KUBERNETES

Complete Interview Preparation Guide

Core Concepts • YAML Examples • Commands • Interview Q&A

Free resource by **juanpablo.tech**

Chapters 1–11 | Architecture · Objects · Networking · RBAC · Scheduling

Chapter 1 Kubernetes Architecture

Kubernetes (K8s) is an open-source container orchestration platform that automates deploying, scaling, and managing containerized applications. Understanding its architecture is the foundation of every interview.

1.1 The Control Plane

The control plane manages the cluster. It makes global decisions about scheduling, detecting and responding to cluster events.

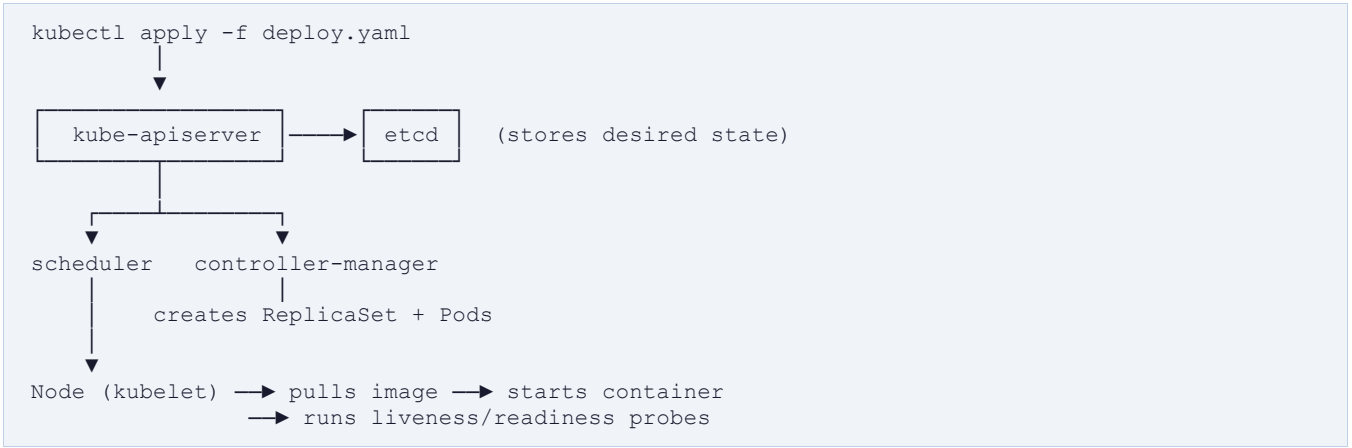
Component	Role	Key Detail
kube-apiserver	Front door to the cluster	All communication goes through it; validates and configures data
etcd	Distributed key-value store	Single source of truth; stores all cluster state
kube-scheduler	Assigns pods to nodes	Considers resources, affinity, taints, and tolerations
kube-controller-manager	Runs controller loops	Node, Replication, Endpoint, Service Account controllers
cloud-controller-manager	Integrates with cloud APIs	Manages load balancers, volumes, routes in cloud providers

1.2 Worker Node Components

Component	Role	Key Detail
kubelet	Node agent	Ensures containers in pods are running and healthy
kube-proxy	Network proxy	Maintains network rules, enables Service communication
Container Runtime	Runs containers	containerd, CRI-O, or Docker (deprecated)

1.3 Architecture Flow

When you run kubectl apply, the request travels through these components in sequence:



🎯 **Interview Tip** Common question: 'What happens when you run kubectl apply?' Walk through: API Server → etcd → controllers → scheduler → kubelet → container runtime.

Chapter 2 Core Kubernetes Objects

2.1 Pod

The smallest deployable unit. A Pod wraps one or more containers that share network and storage. Pods are ephemeral — never manage them directly in production.

```
# pod.yaml — minimal pod example
apiVersion: v1
kind: Pod
metadata:
  name: my-app
  labels:
    app: my-app
    env: production
spec:
  containers:
  - name: my-app
    image: nginx:1.25
    ports:
    - containerPort: 80
    resources:
      requests:
        cpu: '100m'        # 0.1 CPU core
        memory: '128Mi'
      limits:
        cpu: '500m'        # 0.5 CPU core
        memory: '256Mi'
    env:
    - name: ENV_VAR
      value: 'hello'
    - name: SECRET_VAR
      valueFrom:
        secretKeyRef:
          name: my-secret
          key: password
```

