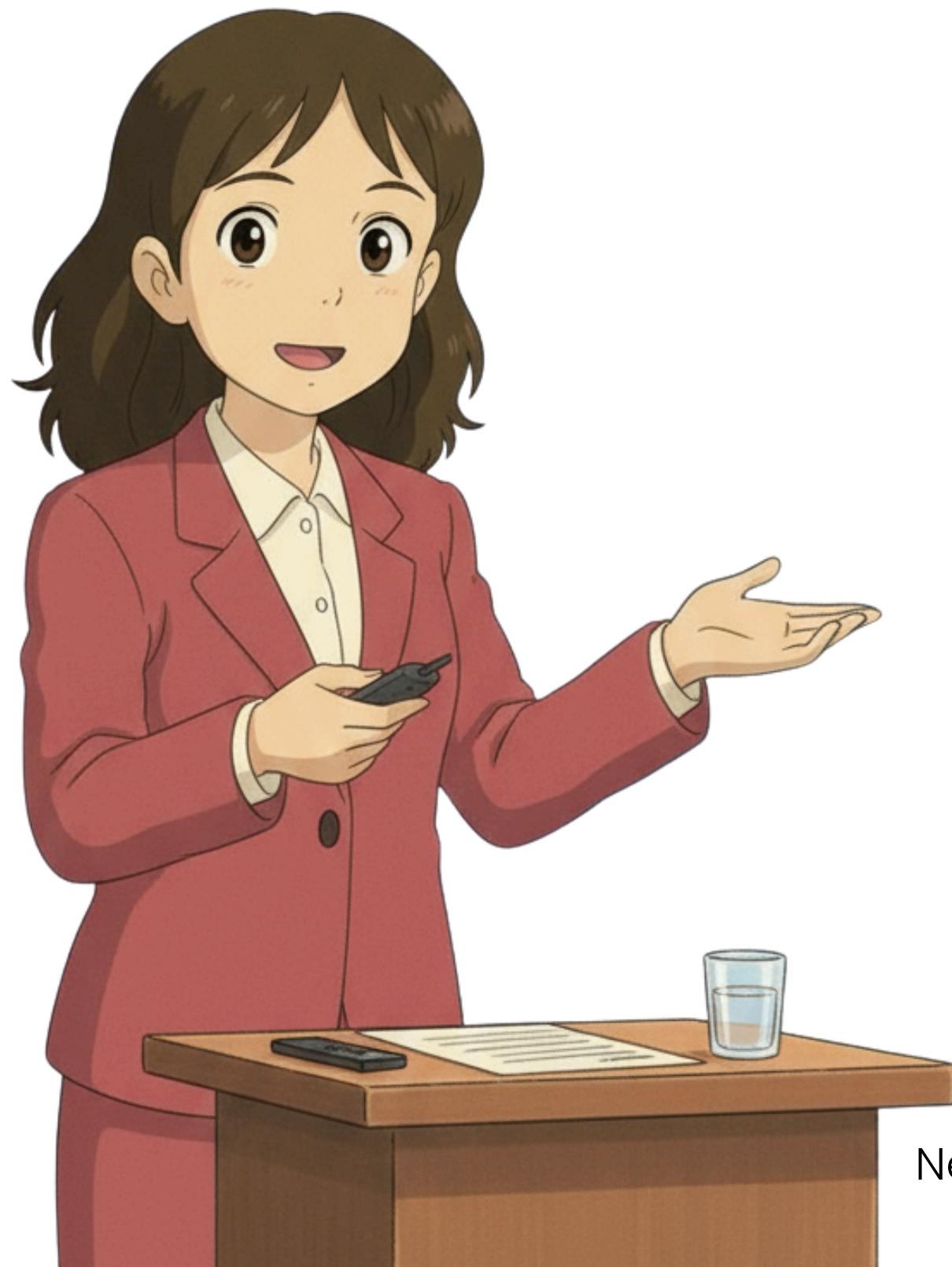


Top 30

Kubernetes Interview

Questions & Answers



Nensi Ravaliya

Top 30 Kubernetes Interview Questions & Answers

Asked at Top MNCs (Google, Amazon, Microsoft, Meta, Apple, Netflix)

SECTION 1: FUNDAMENTALS & CORE CONCEPTS

Q1: What is Kubernetes and why is it so popular?

Answer:

Kubernetes (K8s) is an open-source container orchestration platform originally developed by Google and now maintained by the Cloud Native Computing Foundation (CNCF). It automates the deployment, scaling, and management of containerized applications.

Key Features:

- **Container Orchestration:** Manages containerized applications across clusters of machines
- **Self-Healing:** Automatically restarts failed containers, replaces unresponsive nodes, and reschedules workloads
- **Auto-Scaling:** Scales applications horizontally (more pods) or vertically (more resources) based on demand
- **Load Balancing:** Distributes traffic efficiently between pods
- **Rolling Updates & Rollbacks:** Allows zero-downtime deployments with automatic rollback capability
- **Declarative Configuration:** Infrastructure as Code using YAML manifests

Popularity Reasons:

- Over 70% of Fortune 500 companies use Kubernetes in production
- Cloud-agnostic (works with AWS, Azure, GCP, on-premises)

- Simplifies microservices architecture management
 - Strong ecosystem and community support
 - Industry standard for container orchestration (replaced Docker Swarm)
-

Q2: Explain the Kubernetes Architecture

Answer:

Kubernetes follows a **master-worker** (control plane-data plane) architecture:

Control Plane Components (Master Node):

1. API Server (**kube-apiserver**)

- Front-end of Kubernetes cluster
- Handles all REST API requests
- Validates and persists data to etcd
- Acts as the gateway for cluster communication

2. etcd

- Distributed key-value store
- Stores entire cluster state and configuration
- Single source of truth for cluster data
- Requires backup strategy for disaster recovery

3. Scheduler (**kube-scheduler**)

- Assigns newly created pods to nodes
- Considers resource availability, constraints, and policies
- Makes scheduling decisions based on pod requirements

4. Controller Manager (**kube-controller-manager**)

- Runs controller processes
- Manages ReplicaSets, Deployments, StatefulSets, DaemonSets
- Handles node failures and maintains desired state

Worker Node Components:

1. Kubelet

- Agent running on each worker node
- Ensures containers run as specified
- Reports node and pod status to API server
- Executes pod lifecycle management

2. Kube-proxy

- Network proxy on each node
- Manages network rules for pod communication
- Implements service abstraction and load balancing
- Maintains network connectivity

3. Container Runtime

- Executes containers (Docker, containerd, CRI-O)
- Pulls images and runs containers
- Manages container lifecycle

How They Work Together:

User submits deployment → API Server validates → Stored in etcd → Scheduler assigns pods to nodes → Kubelet creates containers → Kube-proxy manages networking

Q3: What is a Pod and how does it differ from Docker containers?

Answer:

A **Pod** is the smallest deployable unit in Kubernetes, not containers themselves.

