MASTERING KUBERNETES

A Complete Guide to Concepts, Architecture & Best Practices

From Fundamentals to Production-Ready Deployments

November 22, 2025

Contents

Chapter 1

Introduction to Kubernetes

1.1 What is Kubernetes?

Kubernetes, often abbreviated as 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 across clusters of machines.

Kubernetes provides a framework for running distributed systems resiliently, handling scaling and failover for applications, providing deployment patterns, and offering self-healing capabilities including auto-placement, auto-restart, auto-replication, and scaling based on resource usage.

Info

The name "Kubernetes" originates from Greek, meaning "helmsman" or "pilot." The abbreviation K8s comes from replacing the eight letters between "K" and "s" with the number 8.

1.2 Why Kubernetes?

Modern applications are increasingly deployed as microservices running in containers. Managing containers at scale requires sophisticated orchestration:

- **Automated Deployment**: Deploy containers across multiple hosts automatically
- **Service Discovery**: Automatically expose containers using DNS or IP addresses
- **Load Balancing**: Distribute network traffic to stabilize deployments
- **Self-Healing**: Restart failed containers and replace unhealthy ones
- **Horizontal Scaling**: Scale applications up or down based on demand
- **Rolling Updates**: Update applications with zero downtime
- **Secret Management**: Store and manage sensitive information securely

1.3 Key Terminology

Term	**Description**
k8slightblue Cluster	A set of nodes running containerized applications
Node	A physical or virtual machine in the cluster
k8slightblue Pod	The smallest deployable unit containing one or more containers
Service	An abstract way to expose an application running on pods
k8slightblue Namespace	Virtual clusters within a physical cluster
Deployment	Declarative updates for Pods and ReplicaSets

Table 1.1: Essential Kubernetes Terminology

Chapter 2

Kubernetes Architecture

Kubernetes follows a distributed architecture consisting of a **Control Plane** and one or more **Worker Nodes**. The Control Plane manages the cluster state and configuration, while Worker Nodes run the actual application workloads.

2.1 Cluster Overview

A Kubernetes cluster consists of two main layers:

1. **Control Plane**: The brain of the cluster that makes global decisions about scheduling, deployment, and responding to cluster events
2. **Worker Nodes (Data Plane)**: Execute application workloads by running pods

Info

For production reliability, multiple control plane nodes should be deployed to create a High Availability (HA) architecture. This ensures the cluster remains operational even if one or more components fail.

2.2 Control Plane Components

The Control Plane is responsible for maintaining the desired state of the cluster, orchestrating operations, and managing worker nodes.

2.2.1 kube-apiserver

The API Server is the central hub that exposes the Kubernetes API. It serves as the front end for the Kubernetes control plane and handles all REST API requests.

Key Responsibilities:

- Process and validate API requests
- Serve as the gateway for all cluster communication
- Authenticate and authorize requests
- Update objects in etcd

Listing 2.1: Accessing the API Server

```
# Check API server health
kubectl cluster-info

# View API resources
kubectl api-resources

# Direct API call
curl -k https://<api-server>:6443/api/v1/namespaces
```

2.2.2 etcd

etcd is a distributed, consistent key-value store that serves as Kubernetes' backing store for all cluster data. It stores the configuration, state, and metadata for all Kubernetes objects.

Data Stored in etcd:

- Cluster configuration and metadata
- Workloads: Pods, Deployments, ReplicaSets, StatefulSets
- Services and networking configurations
- Secrets and credentials
- Persistent volumes and storage classes

Warning

etcd is the single source of truth for the cluster. Always maintain proper backup strategies for etcd data. Loss of etcd data means loss of the entire cluster state.

2.2.3 kube-scheduler

The Scheduler watches for newly created Pods without assigned nodes and selects optimal nodes for them to run on.

Scheduling Factors:

- Resource requirements (CPU, memory)
- Hardware/software constraints
- Affinity and anti-affinity specifications
- Data locality
- Taints and tolerations

2.2.4 kube-controller-manager

The Controller Manager runs controller processes that regulate the state of the cluster. Each controller watches the shared state through the API server and makes changes to move the current state toward the desired state.

