helpers.tpl: shared template functions and name helpers.
- charts/: sub-charts (chart dependencies) — e.g. your app chart depends on a PostgreSQL chart.

■ Example — YAML / Commands

```
# Create a new chart scaffold
helm create my-app
# Chart structure
my-app/
Chart.yaml # metadata
values.yaml # default values
templates/
deployment.yaml # uses {{ .Values.image.tag }}
service.yaml
ingress.yaml
_helpers.tpl # shared helpers
charts/ # dependencies
# Render templates locally (no cluster needed)
helm template my-release ./my-app -f prod-values.yaml
# Lint the chart
helm lint ./my-app
```

■ Key Insight

Always helm lint + helm template in CI before merging chart changes. Catch errors before they hit production.

Q78

CI/CD &
Packaging
(Helm)

What is a Helm release?

- A Helm release is a named, deployed instance of a chart in the cluster.
- Same chart can be deployed multiple times as different releases: my-app-dev, my-app-staging, my-app-prod.
- Each release has a revision history — every install, upgrade, or rollback is a new revision.
- Release state is stored as Secrets in the target namespace — helm reads them to manage the release.
- `helm list` shows all releases: name, namespace, revision, status, chart version, app version.
- `helm history` shows all revisions of a release — useful before rollback.

■ Example — YAML / Commands

```
# List all releases across all namespaces
helm list -A

# Show detailed release info
helm status my-app -n production

# Show full revision history of a release
helm history my-app -n production

# Get the values used in the current release
helm get values my-app -n production

# Get all Kubernetes manifests of a release
helm get manifest my-app -n production

# Release state is stored as secrets
kubectl get secrets -n production | grep helm
```

■ Key Insight

`helm history` is your friend before any rollback. Always check what changed between revisions before reverting.

Q79

CI/CD &
Packaging
(Helm)

How do you deploy an app using Helm?

- Step 1: helm repo add — add the chart repository.
- Step 2: helm repo update — refresh the local chart cache.
- Step 3: helm show values <chart> — inspect default values to know what to override.
- Step 4: create your own values file (prod-values.yaml) with environment-specific overrides.
- Step 5: helm install (first time) or helm upgrade --install (idempotent — works for both).
- Step 6: helm status to verify, kubectl get pods to watch pods start.

■ Example — YAML / Commands

```
# Full deploy workflow
helm repo add ingress-nginx https://kubernetes.github.io/ingress-nginx
helm repo update
# Inspect defaults
helm show values ingress-nginx/ingress-nginx > defaults.yaml
# Install with custom values
helm upgrade --install ingress-nginx ingress-nginx/ingress-nginx \
--namespace ingress-nginx \
--create-namespace \
-f prod-values.yaml \
--atomic \
--timeout 5m
# Verify
helm status ingress-nginx -n ingress-nginx
kubectl get pods -n ingress-nginx
```

■ Key Insight

Use --atomic in CI/CD — if any pod fails to become ready, Helm automatically rolls back to the previous release.

Q80

CI/CD &
Packaging
(Helm)

How do you rollback a Helm release?

- helm rollback reverts a release to a previous revision — fast because K8s manifests are re-applied.
- Check helm history before rolling back — find the revision you want to go back to.
- helm rollback without a revision number goes back to the previous revision (one step back).
- Rollback creates a new revision number — does not erase history.
- After rollback: helm status and kubectl get pods to confirm the previous version is running.
- For failed releases: if --atomic was used, rollback happens automatically on failure.

■ Example — YAML / Commands

```
# See revision history
helm history my-app -n production
# Rollback to previous revision
helm rollback my-app -n production
# Rollback to specific revision
helm rollback my-app 3 -n production
# Watch rollback progress
helm status my-app -n production
kubectl rollout status deployment/my-app -n production
# After rollback, verify
kubectl get pods -n production
kubectl describe deployment my-app -n production | grep Image
```

■ Key Insight

Helm rollback is faster than re-deploying. Keep at least 10 revisions (--history-max=10) so you can always roll back.

