

Table of Contents

Definitions & Core Concepts	20
Container Runtime Fundamentals.....	20
What is a Container Runtime?	20
Docker vs Containerd	20
CLI Tools Comparison	20
Key Commands	20
Pod Fundamentals	21
What is a Pod?	21
Pod Characteristics	21
Multi-Container Pod Patterns	21
Pod Creation	21
Controllers and Workload Management	21
What is a Controller?	21
ReplicationController vs ReplicaSet	21
What is a Deployment?	22
Deployment Strategies.....	22
Key Commands	22
RollingUpdate Strategy Parameters	24
Namespaces and Resource Isolation	24
What is a Namespace?	24
Default Namespaces	24
DNS in Namespaces	25
Resource Quotas	25
LimitRange	25

Services and Networking	25
What is a Service?	25
Service Types	26
Service Port Definitions	26
Service Example	26
What is a Node IP?	26
Configuration	Management
.....	27
What are ConfigMaps?	27
What are Secrets?	27
Using ConfigMaps and Secrets	27
Kubernetes Control Plane Components	28
What is etcd?	28
What is the API Server?	28
What is the Scheduler?	28
What is the Controller Manager?	28
What is Kubelet?	29
What is Kube-proxy?	29
Resource Management	29
What are Resource Requests?	29
What are Resource Limits?	29
Resource Units	29
Best Practices	30
Declarative vs Imperative.....	30
Essential kubectl Commands	30
Kubernetes Scheduling	30
DaemonSets	30
What is a DaemonSet?	30
DaemonSet Use Cases.....	30
DaemonSet vs Deployment	31
DaemonSet Definition	31

DaemonSet Scheduling Evolution	31
DaemonSet Commands	31
Static Pods.....	32
What are Static Pods?	32
Key Characteristics	32
Static Pod Configuration	32
Static Pod Use Cases.....	32
Static Pod Lifecycle	33
Static Pod Example	33
Static Pods vs DaemonSets	33
Viewing Static Pods	33
Taints and Tolerations	34
What are Taints and Tolerations?	34
Taint Effects	34
Applying Taints	34
Pod Tolerations.....	34
Master Node Taints	34
Toleration Operators	34
Node Selectors and Node Affinity	35
What is Node Selection?	35
Node Selectors (Simple Method)	35
Node Affinity (Advanced Method)	35
Node Affinity Types.....	36
Manual Scheduling	37
What is Manual Scheduling?	37
When Manual Scheduling is Needed	37
Manual Scheduling Methods	37
Multiple Schedulers	38
What are Multiple Schedulers?	38
Use Cases for Multiple Schedulers	38

Deploying Custom Scheduler	38
Using Custom Scheduler	39
Monitoring Multiple Schedulers	39
Network Policies	39
What are Network Policies?	39
Network Policy Components	39
Basic Network Policy Example	39
Network Policy Rules	40
Ingress	40
What is Ingress?.....	40
Ingress Components	41
Ingress vs Services	41
Ingress Controller Deployment	41
Ingress Resource Examples	42
Key Commands	
.....	43
Static Pods	43
Taints and Tolerations	43
Node Affinity	43
DaemonSets	43
Multiple Schedulers	44
Network Policies	44
Ingress	44
Logging and Monitoring	44
Container Logging	44
What is Container Logging?	44
Docker vs Kubernetes Logging	45
Log Access Methods	45
Pod Definition for Logging	45
Log Storage Locations	46
Log Limitations	46

Kubernetes Monitoring	46
What is Monitoring in Kubernetes?.....	46
Key Metrics to Monitor	46
Monitoring Solutions	47
Metrics Server	47
What is Metrics Server?	47
Metrics Server vs Heapster	47
Metrics Server Architecture	47
Metrics Server Installation	47
Using Metrics Server	48
Metrics Server Data Storage	48
Advanced Monitoring Concepts	49
Resource Requests vs Usage	49
Horizontal Pod Autoscaler (HPA) Integration	49
Custom Metrics	49
Logging Best Practices	49
Application Logging	49
Container Logging Standards	49
Log Management	49
Monitoring Best Practices	50
Metrics Collection	50
Alerting Strategy	50
Dashboard Design	50
Troubleshooting Common Issues	50
Metrics Server Issues	50
Logging Issues	50
Performance Troubleshooting	51
Version Compatibility Notes	51
Kubernetes Version Support	51
Migration Considerations	51

Key Commands Summary	51
Logging Commands	51
Monitoring Commands	51
Troubleshooting Commands	52
Application Lifecycle Management	52
Rolling Updates and Rollbacks	52
What is a Rollout?	52
Rollout Revisions	52
Deployment Strategies.....	52
Rolling Update Process	52
Rollout Commands	52
Example Rollout Output	53
Rollback Operations	53
Deployment Definition	53
Commands and Arguments	54
Docker Container Behavior	54
Docker Command Examples.....	54
Dockerfile Instructions	54
Kubernetes Commands and Arguments	55
Environment Variables.....	55
What are Environment Variables?	55
Flask Application Example	55
Setting Environment Variables	55
Environment Variable Value Types	56
ConfigMaps	56
What are ConfigMaps?	56
ConfigMap Use Cases	56
Creating ConfigMaps	57
Viewing ConfigMaps	57
Using ConfigMaps in Pods	57

Secrets	58
What are Secrets?	58
Secret vs ConfigMap	58
Creating Secrets	58
Base64 Encoding/Decoding	59
Viewing Secrets	59
Using Secrets in Pods	59
Secrets as Volumes	59
Multi-Container Pods	60
What are Multi-Container Pods?	60
Multi-Container Patterns	60
Shared Resources in Multi-Container Pods.....	60
Multi-Container Pod Definition	61
Use Cases for Multi-Container Pods	61
Application Lifecycle Commands Summary	61
Rolling Updates and Rollbacks	61
ConfigMaps and Secrets	61
Pod Management.....	61
Horizontal Pod Autoscaler (HPA) and Vertical Pod Autoscaler (VPA).....	62
Horizontal Pod Autoscaler (HPA)	62
What is HPA?	62
Key Concepts	62
HPA Architecture	62
HPA Prerequisites	62
1. Metrics Server.....	62
2. Resource Requests	63
Creating	
HPA.....	63
1. Imperative Creation	63
2. Declarative Creation (CPU-based)	63
3. Memory-based HPA	64

4. Multi-metric HPA.....	64
HPA Behavior Configuration	65
Scaling Policies	65
Behavior Parameters	66
Custom Metrics HPA	66
Prerequisites for Custom Metrics	66
Custom Metrics HPA Example	66
Object Metrics	67
Vertical Pod Autoscaler (VPA)	68
What is VPA?	68
VPA vs HPA	68
VPA Components	68
Installing VPA	68
VPA Installation	68
Check VPA Components	68
VPA Modes	68
1. Off Mode (Recommendation Only)	68
2. Initial Mode (Set on Pod Creation)	69
3. Auto Mode (Automatic Updates)	69
VPA Resource Policies	69
Comprehensive Resource Policy	69
Resource Policy Options	70
VPA Recommendations	70
Viewing VPA Recommendations	70
Sample VPA Output	70
HPA + VPA Combination	71
Safe Combination (Different Metrics)	71
Avoiding Conflicts	71

Monitoring and Observability	72
HPA Monitoring	72
VPA Monitoring	72
Metrics to Monitor	72
Troubleshooting	72
Common HPA Issues	72
Common VPA Issues	73
Real-World Examples.....	74
E-commerce Application Scaling	74
Batch Processing Application	75
CKA Exam Scenarios	76
Scenario 1: Create HPA for Deployment	76
Scenario 2: Multi-metric HPA with Custom Behavior	76
Scenario 3: VPA for Resource Optimization.....	77
Best Practices	77
HPA Best Practices	77
VPA Best Practices	78
General Autoscaling Best Practices	78
Commands Summary	78
HPA Commands	78
VPA Commands	78
Metrics and Debugging	79
Key Points for CKA Exam	79
Must Know Concepts	79
Common Exam Tasks	79
Critical Configuration	79
Security	80
Security Primitives	80
What is Kubernetes Security?	80
Core Security Components	80

Authentication Mechanisms	80
What is Authentication?	80
Authentication Methods	80
API Server Configuration	81
TLS Certificates	81
What are TLS Certificates?	81
Certificate Types.....	81
Certificate Generation Process	81
Certificate Requirements	82
Certificate Inspection	82
Authorization	82
What is Authorization?	82
Authorization Modes	82
API Server Authorization Configuration	82
Role-Based Access Control (RBAC)	83
What is RBAC?	83
RBAC Components	83
RBAC Verbs	84
Resource Names (Specific Resource Access)	84
RBAC Management Commands	84
KubeConfig Files.....	84
What is KubeConfig?	84
KubeConfig Structure	85
Certificate Data Encoding	85
What are Contexts?	85
Context Components	86
Context Structure	86
Context Management Commands	86
Creating and Modifying Contexts	86
Context Examples	87

Context Best Practices.....	87
Context Switching Workflows	87
Advanced Context Operations	88
Multi-Cluster Context Setup	88
Context Security Considerations	89
Troubleshooting Context Issues	89
Image Security	90
What is Image Security?	90
Private Registry Authentication	90
Using Image Pull Secrets	90
Image Security Best Practices	90
Security Contexts	90
What are Security Contexts?	90
Pod Security Context	90
Security Context Options	91
Network Policies	91
What are Network Policies?	91
Default Behavior	91
Network Policy Components	91
Basic Network Policy Example	91
Network Policy Selectors	92
Deny All Policy	93
CNI Plugin Requirements	93
ConfigMaps and Secrets Security	93
What are ConfigMaps?	93
What are Secrets?	93
Security Differences	93
Secret Types	93
Creating Secrets	94
Using Secrets in Pods	94

ConfigMap Security Best Practices	94
Secret Security Best Practices	95
Certificate API	95
What is the Certificates API?	95
Certificate Signing Request (CSR)	95
CSR Workflow.....	95
Controller Manager Configuration	96
Security Best Practices	96
Cluster Security	96
Authentication Security	96
Authorization Security	96
Secret Management	96
Troubleshooting Security Issues	96
Common Authentication Problems	96
Common Authorization Problems	97
Network Policy Debugging	97
Key Commands Summary	97
Authentication & Certificates	97
RBAC Management	97
Secrets and ConfigMaps	97
Network Policies	98
Service Accounts	98
What are Service Accounts?	98
Key Characteristics	98
Default Service Account	98
Automatic Behavior	98
Default Service Account Properties	98
Default Token Structure	98
Creating Service Accounts	99
Imperative Creation	99

Declarative Creation	99
Service Account with Secrets	99
Service Account Tokens	99
What are Service Account Tokens?	99
Token Types	99
Manual Token Creation (Kubernetes 1.24+)	100
Using Service Accounts in Pods	101
Specifying Service Account	101
Deployment with Service Account	101
Disabling Automatic Token Mounting.....	101
Service Accounts and RBAC	102
What is Service Account RBAC?	102
Creating Role for Service Account	102
Binding Service Account to Role	102
ClusterRole for Service Account	103
Service Account Subject Format	103
Advanced Service Account Usage	103
Multiple Service Accounts Example	103
Service Account with Image Pull Secrets	104
Service Account Labels and Annotations	104
Service Account Security Best Practices	105
Principle of Least Privilege	105
Namespace Isolation	105
Disable Default Service Account	105
Token Security	106
Service Account Troubleshooting	106
Common Issues and Solutions	106
Debugging Commands	106
CKA Exam Scenarios	107
Scenario 1: Create Service Account and Bind to Role	107

Scenario 2: Fix Pod Service Account Issue	107
Scenario 3: Security Hardening	107
Service Account Commands Summary	108
Basic Operations	108
Token Operations	108
RBAC Integration	108
Debugging	109
Key Points for CKA Exam	109
Must Know Concepts	109
Common Exam Tasks	109
Important File Paths	109
Critical Commands	109
Storage	110
Volumes	110
What are Volumes in Kubernetes?.....	110
Volume Purpose and Benefits	110
Container Storage Behavior.....	110
Volume Types	110
What are Volume Types?	110
Host-Based Volume Types	110
Network-Based Volume Types	111
Distributed Storage Volume Types	111
Volume Mounting Process.....	112
Volume Challenges in Large Deployments	112
Multi-User Environment Problems	112
The Need for Abstraction.....	112
Persistent Volumes (PVs)	112
What are Persistent Volumes?.....	112
PV Benefits	112
Persistent Volume Definition	112

PV Specification Components	113
Creating and Viewing PVs	113
PV Status States	113
Persistent Volume Claims (PVCs)	114
What are Persistent Volume Claims?	114
PVC Purpose	114
PVC Definition	114
Creating and Managing PVCs	114
PV and PVC Binding Process	115
What is PV-PVC Binding?	115
Binding Criteria	115
Binding Examples	116
One-to-One Relationship	116
Using PVCs in Pods	116
Mounting PVCs in Pods	116
PVC Lifecycle in Pods	117
Imperative Commands for Storage	117
PV Management Commands	117
PVC Management Commands	117
Combined PV/PVC Operations	118
Declarative Configurations	118
Complete PV-PVC-Pod Example.....	118
Deployment with PVC	119
Storage Classes (Dynamic Provisioning)	120
What are Storage Classes?	120
Storage Class Benefits	120
Storage Class Definition	120
Using Storage Classes with PVCs	120
Reclaim Policies in Detail	120
Understanding Reclaim Policies	120

Retain Policy	121
Delete Policy.....	121
Recycle Policy (Deprecated)	121
Storage Best Practices.....	121
PV/PVC Design Principles	121
Security Considerations	121
Performance Optimization	121
Troubleshooting Storage Issues	122
Common PVC Problems	122
PV Binding Issues	122
Pod Mount Failures	122
Key Storage Commands Summary	122
Essential PV Commands	122
Essential PVC Commands	123
Storage Monitoring Commands	123
Networking	123
Networking Fundamentals	123
What is Kubernetes Networking?	123
Core Networking Principles	123
Why Kubernetes Networking Matters	123
The Kubernetes Network Model	124
What is the Kubernetes Network Model?	124
Network Model Components	124
Network Model Implementation	125
Container Network Interface (CNI)	125
What is CNI?	125
CNI Responsibilities	125
Popular CNI Plugins	125
CNI Configuration	126

IP Address Management (IPAM)	127
What is IPAM?	127
IPAM Plugins	127
IP Range Planning	127
Services and Service Discovery	128
What are Services?	128
Service Types	128
Service Discovery	129
Ingress and Ingress Controllers	130
What is Ingress?.....	130
Ingress Components	130
Advanced Ingress Features	131
Kubernetes Gateway API	132
What is the Gateway API?	132
Gateway API Benefits	132
Gateway API Architecture	132
HTTP Routing with Gateway API.....	133
Traffic Management with Gateway API	134
Advanced Gateway API Features	137
Installing NGINX Gateway Fabric	139
Practical Gateway API Examples	140
Network Policies	141
What are Network Policies?	141
Network Policy Components	141
Network Policy Examples	143
Network Policy Best Practices	145
DNS in Kubernetes	146
What is Kubernetes DNS?	146
DNS Architecture	146
DNS Records	147

DNS Configuration	148
Load Balancing and Service Mesh	148
Service Load Balancing	148
External Load Balancers	149
Imperative Commands for Networking.....	149
Service Management Commands	149
Ingress Management Commands	150
Network Policy Management Commands	150
DNS and Service Discovery Commands	150
Gateway API Commands	151
Declarative Configurations	151
Complete Network Stack Example.....	151
Troubleshooting Networking Issues	154
Common Networking Problems	154
Network Diagnostics Tools	155
Key Networking Commands Summary	155
Essential Service Commands.....	155
Essential Ingress Commands	156
Essential Network Policy Commands	156
Essential DNS Commands	156
Essential Gateway API Commands	156
Design and Install a Kubernetes Cluster HA (High Availability)	156
High Availability Overview	156
What is Kubernetes High Availability?.....	156
Why HA is Critical for Production	156
HA Topology Options	157
Stacked Control Plane Topology	157
External etcd Topology	157
Design Considerations	158
Infrastructure Requirements	158

Network Architecture Planning	158
Node Distribution Strategy	159
Storage Considerations.....	159
etcd High Availability	159
etcd Cluster Fundamentals	159
etcd Quorum and Fault Tolerance	160
etcd Best Practices	160
Control Plane High Availability	160
API Server HA Configuration	160
Controller Manager HA with Leader Election	161
Scheduler HA with Leader Election	161
Installation Process	162
Prerequisites and Planning	162
Step 1: Provision Infrastructure	162
Step 2: Install Container Runtime	163
Step 6: Configure Load Balancer	164
Step 8: Configure Networking	164
Step 9: Join Worker Nodes	164
Validation and Testing	164
Cluster Health Verification	164
Application Deployment Testing	165
Failure Testing	165
Production Best Practices	166
Security Hardening.....	166
Monitoring and Alerting	166
Backup and Disaster Recovery	166
Troubleshooting Common Issues	167
etcd Cluster Issues	167
API Server Connectivity Issues.....	167
Control Plane Recovery	167

Key Commands Summary	168
Installation Commands	168
Health Check Commands	168
Maintenance Commands	168
Helm and Kustomize - CKAD Essentials	169
Helm - Package Manager for Kubernetes	169
What is Helm?	169
Why Use Helm?	169
Helm Key Concepts.....	169
Basic Helm Usage	169
Installing Helm	169
Working with Repositories	169
Essential Helm Commands.....	169
Creating Simple Helm Charts	170
Chart Structure	170
Chart.yaml (Metadata)	170
values.yaml (Configuration)	170
Simple Deployment Template	171
Basic Template Functions	171
Working with Your Chart	172
Kustomize - Kubernetes Native Configuration	172
What is Kustomize?	172
Why Use Kustomize?	172
Basic Kustomize Structure	172
Basic Kustomize Usage	173
Base Configuration	173
Development Overlay	174
Production Overlay	174
Essential Kustomize Operations	175
Basic Commands	175

Common Transformers	175
Simple Patches	176
ConfigMaps and Secrets with Kustomize	176
Generate ConfigMaps	176
Generate Secrets	176
When to Use Helm vs Kustomize.....	177
Use Helm When:	177
Use Kustomize When:	177
Common Patterns	177
Environment Management with Kustomize	177
Application Distribution with Helm	177
Key Commands Summary	177
Helm Commands	177
Kustomize Commands	178
Best Practices	178
Helm Best Practices	178
Kustomize Best Practices	178

Definitions & Core Concepts

Container Runtime Fundamentals

What is a Container Runtime?

A **Container Runtime** is the software responsible for running containers on a host system. It manages the complete container lifecycle from creation to destruction.

Docker vs Containerd

- **Docker:** Complete container platform with CLI, API, image management, volumes, networking, and security features
- **Containerd:** Lightweight, industry-standard container runtime focused on core container operations
- **CRI (Container Runtime Interface):** Kubernetes standard for communicating with container runtimes
- **OCI (Open Container Initiative):** Industry standards for container formats and runtimes

CLI Tools Comparison

Tool	Purpose	Target Runtime	Use Case
docker	General container management general use	Docker daemon	Development, Limited debugging
ctr	Basic containerd control Docker-compatible CLI debugging	Containerd	nerdctl CRI
	Any CRI runtime	Kubernetes	troubleshooting

Key Commands

```
# Docker
docker run --name redis redis:alpine docker
ps -a

# Containerd (ctr)
ctr images pull docker.io/library/redis:alpine ctr
run docker.io/library/redis:alpine redis

# Nerdctl (Docker-Like for containerd)
nerdctl run --name redis redis:alpine nerdctl
ps -a
```

```
# CRI debugging
crictl pull busybox
crictl images
crictl ps -a crictl
pods
```

Pod Fundamentals

What is a Pod?

A **Pod** is the smallest deployable unit in Kubernetes that represents one or more containers sharing the same network and storage.

Pod Characteristics

- **Shared Network:** All containers in a pod share the same IP address and port space
- **Shared Storage:** Containers can share volumes mounted in the pod
- **Lifecycle Coupling:** All containers start and stop together
- **Ephemeral:** Pods are disposable and replaceable

Multi-Container Pod Patterns

- **Sidecar:** Helper container alongside main application (logging, monitoring)
- **Ambassador:** Proxy container for external connections
- **Adapter:** Container that standardizes output from main container

Pod Creation

```
# Imperative kubectl run nginx -
-image=nginx

# Declarative
kubectl apply -f pod-definition.yaml

apiVersion: v1
kind: Pod
metadata:
  name: myapp-pod
labels:
  app: myapp
spec:
  containers:
    - name: nginx-container
  image: nginx
```

Controllers and Workload Management

What is a Controller?

A **Controller** is a Kubernetes component that watches the cluster state and makes changes to move the current state toward the desired state.

ReplicationController vs ReplicaSet

Feature	ReplicationController	ReplicaSet
API Version	v1	apps/v1
Selector	Equality-based only	Set-based (matchLabels)
Feature	ReplicationController	ReplicaSet
Status	Legacy	Current standard
Used by	Older deployments	Deployments

What is a Deployment?

A **Deployment** provides declarative updates for pods and ReplicaSets, enabling rolling updates, rollbacks, and scaling.

Deployment Strategies

- **RollingUpdate** (Default): Gradually replace old pods with new ones
- **Recreate**: Kill all old pods before creating new ones

Key Commands

```
# Create deployment kubectl create deployment
nginx --image=nginx

# Scale deployment kubectl scale deployment
nginx --replicas=5

# Update image kubectl set image deployment/nginx
nginx=nginx:1.18

# Rollback kubectl rollout undo
deployment/nginx
```

```
# Check rollout status kubectl rollout
status deployment/nginx

### Example Deployment YAML with RollingUpdate Strategy

# API version for Deployment resource (apps/v1 is the current stable version)
apiVersion: apps/v1
# Resource type - Deployment manages ReplicaSets and provides declarative
updates kind: Deployment metadata:
    # Unique name for this deployment
name: nginx-deployment
    # Labels for organizing and selecting resources
labels:
    app: nginx spec:
        # Number of replicas (pods) to maintain
replicas: 3

    # Deployment strategy configuration
strategy:
    # RollingUpdate: Gradually replace old pods with new ones (default)
    # Alternative: Recreate (kills all old pods before creating new ones)
type: RollingUpdate      rollingUpdate:
    # maxSurge: Maximum number of pods above desired replicas during update
    # 25% means with 3 replicas, up to 1 additional pod (3 * 0.25 = 0.75, rounded
    # up to 1)
        # Can be absolute number (e.g., 2) or percentage (e.g., 25%)
maxSurge: 25%

    # maxUnavailable: Maximum number of pods unavailable during update
    # 25% means with 3 replicas, minimum 2 pods available (3 * 0.25 = 0.75,
    # rounded down to 0)
        # Can be absolute number (e.g., 1) or percentage (e.g., 25%)
maxUnavailable: 25%

    # Pod selector - determines which pods this deployment manages
selector:      matchLabels:          app: nginx

    # Pod template - defines the desired state for pods created by this
deployment template:      metadata:
        # Labels that pods created by this deployment will have
labels:
```

```

    app: nginx
spec:
containers:
name: nginx
  # Container image to use
image: nginx:1.18

  # Ports that the container exposes
ports:
- containerPort: 80

  # Resource requirements and limits
resources:
  # Minimum resources the container needs to run
requests:
    cpu: "100m"          # 0.1 CPU cores (100 millicores)
memory: "128Mi"          # 128 megabytes           # Maximum
resources the container can use           limits:
    cpu: "500m"          # 0.5 CPU cores (500 millicores)
memory: "512Mi"          # 512 megabytes
  # Health check to determine if container is alive
livenessProbe:           httpGet:           path: /
# HTTP path to check           port: 80           # Port to
check                  initialDelaySeconds: 30 # Wait 30s before first
probe                  periodSeconds: 10   # Check every 10
seconds

  # Health check to determine if container is ready to receive traffic
readinessProbe:          httpGet:           path: /           # HTTP path
to check                 port: 80           # Port to check
initialDelaySeconds: 5   # Wait 5s before first probe
periodSeconds: 5         # Check every 5 seconds

```

RollingUpdate Strategy Parameters

Parameter	Description	Default	Example
maxSurge	Maximum number of pods above desired replicas during update. Controls how many extra pods can be created during rollout.	25%	maxSurge: 2 or maxSurge: 25%

maxUnavailable Maximum number of pods 25% `maxUnavailable: 1` or unavailable during update. Ensures `maxUnavailable: 25%` minimum availability during rollout.

Example Calculation with 3 replicas:

- **maxSurge: 25%** = Up to 1 extra pod ($3 \times 0.25 = 0.75$, rounded up to 1)
- **maxUnavailable: 25%** = Minimum 2 pods available ($3 \times 0.25 = 0.75$, rounded down to 0 unavailable)
- **Result:** During update, you can have 1-4 pods total, with minimum 2 always available

Namespaces and Resource Isolation

What is a Namespace?

A **Namespace** provides virtual clustering within a physical Kubernetes cluster, enabling resource isolation and multi-tenancy.

Default Namespaces

- **default:** Default namespace for objects with no specified namespace
- **kube-system:** Namespace for Kubernetes system components
- **kube-public:** Publicly readable namespace for cluster information
- **kube-node-lease:** Namespace for node heartbeat objects

DNS in Namespaces

```
# Same namespace service-name
```

```
# Different namespace service-name.namespace-name.svc.cluster.local
```

Resource Quotas

Resource Quota limits the total resource consumption within a namespace.

```
apiVersion: v1 kind:  
ResourceQuota  
metadata: name:  
compute-quota  
namespace: dev spec:  
hard:  
  pods: "10"  
requests.cpu: "4"
```

```
requests.memory: 5Gi
limits.cpu: "10"
limits.memory: 10Gi
```

LimitRange

LimitRange sets minimum and maximum resource constraints for individual objects in a namespace.

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

Services and Networking

What is a Service?

