DevOps Engineering · Container Orchestration Series

Kubernetes (K8s)

Complete Interview Preparation Guide · 40 Questions with Detailed Answers

Sandhya · DevOps Engineer

40	**7**	**Easy/Med/Hard**	**kubectl**
Questions	Core Sections	All Levels	Code Examples

WHAT'S COVERED IN THIS GUIDE

HOW TO USE: Every question has a full Answer, Key Points to memorize, and an Interview Tip with real production context. kubectl commands are included throughout.

01 Kubernetes Architecture

Q01–Q06 · Control Plane, Worker Nodes, etcd, API Server, Scheduler, kubelet

Q01 What is Kubernetes? Why is it used in modern DevOps? Easy

✓ANSWER

Kubernetes (K8s) is an open-source container orchestration platform that automates the deployment, scaling, and management of containerized applications across clusters of machines.

Why Kubernetes is essential in DevOps:

- Automated scheduling — places containers on the right nodes based on resource availability
- Self-healing — automatically restarts failed containers, replaces unhealthy nodes
- Horizontal scaling — scale applications up or down with a single command or automatically
- Rolling updates & rollbacks — zero-downtime deployments with instant rollback capability
- Service discovery & load balancing — built-in DNS and traffic distribution
- Storage orchestration — automatically mounts storage systems (cloud, local, NFS)
- Secret & config management — manage sensitive data without rebuilding images
- Multi-cloud & on-prem — runs on AWS, GCP, Azure, or bare metal identically

KEY POINTS

- K8s = Greek for 'helmsman' — the 8 replaces 8 letters in 'ubernete'
- Google open-sourced K8s in 2014 based on internal Borg system
- CNCF (Cloud Native Computing Foundation) maintains K8s
- Industry standard for container orchestration — Docker Swarm is legacy

INTERVIEW TIP

Open with the business value: 'Kubernetes solves the problem of running containers at scale reliably. Docker runs one container on one machine. Kubernetes runs thousands of containers across hundreds of machines — with automatic failover, scaling, and zero-downtime deployments.' Then mention self-healing as the killer feature.

Q02 Explain the complete Kubernetes architecture — Control Plane and Worker Nodes. Hard

✓ANSWER

Kubernetes has a master-worker architecture split into two planes:

CONTROL PLANE (Master Node) — manages the cluster:

- API Server (kube-apiserver) — the front door of Kubernetes. ALL communication goes through it. Validates and persists state to etcd. Exposes REST API.
- etcd — distributed key-value store. Stores ALL cluster state (pods, configs, secrets). Source of truth for K8s.
- Scheduler (kube-scheduler) — watches for new pods with no assigned node. Selects best node based on resource requirements, affinity rules, taints/tolerations.

- Controller Manager (kube-controller-manager) — runs controller loops: ReplicaSet controller (ensures desired pod count), Node controller (handles node failures), Job controller, Endpoints controller.
- Cloud Controller Manager — integrates K8s with cloud provider APIs (AWS, GCP, Azure) for load balancers, storage, nodes.

WORKER NODES — run application workloads:

- kubelet — agent running on every node. Receives pod specs from API Server. Ensures containers in pods are running and healthy.
- kube-proxy — network proxy on each node. Maintains network rules (iptables/ipvs) for Service routing. Handles load balancing across pod endpoints.
- Container Runtime — actually runs containers: containerd (default), CRI-O, or Docker (deprecated).
- Pods — smallest deployable unit. One or more containers sharing network namespace and storage volumes.

kubectl / YAML

```
# View cluster components
kubectl get componentstatuses
kubectl get nodes -o wide

# Control plane pods (in kubeadm clusters):
kubectl get pods -n kube-system

# API server info:
kubectl cluster-info

# Node details (see kubelet version, container runtime):
kubectl describe node <node-name>
```

KEY POINTS

- API Server = only component that talks to etcd
- etcd = cluster's database — back it up regularly!
- kubelet = node agent, not a container itself
- kube-proxy = handles Service IP routing via iptables/ipvs

INTERVIEW TIP

Draw this architecture mentally: Control Plane (API Server ↔ etcd + Scheduler + Controller Manager) and Worker Nodes (kubelet + kube-proxy + container runtime + pods). Interviewers love when you explain the flow: 'kubectl → API Server → etcd (store state) → Scheduler (assign node) → kubelet (run pod).'

Q03 What is etcd in Kubernetes? Why is it critical?

Medium

✓ANSWER

etcd is a distributed, consistent key-value store that serves as Kubernetes' entire backend database — it stores the complete state of the cluster.

What etcd stores:

- All pod definitions and their current state
- Node registrations and status

- ConfigMaps and Secrets
- Service definitions and endpoints
- Deployment, ReplicaSet, and StatefulSet specs
- RBAC policies and ServiceAccounts
- Namespace definitions

Key characteristics of etcd:

- Uses Raft consensus algorithm — guarantees consistency across multiple etcd instances
- Strongly consistent — reads always return the latest committed data
- Highly available — typically run as 3 or 5 node cluster (odd numbers for quorum)
- Watch mechanism — clients (like API Server) subscribe to key changes for real-time updates

Why etcd is critical:

- If etcd is lost and has no backup, the ENTIRE cluster state is gone
- ALL Kubernetes API calls ultimately read from or write to etcd
- Production clusters must back up etcd regularly using etcdctl snapshot save

kubectl / YAML

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

# Verify snapshot:
ETCDCTL_API=3 etcdctl snapshot status /backup/etcd-snapshot.db

# etcd is on port 2379 (client) and 2380 (peer)
```

KEY POINTS

- etcd = single source of truth for all cluster state
- Raft consensus — requires (n/2)+1 nodes for quorum
- Run odd number of etcd nodes (3 or 5) for HA
- Back up etcd regularly — losing it = losing the cluster

INTERVIEW TIP

The backup point is what separates experienced engineers. Say: 'In production, I schedule automated etcd snapshots every hour using etcdctl snapshot save. Before any major cluster upgrade, I take a manual snapshot. Losing etcd without a backup means rebuilding the entire cluster from scratch.' This shows production maturity.

Q04 What is the role of the Scheduler in Kubernetes? How does it decide which node to place a pod on?

Medium

✓ANSWER

The Kubernetes Scheduler (kube-scheduler) is responsible for watching for newly created pods that have no assigned node, and selecting the best node for them to run on.

Scheduling process — two phases:

Phase 1 — FILTERING (eliminates nodes that cannot run the pod):

- Node has enough CPU and memory (requests vs available)
- Node satisfies nodeSelector labels
- Node has required taints tolerated by the pod
- Node passes affinity/anti-affinity rules
- Node has required ports free
- Node has the volume the pod needs available

Phase 2 — SCORING (ranks remaining nodes to find the best one):

- Least resource usage (spreads load evenly)
- Node affinity preference weights
- Pod affinity/anti-affinity soft rules
- Image already pulled on node (faster startup)
- The node with highest score wins

After selection: Scheduler writes the node name to the pod spec via API Server. kubelet on that node sees the assignment and starts the pod.

Manual override: Use nodeName or nodeSelector to bypass the scheduler.

kubectl / YAML

```
# See why a pod is Pending (scheduling failure):
kubectl describe pod <pod-name>
# Look for: Events section — 'FailedScheduling'

# Force pod to specific node (bypass scheduler):
# In pod spec:
# nodeName: worker-node-1

# Node selector (schedule on labeled nodes):
# nodeSelector:
#   disktype: ssd

# Label a node:
kubectl label node worker-node-1 disktype=ssd

# View node resources:
kubectl describe nodes | grep -A 5 'Allocated resources'
```

KEY POINTS

- Scheduler = filter → score → assign (two-phase process)
- Scheduler only assigns — kubelet does the actual pod creation
- nodeName in pod spec bypasses scheduler completely

INTERVIEW TIP

Mention the filter-then-score two-phase process — it shows deep understanding. Real scenario: 'In one project, pods were stuck in Pending state. I ran kubectl describe pod and saw the scheduler couldn't find a node with enough memory. Adding a node with more RAM resolved it immediately.'

- Custom schedulers can be deployed alongside default scheduler

Q05 What is kubelet? What is its role in a Kubernetes cluster? Medium

✓ANSWER

kubelet is the primary node agent that runs on every worker node in a Kubernetes cluster. It is the bridge between the Control Plane and the actual container runtime on each node.

What kubelet does:

1. Watches the API Server for pod specs assigned to its node
2. Ensures the containers described in pod specs are running and healthy
3. Communicates with the container runtime (containerd) via CRI (Container Runtime Interface) to start/stop containers
4. Runs liveness and readiness probes to check container health
5. Reports node and pod status back to the API Server
6. Manages container lifecycle: start, restart on failure, stop
7. Mounts volumes into containers
8. Handles log collection from containers

Important: kubelet does NOT manage containers not created by Kubernetes (e.g., containers run directly with docker run).

kubelet vs kube-proxy:

- kubelet = manages POD lifecycle on the node
- kube-proxy = manages NETWORK RULES for Service routing on the node

kubectl / YAML

```
# Check kubelet status on a node:
systemctl status kubelet

# kubelet config location:
cat /var/lib/kubelet/config.yaml

# kubelet logs:
journalctl -u kubelet -f

# kubelet runs containers via containerd:
# kubelet → CRI → containerd → runc → container

# View pods on a specific node:
kubectl get pods --field-selector spec.nodeName=<node-name>
```

KEY POINTS

- kubelet runs on EVERY worker node as a systemd service

INTERVIEW TIP

A common interview trap: 'Is kubelet a pod?' — No! kubelet is a systemd service running directly on the node, not a pod managed by K8s. It's the

▸ kubelet = the only K8s component that runs containers directly ▸ kubelet communicates with container runtime via CRI ▸ kubelet does NOT manage containers created outside K8s	component that bootstraps itself and then starts managing pods. This detail shows genuine architecture knowledge.

Q06 What is kube-proxy and how does Service routing work in Kubernetes? Hard

✓**ANSWER**

kube-proxy is a network proxy that runs on each node in a Kubernetes cluster. It maintains network rules that allow network communication to pods from inside or outside the cluster.

What kube-proxy does:

- Watches the API Server for Service and Endpoints objects
- When a Service is created, kube-proxy creates network rules to route traffic to the appropriate pods
- Implements load balancing across pod endpoints

How Service routing works:

9. You create a Service with a ClusterIP (virtual IP, e.g., 10.96.0.1)
10. kube-proxy watches this Service and gets the list of pod endpoints
11. kube-proxy creates iptables rules (or IPVS rules) on every node
12. When traffic hits the ClusterIP:port, iptables/IPVS rewrites the destination to a real pod IP
13. Traffic reaches the pod directly via the cluster network

kube-proxy modes:

- iptables mode (default) — creates iptables rules for each Service. Simple but can be slow at scale (10,000+ Services).
- IPVS mode — uses Linux kernel's IP Virtual Server. Hash table lookups = O(1) vs iptables O(n). Better for large clusters.
- userspace mode (legacy) — deprecated. kube-proxy itself proxies traffic.

kubectl / YAML

```
# See iptables rules created by kube-proxy:
sudo iptables -t nat -L KUBE-SERVICES -n

# Switch to IPVS mode (edit kube-proxy configmap):
kubectl edit configmap kube-proxy -n kube-system
# Change mode: 'iptables' to 'ipvs'