Q81

Troubleshooting

Pod in CrashLoopBackOff — step-by-step debug.

- CrashLoopBackOff: container crashes immediately, K8s restarts with exponential backoff (10s, 20s, 40s... max 5min).
- Step 1: `kubectl describe pod` — read Events section. It shows last exit code and crash reason.
- Step 2: `kubectl logs <pod> --previous` — MOST USEFUL. Gets logs from the container before it crashed.
- Step 3: Check exit code in Last State. Code 137 = OOMKilled (memory limit hit). Code 1 = app crash.
- Step 4: Check liveness probe config — too-early `initialDelaySeconds` kills the pod before it's ready.
- Step 5: Run a debug pod with the same image and sleep — `kubectl run debug --image=my-app -- sleep 3600`, then `exec in`.

■ Example — YAML / Commands

```
# Step-by-step crash debug
kubectl get pods # check status
kubectl describe pod my-pod # events + last state
kubectl logs my-pod --previous # crash logs
# Check for OOMKill
kubectl get pod my-pod -o jsonpath=\
' {.status.containerStatuses[0].lastState.terminated} '
# Exec into a briefly-alive pod
kubectl exec -it my-pod -- /bin/sh
# Debug with same image
kubectl run debug --image=my-app:v1 \
--restart=Never --command -- sleep 3600
kubectl exec -it debug -- /bin/sh
```

■ Key Insight

`kubectl logs --previous` is always the first command. The crash log tells you what the app said before dying.

Q82

Troubleshooting

Pod stuck in Pending — why and how to fix?

- Pending = API server accepted the pod, but scheduler hasn't placed it on a node yet.
- Can't get logs from a Pending pod — no container is running. Use `kubectl describe` instead.
- Cause 1: Insufficient CPU/memory — no node has enough for the pod's resource requests.
- Cause 2: Taint/toleration mismatch — all nodes are tainted and pod has no matching toleration.
- Cause 3: Node affinity/selector — pod requires labels no node has.
- Cause 4: PVC not bound — StorageClass doesn't exist, or wrong access mode, or provisioner failing.
- Cause 5: ResourceQuota exceeded — namespace is at its CPU/memory limit.

■ Example — YAML / Commands

```
# ALWAYS start here – events tell you exactly why
kubectl describe pod <pending-pod>
# Look at 'Events:' section at bottom
# Check node capacity
kubectl describe nodes | grep -A10 'Allocated resources'
# Check taints on all nodes
kubectl get nodes -o custom-columns=\
'NAME:.metadata.name,TAINTS:.spec.taints'
# Check namespace quota
kubectl describe resourcequota -n my-namespace
# Check PVC status
kubectl get pvc -n my-namespace
kubectl describe pvc <pvc-name>
```

■ Key Insight

`kubectl describe pod <pending>` Events section always tells you WHY in plain English. Read it before anything else.

Q83

Troubleshooting

Service not accessible — debugging steps.

- Step 1: verify Service exists and has correct type, port, and targetPort.
- Step 2: check Endpoints — `kubectl get endpoints <svc>`. EMPTY = selector doesn't match any pod label.
- Step 3: verify pod labels EXACTLY match the Service selector — one typo = zero endpoints.
- Step 4: test from inside the cluster — exec into a pod and curl the ClusterIP or DNS name.
- Step 5: check if pods are actually Ready — not just Running. Failing readiness probe = removed from endpoints.
- Step 6: check NetworkPolicy — maybe traffic from the test pod is blocked.

■ Example — YAML / Commands

```
# Full service debug flow
kubectl get svc my-svc # verify service exists
kubectl get endpoints my-svc # check if pods are connected
kubectl describe svc my-svc # see selector
kubectl get pods -l app=my-app --show-labels # check pod labels
# Test from inside the cluster
kubectl run test --image=curlimages/curl --rm -it \
--restart=Never -- curl http://my-svc:80
# Check if pod is Ready
kubectl get pod my-pod -o jsonpath=\
'{"status.conditions":[{"@.type=="Ready"}].status}'
# Check NetworkPolicy
kubectl get networkpolicy -n my-namespace
```

■ Key Insight

Empty Endpoints is the #1 Service bug. Selector mismatch between Service and pods. Check labels with `--show-labels`.