2.2 ReplicaSet

Ensures a specified number of pod replicas are running at all times. If a pod dies, the ReplicaSet creates a new one. In practice, you always use a Deployment which manages ReplicaSets for you.

```
# replicaset.yaml
apiVersion: apps/v1
kind: ReplicaSet
metadata:
  name: my-app-rs
spec:
  replicas: 3
  selector:
    matchLabels:
      app: my-app          # must match template labels
  template:
    metadata:
      labels:
        app: my-app
    spec:
      containers:
      - name: my-app
        image: nginx:1.25
```

2.3 Deployment

The most common workload resource. Manages ReplicaSets and provides declarative updates, rolling upgrades, and rollback capability.

```yaml
# deployment.yaml — production-ready example
apiVersion: apps/v1
kind: Deployment
metadata:
  name: my-app
  namespace: production
  labels:
    app: my-app
spec:
  replicas: 3
  selector:
    matchLabels:
      app: my-app
  strategy:
    type: RollingUpdate
    rollingUpdate:
      maxSurge: 1          # max extra pods during update
      maxUnavailable: 0    # zero downtime
  template:
    metadata:
      labels:
        app: my-app
    spec:
      containers:
      - name: my-app
        image: my-app:1.0
        ports:
        - containerPort: 8080
        resources:
          requests:
            cpu: '100m'
            memory: '128Mi'
          limits:
            cpu: '500m'
            memory: '512Mi'
        startupProbe:
          httpGet:
            path: /healthz
            port: 8080
          failureThreshold: 30
          periodSeconds: 10
        livenessProbe:
          httpGet:
            path: /healthz
            port: 8080
          periodSeconds: 5
          failureThreshold: 3
        readinessProbe:
          httpGet:
            path: /ready
            port: 8080
          periodSeconds: 3
          failureThreshold: 2
        lifecycle:
          preStop:
            exec:
              command: ["/bin/sh", "-c", "sleep 5"]
```

2.4 StatefulSet

Used for stateful applications that require stable network identities and persistent storage (databases, message queues). Pods are named sequentially: app-0, app-1, app-2.

```
# statefulset.yaml
apiVersion: apps/v1
kind: StatefulSet
metadata:
  name: postgres
spec:
  serviceName: postgres     # headless service name
  replicas: 3
  selector:
    matchLabels:
      app: postgres
  template:
    metadata:
      labels:
        app: postgres
    spec:
      containers:
      - name: postgres
        image: postgres:15
        ports:
        - containerPort: 5432
        volumeMounts:
        - name: data
          mountPath: /var/lib/postgresql/data
        env:
        - name: POSTGRES_PASSWORD
          valueFrom:
            secretKeyRef:
              name: postgres-secret
              key: password
  volumeClaimTemplates:      # each pod gets its own PVC
  - metadata:
      name: data
    spec:
      accessModes: ['ReadWriteOnce']
      storageClassName: fast-ssd
      resources:
        requests:
          storage: 20Gi
```

2.5 DaemonSet

Ensures one copy of a pod runs on every (or selected) node. Used for node-level agents like log collectors, monitoring agents, and network plugins.

```
# daemonset.yaml
apiVersion: apps/v1
kind: DaemonSet
metadata:
  name: fluentd
  namespace: kube-system
spec:
  selector:
    matchLabels:
      app: fluentd
  template:
    metadata:
      labels:
        app: fluentd
    spec:
      tolerations:
      - key: node-role.kubernetes.io/control-plane
        effect: NoSchedule    # run on control-plane nodes too
      containers:
      - name: fluentd
        image: fluent/fluentd:v1.16
        volumeMounts:
        - name: varlog
          mountPath: /var/log
      volumes:
      - name: varlog
        hostPath:
          path: /var/log
```

2.6 Job & CronJob

Jobs run pods to completion (e.g., database migrations). CronJobs schedule Jobs on a cron schedule.

```
# job.yaml — run once to completion
apiVersion: batch/v1
kind: Job
metadata:
  name: db-migration
spec:
  completions: 1
  parallelism: 1
  backoffLimit: 3          # retry 3 times on failure
  template:
    spec:
      restartPolicy: Never  # Never or OnFailure for Jobs
      containers:
      - name: migration
        image: my-app:1.0
        command: ['python', 'manage.py', 'migrate']
```

```
# cronjob.yaml — scheduled jobs
apiVersion: batch/v1
kind: CronJob
metadata:
  name: backup
spec:
  schedule: '0 2 * * *'    # every day at 2 AM
  jobTemplate:
    spec:
      template:
        spec:
          restartPolicy: OnFailure
          containers:
          - name: backup
            image: backup-tool:latest
            command: ['/scripts/backup.sh']
```

Chapter 3 Services & Networking

A Service provides a stable network endpoint for a set of pods. Since pods are ephemeral and their IPs change, Services provide a consistent IP and DNS name.