# Check kube-proxy logs:
kubectl logs -n kube-system -l k8s-app=kube-proxy

# See IPVS rules (if using IPVS mode):
sudo ipvsadm -ln
```

KEY POINTS

- kube-proxy runs on EVERY node — manages iptables/IPVS rules
- ClusterIP is virtual — iptables rewrites it to pod IP
- IPVS mode preferred for clusters with 1000+ Services
- kube-proxy does NOT proxy traffic directly — it sets up kernel rules

INTERVIEW TIP

The 'ClusterIP is virtual' point is gold. Say: 'ClusterIP doesn't actually exist on any interface. It's a virtual IP that exists only in iptables rules. When a pod connects to 10.96.0.1:80, the kernel intercepts and rewrites the destination to a real pod IP before routing. kube-proxy just maintains these rules.' This shows kernel-level networking understanding.

02 Core Workloads

Q07–Q13 · Pods, Deployments, ReplicaSets, StatefulSets, DaemonSets, Jobs, CronJobs

Q07 What is a Pod in Kubernetes? What makes it different from a container? Easy

✓ANSWER

A Pod is the smallest and most basic deployable unit in Kubernetes. It wraps one or more containers that share the same network namespace, storage volumes, and lifecycle.

Key Pod characteristics:

- Shared network — all containers in a pod share the same IP address and port space. They communicate via localhost.
- Shared storage — containers in a pod can share volumes mounted into the pod
- Co-scheduled — all containers in a pod always run on the same node
- Ephemeral — pods are not self-healing. If a pod dies, it's gone. Deployments and ReplicaSets ensure replacement pods are created.

Pod vs Container:

- A container = a single running process with its own filesystem
- A pod = one or more containers with shared network/storage, treated as a single unit

Single-container pods (most common): One container per pod — simple and recommended for microservices.
Multi-container pods (sidecar pattern): e.g., main app container + log shipping sidecar container. They share localhost and volumes.

Why pods exist (not containers directly):

- Allows sidecar, ambassador, adapter container patterns
- Provides a consistent unit of scheduling, networking, and storage

kubectl / YAML

```
# Create a simple pod:
kubectl run nginx --image=nginx:alpine

# Pod YAML with two containers (sidecar pattern):
# apiVersion: v1
# kind: Pod
# spec:
#   containers:
#   - name: app
#     image: myapp:latest
#   - name: log-shipper
#     image: fluentd:latest
#     volumeMounts:
#     - name: logs
#       mountPath: /logs
# Get pod logs (specific container in multi-container pod):
kubectl logs <pod-name> -c <container-name>

# Shell into a pod:
kubectl exec -it <pod-name> -- bash
```

KEY POINTS

- Pod = one or more containers sharing network + storage
- Containers in same pod communicate via localhost
- Pods are ephemeral — replaced not restarted by K8s
- Smallest schedulable unit — not containers directly

INTERVIEW TIP

Explain the sidecar pattern with a real example: 'In production, I run a fluentd sidecar container in the same pod as the application. They share a volume — the app writes logs to a file, fluentd reads from that same file and ships to Elasticsearch. No networking between them — just a shared volume via localhost.'

Q08 What is a Deployment in Kubernetes? How is it different from a ReplicaSet? Medium

✓ANSWER

A Deployment is the standard way to run stateless applications in Kubernetes. It provides declarative updates, rolling upgrades, and rollback capabilities.

REPLICASET:

- Ensures a specified number of identical pod replicas are running at all times
- If a pod dies, ReplicaSet creates a replacement
- Does NOT provide update strategy or rollback — it only maintains pod count
- Identified by a label selector — owns all pods matching its selector

DEPLOYMENT (manages ReplicaSets):

- Manages ReplicaSets — creates a new ReplicaSet for each version change
- Provides rolling update strategy — gradually replaces old pods with new
- Provides rollback — keep history of previous ReplicaSets for instant rollback
- Provides pause and resume — pause rollout, fix issues, resume

Relationship: Deployment → ReplicaSet → Pods
When you update a Deployment image, it creates a NEW ReplicaSet (v2) and scales it up while scaling down the OLD ReplicaSet (v1). Both exist temporarily during the rollout. After completion, old ReplicaSet stays (for rollback) but with 0 replicas.

Best practice: ALWAYS use Deployments for stateless apps. Never create ReplicaSets or Pods directly.

kubectl / YAML

```
# Create a deployment:
kubectl create deployment myapp --image=myapp:v1 --replicas=3

# Update image (triggers rolling update):
kubectl set image deployment/myapp myapp=myapp:v2
```

```
# Check rollout status:
kubectl rollout status deployment/myapp

# View rollout history:
kubectl rollout history deployment/myapp

# Rollback to previous version:
kubectl rollout undo deployment/myapp

# Rollback to specific revision:
kubectl rollout undo deployment/myapp --to-revision=2

# See ReplicaSets created by deployment:
kubectl get replicasets -l app=myapp
```

KEY POINTS

- Deployment manages ReplicaSets. ReplicaSet manages Pods.
- Deployment = ReplicaSet + update strategy + rollback history
- Never create ReplicaSets directly — use Deployments
- Old ReplicaSets kept for rollback (controlled by revisionHistoryLimit)

INTERVIEW TIP

Show the relationship chain: 'Deployment → creates ReplicaSet → creates Pods. During a rolling update, you temporarily have two ReplicaSets — old one scaling down, new one scaling up. That's how zero-downtime deployment works.' Then mention rollback: 'kubectl rollout undo deployment/myapp instantly switches back to the previous ReplicaSet.'

Q09 What is a StatefulSet? How is it different from a Deployment? Hard

✓ANSWER

A StatefulSet is a Kubernetes workload object designed for stateful applications that require stable network identities, stable persistent storage, and ordered deployment/scaling.

Aspect	Deployment	StatefulSet
Pod names	Random (myapp-xyz)	Ordered (myapp-0, myapp-1, myapp-2)
Pod identity	Interchangeable	Stable — myapp-0 is always myapp-0
Storage	Shared or none	Each pod gets its OWN PVC
Start order	Parallel	Sequential (0 → 1 → 2)
Delete order	Parallel	Reverse sequential (2 → 1 → 0)
DNS	Service ClusterIP	Stable DNS per pod: myapp-0.myapp-svc

Why StatefulSets exist:

- Databases need stable storage — myapp-0's data must persist even if pod is rescheduled to different node
- Clustered databases (MySQL, Cassandra, Elasticsearch) need stable network identities for cluster formation
- Primary/replica patterns — myapp-0 is always the primary, myapp-1 is always a replica

Use cases: PostgreSQL, MySQL, Cassandra, Kafka, Zookeeper, Elasticsearch, Redis Sentinel

StatefulSets require a Headless Service (clusterIP: None) to provide stable DNS for each pod.

kubectl / YAML

```
# StatefulSet YAML key sections:
# kind: StatefulSet
# spec:
#   serviceName: myapp-svc  # headless service name
#   replicas: 3
#   volumeClaimTemplates:   # each pod gets own PVC
#   - metadata:
#       name: data
#     spec:
#       accessModes: [ReadWriteOnce]
#       resources:
#         requests:
#           storage: 10Gi

# Headless service (required):
# kind: Service
# spec:
#   clusterIP: None  # headless!
#   selector:
#     app: myapp

# Stable DNS: <pod-name>.<service-name>.<namespace>.svc.cluster.local
# postgres-0.postgres-svc.default.svc.cluster.local
```

KEY POINTS

- StatefulSet pods have stable names: myapp-0, myapp-1, myapp-2
- Each pod gets its OWN PersistentVolumeClaim (PVC)
- Requires a Headless Service (clusterIP: None) for stable DNS
- Use for: databases, message queues, distributed systems

INTERVIEW TIP

Give the database example: 'For a 3-replica PostgreSQL cluster, I use a StatefulSet. postgres-0 is always the primary — it gets a stable DNS name postgres-0.postgres-svc. postgres-1 and postgres-2 are replicas that connect to postgres-0 for replication. A Deployment couldn't do this — pods are anonymous and interchangeable.'

Q10 What is a DaemonSet in Kubernetes? Give real-world use cases.

Medium

✓ANSWER

A DaemonSet ensures that ONE copy of a pod runs on EVERY node in the cluster (or a subset of nodes via node selectors). When a new node joins the cluster, the DaemonSet automatically adds the pod to it. When a node is removed, the pod is garbage collected.

Key characteristics:

- Exactly one pod per node (not per cluster)
- Automatically adds pod to new nodes as they join

- Automatically removes pod from nodes that leave
- Can be limited to specific nodes using nodeSelector or affinity

Real-world use cases:

- Log collection — Fluentd or Filebeat on every node to ship container logs to Elasticsearch/Splunk
- Monitoring agents — Prometheus node-exporter on every node to collect host-level metrics
- Network plugins — Calico, Flannel, Weave CNI plugins run as DaemonSets
- Security scanning — Falco runtime security agent on every node
- Storage drivers — Ceph or GlusterFS storage drivers on every node
- System-level tools — kube-proxy itself runs as a DaemonSet

DaemonSet vs Deployment:

- Deployment: 'run N replicas somewhere in the cluster'
- DaemonSet: 'run exactly one pod on EVERY node'

kubectl / YAML

```
# Get all DaemonSets:
kubectl get daemonsets --all-namespaces

# See DaemonSet pods per node:
kubectl get pods -o wide -l app=fluentd

# DaemonSet YAML:
# kind: DaemonSet
# spec:
#   selector:
#     matchLabels:
#       app: node-exporter
#   template:
#     spec:
#       containers:
#       - name: node-exporter
#         image: prom/node-exporter
#         ports:
#         - containerPort: 9100
#       tolerations:
#       - operator: Exists  # run on ALL nodes including control plane
```

KEY POINTS

- DaemonSet = one pod per node automatically
- New nodes auto-get the DaemonSet pod
- kube-proxy and CNI plugins are DaemonSets
- Perfect for node-level monitoring and log collection

INTERVIEW TIP

Name-drop real tools: 'In my cluster, I run Fluentd as a DaemonSet for log collection, Prometheus node-exporter as a DaemonSet for metrics, and Falco as a DaemonSet for runtime security. Every node automatically gets all three — I never have to think about node-level agents.' This shows production-level thinking.

Q11 What is a Job and CronJob in Kubernetes? When would you use them? Medium

✅**ANSWER**

JOB:

- A Job creates one or more pods and ensures they complete successfully
- Unlike Deployments, Jobs run to completion — they are not long-running services
- If a pod fails, Job creates a replacement until the task succeeds
- After successful completion, pods are kept (for log inspection) but not restarted

Job use cases:

- Database migrations before new app version deploys
- Batch data processing (ETL pipelines)
- Sending emails or notifications in bulk
- Report generation
- ML model training runs

CRONJOB:

- Runs a Job on a recurring schedule (cron syntax)
- Creates a new Job object at each scheduled time
- Manages history of completed and failed Jobs

CronJob use cases:

- Nightly database backups
- Hourly report generation
- Scheduled cache clearing
- Periodic health check reports
- Automated cleanup of old data

kubectl / YAML

```
# Create a one-time job:
kubectl create job db-migrate --image=myapp:latest -- python migrate.py

# Watch job completion:
kubectl get jobs -w
kubectl logs job/db-migrate

# CronJob (every day at 2am):
# kind: CronJob
# spec:
#   schedule: '0 2 * * *'
#   jobTemplate:
#     spec:
#       template:
#         spec:
#           containers:
#           - name: backup
#             image: backup-tool:latest
#           restartPolicy: OnFailure