Q84

Troubleshooting

Ingress not routing traffic — what to check?

- Step 1: verify Ingress Controller pods are running in the ingress-nginx namespace.
- Step 2: `kubectl describe ingress <n>` — look for backend errors or missing service.
- Step 3: verify the backend Service exists and has populated Endpoints.
- Step 4: check DNS — does the Ingress hostname resolve to the Ingress Controller's external IP?
- Step 5: check TLS secret exists and cert is valid — expired cert = HTTPS broken.
- Step 6: check Ingress Controller logs — they show the exact upstream error (502, connection refused).

■ Example — YAML / Commands

```
# Debug Ingress step by step
kubectl get ingress -n production
kubectl describe ingress my-ingress -n production
kubectl get endpoints my-backend-svc -n production
# Check Ingress Controller is running
kubectl get pods -n ingress-nginx
kubectl logs -n ingress-nginx \
deploy/ingress-nginx-controller | tail -50
# Check DNS resolution
nslookup app.example.com
curl -v https://app.example.com
# Check TLS secret
kubectl get secret tls-cert -n production
kubectl describe secret tls-cert -n production
```

■ Key Insight

Ingress not routing? 9 out of 10 times: Ingress Controller not running, or backend Service Endpoints are empty.

Q85

Troubleshooting

Node NotReady — how to diagnose and fix?

- NotReady = node has stopped sending heartbeats to kube-apiserver (every 10 seconds).
- `kubectl describe node <n>` — check Conditions block: MemoryPressure, DiskPressure, PIDPressure, Ready.
- SSH to the node and check kubelet: `systemctl status kubelet`, `journalctl -u kubelet -n 100`.
- Check container runtime: `systemctl status containerd` — if it crashes, kubelet can't manage containers.
- Check disk space: `df -h`. A full disk causes DiskPressure and prevents new pods from starting.
- Check network: can the node reach the API server? Firewall change or routing issue = NotReady.

■ Example — YAML / Commands

```
# Diagnose NotReady from control plane
kubectl describe node <notready-node>
kubectl get events --field-selector involvedObject.name=<node>

# On the node itself (SSH)
systemctl status kubelet
journalctl -u kubelet -n 100 --no-pager
systemctl status containerd
df -h
free -m

# Check API server connectivity from node
curl -k https://<api-server-ip>:6443/healthz

# If safe, drain and replace the node
kubectl drain <node> --ignore-daemonsets --delete-emptydir-data
kubectl delete node <node>
```

■ Key Insight

Node NotReady = kubelet stopped reporting. SSH in immediately. Check kubelet, containerd, disk, and network in that order.

Q86

Troubleshooting

ImagePullBackOff — causes and fixes.

- ImagePullBackOff: kubelet tried to pull the container image and failed. Retrying with exponential backoff.
- Cause 1: wrong image name or tag — typo in spec.containers.image. Check spelling carefully.
- Cause 2: image doesn't exist in the registry — tag was not pushed, or wrong registry URL.
- Cause 3: private registry: missing or wrong imagePullSecret on the pod.
- Cause 4: registry rate limiting — Docker Hub limits unauthenticated pulls (100/6h per IP).
- Cause 5: network issue — node can't reach the registry. Check firewall/security groups/proxies.

■ Example — YAML / Commands

```
# Read the error message first
kubectl describe pod my-pod | grep -A10 Events
# Create image pull secret for private registry
kubectl create secret docker-registry my-pull-secret \
--docker-server=myregistry.io \
--docker-username=user \
--docker-password=mypass \
-n production
# Use it in pod spec
spec:
  imagePullSecrets:
  - name: my-pull-secret
# Test pull manually on the node (SSH)
crictl pull myregistry.io/my-app:v2
# Check if image exists
docker manifest inspect myregistry.io/my-app:v2
```

■ Key Insight

ImagePullBackOff fix: check the image name and tag first (typos are common), then check pull secret and network.

