Complete Kubernetes Theory Guide - From Zero to Expert

Table of Contents

1. What is Kubernetes and Why Does it Exist?
2. Core Architecture & Components
3. The Pod: Basic Unit of Deployment
4. Controllers: Managing Application State
5. Services & Networking
6. Storage & Persistence
7. Configuration Management
8. Security Model
9. Scaling & Resource Management
10. Advanced Concepts
11. Production Considerations
12. Interview Questions & Scenarios

What is Kubernetes and Why Does it Exist?

The Problem Kubernetes Solves

Before containerization:

- Applications ran directly on servers
- "Works on my machine" problems
- Difficult to scale and manage dependencies
- Resource waste (servers running at 10-20% utilization)

With containers (Docker era):

- Consistent packaging and deployment
- Better resource utilization

- Still needed orchestration at scale

The orchestration challenge:

```
# Managing containers manually becomes impossible at scale
docker run app1 on server1
docker run app1 on server2
docker run app2 on server3
# What if server2 dies? How do you load balance? Auto-scale?
```

What Kubernetes Actually Is

Definition: Kubernetes is a **container orchestration platform** that automates the deployment, scaling, and management of containerized applications.

Key insight: Kubernetes is a **desired state management system**

- You declare what you want (5 replicas of your app)
- Kubernetes continuously ensures that state is maintained
- If reality differs from desired state, Kubernetes takes action to fix it

The Declarative Model

Imperative (traditional):

```
# Step-by-step commands
ssh server1
docker stop old-app
docker run new-app
ssh server2
docker stop old-app
docker run new-app
```

Declarative (Kubernetes):

```yaml
# Describe desired end state
apiVersion: apps/v1
kind: Deployment
metadata:
  name: my-app
spec:
  replicas: 3
  template:
    spec:
      containers:
      - name: app
        image: my-app:v2  # Just change this
```

Benefits of declarative:

- **Idempotent:** Apply same config multiple times = same result
- **Self-healing:** System automatically corrects drift
- **Version controlled:** Infrastructure as Code
- **Predictable:** Same config = same outcome everywhere

Core Architecture & Components

Master Node (Control Plane)

The "brain" of the cluster that makes all decisions:

1. API Server (`kube-apiserver`)

- RESTful API that serves as the front-end
- All communication goes through the API server
- Validates and persists configuration
- Only component that talks to etcd

2. etcd

- Distributed key-value store
- Single source of truth for cluster state
- Stores all configuration, secrets, and current state
- Uses Raft consensus algorithm for consistency

3. Scheduler (`kube-scheduler`)

- Decides which node should run each pod
- Considers resource requirements, constraints, policies
- **Scheduling factors:**
  - Resource availability (CPU, memory)
  - Node selectors and affinity rules
  - Taints and tolerations
  - Data locality

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

- Runs multiple controllers that regulate cluster state
- **Key controllers:**
  - **Deployment Controller:** Manages ReplicaSets
  - **ReplicaSet Controller:** Ensures pod replicas
  - **Node Controller:** Monitors node health
  - **Service Controller:** Creates load balancers

Worker Nodes

Where your applications actually run:

1. kubelet

- Node agent that communicates with API server
- Manages pod lifecycle on the node
- Reports node and pod status back to control plane
- Pulls container images and starts/stops containers

2. kube-proxy

- Network proxy that implements Kubernetes Service concept
- Maintains network rules for pod-to-pod communication
- Implements load balancing for Services
- **Modes:** iptables (default), IPVS, userspace

3. Container Runtime

- Actually runs containers (Docker, containerd, CRI-O)
- Manages container images and lifecycle
- Implements Container Runtime Interface (CRI)

The Control Loop Pattern

Core concept: Everything in Kubernetes follows this pattern:

```
1. OBSERVE current state
2. COMPARE with desired state
3. ACT to reconcile differences
4. REPEAT continuously
```

Example - Deployment Controller:

```yaml
# Desired state: 3 replicas
spec:
  replicas: 3

# Current state: 2 pods running (1 crashed)
# Action: Create 1 new pod
# Result: 3 pods running = desired state achieved
```

The Pod: Basic Unit of Deployment

What is a Pod?

Definition: A pod is the smallest deployable unit in Kubernetes, containing one or more containers that share:

- Network namespace (same IP address)
- Storage volumes
- Lifecycle (created/destroyed together)

Why Pods, Not Just Containers?