A **Service** is an abstraction that defines a logical set of pods and provides stable network access to them.

Service Types

ClusterIP (Default)

- **Purpose:** Internal cluster communication only
- **IP Range:** Cluster-internal IP addresses
- **Access:** Only accessible from within the cluster

NodePort

- **Purpose:** External access via node ports • **Port Range:** 30000-32767
- **Access:** :

LoadBalancer

- **Purpose:** Cloud provider load balancer
- **Requirements:** Cloud provider integration
- **Access:** External IP provided by cloud provider

Service Port Definitions

Port Type	Definition	Example
targetPort	Port on the pod where traffic is sent	80 (nginx container port)
port	Port on the service where it accepts traffic	80 (service port) nodePort
	Port on the node for external access	30008 (external access)

Service Example

```
apiVersion: v1 kind:
Service metadata:
name: myapp-service
spec: type: NodePort
selector: app:
myapp ports:
- targetPort: 80      # Pod port
port: 80            # Service port
nodePort: 30008     # Node port
```

What is a Node IP?

Node IP is the IP address of the physical or virtual machine that runs Kubernetes worker nodes. External traffic reaches services through NodeIP:NodePort combination.

Configuration Management

What are ConfigMaps?

ConfigMaps store non-confidential configuration data in key-value pairs that can be consumed by pods as environment variables, command-line arguments, or configuration files.

```
apiVersion: v1 kind:
ConfigMap metadata: name:
app-config        data:
database.host: "postgres"
database.port: "5432"
app.properties: |
debug=true
log.level=info
```

What are Secrets?

Secrets store sensitive information like passwords, tokens, and keys in base64-encoded format with additional security measures.

```
apiVersion: v1 kind: Secret metadata: name: db-secret type: Opaque data: username: cG9zdGdyZXM= # postgres (base64) password: cGFzc3dvcmQ= # password (base64)
```

Using ConfigMaps and Secrets

```
apiVersion: v1
kind: Pod
metadata:
  name: myapp-pod
spec:
  containers:
    - name: myapp
      image: myapp:latest
      env:
        - name: DB_HOST
          valueFrom:
            configMapKeyRef:
              name: app-config
              key: database.host
        - name: DB_PASSWORD
          valueFrom:
            secretKeyRef:
              name: db-secret
              key: password
```

Kubernetes Control Plane Components

What is etcd?

etcd is a distributed, reliable key-value store that stores all Kubernetes cluster data and state information.

Data Stored in etcd

- Nodes, pods, deployments, services
- ConfigMaps, secrets
- Network policies, RBAC rules
- All cluster configuration

What is the API Server?

kube-apiserver is the central management component that exposes the Kubernetes API and handles all cluster communications.

API Server Responsibilities

- **Authentication:** Verify user identity
- **Authorization:** Check user permissions (RBAC)
- **Admission Control:** Validate and modify requests
- **Persistence:** Store data in etcd

What is the Scheduler?

kube-scheduler assigns pods to nodes based on resource requirements, constraints, and policies.

Scheduling Process

1. **Filtering:** Remove unsuitable nodes
2. **Scoring:** Rank remaining nodes
3. **Binding:** Assign pod to highest-scoring node

What is the Controller Manager?

kube-controller-manager runs various controllers that regulate cluster state.

Key Controllers

- **Node Controller:** Monitors node health
- **Replication Controller:** Maintains desired pod replicas
- **Deployment Controller:** Manages rolling updates
- **Service Account Controller:** Creates default service accounts

What is Kubelet?

kubelet is the node agent that manages pod lifecycle on worker nodes.

Kubelet Responsibilities

- **Pod Management:** Create, update, delete pods
- **Health Monitoring:** Report node and pod status
- **Resource Reporting:** Send resource usage to API server
- **Container Runtime Interface:** Communicate with container runtime

What is Kube-proxy?

kube-proxy maintains network rules and enables service communication within the cluster.

Kube-proxy Functions

- **Service Discovery:** Route traffic to service endpoints
- **Load Balancing:** Distribute traffic across pod replicas
- **Network Rules:** Manage iptables or IPVS rules

Resource Management

What are Resource Requests?

Resource Requests specify the minimum amount of CPU and memory a container needs to run.

What are Resource Limits?

Resource Limits specify the maximum amount of CPU and memory a container can consume.

```
resources:  
requests:  
  cpu: "100m"          # 0.1 CPU core minimum  
  memory: "128Mi"      # 128 megabytes minimum limits:  
    cpu: "500m"          # 0.5 CPU core maximum  
    memory: "512Mi"      # 512 megabytes maximum
```

Resource Units

- **CPU:** Measured in millicores (1000m = 1 core)
- **Memory:** Measured in bytes (Ki, Mi, Gi, Ti)

Best Practices

Declarative vs Imperative

Declarative (Recommended)

- **Definition:** Describe desired state in YAML files
- **Command:** kubectl apply -f file.yaml
- **Benefits:** Version control, reproducibility, collaboration

Imperative

- **Definition:** Direct commands to modify cluster state
- **Commands:** kubectl run, kubectl create, kubectl scale
- **Use Cases:** Quick testing, debugging, learning

Essential kubectl Commands

```
# Resource management
kubectl get kubectl
describe kubectl
create -f kubectl
apply -f kubectl
delete

# Debugging kubectl logs
kubectl exec -it -- /bin/bash
kubectl port-forward 8080:80

# Cluster information
kubectl cluster-info
kubectl get nodes kubectl
get events
```

Kubernetes Scheduling

DaemonSets

What is a DaemonSet?

A **DaemonSet** ensures that a copy of a pod runs on all (or specific) nodes in the cluster.

DaemonSet Use Cases

- **Node monitoring:** Monitoring agents on every node
- **Log collection:** Log collectors like Fluentd or Filebeat
- **Storage daemons:** Distributed storage solutions
- **Network components:** CNI plugins, kube-proxy

DaemonSet vs Deployment

Feature	DaemonSet	Deployment
Replica Logic	One per node	Total replica count
Scheduling	Node-based	Cluster-wide
Use Case	Node-level services	Application services
Updates	Rolling update per node	Rolling update by replica

DaemonSet Definition

```
apiVersion: apps/v1
kind: DaemonSet
metadata:
  name: monitoring-daemon
labels:
  app: monitoring-daemon
spec: selector:
matchLabels:
  app: monitoring-agent
template: metadata:
labels:
  app: monitoring-agent
spec: containers:
- name: monitoring-agent
image: monitoring-agent
resources: requests:
cpu: 100m           memory:
128Mi             limits:
cpu: 200m           memory:
256Mi
```

DaemonSet Scheduling Evolution

- **Before v1.12:** Used nodeName to bypass scheduler
- **v1.12+:** Uses NodeAffinity with default scheduler

DaemonSet Commands

```
# Create DaemonSet kubectl create -f daemon-set-definition.yaml

# View DaemonSets kubectl
get daemonsets

# Describe DaemonSet
kubectl describe daemonset monitoring-daemon

# View DaemonSet pods kubectl get pods -l
app=monitoring-agent -o wide
```

Static Pods

What are Static Pods?

Static Pods are pods managed directly by the kubelet on a specific node, without requiring the Kubernetes API server or any controllers like ReplicaSets or Deployments.

Key Characteristics

- **Managed by kubelet only:** No involvement of kube-apiserver or controllers
- **Node-specific:** Created and managed on the node where kubelet runs
- **Automatic restart:** kubelet automatically restarts static pods if they fail
- **Mirror pods:** API server creates read-only mirror pods for visibility

Static Pod Configuration

Method 1: Pod Manifest Path

```
# Configure kubelet with --pod-manifest-path
ExecStart=/usr/local/bin/kubelet \
--container-runtime=remote \
--container-runtime-endpoint=unix:///var/run/containerd/containerd.sock \
--pod-manifest-path=/etc/kubernetes/manifests \
--kubeconfig=/var/lib/kubelet/kubeconfig \
--network-plugin=cni \
--register-node=true \
--v=2
```

Method 2: Kubelet Configuration File

```
# kubeconfig.yaml staticPodPath:
/etc/kubernetes/manifests

# Use config file
ExecStart=/usr/local/bin/kubelet \
--config=kubeconfig.yaml \
--kubeconfig=/var/lib/kubelet/kubeconfig \
--network-plugin=cni \
--register-node=true \ --v=2
```

Static Pod Use Cases

Control Plane Components

Static pods are commonly used to run control plane components on master nodes:

- **kube-apiserver:** API server pod

- **kube-controller-manager**: Controller manager pod
- **kube-scheduler**: Scheduler pod
- **etcd**: Database pod

Static Pod Lifecycle

1. **Creation**: Place YAML file in `/etc/kubernetes/manifests/`
2. **Detection**: kubelet monitors the directory and detects new files
3. **Pod Creation**: kubelet creates the pod directly
4. **Mirror Pod**: API server creates a read-only mirror for visibility
5. **Management**: kubelet handles restarts, updates, and deletion

Static Pod Example

```
# /etc/kubernetes/manifests/static-web.yaml
apiVersion: v1 kind: Pod metadata:
  name: static-web
  labels:
    app: static-web
spec: containers:
- name: web
  image: nginx
  ports:
    - containerPort: 80
```

Static Pods vs DaemonSets

Feature	Static Pods	DaemonSets
Creation	kubelet directly	kube-apiserver (DaemonSet Controller)
Management	Node-level	Cluster-level
Use Case	Control plane components	Monitoring agents, log collectors
Scheduler	Ignored by scheduler	Uses scheduler
API Visibility	Mirror pods only	Full API objects

Viewing Static Pods

```
# View static pods (will show mirror pods) kubectl
get pods -n kube-system
```

Example output showing static pods with node names

NAME	READY	STATUS	RESTARTS	AGE	etcd-master
1/1 Running 0	15m	kube-apiserver-master		1/1	
Running 0	15m	kube-controller-manager-master	1/1	Running 0	
15m kube-scheduler-master	1/1	Running 0		15m	

Taints and Tolerations

What are Taints and Tolerations?

Taints are applied to nodes to repel pods, while **tolerations** are applied to pods to allow them to be scheduled on tainted nodes.

Taint Effects

Effect	Description
NoSchedule	Prevents new pods from being scheduled
PreferNoSchedule	Avoids scheduling if possible
NoExecute	Evicts existing pods and prevents new ones

Applying Taints

```
# Taint a node kubectl taint nodes node1
app=myapp:NoSchedule
```

```
# Remove a taint kubectl taint nodes node1
app=myapp:NoSchedule-
```

```
# View node taints kubectl describe node
node1 | grep Taint
```

Pod Tolerations

```
apiVersion: v1
kind: Pod
metadata:
  name: myapp-pod
spec:
  containers:
    - name: nginx-container
  image: nginx
  tolerations:
    - key: "app"
      operator: Equal
      value: "myapp"
      effect: "NoSchedule"
```

Master Node Taints

Master nodes are typically tainted to prevent regular workloads:

```
kubectl describe node master | grep Taint  
# Output: Taints: node-role.kubernetes.io/master:NoSchedule
```

Toleration Operators

- **Equal**: Key/value must match exactly
- **Exists**: Only key must match (ignores value)

Node Selectors and Node Affinity

What is Node Selection?

Node selection mechanisms allow you to constrain pods to run on specific nodes based on node labels.

Node Selectors (Simple Method)

```
Label Nodes # Label a node kubectl label  
nodes node-1 size=Large
```

```
# View node Labels kubectl get  
nodes --show-labels
```

```
Use Node Selector  
apiVersion: v1  
kind: Pod  
metadata: name:  
myapp-pod spec:  
containers:  
- name: data-processor  
image: data-processor  
nodeSelector: size:  
Large
```

Node Selector Limitations

- **Simple equality only**: Cannot handle complex logic
- **No OR conditions**: Cannot select “Large OR Medium”
- **No negation**: Cannot specify “NOT Small”

Node Affinity (Advanced Method)

Required Node Affinity

```
apiVersion: v1
kind: Pod
metadata:
  name: myapp-pod
spec:
  containers:
  - name: data-processor    image: data-processor
    affinity:
      nodeAffinity:
        requiredDuringSchedulingIgnoredDuringExecution:
          nodeSelectorTerms:
          - matchExpressions:
            - key: size           operator: In
              values:
              - Large
            - Medium
```

Preferred Node Affinity apiVersion: v1 kind: Pod metadata:

```
name: myapp-pod
spec:
  affinity:
    nodeAffinity:
      preferredDuringSchedulingIgnoredDuringExecution:
      - weight: 100      preference:
        matchExpressions:
        - key: size
          operator: In
          values:
          - Large
```

Node Affinity Operators

- **In:** Value must be in the list
- **NotIn:** Value must not be in the list
- **Exists:** Key must exist (ignores values)
- **DoesNotExist:** Key must not exist
- **Gt:** Greater than (numeric values)
- **Lt:** Less than (numeric values)

Node Affinity Types

Type	During Scheduling	During Execution
requiredDuringSchedulingIgnoredDuringExecution	Required	Ignored
preferredDuringSchedulingIgnoredDuringExecution	Preferred	Ignored
requiredDuringSchedulingRequiredDuringExecution	Required	Required (Planned)

Lifecycle Phases

- **DuringScheduling:** Rules applied when pod is first scheduled
- **DuringExecution:** Rules applied to running pods when node labels change

Manual Scheduling

What is Manual Scheduling?

Manual Scheduling is the process of assigning pods to specific nodes without using the Kubernetes scheduler.

When Manual Scheduling is Needed

- **No scheduler available:** Scheduler is down or not configured
- **Custom placement logic:** Specific node requirements
- **Troubleshooting:** Testing pod placement
- **Static pods:** Control plane components

Manual Scheduling Methods

Method 1: *nodeName* in Pod Spec

```
apiVersion: v1
kind: Pod
metadata:
  name: nginx
spec:
  containers:
    - name: nginx
      image: nginx
    ports:
      - containerPort: 80  nodeName: node02
# Direct node assignment
```

Method 2: Binding API (Post-Creation)

```
# Pod without nodeName (will remain Pending)
apiVersion: v1 kind: Pod metadata:  name:
  nginx spec:  containers:  - name: nginx
  image: nginx

# Binding object
apiVersion: v1
kind: Binding
metadata:
```

```
name: nginx
target:
  apiVersion: v1
kind: Node  name:
node02
# Create binding via API curl --header
"Content-Type:application/json" \
--request POST \
--data '{"apiVersion":"v1", "kind": "Binding", "metadata": {"name": "nginx"}, "target": {"apiVersion": "v1", "kind": "Node", "name": "node02"}}'
\
  http://$SERVER/api/v1/namespaces/default/pods/nginx/binding/
```

Multiple Schedulers

What are Multiple Schedulers?

Kubernetes supports running **multiple schedulers** simultaneously, allowing different scheduling policies for different types of workloads.

Use Cases for Multiple Schedulers

- **Specialized workloads:** GPU, high-memory, or compute-intensive tasks
- **Custom scheduling logic:** Business-specific placement rules
- **Multi-tenancy:** Different schedulers for different teams
- **Performance optimization:** Specialized algorithms for specific use cases

Deploying Custom Scheduler

Binary Deployment

```
# Download scheduler binary
wget
https://storage.googleapis.com/kubernetes-release/release/v1.12.0/bin/linux/amd64/kube-scheduler
```

```
# Configure custom scheduler service
# my-custom-scheduler.service
ExecStart=/usr/local/bin/kube-scheduler \
--config=/etc/kubernetes/config/kube-scheduler.yaml \
--scheduler-name=my-custom-scheduler
kubeadm Deployment # my-custom-
scheduler.yaml apiVersion: v1 kind:
Pod metadata:
```

```
name: my-custom-scheduler
namespace: kube-system spec:
  containers:
    - command:
      - kube-scheduler
      - --address=127.0.0.1
      - --kubeconfig=/etc/kubernetes/scheduler.conf
      - --leader-elect=true
      - --scheduler-name=my-custom-scheduler
      - --lock-object-name=my-custom-scheduler      image: k8s.gcr.io/kube-
scheduler-amd64:v1.11.3      name: my-custom-scheduler
```

Using Custom Scheduler

```
apiVersion: v1 kind: Pod metadata:  name: nginx spec:  containers:
  - name: nginx    image: nginx    schedulerName: my-custom-scheduler
# Specify custom scheduler
```

Monitoring Multiple Schedulers

```
# View all schedulers kubectl get pods -
-namespace=kube-system
```

```
# View scheduler events kubectl
get events
```

```
# View scheduler logs kubectl logs my-custom-scheduler --
namespace=kube-system
```

Network Policies

What are Network Policies?

Network Policies are Kubernetes resources that control network traffic between pods, providing micro-segmentation at the application level.

Network Policy Components

- **podSelector:** Selects pods the policy applies to
- **policyTypes:** Ingress, Egress, or both
- **ingress:** Rules for incoming traffic
- **egress:** Rules for outgoing traffic

Basic Network Policy Example

```
apiVersion: networking.k8s.io/v1
kind: NetworkPolicy metadata:
  name: db-policy spec:
    podSelector:
  matchLabels:
  role: db
  policyTypes: -
  Ingress
  ingress:
  - from:
    podSelector:
    matchLabels:
    name: api-pod
    ports:
    - protocol: TCP
      port: 3306
```

Network Policy Rules

Ingress Rules (Incoming Traffic)

```
spec:
  ingress:
  - from:
    # Pod selector - allow from specific pods
    - podSelector:          matchLabels:
  app: frontend
    # Namespace selector - allow from specific namespaces
    - namespaceSelector:    matchLabels:
      name: production
    # IP block - allow from specific IP ranges
    - ipBlock:
      cidr: 192.168.1.0/24
  except:
  - 192.168.1.5/32    ports:
  - protocol: TCP      port:
  80
```

Egress Rules (Outgoing Traffic) spec:

```
policyTypes:
- Egress
egress:
  - to:
    - ipBlock:
  cidr: 10.0.0.0/24
```

```
ports:
  - protocol: TCP
    port: 5432
```

Ingress

What is Ingress?

Ingress is a Kubernetes resource that manages external access to services, typically HTTP/HTTPS, providing load balancing, SSL termination, and name-based virtual hosting.

Ingress Components

- **Ingress Controller:** Implements ingress functionality (nginx, traefik, etc.)
- **Ingress Resource:** Configuration defining routing rules

Ingress vs Services

Feature	Service (NodePort/LoadBalancer)	Ingress
---------	---------------------------------	---------

Layer	Layer 4 (TCP/UDP)	Layer 7 (HTTP/HTTPS)
Routing	Simple port-based	Path/host-based routing
SSL/TLS	External configuration	Built-in SSL termination
Cost	Multiple load balancers	Single entry point

Ingress Controller Deployment

```
apiVersion: apps/v1 kind:
Deployment metadata:
  name: nginx-ingress-controller
spec: replicas: 1 selector:
matchLabels: name: nginx-
ingress template: metadata:
labels:
  name: nginx-ingress
spec: containers:
- name: nginx-ingress-controller
  image: quay.io/kubernetes-ingress-controller/nginx-
ingresscontroller:0.21.0 args:
- /nginx-ingress-controller
- --configmap=$(POD_NAMESPACE)/nginx-configuration env:
- name: POD_NAME valueFrom: fieldRef:
  fieldPath: metadata.name
- name: POD_NAMESPACE valueFrom:
fieldRef: fieldPath:
metadata.namespace ports: -
```

```
name: http           containerPort: 80
- name: https        containerPort: 443
```

Ingress Resource Examples

```
Simple Ingress  apiVersion:
networking.k8s.io/v1      kind:
Ingress metadata:
  name: ingress-wear
spec: rules: - http:
paths:   - path: /
pathType: Prefix
backend:
service:
  name: wear-service
port:      number: 80
```

```
Path-based Routing  apiVersion:
networking.k8s.io/v1      kind:
Ingress metadata:
  name: ingress-wear-watch
spec: rules: - http:
paths:   - path: /wear
pathType: Prefix
backend:
service:
  name: wear-service
port:      number: 80
- path: /watch
pathType: Prefix
backend:      service:
  name: watch-service
port:      number: 80
```

```
Host-based Routing  apiVersion:
networking.k8s.io/v1      kind:
Ingress metadata:
  name: ingress-wear-watch
spec: rules:
  - host: wear.my-online-store.com
http:   paths:   - path: /
pathType: Prefix      backend:
service:
```

```
        name: wear-service
port:
    number: 80 - host:
watch.my-online-store.com     http:
paths:      - path: /
pathType: Prefix           backend:
service:
    name: watch-service
port:          number: 80
```

Key Commands

Static Pods

```
# Check static pod directory ls
/etc/kubernetes/manifests/
```



```
# View kubelet configuration systemctl
status kubelet
```

Taints and Tolerations

```
# Apply taint kubectl taint nodes node1
key=value:NoSchedule
```



```
# Remove taint kubectl taint nodes node1
key=value:NoSchedule-
```



```
# Check node taints kubectl describe
node node1 | grep Taint
```

Node Affinity

```
# Label nodes kubectl label nodes
node1 size=Large
```

```
# View node labels kubectl get
nodes --show-labels
```

DaemonSets

```
# Create DaemonSet kubectl
create -f daemonset.yaml # View
DaemonSets
kubectl get daemonsets
```

```
# View DaemonSet pods kubectl get pods -l  
app=monitoring-agent -o wide
```

Multiple Schedulers

```
# View schedulers  
kubectl get pods -n kube-system
```

```
# View scheduler events kubectl  
get events
```

```
# View scheduler logs kubectl logs  
scheduler-name -n kube-system
```

Network Policies

```
# Create network policy  
kubectl create -f network-policy.yaml
```

```
# View network policies kubectl  
get networkpolicies
```

```
# Describe network policy kubectl describe  
networkpolicy policy-name
```

Ingress

```
# Create ingress  
kubectl create -f ingress.yaml
```

```
# View ingress kubectl  
get ingress
```

```
# Describe ingress kubectl describe  
ingress ingress-name
```

Logging and Monitoring

Container Logging

What is Container Logging?

Container Logging is the process of collecting, storing, and accessing application logs generated by containers running in pods.

Docker vs Kubernetes Logging

Docker Logging

```
# Run container and view logs directly  
docker run kodekloud/event-simulator
```

```
# Run in detached mode and view logs docker  
run -d kodekloud/event-simulator docker  
logs -f
```

```
Kubernetes Logging # Create pod and view  
Logs  kubectl create -f event-  
simulator.yaml kubectl logs -f event-  
simulator-pod
```

Log Access Methods

Single Container Pod

```
# View logs from single container pod kubectl  
logs
```

```
# Follow logs in real-time kubectl  
logs -f
```

```
# View previous container logs (after restart) kubectl  
logs --previous
```

Multi-Container Pod

```
# Specify container name for multi-container pods kubectl  
logs -f
```

```
# Example with multi-container pod kubectl logs -f  
event-simulator-pod event-simulator
```

Pod Definition for Logging

```
# Single container pod  
apiVersion: v1 kind:  
Pod metadata:  
  name: event-simulator-pod spec:
```

```
containers: - name: event-
simulator image:
kodekloud/event-simulator

# Multi-container pod
apiVersion: v1 kind: Pod
metadata: name: event-
simulator-pod spec:
containers:
- name: event-simulator
  image: kodekloud/event-simulator
- name: image-processor    image: some-image-
processor
```

Log Storage Locations

- **Container Runtime:** Docker stores logs in `/var/lib/docker/containers/`
- **Kubelet:** Manages log rotation and cleanup
- **Node-level:** Logs accessible via `kubectl` from any cluster node

Log Limitations

- **Ephemeral:** Logs are lost when pods are deleted
- **No centralized storage:** Logs stored locally on nodes
- **Limited retention:** Subject to log rotation policies
- **No aggregation:** No built-in cross-pod log correlation

Kubernetes Monitoring

What is Monitoring in Kubernetes?

Monitoring is the continuous observation of cluster and application performance through metrics collection, storage, and analysis.

Key Metrics to Monitor

Node-Level Metrics

- **CPU Usage:** Processor utilization across nodes
- **Memory Usage:** RAM consumption and availability
- **Disk Usage:** Storage utilization and I/O performance
- **Network:** Traffic, latency, and packet loss

Pod-Level Metrics

- **Resource Consumption:** CPU, memory usage per pod
- **Restart Count:** Pod failure and restart frequency
- **Health Status:** Pod readiness and liveness states

Cluster-Level Metrics

- **API Server Performance:** Request latency and throughput
- **Scheduler Performance:** Pod placement efficiency
- **Controller Performance:** Reconciliation loop metrics

Monitoring Solutions

Built-in Solutions

- **Metrics Server:** Lightweight, resource metrics collection
- **cAdvisor:** Container-level resource usage statistics

Third-Party Solutions

- **Prometheus:** Open-source monitoring and alerting
- **Elastic Stack:** Log aggregation and analysis
- **Datadog:** Commercial monitoring platform
- **Dynatrace:** Application performance monitoring

Metrics Server

What is Metrics Server?

Metrics Server is a cluster-wide aggregator of resource usage data that provides CPU and memory metrics for nodes and pods.