# List cronjobs:
kubectl get cronjobs
```

KEY POINTS

- Job = run to completion. Deployment = run forever.
- Job retries on failure until successfulCompletions reached
- CronJob creates a new Job object at each schedule
- completions + parallelism controls batch processing scale

INTERVIEW TIP

Real scenario: 'Before every deployment, I run a Kubernetes Job to execute database migrations. The Job must complete successfully before the new Deployment pods start — I implement this in my CI/CD pipeline by waiting for the Job to succeed before applying the Deployment manifest.'

Q12 What is the difference between a Liveness Probe, Readiness Probe, and Startup Probe?

Hard

ANSWER

Kubernetes uses probes to determine the health of containers. Getting probes right is critical for production reliability.

LIVENESS PROBE — Is the container alive?

- Checks if the container is running correctly
- If liveness probe FAILS: K8s kills the container and restarts it (based on restartPolicy)
- Use for: detecting deadlocks, infinite loops, corrupted state the app can't recover from
- Example: HTTP GET /health returns 200, or TCP port is open

READINESS PROBE — Is the container ready to serve traffic?

- Checks if the container is ready to accept requests
- If readiness probe FAILS: pod is removed from Service endpoints (no traffic sent to it)
- Container is NOT killed — it just stops receiving traffic until it recovers
- Use for: app startup time, dependency checks, temporary overload
- Example: check DB connection is established before accepting requests

STARTUP PROBE — Has the container started yet?

- Introduced to handle slow-starting containers
- While startup probe is running, liveness and readiness probes are DISABLED
- If startup probe fails after failureThreshold: container is killed
- Use for: legacy apps that take long to initialize (Java apps, heavy ML models)

Key difference: Readiness failure = stop traffic (pod stays). Liveness failure = kill and restart (pod restarts).

kubectl / YAML

```
# Pod spec with all three probes:
# livenessProbe:
#   httpGet:
#     path: /health
#     port: 8080
#   initialDelaySeconds: 30
#   periodSeconds: 10
#   failureThreshold: 3

# readinessProbe:
#   httpGet:
#     path: /ready
#     port: 8080
#   initialDelaySeconds: 5
#   periodSeconds: 5

# startupProbe:
#   httpGet:
#     path: /health
#     port: 8080
#   failureThreshold: 30  # 30 * 10s = 5 min max startup
#   periodSeconds: 10
```

KEY POINTS

- Liveness failure → container RESTART
- Readiness failure → removed from Service endpoints, NOT restarted
- Startup probe → disables liveness/readiness until app starts
- Always set initialDelaySeconds to avoid false failures during startup

INTERVIEW TIP

The readiness vs liveness distinction is the classic interview test. Say: 'I use readiness probes for the warm-up period — my Java app takes 30 seconds to start. Without a readiness probe, K8s sends traffic immediately and users get errors. With readiness probe, traffic only arrives after /health returns 200. I use liveness for detecting stuck states that only a restart can fix.'

Q13 What are resource requests and limits in Kubernetes? Why do they matter?

Medium

✓ANSWER

Resource requests and limits are how you tell Kubernetes how much CPU and memory a container needs and is allowed to use.

REQUESTS — what the container is GUARANTEED:

- Used by the Scheduler to find a node with sufficient available resources
- The node must have at least this much CPU/memory available for the pod to be scheduled
- The container is guaranteed this much — it will never have less

LIMITS — the MAXIMUM the container can use:

- Container cannot exceed this amount
- CPU limit: container is throttled (slowed down) if it exceeds the CPU limit
- Memory limit: if container exceeds memory limit, it is OOMKilled (killed immediately)

What happens without limits:

- A runaway container can consume all node resources, evicting other pods (noisy neighbor problem)
- Node becomes unstable — kubelet starts evicting pods to reclaim memory

QoS Classes (determined by requests/limits):

- Guaranteed — requests == limits. Highest priority. Never evicted first.

- Burstable — requests < limits. Medium priority.
- BestEffort — no requests or limits. Lowest priority. First to be evicted.

kubectl / YAML

```
# Pod resource spec:
# resources:
#   requests:
#     memory: '128Mi'
#     cpu: '250m'      # 250 millicores = 0.25 CPU
#   limits:
#     memory: '512Mi'
#     cpu: '1000m'    # 1000 millicores = 1 CPU

# View resource usage:
kubectl top pods
kubectl top nodes

# Check for OOMKilled containers:
kubectl describe pod <pod-name>
# Look for: 'OOMKilled' in Last State section

# 250m = 0.25 CPU core. 1000m = 1 full core.
```

KEY POINTS

- Requests = scheduling guarantee. Limits = hard maximum.
- CPU over limit = throttled. Memory over limit = OOMKilled.
- No limits = BestEffort QoS — first evicted under pressure
- Always set requests and limits in production workloads

INTERVIEW TIP

OOMKill is a very common production issue. Say: 'I've debugged OOMKill situations by running kubectl describe pod — the Last State shows OOMKilled with exit code 137. The fix is either increasing the memory limit or finding the memory leak in the application. I always monitor memory usage with kubectl top pods to right-size limits.'

03 Networking & Services

Q14–Q19 · ClusterIP, NodePort, LoadBalancer, Ingress, DNS, NetworkPolicy

Q14 Explain the four Kubernetes Service types — ClusterIP, NodePort, LoadBalancer, ExternalName.

Medium

✓**ANSWER**

A Service provides a stable network endpoint for a set of pods, regardless of pod restarts or rescheduling.

14. ClusterIP (default):
    - Exposes the service on an internal IP within the cluster
    - Only accessible from within the cluster
    - Use for: internal microservice communication (API → Database, API → Cache)
15. NodePort:
    - Exposes the service on each node's IP at a static port (30000-32767)
    - External access: NodeIP:NodePort
    - Not recommended for production (depends on node IPs, limited port range)
    - Use for: development, on-premise without load balancer
16. LoadBalancer:
    - Creates a cloud load balancer (AWS ALB/NLB, GCP LB, Azure LB) automatically
    - Gives a public IP/DNS for external access
    - Works on cloud providers. On bare-metal: use MetalLB.
    - Use for: production external-facing services
17. ExternalName:
    - Maps a service to an external DNS name
    - Returns a CNAME record, no proxying
    - Use for: accessing external databases or services by a K8s-internal name

kubectl / YAML

```
# ClusterIP (internal only):
kubectl expose deployment myapp --port=80 --target-port=8080

# NodePort:
kubectl expose deployment myapp --type=NodePort --port=80

# LoadBalancer:
kubectl expose deployment myapp --type=LoadBalancer --port=80

# Get service details:
kubectl get svc myapp
kubectl describe svc myapp
```

```
# ExternalName YAML:
# kind: Service
# spec:
#   type: ExternalName
#   externalName: my-database.company.com
```

KEY POINTS

- ClusterIP = internal only (default)
- NodePort = static port on every node (30000-32767)
- LoadBalancer = cloud LB with public IP (costs money)
- ExternalName = CNAME to external service (no proxying)

INTERVIEW TIP

The hierarchy is important: LoadBalancer builds on NodePort which builds on ClusterIP. When you create a LoadBalancer service, K8s also creates a NodePort and ClusterIP automatically. The cloud LB sends traffic to the NodePort. Understanding this chain separates strong candidates.

Q15 What is a Kubernetes Ingress? How is it different from a LoadBalancer Service?

Hard

✓ANSWER

An Ingress is a Kubernetes API object that manages external HTTP/HTTPS access to services, providing routing rules, SSL termination, and virtual hosting — all in one resource.

LoadBalancer Service vs Ingress:

- LoadBalancer: one cloud LB per Service = expensive and unscalable (10 services = 10 LBs, 10 public IPs)
- Ingress: ONE LB/IP for ALL services. Routes by path or hostname. Far more cost-efficient.

Ingress capabilities:

- Path-based routing: /api → api-service, /web → web-service
- Host-based routing: api.myapp.com → api-service, web.myapp.com → web-service
- SSL/TLS termination: handles HTTPS, terminates at Ingress, sends HTTP to pods
- Authentication, rate limiting (depending on controller)

Ingress Controller (required — Ingress resource alone does nothing):

- NGINX Ingress Controller — most popular, open source
- Traefik — modern, auto-discovers K8s resources
- AWS ALB Ingress Controller — uses AWS Application Load Balancer natively
- Kong — API gateway features
- HAProxy Ingress

The Ingress Controller is the actual proxy (NGINX/Traefik etc.) that reads Ingress rules and routes traffic.

kubectl / YAML

```
# Ingress YAML with path and host routing:
# apiVersion: networking.k8s.io/v1
# kind: Ingress
# metadata:
```

```
#   name: myapp-ingress
#   annotations:
#     nginx.ingress.kubernetes.io/rewrite-target: /
# spec:
#   ingressClassName: nginx
#   tls:
#   - hosts: ['api.myapp.com']
#     secretName: myapp-tls-secret
#   rules:
#   - host: api.myapp.com
#     http:
#       paths:
#       - path: /
#         pathType: Prefix
#         backend:
#           service:
#             name: api-service
#             port:
#               number: 80

kubectl get ingress
kubectl describe ingress myapp-ingress
```

KEY POINTS

- Ingress = one LB for many services (cost-efficient)
- Ingress Controller does the actual routing (NGINX, Traefik, ALB)
- Ingress alone does nothing — must have a controller running
- TLS termination at Ingress level — pods receive plain HTTP

INTERVIEW TIP

The cost argument wins interviews: 'Without Ingress, 20 microservices need 20 cloud load balancers = $20+/month each = $400+/month just for LBs. With Ingress, ONE load balancer routes to all 20 services based on path/hostname = $20/month total. At scale, Ingress saves thousands per month.'

Q16 How does DNS work in Kubernetes? How do pods find each other by name?

Medium

✓ANSWER

Kubernetes runs an internal DNS service (CoreDNS) that automatically creates DNS records for all Services and Pods, enabling name-based service discovery.

CoreDNS:

- Runs as a Deployment in the kube-system namespace
- Every pod has /etc/resolv.conf pointing to CoreDNS
- Creates DNS records for every Service automatically

Service DNS format:

- Short name: myservice (works within same namespace)
- Full name: myservice.mynamespace.svc.cluster.local
- Cross-namespace: myservice.other-namespace.svc.cluster.local

How it works in practice:

- You create a Service named 'postgres' in namespace 'production'
- Any pod in 'production' namespace can connect to 'postgres:5432'
- A pod in 'staging' namespace must use 'postgres.production.svc.cluster.local:5432'

Pod DNS (less commonly used):

- Format: <pod-ip-dashes>.<namespace>.pod.cluster.local
- Example: 10-244-1-15.default.pod.cluster.local
- StatefulSet pods get stable DNS: myapp-0.myapp-svc.default.svc.cluster.local

kubectl / YAML

```
# Check CoreDNS pods:
kubectl get pods -n kube-system -l k8s-app=kube-dns

# Test DNS from inside a pod:
kubectl exec -it <pod-name> -- nslookup kubernetes
kubectl exec -it <pod-name> -- nslookup myservice
kubectl exec -it <pod-name> -- cat /etc/resolv.conf