3.1 Service Types

Type	Access	Use Case
ClusterIP (default)	Internal cluster only	Internal microservice communication
NodePort	External via node IP + port	Dev/test, basic external access
LoadBalancer	External via cloud LB	Production external traffic (cloud only)
ExternalName	Maps to external DNS	Routing to external services
Headless (ClusterIP: None)	Direct pod DNS	StatefulSets, direct pod addressing

3.2 ClusterIP Service

```
# service-clusterip.yaml
apiVersion: v1
kind: Service
metadata:
  name: my-app-svc
  namespace: production
spec:
  type: ClusterIP           # default, internal only
  selector:
    app: my-app             # matches pod labels
  ports:
  - name: http
    port: 80                # service port
    targetPort: 8080        # container port
    protocol: TCP
```

3.3 LoadBalancer Service

```
# service-lb.yaml — production external traffic
apiVersion: v1
kind: Service
metadata:
  name: my-app-lb
  annotations:
    service.beta.kubernetes.io/aws-load-balancer-type: nlb
spec:
  type: LoadBalancer
  selector:
    app: my-app
  ports:
  - port: 443
    targetPort: 8080
  loadBalancerSourceRanges:
  - '10.0.0.0/8'           # restrict to internal IPs
```

3.4 Headless Service (for StatefulSets)

```
# service-headless.yaml
apiVersion: v1
kind: Service
metadata:
  name: postgres
spec:
  clusterIP: None           # headless: no virtual IP
  selector:
    app: postgres
  ports:
  - port: 5432
    targetPort: 5432
# DNS: postgres-0.postgres.default.svc.cluster.local
#      postgres-1.postgres.default.svc.cluster.local
```

3.5 Ingress

Ingress manages external HTTP/HTTPS access to services, providing routing rules, SSL termination, and virtual hosting.

```
# ingress.yaml
apiVersion: networking.k8s.io/v1
kind: Ingress
metadata:
  name: my-app-ingress
  annotations:
    nginx.ingress.kubernetes.io/rewrite-target: /
    cert-manager.io/cluster-issuer: letsencrypt-prod
spec:
  ingressClassName: nginx
  tls:
  - hosts:
    - myapp.example.com
    secretName: myapp-tls
  rules:
  - host: myapp.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
```

3.6 NetworkPolicy

NetworkPolicies restrict traffic between pods. A default-deny policy combined with explicit allow rules gives you fine-grained network segmentation.

```
# networkpolicy.yaml — deny all, allow specific
apiVersion: networking.k8s.io/v1
kind: NetworkPolicy
metadata:
  name: api-netpol
spec:
  podSelector:
    matchLabels:
      app: api
  policyTypes:
  - Ingress
  - Egress
  ingress:
  - from:
    - podSelector:
        matchLabels:
          app: frontend    # only frontend can reach api
    ports:
    - port: 8080
  egress:
  - to:
    - podSelector:
        matchLabels:
          app: postgres    # api can only reach postgres
    ports:
    - port: 5432
```

Chapter 4 Configuration Management

4.1 ConfigMap

Stores non-sensitive configuration data as key-value pairs. Can be consumed as environment variables or mounted as files.

```
# configmap.yaml
apiVersion: v1
kind: ConfigMap
metadata:
  name: app-config
data:
  APP_ENV: production
  LOG_LEVEL: info
  config.yaml: |            # multi-line file content
    server:
      port: 8080
      timeout: 30s
```

```
# Consume in a pod
spec:
  containers:
  - name: app
    envFrom:
    - configMapRef:
        name: app-config    # inject ALL keys as env vars
    volumeMounts:
    - name: config-vol
      mountPath: /etc/config
  volumes:
  - name: config-vol
    configMap:
      name: app-config      # mount as files
```

4.2 Secret

Stores sensitive data (passwords, tokens, certificates). Base64-encoded at rest — use external secret managers (Vault, AWS SSM) for production security.

```
# secret.yaml
apiVersion: v1
kind: Secret
metadata:
  name: app-secret
type: Opaque
stringData:                 # auto-encodes to base64
  DB_PASSWORD: supersecret
  API_KEY: myapikey123
```

```
# Create from CLI (no YAML needed)
kubectl create secret generic app-secret \
  --from-literal=DB_PASSWORD=supersecret \
  --from-file=tls.crt=./cert.pem

# TLS secret
kubectl create secret tls my-tls \
  --cert=tls.crt --key=tls.key
```