Multi-container patterns:

1. Sidecar Pattern

```yaml
# Main app + logging sidecar
containers:
- name: web-app
  image: nginx
- name: log-shipper
  image: fluentd
  # Shares volumes with web-app to access logs
```

2. Ambassador Pattern

```yaml
# Main app + proxy sidecar
containers:
- name: app
  image: my-app
- name: proxy
  image: nginx-proxy
  # Handles SSL termination, load balancing
```

3. Adapter Pattern

```yaml
# Main app + format adapter
containers:
- name: app
  image: legacy-app
- name: adapter
  image: metrics-adapter
  # Converts legacy metrics to Prometheus format
```

Pod Networking

Key concepts:

- Each pod gets a unique IP address
- Containers in a pod communicate via `localhost`
- Pods communicate directly using pod IPs
- No NAT between pods (flat network)

Example:

```yaml
# Pod A (IP: 10.244.1.10)
containers:
- name: web
  ports:
  - containerPort: 8080
- name: cache
  ports:
  - containerPort: 6379

# Web container connects to cache: localhost:6379
# Other pods connect to web: 10.244.1.10:8080
```

Pod Lifecycle

Phases:

1. **Pending:** Pod accepted but not yet running
2. **Running:** At least one container is running
3. **Succeeded:** All containers terminated successfully
4. **Failed:** At least one container failed
5. **Unknown:** Cannot determine pod status

Container states:

- **Waiting:** Not running (pulling image, waiting for secrets)
- **Running:** Executing normally
- **Terminated:** Finished execution (success or failure)

Init Containers

Purpose: Run before main containers to perform setup tasks

```yaml
spec:
  initContainers:
  - name: migration
    image: migrate-tool
    command: ['migrate', 'database']
  - name: permission-fix
    image: busybox
    command: ['chmod', '777', '/shared-volume']
  containers:
  - name: app
    image: my-app
    # Starts only after init containers complete
```

Use cases:

- Database migrations
- Configuration setup
- Waiting for dependencies
- Security setup

Controllers: Managing Application State

ReplicaSet

Purpose: Ensures a specified number of pod replicas are running

```yaml
apiVersion: apps/v1
kind: ReplicaSet
metadata:
  name: frontend
spec:
  replicas: 3
  selector:
    matchLabels:
      app: frontend
  template:
    metadata:
      labels:
        app: frontend
    spec:
      containers:
      - name: web
        image: nginx
```

How it works:

1. Continuously monitors pods matching selector
2. If fewer pods than desired: creates new pods
3. If more pods than desired: deletes excess pods
4. Uses pod template to create new pods

Deployment

Purpose: Provides declarative updates for ReplicaSets and Pods

```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: nginx-deployment
spec:
  replicas: 3
  strategy:
    type: RollingUpdate
    rollingUpdate:
      maxUnavailable: 1
      maxSurge: 1
  selector:
    matchLabels:
      app: nginx
  template:
    metadata:
      labels:
        app: nginx
    spec:
      containers:
      - name: nginx
        image: nginx:1.20
```

Deployment strategies:

1. Rolling Update (default)

```yaml
strategy:
  type: RollingUpdate
  rollingUpdate:
    maxUnavailable: 25%  # Max pods that can be unavailable
    maxSurge: 25%        # Max extra pods during update
```

- **Process:** Gradually replace old pods with new ones
- **Benefit:** Zero downtime
- **Use case:** Most applications

2. Recreate

```yaml
strategy:
  type: Recreate
```

- **Process:** Kill all old pods, then create new ones
- **Benefit:** Simple, no resource overhead
- **Use case:** Applications that can't run multiple versions

StatefulSet

Purpose: For stateful applications that need:

- Stable network identities
- Ordered deployment and scaling
- Persistent storage

```yaml
apiVersion: apps/v1
kind: StatefulSet
metadata:
  name: database
spec:
  serviceName: database
  replicas: 3
  template:
    spec:
      containers:
      - name: postgres
        image: postgres:13
        volumeMounts:
        - name: data
          mountPath: /var/lib/postgresql/data
  volumeClaimTemplates:
  - metadata:
      name: data
    spec:
      accessModes: ["ReadWriteOnce"]
      resources:
        requests:
          storage: 10Gi
```

Key differences from Deployment:

- **Stable hostnames:** `database-0` , `database-1` , `database-2`
- **Ordered operations:** Pods created/deleted in sequence
- **Persistent storage:** Each pod gets its own persistent volume
- **Network identity:** Stable DNS names

Use cases:

- Databases (MySQL, PostgreSQL, MongoDB)
- Message queues (Kafka, RabbitMQ)
- Distributed systems (Elasticsearch, Cassandra)

DaemonSet

Purpose: Ensures a pod runs on every (or selected) node

```yaml
apiVersion: apps/v1
kind: DaemonSet
metadata:
  name: logging-agent
spec:
  selector:
    matchLabels:
      app: logging-agent
  template:
    metadata:
      labels:
        app: logging-agent
    spec:
      containers:
      - name: fluentd
        image: fluentd:latest
        volumeMounts:
        - name: host-logs
          mountPath: /var/log
          readOnly: true
      volumes:
      - name: host-logs
        hostPath:
          path: /var/log
```