# Full service DNS resolution:
# myservice.default.svc.cluster.local
# myservice.production.svc.cluster.local

# Debug DNS issues:
kubectl run dnsutils --image=gcr.io/kubernetes-e2e-test-images/dnsutils --
restart=Never
kubectl exec -it dnsutils -- nslookup myservice
```

KEY POINTS

- CoreDNS runs in kube-system — internal DNS for the cluster
- Service DNS: <service>.<namespace>.svc.cluster.local
- Same namespace: just use service name. Cross-namespace: use full DNS
- StatefulSet pods get stable per-pod DNS via Headless Service

INTERVIEW TIP

Show you apply this: 'In my microservices, I configure connection strings using short service names within the same namespace — e.g., DB_HOST=postgres. Cross-namespace connections use the full DNS. I never hardcode IPs — if the pod restarts on a new IP, the DNS name still resolves correctly.'

Q17 What is a NetworkPolicy in Kubernetes? How do you restrict pod-to-pod communication?

Hard

✓ANSWER

By default, ALL pods in a Kubernetes cluster can communicate with ALL other pods — no restrictions. A NetworkPolicy is a resource that allows you to control which pods can communicate with which other pods and external endpoints.

How NetworkPolicy works:

- NetworkPolicy uses label selectors to target pods

- By default, pods are non-isolated — all traffic allowed
- Once a NetworkPolicy selects a pod, that pod becomes ISOLATED for the direction specified
- Only traffic explicitly allowed by the policy passes through

Policy directions:

- Ingress — controls incoming traffic to selected pods
- Egress — controls outgoing traffic from selected pods

Common patterns:

- Deny all ingress, allow only from specific pods (database protection)
- Allow only same-namespace communication
- Allow only specific ports (80, 443) from anywhere
- Block all egress except to specific IPs (prevent data exfiltration)

Important: NetworkPolicy requires a CNI plugin that supports it (Calico, Cilium, Weave). flannel does NOT support NetworkPolicy.

kubectl / YAML

```
# Deny all ingress to a namespace:
# kind: NetworkPolicy
# spec:
#   podSelector: {}   # select all pods
#   policyTypes: [Ingress]
#   # No ingress rules = deny all ingress

# Allow only from API pods:
# kind: NetworkPolicy
# spec:
#   podSelector:
#     matchLabels:
#       app: postgres
#   policyTypes: [Ingress]
#   ingress:
#   - from:
#     - podSelector:
#         matchLabels:
#           app: api
#     ports:
#     - port: 5432
```

KEY POINTS

- Default K8s: all pods can talk to all pods (no isolation)
- NetworkPolicy = firewall rules using label selectors
- Requires NetworkPolicy-capable CNI: Calico, Cilium (not flannel)
- Selecting a pod with a policy isolates it — only allowed traffic passes

INTERVIEW TIP

Security-first framing: 'I implement zero-trust networking in K8s — I start with a deny-all policy in each namespace, then add specific allow rules. Database pods only accept traffic from API pods on port 5432. Nothing else can reach the database. This is Defense in Depth for containerized workloads.'

Q18 What is a Headless Service in Kubernetes? When would you use one? Medium

ANSWER

A Headless Service is a Service with clusterIP: None. Unlike regular Services which provide a single stable virtual IP with load balancing, Headless Services return the IPs of all individual pod endpoints directly via DNS.

Regular Service DNS:

- Returns ClusterIP (virtual) → iptables routes to one pod
- myservice.default.svc.cluster.local → 10.96.0.100 (VIP)

Headless Service DNS:

- Returns multiple A records — one per pod endpoint
- myservice.default.svc.cluster.local → 10.244.1.2, 10.244.2.5, 10.244.3.8
- Also provides per-pod DNS: pod-name.myservice.default.svc.cluster.local

When to use Headless Services:

18. StatefulSets — enables stable per-pod DNS (myapp-0.myapp-svc)
19. Client-side load balancing — client (e.g., Cassandra driver) connects to all nodes directly
20. Databases with primary/replica awareness — app needs to know WHICH pod is primary
21. Service discovery — discover all pod IPs, not just one

StatefulSets REQUIRE a Headless Service to provide stable per-pod DNS entries.

kubectl / YAML

```
# Headless Service YAML:
# kind: Service
# metadata:
#   name: postgres-svc
# spec:
#   clusterIP: None     # THIS makes it headless
#   selector:
#     app: postgres
#   ports:
#   - port: 5432

# DNS lookup returns ALL pod IPs:
kubectl exec -it debug -- nslookup postgres-svc

# Per-pod DNS (StatefulSet):
# postgres-0.postgres-svc.default.svc.cluster.local
# postgres-1.postgres-svc.default.svc.cluster.local
```

KEY POINTS

- clusterIP: None = Headless Service
- Headless DNS returns all pod IPs (not a VIP)
- StatefulSets need Headless Services for stable pod DNS

INTERVIEW TIP

Connect to StatefulSet knowledge: 'Headless Services are the networking backbone of StatefulSets. A StatefulSet without a Headless Service loses stable pod identity — the primary use case. I always create the Headless Service first,

▸ Used when clients need to connect to specific pods, not any pod	then the StatefulSet that references it via serviceName.'

Q19 What are Taints and Tolerations? How do they control pod scheduling? Hard

✓ANSWER

Taints and Tolerations work together to ensure pods are NOT scheduled on inappropriate nodes. They are the opposite of Node Affinity (which attracts pods to nodes).

TAINTS — applied to NODES:

- A taint marks a node as having a special property that most pods should avoid
- Format: key=value:effect
- Three effects:
  - NoSchedule — pods without matching toleration will NOT be scheduled on this node
  - PreferNoSchedule — K8s tries to avoid scheduling, but may if no other option
  - NoExecute — existing pods without toleration are EVICTED; new pods not scheduled

TOLERATIONS — applied to PODS:

- A toleration says the pod CAN be scheduled on a tainted node
- Toleration must match the taint key, value, and effect
- Toleration does NOT guarantee scheduling — it just allows it

Real-world use cases:

- Control plane nodes — tainted NoSchedule so user workloads don't run on master
- GPU nodes — tainted so only ML workloads (with GPU tolerations) run there
- Dedicated nodes — tainted for specific teams or environments
- Node failure — NoExecute taint automatically added to failed nodes to evict pods

kubectl / YAML

```
# Taint a node (only GPU workloads allowed):
kubectl taint nodes gpu-node-1 gpu=true:NoSchedule

# Remove a taint:
kubectl taint nodes gpu-node-1 gpu=true:NoSchedule-

# View node taints:
kubectl describe node gpu-node-1 | grep Taints

# Toleration in pod spec:
# tolerations:
# - key: 'gpu'
#   operator: 'Equal'
#   value: 'true'
#   effect: 'NoSchedule'

# Control plane taint (added by kubeadm):
# node-role.kubernetes.io/control-plane:NoSchedule
```

KEY POINTS

- Taint on node = repel pods. Toleration on pod = allow on tainted node.
- NoSchedule = won't schedule. NoExecute = evicts existing pods.
- Control plane has taint:NoSchedule by default
- Toleration allows scheduling — doesn't guarantee it (use Node Affinity for that)

INTERVIEW TIP

Combine with affinity: 'Taints/tolerations say what pods CAN run on a node. Node Affinity says what pods WANT to run on a node. I use both together: taint GPU nodes so only ML pods can run there (toleration), then use Node Affinity to ensure ML pods PREFER to run on GPU nodes (attraction).'

04 Storage in Kubernetes

Q20–Q24 · PersistentVolumes, PVCs, StorageClass, ConfigMaps, Secrets

Q20 What is a PersistentVolume (PV), PersistentVolumeClaim (PVC), and StorageClass?

Medium

✓**ANSWER**

Kubernetes separates storage provisioning (PV) from storage consumption (PVC) for clean abstraction.

PERSISTENTVOLUME (PV) — the actual storage resource:

- Represents a piece of actual storage (AWS EBS, GCP disk, NFS, local disk)
- Created by cluster admin (static provisioning) or dynamically by StorageClass
- Has properties: capacity, accessModes, reclaim policy, storage class
- Cluster-scoped (not namespace-specific)

PERSISTENTVOLUMECLAIM (PVC) — request for storage:

- Created by application developers — requests specific amount and type of storage
- K8s finds (or creates) a matching PV and binds them
- Namespace-scoped
- Pod mounts the PVC, not the PV directly

STORAGECLASS — dynamic provisioning:

- Defines the type of storage and how to provision it automatically
- When PVC references a StorageClass, K8s automatically creates a PV on demand
- Examples: aws-ebs (gp2/gp3), gcp-pd (standard/ssd), azure-disk
- Eliminates manual PV creation — cloud storage is provisioned automatically

Access Modes:

- ReadWriteOnce (RWO) — one node can read/write (most cloud block storage)
- ReadOnlyMany (ROX) — many nodes can read
- ReadWriteMany (RWX) — many nodes can read/write (NFS, EFS)

kubectl / YAML

```
# StorageClass (AWS EBS):
# kind: StorageClass
# provisioner: ebs.csi.aws.com
# parameters:
#   type: gp3

# PVC requesting 20Gi:
# kind: PersistentVolumeClaim
# spec:
#   storageClassName: gp3
#   accessModes: [ReadWriteOnce]
#   resources:
#     requests:
```

```
#       storage: 20Gi

# Pod using the PVC:
# volumes:
# - name: data
#   persistentVolumeClaim:
#     claimName: my-pvc

# Check PVC binding:
kubectl get pvc
kubectl describe pvc my-pvc
```

KEY POINTS

- PV = actual storage. PVC = request. StorageClass = dynamic provisioner.
- StorageClass auto-creates PV when PVC is created (dynamic provisioning)
- RWO = one node. RWX = many nodes. Cloud block storage = RWO only.
- PVC binds to PV — pod mounts PVC not PV directly

INTERVIEW TIP

Dynamic provisioning is the real-world approach. Say: 'In production, I never create PVs manually. I define StorageClasses for different storage tiers (fast SSD, standard HDD). When developers create PVCs, K8s automatically provisions the right cloud storage. It's storage-as-code — declarative and automated.'

Q21 What is a ConfigMap in Kubernetes? How do you use it in pods?

Easy

ANSWER

A ConfigMap stores non-sensitive configuration data as key-value pairs. It decouples configuration from container images, so the same image can run in dev, staging, and production with different configs.

What to store in ConfigMaps:

- Application config files (nginx.conf, application.properties)
- Environment-specific settings (feature flags, log levels, timeouts)
- Connection strings without credentials
- Command-line arguments

Three ways to use ConfigMaps in pods:

22. Environment variables — inject config as env vars:

- env: - name: LOG_LEVEL, valueFrom: configMapKeyRef

23. Volume mount — inject as files:

- Mount ConfigMap as a file inside the container
- Changes to ConfigMap are reflected in the mounted file (eventually consistent)
- Best for config files (nginx.conf, application.yaml)

24. Command arguments — pass ConfigMap values as container args

Important: ConfigMaps are NOT encrypted. Do NOT store passwords, tokens, or secrets in ConfigMaps — use Secrets for that.

kubectl / YAML

```
# Create ConfigMap from literal:
kubectl create configmap app-config --from-literal=LOG_LEVEL=INFO --from-
literal=PORT=8080