Metrics Server vs Heapster

Feature	Heapster (Deprecated)	Metrics Server
Status	Deprecated in K8s 1.11	Current standard
Storage	Persistent storage	In-memory only
Scope	Full monitoring solution	Resource metrics only
Performance	Heavier resource usage	Lightweight
API	Custom API	Standard Metrics API

Metrics Server Architecture

Components

- **Metrics Server Pod:** Collects and aggregates metrics
- **Kubelet Integration:** Pulls metrics from kubelet on each node
- **cAdvisor:** Provides container-level metrics to kubelet
- **API Server:** Exposes metrics via Kubernetes API

Data Flow cAdvisor → Kubelet → Metrics Server → Kubernetes API → kubectl top

Metrics Server Installation

Minikube

```
# Enable metrics-server addon minikube
addons enable metrics-server
```

Manual Installation

```
# Clone metrics-server repository
git clone https://github.com/kubernetes-incubator/metrics-server.git
```

```
# Deploy metrics-server kubectl
create -f deploy/1.8+/
```

Installation Output

```
clusterrolebinding "metrics-server:system:auth-delegator" created
rolebinding "metrics-server-auth-reader" created
apiservice "v1beta1.metrics.k8s.io" created
serviceaccount "metrics-server" created
deployment "metrics-server" created
service "metrics-server" created
clusterrole "system:metrics-server" created
clusterrolebinding "system:metrics-server" created
```

Using Metrics Server

View Node Metrics kubectl
top node

NAME	CPU(cores)	CPU%	MEMORY(bytes)	MEMORY%	
kubemaster	166m	8%	1337Mi	70%	kubenode1
36m	1%	1046Mi		55%	kubenode2
1%	1048Mi	55%			39m

View Pod Metrics kubectl
top pod

NAME	CPU(cores)	MEMORY(bytes)
nginx	5m	10Mi
	2m	redis 15Mi

Namespace-Specific Metrics

```
# View pods in specific namespace kubectl
top pod -n kube-system

# View all namespaces kubectl
top pod --all-namespaces
```

Metrics Server Data Storage

- **In-Memory Only:** No persistent storage of historical data
- **Real-Time:** Provides current resource usage only
- **Aggregation Period:** Typically 15-60 second intervals
- **Retention:** Limited to current metrics only

Advanced Monitoring Concepts

Resource Requests vs Usage

- **Requests:** Resources guaranteed by scheduler
- **Usage:** Actual resources consumed by container
- **Monitoring Value:** Usage vs requests comparison shows over/under-provisioning

Horizontal Pod Autoscaler (HPA) Integration

```
apiVersion: autoscaling/v2
kind: HorizontalPodAutoscaler
metadata:
  name: webapp-hpa
spec:
  scaleTargetRef:
    apiVersion: apps/v1    kind:
    Deployment      name: webapp
    minReplicas: 2  maxReplicas:
    10  metrics:  - type:
    Resource      resource:
    name: cpu      target:
    type: Utilization
    averageUtilization: 70
```

Custom Metrics

- **Application Metrics:** Business-specific measurements
- **Custom Resource Metrics:** GPU, network bandwidth

- **External Metrics:** Cloud provider metrics, external services

Logging Best Practices

Application Logging

- **Structured Logging:** Use JSON format for easier parsing
- **Log Levels:** Implement DEBUG, INFO, WARN, ERROR levels
- **Contextual Information:** Include request IDs, user IDs, timestamps

Container Logging Standards

- **Stdout/Stderr:** Write logs to standard output streams
- **Avoid File Logging:** Don't write logs to container filesystem
- **Single Process:** One main process per container for log clarity

Log Management

```
# View Logs with timestamps kubectl
logs --timestamps

# Limit log output
kubectl logs --tail=100

# Filter logs by time kubectl
logs --since=1h
```

Monitoring Best Practices

Metrics Collection

- **Resource Limits:** Set appropriate CPU/memory limits
- **Health Checks:** Implement liveness and readiness probes
- **Service Monitoring:** Monitor service endpoints and response times

Alerting Strategy

- **Threshold-Based:** CPU > 80%, Memory > 90%
- **Trend-Based:** Resource usage growing over time
- **Service-Level:** Application-specific metrics and SLAs

Dashboard Design

- **Node Overview:** Cluster-wide resource utilization
- **Application Metrics:** Service-specific performance

- **Infrastructure Health:** Node status, pod distribution

Troubleshooting Common Issues

Metrics Server Issues

```
# Check metrics-server status  
kubectl get pods -n kube-system | grep metrics-server  
  
# View metrics-server logs  
kubectl logs -n kube-system deployment/metrics-server  
  
# Verify metrics API kubectl get --raw  
/apis/metrics.k8s.io/v1beta1/nodes
```

Logging Issues

```
# Pod not generating logs kubectl  
describe pod  
  
# Container exit codes kubectl  
get pods -o wide  
  
# Previous container logs after restart kubectl  
logs --previous  
Performance Troubleshooting  
# High CPU usage investigation  
kubectl top pods --sort-by=cpu  
  
# Memory pressure analysis kubectl  
top pods --sort-by=memory  
  
# Node resource availability  
kubectl describe node
```

Version Compatibility Notes

Kubernetes Version Support

- **Metrics Server:** Supported in K8s 1.8+
- **Heapster:** Deprecated since K8s 1.11
- **Play-with-K8s:** Uses K8s 1.8, may require Heapster

Migration Considerations

- **Legacy Clusters:** May still use Heapster
- **Upgrade Path:** Migrate from Heapster to Metrics Server
- **API Changes:** Update monitoring tools to use Metrics API

Key Commands Summary

Logging Commands

```
# Basic log viewing kubectl logs
kubectl logs -f    # Follow logs
kubectl logs -c    # Multi-container

# Advanced log options kubectl logs --timestamps
kubectl logs --tail=50 kubectl logs
--since=2h kubectl logs --previous # Previous container
```

Monitoring Commands

```
# Resource usage kubectl top
nodes kubectl top pods kubectl top
pods --all-namespaces
kubectl top pods --sort-by=cpu

# Metrics server
kubectl get pods -n kube-system | grep metrics kubectl
get apiservice | grep metrics
```

Troubleshooting Commands

```
# Pod investigation kubectl
describe pod
kubectl get events --sort-by=.metadata.creationTimestamp

# Node investigation kubectl
describe node kubectl get
nodes -o wide
```

Application Lifecycle Management

Rolling Updates and Rollbacks

What is a Rollout?

A **Rollout** is the process of deploying a new version of an application in Kubernetes. Each rollout creates a new **revision** that can be tracked and managed.

Rollout Revisions

- **Revision 1:** Initial deployment (e.g., nginx:1.7.0)
- **Revision 2:** Updated deployment (e.g., nginx:1.7.1)
- **Revision History:** Kubernetes maintains a history of all revisions

Deployment Strategies

Strategy	Description	Downtime	Use Case
Recreate	Kill all old pods, then create ones apps	Yes	Development, non-critical new
RollingUpdate	Gradually replace old pods with new ones	No	Production applications (default)

Rolling Update Process

1. **Create new ReplicaSet** with updated image
2. **Scale up new ReplicaSet** gradually
3. **Scale down old ReplicaSet** simultaneously
4. **Maintain desired replica count** throughout process

Rollout Commands

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

```
# View rollout history kubectl rollout history
deployment/myapp-deployment
```

```
# Trigger rollout with record
kubectl apply -f deployment-definition.yml --record
```

```
# Update image and trigger rollout kubectl set image
deployment/myapp-deployment nginx=nginx:1.9.1
```

Example Rollout Output

```
kubectl rollout status deployment/myapp-deployment
# Output:
```

```
Waiting for rollout to finish: 0 of 10 updated replicas are available...
Waiting for rollout to finish: 1 of 10 updated replicas are available...
...
deployment "myapp-deployment" successfully rolled out
```

Rollback Operations

```
# Rollback to previous revision kubectl rollout
undo deployment/myapp-deployment

# View ReplicaSets during rollback kubectl
get replicsets

# Before rollback:
# myapp-deployment-67c749c58c    0    0    0    22m  (old)
# myapp-deployment-7d57dbdb8d    5    5    5    20m  (current)

# After rollback:
# myapp-deployment-67c749c58c    5    5    5    22m  (restored)
# myapp-deployment-7d57dbdb8d    0    0    0    20m  (scaled down)
```

Deployment Definition

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: myapp-deployment
labels:
  app: myapp
type: front-end
spec:
  replicas: 3
selector:
  matchLabels:
    type: front-end
template:
metadata:
  name: myapp-pod
labels:
  app: myapp
type: front-end
spec:   containers:
  - name: nginx-container
image: nginx:1.7.1
```

Commands and Arguments

Docker Container Behavior

- **Default Command:** Containers run default command specified in Dockerfile
- **Override Command:** Can override default command at runtime
- **Process Lifecycle:** Container exits when main process completes

Docker Command Examples

```
# Default ubuntu behavior (exits immediately)
docker run ubuntu docker ps -a
# STATUS: Exited (0) 41 seconds ago
```

```
# Override with custom command docker
run ubuntu sleep 5
```

Dockerfile Instructions

CMD Instruction

CMD specifies the default command to run when container starts:

```
FROM Ubuntu
CMD sleep 5

# Shell form
CMD command param1

# Exec form (preferred)
CMD ["command", "param1"]
```

ENTRYPOINT Instruction

ENTRYPOINT specifies the command that always executes:

```
FROM Ubuntu
ENTRYPOINT ["sleep"]

# Usage docker run ubuntu-
sleeper 10 # Command at
startup: sleep 10
```

ENTRYPOINT + CMD Combination

```
FROM Ubuntu
```

```
ENTRYPOINT ["sleep"]
```

CMD ["5"] **Benefits:**

- **Default behavior:** docker run ubuntu-sleeper → sleep 5
- **Override parameter:** docker run ubuntu-sleeper 10 → sleep 10
- **Override entrypoint:** docker run --entrypoint sleep2.0 ubuntu-sleeper 10

Kubernetes Commands and Arguments

Pod	Command	Override
apiVersion: v1	kind: Pod	
metadata: name: ubuntu-		
sleeper-pod	spec:	
containers:		
- name: ubuntu-sleeper	image: ubuntu-	
sleeper	command: ["sleep2.0"] # Overrides	
ENTRYPOINT	args: ["10"]	# Overrides
CMD		

Docker vs Kubernetes Mapping

Docker	Kubernetes	Purpose
--------	------------	---------

ENTRYPOINT	command	Main executable
CMD	args	Default arguments

Environment Variables

What are Environment Variables?

Environment Variables are key-value pairs that provide configuration data to applications at runtime.

Flask Application Example

```
# app.py import
os
from flask import Flask

app = Flask(__name__)
color = os.environ.get('APP_COLOR') # Read from environment
```

```
@app.route("/") def
main():
    print(color)
    return render_template('hello.html', color=color)

if __name__ == "__main__":
    app.run(host="0.0.0.0", port="8080")
```

Setting Environment Variables

Docker

```
# Set environment variable
docker run -e APP_COLOR=blue simple-webapp-color
docker run -e APP_COLOR=green simple-webapp-color docker
run -e APP_COLOR=pink simple-webapp-color
```

Kubernetes Pod

```
apiVersion: v1
kind: Pod
metadata:
  name: simple-webapp-color
spec:
  containers:
  - name: simple-webapp-
    color
    image: simple-
    webapp-color
    ports:
    - containerPort: 8080
  env:
  - name: APP_COLOR
    value: pink
```

Environment Variable Value Types

1. Plain Key-Value

```
env:
- name: APP_COLOR
  value: pink
```

2. ConfigMap Reference

```
env:
- name: APP_COLOR
  valueFrom:
    configMapKeyRef:
      name: app-config
      key: APP_COLOR
```

3. Secret Reference

```
- name: APP_COLOR  
valueFrom:  
secretKeyRef:  
name: app-secret  
key: APP_COLOR
```

ConfigMaps

What are ConfigMaps?

ConfigMaps store configuration data in key-value pairs that can be consumed by pods as environment variables, command-line arguments, or configuration files.

ConfigMap Use Cases

- **Application settings:** Database URLs, feature flags
- **Configuration files:** Application config, logging config
- **Non-sensitive data:** Any configuration that's not secret

Creating ConfigMaps

Imperative Method # From literal values

```
kubectl create configmap app-config \  
--from-literal=APP_COLOR=blue \  
--from-literal=APP_MODE=prod
```

```
# From file kubectl create configmap  
app-config \  
--from-  
file=app_config.properties
```

Declarative Method

```
# config-map.yaml  
apiVersion: v1 kind:  
ConfigMap metadata:  
  name: app-config data:  
    APP_COLOR: blue    APP_MODE: prod  
    app.properties: |      debug=true  
      log.level=info kubectl create -f  
      config-map.yaml
```

Viewing ConfigMaps

```
# List ConfigMaps  
kubectl get configmaps
```

```
# Detailed view kubectl describe configmap app-config
```

Using ConfigMaps in Pods

Environment Variables (All Keys)

```
apiVersion: v1 kind: Pod
metadata:
  name: simple-webapp-color
spec: containers:
- name: simple-
webapp-color    image:
simple-webapp-color
envFrom:
- configMapRef:
name: app-config
```

Single Environment Variable env:

```
- name: APP_COLOR
valueFrom:
configMapKeyRef:
name: app-config
key: APP_COLOR
```

Volume Mount volumes:

```
- name: app-config-
volume configMap:
name: app-config
```

Secrets

What are Secrets?

Secrets store sensitive information like passwords, tokens, and keys in base64-encoded format with additional security measures.

Secret vs ConfigMap

Feature	ConfigMap	Secret
Data Type	Non-sensitive configuration	Sensitive information
Encoding	Plain text	Base64 encoded
Security	Standard	Enhanced (encryption at rest)

Use Cases App config, feature flags Passwords, tokens, certificates

Creating Secrets

Imperative Method kubectl create secret generic app-secret \
--from-literal=DB_Host=mysql \
--from-literal=DB_User=root \
--from-literal=DB_Password=paswrd

Declarative Method

```
# secret-data.yaml
apiVersion: v1
kind: Secret
metadata:
  name: app-secret
data:
  DB_Host: bXlzcWw= # mysql (base64)
  DB_User: cm9vdA== # root (base64)
  DB_Password: cGFzd3Jk # paswrd (base64)
```

Base64 Encoding/Decoding

```
# Encode echo -n 'mysql'
| base64
# Output: bXlzcWw=
```

```
# Decode echo -n 'bXlzcWw=' | base64
--decode # Output: mysql
```

Viewing Secrets

```
# List secrets
kubectl get secrets
```

```
# Describe secret (shows size, not values) kubectl
describe secret app-secret
```

```
# View secret YAML (shows base64 values) kubectl
get secret app-secret -o yaml
```

Using Secrets in Pods

Environment Variables (All Keys)

```
spec:
  containers:
    - name: simple-webapp-color
      image: simple-webapp-color
```

```
envFrom:          - secretRef:  
name: app-secret
```

Single Environment Variable

```
env:      - name:  
DB_PASSWORD  
valueFrom:  
secretKeyRef:  
name:     app-secret  
key: DB_Password
```

Volume Mount volumes:

```
- name: app-secret-volume  
secret: secretName:  
app-secret
```

Secrets as Volumes

When mounted as volumes, each key becomes a file:

```
# Inside container ls  
/opt/app-secret-volumes #  
Output: DB_Host  
DB_Password  DB_User  
  
cat /opt/app-secret-volumes/DB_Password #  
Output: paswrd
```

Multi-Container Pods

What are Multi-Container Pods?

Multi-Container Pods contain multiple containers that share the same lifecycle, network, and storage.

Multi-Container Patterns

Sidecar Pattern

- **Main container:** Application
- **Sidecar container:** Supporting functionality (logging, monitoring)
- **Example:** Web server + log agent

Ambassador Pattern

- **Main container:** Application
- **Ambassador container:** Proxy for external connections
- **Example:** App + database proxy

Adapter Pattern

- **Main container:** Application
- **Adapter container:** Transforms/standardizes output
- **Example:** App + log format converter

Shared Resources in Multi-Container Pods

Lifecycle

- **Start together:** All containers start simultaneously
- **Stop together:** Pod termination affects all containers
- **Restart policy:** Applied to entire pod

Network

- **Shared IP:** All containers share same IP address
- **Localhost communication:** Containers communicate via localhost
- **Port sharing:** Containers must use different ports

Storage

- **Shared volumes:** Containers can share mounted volumes
- **Data exchange:** File-based communication between containers

Multi-Container Pod Definition

```
apiVersion: v1
kind: Pod
metadata:
  name: simple-webapp
spec:
  containers:
  - name: simple-webapp
    image: simple-webapp
    ports:
    - containerPort: 8080
  - name: log-agent
    image: log-agent
```

Use Cases for Multi-Container Pods

- **Logging:** Main app + log collection sidecar
- **Monitoring:** Application + metrics collection

- **Service mesh:** App + proxy (Istio, Linkerd)
- **Data synchronization:** App + data sync agent
- **Security scanning:** App + security agent

Application Lifecycle Commands Summary

Rolling Updates and Rollbacks

Deployment management

```
kubectl create -f deployment-definition.yml kubectl  
get deployments  
kubectl apply -f deployment-definition.yml  
kubectl set image deployment/myapp-deployment nginx=nginx:1.9.1
```

Rollout operations

```
kubectl rollout status deployment/myapp-deployment kubectl  
rollout history deployment/myapp-deployment kubectl  
rollout undo deployment/myapp-deployment
```

ConfigMaps and Secrets

ConfigMap operations

```
kubectl create configmap app-config --from-literal=key=value kubectl  
get configmaps  
kubectl describe configmap app-config
```

Secret operations

```
kubectl create secret generic app-secret --from-literal=key=value  
kubectl get secrets kubectl describe secret app-secret
```

Pod Management

Create and manage pods kubectl create
-f pod-definition.yaml kubectl get
pods kubectl describe pod kubectl logs
-c kubectl exec -it -c -- /bin/bash

Horizontal Pod Autoscaler (HPA) and Vertical Pod Autoscaler (VPA)

Horizontal Pod Autoscaler (HPA)

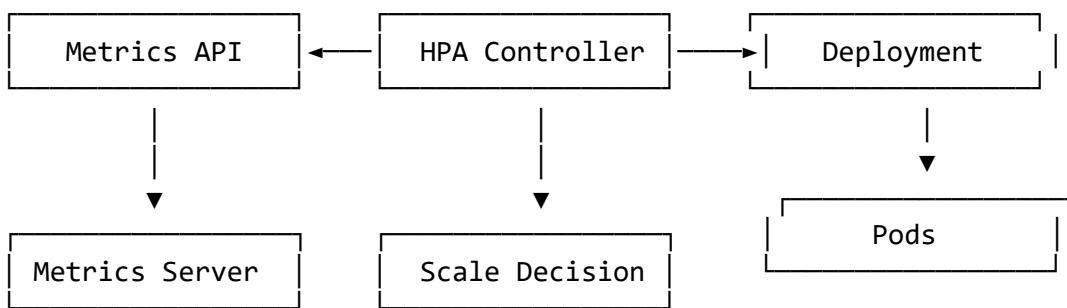
What is HPA?

Horizontal Pod Autoscaler (HPA) automatically scales the number of pods in a deployment, replica set, or stateful set based on observed CPU utilization, memory usage, or custom metrics.[1][2]

Key Concepts

- **Horizontal Scaling:** Adding/removing pod replicas
- **Metrics-based:** Decisions based on resource utilization
- **Target Utilization:** Desired resource usage percentage
- **Scale Up/Down:** Automatic pod count adjustment
- **Control Loop:** Continuous monitoring and adjustment

HPA Architecture



HPA Prerequisites

1. Metrics Server

```

# Check if metrics server is running
kubectl get pods
-n kube-system | grep metrics-server

# If not installed, deploy metrics server
kubectl apply -f https://github.com/kubernetes-
sigs/metricsserver/releases/latest/download/components.yaml

# Verify metrics are available
kubectl top nodes kubectl top
pods
  
```

2. Resource Requests

```
# Pods MUST have resource requests defined
apiVersion: apps/v1 kind: Deployment
metadata:
  name: web-app
spec: replicas:
  selector:
    matchLabels:
      app: web-app
    template:
      metadata:
        labels:
          app: web-app      spec:       containers:      - name:
web-app           image: nginx:latest           resources:
requests:          cpu: 100m      # Required for CPU-
based HPA          memory: 128Mi   # Required for memory-
based HPA          limits:       cpu: 200m      memory:
256Mi
```

Creating HPA

1. Imperative Creation

```
# CPU-based HPA (simple)
kubectl autoscale deployment web-app --cpu-percent=70 --min=2 --max=10

# View HPA kubectl get hpa
kubectl describe hpa web-app
```

2. Declarative Creation (CPU-based)

```
apiVersion: autoscaling/v2
kind: HorizontalPodAutoscaler
metadata:
  name: web-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  
behavior: scaleDown:  
  stabilizationWindowSeconds: 300  
policies: - type: Percent  
value: 10    periodSeconds: 60  
scaleUp:  
  stabilizationWindowSeconds: 0  
policies: - type: Percent  
value: 100   periodSeconds:  
15
```

3. Memory-based HPA

```
apiVersion: autoscaling/v2  
kind: HorizontalPodAutoscaler  
metadata:  
  name: memory-hpa  
spec: scaleTargetRef:  
apiVersion: apps/v1  
kind: Deployment  
name: web-app  
minReplicas: 2  
maxReplicas: 8  
metrics: - type:  
Resource    resource:  
name: memory  
target:  
  type: Utilization  
averageUtilization: 80
```

4. Multi-metric HPA

```
apiVersion: autoscaling/v2  
kind: HorizontalPodAutoscaler  
metadata:  
  name: multi-metric-hpa spec:  
  scaleTargetRef:  
apiVersion: apps/v1  
kind: Deployment
```

```
name: web-app
minReplicas: 2
maxReplicas: 15
metrics: - type:
Resource resource:
name: cpu target:
type: Utilization
averageUtilization: 70
- type:
Resource resource:
name: memory target:
type: Utilization
averageUtilization: 80
```

HPA Behavior Configuration

Scaling Policies

```
apiVersion: autoscaling/v2
kind: HorizontalPodAutoscaler
metadata: name: advanced-
hpa spec: scaleTargetRef:
apiVersion: apps/v1 kind:
Deployment name: web-app
minReplicas: 2 maxReplicas:
20 metrics: - type:
Resource resource:
name: cpu target:
type: Utilization
averageUtilization: 70
behavior: scaleDown:
stabilizationWindowSeconds: 300 # Wait 5 minutes before scaling down
policies: - type: Percent
value: 10 # Scale down by max 10% of current
# Per minute
replicas periodSeconds: 60
- type: Pods
value: 2 # Or max 2 pods per minute
periodSeconds: 60
selectPolicy: Min
pods
scaleUp:
```

```

    stabilizationWindowSeconds: 60    # Wait 1 minute before scaling up
policies:      - type: Percent
               value: 50          # Scale up by max 50% of current
replicas       periodSeconds: 60   # Per minute
               - type: Pods
                 value: 4        # Or max 4 pods per minute
periodSeconds: 60      selectPolicy: Max      # Use the policy
that adds more pods

```

Behavior Parameters

- **stabilizationWindowSeconds:** Time to wait before making scaling decisions
- **selectPolicy:**
 - Max: Use policy that allows highest scaling – Min: Use policy that allows lowest scaling
 - Disabled: Disable scaling in this direction

Custom Metrics HPA

Prerequisites for Custom Metrics

```

# Install custom metrics API server (example with Prometheus)
# 1. Deploy Prometheus
kubectl apply -f
https://raw.githubusercontent.com/prometheusoperator/prometheus-
operator/main/bundle.yaml

# 2. Deploy Prometheus Adapter
helm repo add prometheus-community
https://prometheuscommunity.github.io/helm-charts helm install
prometheus-adapter prometheus-community/prometheus-adapter

```

Custom Metrics HPA Example

```

apiVersion:   autoscaling/v2
kind:        HorizontalPodAutoscaler
metadata:
  name:  custom-metrics-hpa spec:
  scaleTargetRef:
apiVersion:  apps/v1
kind:        Deployment
name:        web-app
minReplicas: 2
maxReplicas: 10

```

```
metrics: # CPU metric
- type: Resource
resource: name:
cpu
  target:
    type: Utilization
averageUtilization: 70 # Custom
metric: requests per second
- type: Pods
pods:
metric:
  name: http_requests_per_second
target:
  type: AverageValue
averageValue: "100" # External
metric: SQS queue Length
- type:
External
external:
metric:
  name: sqs_queue_length
selector:
matchLabels:
  queue: worker-queue
target: type: Value
value: "30"
```

Object Metrics

```
apiVersion: autoscaling/v2
kind: HorizontalPodAutoscaler
metadata:
  name: object-metrics-hpa
spec: scaleTargetRef:
apiVersion: apps/v1
kind: Deployment name:
web-app minReplicas: 1
maxReplicas: 5 metrics:
- type: Object object:
metric:
  name: requests-per-second
describedObject: apiVersion:
networking.k8s.io/v1 kind:
```

```
Ingress          name: main-route
target:         type: Value
value: "2k"
```

Vertical Pod Autoscaler (VPA)

What is VPA?

Vertical Pod Autoscaler (VPA) automatically adjusts CPU and memory requests for containers in pods to optimize resource utilization and improve cluster efficiency.[1][2]

VPA vs HPA

Feature	HPA	VPA
Scaling Type	Horizontal (replica count)	Vertical (resource requests)
Pod Recreation	No	Yes (in most cases)
Use Case	Handle load spikes	Optimize resource allocation
Compatibility	Works with HPA	Can conflict with HPA

VPA Components

- **VPA Recommender:** Analyzes resource usage and provides recommendations
- **VPA Updater:** Applies recommendations by evicting pods
- **VPA Admission Controller:** Sets resource requests on new pods

Installing VPA

VPA Installation

```
# Clone VPA repository
git clone https://github.com/kubernetes/autoscaler.git cd
autoscaler/vertical-pod-autoscaler/
```

```
# Install VPA
./hack/vpa-install.sh
```

```
# Verify installation kubectl get pods -n
kube-system | grep vpa
```

Check VPA Components

```
# VPA components should be running kubectl
get pods -n kube-system
# vpa-admission-controller-xxx
```

```
# vpa-recommender-xxx  
# vpa-updater-xxx
```

VPA Modes

1. Off Mode (Recommendation Only)

```
apiVersion: autoscaling.k8s.io/v1  
kind: VerticalPodAutoscaler  
metadata:  
  name: vpa-off-mode  
spec:  
  targetRef: apiVersion: apps/v1 kind:  
Deployment name: web-app updatePolicy:  
updateMode: "Off" # Only provide recommendations
```

2. Initial Mode (Set on Pod Creation)