Use cases:

- Log collection agents
- Monitoring agents (node-exporter)
- Network plugins
- Storage daemons

Job and CronJob

Job: Runs pods to completion for batch workloads

```yaml
apiVersion: batch/v1
kind: Job
metadata:
  name: data-migration
spec:
  completions: 1
  parallelism: 1
  template:
    spec:
      containers:
      - name: migrator
        image: migration-tool
        command: ["migrate", "--all"]
      restartPolicy: Never
```

CronJob: Runs jobs on a schedule

```yaml
apiVersion: batch/v1
kind: CronJob
metadata:
  name: backup
spec:
  schedule: "0 2 * * *"  # Daily at 2 AM
  jobTemplate:
    spec:
      template:
        spec:
          containers:
          - name: backup
            image: backup-tool
            command: ["backup", "database"]
          restartPolicy: OnFailure
```

Services & Networking

The Service Abstraction

Problem: Pods are ephemeral (die and get recreated with new IPs)
Solution: Services provide stable network endpoints

Service Types

1. ClusterIP (default)

```yaml
apiVersion: v1
kind: Service
metadata:
  name: backend
spec:
  type: ClusterIP
  selector:
    app: backend
  ports:
  - port: 80
    targetPort: 8080
```

- **Scope:** Internal cluster communication only
- **Use case:** Microservice communication
- **Access:** `backend.namespace.svc.cluster.local`

2. NodePort

```yaml
apiVersion: v1
kind: Service
metadata:
  name: frontend
spec:
  type: NodePort
  selector:
    app: frontend
  ports:
  - port: 80
    targetPort: 8080
    nodePort: 30080
```

- **Scope:** External access via node IP:port
- **Port range:** 30000-32767
- **Use case:** Development, simple external access

3. LoadBalancer

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

- **Scope:** Cloud provider creates external load balancer
- **Use case:** Production external access
- **Cost:** Each service = separate load balancer

4. ExternalName

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

- **Purpose:** Alias for external service
- **Use case:** Migrate external dependencies into cluster

How Services Work

Service discovery process:

1. **Label selector** finds matching pods
2. **Endpoints** object lists pod IPs
3. **kube-proxy** updates routing rules
4. **DNS** creates service records

Load balancing:

- **Algorithm:** Round-robin (default)
- **Session affinity:** Route same client to same pod
- **Topology aware:** Prefer local zone pods

Ingress

Purpose: HTTP/HTTPS routing to services

```yaml
apiVersion: networking.k8s.io/v1
kind: Ingress
metadata:
  name: app-ingress
  annotations:
    nginx.ingress.kubernetes.io/rewrite-target: /
spec:
  rules:
  - host: api.example.com
    http:
      paths:
      - path: /users
        pathType: Prefix
        backend:
          service:
            name: user-service
            port:
              number: 80
      - path: /orders
        pathType: Prefix
        backend:
          service:
            name: order-service
            port:
              number: 80
```

Benefits:

- **Cost effective:** One load balancer for multiple services
- **SSL termination:** Handle certificates centrally
- **Advanced routing:** Path-based, host-based, header-based
- **Middleware:** Authentication, rate limiting, caching

Ingress Controllers:

- **nginx-ingress:** Most popular, feature-rich
- **traefik:** Auto-discovery, Let's Encrypt integration
- **istio:** Service mesh with advanced features
- **kong:** API gateway features

Network Policies

Purpose: Control traffic between pods (firewall rules)

```yaml
apiVersion: networking.k8s.io/v1
kind: NetworkPolicy
metadata:
  name: deny-all
spec:
  podSelector: {}
  policyTypes:
  - Ingress
  - Egress
---
apiVersion: networking.k8s.io/v1
kind: NetworkPolicy
metadata:
  name: allow-frontend-to-backend
spec:
  podSelector:
    matchLabels:
      app: backend
  policyTypes:
  - Ingress
  ingress:
  - from:
    - podSelector:
        matchLabels:
          app: frontend
    ports:
    - protocol: TCP
      port: 8080
```

Security model:

- **Default:** All traffic allowed
- **With policies:** Default deny, explicit allow
- **Directions:** Ingress (incoming), Egress (outgoing)
- **Selectors:** Pod labels, namespace labels, IP blocks

Storage & Persistence

Volume Types

1. emptyDir

```yaml
volumes:
- name: cache
  emptyDir: {}
```

- **Lifetime:** Pod lifetime
- **Use case:** Temporary storage, cache

2. hostPath

```yaml
volumes:
- name: host-data
  hostPath:
    path: /data
    type: Directory
```