Q87

Troubleshooting

High CPU/memory usage — how to debug?

- Step 1: `kubectl top nodes` — find which node is under pressure.
- Step 2: `kubectl top pods -A --sort-by=cpu` — find which pods are consuming most resources.
- Step 3: check if CPU limits are set — if not, one pod can consume an entire node.
- Step 4: check for CPU throttling — a pod at 10% CPU usage can still be throttled 80% of the time.
- Step 5: check for memory leaks — is pod memory growing continuously? Restart count increasing?
- Step 6: use Prometheus to see trends — sudden spike vs gradual growth requires different fixes.

■ Example — YAML / Commands

```
# Find the top consumers
kubectl top nodes
kubectl top pods -A --sort-by=cpu | head -15
kubectl top pods -A --sort-by=memory | head -15
# Find pods WITHOUT resource limits (dangerous)
kubectl get pods -A -o json | \
jq '.items[] | select(.spec.containers[].resources.limits==null)
| .metadata.name'
# Check CPU throttling (on node)
cat /sys/fs/cgroup/cpu/docker/<container-id>/cpu.stat
# throttled_time near 0 = good
# Check memory growth over time (Prometheus)
# container_memory_working_set_bytes
# Look for a sawtooth pattern = memory leak + OOMKill loop
```

■ Key Insight

CPU throttling is invisible to `kubectl top`. A pod shows 10% CPU but throttled 80% of the time. Check Prometheus metrics.

Q88

Troubleshooting

Logs not showing — what to do?

- `kubectll` logs returns nothing: check if the pod is running and the container has started.
- `CrashLoopBackOff`: use `--previous` flag to get logs from the container before it crashed.
- Pod is Pending: no container is running, no logs exist. Use `kubectll describe` instead.
- App writes to a file not `stdout`: K8s only captures `stdout/stderr`. Exec in and `cat` the file.
- Pod was deleted: logs are gone forever unless you have centralised logging (Loki, ELK, CloudWatch).
- Multi-container pod: specify `-c <container-name>` to get logs from the right container.

■ Example — YAML / Commands

```
# Try all these in order
kubectll get pod my-pod # check if Running
kubectll logs my-pod # current logs
kubectll logs my-pod --previous # logs from crashed container
kubectll logs my-pod --tail=200 -f # follow last 200 lines
kubectll logs my-pod -c my-container # specific container
kubectll logs my-pod --all-containers=true # all containers
# If app writes to a file
kubectll exec -it my-pod -- cat /var/log/app/app.log
kubectll exec -it my-pod -- tail -f /var/log/app/app.log
# Logs disappeared (pod deleted)?
# Check your centralised logging system
# Loki: {namespace='production', app='my-app'}
```

■ Key Insight

Set up centralised logging on day 1. `kubectll` logs disappear when pods die. Production incidents need historical logs.

Q89

Troubleshooting

Deployment stuck — reasons and fixes.

- Deployment stuck during rollout means: new pods are not becoming Ready.
- Reason 1: new pods failing readiness probe — wrong /health path, wrong port, app startup takes too long.
- Reason 2: new image pull failing — ImagePullBackOff on new pods (check image name and registry).
- Reason 3: insufficient resources — new pods Pending because no node has room.
- Reason 4: maxUnavailable:0 + maxSurge:0 (misconfiguration) — nothing can change.
- Fix: rollback immediately, fix the root cause, then redeploy properly.

■ Example — YAML / Commands

```
# Check rollout status
kubectl rollout status deployment/my-app
# If stuck: 'Waiting for deployment... to rollout'
# See new pods and why they're failing
kubectl get pods -l app=my-app
kubectl describe pod <new-failing-pod>
kubectl logs <new-failing-pod>
# Rollback immediately
kubectl rollout undo deployment/my-app
# Watch rollback complete
kubectl rollout status deployment/my-app
# Check current image after rollback
kubectl describe deployment my-app | grep Image
```

■ Key Insight

Stuck rollout is a safety net. Old pods keep running. Rollback fast, fix the root cause, then redeploy properly.