Key Controllers:

- **Node Controller**: Monitors node health and responds when nodes fail
- **Job Controller**: Creates Pods for one-off tasks

- **EndpointSlice Controller**: Links Services and Pods
- **ServiceAccount Controller**: Creates default ServiceAccounts
- **Deployment Controller**: Manages Deployment resources
- **ReplicaSet Controller**: Maintains the desired number of pod replicas

2.2.5 cloud-controller-manager

The Cloud Controller Manager integrates with cloud provider APIs, separating cloud-specific logic from core Kubernetes components. It handles:

- Node lifecycle management in the cloud
- Route configuration
- Service load balancer provisioning
- Volume management

2.3 Worker Node Components

Worker Nodes run the application workloads and are managed by the Control Plane.

2.3.1 kubelet

The kubelet is an agent that runs on each worker node, ensuring containers are running in Pods as expected.

Responsibilities:

- Register the node with the API server
- Watch for Pod specifications from the API server
- Mount volumes and download secrets
- Execute containers via the container runtime
- Report node and Pod status back to the control plane

2.3.2 kube-proxy

kube-proxy is a network proxy that runs on each node, implementing part of the Kubernetes Service concept.

Functions:

- Maintain network rules for Service IP routing
- Handle TCP, UDP, and SCTP stream forwarding
- Perform connection load balancing across Pods

2.3.3 Container Runtime

The container runtime is responsible for running containers. Kubernetes supports multiple runtimes through the Container Runtime Interface (CRI):

- **containerd**: Industry-standard container runtime
- **CRI-O**: Lightweight container runtime for Kubernetes
- **Docker Engine**: (via cri-dockerd adapter)

2.4 Add-on Components

2.4.1 DNS (CoreDNS)

CoreDNS provides DNS-based service discovery within the cluster, allowing Pods to discover Services by name.

2.4.2 Container Network Interface (CNI)

CNI plugins handle networking between Pods across nodes. Popular options include:

- **Cilium**: eBPF-based networking with advanced security
- **Calico**: Policy-based networking and routing
- **Flannel**: Simple overlay networking
- **Weave Net**: Simple, resilient multi-host networking

2.4.3 Metrics Server

Collects resource metrics from kubelets and exposes them via the Metrics API for use by Horizontal Pod Autoscaler and kubectl top commands.

Chapter 3

Core Kubernetes Concepts

3.1 Pods

A Pod is the smallest deployable unit in Kubernetes, representing a single instance of a running process. Pods can contain one or more containers that share storage and network resources.

Listing 3.1: Basic Pod Definition

```
apiVersion: v1
kind: Pod
metadata:
  name: nginx-pod
  labels:
    app: nginx
spec:
  containers:
  - name: nginx
    image: nginx:1.21
    ports:
    - containerPort: 80
    resources:
      requests:
        memory: "64Mi"
        cpu: "250m"
      limits:
        memory: "128Mi"
        cpu: "500m"
```

Best Practice

Avoid creating “naked” Pods directly. Always use higher-level controllers like Deployments or StatefulSets, which provide self-healing and scaling capabilities.

3.2 Deployments

Deployments provide declarative updates for Pods and ReplicaSets. They manage the desired state of applications, handling rollouts, rollbacks, and scaling.

Listing 3.2: Deployment Example

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: nginx-deployment
spec:
  replicas: 3
  selector:
    matchLabels:
      app: nginx
  template:
    metadata:
      labels:
        app: nginx
    spec:
      containers:
      - name: nginx
        image: nginx:1.21
        ports:
        - containerPort: 80
  strategy:
    type: RollingUpdate
    rollingUpdate:
      maxSurge: 1
      maxUnavailable: 0
```

3.3 Services

Services provide stable networking for a set of Pods, enabling service discovery and load balancing.

3.3.1 Service Types

Type	Description
k8slightblue ClusterIP	Internal cluster IP (default)
NodePort	Exposes on each node's IP at a static port
k8slightblue LoadBalancer	Provisions an external load balancer
ExternalName	Maps to a DNS name

Table 3.1: Kubernetes Service Types

Listing 3.3: Service Definition