- **Lifetime:** Node lifetime
- **Use case:** Node-specific data, debugging

3. PersistentVolume

```yaml
apiVersion: v1
kind: PersistentVolume
metadata:
  name: pv-storage
spec:
  capacity:
    storage: 10Gi
  accessModes:
  - ReadWriteOnce
  persistentVolumeReclaimPolicy: Retain
  storageClassName: fast-ssd
  hostPath:
    path: /mnt/data
```

Persistent Volume Claims

Purpose: Request storage resources

```yaml
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
  name: storage-claim
spec:
  accessModes:
  - ReadWriteOnce
  resources:
    requests:
      storage: 5Gi
  storageClassName: fast-ssd
```

Access modes:

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

Binding process:

1. PVC requests storage with specific requirements
2. Kubernetes finds matching PV or creates one dynamically
3. PVC binds to PV
4. Pod mounts PVC as volume

Storage Classes

Purpose: Dynamic provisioning of storage

```yaml
apiVersion: storage.k8s.io/v1
kind: StorageClass
metadata:
  name: fast-ssd
provisioner: kubernetes.io/aws-ebs
parameters:
  type: gp3
  iops: "3000"
  fsType: ext4
reclaimPolicy: Delete
allowVolumeExpansion: true
```

Dynamic provisioning:

1. PVC references StorageClass
2. Storage provisioner creates actual storage
3. PV automatically created and bound to PVC

Configuration Management

ConfigMaps

Purpose: Store non-sensitive configuration data

```yaml
apiVersion: v1
kind: ConfigMap
metadata:
  name: app-config
data:
  database_url: "postgres://db:5432/app"
  log_level: "info"
  feature_flags: |
    feature_a: true
    feature_b: false
  nginx.conf: |
    server {
        listen 80;
        location / {
            proxy_pass http://backend;
        }
    }
```

Usage in pods:

Environment variables:

```yaml
containers:
- name: app
  envFrom:
  - configMapRef:
      name: app-config
  env:
  - name: LOG_LEVEL
    valueFrom:
      configMapKeyRef:
        name: app-config
        key: log_level
```

Volume mounts:

```yaml
  containers:
  - name: nginx
    volumeMounts:
    - name: config
      mountPath: /etc/nginx/nginx.conf
      subPath: nginx.conf
  volumes:
  - name: config
    configMap:
      name: app-config
```

Secrets

Purpose: Store sensitive data (passwords, tokens, keys)

```yaml
apiVersion: v1
kind: Secret
metadata:
  name: app-secrets
type: Opaque
data:
  username: YWRtaW4=  # base64 encoded
  password: MWYyZDFlMmU2N2Rm  # base64 encoded
stringData:
  api-key: "plain-text-key"  # automatically encoded
```

Secret types:

- **Opaque:** Arbitrary user data
- **kubernetes.io/service-account-token:** Service account tokens
- **kubernetes.io/dockercfg:** Docker registry credentials
- **kubernetes.io/tls:** TLS certificates

Usage:

```yaml
  containers:
  - name: app
    env:
    - name: DB_PASSWORD
      valueFrom:
        secretKeyRef:
          name: app-secrets
          key: password
    volumeMounts:
    - name: certs
      mountPath: /etc/ssl/certs
      readOnly: true
  volumes:
  - name: certs
    secret:
      secretName: tls-secret
```

Resource Quotas

Purpose: Limit resource consumption per namespace

```yaml
apiVersion: v1
kind: ResourceQuota
metadata:
  name: quota
  namespace: development
spec:
  hard:
    requests.cpu: "4"
    requests.memory: 8Gi
    limits.cpu: "8"
    limits.memory: 16Gi
    pods: "10"
    persistentvolumeclaims: "4"
```

Security Model

RBAC (Role-Based Access Control)

Components:

- **Subject:** Users, groups, service accounts
- **Verb:** Actions (get, list, create, update, delete)
- **Resource:** API objects (pods, services, deployments)

Role (namespace-scoped):

```yaml
apiVersion: rbac.authorization.k8s.io/v1
kind: Role
metadata:
  namespace: development
  name: pod-reader
rules:
- apiGroups: [""]
  resources: ["pods"]
  verbs: ["get", "watch", "list"]
```

ClusterRole (cluster-scoped):

```yaml
apiVersion: rbac.authorization.k8s.io/v1
kind: ClusterRole
metadata:
  name: node-reader
rules:
- apiGroups: [""]
  resources: ["nodes"]
  verbs: ["get", "list"]
```

RoleBinding:

```yaml
apiVersion: rbac.authorization.k8s.io/v1
kind: RoleBinding
metadata:
  name: read-pods
  namespace: development
subjects:
- kind: User
  name: jane
  apiGroup: rbac.authorization.k8s.io
roleRef:
  kind: Role
  name: pod-reader
  apiGroup: rbac.authorization.k8s.io
```