Key Pod Characteristics:

- Can contain one or more containers (usually one)
- Containers in a pod share networking namespace (same IP address)
- Containers share storage volumes

- Containers can communicate via localhost
- All containers in a pod are created/destroyed together
- Ephemeral by nature (can be created/destroyed anytime)

Pod vs Container:

Aspect	Pod	Container
Creation	Kubernetes manages	Docker manages
Networking	Shared IP	Individual IP per container
Lifecycle	Managed by controllers	Managed by runtime
Storage	Can share volumes	Isolated storage
Scaling	Via ReplicaSets/Deployments	Manual scaling

Example Pod YAML:

```

apiVersion: v1
kind: Pod
metadata:
  name: nginx-pod
spec:
  containers:
    - name: nginx
      image: nginx:1.21
      ports:
        - containerPort: 80
      resources:
        requests:
          cpu: 100m
          memory: 128Mi
        limits:
          cpu: 500m
          memory: 512Mi

```

When to use Multi-Container Pods:

- Sidecar containers (logging, monitoring)

- Ambassador containers (proxy, configuration management)
 - Init containers (setup tasks)
 - Tightly coupled components that must share resources
-

Q4: What is a Node in Kubernetes?

Answer:

A **Node** is a worker machine in the Kubernetes cluster that runs containerized applications.

Node Types:

1. **Worker Node:** Runs application pods
2. **Master/Control Plane Node:** Manages cluster state and scheduling

Node Characteristics:

- Can be physical machine or virtual machine
- Registers itself with API server
- Has unique name and IP address
- Runs kubelet, kube-proxy, and container runtime
- Has labels for scheduling decisions

Node Status Contains:

- **Address:** Hostname, internal IP, external IP
- **Condition:** Ready, MemoryPressure, DiskPressure, PIDPressure, NetworkUnavailable
- **Capacity:** Total resources available (CPU, memory, pods)
- **Allocatable:** Resources available for pod scheduling

Check Node Status:

```
kubectl get nodes  
kubectl describe node <node-name>
```

```
kubectl top nodes # Resource usage
```

Q5: What is a Namespace and why use them?

Answer:

A **Namespace** is a virtual cluster within a physical Kubernetes cluster, providing logical isolation.

Key Benefits:

- **Resource Isolation:** Separate resources for different teams/projects
- **Multi-Tenancy:** Multiple teams can share same cluster safely
- **Resource Quotas:** Limit resources per namespace
- **RBAC:** Different access controls per namespace
- **Network Policies:** Isolate network traffic between namespaces
- **Naming Scope:** Same names can exist in different namespaces

Default Namespaces:

- `default` : Default namespace for resources
- `kube-system` : Kubernetes system components
- `kube-public` : Public resources
- `kube-node-lease` : Node heartbeat data

Create and Use Namespace:

```
kubectl create namespace dev  
kubectl run nginx --image=nginx --namespace=dev  
kubectl get pods --namespace=dev
```

Use Cases:

- Separate development, staging, production environments
- Multi-team environments with resource isolation
- Compliance requirements for data separation

Q6: What are Labels and Selectors?

Answer:Labels are key-value pairs attached to Kubernetes objects for organization and identification.

Selectors filter resources based on labels.

Example:

```
apiVersion: v1
kind: Pod
metadata:
  name: web-pod
labels:
  app: web-server
  environment: production
  tier: frontend
spec:
  containers:
    - name: nginx
      image: nginx
```

Selector Types:

1. Equality-based:

```
kubectl get pods -l environment=production
kubectl get pods -l app!=web-server
```

1. Set-based:

```
kubectl get pods -l "environment in (prod,staging)"
kubectl get pods -l "tier notin (backend,database)"
```

Use Cases:

- Service discovery and load balancing
- Pod scheduling (node selectors)

- RBAC policies
 - Network policies
 - Resource quotas
 - Monitoring and logging
-

SECTION 2: DEPLOYMENTS & WORKLOAD MANAGEMENT

Q7: Explain Deployments and their lifecycle

Answer:

A **Deployment** is a higher-level abstraction that manages Pods and ReplicaSets, providing declarative updates.

Deployment Structure:

```
Deployment
  ↘
ReplicaSet (v1)
  ↘
Pod Replicas (old version)
```

```
Deployment (after update)
  ↘
ReplicaSet (v2)
  ↘
Pod Replicas (new version)
```

Key Features:

- **Declarative Updates:** Define desired state, Kubernetes handles updates
- **Rollback Support:** Easy rollback to previous versions
- **Rolling Updates:** Gradually replace old pods with new ones
- **Scaling:** Horizontal pod scaling

- **Self-Healing:** Replaces failed pods

Deployment YAML Example:

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: nginx-deployment
spec:
  replicas: 3
  strategy:
    type: RollingUpdate
    rollingUpdate:
      maxSurge: 1
      maxUnavailable: 1
  selector:
    matchLabels:
      app: nginx
  template:
    metadata:
      labels:
        app: nginx
    spec:
      containers:
        - name: nginx
          image: nginx:1.21
          ports:
            - containerPort: 80
      resources:
        requests:
          cpu: 100m
          memory: 128Mi
        limits:
          cpu: 500m
          memory: 512Mi
```

Deployment Lifecycle:

1. **Creation:** Deployment creates ReplicaSet, which creates Pods
2. **Scaling:** Update replicas field to scale up/down
3. **Update:** Change image/config triggers rolling update
4. **Rollback:** Revert to previous ReplicaSet if issues occur
5. **Deletion:** Cascading deletion of ReplicaSets and Pods

Common Commands:

```
kubectl create deployment nginx --image=nginx  
kubectl scale deployment nginx --replicas=5  
kubectl set image deployment/nginx nginx=nginx:1.22  
kubectl rollout status deployment/nginx  
kubectl rollout history deployment/nginx  
kubectl rollout undo deployment/nginx # Rollback to previous
```

Q8: What is the difference between Deployment, StatefulSet, and DaemonSet?

Answer:

Feature	Deployment	StatefulSet	DaemonSet
Use Case	Stateless apps	Stateful apps (databases)	Node-level services
Pod Identity	Interchangeable	Unique, stable (pod-0, pod-1)	One per node
Storage	Shared/No persistence	Persistent per pod	Usually none
Ordering	Parallel	Ordered creation/deletion	Parallel
Scaling	Easy, any replica	Ordered, carefully	One per node
Examples	Web servers, APIs	MySQL, PostgreSQL, Kafka	Logging agents, monitoring

Feature	Deployment	StatefulSet	DaemonSet
Network Identity	ClusterIP	Stable, predictable	Variable
Update Strategy	RollingUpdate, Recreate	Rolling with ordering	Rolling on each node

StatefulSet Example:

```

apiVersion: apps/v1
kind: StatefulSet
metadata:
  name: mysql
spec:
  serviceName: mysql
  replicas: 3
  selector:
    matchLabels:
      app: mysql
  template:
    metadata:
      labels:
        app: mysql
    spec:
      containers:
        - name: mysql
          image: mysql:8.0
          ports:
            - containerPort: 3306
      volumeMounts:
        - name: mysql-storage
          mountPath: /var/lib/mysql
  volumeClaimTemplates:
    - metadata:
        name: mysql-storage
      spec:
        accessModes: ["ReadWriteOnce"]

```

```
resources:  
  requests:  
    storage: 10Gi
```

DaemonSet Example:

```
apiVersion: apps/v1  
kind: DaemonSet  
metadata:  
  name: fluentd-logging  
spec:  
  selector:  
    matchLabels:  
      name: fluentd-logging  
  template:  
    metadata:  
      labels:  
        name: fluentd-logging  
    spec:  
      tolerations:  
        - key: node-role.kubernetes.io/master  
          effect: NoSchedule  
      containers:  
        - name: fluentd  
          image: fluentd:v1.14  
          volumeMounts:  
            - name: varlog  
              mountPath: /var/log  
            - name: varlibdockercontainers  
              mountPath: /var/lib/docker/containers  
              readOnly: true  
          volumes:  
            - name: varlog  
              hostPath:  
                path: /var/log  
            - name: varlibdockercontainers
```

```
hostPath:  
  path: /var/lib/docker/containers
```

When to Use:

- **Deployment:** Web apps, microservices, REST APIs, stateless workloads
- **StatefulSet:** Databases, message queues, distributed systems (Elasticsearch, Kafka)
- **DaemonSet:** Logging agents (Fluentd), monitoring (Prometheus Node Exporter), network plugins

Q9: What is a ReplicaSet?

Answer:

A **ReplicaSet** ensures a specified number of pod replicas are running at all times.

Key Characteristics:

- Created automatically by Deployments (not used directly)
- Continuously monitors pods and creates/deletes as needed
- Uses selectors to identify pods
- Supports both equality and set-based selectors

ReplicaSet vs Replication Controller:

- **ReplicaSet:** Newer, supports set-based selectors
- **Replication Controller:** Older, only equality-based selectors

ReplicaSet YAML:

```
apiVersion: apps/v1  
kind: ReplicaSet  
metadata:  
  name: nginx-rs  
spec:  
  replicas: 3  
  selector:
```

```
matchLabels:  
  app: nginx  
template:  
  metadata:  
    labels:  
      app: nginx  
spec:  
  containers:  
  - name: nginx  
    image: nginx:1.21
```

SECTION 3: SERVICES & NETWORKING

Q10: What is a Kubernetes Service and why is it needed?

Answer:

A **Service** abstracts a set of pods and provides a stable network endpoint for accessing them.

Why Services are Needed:

- Pods are ephemeral (IPs change when pods restart)
- Services provide stable IP and DNS name
- Enable service discovery within cluster
- Load balance traffic across pods
- Expose services internally or externally

Service YAML Example:

```
apiVersion: v1  
kind: Service  
metadata:  
  name: nginx-service  
spec:  
  selector:
```

```
app: nginx
ports:
- protocol: TCP
  port: 80 # Service port
  targetPort: 8080 # Pod port
type: ClusterIP
```

Q11: Explain the four types of Kubernetes Services

Answer:

1. ClusterIP (Default)

- Internal service accessible only within cluster
- Assigns virtual IP from cluster IP range
- Used for internal pod-to-pod communication
- No external access

```
apiVersion: v1
kind: Service
metadata:
  name: internal-service
spec:
  type: ClusterIP
  selector:
    app: api
  ports:
  - port: 80
    targetPort: 8080
```

2. NodePort

- Exposes service on each node's IP at a static port (30000-32767)
- Accessible from outside cluster via <NodeIP>:<NodePort>
- Creates ClusterIP service internally

- Less secure, suitable for development

```
apiVersion: v1
kind: Service
metadata:
  name: web-service
spec:
  type: NodePort
  selector:
    app: web
  ports:
  - port: 80
    targetPort: 8080
    nodePort: 30080 # Optional, auto-assigned if not specified
```

3. LoadBalancer

- Exposes service via cloud provider's load balancer
- Gets external IP from cloud provider
- Each service creates separate load balancer (expensive)
- Recommended for production external services

```
apiVersion: v1
kind: Service
metadata:
  name: api-lb
spec:
  type: LoadBalancer
  selector:
    app: api
  ports:
  - port: 80
    targetPort: 8080
```

4. ExternalName

- Maps service to external hostname/domain
- Returns CNAME record pointing to external service
- No proxy, direct DNS redirect

```
apiVersion: v1
kind: Service
metadata:
  name: external-db
spec:
  type: ExternalName
  externalName: mysql.example.com
```

Service Selection Summary:

Internal communication â†' ClusterIP
 Development/Testing â†' NodePort
 Production external â†' LoadBalancer
 External resource mapping â†' ExternalName

Q12: What is an Ingress and how does it differ from Service?

Answer:

An **Ingress** manages external HTTP/HTTPS access to services within a cluster.

Ingress vs Service:

Aspect	Service	Ingress
Layer	Layer 4 (Transport)	Layer 7 (Application)
Protocol	TCP/UDP	HTTP/HTTPS
Routing	Port-based	Path/hostname-based
Use Case	Direct pod access	Web application routing
Cost	Per service load balancer	Single load balancer for multiple apps

Ingress YAML Example:

```
apiVersion: networking.k8s.io/v1
kind: Ingress
metadata:
  name: multi-app-ingress
  annotations:
    cert-manager.io/cluster-issuer: letsencrypt-prod
spec:
  ingressClassName: nginx
  tls:
    - hosts:
        - app1.example.com
        - app2.example.com
      secretName: tls-secret
  rules:
    - host: app1.example.com
      http:
        paths:
          - path: /
            pathType: Prefix
        backend:
          service:
            name: app1-service
            port:
              number: 80
    - host: app2.example.com
      http:
        paths:
          - path: /api
            pathType: Prefix
        backend:
          service:
            name: app2-service
            port:
              number: 8080
          - path: /static
```

```
pathType: Prefix
backend:
  service:
    name: static-service
    port:
      number: 3000
```

Ingress Requirements:

- Ingress Controller (nginx, Traefik, HAProxy)
- Service for each backend application
- DNS pointing to ingress IP

Q13: What is a Network Policy?

Answer:

A **Network Policy** defines rules for pod-to-pod and pod-to-external communication.

Default Behavior:

- By default, all pods can communicate with all pods
- Network policies restrict this communication

Network Policy YAML:

```
apiVersion: networking.k8s.io/v1
kind: NetworkPolicy
metadata:
  name: app-network-policy
spec:
  podSelector:
    matchLabels:
      role: database
  policyTypes:
    - Ingress
    - Egress
```

```
ingress:  
- from:  
  - podSelector:  
    matchLabels:  
      role: app  
  ports:  
    - protocol: TCP  
      port: 3306  
egress:  
- to:  
  - podSelector:  
    matchLabels:  
      role: app  
  ports:  
    - protocol: TCP  
      port: 8080
```

Key Concepts:

- **podSelector:** Which pods the policy applies to
- **policyTypes:** Ingress (incoming) and/or Egress (outgoing) rules
- **from/to:** Source/destination for traffic
- **ports:** Protocol and port restrictions

Best Practices:

- Implement least privilege access
- Deny all by default, allow specific traffic
- Use pod labels for fine-grained control
- Monitor and audit network policies

SECTION 4: STORAGE & PERSISTENCE

Q14: Explain Persistent Volumes (PV) and Persistent Volume Claims (PVC)

Answer:

Storage in Kubernetes is managed via Persistent Volumes and claims.

Persistent Volume (PV):

- Storage resource provisioned by administrator
- Cluster-level resource (not namespaced)
- Has lifecycle independent of pods
- Supports various storage backends (NFS, iSCSI, cloud storage)

Persistent Volume Claim (PVC):

- User's request for storage
- Binds to a PV
- Can specify storage size and access modes
- Namespaced resource

PV and PVC YAML:

```
# Persistent Volume
apiVersion: v1
kind: PersistentVolume
metadata:
  name: mysql-pv
spec:
  capacity:
    storage: 50Gi
  accessModes:
    - ReadWriteOnce
  persistentVolumeReclaimPolicy: Retain
  storageClassName: fast-ssd
  nfs:
    server: nfs.example.com
    path: "/data/mysql"
```

```
---  
# Persistent Volume Claim  
apiVersion: v1  
kind: PersistentVolumeClaim  
metadata:  
  name: mysql-pvc  
  namespace: production  
spec:  
  accessModes:  
    - ReadWriteOnce  
  storageClassName: fast-ssd  
resources:  
  requests:  
    storage: 50Gi  
  
---  
# Pod using PVC  
apiVersion: v1  
kind: Pod  
metadata:  
  name: mysql  
spec:  
  containers:  
    - name: mysql  
      image: mysql:8.0  
      volumeMounts:  
        - name: mysql-storage  
          mountPath: /var/lib/mysql  
  volumes:  
    - name: mysql-storage  
      persistentVolumeClaim:  
        claimName: mysql-pvc
```

Access Modes:

- **ReadWriteOnce (RWO)**: Single node read-write
- **ReadOnlyMany (ROX)**: Multiple nodes read-only
- **ReadWriteMany (RWX)**: Multiple nodes read-write

Reclaim Policies:

- **Retain**: Manual cleanup required
- **Delete**: Automatically deleted with PVC
- **Recycle**: Basic scrub (deprecated)

Q15: What is a StorageClass?

Answer:

A **StorageClass** enables dynamic provisioning of Persistent Volumes.

Benefits:

- Automatically create PVs based on PVC requests
- No manual PV creation needed
- Different storage tiers/types
- Customizable provisioning parameters

StorageClass Example:

```
apiVersion: storage.k8s.io/v1
kind: StorageClass
metadata:
  name: fast-ssd
provisioner: kubernetes.io/aws-ebs
allowVolumeExpansion: true
parameters:
  type: gp3
  iops: "3000"
  throughput: "125"
  encrypted: "true"
volumeBindingMode: WaitForFirstConsumer
```

```
---
# Using StorageClass
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
  name: app-storage
spec:
  accessModes:
    - ReadWriteOnce
  storageClassName: fast-ssd
  resources:
    requests:
      storage: 100Gi
```

Dynamic Provisioning Flow:

1. User creates PVC with storageClassName
2. Controller detects PVC creation
3. Provisioner creates PV based on StorageClass parameters
4. PVC binds to newly created PV
5. Pod can use the storage

SECTION 5: CONFIGURATION & SECRETS

Q16: What are ConfigMaps and Secrets?

Answer:

ConfigMap:

- Stores non-sensitive configuration data
- Key-value pairs or files
- Mounted as files or environment variables
- Data visible in plain text

Secret:

- Stores sensitive information (passwords, API keys)
- Base64 encoded (not encrypted by default)
- Automatically injected via volumes or environment variables
- Encryption at rest can be enabled

ConfigMap Example:

```
apiVersion: v1
kind: ConfigMap
metadata:
  name: app-config
data:
  DATABASE_HOST: postgres.prod.svc.cluster.local
  DATABASE_PORT: "5432"
  LOG_LEVEL: info
  app.conf: |
    server {
      listen 8080;
      location / {
        proxy_pass http://backend;
      }
    }
  ---  
# Using ConfigMap
apiVersion: v1
kind: Pod
metadata:
  name: app-pod
spec:
  containers:
  - name: app
    image: myapp:1.0
    envFrom:
```

```
- configMapRef:  
  name: app-config  
volumeMounts:  
- name: config-volume  
  mountPath: /etc/config  
volumes:  
- name: config-volume  
  configMap:  
    name: app-config
```

Secret Example:

```
apiVersion: v1  
kind: Secret  
metadata:  
  name: db-credentials  
type: Opaque  
data:  
  username: dXNlcjIyMzIwMjAxMA== # base64 encoded "username"  
  password: cGFzc3dvcmQ= # base64 encoded "password"  
  
---  
# Using Secret  
apiVersion: v1  
kind: Pod  
metadata:  
  name: db-pod  
spec:  
  containers:  
- name: mysql  
  image: mysql:8.0  
  env:  
- name: MYSQL_ROOT_PASSWORD  
  valueFrom:  
    secretKeyRef:  
      name: db-credentials
```

```
key: password
volumeMounts:
- name: secret-volume
  mountPath: /etc/secrets
  readOnly: true
volumes:
- name: secret-volume
  secret:
    secretName: db-credentials
```

Best Practices:

- Use ConfigMaps for non-sensitive config
- Enable encryption at rest for secrets
- Use external secret management (HashiCorp Vault)
- Avoid storing secrets in environment variables
- Implement RBAC for secret access

SECTION 6: SCALING & RESOURCE MANAGEMENT

Q17: How does Horizontal Pod Autoscaler (HPA) work?

Answer:

HPA automatically adjusts the number of pod replicas based on metrics.

How HPA Works:

1. Metrics server collects resource metrics from kubelets
2. HPA controller queries metrics API
3. Compares current metrics with target metrics
4. Calculates desired replica count
5. Updates deployment/statefulset replica count

HPA YAML Example:

```
apiVersion: autoscaling/v2
kind: HorizontalPodAutoscaler
metadata:
  name: app-hpa
spec:
  scaleTargetRef:
    apiVersion: apps/v1
    kind: Deployment
    name: web-app
  minReplicas: 2
  maxReplicas: 10
  metrics:
    - type: Resource
      resource:
        name: cpu
        target:
          type: Utilization
          averageUtilization: 70
    - type: Resource
      resource:
        name: memory
        target:
          type: AverageValue
          averageValue: 500Mi
    - type: Pods
      pods:
        metric:
          name: custom_requests_per_second
        target:
          type: AverageValue
          averageValue: "1000"
  behavior:
    scaleDown:
      stabilizationWindowSeconds: 300
  policies:
```

```
- type: Percent  
  value: 50  
  periodSeconds: 60  
  
scaleUp:  
  stabilizationWindowSeconds: 0  
  
policies:  
- type: Percent  
  value: 100  
  periodSeconds: 30
```

Scaling Formula:

```
desiredReplicas = ceil(currentMetric / targetMetric) - currentReplicas
```

Metric Types:

- **Resource metrics:** CPU, memory
- **Custom metrics:** Application-specific metrics
- **External metrics:** External monitoring systems

Best Practices:

- Set appropriate min/max replicas
- Configure resource requests for accurate scaling
- Use multiple metrics for better decisions
- Monitor scaling events and adjust thresholds

Q18: What are Resource Requests and Limits?

Answer:

Resource requests and limits control CPU and memory allocation.

Requests:

- Minimum guaranteed resources
- Reserved for the pod

- Scheduler uses this for placement
- Pod QoS depends on requests

Limits:

- Maximum resources a container can use
- If exceeded: CPU is throttled, Memory causes OOM kill
- Not guaranteed to be reserved

Resource Requests/Limits YAML:

```
apiVersion: v1
kind: Pod
metadata:
  name: resource-managed-pod
spec:
  containers:
  - name: app
    image: myapp:1.0
    resources:
      requests:
        cpu: 100m # 0.1 CPU cores
        memory: 256Mi
      limits:
        cpu: 500m # 0.5 CPU cores
        memory: 1Gi
```

CPU Units:

- `1` = 1 CPU core = 1000m (millicores)
- `500m` = 0.5 CPU cores
- `0.1` = 100m

Memory Units:

- `1Mi` = 1 Megabyte
- `1Gi` = 1 Gigabyte

- **1Ki** = 1 Kilobyte

Quality of Service (QoS) Classes:

1. Guaranteed (Highest Priority)

- Requests = Limits for both CPU and memory
- Never evicted unless exceeds limits

2. Burstable (Medium Priority)

- Requests < Limits
- Evicted if node runs out of resources

3. BestEffort (Lowest Priority)

- No requests or limits
- First to be evicted

```
# Guaranteed QoS
```

```
resources:
```

```
  requests:
```

```
    cpu: 100m
```

```
    memory: 256Mi
```

```
limits:
```

```
  cpu: 100m
```

```
  memory: 256Mi
```

```
# Burstable QoS
```

```
resources:
```

```
  requests:
```

```
    cpu: 100m
```

```
    memory: 256Mi
```

```
limits:
```

```
  cpu: 500m
```

```
  memory: 1Gi
```

```
# BestEffort QoS (no resources specified)
```

SECTION 7: TROUBLESHOOTING & DEBUGGING

Q19: How do you debug a failing pod?

Answer:

Follow systematic debugging steps:

Step 1: Check Pod Status

```
kubectl get pods  
kubectl describe pod <pod-name> # Check events and conditions
```

Step 2: View Logs

```
kubectl logs <pod-name> # Latest logs  
kubectl logs <pod-name> --previous # Previous crashed container  
kubectl logs <pod-name> -c <container-name> # Specific container  
kubectl logs <pod-name> -f # Follow logs in real-time
```

Step 3: Execute Commands Inside Pod

```
kubectl exec -it <pod-name> -- /bin/sh  
kubectl exec -it <pod-name> -- cat /etc/config
```

Step 4: Check Resource Availability

```
kubectl describe node <node-name>  
kubectl top nodes  
kubectl top pods
```

Step 5: Common Pod Failure States:

State	Cause	Solution
Pending	Insufficient resources, PVC not bound	Add nodes, check PVC/requests

State	Cause	Solution
ImagePullBackOff	Image not found, invalid credentials	Check image name, registry credentials
CrashLoopBackOff	Application error, bug	Check logs, fix application
Evicted	Node out of resources	Free resources, adjust limits
NotReady	Readiness probe failing	Check probe configuration, app health

Debugging Checklist:

1. Pod Events (kubectl describe pod)
2. Container Logs (kubectl logs)
3. Resource Limits & Requests
4. Readiness & Liveness Probes
5. Network Connectivity
6. Volume Mounts & Storage
7. Environment Variables
8. Application Configuration

Q20: What are Liveness and Readiness Probes?

Answer:

Readiness Probe:

- Checks if pod is ready to receive traffic
- Determines if pod should be part of service endpoints
- Failures mean pod is not ready, traffic not sent
- Used during deployment to avoid sending traffic to initializing pods

Liveness Probe:

- Checks if pod application is still running
- Failures trigger pod restart
- Prevents zombie processes

- Detects deadlocked applications

Probe Types:

1. **HTTP**
2. **TCP Socket**
3. **Exec Command**

Probe Configuration YAML:

```

apiVersion: v1
kind: Pod
metadata:
  name: app-with-probes
spec:
  containers:
  - name: app
    image: myapp:1.0

    # HTTP Readiness Probe
    readinessProbe:
      httpGet:
        path: /health
        port: 8080
        scheme: HTTP
      initialDelaySeconds: 10
      periodSeconds: 5
      timeoutSeconds: 2
      failureThreshold: 3

    # TCP Liveness Probe
    livenessProbe:
      tcpSocket:
        port: 3306
      initialDelaySeconds: 15
      periodSeconds: 10
      timeoutSeconds: 5
  
```

```
failureThreshold: 3

# Exec Command Readiness
readinessProbe:
  exec:
    command:
      - /bin/sh
      - -c
      - mysql -h localhost -u root -ppassword -e "SELECT 1"
  initialDelaySeconds: 5
  periodSeconds: 10
```

Probe Fields:

- **initialDelaySeconds**: Delay before first probe
- **periodSeconds**: Frequency of probes
- **timeoutSeconds**: Time to wait for response
- **successThreshold**: Successes needed to mark as healthy
- **failureThreshold**: Failures before action taken

Best Practices:

- Implement both readiness and liveness probes
- Use lightweight health check endpoints
- Set appropriate thresholds to avoid false positives
- Don't rely on external service calls in probes

SECTION 8: SECURITY & RBAC

Q21: What is Role-Based Access Control (RBAC)?

Answer:

RBAC controls who can perform what actions on Kubernetes resources.

RBAC Components:

1. Role

- Namespace-scoped permissions
- Defines allowed verbs and resources

2. ClusterRole

- Cluster-wide permissions
- Can be used across namespaces

3. RoleBinding

- Binds Role to user/group/service account (namespace)

4. ClusterRoleBinding

- Binds ClusterRole to user/group/service account (cluster-wide)

RBAC YAML Examples:

```
# ClusterRole for reading pods
apiVersion: rbac.authorization.k8s.io/v1
kind: ClusterRole
metadata:
  name: pod-reader
rules:
  - apiGroups: []
    resources: ["pods"]
    verbs: ["get", "list", "watch"]
  - apiGroups: []
    resources: ["pods/logs"]
    verbs: ["get"]
```

```
---
```

```
# ClusterRoleBinding
apiVersion: rbac.authorization.k8s.io/v1
kind: ClusterRoleBinding
metadata:
  name: read-pods-globally
subjects:
```

```
- kind: ServiceAccount
  name: developer
  namespace: default
- kind: User
  name: john@example.com
roleRef:
  kind: ClusterRole
  name: pod-reader
  apiGroup: rbac.authorization.k8s.io

---
# Role for namespace-scoped operations
apiVersion: rbac.authorization.k8s.io/v1
kind: Role
metadata:
  name: deployment-manager
  namespace: production
rules:
- apiGroups: ["apps"]
  resources: ["deployments"]
  verbs: ["create", "get", "update", "patch", "delete"]
- apiGroups: [""]
  resources: ["pods", "pods/logs"]
  verbs: ["get", "list", "watch"]

---
# RoleBinding
apiVersion: rbac.authorization.k8s.io/v1
kind: RoleBinding
metadata:
  name: deploy-manager-binding
  namespace: production
subjects:
- kind: User
  name: alice@example.com
roleRef:
```