```
apiVersion: v1
kind: Service
metadata:
  name: nginx-service
spec:
  selector:
    app: nginx
  ports:
  - protocol: TCP
    port: 80
    targetPort: 80
  type: ClusterIP
```

3.4 ConfigMaps and Secrets

3.4.1 ConfigMaps

ConfigMaps store non-confidential configuration data as key-value pairs, which can be consumed by Pods as environment variables or mounted files.

Listing 3.4: ConfigMap Example

```
apiVersion: v1
kind: ConfigMap
metadata:
  name: app-config
data:
  database_host: "db.example.com"
  log_level: "INFO"
```

3.4.2 Secrets

Secrets store sensitive information like passwords, tokens, and keys. They are base64-encoded and can be encrypted at rest.

Listing 3.5: Secret Example

```
apiVersion: v1
kind: Secret
metadata:
  name: db-credentials
type: Opaque
data:
  username: YWRtaW4=
  password: cGFzc3dvcmQxMjM=
```

Warning

Kubernetes Secrets are only base64-encoded by default, not encrypted. Enable encryption at rest and consider using external secret managers like HashiCorp Vault for production environments.

3.5 Namespaces

Namespaces provide logical isolation within a cluster, allowing multiple teams or projects to share cluster resources.

Listing 3.6: Namespace Commands

```
# Create namespaces
kubectl create namespace development
kubectl create namespace staging
kubectl create namespace production

# List namespaces
kubectl get namespaces

# Deploy to specific namespace
kubectl apply -f deployment.yaml -n production
```

3.6 Labels and Selectors

Labels are key-value pairs attached to objects for identification and organization. Selectors use labels to filter and select objects.

Listing 3.7: Labels Example

```
metadata:
  labels:
    app: web-frontend
    environment: production
    team: platform
    version: v1.2.0
```

3.7 Persistent Storage

3.7.1 Persistent Volumes (PV)

Persistent Volumes are cluster-wide storage resources provisioned by administrators.

3.7.2 Persistent Volume Claims (PVC)

PVCs are requests for storage by users, consuming PV resources.

Listing 3.8: PVC Example

```
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
  name: data-pvc
spec:
  accessModes:
    - ReadWriteOnce
  resources:
    requests:
      storage: 10Gi
  storageClassName: standard
```

Chapter 4

Workload Controllers

4.1 ReplicaSets

ReplicaSets ensure a specified number of identical Pod replicas are running at any time. They are typically managed by Deployments rather than created directly.

4.2 StatefulSets

StatefulSets manage stateful applications, providing guarantees about ordering and uniqueness of Pods. They are ideal for databases and applications requiring:

- Stable, unique network identifiers
- Stable, persistent storage
- Ordered, graceful deployment and scaling
- Ordered, automated rolling updates

Listing 4.1: StatefulSet Example

```yaml
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
        volumeMounts:
        - name: data
          mountPath: /var/lib/mysql
  volumeClaimTemplates:
```

```
  - metadata:
      name: data
    spec:
      accessModes: ["ReadWriteOnce"]
      resources:
        requests:
          storage: 10Gi
```

4.3 DaemonSets

DaemonSets ensure that a copy of a Pod runs on all (or selected) nodes. They are useful for:

- Log collection daemons
- Node monitoring agents
- Cluster storage daemons
- Network plugins

4.4 Jobs and CronJobs

4.4.1 Jobs

Jobs create Pods that run to completion, useful for batch processing tasks.

4.4.2 CronJobs

CronJobs create Jobs on a time-based schedule.

Listing 4.2: CronJob Example

```
apiVersion: batch/v1
kind: CronJob
metadata:
  name: backup-job
spec:
  schedule: "0 2 * * *"
  jobTemplate:
    spec:
      template:
        spec:
          containers:
          - name: backup
            image: backup-tool:latest
            command: ["/bin/sh", "-c", "backup.sh"]
          restartPolicy: OnFailure
```

4.5 Horizontal Pod Autoscaler

HPA automatically scales the number of Pod replicas based on observed metrics.

Listing 4.3: HPA Example