# Create from file:
kubectl create configmap nginx-config --from-file=nginx.conf

# Use as env vars in pod:
# env:
# - name: LOG_LEVEL
#   valueFrom:
#     configMapKeyRef:
#       name: app-config
#       key: LOG_LEVEL

# Use as volume mount:
# volumes:
# - name: config
#   configMap:
#     name: nginx-config
# volumeMounts:
# - name: config
#   mountPath: /etc/nginx/conf.d
```

KEY POINTS

- ConfigMap = non-sensitive config. Secret = sensitive data.
- ConfigMap changes auto-update mounted volume files (eventual consistency)
- Env var injection does NOT auto-update — requires pod restart
- Maximum ConfigMap size: 1MB

INTERVIEW TIP

Show separation of config from code: 'I store all environment-specific configuration in ConfigMaps — log levels, feature flags, timeouts. The Docker image is identical in all environments. Only the ConfigMap changes between dev/staging/prod. This means I never rebuild the image for config changes — just update the ConfigMap and rolling restart the deployment.'

Q22 What is a Kubernetes Secret? How is it different from a ConfigMap? Medium

✓ANSWER

A Secret stores sensitive data such as passwords, OAuth tokens, SSH keys, and TLS certificates. It is similar to a ConfigMap but is intended for confidential data.

ConfigMap vs Secret:

- ConfigMap: plain text, non-sensitive config (feature flags, log levels)
- Secret: base64-encoded (NOT encrypted), sensitive data (passwords, tokens, certs)

Important: Secrets are base64-ENCODED, NOT ENCRYPTED by default:

- Anyone with access to kubectl can decode a Secret trivially

- Enable etcd encryption at rest for true Secret encryption
- Use external secret management: HashiCorp Vault, AWS Secrets Manager, External Secrets Operator

Secret types:

- Opaque — generic user-defined data (most common)
- kubernetes.io/tls — TLS certificates and keys
- kubernetes.io/dockerconfigjson — Docker registry credentials
- kubernetes.io/service-account-token — ServiceAccount tokens

Best practices for Secrets:

- Enable etcd encryption at rest
- Use RBAC to restrict who can read Secrets
- Use External Secrets Operator to sync from AWS/GCP/Vault
- Never commit Secrets to git — use sealed-secrets or external secret management

kubectl / YAML

```
# Create secret:
kubectl create secret generic db-creds \
  --from-literal=password=mysupersecret \
  --from-literal=username=admin

# Base64 is NOT encryption — decode easily:
echo 'bXlzdXBlcnNlY3JldA==' | base64 -d

# Use secret as env var:
# env:
# - name: DB_PASSWORD
#   valueFrom:
#     secretKeyRef:
#       name: db-creds
#       key: password

# TLS secret:
kubectl create secret tls myapp-tls \
  --cert=tls.crt --key=tls.key

# Docker registry secret:
kubectl create secret docker-registry regcred \
  --docker-server=my-registry.com \
  --docker-username=admin \
  --docker-password=mypassword
```

KEY POINTS

- Secrets are base64-encoded — NOT encrypted by default
- Enable etcd encryption at rest for real Secret security
- External Secrets Operator: sync from Vault/AWS SM to K8s Secrets
- RBAC to restrict kubectl get secret access

INTERVIEW TIP

The 'base64 is not encryption' point is critical for security awareness. Say: 'Out of the box, K8s Secrets are just base64 — anyone with kubectl can decode them. In production, I use External Secrets Operator to pull secrets from AWS Secrets Manager at runtime. The actual secret value never touches git or the K8s manifest — only a reference to the secret does.'

Q23 What is the reclaim policy for PersistentVolumes? What happens when a PVC is deleted?

Medium

✓**ANSWER**

The reclaim policy defines what happens to a PersistentVolume when the PersistentVolumeClaim that was bound to it is deleted.

Three reclaim policies:

25. Retain (most common for production):
    - PV is NOT deleted when PVC is deleted
    - PV enters 'Released' state — data is preserved
    - Administrator must manually reclaim the PV (delete and recreate, or clean and re-bind)
    - Best for: databases, important data you want to keep

26. Delete:
    - PV AND the underlying cloud storage (EBS volume, GCP disk) are automatically deleted
    - Default for dynamically provisioned PVs with StorageClass
    - Best for: temporary or easily regeneratable data

27. Recycle (DEPRECATED):
    - Runs rm -rf on the volume and makes it available again
    - Deprecated — use dynamic provisioning instead

PVC lifecycle states:

- Pending — waiting for a PV to bind to
- Bound — successfully bound to a PV
- Lost — underlying PV has been deleted while PVC still exists

kubectl / YAML

```
# Check PV reclaim policy:
kubectl get pv
# RECLAIM POLICY column shows Retain/Delete

# Change reclaim policy of existing PV:
kubectl patch pv <pv-name> -p
'{"spec":{"persistentVolumeReclaimPolicy":"Retain"}}'

# PV states after PVC deletion:
kubectl get pv
# STATUS: Released (data intact, not auto-deleted)

# StorageClass default reclaim policy:
# kind: StorageClass
# reclaimPolicy: Retain  # override default Delete
```

KEY POINTS

INTERVIEW TIP

- Retain = keep data, manual cleanup. Delete = auto-delete storage.
- Default for dynamic provisioning StorageClass = Delete
- Check reclaim policy before deleting PVCs in production!
- PV in 'Released' state: data intact but not re-bindable automatically

Production war story: 'I always set reclaim policy to Retain for database PVs. I've seen developers delete a PVC thinking it's safe, and the Delete policy auto-deleted the EBS volume with all production data. With Retain, even if the PVC is accidentally deleted, the underlying volume and data survive.'

Q24 How do you share storage between multiple pods in Kubernetes? Medium

✓ANSWER

Sharing storage between multiple pods in Kubernetes depends on the access mode of the volume and the storage backend used.

AccessMode determines sharing capability:

- ReadWriteOnce (RWO) — ONE node can mount read-write. Multiple pods on the SAME node can share it.
- ReadOnlyMany (ROX) — MANY nodes can mount read-only. All pods get the same data.
- ReadWriteMany (RWX) — MANY nodes can mount read-write. True multi-pod, multi-node sharing.

Storage backends that support RWX:

- AWS EFS (Elastic File System) — NFS-based, RWX support
- Azure Files — NFS/SMB, RWX support
- GCP Filestore — NFS, RWX support
- NFS server — on-premise or self-managed
- CephFS — distributed file system

AWS EBS, Azure Disk, GCP PD — ONLY support RWO. Cannot share between multiple nodes.

Pattern for read-only sharing:

- Use ReadOnlyMany — one pod writes to the volume (or pre-populate), many pods read from it
- Common for shared config files, static content, ML model files

kubectl / YAML

```
# PVC with ReadWriteMany (needs EFS/NFS backend):
# kind: PersistentVolumeClaim
# spec:
#   accessModes:
#   - ReadWriteMany
#   storageClassName: efs-sc  # EFS StorageClass
#   resources:
#     requests:
#       storage: 50Gi

# Multiple pods sharing same PVC:
# Pod A: volumeMounts - name: shared, mountPath: /data
# Pod B: volumeMounts - name: shared, mountPath: /data
# Both reference same PVC by claimName

# Check access modes of a PV:
kubectl get pv -o wide
```

KEY POINTS

- RWX required for multi-pod, multi-node write sharing
- AWS EBS/Azure Disk = RWO only. EFS/Azure Files = RWX.
- RWO allows multiple pods on SAME node to share
- Use EFS/NFS for shared storage in Kubernetes (ML models, media files)

INTERVIEW TIP

Show cloud knowledge: 'For shared storage on AWS EKS, I use AWS EFS with the EFS CSI driver. It supports ReadWriteMany — multiple pods across different nodes can read and write simultaneously. I use this for shared ML model storage where training pods write models and inference pods read them concurrently.'

05 Scaling & Updates

Q25–Q29 · HPA, VPA, Rolling Updates, Rollbacks, Resource Management

Q25 What is a Horizontal Pod Autoscaler (HPA)? How does it work? Medium

✓ANSWER

A Horizontal Pod Autoscaler (HPA) automatically scales the number of pod replicas in a Deployment, ReplicaSet, or StatefulSet based on observed metrics (CPU, memory, or custom metrics).

How HPA works:

28. HPA controller runs in a control loop (default every 15 seconds)
29. Fetches current metrics from Metrics Server (CPU/memory) or custom metrics API
30. Calculates desired replica count: desiredReplicas = ceil(currentReplicas × (currentMetric / targetMetric))
31. If desired != current, updates the Deployment replica count
32. Kubernetes scheduler places new pods on available nodes

HPA requires Metrics Server to be installed in the cluster.

HPA scaling behavior:

- Scale up: aggressive (responds quickly to load spikes)
- Scale down: conservative (waits before scaling down to avoid thrashing)
- Default scale-down stabilization: 5 minutes

Example: target CPU 50%, current replicas 2, current CPU 80%:

desiredReplicas = ceil(2 × (80/50)) = ceil(3.2) = 4 replicas

Custom metrics HPA: scale on requests-per-second, queue depth, latency using Prometheus Adapter or KEDA.

kubectl / YAML

```
# Create HPA (target 70% CPU):
kubectl autoscale deployment myapp --cpu-percent=70 --min=2 --max=10

# HPA YAML with CPU and memory:
# kind: HorizontalPodAutoscaler
# spec:
#   scaleTargetRef:
#     apiVersion: apps/v1
#     kind: Deployment
#     name: myapp
#   minReplicas: 2
#   maxReplicas: 10
#   metrics:
#   - type: Resource
#     resource:
```

```
#       name: cpu
#       target:
#         type: Utilization
#         averageUtilization: 70

# Watch HPA in action:
kubectl get hpa -w
kubectl describe hpa myapp
```

KEY POINTS

- HPA requires Metrics Server installed
- Default check interval: 15 seconds
- Scale-down stabilization window: 5 minutes (prevents thrashing)
- KEDA enables scaling on custom metrics like queue depth

INTERVIEW TIP

Show real-world usage: 'I set HPA target CPU at 70% for my API service. At 70%, pods are busy but not overwhelmed. This gives headroom to handle traffic spikes while new pods are scaling up (which takes 30-60 seconds). I also set minReplicas=2 so there's always redundancy, even at low traffic.'

Q26 What is a rolling update strategy in Kubernetes? How do you configure it? Medium

✓ANSWER

A rolling update is the default Deployment update strategy in Kubernetes. It incrementally replaces old pods with new pods, ensuring the application remains available during the update — zero-downtime deployment.

How rolling updates work:

33. You update the Deployment image version
34. K8s creates a new ReplicaSet with the new image
35. New pods come up one by one (or in batches)
36. Old pods are terminated one by one as new ones become ready
37. Traffic shifts gradually from old pods to new pods via readiness probes
38. Process continues until all pods run new version

Key configuration parameters:

- maxSurge — how many EXTRA pods can exist above desired count during update (default: 25%)
- maxUnavailable — how many pods can be UNAVAILABLE during update (default: 25%)

Example: 4 replicas, maxSurge=1, maxUnavailable=0:

- Temporarily creates 5 pods (4 old + 1 new)
- Waits for new pod to be ready
- Terminates 1 old pod → 3 old + 1 new
- Continues until 4 new pods running

Zero-downtime key: set maxUnavailable=0 to ensure traffic is never disrupted.

kubectl / YAML