4.3 Resource Quotas & Limits

ResourceQuotas cap total resource usage per namespace. LimitRanges set default requests and limits per pod/container.

```
# resourcequota.yaml — namespace-level limits
apiVersion: v1
kind: ResourceQuota
metadata:
  name: team-quota
  namespace: team-a
spec:
  hard:
    requests.cpu: '4'
    requests.memory: 8Gi
    limits.cpu: '8'
    limits.memory: 16Gi
    pods: '20'
    services: '10'
```

```
# limitrange.yaml — default per-pod limits
apiVersion: v1
kind: LimitRange
metadata:
  name: default-limits
spec:
  limits:
  - type: Container
    default:
      cpu: '200m'
      memory: '256Mi'
    defaultRequest:
      cpu: '100m'
      memory: '128Mi'
```

Chapter 5 Health Probes & Lifecycle

5.1 Probe Types Compared

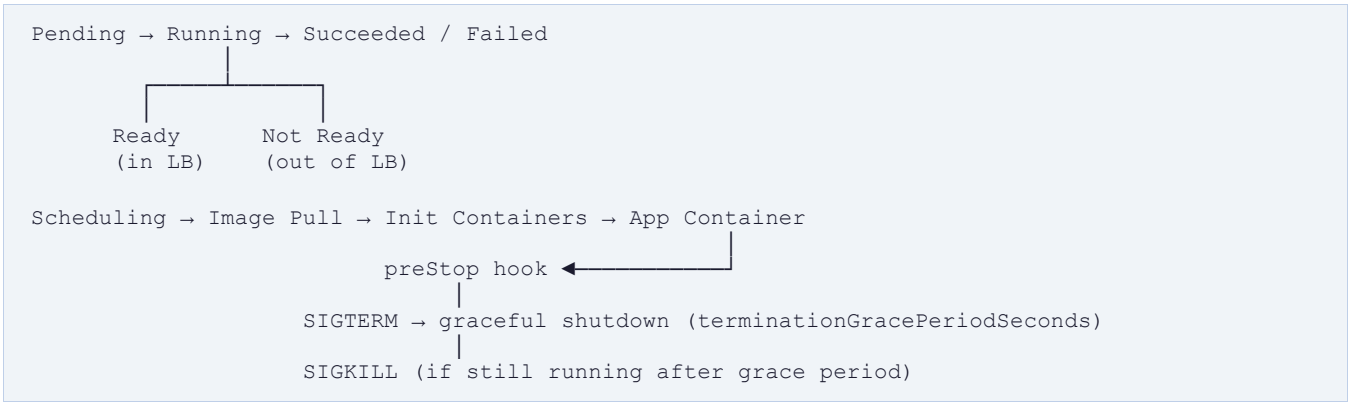
Probe	Failure Action	Use Case
startupProbe	Container restarted	Slow-starting apps; disables other probes until success
livenessProbe	Container restarted	Detect deadlocks, frozen processes
readinessProbe	Removed from Service LB	Temporary unavailability, warmup, DB not ready

5.2 Full Probe Configuration

```
containers:
- name: app
  image: my-app:1.0

  # Startup probe: up to 300s for slow start (30 * 10s)
  startupProbe:
    httpGet:
      path: /healthz
      port: 8080
      httpHeaders:
      - name: X-Health-Check
        value: 'true'
    failureThreshold: 30
    periodSeconds: 10

  # Liveness: restart if app is frozen
  livenessProbe:
    httpGet:
      path: /healthz
      port: 8080
    initialDelaySeconds: 0   # startupProbe handles delay
    periodSeconds: 5
    timeoutSeconds: 3
    failureThreshold: 3
    successThreshold: 1

  # Readiness: stop traffic if not ready
  readinessProbe:
    httpGet:
      path: /ready
      port: 8080
    periodSeconds: 3
    failureThreshold: 2
    successThreshold: 1

  # TCP probe example (for non-HTTP services)
  # livenessProbe:
  #   tcpSocket:
  #     port: 5432

  # Exec probe example
  # livenessProbe:
  #   exec:
  #     command: ['cat', '/tmp/healthy']
```

5.3 Pod Lifecycle

A pod moves through several phases from creation to termination. Understanding the lifecycle is critical for debugging and designing graceful shutdown.