Q90

Troubleshooting

ConfigMap updated but not reflected in pods — why?

- Env var injection: env vars are set at pod startup. Updating the ConfigMap does NOT update running pods.
- Volume-mounted ConfigMap: K8s automatically syncs the file inside the pod (~60 second delay).
- But even with volume mounts: the app must re-read the file — most apps only read config at startup.
- The safest fix for both cases: `kubectl rollout restart` — triggers a rolling pod restart.
- Automation (Helm): add a checksum annotation on the pod template. When CM changes, hash changes → rollout triggered.
- Reloader (3rd party): Stakater Reloader automatically restarts deployments when their ConfigMap/Secret changes.

■ Example — YAML / Commands

```
# Force rolling restart to pick up new ConfigMap
kubectl rollout restart deployment/my-app -n production

# Watch restart progress
kubectl rollout status deployment/my-app

# Verify new config inside pod
kubectl exec -it <new-pod> -- env | grep MY_CONFIG_VAR

# Helm: auto-restart on ConfigMap change
# In deployment.yaml template:
metadata:
  annotations:
    checksum/config: {{ include (print $.Template.BasePath
'/configmap.yaml') . | sha256sum }}

# Install Stakater Reloader (auto-restarts on CM change)
helm install reloader stakater/reloader
```

■ Key Insight

Env vars never hot-reload. Volume mounts sync slowly. Rollout restart is always the safe, reliable fix for CM changes.

Q91

Scenario-Based
Questions

Your app is not accessible externally — end-to-end debug.

- Step 1: check pods — `kubectl get pods`. All Running and Ready? Readiness probe passing?
- Step 2: check Service — `kubectl get svc`. Correct type? EXTERNAL-IP assigned? Endpoints populated?
- Step 3: check Ingress — `kubectl describe ingress`. Correct host and path? Backend Service correct?
- Step 4: check DNS — does the domain resolve to the Ingress Controller external IP? (`nslookup app.example.com`).
- Step 5: check Ingress Controller logs — looking for upstream errors (502, connection refused).
- Step 6: check cloud firewall/security groups — is port 443/80 open to the internet?

■ Example — YAML / Commands

```
# Layer-by-layer debug
kubectl get pods -n production # pods healthy?
kubectl get svc -n production # service exists?
kubectl get endpoints my-svc -n production # endpoints populated?
kubectl get ingress -n production # ingress exists?
kubectl describe ingress my-ingress # any errors?
nslookup app.example.com
# Should resolve to Ingress Controller external IP
kubectl logs -n ingress-nginx \
  deploy/ingress-nginx-controller | grep -E '502|504|refused'
# Test directly via ClusterIP (bypass ingress)
kubectl run test --image=curlimages/curl --rm -it \
  -- curl http://my-svc.production:80
```

■ Key Insight

Work layer by layer: pod → service → endpoints → ingress → DNS → firewall. Most issues are at endpoints or DNS.

Q92

Scenario-Based
Questions

Pods are restarting continuously in production — what to check?

- Step 1: `kubectl get pods` — check restart count column. How many restarts and how often?
- Step 2: `kubectl describe pod` — check `Last State.terminated.reason`. `OOMKilled` = memory issue. `Error` = app crash.
- Step 3: `kubectl logs <pod> --previous` — read what the app said before it died.
- Step 4: if `OOMKilled` — increase memory limit or find the memory leak. Check pod memory trends in Prometheus.
- Step 5: if app crash — check external dependencies. Is the DB down? Is the upstream API returning errors?
- Step 6: check liveness probe — too aggressive settings kill the pod before it's warm. Increase `initialDelaySeconds`.