```
apiVersion: autoscaling.k8s.io/v1  
kind: VerticalPodAutoscaler  
metadata:  
  name: vpa-initial-mode  
spec: targetRef: apiVersion:  
apps/v1 kind: Deployment name: web-app updatePolicy:  
updateMode: "Initial" # Set requests only on pod creation
```

3. Auto Mode (Automatic Updates)

```
apiVersion: autoscaling.k8s.io/v1  
kind: VerticalPodAutoscaler  
metadata:  
  name: vpa-auto-mode  
spec: targetRef:  
apiVersion: apps/v1  
kind: Deployment  
name: web-app  
updatePolicy:  
  updateMode: "Auto" # Automatically update running pods  
resourcePolicy:  
  containerPolicies: -  
  containerName: web-app  
  minAllowed: cpu:  
    100m memory: 50Mi  
  maxAllowed: cpu:  
    1000m memory: 500Mi  
  controlledResources: ["cpu", "memory"]  
controlledValues: "RequestsAndLimits"
```

VPA Resource Policies

Comprehensive Resource Policy

```
apiVersion: autoscaling.k8s.io/v1
kind: VerticalPodAutoscaler
metadata:
  name: advanced-vpa
spec: targetRef:
apiVersion: apps/v1
kind: Deployment
name: web-app
updatePolicy:
  updateMode: "Auto"  resourcePolicy:
containerPolicies: - containerName: web-
app      minAllowed:      cpu: 100m
memory: 128Mi      maxAllowed:      cpu:
2000m      memory: 1Gi
controlledResources: ["cpu", "memory"]
controlledValues: "RequestsAndLimits"
mode: Auto
  - containerName: sidecar      mode: "Off"  #
Don't adjust sidecar container
```

Resource Policy Options

- **controlledResources:** ["cpu"], ["memory"], ["cpu", "memory"]
- **controlledValues:**
 - RequestsOnly: Only adjust requests
 - RequestsAndLimits: Adjust both requests and limits
- **mode:** Auto, Off

VPA Recommendations

Viewing VPA Recommendations

```
# Get VPA status and recommendations
kubectl describe vpa vpa-name
```

```
# View VPA recommendations in detail kubectl
get vpa vpa-name -o yaml
```

Sample VPA Output

```
status:
```

```
conditions: - status: "True"
type: RecommendationProvided
recommendation:
  containerRecommendations:
  - containerName: web-app
    lowerBound:           cpu: 50m
                           memory: 262144k
    target:              cpu: 250m
    memory:              262144k
    uncappedTarget:
      cpu: 250m          memory:
      262144k           upperBound:
      cpu: 366m          memory:
      262144k
```

HPA + VPA Combination

Safe Combination (Different Metrics)

```
# HPA for horizontal scaling based on custom metrics
apiVersion: autoscaling/v2 kind:
HorizontalPodAutoscaler metadata:
  name: custom-hpa
spec: scaleTargetRef:
apiVersion: apps/v1
kind: Deployment
name: web-app
minReplicas: 2
maxReplicas: 10
metrics: - type: Pods
pods:   metric:
  name: http_requests_per_second
target:     type: AverageValue
averageValue: "100"
---
# VPA for vertical scaling based on resource usage
apiVersion: autoscaling.k8s.io/v1 kind:
VerticalPodAutoscaler metadata:
  name: resource-vpa spec:
  targetRef:
    apiVersion: apps/v1
kind: Deployment
name: web-app
```

```
updatePolicy:  
updateMode: "Auto"
```

Avoiding Conflicts

- Don't use HPA with CPU/Memory + VPA together
- Use VPA in "Off" mode for recommendations only when using HPA
- Use different metrics for HPA (custom) and VPA (resources)

Monitoring and Observability

HPA Monitoring

```
# Watch HPA in real-time kubectl  
get hpa --watch  
  
# Check HPA events  
kubectl describe hpa web-app-hpa  
  
# View scaling events kubectl get events --sort-  
by=.metadata.creationTimestamp
```

VPA Monitoring

```
# Check VPA recommendations  
kubectl describe vpa web-app-vpa  
  
# Monitor VPA events kubectl get events | grep  
VerticalPodAutoscaler  
  
# Check resource usage kubectl  
top pods
```

Metrics to Monitor

```
# CPU and memory usage  
kubectl top nodes  
kubectl top pods  
  
# Pod resource requests vs usage kubectl describe pod  
pod-name | grep -A 10 "Requests:"
```

Troubleshooting

Common HPA Issues

```
1. HPA Not Scaling # Check metrics server
kubectl get pods -n kube-system | grep metrics-server kubectl logs -n kube-system deployment/metrics-server

# Check resource requests kubectl describe deployment web-app | grep -A 5 "Requests:"

# Check HPA status kubectl describe hpa web-app-hpa
2. Unable to get Resource Metrics
# Verify metrics API kubectl get apiservice v1beta1.metrics.k8s.io -o yaml

# Check metrics server logs kubectl logs -n kube-system -l k8s-app=metrics-server

# Test metrics availability
kubectl top nodes kubectl top pods

3. Thrashing (Rapid Scaling) #
Add stabilization windows
behavior: scaleDown:
  stabilizationWindowSeconds: 300
scaleUp:
stabilizationWindowSeconds: 60
```

Common VPA Issues

```
1. VPA Not Updating Pods # Check VPA components kubectl get pods -n kube-system | grep vpa

# Check VPA status
kubectl describe vpa web-app-vpa

# Verify admission controller kubectl logs -n kube-system -l app=vpa-admission-controller

2. Pods Being Evicted Too Often
# Use "Initial" mode instead of "Auto" updatePolicy:
updateMode: "Initial"
```

```
# Or use "Off" mode for recommendations only
updatePolicy: updateMode: "Off"
```

3. Resource Recommendations Too High/Low

```
# Set resource limits in VPA
resourcePolicy:
  containerPolicies: -
    containerName: web-app
    minAllowed:     cpu: 100m
    memory: 128Mi
    maxAllowed:
      cpu:       1000m
    memory: 512Mi
```

Real-World Examples

E-commerce Application Scaling

```
# Deployment
apiVersion: apps/v1
kind: Deployment
metadata:
  name: ecommerce-app spec:
replicas: 3 selector:
matchLabels:   app:
ecommerce template:
metadata:   labels:
app: ecommerce   spec:
containers:   - name:
  web-server   image:
  nginx:latest
resources:
requests:           cpu:
  200m           memory:
  256Mi          limits:
cpu: 500m
memory: 512Mi      - name:
  app-server   image:
  myapp:latest
resources:
requests:           cpu:
  300m           memory:
```

```
512Mi           limits:
cpu: 1000m
memory: 1Gi
---
# HPA for handling traffic spikes
apiVersion: autoscaling/v2 kind:
HorizontalPodAutoscaler metadata:
  name: ecommerce-hpa
spec: scaleTargetRef:
apiVersion: apps/v1
kind: Deployment
name: ecommerce-app
minReplicas: 3
maxReplicas: 20
metrics: - type:
Resource resource:
  name: cpu      target:
  type: Utilization
  averageUtilization: 70
  - type: Pods
pods:
metric:
  name: http_requests_per_second
target: type: AverageValue
averageValue: "200" behavior:
scaleDown:
  stabilizationWindowSeconds: 300
scaleUp:
  stabilizationWindowSeconds: 60
---
# VPA for resource optimization (recommendations only) apiVersion:
autoscaling.k8s.io/v1 kind: VerticalPodAutoscaler metadata: name:
ecommerce-vpa spec: targetRef: apiVersion: apps/v1 kind:
Deployment name: ecommerce-app updatePolicy: updateMode: "Off"
# Only recommendations to avoid conflicts with HPA
```

Batch Processing Application

```
# For batch processing, VPA is more appropriate than HPA
apiVersion: autoscaling.k8s.io/v1 kind:
VerticalPodAutoscaler metadata:
  name: batch-processor-vpa
spec: targetRef:
```

```

apiVersion: apps/v1
kind: Deployment      name:
batch-processor
updatePolicy:
  updateMode: "Auto"
resourcePolicy:
containerPolicies:
- containerName:
  processor
minAllowed:
cpu: 500m
memory: 1Gi
maxAllowed:
cpu: 4000m
memory: 8Gi
controlledResources:
["cpu", "memory"]
controlledValues:
"RequestsAndLimits"

```

CKA Exam Scenarios

Scenario 1: Create HPA for Deployment

```

# Given: deployment 'web-app' with resource requests
# Task: Create HPA to scale between 2-8 replicas at 80% CPU

```

```

# Solution:
kubectl autoscale deployment web-app --cpu-percent=80 --min=2 --max=8

# Verify:
kubectl get hpa kubectl
describe hpa web-app

```

Scenario 2: Multi-metric HPA with Custom Behavior

```

# Task: Create HPA with CPU and memory metrics, custom scaling behavior
apiVersion: autoscaling/v2 kind: HorizontalPodAutoscaler metadata:
name: advanced-hpa  namespace: production spec:  scaleTargetRef:
apiVersion: apps/v1    kind: Deployment   name: api-server
minReplicas: 3  maxReplicas: 15  metrics: - type: Resource
resource:      name: cpu      target:
  type: Utilization
averageUtilization: 70

```

```
- type: Resource
resource:
  name: memory
target:      type:
Utilization
averageUtilization: 80
behavior:    scaleDown:
  stabilizationWindowSeconds: 300
policies:    - type: Percent
value: 25      periodSeconds: 60
scaleUp:
  stabilizationWindowSeconds: 60
policies:    - type: Pods
value: 3       periodSeconds: 60
```

Scenario 3: VPA for Resource Optimization

```
# Task: Create VPA to optimize resources for 'database' deployment
```

```
# Solution:
kubectl apply -f - <<EOF apiVersion:
autoscaling.k8s.io/v1 kind:
VerticalPodAutoscaler metadata:
  name: database-vpa
spec:  targetRef:
apiVersion: apps/v1
kind: Deployment
name: database
updatePolicy:
  updateMode: "Auto"
resourcePolicy:
  containerPolicies:  -
    containerName: mysql
    minAllowed:      cpu:
      500m          memory: 1Gi
    maxAllowed:      cpu:
      2000m         memory: 4Gi
EOF
```

```
# Check recommendations:
kubectl describe vpa database-vpa
```

Best Practices

HPA Best Practices

1. **Always set resource requests** on containers
2. **Use stabilization windows** to prevent thrashing
3. **Monitor scaling events** and adjust policies
4. **Test scaling behavior** under load
5. **Use multiple metrics** for better decision making
6. **Set appropriate min/max replicas**

VPA Best Practices

1. **Start with “Off” mode** to analyze recommendations
2. **Set resource limits** to prevent runaway resource allocation
3. **Use “Initial” mode** for predictable workloads
4. **Monitor eviction frequency** in “Auto” mode
5. **Avoid VPA + HPA conflicts** on same resources
6. **Review recommendations regularly**

General Autoscaling Best Practices

1. **Monitor resource utilization** patterns
2. **Set up proper monitoring** and alerting
3. **Test autoscaling** in non-production environments
4. **Document scaling policies** and decisions
5. **Regular review** of autoscaling configurations
6. **Capacity planning** for maximum scale scenarios

Commands Summary

HPA Commands

```
# Create HPA kubectl autoscale deployment NAME --cpu-percent=70 --
min=2 --max=10
```

```
# Manage HPA kubectl get
hpa kubectl describe hpa
NAME kubectl delete hpa
NAME
```

```
# Monitor scaling kubectl get hpa --watch kubectl top
pods kubectl get events --sort-
by=.metadata.creationTimestamp
```

VPA Commands

```
# Install VPA (if needed)
git clone https://github.com/kubernetes/autoscaler.git cd
autoscaler/vertical-pod-autoscaler/
./hack/vpa-install.sh

# Manage VPA kubectl
get vpa kubectl
describe vpa NAME
kubectl delete vpa
NAME

# Monitor VPA kubectl get
vpa NAME -o yaml kubectl
describe vpa NAME
```

Metrics and Debugging

```
# Check metrics server
kubectl get pods -n kube-system | grep metrics-server
kubectl top nodes kubectl top pods

# Debug autoscaling kubectl
describe hpa NAME kubectl
describe vpa NAME kubectl
get events | grep -E
"(HorizontalPodAutoscaler|VerticalPodAutoscaler)"
```

Key Points for CKA Exam

Must Know Concepts

1. **HPA scales replicas** horizontally based on metrics
2. **VPA adjusts resource requests** vertically
3. **Resource requests are required** for HPA/VPA to work
4. **Metrics server is prerequisite** for resource-based scaling
5. **Behavior policies control** scaling rate and timing

Common Exam Tasks

- Create HPA for deployments
- Configure multi-metric HPA
- Set up VPA for resource optimization

- Troubleshoot scaling issues
- Monitor autoscaling behavior

Critical Configuration

- **minReplicas/maxReplicas** for HPA
- **updateMode** for VPA (Off/Initial/Auto)
- **Resource requests** in deployments
- **Stabilization windows** for smooth scaling
- **Resource policies** for VPA boundaries

Security

Security Primitives

What is Kubernetes Security?

Kubernetes Security encompasses the protection of cluster infrastructure, applications, and data through multiple layers of defense including authentication, authorization, network policies, and encryption.

Core Security Components

- **Authentication:** Verifying identity of users and services
- **Authorization:** Determining what authenticated entities can do
- **Admission Control:** Validating and potentially modifying requests
- **Network Security:** Controlling traffic flow between components
- **Secrets Management:** Protecting sensitive data

Authentication Mechanisms

What is Authentication?

Authentication is the process of verifying the identity of users, service accounts, or external systems attempting to access the Kubernetes cluster.

Authentication Methods

1. Static Token Files

```
# Create user-details.csv password123,user1,u0001,group1  
password123,user2,u0002,group1  
password123,user3,u0003,group2
```

```
# Configure API server
--token-auth-file=user-details.csv
```

2. X.509 Client Certificates

```
# Generate user certificate openssl genrsa -out user.key 2048 openssl req -new -key user.key -out user.csr -subj "/CN=john/O=developers" openssl x509 -req -in user.csr -CA ca.crt -CAkey ca.key -out user.crt
```

3. Service Account Tokens

- **Automatic:** Created for each namespace
- **JWT-based:** JSON Web Tokens for pod authentication
- **Projection:** Mounted into pods automatically

API Server Configuration

```
# API Server Authentication Flags
--client-ca-file=/var/lib/kubernetes/ca.pem
--tls-cert-file=/var/lib/kubernetes/apiserver.crt
--tls-private-key-file=/var/lib/kubernetes/apiserver.key
--service-account-key-file=/var/lib/kubernetes/service-account.pem
```

TLS Certificates

What are TLS Certificates?

TLS Certificates provide cryptographic proof of identity and enable encrypted communication between Kubernetes components.

Certificate Types

Server Certificates

- **API Server:** apiserver.crt - Serves HTTPS API endpoints
- **ETCD Server:** etcdserver.crt - Encrypts cluster data storage
- **Kubelet:** kubelet.crt - Secures node agent communication

Client Certificates

- **Admin Users:** admin.crt - Administrator access to cluster
- **Scheduler:** scheduler.crt - Scheduler to API server communication
- **Controller Manager:** controller-manager.crt - Controller authentication
- **Kube-proxy:** kube-proxy.crt - Network proxy authentication

Certificate Generation Process

```
# 1. Generate CA Key and Certificate openssl
genrsa -out ca.key 2048
openssl req -new -x509 -key ca.key -subj "/CN=KUBERNETES-CA" -out ca.crt days
10000

# 2. Generate Server Certificate (API Server) openssl
genrsa -out apiserver.key 2048
openssl req -new -key apiserver.key -subj "/CN=kube-apiserver" -out
apiserver.csr -config openssl.cnf
openssl x509 -req -in apiserver.csr -CA ca.crt -CAkey ca.key -out
apiserver.crt -days 365

# 3. Generate Client Certificate (Admin) openssl
genrsa -out admin.key 2048
openssl req -new -key admin.key -subj "/CN=kube-admin/O=system:masters" -out
admin.csr openssl x509 -req -in admin.csr -CA ca.crt -CAkey ca.key -out
admin.crt -days
365
```

Certificate Requirements

```
# Subject Alternative Names for API Server
DNS.1 = kubernetes
DNS.2 = kubernetes.default
DNS.3 = kubernetes.default.svc
DNS.4 = kubernetes.default.svc.cluster.local
IP.1 = 10.96.0.1
IP.2 = 172.17.0.87
```

Certificate Inspection

```
# View certificate details openssl x509 -in
/etc/kubernetes/pki/apiserver.crt -text -noout

# Check certificate expiration openssl x509
-in apiserver.crt -noout -dates
```

Authorization

What is Authorization?

Authorization determines what actions an authenticated user or service account can perform within the Kubernetes cluster.[3][4]

Authorization Modes

1. Node Authorization

- **Purpose:** Authorizes requests from kubelets
- **Scope:** Limited to node-specific resources
- **Certificate Group:** system:nodes

2. ABAC (Attribute-Based Access Control)

```
{"apiVersion": "abac.authorization.kubernetes.io/v1beta1", "kind": "Policy",  
"spec": {"user": "alice", "namespace": "projectCaribou", "resource": "pods",  
"readonly": true}}
```

3. RBAC (Role-Based Access Control)

Most recommended authorization method for production clusters.[4][3]

4. Webhook Mode

- **External:** Delegates authorization to external services
- **Custom Logic:** Allows complex authorization rules
- **Integration:** Works with external identity providers

API Server Authorization Configuration

```
# Enable multiple authorization modes  
--authorization-mode=Node,RBAC,Webhook
```

Role-Based Access Control (RBAC)

What is RBAC?

RBAC regulates access to Kubernetes resources based on the roles assigned to users, groups, or service accounts.[3][4]

RBAC Components

1. Role (Namespace-scoped)

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

2. ClusterRole (Cluster-wide)

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

3. RoleBinding (Namespace-scoped)

```
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
```

4. ClusterRoleBinding (Cluster-wide)

```
apiVersion: rbac.authorization.k8s.io/v1
kind: ClusterRoleBinding metadata:
  name: read-secrets-global subjects:
- kind: User
  name: manager
  apiGroup: rbac.authorization.k8s.io
  roleRef:
    kind: ClusterRole
    name: secret-reader
    apiGroup: rbac.authorization.k8s.io
```

RBAC Verbs

- **get:** Retrieve specific resource
- **list:** List resources of a type
- **watch:** Watch for resource changes
- **create:** Create new resources
- **update:** Modify existing resources
- **patch:** Partially update resources
- **delete:** Remove resources

Resource Names (Specific Resource Access)

```
rules:
- apiGroups: [""]
  resources: ["pods"]
  verbs: ["get", "update"]
  resourceNames:
  ["blue-pod", "green-pod"]
```

RBAC Management Commands

```
# Check user permissions
kubectl auth can-i create deployments --as=dev-user
kubectl auth can-i delete nodes --as=dev-user --namespace=development

# View RBAC resources kubectl
get roles,rolebindings
kubectl get clusterroles,clusterrolebindings
kubectl describe role pod-reader kubectl
describe rolebinding read-pods
```

KubeConfig Files

What is KubeConfig?

KubeConfig files contain cluster access information including clusters, users, and contexts for kubectl authentication.[2][1]

KubeConfig Structure

```
apiVersion: v1 kind: Config current-context:
dev-user@development-cluster clusters:
- name: development-cluster   cluster:
certificate-authority: /path/to/ca.crt
server: https://dev-cluster:6443 - name:
production-cluster   cluster:
certificate-authority: /path/to/ca.crt
server: https://prod-cluster:6443
contexts:
- name: dev-user@development-cluster
context:   cluster: development-cluster
user: dev-user   namespace: development -
name: admin@production-cluster   context:
cluster: production-cluster   user:
admin users:
- name: dev-user   user:   client-
certificate: /path/to/dev-user.crt
client-key: /path/to/dev-user.key
- name: admin   user:   client-
certificate: /path/to/admin.crt
client-key: /path/to/admin.key
```

Certificate Data Encoding

```
# Instead of file paths, embed base64-encoded data
clusters: - name: my-cluster   cluster:
  certificate-authority-data: LS0tLS1CRUdJTi...
    server: https://cluster:6443 users: - name:
  my-user   user:     client-certificate-data:
  LS0tLS1CRUdJTi...     client-key-data:
  LS0tLS1CRUdJTi...
```

What are Contexts?

Contexts combine cluster, user, and namespace information into a named configuration that allows easy switching between different Kubernetes environments.[1][2]

Context Components

- **Cluster:** Which Kubernetes cluster to connect to
- **User:** Which user credentials to use for authentication
- **Namespace:** Default namespace for kubectl commands (optional)

Context Structure

```
contexts:
-   name: dev-user@development-cluster   context:
  cluster: development-cluster   # Which cluster to
  use   user: dev-user   # Which user
  credentials to use   namespace: development
  # Default namespace (optional)
-   name: admin@production-cluster   context:
  cluster: production-cluster
  user: admin   namespace:
  production
```

Context Management Commands

```
# View all contexts
kubectl config get-contexts

# View current context
kubectl config current-context

# Switch to a different context
kubectl config use-context dev-user@development-cluster

# Switch to production context
kubectl config use-context admin@production-cluster
```

```
# View detailed configuration kubectl  
config view  
  
# View only current context details kubectl  
config view --minify
```

Creating and Modifying Contexts

```
# Create a new context kubectl config  
set-context my-context \  
--cluster=my-cluster \  
--user=my-user \  
--namespace=my-namespace  
  
# Modify existing context namespace kubectl config set-context dev-  
user@development-cluster --namespace=testing
```

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

```
# Create context with cluster and user kubectl  
config set-context staging-context \  
--cluster=staging-cluster \  
--user=staging-user
```

Context Examples

```
# Development environment context kubectl  
config set-context development \  
--cluster=dev-cluster \  
--user=dev-user \  
--namespace=development
```

```
# Production environment context kubectl  
config set-context production \  
--cluster=prod-cluster \  
--user=admin-user \  
--namespace=production
```

```
# Testing environment context kubectl  
config set-context testing \  
--cluster=test-cluster \  
--user=test-user
```

```
--user=test-user \
--namespace=testing
```

Context Best Practices

- **Descriptive naming:** Use clear context names like dev-user@development
- **Environment separation:** Different contexts for dev, staging, production
- **Namespace defaults:** Set appropriate default namespaces
- **Regular cleanup:** Remove unused contexts periodically
- **Backup configs:** Keep backup copies of important kubeconfig files

Context Switching Workflows

```
# Daily workflow example # Start with
development kubectl config use-context
development
kubectl get pods # Shows pods in development namespace
```

```
# Switch to production for deployment kubectl
config use-context production kubectl apply -
f production-deployment.yaml
```

```
# Quick check in staging kubectl
config use-context staging kubectl
get services
```

```
# Back to development
kubectl config use-context development
```

Advanced Context Operations

```
# Delete a context kubectl config delete-
context old-context
```

```
# Rename context (delete and recreate) kubectl
config delete-context old-name
kubectl config set-context new-name --cluster=cluster --user=user
```

```
# Export specific context
kubectl config view --context=production --minify --flatten > prodconfig.yaml
```

```
# Use temporary context without switching
kubectl --context=production get pods kubectl -
-context=development apply -f app.yaml
```

Multi-Cluster Context Setup

```
# Example: Managing multiple environments
apiVersion: v1
kind: Config
current-context: development

# Development cluster
clusters: - name: dev-cluster
  cluster:
    server: https://dev-api.company.com:6443
    certificate-authority-data: LS0tLS...

# Staging cluster
- name: staging-cluster
  cluster:
    server: https://staging-api.company.com:6443
    certificate-authority-data: LS0tLS...

# Production cluster -
name: prod-cluster
cluster:
  server: https://prod-api.company.com:6443
  certificate-authority-data: LS0tLS...

# Users for different environments
users: - name: dev-user
  user:
    client-certificate-data: LS0tLS...
    client-key-data: LS0tLS... - name: staging-user
    user:
      client-certificate-data: LS0tLS...
      client-key-data: LS0tLS...
    - name: prod-admin
      user:
        client-certificate-data: LS0tLS...
        client-key-data: LS0tLS...