Service Accounts

Purpose: Identity for pods to access Kubernetes API

```yaml
apiVersion: v1
kind: ServiceAccount
metadata:
  name: my-service-account
  namespace: development
---
apiVersion: v1
kind: Pod
spec:
  serviceAccountName: my-service-account
  containers:
  - name: app
    image: my-app
```

Security Contexts

Pod-level security:

```yaml
spec:
  securityContext:
    runAsUser: 1000
    runAsGroup: 1000
    fsGroup: 1000
    seccompProfile:
      type: RuntimeDefault
```

Container-level security:

```yaml
containers:
- name: app
  securityContext:
    allowPrivilegeEscalation: false
    runAsNonRoot: true
    readOnlyRootFilesystem: true
    capabilities:
      drop:
      - ALL
      add:
      - NET_BIND_SERVICE
```

Pod Security Standards

Privileged: No restrictions
Baseline: Minimal restrictions, prevents known privilege escalations
Restricted: Heavily restricted, follows pod hardening best practices

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

Scaling & Resource Management

Resource Requests and Limits

Requests: Guaranteed resources for scheduling
Limits: Maximum resources a container can use

```yaml
containers:
- name: app
  resources:
    requests:
      memory: "256Mi"
      cpu: "250m"
    limits:
      memory: "512Mi"
      cpu: "500m"
```

CPU units:

- `1` = 1 CPU core
- `500m` = 0.5 CPU core
- `100m` = 0.1 CPU core

Memory units:

- `Ki` = Kibibytes (1024 bytes)
- `Mi` = Mebibytes (1024 Ki)
- `Gi` = Gibibytes (1024 Mi)

Quality of Service Classes

Guaranteed: Requests = limits for all containers

```yaml
resources:
  requests:
    memory: "256Mi"
    cpu: "250m"
  limits:
    memory: "256Mi"
    cpu: "250m"
```

Burstable: Has requests < limits

```yaml
resources:
  requests:
    memory: "128Mi"
  limits:
    memory: "256Mi"
```

BestEffort: No requests or limits specified

Eviction order: BestEffort → Burstable → Guaranteed

Horizontal Pod Autoscaler (HPA)

Purpose: Automatically scale pod replicas based on metrics

```yaml
apiVersion: autoscaling/v2
kind: HorizontalPodAutoscaler
metadata:
  name: app-hpa
spec:
  scaleTargetRef:
    apiVersion: apps/v1
    kind: Deployment
    name: app
  minReplicas: 2
  maxReplicas: 10
  metrics:
  - type: Resource
    resource:
      name: cpu
      target:
        type: Utilization
        averageUtilization: 70
  - type: Resource
    resource:
      name: memory
      target:
        type: Utilization
        averageUtilization: 80
  behavior:
    scaleUp:
      stabilizationWindowSeconds: 60
      policies:
      - type: Percent
        value: 50
        periodSeconds: 60
    scaleDown:
      stabilizationWindowSeconds: 300
      policies:
      - type: Percent
        value: 10
        periodSeconds: 60
```

Scaling algorithm:

```
desiredReplicas = ceil[currentReplicas * (currentMetricValue / desiredMetricValue)]

Example:
- Current replicas: 3
- Current CPU usage: 90%
- Target CPU usage: 70%
- Calculation: ceil[3 * (90/70)] = ceil[3.86] = 4 replicas
```

Vertical Pod Autoscaler (VPA)

Purpose: Automatically adjust CPU and memory requests/limits

```yaml
apiVersion: autoscaling.k8s.io/v1
kind: VerticalPodAutoscaler
metadata:
  name: app-vpa
spec:
  targetRef:
    apiVersion: apps/v1
    kind: Deployment
    name: app
  updatePolicy:
    updateMode: "Auto"  # Off, Initial, Auto
  resourcePolicy:
    containerPolicies:
    - containerName: app
      maxAllowed:
        cpu: 1
        memory: 2Gi
      minAllowed:
        cpu: 100m
        memory: 128Mi
```

Cluster Autoscaler

Purpose: Automatically adjust cluster size based on pod resource requirements

When it scales up:

- Pods are pending due to insufficient resources
- No suitable node available

When it scales down:

- Node utilization is low (< 50% by default)
- All pods can be rescheduled to other nodes

Advanced Concepts

Custom Resource Definitions (CRDs)

Purpose: Extend Kubernetes API with custom objects

```yaml
apiVersion: apiextensions.k8s.io/v1
kind: CustomResourceDefinition
metadata:
  name: databases.stable.example.com
spec:
  group: stable.example.com
  versions:
  - name: v1
    served: true
    storage: true
    schema:
      openAPIV3Schema:
        type: object
        properties:
          spec:
            type: object
            properties:
              size:
                type: string
                enum: ["small", "medium", "large"]
              version:
                type: string
  scope: Namespaced
  names:
    plural: databases
    singular: database
    kind: Database
```