■ Example — YAML / Commands

```
# Full restart investigation
kubectl get pods -n prod
# RESTARTS column – high number = recurring crash
kubectl describe pod my-pod -n prod
# Last State: Terminated
# Reason: OOMKilled or Error
# Exit Code: 137=OOM, 1=crash, 0=clean exit
kubectl logs my-pod --previous -n prod
# What did the app log before dying?
# Check memory over time (Prometheus)
# container_memory_working_set_bytes{pod='my-pod'}
# Growing without bound = memory leak
# Temporarily increase limits
kubectl set resources deployment/my-app \
--limits=memory=1Gi
```

■ Key Insight

`OOMKilled` = raise the limit. App crash = read previous logs. Always fix root cause, not just raise limits blindly.

Q93

Scenario-Based
Questions

One node is overloaded — how to redistribute workloads?

- Step 1: `kubectl top nodes` — confirm which node is overloaded (CPU or memory).
- Step 2: `kubectl top pods --field-selector spec.nodeName=<node>` — find the heavy pods on that node.
- Step 3: `kubectl drain <node> --ignore-daemonsets` — safely evict all pods to other nodes.
- Step 4: K8s reschedules evicted pods on healthy nodes with capacity.
- Step 5: root cause analysis — why did pods pile up? Missing pod anti-affinity? No resource limits?
- Step 6: add pod anti-affinity to spread replicas across nodes. Enable Cluster Autoscaler for auto-rebalancing.

■ Example — YAML / Commands

```
# Identify overloaded node
kubectl top nodes

# Find heavy pods on that node
kubectl top pods -A --sort-by=cpu | grep <node-name>

# Drain the node (safe eviction)
kubectl drain <overloaded-node> \
  --ignore-daemonsets \
  --delete-emptydir-data

# Pods reschedule automatically
kubectl get pods -A -o wide | grep -v <node-name>

# After fixing root cause, uncordon the node
kubectl uncordon <overloaded-node>

# Add anti-affinity to prevent future pileup
# topologyKey: kubernetes.io/hostname
```

■ Key Insight

Drain redistributes workloads safely. Then add anti-affinity to prevent future pile-ups on a single node.

Q94

Scenario-Based
Questions

Deployment failed after rollout — how do you rollback?

- Step 1: confirm the rollout is failing — `kubectl rollout status deployment/my-app` shows error or stall.
- Step 2: check what went wrong — `kubectl describe pod <new-pod>` and `kubectl logs <new-pod> --previous`.
- Step 3: rollback immediately — `kubectl rollout undo deployment/my-app`. Old ReplicaSet scales back up.
- Step 4: verify rollback succeeded — `kubectl rollout status` and `kubectl get pods` show stable state.
- Step 5: check the image version running after rollback — `kubectl describe deployment | grep Image`.
- Step 6: investigate root cause before re-deploying — fix the bug/config, test in staging, then re-push.

■ Example — YAML / Commands

```
# See that rollout is failing
kubectl rollout status deployment/my-app
kubectl get pods -l app=my-app
# Read the failure reason
kubectl describe pod <failing-new-pod>
kubectl logs <failing-new-pod> --previous
# ROLLBACK NOW
kubectl rollout undo deployment/my-app
# Watch it stabilise
kubectl rollout status deployment/my-app
# Verify old version is back
kubectl describe deployment my-app | grep Image
# Check rollout history
kubectl rollout history deployment/my-app
```

■ Key Insight

Rollback first, investigate second. Every minute a broken version runs is more user impact. Undo now, fix and redeploy later.

Q95

Scenario-Based
Questions

You updated ConfigMap but app not picking changes — fix?

- Diagnosis: how is the ConfigMap consumed? As env vars (never auto-updates) or volume mount (updates slowly)?
- Env vars from ConfigMap: NEVER hot-reload. Pod must be restarted to pick up new values.
- Volume-mounted ConfigMap: syncs automatically in ~60 seconds, but app must actively re-read the file.
- Quick fix: `kubectl rollout restart deployment/my-app` — rolling restart picks up new values.
- Automated fix (Helm): add `sha256sum` annotation to pod template. CM change = annotation change = auto-rollout.
- Automated fix (operator): Stakater Reloader watches CM changes and triggers restart automatically.