Chapter 6 Storage & Volumes

6.1 Volume Types

Type	Scope	Use Case
emptyDir	Pod lifetime	Scratch space, inter-container sharing
hostPath	Node filesystem	DaemonSet log access, dev only
configMap / secret	Namespace	Mount config files into containers
persistentVolumeClaim	Cluster	Durable storage for stateful apps
projected	External	Combine multiple sources into one mount point

6.2 PersistentVolume & PVC

The storage lifecycle: a StorageClass provisions a PV → a PVC binds to the PV → a Pod mounts the PVC. Dynamic provisioning (via StorageClass) is the production standard.

```
# persistentvolume.yaml — cluster-wide resource
apiVersion: v1
kind: PersistentVolume
metadata:
  name: pv-100gi
spec:
  capacity:
    storage: 100Gi
  accessModes:
  - ReadWriteOnce          # RWO: one node at a time
  # - ReadWriteMany        # RWX: many nodes (NFS etc.)
  # - ReadOnlyMany         # ROX: read-only many nodes
  persistentVolumeReclaimPolicy: Retain  # or Delete, Recycle
  storageClassName: fast-ssd
  awsElasticBlockStore:
    volumeID: vol-12345678
    fsType: ext4
```

```
# persistentvolumeclaim.yaml — namespace resource
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
  name: my-pvc
spec:
  accessModes:
  - ReadWriteOnce
  storageClassName: fast-ssd
  resources:
    requests:
      storage: 20Gi
```

```
# Use in pod
volumes:
- name: data
  persistentVolumeClaim:
    claimName: my-pvc
volumeMounts:
- name: data
  mountPath: /data
```

Chapter 7 RBAC & Security

7.1 RBAC Concepts

Object	Scope	Purpose
Role	Namespace	Defines allowed verbs on resources within a namespace
ClusterRole	Cluster-wide	Defines allowed verbs on cluster-level resources
RoleBinding	Namespace	Binds a Role to a user/group/serviceaccount
ClusterRoleBinding	Cluster-wide	Binds a ClusterRole cluster-wide
ServiceAccount	Namespace	Identity for pods (not humans)

7.2 RBAC YAML Examples

```
# role.yaml — namespace-scoped permissions
apiVersion: rbac.authorization.k8s.io/v1
kind: Role
metadata:
  name: pod-reader
  namespace: production
rules:
- apiGroups: ['']           # '' = core API group
  resources: ['pods', 'pods/log']
  verbs: ['get', 'list', 'watch']
- apiGroups: ['apps']
  resources: ['deployments']
  verbs: ['get', 'list', 'watch', 'update', 'patch']
```

```
# rolebinding.yaml
apiVersion: rbac.authorization.k8s.io/v1
kind: RoleBinding
metadata:
  name: read-pods
  namespace: production
subjects:
- kind: User
  name: jane
  apiGroup: rbac.authorization.k8s.io
- kind: ServiceAccount
  name: my-sa
  namespace: production
roleRef:
  kind: Role
  name: pod-reader
  apiGroup: rbac.authorization.k8s.io
```

7.3 ServiceAccount for Pods

```
# serviceaccount.yaml
apiVersion: v1
kind: ServiceAccount
metadata:
  name: my-app-sa
  namespace: production
automountServiceAccountToken: false  # security best practice

# Assign to pod
spec:
  serviceAccountName: my-app-sa
  automountServiceAccountToken: true  # only if needed
```

7.4 Security Context

```
spec:
  securityContext:
    runAsNonRoot: true       # pod-level
    runAsUser: 1000
    fsGroup: 2000
  containers:
  - name: app
    securityContext:
      allowPrivilegeEscalation: false  # container-level
      readOnlyRootFilesystem: true
      capabilities:
        drop:
        - ALL                # drop all linux capabilities
        add:
        - NET_BIND_SERVICE   # only add what you need
```

Chapter 8 Scheduling & Affinity

8.1 Taints & Tolerations

Taints repel pods from nodes. Tolerations allow pods to be scheduled on tainted nodes.

```
# Add taint to node
kubectl taint nodes node1 gpu=true:NoSchedule
kubectl taint nodes node1 maintenance=true:NoExecute  # evicts existing pods

# Remove taint
kubectl taint nodes node1 gpu=true:NoSchedule-

# Toleration in pod spec
spec:
  tolerations:
  - key: 'gpu'
    operator: 'Equal'
    value: 'true'
    effect: 'NoSchedule'
  - key: 'node.kubernetes.io/not-ready'
    operator: 'Exists'
    effect: 'NoExecute'
    tolerationSeconds: 300  # tolerate for 300s before eviction
```