Using the CRD:

```yaml
apiVersion: stable.example.com/v1
kind: Database
metadata:
  name: my-database
spec:
  size: medium
  version: "13.4"
```

Operators

Purpose: Automate complex application lifecycle management

Operator pattern:

1. **Custom Resource:** Desired state of application
2. **Controller:** Watches CR and manages application
3. **Domain knowledge:** Encoded operational procedures

Example - Database Operator:

- Creates database pods, services, persistent volumes
- Handles backups, restores, upgrades
- Monitors health and performs recovery

Service Mesh (Istio)

Purpose: Handle service-to-service communication, security, observability

Components:

- **Data plane:** Sidecar proxies (Envoy) in each pod
- **Control plane:** Manages proxy configuration

Features:

- **Traffic management:** Load balancing, circuit breaking, retries
- **Security:** mTLS, authorization policies
- **Observability:** Metrics, tracing, logging

Helm

Purpose: Package manager for Kubernetes

Chart structure:

```
mychart/
  Chart.yaml          # Chart metadata
  values.yaml         # Default configuration values
  templates/          # Kubernetes YAML templates
    deployment.yaml
    service.yaml
    ingress.yaml
```

Templating:

```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: {{ .Values.name }}
spec:
  replicas: {{ .Values.replicas }}
  template:
    spec:
      containers:
      - name: {{ .Values.name }}
        image: {{ .Values.image.repository }}:{{ .Values.image.tag }}
```

Production Considerations

High Availability

Control plane HA:

- Multiple master nodes (3 or 5)
- Load balancer in front of API servers
- etcd cluster with odd number of nodes

Application HA:

- Multiple replicas across zones
- Pod disruption budgets
- Readiness and liveness probes
- Circuit breakers and retries

Monitoring and Observability

The three pillars:

1. Metrics (Prometheus)

```yaml
# ServiceMonitor for Prometheus
apiVersion: monitoring.coreos.com/v1
kind: ServiceMonitor
metadata:
  name: app-metrics
spec:
  selector:
    matchLabels:
      app: my-app
  endpoints:
  - port: metrics
    interval: 30s
    path: /metrics
```

2. Logging (ELK Stack)

```yaml
# Fluentd DaemonSet for log collection
containers:
- name: fluentd
  image: fluentd:v1.14
  volumeMounts:
  - name: host-logs
    mountPath: /var/log
    readOnly: true
```

3. Tracing (Jaeger)

- Distributed tracing across microservices
- Request flow visualization
- Performance bottleneck identification

Backup and Disaster Recovery

etcd backup:

```
etcdctl snapshot save backup.db \
  --endpoints=https://127.0.0.1:2379 \
  --cacert=/etc/ssl/etcd/ca.crt \
  --cert=/etc/ssl/etcd/client.crt \
  --key=/etc/ssl/etcd/client.key
```

Application data backup:

- Volume snapshots
- Database dumps
- Cross-region replication

Security Best Practices

Network security:

- Network policies for microsegmentation
- Service mesh for mTLS
- Ingress controllers with WAF

Image security:

- Scan images for vulnerabilities
- Use minimal base images (distroless)
- Sign images with cosign

Runtime security:

- Read-only root filesystems
- Non-root users
- Security contexts and PSPs/PSAs

Performance Optimization

Resource optimization:

- Right-size requests and limits
- Use VPA for recommendations

- Monitor resource utilization

Network optimization:

- Keep related services in same zone
- Use topology-aware routing
- Optimize service mesh configuration

Storage optimization:

- Choose appropriate storage classes
- Use local storage for high IOPS
- Implement data lifecycle policies

Interview Questions & Scenarios

Basic Concepts

Q: What is the difference between a Pod and a Deployment?
A: A Pod is a single instance of an application, while a Deployment manages multiple Pod replicas and provides declarative updates, rolling deployments, and rollback capabilities.

Q: Explain the Kubernetes networking model.
A:

- Every pod gets a unique IP address
- Pods can communicate directly without NAT
- Services provide stable endpoints for pod groups
- Ingress handles external HTTP/HTTPS traffic routing

Q: What happens when you run `kubectl apply -f deployment.yaml`?
A:

1. kubectl sends YAML to API server
2. API server validates and stores in etcd
3. Deployment controller sees new deployment
4. Creates ReplicaSet with pod template
5. ReplicaSet controller creates pods
6. Scheduler assigns pods to nodes

7. kubelet starts containers on assigned nodes

Troubleshooting Scenarios

Q: A pod is stuck in Pending state. How do you troubleshoot?