# Contexts combining clusters and users
contexts: - name: development
  context:
    cluster: dev-cluster
    user: dev-user
    namespace: development - name: staging
    context:
      cluster: staging-cluster
      user: staging-user
      namespace: staging - name: production
    context:
      cluster: prod-cluster
      user: prod-admin
      namespace: production
```

Context Security Considerations

- **User isolation:** Use different users for different environments
- **Namespace boundaries:** Set appropriate default namespaces
- **Access control:** Ensure users only have access to appropriate clusters
- **Audit trails:** Context switches can be logged for security auditing
- **Credential management:** Rotate certificates and tokens regularly

Troubleshooting Context Issues

```
# Check context configuration kubectl config view  
--context=problematic-context

# Verify cluster connectivity  
kubectl --context=production cluster-info

# Test authentication kubectl --  
context=development auth can-i get pods

# Debug certificate issues kubectl  
config view --raw -o  
jsonpath='{.contexts[?(@.name=="production")].context}'

# Check current context settings  
kubectl config get-contexts $(kubectl config current-context)
```

Image Security

What is Image Security?

Image Security involves securing container images and controlling access to private registries.[1][2]

Private Registry Authentication

```
# Login to private registry  
docker login private-registry.io  
Username: registry-user  
Password: registry-password

# Create secret for registry authentication kubectl  
create secret docker-registry regcred \  
--docker-server=private-registry.io \  
--docker-username=registry-  
user \  
--docker-password=registry-password \  

```

```
--docker-email=user@company.com
```

Using Image Pull Secrets

```
apiVersion: v1 kind:  
Pod metadata:  
  name: private-app spec: containers: - name:  
    app image: private-  
    registry.io/apps/internal-app  
  imagePullSecrets: - name: regcred
```

Image Security Best Practices

- **Scan images** for vulnerabilities before deployment
- **Use specific tags** instead of latest
- **Minimize base images** to reduce attack surface
- **Regular updates** of base images and dependencies

Security Contexts

What are Security Contexts?

Security Contexts define privilege and access control settings for pods and containers.[2][1]

Pod Security Context

```
apiVersion: v1  
kind: Pod  
metadata:  
  name: security-context-demo spec:  
    securityContext: runAsUser: 1000  
    runAsGroup: 3000 fsGroup: 2000  
    runAsNonRoot: true containers: -  
      name: sec-ctx-demo image: busybox  
    securityContext:  
      allowPrivilegeEscalation: false  
      runAsUser: 2000 capabilities:  
      add: ["NET_ADMIN", "SYS_TIME"]  
      drop: ["ALL"]  
      readOnlyRootFilesystem: true
```

Security Context Options

- **runAsUser:** User ID for running containers
- **runAsGroup:** Primary group ID for containers

- **runAsNonRoot**: Prevents running as root user
- **fsGroup**: Group ownership for mounted volumes
- **allowPrivilegeEscalation**: Controls privilege escalation
- **capabilities**: Linux capabilities to add/drop
- **readOnlyRootFilesystem**: Makes root filesystem read-only

Network Policies

What are Network Policies?

Network Policies control network traffic flow between pods, providing microsegmentation at the application layer.[1][2]

Default Behavior

- **All Allow**: By default, all pods can communicate with all other pods
- **No Isolation**: No network restrictions until policies are applied

Network Policy Components

- **podSelector**: Which pods the policy applies to
- **policyTypes**: Ingress, Egress, or both
- **ingress**: Rules for incoming traffic
- **egress**: Rules for outgoing traffic

Basic Network Policy Example

```
apiVersion: networking.k8s.io/v1
kind: NetworkPolicy
metadata:
  name: test-network-policy
  namespace: default
spec:
  podSelector:
    matchLabels: role: db
  policyTypes: - Ingress - Egress
  ingress:
  - from:
    - ipBlock:
        cidr: 172.17.0.0/16
  except:
  - 172.17.1.0/24
  - namespaceSelector:
    matchLabels:
```

```
name: prod      -
podSelector:
matchLabels:
role: frontend
ports:      - protocol:
TCP        port: 6379
egress:
-   to:      -
ipBlock:      cidr:
10.0.0.0/8    ports:
- protocol: TCP
port: 5978
```

Network Policy Selectors

Pod Selector ingress:

```
-   from:      -
podSelector:
matchLabels:
app: frontend
```

Namespace Selector ingress:

```
-   from:      -
namespaceSelector:
matchLabels:
name: production
IP Block ingress:
-   from:      -
ipBlock:
  cidr: 192.168.1.0/24
except:      -
192.168.1.5/32
```

Deny All Policy

```
apiVersion: networking.k8s.io/v1
kind: NetworkPolicy metadata:
  name: deny-all
spec:
  podSelector: {}
  policyTypes:      -
    Ingress
    - Egress
```

CNI Plugin Requirements

Network Policies require a compatible CNI plugin:

- **Calico**: Full network policy support
- **Weave**: Supports network policies
- **Flannel**: Does **NOT** support network policies

ConfigMaps and Secrets Security

What are ConfigMaps?

ConfigMaps store non-confidential configuration data in key-value pairs that can be consumed by pods.[2]

What are Secrets?

Secrets store sensitive information such as passwords, tokens, and keys with additional security measures.[2]

Security Differences

Feature	ConfigMap	Secret
Data Type	Non-sensitive configuration	Sensitive information
Encoding	Plain text	Base64 encoded
Storage	etcd (plain text)	etcd (can be encrypted at rest)
Access	Standard RBAC	Enhanced RBAC controls

Secret Types

- **Opaque**: General-purpose secrets (default)
- **kubernetes.io/service-account-token**: Service account tokens
- **kubernetes.io/dockercfg**: Docker registry authentication
- **kubernetes.io/tls**: TLS certificates and keys

Creating Secrets

```
# Imperative creation kubectl create secret  
generic app-secret \  
--from-literal=DB_Host=mysql \  
--from-literal=DB_User=root \  
--from-literal=DB_Password=password
```

```
# Declarative creation
apiVersion: v1
kind: Secret
metadata:
  name: app-secret
type: Opaque
data:
  DB_Host: bXlzcWw=      # mysql (base64)
  DB_User: cm9vdA==       # root (base64)
  DB_Password: cGFzc3dvcmQ= # password (base64)
```

Using Secrets in Pods

```
apiVersion: v1
kind: Pod
metadata:
  name: secret-pod
spec:
  containers:
    - name: app
      image: myapp
      env:
        - name: DB_PASSWORD
      valueFrom:
        secretKeyRef:
          name: app-secret
          key: DB_Password
      volumeMounts:
        - name: secret-volume
          mountPath: /etc/secrets
      volumes:
        - name: secret-volume
          secret:
            secretName: app-secret
```

ConfigMap Security Best Practices

- **Separate sensitive data:** Never store passwords in ConfigMaps
- **Use RBAC:** Control access to ConfigMaps
- **Namespace isolation:** Keep ConfigMaps in appropriate namespaces
- **Regular audits:** Review ConfigMap contents regularly

Secret Security Best Practices

- **Encryption at rest:** Enable etcd encryption
- **RBAC controls:** Strict access controls for secrets

- **Secret rotation:** Regular rotation of sensitive data
- **Avoid logging:** Prevent secrets from appearing in logs
- **Image security:** Don't embed secrets in container images

Certificate API

What is the Certificates API?

The **Certificates API** allows you to provision TLS certificates signed by a Certificate Authority (CA) that you control.[1][2]

Certificate Signing Request (CSR)

```
apiVersion: certificates.k8s.io/v1      kind:  
CertificateSigningRequest metadata: name: john-  
developer spec: request: LS0tLS1CRUdJTi... #  
base64 encoded CSR signerName:  
kubernetes.io/kube-apiserver-client usages: -  
client auth
```

CSR Workflow

```
# 1. Generate private key  
openssl genrsa -out john.key 2048  
  
# 2. Create certificate signing request openssl req -new -key john.key -  
subj "/CN=john/O=developers" -out john.csr  
  
# 3. Submit CSR to Kubernetes cat  
john.csr | base64 | tr -d '\n'  
  
# 4. Create CSR object kubectl  
apply -f john-csr.yaml  
  
# 5. Approve CSR kubectl certificate  
approve john-developer  
  
# 6. Get certificate  
kubectl get csr john-developer -o jsonpath='{.status.certificate}' | base64 d  
> john.crt
```

Controller Manager Configuration

```
# Controller manager handles certificate signing
```

```
--cluster-signing-cert-file=/etc/kubernetes/pki/ca.crt  
--cluster-signing-key-file=/etc/kubernetes/pki/ca.key
```

Security Best Practices

Cluster Security

- **Regular updates:** Keep Kubernetes and components updated
- **Network segmentation:** Use network policies for micro-segmentation
- **Least privilege:** Apply minimal necessary permissions
- **Audit logging:** Enable comprehensive audit logging
- **Resource limits:** Set appropriate resource quotas and limits

Authentication Security

- **Strong certificates:** Use proper key lengths and algorithms
- **Certificate rotation:** Regular rotation of certificates
- **Multi-factor authentication:** Where possible, implement MFA
- **Service accounts:** Use dedicated service accounts for applications

Authorization Security

- **RBAC enforcement:** Use RBAC over other authorization modes
- **Regular reviews:** Audit permissions regularly
- **Namespace isolation:** Use namespaces for logical separation
- **Avoid wildcards:** Be specific in permission grants

Secret Management

- **External secret management:** Consider tools like HashiCorp Vault
- **Encryption at rest:** Enable etcd encryption
- **Secret scanning:** Scan for secrets in code and images
- **Rotation policies:** Implement automated secret rotation

Troubleshooting Security Issues

Common Authentication Problems

```
# Check API server configuration  
kubectl get pods -n kube-system  
kubectl logs kube-apiserver-master -n kube-system
```

```
# Verify certificate details  
openssl x509 -in client.crt -text -noout
```

```
# Test authentication
kubectl auth can-i get pods --as=system:serviceaccount=default:default
```

Common Authorization Problems

```
# Check RBAC configuration kubectl get
roles,rolebindings -A kubectl describe
role developer kubectl describe
rolebinding dev-binding
```

Test permissions

```
kubectl auth can-i create deployments --as=dev-user kubectl
auth can-i list secrets --as=dev-user -n production
```

Network Policy Debugging

```
# Check if network policies are applied kubectl
get networkpolicies
kubectl describe networkpolicy deny-all
```

Test connectivity

```
kubectl exec -it test-pod -- nc -zv target-service 80
kubectl exec -it test-pod -- nslookup target-service
```

Key Commands Summary

Authentication & Certificates

```
# Certificate operations openssl genrsa -out user.key 2048 openssl req
-new -key user.key -out user.csr -subj "/CN=user" openssl x509 -req -
-in user.csr -CA ca.crt -CAkey ca.key -out user.crt
```

```
# CSR management kubectl get csr
kubectl certificate approve user-csr
kubectl certificate deny user-csr
```

RBAC Management

```
# Create RBAC resources
kubectl create role developer --verb=get,list,create --resource=pods kubectl
create rolebinding dev-binding --role=developer --user=dev-user kubectl
create clusterrole cluster-reader --verb=get,list --resource=nodes kubectl
create clusterrolebinding cluster-reader-binding -clusterrole=cluster-
reader --user=admin
```

Secrets and ConfigMaps

```
# Secret operations
kubectl create secret generic app-secret --from-literal=key=value kubectl
get secrets
kubectl describe secret app-secret

# ConfigMap operations
kubectl create configmap app-config --from-literal=key=value kubectl
get configmaps
kubectl describe configmap app-config

Network Policies
# Network policy management kubectl
apply -f network-policy.yaml kubectl
get networkpolicies kubectl describe
networkpolicy deny-all
```

Service Accounts

What are Service Accounts?

Service Accounts are special types of accounts in Kubernetes that provide an identity for processes running in pods. Unlike user accounts (which are for humans), service accounts are designed for applications and system components.[1][2]

Key Characteristics

- **Pod Identity:** Every pod runs with a service account
- **API Access:** Service accounts enable pods to interact with the Kubernetes API
- **Namespace Scoped:** Each service account belongs to a specific namespace
- **Automatic Creation:** Every namespace gets a default service account
- **Token-Based:** Authentication uses JWT tokens

Default Service Account

Automatic Behavior

```
# Every namespace gets a default service account
kubectl get serviceaccounts kubectl get sa #  
Short form  
  
# Output shows:
```

```
# NAME      SECRETS   AGE #
default    1          5d
```

Default Service Account Properties

- **Name:** default
- **Automatic Mounting:** Token automatically mounted to all pods
- **Limited Permissions:** Can only access basic API discovery endpoints
- **Location:** /var/run/secrets/kubernetes.io/serviceaccount/

Default Token Structure

```
# Inside a pod, check mounted service account ls
/var/run/secrets/kubernetes.io/serviceaccount/
# ca.crt    namespace    token

# View the token cat
/var/run/secrets/kubernetes.io/serviceaccount/token #
View the namespace cat
/var/run/secrets/kubernetes.io/serviceaccount/namespac
e
```

Creating Service Accounts

Imperative Creation

```
# Create a service account
kubectl create serviceaccount my-service-account kubectl
create sa my-app-sa # Short form

# Create in specific namespace kubectl create
serviceaccount my-sa -n development

# List service accounts kubectl
get serviceaccounts -A kubectl
get sa --all-namespaces
```

Declarative Creation

```
apiVersion: v1 kind: ServiceAccount metadata: name:
my-app-service-account namespace: production
labels: app: my-application tier: backend
automountServiceAccountToken: true # Default: true
```

Service Account with Secrets

```
apiVersion: v1 kind:  
ServiceAccount metadata:  
  name: build-robot  
secrets:  
- name: build-robot-secret imagePullSecrets:  
- name: private-registry-secret
```

Service Account Tokens

What are Service Account Tokens?

Service Account Tokens are JWT (JSON Web Token) credentials that pods use to authenticate with the Kubernetes API server.[2][1]

Token Types

1. Automatic Tokens (Legacy)

```
# Kubernetes < 1.24: Automatic secret creation kubectl  
create serviceaccount token-test kubectl get secrets  
# Shows: token-test-token-xxxxx  
  
# View token secret kubectl describe secret  
token-test-token-xxxxx
```

```
2. Projected Tokens (Current) apiVersion: v1 kind: Pod  
  metadata:  
    name: nginx spec: serviceAccountName:  
    my-service-account    containers: -  
      image: nginx        name: nginx  
      volumeMounts:  
        - mountPath:  
          /var/run/secrets/kubernetes.io/serviceaccount  
          name: kube-api-access-xxxxx      readOnly: true  
          volumes:  
            - name: kube-api-access-xxxxx      projected:  
              sources:  
                - serviceAccountToken:  
                  expirationSeconds: 3607  
path: token      - configMap:  
items:           - key: ca.crt  
path: ca.crt      name:  
kube-root-ca.crt -  
downwardAPI:
```

```
        items:
- fieldRef:
      apiVersion: v1
      fieldPath: metadata.namespace
path: namespace

Manual Token Creation (Kubernetes 1.24+)
# Create long-lived token secret kubectl
create token my-service-account

# Create token with expiration kubectl create token
my-service-account --duration=1h

# Create token for specific audience kubectl create token my-service-account
--audience=https://my-api.example.com

# Create permanent token secret (legacy style)
kubectl apply -f - <<EOF
apiVersion: v1 kind:
Secret metadata:
  name: my-sa-token annotations:
  kubernetes.io/service-account.name: my-service-account
  type: kubernetes.io/service-account-token EOF
```

Using Service Accounts in Pods

Specifying Service Account

```
apiVersion: v1 kind: Pod metadata: name:
my-pod spec: serviceAccountName: my-
service-account containers: - name:
app image: my-app:latest
```

Deployment with Service Account

```
apiVersion: apps/v1 kind:
Deployment metadata: name:
my-app spec:
  replicas: 3
  selector:
    matchLabels:
      app: my-app
  template:
```

```
metadata:  
labels:  
    app: my-app  
spec:  
    serviceAccountName: my-app-service-account  
containers: - name: app           image: my-  
app:latest
```

Disabling Automatic Token Mounting

```
apiVersion: v1  
kind: Pod metadata:  
    name: no-token-pod spec:  
    serviceAccountName: my-service-account  automountServiceAccountToken: false  
containers: - name: app     image: my-app:latest  
  
# Disable at service account level  
apiVersion: v1 kind: ServiceAccount  
metadata: name: no-auto-mount-sa  
automountServiceAccountToken: false
```

Service Accounts and RBAC

What is Service Account RBAC?

Service accounts integrate with Kubernetes RBAC to define what API operations pods can perform.[3][4]

Creating Role for Service Account

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

Binding Service Account to Role

```
apiVersion: rbac.authorization.k8s.io/v1  
kind: RoleBinding metadata:
```

```

name: read-pods
namespace: development
subjects:
- kind: ServiceAccount
  name: my-app-service-account
  namespace: development
roleRef:
  kind: Role
  name: pod-reader
  apiGroup: rbac.authorization.k8s.io
ClusterRole for Service Account
apiVersion: rbac.authorization.k8s.io/v1
kind: ClusterRole metadata:
  name: node-reader rules: -
apiGroups: [""]
resources:
["nodes"]
verbs: ["get",
"watch", "list"]

apiVersion: rbac.authorization.k8s.io/v1
kind: ClusterRoleBinding metadata:
  name: read-nodes-global subjects:
- kind: ServiceAccount
  name: node-reader-sa
  namespace: monitoring roleRef: kind:
  ClusterRole
  name: node-reader
  apiGroup:
  rbac.authorization.k8s.io

```

Service Account Subject Format

```

# In RoleBinding/ClusterRoleBinding
subjects:
- kind: ServiceAccount
  name: service-
  account-name
  namespace: namespace-name # Required for service accounts

```

Advanced Service Account Usage**Multiple Service Accounts Example**

```

# Development service account
apiVersion: v1 kind:
ServiceAccount metadata:
  name: dev-app-sa
  namespace: development
---
```

```
# Production service account
apiVersion: v1 kind:
ServiceAccount metadata:
  name: prod-app-sa
  namespace: production
---
# Different permissions for each environment
apiVersion: rbac.authorization.k8s.io/v1 kind:
Role
metadata:  namespace:
development  name: dev-
permissions rules: -
apiGroups: [""]
  resources: ["pods", "configmaps", "secrets"]
verbs: [*"] ---
apiVersion: rbac.authorization.k8s.io/v1
kind: Role metadata:  namespace:
production  name: prod-permissions
rules: - apiGroups: ["]  resources:
["pods", "configmaps"]  verbs: ["get",
"list"]
```

Service Account with Image Pull Secrets

```
apiVersion: v1 kind:
ServiceAccount
metadata:  name:
registry-sa
imagePullSecrets:
- name: private-registry-secret
--- apiVersion:
v1 kind: Pod
metadata:
  name: private-image-pod spec:
serviceAccountName: registry-sa
containers: - name: app    image: private-
registry.io/my-app:latest
```

Service Account Labels and Annotations

```
apiVersion: v1 kind:
ServiceAccount
metadata:
```

```

name: labeled-sa
namespace: production
labels:
  app: my-application
environment: production
team: backend annotations:
  description: "Service account for backend application"
owner: "backend-team@company.com" compliance.requirement:
"high-security"

```

Service Account Security Best Practices

Principle of Least Privilege

```

# ❌ BAD: Too broad permissions
apiVersion: rbac.authorization.k8s.io/v1
kind: ClusterRoleBinding metadata:
  name: bad-binding
subjects:
- kind: ServiceAccount name: app-sa
namespace: default roleRef: kind:
ClusterRole name: cluster-admin # Too
much access! apiGroup:
rbac.authorization.k8s.io

# ✅ GOOD: Specific permissions apiVersion:
rbac.authorization.k8s.io/v1 kind: Role metadata:
namespace: default name: specific-permissions
rules: - apiGroups: [""] resources: ["configmaps"]
verbs: ["get"] resourceNames: ["app-config"] #
Specific resource

```

Namespace Isolation

```

# Create dedicated namespaces kubectl
create namespace app-production kubectl
create namespace app-development

# Create service accounts per namespace
kubectl create serviceaccount prod-app-sa -n app-production kubectl
create serviceaccount dev-app-sa -n app-development

```

Disable Default Service Account

```
# Patch default service account to disable auto-mounting kubectl
patch serviceaccount default -p
'{"automountServiceAccountToken":false}'
```



```
# Or create pod without service account
apiVersion: v1 kind: Pod metadata:
  name: no-service-account-pod spec:
  automountServiceAccountToken: false
  containers:
    - name:
      app: my-
      image: my-
      app:latest
```

Token Security

```
# Use short-lived tokens
kubectl create token my-sa --duration=1h
```



```
# Avoid long-lived tokens in secrets
# Use projected service account tokens instead
```

Service Account Troubleshooting

Common Issues and Solutions

1. Permission Denied Errors

```
# Check service account permissions kubectl auth can-i get pods --
as=system:serviceaccount:default:my-sa
```

```
# Check what the service account can do kubectl auth can-i --list
--as=system:serviceaccount:default:my-sa
```

```
# Debug RBAC kubectl get rolebindings,clusterrolebindings -A -o wide
| grep my-sa
```

2. Token Not Found

```
# Check if service account exists
kubectl get serviceaccount my-sa
```

```
# Check token mounting kubectl describe pod my-pod | grep -
A5 -B5 "service-account"
```

```
# Manually create token kubectl
create token my-sa
```

3. Wrong Namespace

```
# Check service account namespace kubectl get serviceaccount my-sa -n correct-namespace  
  
# Service accounts are namespace-scoped kubectl get sa -A | grep my-sa
```

Debugging Commands

```
# View service account details  
kubectl describe serviceaccount my-sa
```

```
# Check mounted token in pod kubectl exec -it my-pod -- ls -la  
/var/run/secrets/kubernetes.io/serviceaccount/
```

```
# View token content (be careful with sensitive data) kubectl exec -it my-pod -- cat /var/run/secrets/kubernetes.io/serviceaccount/token
```

```
# Test API access from within pod kubectl exec -it my-pod -- curl -H "Authorization: Bearer $(cat /var/run/secrets/kubernetes.io/serviceaccount/token)" https://kubernetes.default.svc/api/v1/namespaces/default/pods --cacert /var/run/secrets/kubernetes.io/serviceaccount/ca.crt
```

CKA Exam Scenarios

Scenario 1: Create Service Account and Bind to Role

```
# 1. Create service account kubectl create serviceaccount app-service-account -n production  
  
# 2. Create role kubectl create role app-role --verb=get,list,create -resource=pods,configmaps -n production  
  
# 3. Create role binding kubectl create rolebinding app-binding --role=app-role -serviceaccount=production:app-service-account -n production  
  
# 4. Test permissions
```

```
kubectl auth can-i get pods --as=system:serviceaccount:production:appservice-account -n production
```

Scenario 2: Fix Pod Service Account Issue

```
# Problem: Pod cannot access API kubectl
describe pod failing-pod

# Solution: Check and fix service account
kubectl get serviceaccount kubectl create
serviceaccount correct-sa kubectl patch
deployment my-app -p
'{"spec": {"template": {"spec": {"serviceAccountName": "correct-sa"} }}}'
```

Scenario 3: Security Hardening

```
# Disable default service account auto-mount kubectl
patch serviceaccount default -p
'{"automountServiceAccountToken":false}'

# Create minimal permission service account kubectl
create serviceaccount minimal-sa
kubectl create role minimal-role --verb=get --resource=configmaps --
resourcename=app-config
kubectl create rolebinding minimal-binding --role=minimal-role -
serviceaccount=default:minimal-sa
```

Service Account Commands Summary

Basic Operations

```
# Create service account kubectl
create serviceaccount NAME kubectl
create sa NAME -n NAMESPACE
```

```
# List service accounts
kubectl get serviceaccounts
kubectl get sa kubectl get
sa -A
```

```
# Describe service account kubectl
describe serviceaccount NAME kubectl
describe sa NAME
```

```
# Delete service account kubectl  
delete serviceaccount NAME kubectl  
delete sa NAME
```

Token Operations

```
# Create token (Kubernetes 1.24+) kubectl create  
token SERVICE_ACCOUNT_NAME kubectl create token  
SA_NAME --duration=1h kubectl create token SA_NAME  
--audience=my-audience
```

```
# View mounted token in pod kubectl  
exec POD_NAME -- cat  
/var/run/secrets/kubernetes.io/serviceaccount/token
```

RBAC Integration

```
# Create role and binding for service account  
kubectl create role ROLE_NAME --verb=VERBS --resource=RESOURCES kubectl  
create rolebinding BINDING_NAME --role=ROLE_NAME -  
serviceaccount=NAMESPACE:SA_NAME
```

```
# Create cluster role and binding  
kubectl create clusterrole CLUSTER_ROLE_NAME --verb=VERBS -resource=RESOURCES  
kubectl create clusterrolebinding BINDING_NAME --  
clusterrole=CLUSTER_ROLE_NAME --serviceaccount=NAMESPACE:SA_NAME
```

```
# Test service account permissions  
kubectl auth can-i VERB RESOURCE --as=system:serviceaccount:NAMESPACE:SA_NAME  
kubectl auth can-i --list --as=system:serviceaccount:NAMESPACE:SA_NAME
```

Debugging

```
# Check service account in pod kubectl describe pod  
POD_NAME | grep -i "service account"
```

```
# View service account tokens kubectl get  
secrets | grep service-account-token
```

```
# Check RBAC bindings kubectl get  
rolebindings,clusterrolebindings -A | grep SA_NAME
```

Key Points for CKA Exam

Must Know Concepts

1. **Default Behavior:** Every pod gets the default service account if none specified
2. **Namespace Scope:** Service accounts are namespace-scoped resources
3. **RBAC Integration:** Service accounts work with RBAC for authorization
4. **Token Mounting:** Tokens automatically mounted at
`/var/run/secrets/kubernetes.io/serviceaccount/`
5. **Security:** Follow least privilege principle

Common Exam Tasks

- Create service accounts
- Bind service accounts to roles/cluster roles
- Configure pods to use specific service accounts
- Troubleshoot permission issues
- Disable automatic token mounting
- Create manual tokens

Important File Paths

- **Token:** `/var/run/secrets/kubernetes.io/serviceaccount/token`
- **CA Certificate:** `/var/run/secrets/kubernetes.io/serviceaccount/ca.crt`
- **Namespace:** `/var/run/secrets/kubernetes.io/serviceaccount/namespace`

Critical Commands

```
kubectl create serviceaccount NAME
kubectl create role NAME --verb=VERBS --resource=RESOURCES kubectl create
rolebinding NAME --role=ROLE --serviceaccount=NAMESPACE:SA kubectl auth
can-i VERB RESOURCE --as=system:serviceaccount:NAMESPACE:SA
```

Storage

Volumes

What are Volumes in Kubernetes?

Volumes provide persistent storage for pods that survives container restarts. Unlike the container's ephemeral filesystem, volumes persist data even when containers are recreated or restarted.[1]

Volume Purpose and Benefits

- **Data persistence:** Maintains data across container restarts
- **Data sharing:** Enables multiple containers in a pod to share data
- **External storage:** Connects pods to external storage systems

- **Decoupling:** Separates storage concerns from container lifecycle

Container Storage Behavior

When containers are created, Kubernetes creates a temporary filesystem that gets destroyed when the container is removed. Volumes solve this problem by providing persistent storage that exists independently of container lifecycle.[1]

Volume Types

What are Volume Types?

Volume Types determine where and how data is stored. Kubernetes supports numerous volume types to accommodate different storage requirements and infrastructure setups.[1]

Host-Based Volume Types *hostPath Volume* **hostPath** mounts a directory from the node's filesystem into the pod:

```
apiVersion: v1
kind: Pod
metadata:
  name: random-number-generator
spec:
  containers:
    - image: alpine
      name: alpine
    command: ["/bin/sh", "-c"]
      args: ["shuf -i 0-100 -n 1 >> /opt/number.out;"]
      volumeMounts:
        - mountPath: /opt
          name: data-volume
  volumes:
    - name: data-volume
  hostPath:
    path: /data
    type: Directory
```

Characteristics:

- **Node-specific:** Data tied to specific node
- **Not portable:** Pods scheduled on different nodes can't access data
- **Development use:** Suitable for single-node testing
- **Production caution:** Not recommended for multi-node clusters

Network-Based Volume Types

```
NFS (Network File System)
volumes:
  - name: nfs-volume
    nfs:
      server: nfs-server.example.com
      path: /path/to/nfs/share
```

Cloud Provider Volumes AWS

Elastic Block Store (EBS):

```
volumes:  
-   name: data-volume  
awsElasticBlockStore:  
volumeID: vol-  
1234567890abcdef0  
fsType: ext4
```

Google Cloud Persistent Disk:

```
volumes:  
-   name: data-volume  
gcePersistentDisk:  
pdName: my-disk  
fsType: ext4
```

Azure Disk:

```
volumes:  
-   name: data-volume  
azureDisk:  
    diskName: my-disk.vhd      diskURI:  
https://myaccount.blob.core.windows.net/vhds/my-disk.vhd
```

Distributed Storage Volume Types

- **Ceph**: Distributed storage system
- **GlusterFS**: Scale-out network-attached storage
- **ScaleIO**: Software-defined storage platform
- **Flocker**: Container data volume manager

Volume Mounting Process

1. **Volume definition**: Specify volume in pod spec
2. **Container mounting**: Mount volume into container filesystem
3. **Path mapping**: Map volume to specific container directory
4. **Data persistence**: Data persists beyond container lifecycle

Volume Challenges in Large Deployments

Multi-User Environment Problems

In environments with many users creating pods with different volume requirements:

- **Configuration complexity:** Each user must configure volumes manually
- **Storage management:** Admins must provision storage for every pod
- **Resource conflicts:** Multiple users might try to use same storage
- **Inconsistent setup:** Different volume configurations across teams

The Need for Abstraction

Problem: Direct volume management becomes unmanageable at scale **Solution:** Kubernetes provides **Persistent Volumes** and **Persistent Volume Claims** to abstract storage management[1]

Persistent Volumes (PVs)

What are Persistent Volumes?

Persistent Volumes (PVs) are cluster-wide storage resources provisioned by administrators. They abstract the underlying storage implementation and provide a standardized interface for storage consumption.[1]

PV Benefits

- **Centralized management:** Admins manage storage pool centrally
- **User abstraction:** Users don't need to know storage details
- **Resource pooling:** Create pool of storage resources
- **Standardization:** Consistent storage interface across cluster

Persistent Volume Definition

```
apiVersion: v1 kind:  
PersistentVolume  
metadata:  
  name: pv-vol1 spec:  
    accessModes:  
    - ReadWriteOnce  
    capacity:  
      storage: 1Gi  
    persistentVolumeReclaimPolicy: Retain  
    awsElasticBlockStore:      volumeID:  
      vol-1234567890abcdef0      fsType: ext4
```

PV Specification Components

Access Modes

Access Modes define how the volume can be mounted:

Mode	Description	Use Case
ReadWriteOnce	Single node read-write	Databases, single-instance apps (RWO)
ReadOnlyMany (ROX)	Multiple nodes read-only	Static content, shared configuration
ReadWriteMany (RWX)	Multiple nodes read-write	Shared filesystems, collaborative apps

```
Capacity capacity: storage: 1Gi #
Available storage space
```

Reclaim Policy

Reclaim Policy determines what happens when PV is released:

Policy	Behavior	Description
Retain	Manual cleanup	Admin must manually reclaim
Delete	Automatic deletion	Storage resource deleted automatically
Recycle	Data scrubbing	Volume scrubbed and made available (deprecated)

Creating and Viewing PVs

```
# Create PV kubectl create -f pv-
definition.yaml

# List PVs
kubectl get persistentvolume kubectl
get pv

# Detailed PV information kubectl
describe pv pv-vol1
```

PV Status States

Status	Description
Available	Ready for use, not yet claimed
Status	Description

Bound Bound to a PVC

Released PVC deleted, but not yet

reclaimed **Failed** Reclamation failed

Persistent Volume Claims (PVCs)

What are Persistent Volume Claims?

Persistent Volume Claims (PVCs) are requests for storage by users. They specify storage requirements and are matched with available Persistent Volumes.[1]

PVC Purpose

- **User interface:** Simple way for users to request storage
- **Resource claiming:** Claims available PVs based on requirements
- **Abstraction:** Users don't need to know underlying storage details
- **Automatic binding:** Kubernetes handles PV-PVC binding

```
PVC Definition apiVersion: v1
kind: PersistentVolumeClaim
metadata:
  name: myclaim
spec: accessModes:
- ReadWriteOnce
resources:
requests:
storage: 500Mi
```

Creating and Managing PVCs

```
# Create PVC kubectl create -f pvc-
definition.yaml
```

```
# List PVCs
kubectl get persistentvolumeclaim kubectl
get pvc
```

```
# Describe PVC kubectl
describe pvc myclaim
```

```
# Delete PVC
kubectl delete persistentvolumeclaim myclaim
```

PV and PVC Binding Process

What is PV-PVC Binding?

Binding is the process where Kubernetes automatically matches PVCs with suitable PVs based on specified criteria.[1]

Binding Criteria

Kubernetes considers multiple factors when binding PVCs to PVs:

1. *Sufficient Capacity* # PVC requests 500Mi resources: requests:
storage: 500Mi

PV provides 1Gi (sufficient)
capacity: storage: 1Gi
2. *Access Modes Compatibility*
Both must have compatible access modes
PVC
accessModes: -
ReadWriteOnce

PV
accessModes: -
ReadWriteOnce
3. *Volume Modes*
Block or Filesystem mode compatibility volumeMode:
Filesystem
4. *Storage Class*
Must match storage class (if specified) storageClassName:
fast-ssd
5. *Selector Labels* # PV with Labels metadata: labels:
name: my-pv
environment: production

PVC with selector spec:
selector:
matchLabels:

```
name: my-pv
environment: production
```

Binding Examples

Successful Binding #

Before binding

```
kubectl get pv,pvc
NAME                                     CAPACITY   ACCESS MODES   RECLAIM POLICY   STATUS
persistentvolume/pv-vol1    1Gi        RWO            Retain          Available
```

```
NAME                                     STATUS      VOLUME   CAPACITY   ACCESS MODES
persistentvolumeclaim/myclaim  Pending
```

After binding

```
NAME                                     CAPACITY   ACCESS MODES   RECLAIM POLICY   STATUS
CLAIM
persistentvolume/pv-vol1    1Gi        RWO            Retain          Bound
default/myclaim
```

```
NAME                                     STATUS      VOLUME   CAPACITY   ACCESS MODES
persistentvolumeclaim/myclaim  Bound       pv-vol1  1Gi        RWO
```

Failed Binding (Pending State)

If no suitable PV is available, PVC remains in **Pending** status:

```
NAME                                     STATUS      VOLUME   CAPACITY   ACCESS MODES
persistentvolumeclaim/myclaim  Pending
```

One-to-One Relationship

- **Exclusive binding:** Each PV can bind to only one PVC
- **Capacity allocation:** Entire PV capacity allocated to PVC (even if PVC requests less)
- **No sharing:** PVs cannot be shared between multiple PVCs

Using PVCs in Pods

Mounting PVCs in Pods

Once a PVC is bound to a PV, pods can use the PVC to access persistent storage:

```
apiVersion: v1
kind: Pod
metadata:
```

```
name: mypod
spec:
containers: -
  name: frontend
  image: nginx
  volumeMounts:
  - mountPath: "/var/www/html"
    name: mypd
volumes: - name:
  mypd
  persistentVolumeCl
aim:
  claimName: myclaim
```

PVC Lifecycle in Pods

1. **PVC creation:** User creates PVC
2. **Binding:** Kubernetes binds PVC to suitable PV
3. **Pod usage:** Pod references PVC in volume definition
4. **Mount:** Container mounts PVC at specified path
5. **Data persistence:** Data persists across pod restarts

Imperative Commands for Storage

PV Management Commands

```
# Create PV from YAML
kubectl create -f pv-definition.yaml
```

```
# List all PVs kubectl
get pv
kubectl get persistentvolumes
```

```
# Describe specific PV kubectl
describe pv pv-vol1
```

```
# Delete PV
kubectl delete pv pv-vol1
```

```
# Edit PV
kubectl edit pv pv-vol1
```

```
# View PV in different output formats
kubectl get pv -o wide kubectl get pv
-o yaml kubectl get pv -o json

PVC Management Commands # Create PVC imperatively kubectl create
pvc my-pvc --size=1Gi --access-modes=ReadWriteOnce

# Create from YAML kubectl create -f
pvc-definition.yaml

# List PVCs kubectl
get pvc
kubectl get persistentvolumeclaims

# Describe PVC kubectl
describe pvc myclaim

# Delete PVC kubectl
delete pvc myclaim

# View PVC details kubectl get
pvc -o wide kubectl get pvc
myclaim -o yaml

Combined PV/PVC Operations
# View both PVs and PVCs
kubectl get pv,pvc

# Monitor binding status kubectl
get pv,pvc -w

# Check events related to storage
kubectl get events --field-selector involvedObject.kind=PersistentVolume
kubectl get events --field-selector involvedObject.kind=PersistentVolumeClaim

Declarative Configurations

Complete PV-PVC-Pod Example
# pv-definition.yaml
apiVersion: v1 kind:
```

```
PersistentVolume
metadata:
  name: task-pv-volume
labels:
  type: local spec:
  storageClassName: manual
capacity:
  storage: 10Gi
accessModes: -
ReadWriteOnce
hostPath:
  path: "/tmp/data"

---
# pvc-definition.yaml
apiVersion: v1 kind:
PersistentVolumeClaim
metadata:  name: task-pv-
claim spec:
storageClassName: manual
accessModes: -
ReadWriteOnce  resources:
requests:      storage:
3Gi

---
# pod-definition.yaml
apiVersion: v1 kind:
Pod metadata:
  name: task-pv-pod spec:
volumes: - name: task-pv-
storage
persistentVolumeClaim:
  claimName: task-pv-claim
containers: - name: task-pv-
container  image: nginx
ports: - containerPort: 80
name: "http-server"
volumeMounts:
  - mountPath: "/usr/share/nginx/html"
name: task-pv-storage
```

Deployment with PVC

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: webapp-deployment spec:
  replicas: 3 selector:
matchLabels: app: webapp
template: metadata:
  labels: app: webapp
spec: containers:
  - name: webapp image: nginx
  volumeMounts:
    - name: webapp-storage
      mountPath: /var/www/html
volumes:
  - name: webapp-storage
    persistentVolumeClaim:
      claimName: webapp-pvc Storage
```

Classes (Dynamic Provisioning)

What are Storage Classes?

Storage Classes enable dynamic provisioning of storage resources, automatically creating PVs when PVCs are created.[1]

Storage Class Benefits

- **Automatic provisioning:** No need to pre-create PVs
- **On-demand storage:** Storage created when needed
- **Different tiers:** Support multiple storage performance tiers
- **Cloud integration:** Seamless integration with cloud storage services

Storage Class Definition

```
apiVersion: storage.k8s.io/v1
kind: StorageClass metadata:
  name: fast-ssd provisioner:
    kubernetes.io/aws-ebs parameters:
      type: gp2
      zones: us-west-2a, us-west-2b
      encrypted: "true" reclaimPolicy: Delete
      allowVolumeExpansion: true
      volumeBindingMode: WaitForFirstConsumer
```

Using Storage Classes with PVCs

```
apiVersion: v1 kind:  
PersistentVolumeClaim  
metadata:  
  name: dynamic-pvc spec:  
    storageClassName: fast-ssd  
accessModes: -  
  ReadWriteOnce resources:  
  requests: storage: 100Gi
```

Reclaim Policies in Detail

Understanding Reclaim Policies

Reclaim Policies determine what happens to a PV after its associated PVC is deleted.[1]

Retain Policy persistentVolumeReclaimPolicy:

Retain

- **Manual intervention:** Admin must manually handle PV
- **Data preservation:** Data remains intact
- **Status change:** PV status changes to “Released”
- **Reuse restriction:** Cannot be claimed by new PVC until manually cleaned

Delete Policy persistentVolumeReclaimPolicy:

Delete

- **Automatic cleanup:** PV and underlying storage deleted automatically
- **Data loss:** All data permanently lost
- **Cloud storage:** Actual cloud storage resources deleted
- **Default:** Often default for dynamically provisioned volumes

Recycle Policy (Deprecated) persistentVolumeReclaimPolicy:

Recycle

- **Data scrubbing:** Basic scrub (rm -rf /thevolume/*)
- **Availability:** PV becomes available for new claims
- **Deprecation:** Replaced by dynamic provisioning

Storage Best Practices

PV/PVC Design Principles

- **Right-size storage:** Request appropriate storage amounts

- **Access mode selection:** Choose correct access modes for use case
- **Storage class strategy:** Use storage classes for different performance tiers
- **Backup planning:** Implement backup strategies for critical data

Security Considerations

- **Access controls:** Use RBAC to control PVC creation
- **Namespace isolation:** Keep storage resources in appropriate namespaces
- **Data encryption:** Enable encryption for sensitive data
- **Network policies:** Secure storage network communications

Performance Optimization

- **Storage location:** Consider data locality for performance
- **IOPS requirements:** Match storage type to performance needs
- **Caching strategies:** Implement appropriate caching layers
- **Monitoring:** Monitor storage performance and usage

Troubleshooting Storage Issues

Common PVC Problems # PVC stuck in Pending
kubectl describe pvc problematic-pvc # Look for events and error messages

Check available PVs kubectl get pv
Verify capacity, access modes, storage class

Check storage class kubectl get storageclass kubectl describe storageclass fast-ssd

PV Binding Issues

Check PV Labels and selectors
kubectl describe pv my-pv kubectl describe pvc my-pvc

Verify access modes compatibility kubectl get pv my-pv -o yaml | grep accessModes
kubectl get pvc my-pvc -o yaml | grep accessModes

```
# Check storage capacity kubectl describe
pv my-pv | grep Capacity kubectl describe
pvc my-pvc | grep storage

Pod Mount Failures # Check pod
events kubectl describe pod
my-pod
kubectl get events --field-selector involvedObject.name=my-pod

# Verify PVC status kubectl
get pvc
kubectl describe pvc my-claim

# Check volume mounts kubectl describe pod my-pod
| grep -A 5 "Mounts"
```

Key Storage Commands

Summary

Essential PV Commands

```
# PV Lifecycle
kubectl create -f pv-definition.yaml
kubectl get pv kubectl describe pv
kubectl delete pv
```

Essential PVC Commands

```
# PVC Lifecycle
kubectl create -f pvc-definition.yaml
kubectl get pvc kubectl describe pvc
kubectl delete pvc
```

Storage Monitoring Commands

```
# Combined view kubectl
get pv,pvc
kubectl get pv,pvc -o wide
```

```
# Storage events
kubectl get events | grep -i volume kubectl
get events | grep -i persistent
```

```
# Storage class information
kubectl get storageclass kubectl
describe storageclass
```

Networking

Networking Fundamentals

What is Kubernetes Networking?

Kubernetes Networking is the mechanism by which different resources within and outside your cluster are able to communicate with each other. It provides the foundation for understanding how containers, pods, and services within Kubernetes communicate seamlessly across the cluster.[1][2]

Core Networking Principles

Kubernetes networking is built on several fundamental principles that ensure consistent and reliable communication:[3][1]

- **Every pod gets its own IP address:** Each pod receives a unique cluster-wide IP address
- **Containers within a pod share the pod IP address:** All containers in a pod communicate via localhost
- **Pods can communicate with all other pods:** Direct communication without NAT across the entire cluster
- **Network isolation:** Defined using network policies rather than network structure

Why Kubernetes Networking Matters

Understanding Kubernetes networking is crucial for:[2]

- **Proper environment configuration:** Setting up production-ready clusters
- **Complex networking scenarios:** Implementing advanced traffic management
- **Security implementation:** Controlling communication flows
- **Troubleshooting:** Diagnosing connectivity issues

The Kubernetes Network Model

What is the Kubernetes Network Model?

The **Kubernetes Network Model** specifies how different components communicate within a cluster. It provides a flat network structure that makes pod-to-pod communication simple and predictable.[1]

Network Model Components

Pod Network (Cluster Network)

The **Pod Network** handles communication between pods across the cluster:[3]

```
# Every pod gets a unique IP from the cluster CIDR
apiVersion: v1 kind: Pod metadata:
  name: web-pod
spec:
  containers: -
    name: nginx
    image: nginx
  ports:
    - containerPort: 80
```

characteristics:

- **Cluster-wide IP addressing:** All pods can reach each other using IP addresses
- **No NAT required:** Direct pod-to-pod communication
- **Cross-node communication:** Pods on different nodes communicate seamlessly
- **Network namespace sharing:** Containers in same pod share network namespace

Service Network

Services provide stable network identities for accessing groups of pods:[3]

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

Network Model Implementation

The Kubernetes network model is implemented using:[4]

- **Container Runtime:** Sets up network namespaces for containers
- **CNI Plugin:** Configures networking rules and policies
- **Kube-proxy:** Handles service networking and load balancing

Container Network Interface (CNI)

What is CNI?

Container Network Interface (CNI) is a standard interface specification for network plugins in container orchestration systems. It defines how container runtimes collaborate with networking plugins to configure networking for containers and pods.[4]

CNI Responsibilities

CNI plugins handle several critical networking tasks:[4]

- **IP address assignment:** Allocating unique IPs to containers
- **Network interface setup:** Configuring network interfaces
- **Route definition:** Setting up routing tables
- **Network policy enforcement:** Implementing security rules

Popular CNI Plugins

Weave Net

Weave Net creates a virtual network that connects containers across multiple hosts:

```
# Deploy Weave Net
kubectl apply -f "https://cloud.weave.works/k8s/net?k8s-version=$(kubectl
version | base64 | tr -d '\n')"
```

Features:

- **Automatic IP allocation:** Dynamic IP address management
- **Encryption:** Secure pod-to-pod communication
- **Network policies:** Traffic filtering capabilities
- **Multi-cloud support:** Works across different cloud providers *Calico*

Calico provides networking and network security for containers:

```
# Deploy Calico
kubectl apply -f
https://docs.projectcalico.org/manifests/calico.yaml
```

Features:

- **Network policies:** Advanced traffic filtering
- **BGP routing:** Efficient inter-node communication
- **Network security:** Built-in firewall capabilities
- **Performance:** High-performance networking

Flannel

Flannel is a simple overlay network for Kubernetes:

```
# Deploy Flannel
kubectl apply -f
https://raw.githubusercontent.com/flannelio/flannel/master/
Documentation/kube-flannel.yml
```

Features:

- **Simplicity:** Easy to deploy and manage
- **Overlay networking:** VXLAN-based networking
- **Cross-platform:** Works on various operating systems

CNI Configuration

CNI is configured through the kubelet:[5]

```
# Kubelet CNI configuration
ExecStart=/usr/local/bin/kubelet \
--network-plugin=cni \
--cni-bin-dir=/opt/cni/bin \
--cni-conf-dir=/etc/cni/net.d
```

Configuration files:

```
{
  "cniVersion": "0.2.0",
  "name": "mynet",
  "type": "bridge",
  "bridge": "cni0",
  "isGateway": true,
  "ipMasq": true,
  "ipam": {
    "type": "host-local",
    "subnet": "10.244.0.0/16",
    "routes": [
      { "dst": "0.0.0.0/0" }
    ]
  }
}
```

IP Address Management (IPAM)

What is IPAM?

IP Address Management (IPAM) ensures that each pod receives a unique IP address within the cluster. It prevents IP conflicts and manages IP allocation across nodes.[5]

IPAM Plugins

Host-Local IPAM

Host-local maintains IP allocation on each node locally:

```
{
  "ipam": {
    "type": "host-local",
    "subnet": "10.244.0.0/16",
    "routes": [
      { "dst": "0.0.0.0/0" }
    ]
  }
}
```

Characteristics:

- **Node-specific allocation:** Each node manages its own IP range
- **Persistent storage:** IP allocations stored locally
- **Simple configuration:** Easy to set up and manage

DHCP IPAM

DHCP uses external DHCP servers for IP allocation:

```
{
  "ipam": {
    "type": "dhcp"
  }
}
```

IP Range Planning

Proper IP range planning is crucial for large clusters:[5]

```
# Cluster CIDR configuration kube-apiserver --service-
cluster-ip-range=10.96.0.0/12
```

Common IP ranges:

- **Pod Network:** 10.244.0.0/16 (65,534 IPs)
- **Service Network:** 10.96.0.0/12 (1,048,574 IPs)
- **Node Network:** 192.168.1.0/24 (254 IPs)

Services and Service Discovery

What are Services?

Services provide stable network endpoints for accessing groups of pods. They abstract away the dynamic nature of pod IP addresses and provide load balancing.[6]

Service Types

ClusterIP Service

ClusterIP exposes the service internally within the cluster:

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

- **Internal communication:** Service-to-service communication
- **Database access:** Backend services accessing databases
- **Microservices:** Communication between microservices

NodePort Service

NodePort exposes the service on each node's IP at a static port:

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

Characteristics:

- **External access:** Accessible from outside the cluster
- **Port range:** 30000-32767 by default

- **All nodes:** Service accessible on all nodes

LoadBalancer Service

LoadBalancer provisions an external load balancer:

```
apiVersion: v1 kind:  
Service metadata:  
  name: loadbalancer-service  
spec: type: LoadBalancer  
selector: app: web  
ports: - port: 80  
targetPort: 80 Features:
```

- **Cloud integration:** Works with cloud provider load balancers
 - **Automatic provisioning:** Load balancer created automatically
- **High availability:** Built-in load balancing and failover **Service**

Discovery

Kubernetes provides automatic service discovery through:[5]

DNS-Based Discovery

Every service gets a DNS name:

```
# Service DNS format  
.svc.cluster.local  
  
# Examples  
web-service.default.svc.cluster.local database.production.svc.cluster.local
```

Environment Variables

Services are exposed as environment variables:

```
# Service environment variables  
WEB_SERVICE_HOST=10.96.1.100  
WEB_SERVICE_PORT=80  
WEB_SERVICE_PORT_80_TCP=tcp://10.96.1.100:80
```

Ingress and Ingress Controllers

What is Ingress?

Ingress manages external access to services in a cluster, typically HTTP and HTTPS. It provides URL-based routing, SSL termination, and load balancing.[5]

Ingress Components

Ingress Resource

Ingress Resource defines the routing rules:

```
apiVersion: networking.k8s.io/v1 kind: Ingress
metadata: name: web-ingress annotations:
nginx.ingress.kubernetes.io/rewrite-target: /
spec: rules:
- host: example.com
http: paths:
- path: /app
pathType: Prefix
backend:
service:
  name: app-service
port: number:
80 - path: /api
pathType: Prefix
backend: service:
  name: api-service
port: number:
8080
```

Ingress Controller

Ingress Controller implements the ingress rules:

```
# Deploy NGINX Ingress Controller
kubectl apply -f
https://raw.githubusercontent.com/kubernetes/ingressnginx/controller-
v1.8.1/deploy/static/provider/cloud/deploy.yaml
```

Advanced Ingress Features

```
SSL/TLS Termination apiVersion:
networking.k8s.io/v1 kind:
Ingress metadata:
  name: tls-ingress spec:
    tls: - hosts: -
      example.com
```

```
secretName: tls-secret
rules: - host:
example.com     http:
paths:   - path: /
pathType: Prefix
backend:
service:
    name: web-service
port:           number:
80

Path-Based Routing spec:
rules: - host:
api.example.com     http:
paths:   - path: /v1
pathType: Prefix
backend:
service:
    name: api-v1-service
port:           number: 80
- path: /v2           pathType:
Prefix           backend:
service:
    name: api-v2-service
port:           number: 80
```

Kubernetes Gateway API

What is the Gateway API?