```
# Deployment rolling update strategy:
```

```
# spec:
#   strategy:
#     type: RollingUpdate
#     rollingUpdate:
#       maxSurge: 1         # allow 1 extra pod above desired
#       maxUnavailable: 0   # no pods can be unavailable = zero downtime

# Trigger rolling update:
kubectl set image deployment/myapp myapp=myapp:v2

# Monitor rollout:
kubectl rollout status deployment/myapp

# Pause rollout (check the new pods):
kubectl rollout pause deployment/myapp

# Resume rollout:
kubectl rollout resume deployment/myapp

# Rollback:
kubectl rollout undo deployment/myapp
```

KEY POINTS

- Default update strategy = RollingUpdate (not Recreate)
- maxSurge: extra pods during update. maxUnavailable: pods that can be down.
- maxUnavailable=0 guarantees zero downtime (requires extra capacity)
- Readiness probes control when old pods receive no more traffic

INTERVIEW TIP

Always mention readiness probes: 'Rolling updates only work for zero-downtime if readiness probes are configured. Without them, K8s sends traffic to new pods before they're ready. I always add a readiness probe that checks the /health endpoint returns 200 before K8s removes the old pod from the Service endpoints.'

Q27 How do you perform a blue-green or canary deployment in Kubernetes? Hard

✓ANSWER

Kubernetes supports advanced deployment patterns beyond rolling updates. Blue-green and canary deployments give you more control over traffic shifting.

BLUE-GREEN DEPLOYMENT:

- Run two complete environments: Blue (current) and Green (new version)
- Both are running simultaneously
- Switch traffic from Blue to Green instantly by updating Service selector
- Zero-downtime cutover. Instant rollback by switching selector back.
- Requires 2x resources (both environments running)

Implementation: Two Deployments (myapp-blue, myapp-green). Service selector switches between them.

CANARY DEPLOYMENT:

- Release new version to a SMALL subset of users first (e.g., 10%)
- Monitor for errors, latency, business metrics
- If healthy, gradually increase traffic to new version
- If problems, rollback affects only the small canary percentage

Implementation options:

- Kubernetes-native: run 1 canary pod alongside 9 stable pods (1/10 = 10% traffic via Service)
- Ingress-based: NGINX Ingress canary annotations for precise traffic splitting
- Argo Rollouts: sophisticated progressive delivery with analysis
- Istio: precise traffic splitting using VirtualService weight rules

kubectl / YAML

```
# Blue-Green via Service selector:
# Service selector points to blue:
# selector:
#   app: myapp
#   version: blue  # switch to 'green' for cutover

# NGINX Ingress Canary (10% to new version):
# apiVersion: networking.k8s.io/v1
# kind: Ingress
# metadata:
#   annotations:
#     nginx.ingress.kubernetes.io/canary: 'true'
#     nginx.ingress.kubernetes.io/canary-weight: '10'

# Canary with Argo Rollouts:
# kind: Rollout
# spec:
#   strategy:
#     canary:
#       steps:
#       - setWeight: 20   # 20% traffic to new version
#       - pause: {duration: 10m}
#       - setWeight: 80
#       - pause: {duration: 10m}
```

KEY POINTS

- Blue-green: instant cutover via selector change. 2x resource cost.
- Canary: gradual traffic shift to new version. Safer for risky changes.
- Ingress annotations enable % traffic split for canary
- Argo Rollouts and Istio for enterprise-grade progressive delivery

INTERVIEW TIP

Show tool knowledge: 'For canary deployments at scale, I use Argo Rollouts. It integrates with Prometheus to automatically promote or rollback based on error rate metrics — if error rate exceeds 5% in the canary, it auto-rolls back without human intervention. This is what GitOps-based progressive delivery looks like in practice.'

Q28 What is a VerticalPodAutoscaler (VPA)? How does it differ from HPA? Medium

✅ANSWER

A Vertical Pod Autoscaler (VPA) automatically adjusts the CPU and memory requests and limits of pods based on their actual usage. While HPA scales OUT (more pods), VPA scales UP (bigger pods).

VPA vs HPA:

- HPA: scales number of replicas horizontally (2 pods → 10 pods)
- VPA: adjusts resource requests/limits vertically (256Mi → 512Mi memory)

VPA modes:

- Off — only provides recommendations, does not apply changes
- Initial — sets resources only when pods are first created, does not change running pods
- Auto — automatically applies recommendations. Note: requires pod restart (current limitation)
- Recreate — evicts pods when resource change is needed, allows new pods with updated resources

VPA use cases:

- Right-sizing containers when you don't know the correct resource requests initially
- Stateful workloads that are hard to scale horizontally (databases, Kafka brokers)
- Workloads with variable resource needs throughout the day

Important limitation: VPA cannot currently update running pod resources in-place — it evicts and recreates pods. HPA + VPA can conflict — use carefully.

kubectl / YAML

```
# VPA YAML:
# kind: VerticalPodAutoscaler
# spec:
#   targetRef:
#     apiVersion: apps/v1
#     kind: Deployment
#     name: myapp
#   updatePolicy:
#     updateMode: 'Off'   # recommendation only

# Get VPA recommendations:
kubectl describe vpa myapp-vpa
# Look for:
# Recommendation:
#   Container Recommendations:
#   - Target:
#       cpu: 150m
#       memory: 400Mi

# Install VPA:
# kubectl apply -f https://github.com/kubernetes/autoscaler/tree/master/vertical-
pod-autoscaler
```

KEY POINTS

- VPA = scale UP (bigger). HPA = scale OUT (more pods).
- VPA requires pod restart to apply resource changes (current limitation)

INTERVIEW TIP

Show nuanced understanding: 'I use VPA in recommendation mode first to understand actual resource usage before setting limits. After a week of data, VPA tells me the app consistently needs

▸ VPA 'Off' mode = recommendations only — great for right-sizing analysis ▸ KEDA is often preferred over HPA for event-driven scaling	400Mi memory but I allocated 256Mi. I update the Deployment manually based on VPA recommendations. I avoid VPA Auto mode in production because the pod evictions are disruptive.'

Q29 What are Kubernetes resource quotas and LimitRanges? How do they enforce governance?

Medium

✓ANSWER

ResourceQuota and LimitRange are admission controllers that enforce resource governance at the namespace level, preventing any single team or application from consuming all cluster resources.

RESOURCEQUOTA — limits total resources for a namespace:

- ▸ Sets maximum total CPU, memory, pods, services, PVCs across ALL objects in the namespace
- ▸ Prevents one team's namespace from consuming all cluster capacity
- ▸ If a new object would exceed the quota, API Server rejects it

LIMITRANGE — sets default and max/min per pod/container:

- ▸ Sets default requests/limits for containers that don't specify them
- ▸ Sets minimum and maximum allowed requests/limits per container
- ▸ Prevents running containers with no limits (noisy neighbor)
- ▸ Prevents requesting more resources than a node can provide

Why both are needed:

- ▸ ResourceQuota: prevents total namespace overconsumption
- ▸ LimitRange: ensures every container has appropriate resource boundaries
- ▸ Without LimitRange, containers without resource specs consume cluster capacity with no bounds

kubectl / YAML

```
# ResourceQuota:
# kind: ResourceQuota
# metadata:
#   namespace: team-a
# spec:
#   hard:
#     requests.cpu: '10'
#     requests.memory: 20Gi
#     limits.cpu: '20'
#     limits.memory: 40Gi
#     pods: '50'
#     services: '10'

# LimitRange (default + max per container):
# kind: LimitRange
# spec:
#   limits:
#   - type: Container
#     default:
```

```
#         cpu: 500m
#         memory: 256Mi
#       max:
#         cpu: '2'
#         memory: 4Gi

kubectl describe resourcequota -n team-a
```

KEY POINTS

- ResourceQuota = total namespace limits. LimitRange = per-pod limits.
- LimitRange sets defaults for containers with no resource specs
- API Server rejects objects that would exceed ResourceQuota
- Use both in multi-tenant clusters to prevent noisy neighbor issues

INTERVIEW TIP

Frame this as multi-team governance: 'In our multi-tenant cluster, each team gets a namespace with a ResourceQuota — team-a gets 10 CPU, 20Gi memory max. A LimitRange ensures no single container requests more than 2 CPU or 4Gi. Without these, one team could starve all others. Quotas are infrastructure governance-as-code.'

06 Security & RBAC

Q30–Q34 · Namespaces, Roles, ClusterRoles, ServiceAccounts, SecurityContext, RBAC

Q30 What is RBAC in Kubernetes? Explain Role, ClusterRole, RoleBinding, ClusterRoleBinding.

Hard

✓**ANSWER**

RBAC (Role-Based Access Control) is Kubernetes' authorization mechanism. It controls who (subjects) can perform what actions (verbs) on which resources.

Four core RBAC objects:

ROLE — namespace-scoped permissions:

- Defines what actions are allowed on which resources within ONE namespace
- Example: allow get, list, watch on pods in 'production' namespace

CLUSTERROLE — cluster-scoped permissions:

- Same as Role but applies cluster-wide (all namespaces) OR to cluster-level resources (nodes, PVs)
- Example: cluster-admin, view, edit are built-in ClusterRoles

ROLEBINDING — binds a Role or ClusterRole to subjects in ONE namespace:

- Subject types: User, Group, ServiceAccount
- Can bind a ClusterRole to a specific namespace (scope-down pattern)

CLUSTERROLEBINDING — binds a ClusterRole to subjects cluster-wide:

- Gives subjects the ClusterRole permissions across ALL namespaces

Subjects:

- User — human user (authenticated via certificates, OIDC)
- Group — group of users
- ServiceAccount — pod identity for machine-to-machine auth

kubectl / YAML

```
# Create Role (allow reading pods in production namespace):
kubectl create role pod-reader \
  --verb=get,list,watch \
  --resource=pods \
  --namespace=production

# Bind role to user:
kubectl create rolebinding jane-pod-reader \
  --role=pod-reader \
  --user=jane \
  --namespace=production
```

```
# Check permissions:
kubectl auth can-i get pods --namespace=production --as=jane

# View RBAC:
kubectl get roles,rolebindings -n production
kubectl describe clusterrole cluster-admin
```

KEY POINTS

- Role = namespace. ClusterRole = cluster-wide or cluster resources.
- Binding = connect Role/ClusterRole to subjects (user/SA/group)
- ServiceAccount = pod's identity for API access
- Always follow least privilege — grant minimum permissions needed

INTERVIEW TIP

Show the mental model: 'I always think in three questions: WHO needs access (user/SA)? WHAT do they need (get pods? create services?)? WHERE (one namespace or cluster-wide)? Then: Role for one namespace, ClusterRole for cluster-wide. RoleBinding to scope. Least privilege by default.'

Q31 What is a ServiceAccount in Kubernetes? How do pods use them? Medium

✓ANSWER

A ServiceAccount is a non-human identity for processes running inside pods. It allows pods to authenticate to the Kubernetes API Server and other services.

Why ServiceAccounts exist:

- Pods often need to interact with the K8s API (list other pods, read secrets, update ConfigMaps)
- Human user accounts are not appropriate for automated processes
- ServiceAccounts give pods a controlled, auditable identity

How it works:

- Every namespace has a 'default' ServiceAccount
- If you don't specify a ServiceAccount, pods use 'default'
- K8s automatically mounts a ServiceAccount token into every pod
- Token is mounted at: /var/run/secrets/kubernetes.io/serviceaccount/token
- Pod uses this token to authenticate to the K8s API

RBAC + ServiceAccount pattern:

39. Create ServiceAccount: my-app-sa
40. Create Role: permissions the app needs (read ConfigMaps, list pods)
41. Create RoleBinding: bind Role to ServiceAccount
42. Assign ServiceAccount to Deployment spec

Best practice: create dedicated ServiceAccounts for each application — not using 'default'.

kubectl / YAML