8.2 Node Affinity

```
spec:
  affinity:
    nodeAffinity:
      requiredDuringSchedulingIgnoredDuringExecution:  # hard rule
        nodeSelectorTerms:
        - matchExpressions:
          - key: kubernetes.io/arch
            operator: In
            values: ['amd64']
      preferredDuringSchedulingIgnoredDuringExecution:  # soft rule
      - weight: 100
        preference:
          matchExpressions:
          - key: node-type
            operator: In
            values: ['high-mem']

    podAntiAffinity:        # spread pods across nodes
      requiredDuringSchedulingIgnoredDuringExecution:
      - labelSelector:
          matchLabels:
            app: my-app
        topologyKey: kubernetes.io/hostname
```

8.3 HorizontalPodAutoscaler

```
# hpa.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  # scale when CPU > 70%
  - type: Resource
    resource:
      name: memory
      target:
        type: Utilization
        averageUtilization: 80
```

Chapter 9 Essential kubectl Commands

9.1 Context & Cluster

```
# Switch context (cluster)
kubectl config get-contexts
kubectl config use-context prod-cluster
kubectl config current-context

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

# Cluster info
kubectl cluster-info
kubectl get nodes -o wide
kubectl describe node <node-name>
kubectl top nodes
```

9.2 Get & Describe

```
# Get resources
kubectl get pods -n <namespace>
kubectl get pods -A                          # all namespaces
kubectl get pods -o wide                     # with IPs and nodes
kubectl get pods -o yaml                     # full YAML output
kubectl get pods -l app=my-app               # label selector
kubectl get pods --field-selector status.phase=Running

# Watch live
kubectl get pods -w

# Describe (most detailed info)
kubectl describe pod <pod-name>
kubectl describe deployment <deploy-name>
kubectl describe node <node-name>

# Get all resource types
kubectl get all -n <namespace>
kubectl api-resources                        # list all resource types
```

9.3 Logs

```
# Basic logs
kubectl logs <pod-name>
kubectl logs <pod-name> -n <namespace>

# Previous (crashed) container logs — most useful for crash loops
kubectl logs <pod-name> --previous

# Stream live logs
kubectl logs <pod-name> -f

# Last N lines
kubectl logs <pod-name> --tail=100

# Since time
kubectl logs <pod-name> --since=1h
kubectl logs <pod-name> --since-time='2024-01-15T10:00:00Z'

# Specific container in multi-container pod
kubectl logs <pod-name> -c <container-name>

# All pods with a label
kubectl logs -l app=my-app --all-containers=true
```

9.4 Deployments

```
# Apply changes
kubectl apply -f deployment.yaml
kubectl apply -f ./k8s/                       # apply all YAMLs in dir

# Rollout management
kubectl rollout status deployment/my-app
kubectl rollout history deployment/my-app
kubectl rollout undo deployment/my-app
kubectl rollout undo deployment/my-app --to-revision=2
kubectl rollout restart deployment/my-app     # rolling restart
kubectl rollout pause deployment/my-app
kubectl rollout resume deployment/my-app

# Scale
kubectl scale deployment/my-app --replicas=5
kubectl autoscale deployment/my-app --min=2 --max=10 --cpu-percent=70

# Update image
kubectl set image deployment/my-app app=my-app:2.0

# Delete
kubectl delete deployment my-app
kubectl delete -f deployment.yaml
```

9.5 Debugging

```
# Execute into container
kubectl exec -it <pod-name> -- /bin/bash
kubectl exec -it <pod-name> -c <container> -- /bin/sh

# Run temporary debug pod
kubectl run debug --image=busybox -it --rm --restart=Never -- sh

# Debug with ephemeral container (K8s 1.23+)
kubectl debug -it <pod-name> --image=ubuntu --target=<container>

# Port forward
kubectl port-forward pod/<pod-name> 8080:8080
kubectl port-forward service/<svc-name> 8080:80
kubectl port-forward deployment/<deploy-name> 8080:8080

# Copy files
kubectl cp <pod-name>:/app/log.txt ./log.txt
kubectl cp ./file.txt <pod-name>:/tmp/file.txt

# Events — most useful for debugging
kubectl get events -n <namespace> --sort-by='.lastTimestamp'
kubectl get events --field-selector type=Warning

# Resource usage
kubectl top pods -n <namespace>
kubectl top pods --containers

# Check RBAC permissions
kubectl auth can-i create pods --as=jane
kubectl auth can-i '*' '*'                  # are you admin?
```

9.6 Namespace Operations

```
# Create namespace
kubectl create namespace staging

# Work in namespace
kubectl get pods -n staging
kubectl apply -f app.yaml -n staging

# Set default namespace for session
kubectl config set-context --current --namespace=staging

# Delete namespace (deletes everything in it!)
kubectl delete namespace staging
```

Chapter 10 Top Interview Questions & Answers

Architecture & Core Concepts

Q: What is the difference between a Pod, Deployment, and ReplicaSet?