A:

```
# Check pod status and events
kubectl describe pod <pod-name>

# Common causes:
# 1. Insufficient resources on nodes
kubectl top nodes

# 2. Image pull errors
kubectl describe pod <pod-name> | grep -i image

# 3. Storage issues
kubectl get pvc
kubectl describe pvc <pvc-name>

# 4. Scheduling constraints
kubectl describe pod <pod-name> | grep -i nodeaffinity
```

Q: Pods are running but service isn't working. What could be wrong?

A:

```
# Check service endpoints
kubectl get endpoints <service-name>

# If no endpoints, check:
# 1. Service selector matches pod labels
kubectl describe service <service-name>
kubectl get pods --show-labels

# 2. Pod readiness probe status
kubectl describe pod <pod-name>

# 3. Network policies blocking traffic
kubectl get networkpolicies
```

Q: Application is running out of memory and getting killed. How do you fix it?

A:

```
# Check resource usage
kubectl top pods

# Check events for OOMKilled
kubectl describe pod <pod-name>

# Solutions:
# 1. Increase memory limits
# 2. Use VPA for recommendations
# 3. Optimize application memory usage
# 4. Use HPA to scale horizontally
```

Scaling Scenarios

Q: Your application needs to handle 10x traffic during Black Friday. How do you prepare?

A:

1. **Horizontal Pod Autoscaler:**

   ```yaml
   spec:
     minReplicas: 10    # Pre-scale before event
     maxReplicas: 100   # Allow significant scaling
     metrics:
     - type: Resource
       resource:
         name: cpu
         target:
           averageUtilization: 60  # Scale earlier
   ```

2. **Cluster Autoscaler:** Ensure cluster can add nodes
3. **Load testing:** Verify scaling behavior
4. **Resource limits:** Ensure adequate quotas
5. **Monitoring:** Set up alerts for key metrics

Q: How do you perform a zero-downtime deployment?

A:

```yaml
# Rolling update strategy
strategy:
  type: RollingUpdate
  rollingUpdate:
    maxUnavailable: 0      # Keep all pods running
    maxSurge: 1           # Add one pod at a time

# Readiness probe to verify new pods
readinessProbe:
  httpGet:
    path: /health
    port: 8080
  initialDelaySeconds: 5
  periodSeconds: 5

# Pre-stop hook for graceful shutdown
lifecycle:
  preStop:
    exec:
      command: ["/bin/sh", "-c", "sleep 10"]
```

Security Scenarios

Q: How do you secure secrets in Kubernetes?

A:

1. **Use Secrets instead of ConfigMaps** for sensitive data
2. **Encrypt at rest:** Enable etcd encryption
3. **RBAC:** Limit access to secrets
4. **External secret management:** Use Vault, AWS Secrets Manager
5. **Rotation:** Implement secret rotation policies

Q: A pod needs to access AWS resources. How do you handle authentication securely?

A:

```yaml
# Option 1: IAM Roles for Service Accounts (IRSA)
apiVersion: v1
kind: ServiceAccount
metadata:
  name: aws-service-account
  annotations:
    eks.amazonaws.com/role-arn: arn:aws:iam::ACCOUNT:role/EKS_ROLE

# Option 2: Workload Identity (GKE)
# Option 3: External Secrets Operator
```

Advanced Scenarios

Q: You need to run a database in Kubernetes. What considerations are important?

A:

1. **StatefulSet** for stable identities and ordered deployment
2. **Persistent Volumes** for data persistence
3. **Resource guarantees** with requests = limits
4. **Backup strategy** with volume snapshots
5. **Monitoring** for database-specific metrics
6. **Security** with network policies and encryption

Q: How do you implement blue-green deployments in Kubernetes?

A:

```yaml
# Strategy 1: Two deployments + service selector
apiVersion: v1
kind: Service
metadata:
  name: app-service
spec:
  selector:
    app: myapp
    version: blue  # Switch to 'green' for deployment

---
# Blue deployment
apiVersion: apps/v1
kind: Deployment
metadata:
  name: app-blue
spec:
  replicas: 3
  selector:
    matchLabels:
      app: myapp
      version: blue

---
# Green deployment
apiVersion: apps/v1
kind: Deployment
metadata:
  name: app-green
spec:
  replicas: 3
  selector:
    matchLabels:
      app: myapp
      version: green
```

Performance Questions

Q: Your application has high latency. How do you investigate and optimize?