```
# Create ServiceAccount:
kubectl create serviceaccount my-app-sa -n production
```

```
# Assign to Deployment:
# spec:
#   template:
#     spec:
#       serviceAccountName: my-app-sa

# Disable auto-mount if pod doesn't need API access:
# spec:
#   automountServiceAccountToken: false

# Token inside pod:
kubectl exec -it <pod> -- cat /run/secrets/kubernetes.io/serviceaccount/token

# RBAC: bind role to ServiceAccount:
kubectl create rolebinding my-app-binding \
  --role=pod-reader \
  --serviceaccount=production:my-app-sa \
  --namespace=production
```

KEY POINTS

- Every pod has a ServiceAccount (default if not specified)
- Token auto-mounted at /var/run/secrets/kubernetes.io/serviceaccount/
- ServiceAccount + Role + RoleBinding = pod API access control
- Use automountServiceAccountToken: false if pod doesn't need API access

INTERVIEW TIP

Security best practice: 'I always create application-specific ServiceAccounts with minimum RBAC permissions. I never use the default ServiceAccount — it has no permissions by default but using it makes it harder to audit access. I also set automountServiceAccountToken: false for pods that don't need API access — reducing the token attack surface.'

Q32 What is a SecurityContext in Kubernetes? What security settings can you configure?

Hard

✓ANSWER

A SecurityContext defines privilege and access control settings for a Pod or Container. It applies Linux security features at the pod/container level.

Settings available at POD level (apply to all containers):

- runAsUser — UID to run all containers as (e.g., 1000 = non-root)
- runAsGroup — GID for all containers
- runAsNonRoot — fail if container tries to run as root
- fsGroup — volume files owned by this GID
- sysctls — kernel parameter overrides

Settings at CONTAINER level:

- privileged — run as privileged (like root on host) — NEVER in production
- allowPrivilegeEscalation — prevent sudo-like escalation (set false)
- readOnlyRootFilesystem — prevent writes to container root FS

- capabilities — add or drop Linux capabilities
  - drop: ALL — remove all capabilities
  - add: NET_BIND_SERVICE — add only needed capability

Pod Security Standards (replaced PodSecurityPolicy):

- Privileged — no restrictions
- Baseline — prevents known privilege escalation
- Restricted — heavily restricted, follows security best practices

kubectl / YAML

```
# Secure pod SecurityContext:
# spec:
#   securityContext:
#     runAsNonRoot: true
#     runAsUser: 1000
#     runAsGroup: 3000
#     fsGroup: 2000
#   containers:
#   - name: app
#     securityContext:
#       allowPrivilegeEscalation: false
#       readOnlyRootFilesystem: true
#       capabilities:
#         drop:
#         - ALL
#         add:
#         - NET_BIND_SERVICE

# Apply Pod Security Standards to namespace:
kubectl label namespace production pod-security.kubernetes.io/enforce=restricted
```

KEY POINTS

- runAsNonRoot: true prevents root container execution
- readOnlyRootFilesystem: true prevents filesystem tampering
- capabilities drop:ALL + add only needed = least privilege
- Pod Security Standards: Privileged / Baseline / Restricted

INTERVIEW TIP

Show defense-in-depth: 'I enforce SecurityContext on all production pods: runAsNonRoot, readOnlyRootFilesystem, drop ALL capabilities and add back only NET_BIND_SERVICE if needed. Combined with NetworkPolicy, this means even if an attacker compromises the container, they have no root, can't write to filesystem, and can't make privileged syscalls. Defense in depth.'

Q33 What is a Kubernetes Namespace? How does it provide multi-tenancy? Easy

✓ANSWER

A Namespace is a virtual partition within a single Kubernetes cluster that isolates groups of resources. They are a way to divide cluster resources between multiple users, teams, or projects.

What Namespaces provide:

- Resource isolation — resources in different namespaces don't conflict by name

- Access control — RBAC policies scoped per namespace
- Resource quotas — CPU/memory limits per namespace
- Network policies — traffic isolation between namespaces

Default namespaces in every cluster:

- default — where objects go if no namespace is specified
- kube-system — Kubernetes system components (API Server, CoreDNS, kube-proxy)
- kube-public — publicly readable, for cluster info
- kube-node-lease — node heartbeat objects

Multi-tenancy patterns:

- One namespace per environment: dev, staging, production
- One namespace per team: team-frontend, team-backend, team-data
- One namespace per application: myapp-frontend, myapp-api, myapp-database

Namespace limitations:

- Not strong isolation — workloads share the same kernel and hardware
- Cluster-level resources (nodes, PVs, ClusterRoles) are NOT namespaced

kubectl / YAML

```
# Create namespace:
kubectl create namespace production

# Work in a namespace:
kubectl get pods -n production
kubectl get all -n production

# Set default namespace for kubectl:
kubectl config set-context --current --namespace=production

# List all namespaces:
kubectl get namespaces

# Namespace in YAML:
# metadata:
#   name: myapp
#   namespace: production

# Resources NOT namespaced:
kubectl api-resources --namespaced=false
```

KEY POINTS

- kube-system = K8s system components. Never delete.
- Cluster-scoped resources: Nodes, PVs, ClusterRoles (not namespaced)
- Namespaces provide soft isolation — not VM-level security
- Cross-namespace DNS: service.namespace.svc.cluster.local

INTERVIEW TIP

Show production structure: 'Our cluster has namespaces per team and per environment: frontend-prod, frontend-staging, backend-prod, backend-staging, data-prod. Each namespace has ResourceQuota, LimitRange, and NetworkPolicy. RBAC gives each team admin access to their namespaces but not others. This gives autonomy with guardrails.'

Q34 How does Kubernetes authentication and authorization work?

Hard

✓ANSWER

Kubernetes uses a multi-stage security pipeline: Authentication → Authorization (RBAC) → Admission Control. Every API request passes through all three.

STAGE 1 — AUTHENTICATION (Who are you?):

- Client certificates (X.509) — used by kubectl, kubeadm, system components
- Bearer tokens — ServiceAccount tokens, OIDC tokens
- OIDC (OpenID Connect) — integrate with SSO (Okta, Google, Azure AD, GitHub)
- Webhook token authentication — custom external auth service

STAGE 2 — AUTHORIZATION (What can you do?) — RBAC:

- Once authenticated, request is checked against RBAC policies
- If no RBAC policy allows the action → 403 Forbidden
- Modes: RBAC (standard), ABAC (deprecated), Node authorization

STAGE 3 — ADMISSION CONTROL (Should this be allowed?):

- Validates and mutates resources before persisting to etcd
- Built-in: ResourceQuota, LimitRanger, NamespaceLifecycle, PodSecurity
- Custom: ValidatingWebhook, MutatingWebhook (OPA Gatekeeper, Kyverno)

Flow: kubectl → API Server (TLS) → Authentication → RBAC → Admission → etcd

kubectl / YAML

```
# Check current user/context:
kubectl config current-context
kubectl config view

# Test permissions:
kubectl auth can-i create pods
kubectl auth can-i get secrets -n production --as=jane
kubectl auth can-i '*' '*' # check if cluster-admin

# View OIDC configuration:
kubectl get configmap -n kube-system oidc-config

# Check admission controllers:
kubectl describe pod kube-apiserver-master -n kube-system | grep admission
```

KEY POINTS

- Three stages: Authentication → RBAC Authorization → Admission Control
- OIDC integrates with corporate SSO (Okta, Azure AD)
- Admission webhooks enable custom policy enforcement (OPA, Kyverno)

INTERVIEW TIP

Show enterprise knowledge: 'In production, I integrate K8s with our corporate identity provider via OIDC. Engineers authenticate with their SSO credentials — no managing separate K8s users. RBAC maps SSO groups to ClusterRoles. OPA Gatekeeper as a ValidatingWebhook enforces

▸ All API traffic encrypted with TLS — no plaintext access	policies like no privileged containers and required resource limits — policy-as-code.'

07 Real-World & Troubleshooting

Q35–Q40 · CrashLoopBackOff, Pending pods, OOMKill, debugging, kubectl tips

Q35 What is CrashLoopBackOff? How do you troubleshoot it? Hard

✓ANSWER

CrashLoopBackOff is a Kubernetes pod status indicating the container is crashing repeatedly. K8s restarts it, it crashes again, K8s waits increasingly longer (exponential backoff: 10s, 20s, 40s, 80s, 160s, 300s max) before retrying.

Common causes of CrashLoopBackOff:

43. Application error on startup — missing env var, missing config file, wrong command
44. Missing dependencies — DB not reachable, required service down
45. Wrong CMD/ENTRYPOINT in Dockerfile — container starts and exits immediately
46. OOM kill — container needs more memory than its limit allows
47. Readiness probe failing causing restart? (No — that's restart policy, not CrashLoop)
48. Permissions error — app can't read mounted config/secret files

Troubleshooting steps:

49. kubectl describe pod <name> → Events section shows exit code and reason
50. kubectl logs <pod> → see application error before crash
51. kubectl logs <pod> --previous → logs from previous crashed container
52. Exit code 1 = application error. Exit code 137 = OOMKilled. Exit code 126/127 = wrong command.
53. kubectl exec -it <pod> -- bash → if pod briefly stays up, get inside
54. Override command: kubectl run debug --image=myimage -- sleep 3600 → inspect without app starting

kubectl / YAML

```
# Check pod status:
kubectl get pods
# STATUS: CrashLoopBackOff

# Get events and exit code:
kubectl describe pod <pod-name>
# Events: Back-off restarting failed container
# Last State: Terminated, Exit Code: 1

# Get logs from crashed container:
kubectl logs <pod-name> --previous

# Run debugging container (override CMD):
kubectl run debug --image=myapp:latest -- sleep 3600
kubectl exec -it debug -- bash

# Test ENV vars and config inside:
env | grep DB_
```

```
curl http://postgres-svc:5432
```

▯ KEY POINTS

- --previous flag shows logs from crashed container (critical!)
- Exit code 137 = OOMKill. Exit code 1 = app error. Code 127 = command not found.
- Exponential backoff: 10s → 20s → 40s → 80s → 160s → 5min max
- kubectl run with sleep command overrides CMD for debugging

▯ INTERVIEW TIP

The --previous flag saves lives: 'The most valuable command for CrashLoopBackOff is kubectl logs <pod> --previous. By the time you look, the current container is already crashed and the new one may not have logs yet. --previous gets logs from the just-crashed container — this almost always shows the exact error line that caused the crash.'

Q36 A pod is stuck in 'Pending' status. How do you troubleshoot it? Medium

✓ANSWER

A pod in Pending state means the Scheduler could not find a suitable node to place it. The pod is not running yet. kubectl describe pod is the critical first command.

Common reasons for Pending pods:

55. Insufficient resources (most common):
- No node has enough CPU or memory to satisfy the pod's requests
- Fix: add nodes, reduce requests, or remove other pods

56. Node selector / affinity not satisfied:
- Pod requires a label that no node has
- Fix: add label to node, or fix the selector

57. Taint without toleration:
- All nodes are tainted and pod has no matching toleration
- Fix: add toleration to pod, or remove taint from a node

58. PVC not bound (PersistentVolumeClaim stuck in Pending):
- No PV matches the PVC's storage class or size
- Fix: create matching PV or check StorageClass

59. Too many pods per node:
- Node has reached max pod limit (default 110 per node)
- Fix: add nodes

60. ImagePullBackOff as next step:
- Pod scheduled but image cannot be pulled