```
apiVersion: autoscaling/v2
kind: HorizontalPodAutoscaler
```

```yaml
metadata:
  name: web-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
```

Chapter 5

Networking in Kubernetes

5.1 Networking Model

Kubernetes implements a flat networking model with these fundamental requirements:

- All Pods can communicate with all other Pods without NAT
- Agents on nodes can communicate with all Pods
- Pods see their own IP as the same IP others see

5.2 Service Networking

5.2.1 ClusterIP

The default service type, accessible only within the cluster.

5.2.2 NodePort

Exposes the service on each node's IP at a static port (30000-32767 range).

5.2.3 LoadBalancer

Provisions an external load balancer (cloud provider dependent).

5.2.4 Ingress

Ingress manages external HTTP/HTTPS access to services, providing:

- URL-based routing
- SSL/TLS termination
- Name-based virtual hosting
- Load balancing

Listing 5.1: Ingress Example

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

5.3 Network Policies

Network Policies control traffic flow between Pods, providing microsegmentation.

Listing 5.2: Network Policy Example

```
apiVersion: networking.k8s.io/v1
kind: NetworkPolicy
metadata:
  name: db-policy
  namespace: production
spec:
  podSelector:
    matchLabels:
      app: database
  policyTypes:
  - Ingress
  - Egress
  ingress:
  - from:
    - podSelector:
        matchLabels:
          app: backend
    ports:
    - protocol: TCP
      port: 5432
  egress:
  - to:
    - podSelector:
        matchLabels:
          app: backend
```

Best Practice

Implement a "deny by default" approach: block all traffic and selectively allow only necessary communication paths.

Chapter 6

Security Best Practices

6.1 Security Overview

Kubernetes security follows the "4C" model:

1. **Cloud**: The underlying cloud/infrastructure security
2. **Cluster**: Kubernetes cluster-level security
3. **Container**: Container image and runtime security
4. **Code**: Application code security

6.2 Role-Based Access Control (RBAC)

RBAC regulates access to Kubernetes resources based on user roles.

6.2.1 RBAC Components

- **Role**: Defines permissions within a namespace
- **ClusterRole**: Defines cluster-wide permissions
- **RoleBinding**: Grants Role permissions to users
- **ClusterRoleBinding**: Grants ClusterRole permissions

Listing 6.1: RBAC Role and Binding

```
apiVersion: rbac.authorization.k8s.io/v1
kind: Role
metadata:
  namespace: development
  name: pod-reader
rules:
- apiGroups: [""]
  resources: ["pods"]
  verbs: ["get", "list", "watch"]
---
apiVersion: rbac.authorization.k8s.io/v1
kind: RoleBinding
metadata:
  name: read-pods
  namespace: development
subjects:
- kind: User
  name: developer
  apiGroup: rbac.authorization.k8s.io
roleRef:
  kind: Role
  name: pod-reader
  apiGroup: rbac.authorization.k8s.io
```

6.2.2 RBAC Best Practices

- Apply the principle of least privilege
- Use namespaced Roles over ClusterRoles when possible
- Avoid adding users to the system:masters group
- Regularly audit RBAC configurations
- Disable auto-mounting of service account tokens

6.3 Pod Security Standards

Pod Security Standards define three policy levels:

Level	Description
k8slightblue Privileged	Unrestricted (for system components)
Baseline	Minimally restrictive, prevents known escalations
k8slightblue Restricted	Highly restrictive, security best practices

Table 6.1: Pod Security Standard Levels

Listing 6.2: Pod Security Context