■ Example — YAML / Commands

```
# Immediate fix: rolling restart
kubectl rollout restart deployment/my-app -n production
kubectl rollout status deployment/my-app -n production

# Verify inside new pod
kubectl exec -it $(kubectl get pod -l app=my-app \
-o jsonpath='{.items[0].metadata.name}') \
-- env | grep MY_CONFIG

# Helm automation: checksum annotation
# In deployment.yaml template spec.template.metadata:
annotations:
checksum/config: {{ include (print $.Template.BasePath
'/configmap.yaml') . | sha256sum }}

# Every CM change updates the hash -> triggers rollout
helm upgrade my-app ./chart -f values.yaml
```

■ Key Insight

rollout restart is the safe, universal fix. Automate it with checksum annotations in Helm so it happens automatically.

Q96

Scenario-Based
Questions

Traffic not reaching Pod via Ingress — possible issues?

- Issue 1: Ingress Controller not running or crashed — check pods in ingress-nginx namespace.
- Issue 2: Ingress rule host mismatch — spec.rules.host must match the request's Host header exactly.
- Issue 3: Backend Service doesn't exist or selector doesn't match pod labels → empty Endpoints.
- Issue 4: Pod not Ready — failing readiness probe removes pod from Endpoints → 502 from Ingress.
- Issue 5: NetworkPolicy blocking traffic between Ingress Controller namespace and pod namespace.
- Issue 6: TLS secret missing or expired — HTTPS requests fail before reaching the backend.

■ Example — YAML / Commands

```
# Full Ingress traffic debug
kubectl get pods -n ingress-nginx
kubectl describe ingress my-ingress -n production
kubectl get svc my-svc -n production
kubectl get endpoints my-svc -n production
kubectl get pods -l app=my-app -n production
# Check readiness
kubectl get pod <pod> -o jsonpath=\
'{}.status.containerStatuses[0].ready}'
# Ingress controller logs
kubectl logs -n ingress-nginx \
deploy/ingress-nginx-controller | grep -E 'error|502|refused'
# Check NetworkPolicy
kubectl get networkpolicy -n production
# Check TLS cert
kubectl get secret tls-cert -n production -o yaml | \
grep 'tls.crt' | awk '{print $2}' | base64 -d | openssl x509 -noout -dates
```

■ Key Insight

Work in order: controller running? → rule correct? → endpoints populated? → pod ready? →
NetworkPolicy? → TLS valid?

Q97

Scenario-Based
Questions

Cluster running out of memory — what steps?

- Step 1: `kubectl top nodes` — identify which nodes are at or near 100% memory.
- Step 2: `kubectl top pods -A --sort-by=memory` — identify memory hogs.
- Step 3: check for pods without memory limits — they can consume unlimited memory.
- Step 4: apply `LimitRange` to namespaces — sets default limits on all pods going forward.
- Step 5: evict `BestEffort` pods — they have no limits and get evicted first anyway.
- Step 6: scale out — add more nodes via `Cluster Autoscaler` or manually scale node group.

■ Example — YAML / Commands

```
# Find memory pressure
kubectl top nodes
kubectl top pods -A --sort-by=memory | head -20
# Find pods with no limits
kubectl get pods -A -o json | jq '.items[] |
select(.spec.containers[].resources.limits==null) |
.metadata.name'
# Apply LimitRange to cap unlimited pods
apiVersion: v1
kind: LimitRange
metadata:
  name: default-limits
  namespace: production
spec:
  limits:
    - type: Container
  default:
    memory: 512Mi
    cpu: 500m
  defaultRequest:
    memory: 256Mi
    cpu: 100m
```

■ Key Insight

`LimitRange` prevents future unlimited pods. Clean up existing ones. Scale nodes. Then enable `Cluster Autoscaler` permanently.

Q98

Scenario-Based
Questions

Need to run one Pod per node — how?