```
▯ kubectl / YAML
```

```
# First command for any Pending pod:
kubectl describe pod <pod-name>
# Look for Events section:
# Warning  FailedScheduling  0/3 nodes are available:
#   3 Insufficient memory

# Check node resources:
kubectl describe nodes | grep -A 5 'Allocated resources'

# Check all events sorted by time:
kubectl get events --sort-by=.lastTimestamp

# Check if PVC is bound:
kubectl get pvc

# Check node labels (for nodeSelector issues):
kubectl get nodes --show-labels

# Check node taints:
kubectl describe nodes | grep Taints
```

KEY POINTS

- kubectl describe pod → Events section shows EXACTLY why it's Pending
- FailedScheduling event = scheduler couldn't find suitable node
- kubectl describe node → shows allocatable vs allocated resources
- PVC Pending = no matching PV or StorageClass issue

INTERVIEW TIP

Structured approach impresses: 'My Pending pod checklist: 1) kubectl describe pod — read the FailedScheduling event message. 2) kubectl describe nodes — check allocatable vs requested resources. 3) kubectl get pvc — check if PVC is bound. 4) kubectl get events --sort-by=.lastTimestamp — cluster-wide event timeline. Usually the answer is in step 1.'

Q37 What is ImagePullBackOff? How do you fix it?

Easy

✓ANSWER

ImagePullBackOff (also shown as ErrImagePull) means Kubernetes cannot pull the container image specified in the pod spec. K8s tries, fails, waits (backoff), and retries.

Common causes and fixes:

61. Image does not exist or wrong tag:
- nginx:ltest instead of nginx:latest
- Fix: correct the image name/tag in the pod spec

62. Private registry — no credentials:
- Image is in a private registry (ECR, GCR, ACR) but no pull secret configured
- Fix: create imagePullSecret and reference in pod spec

63. Registry authentication expired:

- AWS ECR tokens expire after 12 hours
- Fix: refresh the imagePullSecret, or use IAM role for ECR access

64. Registry unreachable:
- Network policy blocking egress, or registry is down
- Fix: check network connectivity from node to registry

65. Image too large — timeout:
- Large image takes too long and times out
- Fix: use multi-stage builds to reduce image size

Diagnosis: kubectl describe pod → Events show exact error message from registry.

kubectl / YAML

```
# Check image pull error:
kubectl describe pod <pod-name>
# Events: Failed to pull image: ...unauthorized

# Create Docker registry pull secret:
kubectl create secret docker-registry ecr-secret \
  --docker-server=123.dkr.ecr.us-east-1.amazonaws.com \
  --docker-username=AWS \
  --docker-password=$(aws ecr get-login-password --region us-east-1)

# Reference in pod spec:
# spec:
#   imagePullSecrets:
#   - name: ecr-secret

# Test image pull manually on node:
# ssh to node
# docker pull <image>  # see exact error
```

KEY POINTS

- ImagePullBackOff = cannot pull image. Check name, tag, credentials.
- ECR credentials expire every 12 hours — use IAM roles instead
- kubectl describe pod Events section shows exact registry error
- imagePullSecrets in pod spec required for private registries

INTERVIEW TIP

ECR expiry is a real production incident. Say: 'I've seen production outages from ECR credential expiry. The fix is to use IAM roles for EKS worker nodes instead of static imagePullSecrets — the ECR authorization happens automatically via instance role. No credentials to expire. This is the correct production pattern for AWS.'

Q38 How do you troubleshoot a pod that is Running but the application is not responding?

Hard

✓ANSWER

Container shows Running but app returns errors or is unreachable. Systematic investigation at multiple layers:

Phase 1 — Verify the problem:

- kubectl exec -it <pod> -- curl localhost:8080/health → test from INSIDE the pod
- If works inside but not from outside → Service/network issue
- If fails inside → application issue

Phase 2 — Check resource usage:

- kubectl top pod <pod> → CPU/memory usage
- Memory at limit = OOM pressure. CPU throttled = slow responses.

Phase 3 — Read the logs:

- kubectl logs <pod> --tail=100 → recent logs
- kubectl logs <pod> -f → follow live
- Look for: exceptions, connection errors, out of memory, deadlock messages

Phase 4 — Check readiness probe:

- kubectl describe pod → Events showing readiness probe failures
- If readiness failing, pod removed from Service endpoints → no traffic routing

Phase 5 — Check Service endpoints:

- kubectl get endpoints <service-name> → are pod IPs listed?
- Empty endpoints = no ready pods matching selector

Phase 6 — Check network connectivity:

- kubectl exec -it <pod> -- env | grep DB_HOST → correct connection config?
- kubectl exec -it <pod> -- curl http://db-service:5432 → can reach dependencies?

kubectl / YAML

```
# Test from inside pod:
kubectl exec -it <pod> -- curl localhost:8080/health

# Check Service endpoints:
kubectl get endpoints myapp-service
# NO endpoints = no ready pods

# Check resource usage:
kubectl top pod <pod>

# Follow logs:
kubectl logs <pod> -f

# Check Service selector matches pod labels:
kubectl describe service myapp-service
kubectl get pods --show-labels

# Full network test:
kubectl exec -it <pod> -- wget -qO- http://db-service:5432
```

KEY POINTS

- Test from INSIDE pod first: exec + curl localhost
- kubectl get endpoints — empty = no ready pods in Service
- kubectl top pods — resource exhaustion causes silent failures
- kubectl describe pod Events — readiness probe failures remove pod from LB

INTERVIEW TIP

The endpoints check is the most commonly missed step. Say: 'The fastest diagnosis for Service routing issues is kubectl get endpoints <service-name>. If it shows no addresses, no pods are passing readiness checks. If addresses are there but traffic fails, it's a network or application problem. This single command narrows down 80% of Service connectivity issues.'

Q39 What are the most important kubectl commands every DevOps engineer must know?

Medium

✓ANSWER

Essential kubectl command categories:

CLUSTER OVERVIEW:

- kubectl get nodes -o wide — all nodes with IPs and roles
- kubectl cluster-info — API server and CoreDNS endpoints
- kubectl get all --all-namespaces — everything in all namespaces

DEBUGGING (most used daily):

- kubectl describe pod <name> — full pod details, events, probe status
- kubectl logs <pod> --previous — logs from crashed container
- kubectl exec -it <pod> -- bash — shell into running container
- kubectl top pods / kubectl top nodes — resource usage
- kubectl get events --sort-by=.lastTimestamp — time-ordered cluster events

DEPLOYMENTS:

- kubectl rollout status deployment/<name> — watch rollout progress
- kubectl rollout history deployment/<name> — see revision history
- kubectl rollout undo deployment/<name> — instant rollback
- kubectl scale deployment/<name> --replicas=5 — manual scaling

COPYING & EDITING:

- kubectl cp <pod>:/path/file ./local — copy file from pod
- kubectl edit deployment/<name> — live edit (not recommended in prod)

PORT FORWARDING (debugging without exposing services):

- kubectl port-forward pod/<name> 8080:80 — forward local port to pod
- kubectl port-forward svc/<name> 8080:80 — forward to service

kubectl / YAML

```
# Power aliases:
alias k=kubectl
alias kgp='kubectl get pods'
alias kgs='kubectl get svc'
alias kdp='kubectl describe pod'

# Port forward to debug without exposure:
kubectl port-forward pod/myapp-abc123 8080:3000
kubectl port-forward svc/myapp-service 8080:80

# Watch resources in real-time:
kubectl get pods -w

# Get pod YAML (clean output):
kubectl get pod <name> -o yaml

# Quick log tail for multiple pods:
kubectl logs -l app=myapp --tail=50

# Delete stuck pods:
kubectl delete pod <name> --force --grace-period=0

# Dry run (test without applying):
kubectl apply -f manifest.yaml --dry-run=client
```

KEY POINTS

- kubectl describe and logs = first two commands for ANY issue
- --previous flag for CrashLoopBackOff logs
- port-forward for secure local access to any pod/service
- kubectl get events --sort-by is underrated — shows cluster history

INTERVIEW TIP

Show power-user knowledge: 'I use kubectl with aliases in production — k=kubectl, kgp=kubectl get pods, kdp=kubectl describe pod. I also use kubectl stern (multi-pod log tailing), kubectl neat (clean YAML output), and k9s (terminal UI). These tools 10x my debugging speed compared to raw kubectl.'

Q40 What is GitOps and how does Kubernetes enable it? What tools implement GitOps?

Hard

✓ANSWER

GitOps is an operational framework where Git is the single source of truth for declarative infrastructure and application configuration. All changes to the cluster go through Git commits and pull requests — never through direct kubectl commands.

GitOps principles:

- Declarative — all desired cluster state defined in YAML files in Git
- Versioned — Git history = full audit trail of every change
- Automated — a GitOps operator continuously syncs cluster state with Git
- Reconciled — if cluster state drifts from Git, the operator auto-corrects

How Kubernetes enables GitOps:

- Kubernetes is declarative by design (YAML manifests describe desired state)

- K8s API is idempotent — applying the same manifest twice is safe
- Controllers are reconciliation loops — perfect model for GitOps operators

GitOps tools:

- Argo CD — most popular. Monitors Git repos and syncs to K8s. Visual UI.
- Flux CD — CNCF project. Lightweight. Kubernetes-native GitOps.
- Jenkins X — CI/CD + GitOps for K8s native apps

GitOps workflow:

66. Developer commits YAML change to Git (new image tag)
67. PR reviewed and merged
68. Argo CD detects Git change
69. Argo CD applies new manifests to K8s cluster automatically
70. If someone manually changes cluster, Argo CD reverts it (drift detection)

kubectl / YAML

```
# Argo CD — install:
kubectl create namespace argocd
kubectl apply -n argocd -f https://raw.githubusercontent.com/argoproj/argo-
cd/stable/manifests/install.yaml

# Create Argo CD Application:
# kind: Application
# spec:
#   source:
#     repoURL: https://github.com/myorg/myapp-config
#     path: manifests/production
#     targetRevision: HEAD
#   destination:
#     server: https://kubernetes.default.svc
#     namespace: production
#   syncPolicy:
#     automated:
#       prune: true      # remove resources deleted from Git
#       selfHeal: true   # fix manual drift

# Sync manually:
argocd app sync myapp
```

KEY POINTS

- GitOps = Git as source of truth. No direct kubectl in production.
- Argo CD = most popular GitOps operator for Kubernetes
- Drift detection: if cluster diverges from Git, operator auto-corrects
- GitOps enables full audit trail — every change linked to a Git commit

INTERVIEW TIP

Show strategic awareness: 'GitOps is how I manage production K8s at scale. No engineer runs kubectl apply in production — everything goes through a PR to the GitOps repo. Argo CD syncs automatically. This gives us: audit trail of every change, rollback by reverting a commit, and disaster recovery by re-running Argo CD against a fresh cluster. The cluster becomes cattle, not pets.'