```
kind: Role
name: deployment-manager
apiGroup: rbac.authorization.k8s.io
```

Common Verbs:

- `get`, `list`, `watch` - Read operations
- `create`, `update`, `patch` - Write operations
- `delete`, `deletecollection` - Deletion
- `exec`, `attach` - Interactive operations

Service Accounts:

- Identity for pods
- Automatically mounted as volumes
- Used for pod authentication to API server

```
apiVersion: v1
kind: ServiceAccount
metadata:
  name: app-service-account
  namespace: production
```

Q22: What is Pod Security Policy (PSP)?

Answer:

Pod Security Policy defines security standards for pods (being replaced by Pod Security Admission).

PSP Controls:

- Privileged container execution
- Host network/IPC access
- Host port exposure
- Volume types

- Running as root user
- Capabilities
- AppArmor/SELinux labels

PSP Example:

```

apiVersion: policy/v1beta1
kind: PodSecurityPolicy
metadata:
  name: restricted
spec:
  privileged: false
  allowPrivilegeEscalation: false
  requiredDropCapabilities:
    - ALL
  volumes:
    - 'configMap'
    - 'emptyDir'
    - 'projected'
    - 'secret'
    - 'downwardAPI'
    - 'persistentVolumeClaim'
  hostNetwork: false
  hostIPC: false
  hostPID: false
  runAsUser:
    rule: 'MustRunAsNonRoot'
  seLinux:
    rule: 'MustRunAs'
    seLinuxOptions:
      level: "s0:c123,c456"
  fsGroup:
    rule: 'MustRunAs'
    ranges:
      - min: 1

```

```
max: 65535  
readOnlyRootFilesystem: true
```

Pod Security Admission (Kubernetes 1.25+):

```
apiVersion: v1  
kind: Namespace  
metadata:  
  name: production  
  labels:  
    pod-security.kubernetes.io/enforce: restricted  
    pod-security.kubernetes.io/audit: restricted  
    pod-security.kubernetes.io/warn: restricted
```

SECTION 9: ADVANCED TOPICS

Q23: What is a Service Mesh and why use it?

Answer:

A **Service Mesh** manages service-to-service communication infrastructure layer (Layer 5).

Key Features:

1. Traffic Management

- Load balancing algorithms
- Canary deployments
- Circuit breaking
- Retries and timeouts

2. Security

- Mutual TLS (mTLS) encryption
- Authentication and authorization
- Certificate management

3. Observability

- Distributed tracing
- Metrics collection
- Log aggregation

4. Resilience

- Fault injection testing
- Traffic mirroring
- Rate limiting

Popular Service Meshes:

- **Istio**: Full-featured, complex
- **Linkerd**: Lightweight, high-performance
- **Consul**: Multi-cloud, service networking

Benefits:

- Transparent traffic control without application changes
- Security policies at infrastructure level
- Better observability of service communication
- Zero trust networking implementation

Challenges:

- Added complexity and learning curve
- Performance overhead
- Operational burden

Q24: What is an Operator?

Answer:

A **Kubernetes Operator** extends Kubernetes to manage complex applications.

Components:

1. **Custom Resource Definition (CRD)**: New API resource
2. **Controller**: Business logic that reconciles actual vs desired state
3. **Reconciliation Loop**: Continuous monitoring and correction

Operator Pattern Flow:

```
User creates Custom Resource
  +
  Operator detects creation
  +
  Operator performs setup (create databases, services, configs)
  +
  Operator monitors actual state
  +
  If mismatch detected, reconcile to desired state
  +
  Application managed by operator
```

Example Operators:

- **Prometheus Operator**: Manages Prometheus monitoring
- **MySQL Operator**: Manages MySQL database clusters
- **Kafka Operator**: Manages Kafka clusters
- **ArgoCD Operator**: Manages GitOps workflow

Advantages:

- Automate complex operational tasks
- Encode operational knowledge
- Consistent deployments
- Self-healing applications

Q25: What is etcd and its role?

Answer: etcd is a distributed key-value store that stores Kubernetes cluster state.

Role in Kubernetes:

- Single source of truth for cluster state
- Stores all API objects (pods, services, deployments, etc.)
- Enables cluster consistency
- Provides watch mechanism for state changes

Key Characteristics:

- Strongly consistent
- Highly available (cluster mode)
- Fast (optimized for reads)
- RAFT consensus algorithm
- Transactional support

etcd Backup/Restore:

```
# Backup etcd
ETCDCTL_API=3 etcdctl \
--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 \
snapshot save /backup/etcd-snapshot.db

# Restore etcd
ETCDCTL_API=3 etcdctl \
--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 \
snapshot restore /backup/etcd-snapshot.db
```

Best Practices:

- Run etcd in cluster mode (3, 5, or 7 nodes)

- Enable encryption at rest
 - Implement backup strategy
 - Monitor etcd disk I/O and latency
 - Use SSD storage for etcd
-

SECTION 10: REAL-WORLD SCENARIOS (MNC Interview Questions)

Q26: How would you migrate a legacy monolithic application to Kubernetes?

Answer: Migration Strategy:

Phase 1: Containerization

- Containerize the application (create Dockerfile)
- Test container locally
- Push to container registry
- Ensure no host dependencies

Phase 2: Kubernetes Deployment

- Create deployment manifest
- Configure resource requests/limits
- Set up liveness/readiness probes
- Configure environment variables

Phase 3: Data & State Management

- Identify stateful components (databases)
- Move to StatefulSets if needed
- Set up persistent volumes
- Ensure data migration strategy

Phase 4: Networking & Service Discovery

- Expose via Service (ClusterIP/LoadBalancer)
- Configure Ingress if needed
- Update DNS/load balancers
- Test internal connectivity

Phase 5: Monitoring & Logging

- Integrate monitoring (Prometheus)
- Set up centralized logging (ELK, Loki)
- Create alerts
- Dashboard creation

Phase 6: Zero-Downtime Cutover

- Run legacy and Kubernetes versions in parallel
- Gradual traffic migration
- Monitor both environments
- Rollback plan ready
- Full cutover once stable

Challenges & Solutions:

Challenge	Solution
Database migrations	Use database migration tools, backup strategy
Session management	Implement session store (Redis, database)
Configuration differences	Use ConfigMaps, secrets for environment-specific config
Performance issues	Profile, optimize, use resource requests/limits
Logging complexity	Centralized logging, tracing infrastructure

Q27: Design a production-grade Kubernetes cluster for a SaaS platform

Answer:

Cluster Architecture:

```
Internet
  ↗
Cloud Load Balancer
  ↗
Ingress Controller (nginx)
  ↗
API Services (Multiple Replicas)
  ↗
Database Services (StatefulSet)
Database (RDS/Managed)
```

Control Plane:

- Multi-zone deployment (3+ nodes)
- Managed Kubernetes service (EKS, GKE, AKS)
- Auto-backup of etcd
- High availability API server

Worker Nodes:

- Multiple node groups for different workloads
- Auto-scaling enabled
- Regular security patching
- Enough resources for failover

Networking:

```
# Ingress for external traffic
- Use managed load balancer
- SSL/TLS termination
- Rate limiting
- DDoS protection