- Use a DaemonSet — it automatically places exactly one pod on every node.
- When a new node joins, DaemonSet adds a pod. When a node leaves, the pod is cleaned up.
- Can use nodeSelector or affinity to target only specific nodes (e.g. only worker nodes, not control plane).
- DaemonSet ignores replica count — you cannot set replicas. Count = number of matching nodes.
- Use cases: log agents, node monitoring (Node Exporter), security scanners, CNI plugins.
- If you only want it on some nodes: use nodeSelector with a label, and add tolerations for tainted nodes.

■ Example — YAML / Commands

```
apiVersion: apps/v1
kind: DaemonSet
metadata:
  name: node-monitor
  namespace: monitoring
spec:
  selector:
    matchLabels:
      app: node-monitor
  template:
    metadata:
      labels:
        app: node-monitor
    spec:
      tolerations:
        - key: node-role.kubernetes.io/control-plane
          effect: NoSchedule
      containers:
        - name: monitor
          image: datadog/agent:7
kubectl get daemonset node-monitor -n monitoring
kubectl get pods -n monitoring -o wide # one per node
```

■ Key Insight

DaemonSet = one pod per node, always. It self-manages — adding nodes automatically gets a pod. Removing nodes cleans up.

Q99

Scenario-Based
Questions

Need to isolate workloads between teams — how?

- Namespaces: create one per team (team-a, team-b). Separates resources, allows scoped controls.
- ResourceQuota: limit total CPU/memory/pods per namespace — one team can't starve another.
- RBAC: each team can only access their own namespace — RoleBindings scoped per namespace.
- NetworkPolicy: deny all cross-namespace traffic by default, whitelist only what is needed.
- PodSecurity Admission: enforce baseline or restricted per namespace — no privileged containers.
- LimitRange: set default resource limits so pods without explicit limits still get capped.

■ Example — YAML / Commands

```
kubectl create namespace team-a
kubectl create namespace team-b
# ResourceQuota for team-a
apiVersion: v1
kind: ResourceQuota
metadata:
  name: team-a-quota
  namespace: team-a
spec:
  hard:
    requests.cpu: '8'
    requests.memory: 16Gi
    limits.cpu: '16'
    limits.memory: 32Gi
    pods: '50'
# RBAC: team-a members can only access team-a
kubectl create rolebinding team-a-access \
  --clusterrole=edit \
  --group=team-a \
  -n team-a
# PodSecurity for the namespace
kubectl label namespace team-a \
  pod-security.kubernetes.io/enforce=baseline
```

■ Key Insight

Namespaces alone don't isolate. You need all four: ResourceQuota + RBAC + NetworkPolicy + PodSecurity together.

Q100

Scenario-Based
Questions

How to secure communication between services — mTLS and service mesh?

- mTLS (Mutual TLS): both client and server verify each other's certificates — proves identity + encrypts traffic.
- Without mTLS: any pod in the cluster can call any other pod — no identity verification.
- Istio: injects Envoy sidecar into every pod automatically. Handles mTLS transparent to the app.
- Enable strict mTLS with PeerAuthentication — all service-to-service traffic is encrypted and mutually verified.
- DestinationRule: configure how traffic goes to a service — TLS mode, load balancing, circuit breaking.
- Alternative: Linkerd — lighter weight than Istio, also auto-injects mTLS sidecars. Simpler but less features.

■ Example — YAML / Commands

```
# Install Istio
istioctl install --set profile=production
# Label namespace for auto sidecar injection
kubectl label namespace production istio-injection=enabled
# Enable strict mTLS for entire namespace
apiVersion: security.istio.io/v1beta1
kind: PeerAuthentication
metadata:
  name: default
  namespace: production
spec:
  mTLS:
    mode: STRICT
---
# DestinationRule: enforce TLS on outbound
apiVersion: networking.istio.io/v1beta1
kind: DestinationRule
metadata:
  name: default
  namespace: production
spec:
  host: '*.production.svc.cluster.local'
  trafficPolicy:
    tls:
      mode: ISTIO_MUTUAL
```

■ Key Insight

mTLS with Istio = zero-trust networking inside your cluster. Every service connection is encrypted and identity-verified.