Pod: The smallest unit; one or more containers sharing network and storage. Ephemeral.
ReplicaSet: Ensures N pods are always running. If one dies, it creates a new one.
Deployment: Manages ReplicaSets. Adds rolling updates, rollback, and declarative update strategy.

Always use Deployments — never manage ReplicaSets directly.

Q: What happens when a node goes down?

1. Node Controller detects node is unreachable (after ~40 seconds).
2. After 5 minutes (pod-eviction-timeout), pods are marked for eviction.
3. Scheduler reschedules pods to healthy nodes.
4. If PodDisruptionBudgets are set, they are respected during eviction.

Note: Pods in a StatefulSet retain their identity when rescheduled.

Q: What is the difference between liveness and readiness probes?

Liveness: 'Is the app alive?' Failure = container restarted. Use for deadlocks/frozen apps.
Readiness: 'Is the app ready for traffic?' Failure = removed from Service load balancer, NOT restarted.
Startup: 'Has the app finished starting?' Disables other probes until it passes. Use for slow-start apps.

Q: How does a Service route traffic to Pods?

A Service uses a label selector (e.g., app: my-app) to find matching pods. kube-proxy on each node maintains iptables/IPVS rules that forward traffic from the Service's ClusterIP to one of the pod IPs.

The Endpoints object lists the pod IPs matched by the selector and is kept up-to-date by the Endpoints controller.

Q: What is the difference between ConfigMap and Secret?

Both store key-value data for pods. Secrets are base64-encoded and have stricter access controls (RBAC). ConfigMaps are for non-sensitive config; Secrets for passwords, tokens, and certificates.

Best practice: Never store sensitive data in ConfigMaps. Use Secrets + external secret managers (HashiCorp Vault, AWS Secrets Manager) for production.

Networking

Q: What are the 4 Kubernetes networking rules?

1. Pods can communicate with all other pods without NAT.
2. Nodes can communicate with all pods without NAT.
3. The IP a pod sees itself as is the same IP others see.
4. Services provide stable IPs for sets of ephemeral pods.

Implemented by CNI plugins: Calico, Flannel, Cilium, Weave.

Q: What is the difference between NodePort, ClusterIP, and LoadBalancer?

ClusterIP: Internal only. Default type. Stable IP inside the cluster.
NodePort: Exposes service on every node's IP at a static port (30000-32767). External access via <nodeIP>:<nodePort>.
LoadBalancer: Creates a cloud load balancer. Traffic flows: External LB → NodePort → ClusterIP → Pod.

Requires cloud provider support.

Storage & State

Q: What is the difference between StatefulSet and Deployment?

Deployment: Pods are interchangeable. Any pod can be replaced by any other. No stable identity.
StatefulSet: Pods have stable, ordered names (app-0, app-1). Ordered startup and shutdown. Each pod can have its own PVC via volumeClaimTemplates. Stable DNS: pod-name.service-name.

Use StatefulSets for: databases, Kafka, ZooKeeper, Elasticsearch.

Q: What is a PersistentVolume, PVC, and StorageClass?

PersistentVolume (PV): Cluster-level storage resource, provisioned by admin or dynamically.
PersistentVolumeClaim (PVC): Namespace-level request for storage by a pod. Binds to a PV.
StorageClass: Defines how storage is dynamically provisioned (AWS EBS, GCP PD, Azure Disk).

Flow: PVC created → StorageClass provisions PV → PV binds to PVC → Pod mounts PVC.

Operations & Troubleshooting

Q: A pod is in CrashLoopBackOff. How do you debug it?

1. kubectl describe pod <name> → check Events and Last State/Exit Code.
2. kubectl logs <name> --previous → logs from the crashed container.
3. Exit code 137 = OOMKilled (increase memory limit).
4. Exit code 1 = App error (check app logs).
5. Check liveness probe — may be too aggressive (lower failureThreshold, add startupProbe).
6. Check missing ConfigMaps/Secrets referenced in env or volumes.

Q: How do rolling updates work and how do you roll back?