```
spec:
  securityContext:
    runAsNonRoot: true
    runAsUser: 1000
    fsGroup: 2000
  containers:
  - name: app
    securityContext:
      allowPrivilegeEscalation: false
      readOnlyRootFilesystem: true
      capabilities:
        drop:
          - ALL
```

6.4 Network Security

6.4.1 Implementing Network Policies

- Define default deny policies for all namespaces

- Allow only required ingress and egress traffic
- Use labels consistently for policy targeting
- Regularly review and update policies

6.4.2 API Server Security

- Restrict network access to the API server
- Enable audit logging
- Use TLS for all communications
- Implement admission controllers

6.5 Secrets Management

Best Practice

Best practices for managing secrets:

- Enable encryption at rest for Secrets
- Use external secret managers (Vault, AWS Secrets Manager)
- Rotate secrets regularly
- Limit secret access via RBAC
- Avoid storing secrets in container images or Git

6.6 Image Security

- Use minimal base images (distroless, Alpine)
- Scan images for vulnerabilities regularly
- Sign and verify images with tools like cosign
- Use private registries with authentication
- Implement image pull policies (Always for production)
- Pin image versions, avoid using ‘latest‘ tag

Chapter 7

Monitoring and Observability

7.1 The Three Pillars of Observability

7.1.1 Metrics

Numerical data points collected over time, ideal for alerting and dashboards.

Key Metrics to Monitor:

- **Node Level**: CPU, memory, disk, network usage
- **Pod Level**: Resource usage, restart counts, status
- **Application Level**: Request rates, latency, error rates
- **Cluster Level**: API server latency, etcd health

7.1.2 Logs

Timestamped records of discrete events.

Logging Best Practices:

- Use structured logging (JSON format)
- Include correlation IDs for request tracing
- Aggregate logs centrally (EFK/ELK stack, Loki)
- Implement log rotation and retention policies

7.1.3 Traces

Distributed tracing tracks requests across services.

7.2 Monitoring Stack

7.2.1 Prometheus

Prometheus is the de facto standard for Kubernetes monitoring, providing metric collection, storage, and alerting.

Listing 7.1: ServiceMonitor for Prometheus

```
apiVersion: monitoring.coreos.com/v1
kind: ServiceMonitor
```

```
metadata:
  name: app-monitor
spec:
  selector:
    matchLabels:
      app: web-app
  endpoints:
  - port: metrics
    interval: 30s
    path: /metrics
```

7.2.2 Grafana

Grafana provides visualization dashboards for metrics from Prometheus and other data sources.

7.2.3 Alertmanager

Alertmanager handles alerts from Prometheus, managing deduplication, grouping, and routing to notification channels.

Listing 7.2: PrometheusRule Alert

```
apiVersion: monitoring.coreos.com/v1
kind: PrometheusRule
metadata:
  name: pod-alerts
spec:
  groups:
  - name: pod.rules
    rules:
    - alert: PodCrashLooping
      expr: rate(kube_pod_container_status_restarts_total[15m]) > 0
      for: 5m
      labels:
        severity: warning
      annotations:
        summary: "Pod {{ $labels.pod }} is crash looping"
```

7.3 Logging Architecture

Info

Common logging patterns in Kubernetes:

- **Node-level logging agent**: DaemonSet collecting logs from all Pods
- **Sidecar container**: Dedicated logging container per Pod
- **Direct push**: Applications push logs directly to backend

7.4 Health Checks

Kubernetes provides three types of probes:

Listing 7.3: Health Probes Configuration

```yaml
spec:
  containers:
  - name: app
    livenessProbe:
      httpGet:
        path: /healthz
        port: 8080
      initialDelaySeconds: 15
      periodSeconds: 10
      failureThreshold: 3
    readinessProbe:
      httpGet:
        path: /ready
        port: 8080
      initialDelaySeconds: 5
      periodSeconds: 5
    startupProbe:
      httpGet:
        path: /startup
        port: 8080
      failureThreshold: 30
      periodSeconds: 10
```

Chapter 8

CI/CD and GitOps

8.1 Continuous Integration/Continuous Deployment

8.1.1 CI/CD Pipeline Stages

1. **Build**: Compile code, run unit tests
2. **Test**: Integration tests, security scans
3. **Package**: Build container images
4. **Publish**: Push images to registry
5. **Deploy**: Update Kubernetes manifests
6. **Verify**: Run smoke tests, monitor rollout

8.2 GitOps Principles

GitOps is an operational framework that uses Git as the single source of truth for declarative infrastructure and applications.

Core Principles:

- Declarative configuration stored in Git
- Automated synchronization to cluster
- Pull-based deployment model
- Continuous reconciliation

8.2.1 GitOps Tools

Tool	**Description**
k8slightblue Argo CD	Declarative GitOps CD for Kubernetes
Flux	GitOps toolkit for Kubernetes
k8slightblue Jenkins X	CI/CD solution for cloud-native apps
Tekton	Kubernetes-native CI/CD pipelines

Table 8.1: Popular GitOps and CI/CD Tools

8.3 Deployment Strategies

8.3.1 Rolling Update

Gradually replaces old Pods with new ones.

8.3.2 Blue-Green Deployment

Maintains two identical environments, switching traffic instantly.

8.3.3 Canary Deployment

Routes a small percentage of traffic to new version before full rollout.

Listing 8.1: Canary Deployment with Argo Rollouts