The **Kubernetes Gateway API** is a modern, extensible approach to managing ingress and traffic routing in Kubernetes. It provides more expressive, structured routing capabilities compared to traditional Ingress resources.[7]

Gateway API Benefits

The Gateway API offers several advantages over Ingress:[8]

- **More expressive:** Rich traffic management capabilities
- **Role separation:** Clear separation between infrastructure and application concerns
- **Extensibility:** Built-in extension points for custom functionality
- **Protocol support:** Native support for HTTP, HTTPS, TCP, UDP, and gRPC
- **Implementation agnostic:** Works with different gateway controllers

Gateway API Architecture

GatewayClass

GatewayClass defines a set of gateways implemented by a specific controller:[7]

```
apiVersion: gateway.networking.k8s.io/v1
kind: GatewayClass metadata:
  name: nginx spec: controllerName:
    nginx.org/gateway-controller description:
      "NGINX Gateway Controller" Purpose:
```

- **Decouples configuration from implementation:** Allows switching between different gateway implementations
- **Multi-controller support:** Multiple gateway implementations can coexist
- **Administrative control:** Platform administrators control available implementations

Gateway

Gateway defines how traffic enters the cluster:[7]

```
apiVersion: gateway.networking.k8s.io/v1
kind: Gateway metadata:
  name: nginx-gateway
  namespace: default
spec:
  gatewayClassName: nginx
  listeners: - name:
    http protocol: HTTP
    port: 80
    allowedRoutes:
    namespaces:
      from: All - name:
        https protocol:
        HTTPS port: 443
        tls: mode:
        Terminate
        certificateRefs:
        - kind: Secret
        name: tls-secret
    allowedRoutes:
    namespaces:
      from: All Key features:
```

- **Multiple protocols:** HTTP, HTTPS, TCP, UDP support
 - **TLS termination:** Built-in SSL/TLS handling
- **Namespace control:** Fine-grained route permissions

HTTP Routing with Gateway API

Basic HTTP Routing

HTTPRoute defines how HTTP traffic is routed to services:

```
apiVersion: gateway.networking.k8s.io/v1
kind: HTTPRoute metadata:
  name: basic-route
  namespace: default spec:
    parentRefs: - name:
      nginx-gateway
    hostnames: -
      "api.example.com"
    rules:
      - matches:
        - path:
          type: PathPrefix
          value: /api/v1
        backendRefs: - name:
          api-v1-service
          port: 80      weight:
100 Advanced HTTP Routing
```

Features

Header-Based Routing:

```
rules:
  - matches:
    - headers: - name:
      "version"
      value: "v2"      path:
        type: PathPrefix
      value: /api
    backendRefs: - name:
      api-v2-service      port:
80 Method-Based Routing:
```

```
rules: -  
  matches: -  
    method: GET  
    path:  
      type: PathPrefix  
    value: /api/users  
  backendRefs: - name:  
    user-read-service port:  
    80 - matches: - method:  
    POST path:  
      type: PathPrefix  
    value: /api/users  
  backendRefs: - name: user-  
    write-service port: 80
```

Traffic Management with Gateway API

HTTP Redirects HTTP to

HTTPS Redirect:

```
apiVersion: gateway.networking.k8s.io/v1  
kind: HTTPRoute metadata:  
  name: https-redirect  
spec:  
  parentRefs:  
  - name: nginx-gateway  
  hostnames:  
  - "example.com"  
  rules:  
  - filters:  
    - type:  
      RequestRedirect  
    requestRedirect:  
      scheme: https  
      statusCode: 301  
      Path
```

Redirects:

```
rules:  
- matches:  
- path: type:  
  PathPrefix value:  
  /old-api filters:  
  - type: RequestRedirect  
  requestRedirect:
```

```
path:           type:  
ReplacePrefixMatch  
replacePrefixMatch: /new-  
api      statusCode: 302
```

URL Rewriting Path

Rewriting:

```
apiVersion: gateway.networking.k8s.io/v1  
kind: HTTPRoute metadata:  
  name: rewrite-path  
namespace: default spec:  
  parentRefs: - name:  
nginx-gateway  rules:  
- matches:  
- path:           type:  
  PathPrefix       value:  
  /external-api   filters:  
  - type: URLRewrite  
    urlRewrite:     path:  
      type: ReplacePrefixMatch  
replacePrefixMatch: /internal-api  
backendRefs: - name: internal-service  
port: 80
```

Header Modification Request

Header Modification:

```
rules: -  
filters:  
- type:  
  RequestHeaderModifier  
  requestHeaderModifier:  
    add:  
- name: "X-Environment"  
  value: "production" -  
  name: "X-Request-ID"  
  value: "generated-uuid"  
  set:  
- name: "Authorization"  
  value: "Bearer token"  
  remove: - "X-Debug"
```

```
backendRefs: - name:  
  backend-service port:
```

80 Response Header

Modification:

```
filters:  
- type:  
  ResponseHeaderModifier  
  responseHeaderModifier:  
    add:  
    - name: "X-Cache-Status"  
      value: "MISS" set:  
    - name: "Cache-Control"  
      value: "max-age=3600"  
    remove: - "Server"
```

Traffic Splitting

Canary Deployments:

```
apiVersion: gateway.networking.k8s.io/v1  
kind: HTTPRoute metadata:  
  name: canary-split  
  namespace: default  
spec: parentRefs: -  
name: nginx-gateway  
rules: - backendRefs:  
- name: stable-service  
  port: 80 weight:  
90 - name: canary-  
service port: 80  
weight: 10 A/B Testing:
```

```
rules:  
- matches:  
- headers: - name: "X-  
  User-Group" value:  
  "beta" backendRefs: -  
  name: beta-service  
  port: 80 - backendRefs:  
  - name: stable-service  
  port: 80
```

Request Mirroring

Traffic Mirroring for Testing:

```
apiVersion: gateway.networking.k8s.io/v1
kind: HTTPRoute
metadata:
  name: mirror-traffic
  namespace: default
spec:
  parentRefs:
    - name: nginx-gateway
  rules:
    - filters: []
      type: RequestMirror
      requestMirror:
        backendRef:
          name: test-service
        port: 80
        backendRefs:
          - name: production-service
            port: 80
```

Advanced Gateway API Features

TCP and UDP Routing

TCP Route:

```
apiVersion: gateway.networking.k8s.io/v1alpha2
kind: TCPRoute
metadata:
  name: database-route
  namespace: default
spec:
  parentRefs:
    - name: tcp-gateway
  rules:
    - backendRefs:
      - name: postgres-service
        port: 5432
```

5432 UDP Route:

```
apiVersion: gateway.networking.k8s.io/v1alpha2
kind: UDPRoute
metadata:
  name: dns-route
  namespace: default
spec:
  parentRefs: []
  name: udp-gateway
  rules:
    - backendRefs:
      - name: dns-service
        port: 53
```

TLS Configuration TLS

Termination:

```
apiVersion: gateway.networking.k8s.io/v1
kind: Gateway metadata:
  name: tls-gateway
spec:
  gatewayClassName: nginx
  listeners:
    - name:
      https: protocol: HTTPS
      port: 443   tls:
        mode: Terminate
      certificateRefs:
        - kind: Secret
          name: example-com-tls
          namespace:
        default: allowedRoutes:
        namespaces: from: All
TLS Passthrough:
  listeners:
    - name: tls-passthrough
      protocol: TLS  port:
      443   tls: mode:
      Passthrough
      allowedRoutes:
      namespaces: from:
      All
```

Installing NGINX Gateway Fabric

Prerequisites

Before installing NGINX Gateway Fabric:[9]

```
# Install Gateway API CRDs
kubectl kustomize "https://github.com/nginx/nginx-gateway-fabric/config/crd/gateway-api/standard?ref=v1.6.2" | kubectl apply -f -
kubectl kustomize "https://github.com/nginx/nginx-gatewayfabric/config/crd/gateway-api/experimental?ref=v1.6.2" | kubectl apply -f -
```

Installation via Helm

Install NGINX Gateway Fabric:[9]

```
# Add NGINX Helm repository
```

```
helm repo add nginx-stable https://helm.nginx.com/stable helm
repo update
```

```
# Install NGINX Gateway Fabric helm install ngf
oci://ghcr.io/nginx/charts/nginx-gateway-fabric \
--create-namespace \
-n nginx-gateway \
--wait
```

Verify Installation

```
# Check Gateway Fabric pods kubectl
get pods -n nginx-gateway
```

```
# Check GatewayClass kubectl
get gatewayclass
```

```
# Check Gateway API resources
kubectl get gateways kubectl
get httproutes
```

Practical Gateway API Examples

Multi-Service Application

Complete application setup:

```
# GatewayClass
apiVersion: gateway.networking.k8s.io/v1 kind:
GatewayClass metadata: name: nginx spec:
controllerName: nginx.org/gateway-controller

--- #
Gateway
apiVersion: gateway.networking.k8s.io/v1
kind: Gateway metadata:
  name: app-gateway
  namespace: default spec:
  gatewayClassName: nginx
  listeners: - name: http
    protocol: HTTP      port:
    80      - name: https
    protocol: HTTPS     port:
    443     tls:         mode:
```

```
Terminate
certificateRefs:      -
kind: Secret
name: app-tls

--- # Frontend
Route
apiVersion: gateway.networking.k8s.io/v1
kind: HTTPRoute metadata:
  name: frontend-route
namespace: default spec:
  parentRefs:  -
  name: app-gateway
  hostnames:  -
  "app.example.com"
  rules:
  - matches:
  - path:        type:
    PathPrefix
    value: /
    backendRefs:
  - name: frontend-
    service     port: 80

---
# API Route with versioning apiVersion:
gateway.networking.k8s.io/v1 kind:
HTTPRoute metadata:  name: api-route
namespace: default spec:  parentRefs:
- name: app-gateway  hostnames:  -
  "api.example.com"  rules:
- matches:
- path:        type:
  PathPrefix
  value: /v1
  backendRefs:
- name: api-v1-service
  port: 80  - matches:
- path:        type:
  PathPrefix
  value: /v2
  backendRefs:
```

```
- name: api-v2-service
  port: 80
```

Network Policies

What are Network Policies?

Network Policies are Kubernetes resources that control traffic flow between pods, providing network-level security and segmentation. They act as a firewall for your pods, defining which traffic is allowed or denied.[6]

Network Policy Components

Pod Selector

Pod Selector determines which pods the policy applies to:

```
apiVersion: networking.k8s.io/v1
kind: NetworkPolicy metadata:
  name: web-policy
  namespace: production
  spec: podSelector:
    matchLabels: app:
      web tier:
      frontend policyTypes:
        - Ingress - Egress
```

Ingress Rules

Ingress Rules control incoming traffic to selected pods:

```
spec:
  ingress:
    - from:
        # Allow traffic from pods with specific labels
      - podSelector: matchLabels: app:
        api
          # Allow traffic from specific namespaces
      - namespaceSelector: matchLabels:
        environment: production # Allow traffic
        from specific IP ranges - ipBlock:
          cidr: 192.168.1.0/24
    except:
      - 192.168.1.100/32 ports:
```

```
- protocol: TCP      port:
80     - protocol: TCP
port: 443
```

Egress Rules

Egress Rules control outgoing traffic from selected pods:

```
spec:
egress:
  - to:
    # Allow traffic to database pods
  - podSelector:           matchLabels:
app: database
  ports:   -
protocol: TCP
port: 5432  - to:
  # Allow traffic to external services
- ipBlock:
  cidr:  0.0.0.0/0
ports:   - protocol: TCP
port: 443  # HTTPS  -
protocol: TCP      port:
53  # DNS  - protocol:
UDP      port: 53  # DNS
```

Network Policy Examples

Default Deny All Policy

Deny all ingress and egress traffic:

```
apiVersion: networking.k8s.io/v1 kind: NetworkPolicy
metadata: name: default-deny-all  namespace:
production spec: podSelector: {}  # Applies to all
pods in namespace  policyTypes:  - Ingress  -
Egress
```

Allow Specific Traffic

Three-tier application policy:

```
# Frontend policy - allows ingress from anywhere, egress to API
apiVersion: networking.k8s.io/v1 kind: NetworkPolicy metadata:
name: frontend-policy  namespace: production spec:
```

```
podSelector:
matchLabels:
tier: frontend
policyTypes: - 
Ingress - Egress
ingress:
- {} # Allow all
  ingress
  egress:
- to: - podSelector:
  matchLabels:
    tier: api    ports:
    - protocol: TCP
    port: 8080

---
# API policy - allows ingress from frontend, egress to database
apiVersion: networking.k8s.io/v1 kind: NetworkPolicy metadata:
name: api-policy namespace: production spec: podSelector:
matchLabels: tier: api policyTypes: - Ingress - 
Egress ingress:
- from: -
  podSelector:
  matchLabels:
    tier: frontend
  ports: - protocol:
    TCP      port: 8080
  egress:
- to: - podSelector:
  matchLabels:
    tier: database
  ports: - protocol:
    TCP      port: 5432

---
# Database policy - allows ingress from API only
apiVersion: networking.k8s.io/v1 kind:
NetworkPolicy metadata:
  name: database-policy
namespace: production
spec: podSelector:
matchLabels:
```

```
tier: database
policyTypes: -
Ingress ingress:
- from: -
podSelector:
matchLabels:
tier: api ports:
- protocol: TCP
port: 5432
```

Network Policy Best Practices

Namespace Isolation

Isolate different environments:

```
# Production namespace policy
apiVersion: networking.k8s.io/v1
kind: NetworkPolicy metadata:
name: production-isolation
namespace: production spec:
podSelector: {} policyTypes:
- Ingress ingress:
- from:
- namespaceSelector:
  matchLabels:
    environment: production
namespaceSelector: matchLabels: name:
ingress-nginx # Allow ingress controller
```

DNS Access Policy Allow

DNS resolution:

```
apiVersion: networking.k8s.io/v1
kind: NetworkPolicy metadata:
  name: allow-dns
namespace: production spec:
  podSelector: {}
  policyTypes: -
  Egress egress:
  - to:
  - namespaceSelector:
    matchLabels:
```

```
        name: kube-system
- podSelector:
matchLabels:
    k8s-app: kube-dns
ports:   - protocol: TCP
port: 53   - protocol:
          UDP port: 53
```

DNS in Kubernetes

What is Kubernetes DNS?

Kubernetes DNS provides service discovery and name resolution within the cluster. It automatically creates DNS records for services and pods, enabling communication using human-readable names instead of IP addresses.[5]

DNS Architecture

CoreDNS

CoreDNS is the default DNS server in Kubernetes:[5]

```
apiVersion: v1 kind:
ConfigMap metadata:
    name: coredns
namespace: kube-system
data: Corefile: |
.:53 {           errors
health {
lameduck 5s
}
ready
    kubernetes cluster.local in-addr.arpa ip6.arpa {
pods insecure
        fallthrough in-addr.arpa ip6.arpa
ttl 30
    }
    prometheus :9153
forward . /etc/resolv.conf
cache 30
loop
reload
    loadbalance
}
```

DNS Records

Service DNS Records

Services get DNS records in the following format:[5]

```
# Service DNS format
..svc.cluster.local
```

Examples

```
web-service.default.svc.cluster.local
database.production.svc.cluster.local
api-gateway.kube-system.svc.cluster.local
```

```
# ClusterIP service
# Database service
# System service
```

Service record types:

- **A Record:** Points to service ClusterIP
- **SRV Record:** Includes port information
- **CNAME Record:** For external services

Pod DNS Records

Pods get DNS records based on their IP addresses:[5]

```
# Pod DNS format
..pod.cluster.local
```

Examples

```
10-244-1-5.default.pod.cluster.local      # Pod with IP 10.244.1.5
172-17-0-3.kube-system.pod.cluster.local # System pod
```

Headless Service Records

Headless Services create DNS records for individual pods:

```
apiVersion: v1
kind: Service
metadata:
  name: headless-service
spec:
  clusterIP: None    # Makes service headless
  selector:
    app: database
  ports:
    - port: 5432
# Headless service DNS records
```

```
database-0.headless-service.default.svc.cluster.local database-1.headless-
service.default.svc.cluster.local database-2.headless-
service.default.svc.cluster.local
```

DNS Configuration

Pod DNS Configuration

Pods automatically get DNS configuration:[5]

```
# /etc/resolv.conf in pod nameserver
10.96.0.10
search default.svc.cluster.local svc.cluster.local cluster.local options
ndots:5
```

Custom DNS Configuration Customize

DNS for specific pods:

```
apiVersion: v1
kind: Pod metadata:
  name: custom-dns-pod
spec: dnsPolicy:
  "None" dnsConfig:
  nameservers: -
    8.8.8.8 - 8.8.4.4
  searches: -
    custom.local
  options: - name:
    ndots value: "2"
  containers: - name:
    app image: nginx
```

Load Balancing and Service Mesh

Service Load Balancing

Kubernetes provides built-in load balancing through Services:[5]

Service Load Balancing Algorithms

Round Robin (default):

```
apiVersion: v1 kind:
Service metadata:
```

```

name: web-service spec:
  selector: app: web ports: -
  port: 80 targetPort: 8080
  sessionAffinity: None # Round robin
Session Affinity:
  spec: sessionAffinity: ClientIP
  sessionAffinityConfig: clientIP:
  timeoutSeconds: 10800 # 3 hours

```

External Load Balancers

Cloud Provider Load Balancers AWS

Application Load Balancer:

```

apiVersion: v1 kind: Service metadata: name: aws-lb-service annotations:
  service.beta.kubernetes.io/aws-load-balancer-type: "nlb"
  service.beta.kubernetes.io/aws-load-balancer-backend-protocol: "http"
spec: type: LoadBalancer selector: app: web ports: - port: 80
  targetPort: 8080
Google Cloud Load Balancer:

```

```

metadata:
  annotations:
    cloud.google.com/load-balancer-type: "External"
    cloud.google.com/backend-config: '{"default": "my-backendconfig"}'

```

Imperative Commands for Networking

Service Management Commands

```

# Create services
kubectl expose deployment nginx --port=80 --target-port=8080 --type=ClusterIP
kubectl expose pod nginx --port=80 --type=NodePort kubectl create service
loadbalancer web --tcp=80:8080

```

```

# View services kubectl
get services kubectl
get svc -o wide
kubectl describe service web-service

```

```

# Edit services
kubectl edit service web-service
kubectl patch service web-service -p '{"spec":{"type":"LoadBalancer"}}'

```

```
# Delete services kubectl delete  
service web-service
```

Ingress Management Commands

```
# Create ingress
```

```
kubectl create ingress web-ingress --rule="example.com/api*=api-service:80"  
kubectl create ingress tls-ingress --  
rule="secure.com/*=webservice:80,tls=tls-secret"
```

```
# View ingress kubectl
```

```
get ingress  
kubectl describe ingress web-ingress
```

```
# Edit ingress
```

```
kubectl edit ingress web-ingress  
kubectl annotate ingress web-ingress  
nginx.ingress.kubernetes.io/rewritetarget=/
```

```
# Delete ingress kubectl delete  
ingress web-ingress
```

Network Policy Management Commands

```
Create network policy (declarative only)
```

```
kubectl apply -f network-policy.yaml
```

```
# View network policies kubectl  
get networkpolicies kubectl get  
netpol  
kubectl describe networkpolicy web-policy
```

```
# Delete network policies kubectl  
delete networkpolicy web-policy kubectl  
delete netpol --all
```

DNS and Service Discovery Commands

```
# DNS testing
```

```
kubectl run test-pod --image=busybox --rm -it -- nslookup web-service kubectl  
exec -it test-pod -- dig web-service.default.svc.cluster.local
```

```
# Service endpoints
```

```
kubectl get endpoints web-service kubectl  
describe endpoints web-service  
  
# DNS debugging  
kubectl exec -it dns-test -- cat /etc/resolv.conf kubectl  
logs -n kube-system deployment/coredns  
  
Gateway API Commands #  
Install Gateway API CRDs  
kubectl kustomize "https://github.com/kubernetes-  
sigs/gatewayapi/config/crd?ref=v1.0.0" | kubectl apply -f -  
  
# View Gateway API resources  
kubectl get gatewayclasses  
kubectl get gateways kubectl  
get httproutes kubectl get  
tcproutes  
  
# Describe Gateway resources kubectl  
describe gatewayclass nginx kubectl  
describe gateway app-gateway kubectl  
describe httproute api-route  
  
# Gateway API debugging kubectl get gateway app-gateway -o yaml  
kubectl get events --field-selector involvedObject.kind=Gateway
```

Declarative Configurations

Complete Network Stack Example

```
# Namespace  
apiVersion: v1 kind:  
Namespace metadata:  
  name: ecommerce  
labels:  
  environment: production  
  
---  
# Network Policy - Default deny  
apiVersion: networking.k8s.io/v1  
kind: NetworkPolicy metadata:  
  name: default-deny  
  namespace: ecommerce  
spec: podSelector:
```

```
{} policyTypes: -  
Ingress  
- Egress  
  
---  
# Network Policy - Web tier  
apiVersion: networking.k8s.io/v1  
kind: NetworkPolicy metadata:  
name: web-policy namespace:  
ecommerce spec: podSelector:  
matchLabels: tier: web  
policyTypes: - Ingress -  
Egress ingress:  
- from:  
  namespaceSelector:  
  matchLabels:  
  name: ingress-nginx  
  ports:  
    - protocol: TCP  
    port: 80 egress:  
- to:  
  podSelector:  
  matchLabels:  
  tier: api ports:  
  protocol: TCP port:  
  8080 - to: {} # Allow  
  DNS ports:  
  protocol: UDP port:  
  53  
  
--- # ClusterIP  
Service apiVersion:  
v1 kind: Service  
metadata:  
  name: web-service  
namespace: ecommerce  
spec: selector:  
  app: web tier: web  
ports:  
  - port: 80  
targetPort: 80 type:  
ClusterIP  
  
---
```

```
# LoadBalancer Service apiVersion: v1 kind: Service metadata:
name: web-external namespace: ecommerce annotations:
service.beta.kubernetes.io/aws-load-balancer-type: "nlb"
spec: selector: app: web tier: web ports: -
port: 80 targetPort: 80 type: LoadBalancer

--- #
Ingress
apiVersion: networking.k8s.io/v1 kind: Ingress
metadata: name: ecommerce-ingress namespace:
ecommerce annotations:
nginx.ingress.kubernetes.io/rewrite-target: /
nginx.ingress.kubernetes.io/ssl-redirect: "true"
cert-manager.io/cluster-issuer: "letsencrypt-prod"
spec: tls: - hosts:
- shop.example.com -
api.example.com
secretName: ecommerce-tls
rules:
- host:
shop.example.com http:
paths: - path: /
pathType: Prefix
backend: service:
name: web-service
port:
number: 80
- host: api.example.com
http: paths: -
path: /v1
pathType: Prefix
backend:
service:
name: api-v1-service
port: number: 8080
```

Troubleshooting Networking Issues

Common Networking Problems

Service Discovery Issues # Check service endpoints kubectl get endpoints

```
service-name kubectl describe service
service-name

# Test DNS resolution
kubectl run debug --image=busybox --rm -it -- nslookup service-name kubectl
run debug --image=busybox --rm -it -- wget -qO- http://service-name

# Check CoreDNS
kubectl logs -n kube-system deployment/coredns kubectl
get configmap coredns -n kube-system -o yaml

Network Policy Problems # Check
network policies kubectl get
networkpolicies -A
kubectl describe networkpolicy policy-name

# Test connectivity
kubectl exec -it source-pod -- nc -zv target-service 80 kubectl
exec -it source-pod -- curl -v http://target-service

# Debug network policy kubectl get pods --show-
labels kubectl describe pod pod-name | grep -A 5
Labels

Ingress Issues
# Check ingress controller kubectl
get pods -n ingress-nginx
kubectl logs -n ingress-nginx deployment/ingress-nginx-controller

# Check ingress resources kubectl
get ingress -A
kubectl describe ingress ingress-name

# Test external access
curl -H "Host: example.com" http://ingress-ip/path
kubectl port-forward -n ingress-nginx service/ingress-nginx-controller
8080:80
```

Network Diagnostics Tools

Network Testing Pod

```
apiVersion: v1
```

```
kind: Pod
metadata:
  name: network-debug spec:
containers: - name: debug
image: nicolaka/netshoot
command: ["sleep", "3600"]
securityContext:
capabilities: add:
["NET_ADMIN"]
```

Common Debugging Commands

```
# Inside debug pod nslookup
service-name
dig service-name.namespace.svc.cluster.local
nc -zv service-name port traceroute service-
ip tcpdump -i any port 80
```

```
# Pod connectivity ping
pod-ip telnet service-
name port
wget -qO- http://service-name/health
```

```
# Network interface information
ip addr show ip route show
netstat -tlnp ss -tlnp
```

Key Networking Commands Summary

Essential Service Commands

```
# Service Lifecycle
kubectl create service clusterip web --tcp=80:8080
kubectl expose deployment app --port=80 --type=NodePort
kubectl get services -o wide kubectl describe service
service-name kubectl delete service service-name
```

Essential Ingress Commands

```
# Ingress management
kubectl create ingress web --rule="host/path*=service:port"
kubectl get ingress -A kubectl describe ingress ingress-
name kubectl edit ingress ingress-name
```

Essential Network Policy Commands

```
# Network policy management kubectl apply  
-f network-policy.yaml kubectl get  
networkpolicies -A kubectl describe  
networkpolicy policy-name kubectl delete  
networkpolicy policy-name
```

Essential DNS Commands # DNS

testing and debugging

```
kubectl run test --image=busybox --rm -it -- nslookup service-name  
kubectl exec pod -- dig service.namespace.svc.cluster.local kubectl  
logs -n kube-system deployment/coredns
```

Essential Gateway API Commands

```
# Gateway API resources kubectl get  
gatewayclasses kubectl get gateways -  
A kubectl get httproutes -A kubectl  
describe gateway gateway-name kubectl  
describe httproute route-name
```

Design and Install a Kubernetes Cluster HA (High Availability)

High Availability Overview

What is Kubernetes High Availability?