Rolling update: Deployment creates a new ReplicaSet, scales it up gradually while scaling down the old one. Controlled by maxSurge (extra pods allowed) and maxUnavailable (pods allowed down).

Rollback: kubectl rollout undo deployment/my-app rolls back to the previous ReplicaSet. Use --to-revision=N for a specific version.

Best practice: Always set maxUnavailable: 0 for zero-downtime deployments.

Q: What is a DaemonSet and when would you use it?

A DaemonSet ensures one pod runs on every node (or selected nodes via nodeSelector/tolerations). New nodes automatically get the pod; removed nodes clean it up.

Common uses: Log collectors (Fluentd, Filebeat), monitoring agents (Prometheus node-exporter, Datadog), network plugins (CNI), security agents.

Q: How does Kubernetes handle resource limits and OOM?

Requests: Minimum guaranteed resources; used for scheduling decisions.
Limits: Maximum allowed. CPU is throttled when exceeded. Memory causes OOMKilled (exit 137) when exceeded.

QoS Classes:
Guaranteed (requests=limits) → highest priority
Burstable (requests<limits) → medium priority
BestEffort (no requests/limits) → evicted first under pressure

Best practice: Always set both requests and limits. Use LimitRange for namespace defaults.

Chapter 11 Quick Reference Cheat Sheet

11.1 Resource Shortnames

Resource	Short	Resource	Short
pods	po	services	svc
deployments	deploy	replicasets	rs
namespaces	ns	configmaps	cm
persistentvolumeclaims	pvc	persistentvolumes	pv
serviceaccounts	sa	horizontalpodautoscalers	hpa
ingresses	ing	statefulsets	sts

11.2 Essential One-Liners

```
# Create resources quickly (imperative)
kubectl run nginx --image=nginx --port=80
kubectl create deployment web --image=nginx --replicas=3
kubectl expose deployment web --port=80 --type=LoadBalancer
kubectl create configmap app-cfg --from-literal=KEY=VALUE
kubectl create secret generic db-secret --from-literal=PASS=secret
kubectl create namespace dev
kubectl create serviceaccount my-sa

# Generate YAML without applying (--dry-run)
kubectl create deployment web --image=nginx --dry-run=client -o yaml > deploy.yaml
kubectl expose deployment web --port=80 --dry-run=client -o yaml > svc.yaml

# Quick edit
kubectl edit deployment my-app
kubectl patch deployment my-app -p '{"spec":{"replicas":5}}'

# Labels and annotations
kubectl label pod my-pod env=production
kubectl annotate deployment my-app team=backend
kubectl get pods -l env=production,app=my-app

# Force delete stuck pod
kubectl delete pod <name> --force --grace-period=0

# Get resource as JSON and query with jq
kubectl get pod <name> -o json | jq '.status.conditions'

# Check what changed in a rollout
kubectl rollout history deployment/my-app --revision=3
```

11.3 Probe & Resource Quick Reference

Setting	Value	Notes
initialDelaySeconds	0–30	Use startupProbe instead for slow starts
periodSeconds	5–10	How often to check
failureThreshold	3	Failures before action taken
timeoutSeconds	1–3	Probe timeout
CPU request	100m–500m	100m = 0.1 core
Memory request	128Mi–512Mi	Always set to avoid OOM eviction
terminationGrace	30s	Increase for long-lived connections

11.4 Exit Code Reference

Exit Code	Meaning	Fix
0	Clean exit	Check why app stopped — intended?
1	App error	Check app logs for exception/error
137	OOMKilled (128+9)	Increase memory limit
139	Segfault	App or native dependency bug
143	SIGTERM received	Normal graceful shutdown
255	Unknown / overflow	Check app logs carefully

11.5 YAML Structure Reference

```
# Every K8s YAML has these 4 top-level fields:
apiVersion: apps/v1         # API group/version
kind: Deployment            # resource type
metadata:                   # name, namespace, labels
  name: my-app
  namespace: production
  labels:
    app: my-app
spec:                       # desired state (varies by kind)
  ...

# Common apiVersions:
# v1              → Pod, Service, ConfigMap, Secret, PV, PVC, Namespace
# apps/v1         → Deployment, ReplicaSet, StatefulSet, DaemonSet
# batch/v1        → Job, CronJob
# networking.k8s.io/v1 → Ingress, NetworkPolicy
# rbac.authorization.k8s.io/v1 → Role, ClusterRole, RoleBinding
# autoscaling/v2  → HorizontalPodAutoscaler
```

Good luck with your Kubernetes interview! ☸

More free resources at **juanpablo.tech**