```yaml
apiVersion: argoproj.io/v1alpha1
kind: Rollout
metadata:
  name: web-app
spec:
  replicas: 10
  strategy:
    canary:
      steps:
      - setWeight: 10
      - pause: {duration: 5m}
      - setWeight: 30
      - pause: {duration: 5m}
      - setWeight: 50
      - pause: {duration: 5m}
  selector:
    matchLabels:
      app: web-app
  template:
    metadata:
      labels:
        app: web-app
    spec:
      containers:
      - name: web
        image: web-app:v2
```

Chapter 9

Production Best Practices

9.1 Resource Management

9.1.1 Resource Requests and Limits

Always define resource requests and limits for containers.

Listing 9.1: Resource Configuration

```yaml
resources:
  requests:
    memory: "256Mi"
    cpu: "250m"
  limits:
    memory: "512Mi"
    cpu: "500m"
```

Best Practice

Guidelines for Resource Configuration:

- Set requests based on typical usage
- Set limits based on maximum acceptable usage
- Use Vertical Pod Autoscaler (VPA) for recommendations
- Monitor actual usage and adjust accordingly

9.1.2 Resource Quotas

Limit resource consumption per namespace.

Listing 9.2: ResourceQuota Example

```yaml
apiVersion: v1
kind: ResourceQuota
metadata:
  name: compute-quota
  namespace: development
spec:
  hard:
    requests.cpu: "10"
    requests.memory: 20Gi
    limits.cpu: "20"
    limits.memory: 40Gi
    pods: "50"
```

9.1.3 Limit Ranges

Set default and maximum resource limits for containers.

Listing 9.3: LimitRange Example

```
apiVersion: v1
kind: LimitRange
metadata:
  name: default-limits
spec:
  limits:
  - default:
      cpu: "500m"
      memory: "512Mi"
    defaultRequest:
      cpu: "100m"
      memory: "128Mi"
    type: Container
```

9.2 High Availability

9.2.1 Pod Disruption Budgets

Ensure minimum availability during voluntary disruptions.

Listing 9.4: PodDisruptionBudget

```
apiVersion: policy/v1
kind: PodDisruptionBudget
metadata:
  name: web-pdb
spec:
  minAvailable: 2
  selector:
    matchLabels:
      app: web-app
```

9.2.2 Pod Anti-Affinity

Spread Pods across nodes for resilience.

Listing 9.5: Pod Anti-Affinity

```
spec:
  affinity:
    podAntiAffinity:
      requiredDuringSchedulingIgnoredDuringExecution:
      - labelSelector:
          matchExpressions:
          - key: app
            operator: In
            values:
            - web-app
        topologyKey: kubernetes.io/hostname
```

9.2.3 Topology Spread Constraints

Distribute Pods evenly across failure domains.

Listing 9.6: Topology Spread

```
spec:
  topologySpreadConstraints:
  - maxSkew: 1
    topologyKey: topology.kubernetes.io/zone
    whenUnsatisfiable: DoNotSchedule
    labelSelector:
      matchLabels:
        app: web-app
```

9.3 Multi-Cluster Strategies

- **Active-Active**: Multiple clusters serving traffic
- **Active-Passive**: Standby cluster for failover
- **Federation**: Unified management across clusters

9.4 Backup and Disaster Recovery

9.4.1 etcd Backup

Listing 9.7: etcd Backup Command