A:

1. **Application metrics:** Response times, error rates

2. **Resource utilization:** CPU, memory, network, disk
3. **Dependencies:** Database, external APIs, other services
4. **Network latency:** Inter-pod communication
5. **Optimization strategies:**
   - Horizontal scaling with HPA
   - Resource tuning
   - Caching layers
   - Database optimization
   - CDN for static content

Q: How do you ensure your cluster can handle expected load?
A:

1. **Capacity planning:**
   - Calculate resource requirements per user
   - Plan for peak load + buffer
   - Consider failure scenarios
2. **Load testing:**

```
# Use tools like k6, JMeter, or Artillery
k6 run --vus 1000 --duration 30m load-test.js
```

3. **Monitoring:**
   - Set up alerts for resource utilization
   - Monitor application SLIs/SLOs
   - Track business metrics

Real-World Problem Solving

Q: Your production cluster has a memory leak causing nodes to run out of memory. How do you handle this emergency?
A:
Immediate actions:

1. Scale out healthy nodes with cluster autoscaler
2. Identify problematic pods: `kubectl top pods --sort-by=memory`
3. Restart high-memory pods: `kubectl delete pod <pod-name>`
4. Add resource limits to prevent future issues

Long-term solutions:

1. Implement proper monitoring and alerting
2. Set up VPA to right-size resources
3. Use chaos engineering to test resilience
4. Implement pod disruption budgets

Q: A critical service is down. Walk me through your incident response.

A:

1. **Detection:** Monitoring alerts, user reports
2. **Assessment:**

   ```
   kubectl get pods -l app=critical-service
   kubectl describe pods -l app=critical-service
   kubectl logs -l app=critical-service --tail=100
   ```

3. **Mitigation:**
   - Scale up replicas if resource issue
   - Rollback if recent deployment caused issue
   - Route traffic to backup service
4. **Communication:** Update status page, notify stakeholders
5. **Resolution:** Fix root cause
6. **Post-mortem:** Document learnings, improve processes

Architecture Questions

Q: Design a microservices architecture on Kubernetes for an e-commerce platform.

A:

```yaml
# Services breakdown:
# - Frontend (React SPA)
# - User Service (authentication, profiles)
# - Product Service (catalog, search)
# - Order Service (cart, checkout, orders)
# - Payment Service (payment processing)
# - Inventory Service (stock management)
# - Notification Service (emails, SMS)

# Example architecture:
apiVersion: v1
kind: Namespace
metadata:
  name: ecommerce

---
# API Gateway (Ingress)
apiVersion: networking.k8s.io/v1
kind: Ingress
metadata:
  name: api-gateway
  namespace: ecommerce
spec:
  rules:
  - host: api.ecommerce.com
    http:
      paths:
      - path: /users
        pathType: Prefix
        backend:
          service:
            name: user-service
            port:
              number: 80
      - path: /products
        pathType: Prefix
        backend:
          service:
            name: product-service
            port:
              number: 80

---
```

```yaml
# User Service
apiVersion: apps/v1
kind: Deployment
metadata:
  name: user-service
  namespace: ecommerce
spec:
  replicas: 3
  selector:
    matchLabels:
      app: user-service
  template:
    metadata:
      labels:
        app: user-service
    spec:
      containers:
      - name: user-service
        image: user-service:v1.0
        ports:
        - containerPort: 8080
        env:
        - name: DB_HOST
          value: user-db
        - name: REDIS_HOST
          value: redis
```

Key considerations:

- **Service mesh** (Istio) for inter-service communication
- **Database per service** pattern with appropriate storage
- **Event-driven architecture** using message queues (RabbitMQ/Kafka)
- **Observability** with distributed tracing
- **Security** with network policies and mTLS
- **CI/CD** with GitOps (ArgoCD)

Cost Optimization

Q: Your Kubernetes costs are too high. How do you optimize them?

A:

1. **Right-sizing:**

- Use VPA recommendations
- Implement resource requests/limits
- Remove over-provisioned resources

2. **Scaling optimization:**
    - Use HPA for automatic scaling
    - Implement cluster autoscaler
    - Use spot instances for non-critical workloads
3. **Storage optimization:**
    - Use appropriate storage classes
    - Implement data lifecycle policies
    - Clean up unused PVCs
4. **Monitoring and governance:**
    - Implement resource quotas
    - Track costs per namespace/team
    - Use tools like KubeCost

Final Advice for Interviews

Key topics to master:

1. **Core concepts:** Pods, Services, Deployments, ConfigMaps, Secrets
2. **Networking:** How services work, ingress, network policies
3. **Storage:** PVs, PVCs, storage classes
4. **Scaling:** HPA, VPA, cluster autoscaler
5. **Security:** RBAC, security contexts, network policies
6. **Troubleshooting:** Common issues and debugging techniques

Hands-on experience matters:

- Deploy real applications
- Practice troubleshooting scenarios
- Understand kubectl commands deeply
- Know how to read and write YAML manifests

Architecture thinking:

- Understand when to use different Kubernetes resources
- Know trade-offs between different approaches
- Think about production concerns (HA, security, monitoring)

Remember: Kubernetes interviews often focus on practical problem-solving. Be prepared to explain not just what something does, but why you would use it and how it fits into a larger system design.