# Internal networking
- Service discovery via CoreDNS
```

- Network policies for security
- Pod security policies

Storage:

- # Application data
 - Dynamic provisioning via StorageClasses
 - Backup strategy (snapshots)
 - Multi-zone redundancy
- # Database
 - Managed database service (RDS, CloudSQL)
 - Automated backups
 - Point-in-time recovery
 - Read replicas for scaling

Security:

- # Authentication & Authorization
 - RBAC for all users
 - Service accounts with minimal permissions
 - Pod service accounts
- # Network Security
 - Network policies to restrict traffic
 - Private cluster (no public IPs for nodes)
 - VPC security groups
- # Data Security
 - Encryption at rest (etcd, volumes)
 - Encryption in transit (TLS)
 - Secrets management (Vault)
- # Container Security
 - Image scanning for vulnerabilities

- Private container registry
- Pod security policies

Monitoring & Logging:

Metrics

- Prometheus for collection
- Grafana for visualization
- Custom metrics for application

Logging

- ELK/Loki for centralized logging
- Fluentd/Filebeat for collection
- Kibana/Grafana for analysis

Tracing

- Jaeger for distributed tracing
- OpenTelemetry SDK in applications

Alerting

- Alert rules for anomalies
- Notification channels (PagerDuty, Slack)
- Runbooks for incidents

CI/CD Integration:

Git-based workflow

- GitOps for deployments
- ArgoCD or Flux for reconciliation
- Automated testing in pipelines
- Canary/Blue-Green deployments

Secrets Management

- External secret store (Vault)

- Automatic rotation
- RBAC for secret access

Disaster Recovery:

- # Backup Strategy
 - Regular cluster state backups
 - Database backups (automated)
 - Multi-region setup for failover
 - RTO/RPO defined and tested

- # Scaling
 - Horizontal Pod Autoscaler
 - Cluster Autoscaler
 - Vertical Pod Autoscaler for fine-tuning

Q28: How would you handle scaling issues in a Kubernetes cluster under high traffic?

Answer:

Identifying Scaling Issues:

```
# Check current resource usage  
kubectl top nodes  
kubectl top pods --all-namespaces  
  
# Check pod scheduling failures  
kubectl get pods --field-selector=status.phase=Pending  
  
# Check HPA status  
kubectl get hpa  
kubectl describe hpa <hpa-name>
```

```
# Check events for errors
kubectl get events --all-namespaces --sort-by='.lastTimestamp'
```

Scaling Strategy:

1. Pod-Level Scaling (HPA)

```
apiVersion: autoscaling/v2
kind: HorizontalPodAutoscaler
metadata:
  name: api-hpa
spec:
  scaleTargetRef:
    apiVersion: apps/v1
    kind: Deployment
    name: api-server
  minReplicas: 5
  maxReplicas: 100
  metrics:
    - type: Resource
      resource:
        name: cpu
        target:
          type: Utilization
          averageUtilization: 70
    - type: Resource
      resource:
        name: memory
        target:
          type: Utilization
          averageUtilization: 80
  behavior:
    scaleUp:
      stabilizationWindowSeconds: 0
    policies:
      - type: Percent
```

```
    value: 200
    periodSeconds: 30
    selectPolicy: Max
  scaleDown:
    stabilizationWindowSeconds: 300
    policies:
      - type: Percent
        value: 50
        periodSeconds: 60
```

2. Node-Level Scaling (Cluster Autoscaler)

```
# Deploy Cluster Autoscaler
# Works with AWS EKS, GKE, AKS

apiVersion: apps/v1
kind: Deployment
metadata:
  name: cluster-autoscaler
  namespace: kube-system
spec:
  replicas: 1
  selector:
    matchLabels:
      app: cluster-autoscaler
  template:
    metadata:
      labels:
        app: cluster-autoscaler
    spec:
      serviceAccountName: cluster-autoscaler
      containers:
        - image: k8s.gcr.io/autoscaling/cluster-autoscaler:v1.24.0
          name: cluster-autoscaler
          command:
            - ./cluster-autoscaler
```

```
- --cloud-provider=aws  
- --nodes=1:100:asg-name  
- --skip-nodes-with-local-storage=false
```

3. Resource Optimization

```
# Set appropriate requests and limits  
resources:  
  requests:  
    cpu: 100m  
    memory: 128Mi  
  limits:  
    cpu: 500m  
    memory: 512Mi  
  
# Use resource quotas per namespace  
apiVersion: v1  
kind: ResourceQuota  
metadata:  
  name: team-quota  
spec:  
  hard:  
    requests.cpu: "100"  
    requests.memory: "200Gi"  
    limits.cpu: "200"  
    limits.memory: "400Gi"
```

4. Connection Pool & Rate Limiting

```
# Configure Ingress rate limiting  
apiVersion: networking.k8s.io/v1  
kind: Ingress  
metadata:  
  name: api-ingress  
  annotations:  
    nginx.ingress.kubernetes.io/limit-rps: "1000"
```

```
spec:  
  ingressClassName: nginx  
  rules:  
    - host: api.example.com  
      http:  
        paths:  
          - path: /  
            pathType: Prefix  
        backend:  
          service:  
            name: api-service  
            port:  
              number: 8080
```

5. Cache Layer

```
# Deploy Redis cache  
apiVersion: apps/v1  
kind: Deployment  
metadata:  
  name: redis-cache  
spec:  
  replicas: 3  
  selector:  
    matchLabels:  
      app: redis  
  template:  
    metadata:  
      labels:  
        app: redis  
    spec:  
      containers:  
        - name: redis  
          image: redis:7.0  
      ports:  
        - containerPort: 6379
```

```
resources:  
  requests:  
    cpu: 500m  
    memory: 1Gi  
  limits:  
    cpu: 1000m  
    memory: 2Gi
```

Monitoring Scaling:

```
# Watch HPA scaling  
kubectl get hpa -w  
  
# Check scaling events  
kubectl describe hpa api-hpa  
  
# View node autoscaler logs  
kubectl logs -n kube-system -l app=cluster-autoscaler -f  
  
# Metrics monitoring  
kubectl get --raw /apis/metrics.k8s.io/v1beta1/nodes
```

Q29: How would you handle a Pod that crashes repeatedly?

Answer:

Step 1: Identify the Issue

```
kubectl describe pod <pod-name> # Check events and conditions  
kubectl logs <pod-name> --previous # View logs from previous crash  
kubectl logs <pod-name> -f # Follow current logs
```

Common Causes & Solutions:

Issue	Symptoms	Solution
Application Bug	Container exits immediately	Review logs, fix code, redeploy
Memory Leak	OOMKilled	Increase memory limit, profile app
Infinite Loop	CPU spike then restart	Debug application code
External Dependency	Cannot connect to database	Check connectivity, ensure dependency is running
Configuration Error	Application fails to start	Validate ConfigMaps, Secrets, environment variables
Image Issue	ImagePullBackOff	Check image name, registry credentials
Resource Limits	Insufficient CPU/memory	Increase limits, optimize application

Crash Loop Backoff:

```
# Kubernetes auto-restart with exponential backoff
# Delays: 10s, 20s, 40s, 80s, 160s, 300s (max)