```
# Create etcd snapshot
ETCDCTL_API=3 etcdctl snapshot save backup.db \
  --endpoints=https://127.0.0.1:2379 \
  --cacert=/etc/kubernetes/pki/etcd/ca.crt \
  --cert=/etc/kubernetes/pki/etcd/server.crt \
  --key=/etc/kubernetes/pki/etcd/server.key
```

9.4.2 Velero for Cluster Backup

Velero provides backup and restore capabilities for Kubernetes clusters.

Listing 9.8: Velero Commands

```
# Create backup
velero backup create full-backup --include-namespaces '*'

# Restore from backup
velero restore create --from-backup full-backup
```

9.5 Cost Optimization

- Right-size resource requests and limits
- Use Cluster Autoscaler for dynamic node scaling
- Implement Pod Priority and Preemption

- Leverage spot/preemptible instances for non-critical workloads
- Clean up unused resources regularly
- Use namespace resource quotas

Chapter 10

Troubleshooting Guide

10.1 Common Debugging Commands

Listing 10.1: Essential kubectl Commands

```
# Cluster information
kubectl cluster-info
kubectl get nodes -o wide
kubectl top nodes

# Pod troubleshooting
kubectl get pods -A
kubectl describe pod <pod-name>
kubectl logs <pod-name> [-c container] [--previous]
kubectl exec -it <pod-name> -- /bin/sh

# Events and debugging
kubectl get events --sort-by=.metadata.creationTimestamp
kubectl get events -w  # Watch events

# Resource inspection
kubectl get all -n <namespace>
kubectl api-resources
kubectl explain <resource>
```

10.2 Common Issues and Solutions

10.2.1 Pod Stuck in Pending

- **Cause**: Insufficient resources, node selector mismatch, PVC not bound
- **Debug**: `kubectl describe pod <name>`
- **Solution**: Check events, verify resource availability, fix selectors

10.2.2 Pod in CrashLoopBackOff

- **Cause**: Application error, misconfiguration, failed probes
- **Debug**: `kubectl logs <pod> -previous`
- **Solution**: Fix application errors, adjust probe settings

10.2.3 ImagePullBackOff

- **Cause**: Wrong image name, missing credentials, registry issues
- **Debug**: Check image name, verify imagePullSecrets
- **Solution**: Correct image reference, configure registry auth

10.2.4 Service Not Accessible

- **Cause**: Selector mismatch, wrong ports, network policies
- **Debug**: `kubectl get endpoints <service>`
- **Solution**: Verify selectors match Pod labels, check ports

10.3 Debugging Tools

Listing 10.2: Advanced Debugging

```
# Ephemeral debug containers
kubectl debug -it <pod> --image=busybox --target=<container>

# Network debugging
kubectl run netshoot --rm -it --image=nicolaka/netshoot -- /bin/bash

# DNS debugging
kubectl run dnsutils --rm -it --image=gcr.io/kubernetes-e2e-test-
    images/dnsutils -- nslookup kubernetes

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

Chapter 11

Kubernetes Ecosystem

11.1 Helm - Package Manager

Helm simplifies Kubernetes application deployment through charts.

Listing 11.1: Helm Commands

```
# Add repository
helm repo add bitnami https://charts.bitnami.com/bitnami

# Search charts
helm search repo nginx

# Install chart
helm install my-release bitnami/nginx

# Upgrade release
helm upgrade my-release bitnami/nginx --set replicaCount=3

# List releases
helm list
```

11.2 Service Mesh

Service meshes provide advanced traffic management, security, and observability.

Service Mesh	**Key Features**
k8slightblue Istio	Full-featured, traffic management, security
Linkerd	Lightweight, simple, ultra-fast
k8slightblue Cilium	eBPF-based, high performance
Consul Connect	HashiCorp ecosystem integration

Table 11.1: Popular Service Mesh Solutions

11.3 Operators

Operators extend Kubernetes to automate complex application management.

Info

Operators encode operational knowledge into software, automating tasks like:

- Application deployment and upgrades
- Backup and restore operations
- Scaling and self-healing
- Configuration management

11.4 Managed Kubernetes Services

Provider	**Service**
k8slightblue Amazon Web Services	Elastic Kubernetes Service (EKS)
Google Cloud	Google Kubernetes Engine (GKE)
k8slightblue Microsoft Azure	Azure Kubernetes Service (AKS)
DigitalOcean	DigitalOcean Kubernetes (DOKS)

Table 11.2: Major Managed Kubernetes Offerings

Chapter 12

Kubernetes Mastery Roadmap

12.1 Learning Path

12.1.1 Beginner Level

1. Understand containers and Docker basics
2. Learn Kubernetes architecture and components
3. Master kubectl commands
4. Deploy simple applications with Pods and Deployments
5. Understand Services and basic networking

12.1.2 Intermediate Level

1. Configure persistent storage (PV/PVC)
2. Implement ConfigMaps and Secrets
3. Set up Ingress controllers
4. Understand RBAC and security contexts
5. Work with StatefulSets and DaemonSets
6. Implement health checks and resource management

12.1.3 Advanced Level

1. Design multi-cluster architectures
2. Implement GitOps workflows
3. Configure service mesh (Istio/Linkerd)
4. Build custom operators
5. Optimize performance and costs
6. Implement comprehensive monitoring and alerting

Certification	**Focus**
k8slightblue CKA	Certified Kubernetes Administrator
CKAD	Certified Kubernetes Application Developer
k8slightblue CKS	Certified Kubernetes Security Specialist

Table 12.1: CNCF Kubernetes Certifications

12.2 Certifications

12.3 Essential Resources

- **Official Documentation**: kubernetes.io/docs
- **Kubernetes The Hard Way**: Deep-dive setup guide
- **CNCF Landscape**: Cloud-native ecosystem overview
- **KillerCoda/Killershell**: Interactive labs
- **Kubernetes Slack**: Community support

Appendix A

Quick Reference

A.1 Essential kubectl Commands

Listing A.1: kubectl Cheat Sheet

```
# Context and configuration
kubectl config get-contexts
kubectl config use-context <context>
kubectl config set-context --current --namespace=<ns>

# Resource management
kubectl apply -f <file.yaml>
kubectl delete -f <file.yaml>
kubectl get <resource> -o yaml
kubectl edit <resource> <name>
kubectl patch <resource> <name> -p '<patch>'

# Scaling
kubectl scale deployment <name> --replicas=5
kubectl autoscale deployment <name> --min=2 --max=10

# Rollouts
kubectl rollout status deployment/<name>
kubectl rollout history deployment/<name>
kubectl rollout undo deployment/<name>
kubectl rollout restart deployment/<name>

# Debugging
kubectl describe <resource> <name>
kubectl logs <pod> [-f] [--previous]
kubectl exec -it <pod> -- <command>
kubectl port-forward <pod> <local>:<remote>
kubectl cp <pod>:<path> <local-path>
```

A.2 YAML Templates

Listing A.2: Production-Ready Deployment Template

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: app-name
```

```yaml
    labels:
      app: app-name
  spec:
    replicas: 3
    selector:
      matchLabels:
        app: app-name
    strategy:
      type: RollingUpdate
      rollingUpdate:
        maxSurge: 1
        maxUnavailable: 0
    template:
      metadata:
        labels:
          app: app-name
      spec:
        securityContext:
          runAsNonRoot: true
          runAsUser: 1000
        containers:
        - name: app
          image: app:v1.0.0
          ports:
          - containerPort: 8080
          resources:
            requests:
              cpu: "100m"
              memory: "128Mi"
            limits:
              cpu: "500m"
              memory: "512Mi"
          livenessProbe:
            httpGet:
              path: /healthz
              port: 8080
            initialDelaySeconds: 15
            periodSeconds: 10
          readinessProbe:
            httpGet:
              path: /ready
              port: 8080
            initialDelaySeconds: 5
            periodSeconds: 5
          securityContext:
            allowPrivilegeEscalation: false
            readOnlyRootFilesystem: true
        affinity:
          podAntiAffinity:
            preferredDuringSchedulingIgnoredDuringExecution:
            - weight: 100
              podAffinityTerm:
                labelSelector:
                  matchLabels:
                    app: app-name
                topologyKey: kubernetes.io/hostname
```

Happy Orchestrating!

kubernetes.io