# To prevent restart
spec:
  restartPolicy: Never # Or OnFailure

# To debug without restart
kubectl run debug-pod --image=alpine --restart=Never -- sleep 3600
kubectl exec -it debug-pod -- sh
```

Debugging Steps:

```
# 1. Check events
kubectl describe pod <pod-name>

# 2. View application logs
kubectl logs <pod-name>
kubectl logs <pod-name> --previous

# 3. Check resource usage
```

```

kubectl describe pod <pod-name> | grep -A 5 "Limits\|Requests"

# 4. Check environment and configs
kubectl exec <pod-name> -- env
kubectl exec <pod-name> -- cat /etc/config/app.conf

# 5. Test connectivity
kubectl exec <pod-name> -- ping database-service
kubectl exec <pod-name> -- curl http://api-service:8080/health

# 6. Check readiness probe
kubectl get pod <pod-name> -o yaml | grep -A 10 "readinessProbe"

# 7. Test locally
docker run --rm myapp:latest

```

Prevention Strategies:

```

# Implement readiness probes
readinessProbe:
  httpGet:
    path: /health
    port: 8080
  initialDelaySeconds: 10
  periodSeconds: 5
  failureThreshold: 3

# Set resource limits
resources:
  requests:
    cpu: 100m
    memory: 256Mi
  limits:
    cpu: 500m
    memory: 1Gi

```

```

# Use init containers for setup
initContainers:
- name: setup
  image: busybox
  command: ['sh', '-c', 'until nslookup database; do echo waiting; sleep 2; done']
# Add startup probe (K8s 1.18+)
startupProbe:
  httpGet:
    path: /startup
    port: 8080
  initialDelaySeconds: 0
  periodSeconds: 10
  failureThreshold: 30

```

Q30: Design a disaster recovery strategy for a Kubernetes cluster

Answer:

RTO/RPO Definition:

- **RTO (Recovery Time Objective):** Maximum acceptable downtime (target: 15-30 minutes)
- **RPO (Recovery Point Objective):** Maximum data loss acceptable (target: 5-15 minutes)

Multi-Layer DR Strategy:

1. Cluster-Level Backup:

```

# Backup etcd (cluster state)
ETCDCTL_API=3 etcdctl snapshot save etcd-backup-$(date +%s).db

# Script for automated backups
#!/bin/bash
BACKUP_DIR="/backups/etcd"

```

```

mkdir -p $BACKUP_DIR

ETCDCTL_API=3 etcdctl \
--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 \
snapshot save $BACKUP_DIR/etcd-$(date +%Y%m%d-%H%M%S).db

# Cleanup old backups (keep last 7 days)
find $BACKUP_DIR -type f -mtime +7 -delete

```

2. Application-Level Backup:

```

# Backup volumes
apiVersion: v1
kind: CronJob
metadata:
  name: volume-backup
spec:
  schedule: "*/5 * * * *" # Every 5 minutes
  jobTemplate:
    spec:
      template:
        spec:
          containers:
            - name: backup
              image: velero:latest
              command:
                - velero
                - backup
                - create
                - --include-namespaces=*
  restartPolicy: OnFailure

```

3. Multi-Region Setup:

Region 1 (Primary)
 Cluster A (Active)
 Database (Primary)
 Load Balancer

Region 2 (Standby)
 Cluster B (Standby)
 Database (Replica)
 Load Balancer

Global DNS (Route53, CloudDNS)
 Route 90% to Region 1
 Route 10% to Region 2 (canary)

4. Database Replication:

Primary-Standby replication
Primary DB:
 Write operations
 Replication to Standby

Standby DB:
 Read-only replica
 Can be promoted to primary

5. GitOps for Infrastructure Recovery:

Store all configurations in Git
Use ArgoCD to reconcile cluster state

```
apiVersion: argoproj.io/v1alpha1
kind: Application
metadata:
  name: app-deployment
spec:
```

```
project: default
source:
  repoURL: https://github.com/company/k8s-configs
  targetRevision: main
  path: production/
destination:
  server: https://kubernetes.default.svc
  namespace: production
syncPolicy:
  automated:
    prune: true
    selfHeal: true
```

6. Testing DR:

```
# Regular DR drills
# 1. Backup cluster to secondary
# 2. Restore from backup to test cluster
# 3. Verify data consistency
# 4. Validate application functionality
# 5. Measure RTO
# 6. Document findings and improvements

# Automated DR test script
#!/bin/bash
# Create test cluster from backup
# Run smoke tests
# Measure recovery time
# Generate report
```

7. Monitoring & Alerting:

```
# Monitor backup health
- alert: BackupFailed
  expr: backup_success{namespace="velero"} == 0
  for: 1h
```

```

annotations:
  summary: "Velero backup failed"

# Monitor replication lag
- alert: DatabaseReplicationLag
  expr: replication_lag_seconds > 60
  annotations:
    summary: "Database replication lagging"

# Monitor cluster health
- alert: KubernetesNodeDown
  expr: up{job="kubernetes-nodes"} == 0
  for: 5m

```

KEY TAKEAWAYS

For MNC Interviews:

1. Master core concepts (Pods, Deployments, Services, Storage)
2. Understand troubleshooting and debugging methodologies
3. Know security best practices (RBAC, Network Policies, Secrets Management)
4. Be prepared for real-world scenarios (migrations, scaling, DR)
5. Demonstrate hands-on experience with kubectl and manifests
6. Understand production considerations (monitoring, logging, HA)
7. Know different deployment strategies (Rolling, Blue-Green, Canary)
8. Understand networking and service mesh concepts
9. Be familiar with ecosystem tools (Helm, Prometheus, Loki, Istio)
10. Practice troubleshooting by actually deploying and breaking things

Preparation Tips:

- Set up a local Kubernetes cluster (Minikube, Docker Desktop)
- Deploy real applications and troubleshoot issues

- Read Kubernetes documentation thoroughly
 - Follow Kubernetes best practices guide
 - Study case studies from companies using Kubernetes
 - Practice mock interviews with scenario-based questions
 - Understand your cloud provider's managed Kubernetes service (EKS, GKE, AKS)
-

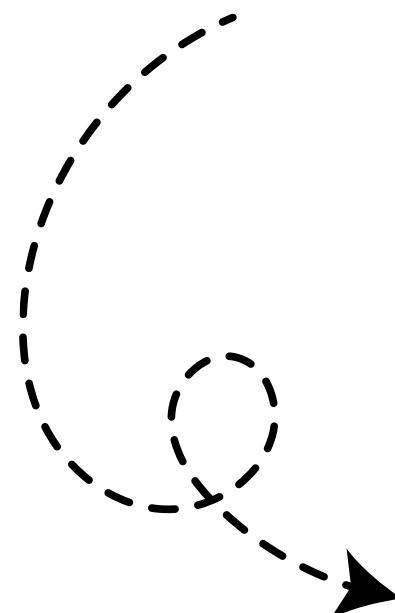
Interview Level: Beginner to Advanced (5+ years experience)

Target Companies: Google, Amazon, Microsoft, Meta, Apple, Netflix, Uber, Airbnb, etc.

Repost and Follow

Nensi Ravaliya

for more content



**Want to build your
career in cloud?**

**Subscribe to
Yatri Cloud Channel**