High Availability (HA) in Kubernetes ensures that the cluster remains operational even when individual components fail. HA Kubernetes clusters eliminate single points of failure and provide continuous service availability for production workloads.[1]

Why HA is Critical for Production

- **Business continuity:** Prevents costly downtime and service disruptions
- **Data protection:** Maintains cluster state and application data integrity
- **Scalability:** Supports enterprise-scale deployments up to 5,000 nodes[2]
- **Fault tolerance:** Automatic recovery from component failures
- **Regulatory compliance:** Meets enterprise availability requirements

HA Topology Options

Stacked Control Plane Topology

Stacked topology co-locates etcd members with control plane nodes:[3]

```
# Stacked HA Configuration apiVersion:
kubeadm.k8s.io/v1beta4 kind: ClusterConfiguration
kubernetesVersion: stable controlPlaneEndpoint:
"loadbalancer.example.com:6443" etcd: local:
dataDir: "/var/lib/etcd" Advantages:
```

- **Simpler setup:** Fewer infrastructure components to manage
- **Lower cost:** Requires fewer nodes than external etcd topology
- **Easier management:** Co-located components simplify administration
- **Default approach:** Standard kubeadm topology for HA clusters[3] **Disadvantages:**
- **Coupled failure risk:** Loss of one node affects both etcd and control plane
- **Resource contention:** etcd and API server compete for resources
- **Limited scalability:** Not ideal for very large clusters

External etcd Topology

External etcd topology separates etcd cluster from control plane nodes:[3]

```
# External etcd Configuration apiVersion:
kubeadm.k8s.io/v1beta4 kind: ClusterConfiguration
kubernetesVersion: stable controlPlaneEndpoint:
"loadbalancer.example.com:6443" etcd: external:
endpoints:
  - "https://10.100.0.10:2379"
  - "https://10.100.0.11:2379"      -
    "https://10.100.0.12:2379"      caFile:
      "/etc/kubernetes/pki/etcd/ca.crt"
    certFile: "/etc/kubernetes/pki/apiserver-etcd-client.crt"
keyFile: "/etc/kubernetes/pki/apiserver-etcd-client.key" Advantages:
```

- **Reduced risk:** Control plane and etcd failures are decoupled
- **Better performance:** Dedicated resources for etcd operations
- **Higher scalability:** Better suited for large-scale deployments
- **Independent maintenance:** Can maintain etcd and control plane separately

Disadvantages:

- **Complex setup:** Requires additional infrastructure and configuration
- **Higher cost:** More nodes required for full HA setup
- **Management overhead:** Multiple clusters to monitor and maintain

Design Considerations

Infrastructure Requirements

Based on cluster size and workload demands:[4]

Nodes	GCP Instance	AWS Instance	Use Case
1-5	N1-standard-1 (1 vCPU, 3.75GB)	M3.medium (1 vCPU, 3.75GB)	Development/Testing
6-10	N1-standard-2 (2 vCPU, 7.5GB)	M3.large (2 vCPU, 7.5GB)	Small Production
11-100	N1-standard-4 (4 vCPU, 15GB)	M3.xlarge (4 vCPU, 15GB)	Medium Production 15GB)
101-250	N1-standard-8 (8 vCPU, 30GB)	M3.2xlarge (8 vCPU, 30GB)	Large Production
251-500	N1-standard-16 (16 vCPU, 60GB)	C4.4xlarge (16 vCPU, 30GB)	Enterprise
>500	N1-standard-32 (32 vCPU, 120GB)	C4.8xlarge (36 vCPU, 60GB)	Hyperscale

Network Architecture Planning

Load Balancer Configuration:

```
# HAProxy Load Balancer Configuration
global      daemon

defaults    mode http
timeout connect 5000ms
timeout client 50000ms
timeout server 50000ms

frontend kubernetes
bind *:6443    option
tcplog      mode tcp
    default_backend kubernetes-master-nodes

backend kubernetes-master-nodes    mode tcp    balance
roundrobin    option tcp-check    server master1
192.168.5.11:6443 check fall 3 rise 2    server master2
```

```
192.168.5.12:6443 check fall 3 rise 2      server master3
192.168.5.13:6443 check fall 3 rise 2
```

Node Distribution Strategy

Minimum HA Requirements:

- **Control Plane:** 3 master nodes for quorum
- **Worker Nodes:** Minimum 2 workers for workload distribution
- **Load Balancer:** 1-2 load balancers for API access
- **etcd Cluster:** 3 or 5 nodes for data consistency

Storage Considerations

High-Performance Storage Requirements:[2]

- **SSD-backed storage:** For high-performance applications
- **Network-based storage:** For multiple concurrent connections
- **Persistent volumes:** Shared access across multiple pods
- **Backup strategy:** Regular etcd and application data backups **etcd High**

Availability etcd Cluster Fundamentals etcd is a distributed key-value store that

maintains cluster state:[4]

```
# etcd Service Configuration
ExecStart=/usr/local/bin/etcd \
--name ${ETCD_NAME} \
--cert-file=/etc/etcd/kubernetes.pem \
--key-file=/etc/etcd/kubernetes-key.pem \
--peer-cert-file=/etc/etcd/kubernetes.pem \
--peer-key-file=/etc/etcd/kubernetes-key.pem \
--trusted-ca-file=/etc/etcd/ca.pem \
--peer-trusted-ca-file=/etc/etcd/ca.pem \
--peer-client-cert-auth \
--client-cert-auth \
--initial-advertise-peer-urls https://${INTERNAL_IP}:2380 \
--listen-peer-urls https://${INTERNAL_IP}:2380 \
--listen-client-urls https://${INTERNAL_IP}:2379,https://127.0.0.1:2379 \
--advertise-client-urls https://${INTERNAL_IP}:2379 \
--initial-cluster-token etcd-cluster-0 \
```

```
--initial-cluster controller-0=https://${CONTROLLER0_IP}:2380,controller-1=https://${CONTROLLER1_IP}:2380,controller-2=https://${CONTROLLER2_IP}:2380 \
--initial-cluster-state new \
--data-dir=/var/lib/etcd    etcd
```

Quorum and Fault Tolerance

Quorum Requirements:[4]

Instances	Quorum	Fault Tolerance	Recommendation
-----------	--------	-----------------	----------------

1	1 0	Development only	
2	2 0	Not recommended	
3	2 1	Minimum for production	
4	3 1	Same as 3, higher cost	
5	3 2	Recommended for critical workloads	
6	4 2	Same as 5, higher cost	7 4 3 Maximum practical size

etcd Best Practices

Optimal Configuration:

- **Odd numbers:** Always use odd number of etcd instances
- **Fast storage:** Use SSD storage for etcd data directory
- **Network latency:** Keep etcd nodes in same data center
- **Resource isolation:** Dedicated CPU and memory for etcd
- **Regular backups:** Automated etcd snapshot backups

Control Plane High Availability

API Server HA Configuration

Multiple API Server Setup:[4]

```
# API Server Service Configuration
ExecStart=/usr/local/bin/kube-apiserver \
--advertise-address=${INTERNAL_IP} \
--allow-privileged=true \
--apiserver-count=3 \
```

```
--audit-log-maxage=30 \
--audit-log-maxbackup=3 \
--audit-log-maxsize=100 \
--audit-log-path=/var/log/audit.log \
--authorization-mode=Node,RBAC \
--bind-address=0.0.0.0 \
--client-ca-file=/var/lib/kubernetes/ca.pem \
--enable-admission-
plugins=NamespaceLifecycle,NodeRestriction,LimitRanger,ServiceAccount,Default
StorageClass,ResourceQuota \
--etcd-cafile=/var/lib/kubernetes/ca.pem \
--etcd-certfile=/var/lib/kubernetes/kubernetes.pem \
--etcd-keyfile=/var/lib/kubernetes/kubernetes-key.pem \
--etcd-
servers=https://10.240.0.10:2379,https://10.240.0.11:2379,https://10.240.0.12
:2379 \
--event-ttl=1h \
--encryption-provider-config=/var/lib/kubernetes/encryption-config.yaml \
--kubelet-certificate-authority=/var/lib/kubernetes/ca.pem \
--kubelet-client-certificate=/var/lib/kubernetes/kubernetes.pem \
--kubelet-client-key=/var/lib/kubernetes/kubernetes-key.pem \
--runtime-config='api/all=true' \
--service-account-key-file=/var/lib/kubernetes/service-account.pem \
--service-account-signing-key-file=/var/lib/kubernetes/service-accountkey.pem \
--service-account-issuer=https://${KUBERNETES_PUBLIC_ADDRESS}:6443 \
--service-cluster-ip-range=10.32.0.0/24 \
--service-node-port-range=30000-32767 \
--tls-cert-file=/var/lib/kubernetes/kubernetes.pem \
--tls-private-key-file=/var/lib/kubernetes/kubernetes-key.pem \
v=2
```

Controller Manager HA with Leader Election

Leader Election Configuration:[4]

```
# Controller Manager Service
ExecStart=/usr/local/bin/kube-controller-manager \
--bind-address=0.0.0.0 \
--cluster-cidr=10.200.0.0/16 \
--cluster-name=kubernetes \
--cluster-signing-cert-file=/var/lib/kubernetes/ca.pem \
```

```
--cluster-signing-key-file=/var/lib/kubernetes/ca-key.pem \
--kubeconfig=/var/lib/kubernetes/kube-controller-manager.kubeconfig \
--leader-elect=true \
--leader-elect-lease-duration=15s \
--leader-elect-renew-deadline=10s \
--leader-elect-retry-period=2s \
--root-ca-file=/var/lib/kubernetes/ca.pem \
--service-account-private-key-file=/var/lib/kubernetes/service-
accountkey.pem \
--service-cluster-ip-range=10.32.0.0/24 \
--use-service-account-credentials=true \
--v=2
```

Scheduler HA with Leader Election

Scheduler Leader Election:[4]

```
# Scheduler Service
ExecStart=/usr/local/bin/kube-scheduler \
--config=/etc/kubernetes/config/kube-scheduler.yaml \
--leader-elect=true \
--leader-elect-lease-duration=15s \
--leader-elect-renew-deadline=10s \
--leader-elect-retry-period=2s \
--v=2
```

Installation Process

Prerequisites and Planning

Infrastructure Preparation:

1. Node Requirements:[4]

- **Operating System:** Linux x86_64 architecture
- **Container Runtime:** Docker 1.11.1, 1.12.1, 1.13.1, 17.03, 17.06, 17.09, 18.06
- **Network:** All nodes must communicate on required ports
- **Storage:** SSD storage recommended for etcd

2. Network Configuration:

- **Pod Network:** 10.244.0.0/16 (default)

- **Service Network:** 10.96.0.0/12 (default)
- **DNS:** CoreDNS for service discovery

Step 1: Provision Infrastructure

Using Vagrant for Lab Setup:[4]

```
# Vagrantfile for HA Cluster
Vagrant.configure("2") do |config|
  # Load Balancer
  config.vm.define "loadbalancer" do |lb|
    lb.vm.box = "ubuntu/bionic64"      lb.vm.hostname
    = "loadbalancer"
    lb.vm.network "private_network", ip: "192.168.5.30"
    lb.vm.provider "virtualbox" do |v|          v.name =
    "loadbalancer"
      v.memory = 1024
      v.cpus = 1
    end  end

    # Master Nodes
    (1..3).each do |i|
      config.vm.define "master-#{i}" do |master|
        master.vm.box = "ubuntu/bionic64"      master.vm.hostname
        = "master-#{i}"      master.vm.network "private_network",
        ip: "192.168.5.1#{i}"      master.vm.provider
        "virtualbox" do |v|          v.name = "master-#{i}"
          v.memory = 2048
          v.cpus = 2
        end  end  end

    # Worker Nodes
    (1..2).each do |i|
      config.vm.define "worker-#{i}" do |worker|
        worker.vm.box = "ubuntu/bionic64"
        worker.vm.hostname = "worker-#{i}"
        worker.vm.network "private_network", ip: "192.168.5.2#{i}"
        worker.vm.provider "virtualbox" do |v|          v.name =
        "worker-
        #{i}"
          v.memory = 2048
          v.cpus = 2
        end  end
      end
    end
  end
end
```

Step 2: Install Container Runtime

Docker Installation on All Nodes:

```
# Install Docker
curl -fsSL https://download.docker.com/linux/ubuntu/gpg | sudo apt-key add -
sudo add-apt-repository "deb [arch=amd64]
https://download.docker.com/linux/ubuntu $(lsb_release -cs) stable" sudo
apt-get update
sudo apt-get install -y docker-ce=5:18.09.7~3-0~ubuntu-bionic

# Configure Docker daemon
sudo tee /etc/docker/daemon.json apiserver-csr.conf kubeadm-config.yaml
apiVersion:     kubeadm.k8s.io/v1beta4    kind:      ClusterConfiguration
kubernetesVersion:  v1.21.0   controlPlaneEndpoint:  "192.168.5.30:6443"
networking:
  podSubnet: "10.244.0.0/16" etcd:
    external:
endpoints:
- "https://192.168.5.11:2379"
- "https://192.168.5.12:2379"    -
  "https://192.168.5.13:2379"    caFile:
    "/etc/kubernetes/pki/etcd/ca.crt"    certFile:
    "/etc/kubernetes/pki/apiserver-etcd-client.crt"
    keyFile: "/etc/kubernetes/pki/apiserver-etcd-client.key"
---
apiVersion: kubeadm.k8s.io/v1beta4
kind: InitConfiguration
localAPIEndpoint:
advertiseAddress: "192.168.5.11"
  bindPort: 6443
EOF

# Initialize first master node sudo kubeadm init --config
kubeadm-config.yaml --upload-certs
```

Step 6: Configure Load Balancer

HAProxy Configuration for API Server:

```
# Install HAProxy sudo apt-get
update sudo apt-get install -y
haproxy
```

```
# Configure HAProxy cat \
  --discovery-token-ca-cert-hash \
  --control-plane \
  --certificate-key
```

Step 8: Configure Networking

Install CNI Plugin (Flannel):

```
# Apply Flannel CNI
kubectl apply -f
https://raw.githubusercontent.com/flannelio/flannel/master/Documentation/kube
-flannel.yml
```

Step 9: Join Worker Nodes

Adding Worker Nodes to Cluster:

```
# On worker nodes sudo kubeadm join
192.168.5.30:6443 \
  --token \
  --discovery-token-ca-cert-hash
```

Validation and Testing

Cluster Health Verification Basic

Health Checks:

```
# Check node status kubectl
get nodes -o wide # Check
control plane pods
kubectl get pods -n kube-system

# Check etcd cluster health kubectl -n kube-system exec
etcd-master-1 -- etcdctl \
  --ca-file=/etc/kubernetes/pki/etcd/ca.crt \
  --cert-file=/etc/kubernetes/pki/etcd/server.crt \
  --key-file=/etc/kubernetes/pki/etcd/server.key \
  --endpoints=https://127.0.0.1:2379 \
  endpoint
health
```

```
# Test API server accessibility curl -k  
https://192.168.5.30:6443/healthz
```

Application Deployment Testing Deploy

Test Application:

```
# Create test deployment kubectl create deployment test-nginx -  
-image=nginx --replicas=3
```

```
# Expose deployment kubectl expose deployment test-nginx --  
port=80 --type=NodePort
```

```
# Scale deployment kubectl scale deployment  
test-nginx --replicas=6
```

```
# Check pod distribution kubectl  
get pods -o wide
```

Failure Testing

Simulate Node Failures:

```
# Stop one master node sudo  
systemctl stop kubelet  
sudo systemctl stop docker
```

```
# Verify cluster continues to operate kubectl  
get nodes
```

```
# Test application availability kubectl  
get pods
```

```
# Restart failed node sudo  
systemctl start docker sudo  
systemctl start kubelet
```

Production Best Practices

Security Hardening

Essential Security Measures:

- RBAC: Enable role-based access control

- **Network Policies:** Implement pod-to-pod communication rules
- **TLS:** Use TLS for all component communication
- **Encryption:** Enable etcd encryption at rest
- **Regular Updates:** Keep Kubernetes components updated

Monitoring and Alerting

Monitoring Stack Setup:

```
# Prometheus monitoring configuration
apiVersion: v1 kind: ConfigMap
metadata: name: prometheus-config
namespace: monitoring data:
prometheus.yml: | global:
scrape_interval: 15s
scrape_configs:
  - job_name: 'kubernetes-apiservers' kubernetes_sd_configs: -
role: endpoints scheme: https tls_config: ca_file:
/var/run/secrets/kubernetes.io/serviceaccount/ca.crt bearer_token_file:
/var/run/secrets/kubernetes.io/serviceaccount/token Backup and Disaster Recovery
etcd Backup Strategy:
```

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

```
# Verify backup
ETCDCTL_API=3 etcdctl --write-out=table snapshot status snapshot.db
```

```
# Schedule automated backups
cat << EOF | sudo tee /etc/cron.d/etcd-backup
0 2 * * * root /usr/local/bin/backup-etcd.sh
EOF
```

Troubleshooting Common Issues

etcd Cluster Issues

Common etcd Problems:

```
# Check etcd cluster status
ETCDCTL_API=3 etcdctl endpoint status \
  --
endpoints=https://192.168.5.11:2379,https://192.168.5.12:2379,https://192.168.5.13:2379 \
  --cacert=/etc/kubernetes/pki/etcd/ca.crt \
  --cert=/etc/kubernetes/pki/etcd/server.crt \
  --key=/etc/kubernetes/pki/etcd/server.key

# Check etcd member list
ETCDCTL_API=3 etcdctl member list \
  --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

# View etcd logs sudo
journalctl -u etcd -f
```

API Server Connectivity Issues

Load Balancer Troubleshooting:

```
# Test Load balancer connectivity nc
-zv 192.168.5.30 6443

# Check HAProxy status sudo
systemctl status haproxy

# View HAProxy Logs sudo tail -f
/var/log/haproxy.log
```

```
# Test direct API server access curl -k
https://192.168.5.11:6443/healthz curl -k
https://192.168.5.12:6443/healthz curl -k
https://192.168.5.13:6443/healthz
```

Control Plane Recovery

Master Node Recovery Procedures:

```
# Recover from control plane failure sudo kubeadm init phase control-plane all --config kubeadm-config.yaml
```

```
# Recreate admin kubeconfig sudo kubeadm init phase kubeconfig admin  
--config kubeadm-config.yaml
```

```
# Restart kubelet sudo  
systemctl restart kubelet Key
```

Commands Summary

Installation Commands # Initialize HA cluster sudo kubeadm init -
-config kubeadm-config.yaml --upload-certs

Join control plane node
sudo kubeadm join LOAD_BALANCER:6443 --token TOKEN --control-plane -
certificate-key KEY

Join worker node
sudo kubeadm join LOAD_BALANCER:6443 --token TOKEN --discovery-token-ca-
certhash HASH

Generate new join token kubeadm token
create --print-join-command

Health Check Commands

Cluster status kubectl
cluster-info kubectl get
nodes
kubectl get componentstatuses

Control plane health kubectl
get pods -n kube-system kubectl
top nodes

etcd health kubectl -n kube-system exec etcd-master-1 -- etcdctl
endpoint health

Maintenance Commands # Drain node for maintenance kubectl drain
node-name --ignore-daemonsets --delete-emptydir-data

Uncordon node
kubectl uncordon node-name

```
# Upgrade cluster sudo kubeadm
upgrade plan sudo kubeadm upgrade
apply v1.21.1
```

Helm and Kustomize - CKAD Essentials

Helm - Package Manager for Kubernetes

What is Helm?

Helm is a package manager for Kubernetes that simplifies deploying and managing applications using templates and reusable packages called **Charts**.

Why Use Helm?

- **Templating:** Create reusable Kubernetes manifests
- **Package Management:** Install, upgrade, and rollback applications easily
- **Version Control:** Track application releases
- **Configuration Management:** Separate config from application logic

Helm Key Concepts

- **Chart:** A Helm package containing Kubernetes templates
- **Release:** An instance of a chart deployed to Kubernetes
- **Values:** Configuration data for charts
- **Repository:** A collection of charts

Basic Helm Usage

Installing Helm

Install Helm

```
curl https://raw.githubusercontent.com/helm/helm/main/scripts/get-helm-3 | bash
```

```
# Verify installation helm
version
```

Working with Repositories

Add popular repositories

```
helm repo add bitnami https://charts.bitnami.com/bitnami helm
repo add nginx https://helm.nginx.com/stable
```

```
# Update repositories helm
repo update
```

```
# Search for charts helm
search repo nginx helm
search repo mysql
```

Essential Helm Commands

```
# Install a chart helm install
myapp bitnami/nginx # Install
with custom values helm install
myapp bitnami/nginx --set
replicaCount=3
```

```
# List releases helm
list
```

```
# Upgrade release helm upgrade myapp bitnami/nginx --
set image.tag=1.21
```

```
# Rollback release helm
rollback myapp 1
```

```
# Uninstall release helm
uninstall myapp
```

```
# Get release info helm
status myapp helm get values
```

Creating Simple Helm Charts

```
Chart Structure #
Create new chart
helm create webapp
```

```
# Basic chart structure webapp/
├── Chart.yaml          # Chart metadata
├── values.yaml         # Default values
└── templates/           # Kubernetes templates
    ├── deployment.yaml
    └── service.yaml
```

```
└── _helpers.tpl
```

Chart.yaml (Metadata)

```
apiVersion: v2
name: webapp
description: Simple web application
version: 0.1.0
appVersion: "1.0.0"
```

values.yaml (Configuration)

```
# Simple configuration
replicaCount:
  3
```

```
image:
  repository: nginx
  tag: "1.21.0"
  pullPolicy: IfNotPresent
  service:
    type: ClusterIP
  port: 80
```

```
# Application config
app:
  debug: false
  logLevel: "info"
```

Simple Deployment Template #

```
templates/deployment.yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: {{ .Chart.Name }}
labels:
  app: {{ .Chart.Name }}
spec:
  replicas: {{ .Values.replicaCount }}
  selector:
    matchLabels:
      app: {{ .Chart.Name }}
  template:
    metadata:
      labels:
        app: {{ .Chart.Name }}
    spec:
      containers:
        - name: {{ .Chart.Name }}
          image: "{{ .Values.image.repository }}:{{ .Values.image.tag }}"
  imagePullPolicy: {{ .Values.image.pullPolicy }}
  ports:
    - containerPort: 80
      env:
        - name: DEBUG
          value: "{{ .Values.app.debug }}"
```

```
- name: LOG_LEVEL           value: "{{  
  .Values.app.logLevel }}"  
  
Basic Template Functions #  
Common template functions  
{{ .Values.name | upper }}          # Uppercase  
{{ .Values.name | lower }}          # Lowercase  
{{ .Values.replicas | int }}        # Convert to integer  
{{ .Values.tag | default "latest" }} # Default value  
  
# Conditionals  
{%- if .Values.ingress.enabled %}  
# ingress configuration  
{%- end %}  
  
# Loops  
{%- range .Values.environments %}  
- name: {{ .name }}  
value: {{ .value }}{%-  
end %}  
  
Working with Your Chart # Test  
template rendering helm  
template webapp ./webapp/  
  
# Install your chart helm  
install myapp ./webapp/  
  
# Install with custom values helm install myapp  
. /webapp/ --set replicaCount=5  
  
# Package chart helm  
package webapp/
```

Kustomize - Kubernetes Native Configuration

What is Kustomize?

Kustomize is a Kubernetes-native tool for customizing YAML configurations without templates. It uses an overlay approach to modify base configurations.

Why Use Kustomize?

- **Template-free:** Works with standard Kubernetes YAML
- **Overlay approach:** Modify configs without changing originals
- **Built into kubectl:** No extra tools needed
- **GitOps friendly:** Easy version control

Basic Kustomize Structure

```
# Simple project structure app/
└── base/                      # Base configuration
    ├── kustomization.yaml      # Base kustomization file
    ├── deployment.yaml         # Base deployment
    └── service.yaml            # Base service
└── overlays/                   # Environment-specific changes
    ├── development/
    │   └── kustomization.yaml
    └── production/
        └── kustomization.yaml
```

Basic Kustomize Usage

Base Configuration

base/kustomization.yaml:

```
apiVersion: kustomize.config.k8s.io/v1beta1 kind: Kustomization
resources: - deployment.yaml
           - service.yaml
```

```
commonLabels: app:
webapp namePrefix:
webapp-
```

base/deployment.yaml:

```
apiVersion: apps/v1
kind: Deployment
metadata: name:
app spec:
  replicas: 3
selector:
matchLabels:
  app: webapp
```

```
template:
metadata:
labels:
app: webapp
spec:
  containers: -
name: app           image:
nginx:1.20.0       ports:
- containerPort: 80
base/service.yaml:

apiVersion: v1 kind:
Service metadata:
  name: app-service
spec: selector:
app: webapp
ports: - port: 80
      targetPort: 80      type: ClusterIP

Development          Overlay

overlays/development/kustomization.yaml:

apiVersion: kustomize.config.k8s.io/v1beta1 kind:
Kustomization

# Include base configuration
resources: - ../../base

# Development-specific settings
namespace: development namePrefix:
dev-

# Change image tag for development
images: - name: nginx
  newTag: latest

# Override replica count
replicas: - name: app
count: 1

# Add development Labels
commonLabels:
environment: development Production
```

Overlay

overlays/production/kustomization.yaml:

```
apiVersion: kustomize.config.k8s.io/v1beta1 kind: Kustomization

resources: - ../../base

# Production settings namespace:
production namePrefix: prod-

# Use specific image version
images: - name: nginx
newTag: 1.21.0

# Production replica count
replicas: - name: app
count: 5

# Production Labels
commonLabels: environment:
production
```

Essential Kustomize Operations

Basic Commands # Preview configuration
kubectl kustomize ./overlays/development/

Apply configuration kubectl apply -k
./overlays/development/
kubectl apply -k ./overlays/production/

Delete resources kubectl delete -k
./overlays/development/

Common Transformers

Change image tags
images: - name:
nginx newTag:
1.21.0

```
# Modify replica counts
replicas: - name: app-
deployment      count: 5

# Add name prefix/suffix
namePrefix: dev- nameSuffix:
-v1

# Set namespace namespace:
development

# Add common Labels
commonLabels:
app: myapp
environment: dev

# Add common annotations
commonAnnotations: description:
"Development deployment"

Simple Patches
# overlays/production/patch-resources.yaml
apiVersion: apps/v1 kind: Deployment
metadata: name: app spec: template:
spec: containers: - name: app
resources: limits:
cpu: 500m           memory: 512Mi
requests:          cpu: 250m
memory: 256Mi

# Include patch in kustomization.yaml
patchesStrategicMerge: - patch-
resources.yaml
```

ConfigMaps and Secrets with Kustomize

```
Generate ConfigMaps #
From Literal values
configMapGenerator:
- name: app-config
literals:
- DEBUG=true
- LOG_LEVEL=info
```

```
# From files
configMapGenerator: -
  name: app-config
  files:
    - config.properties
    - app.yaml
```

```
Generate Secrets # From
literal values
secretGenerator: - name:
  app-secrets literals:
  - username=admin
  password=secret123
  type: Opaque
```

When to Use Helm vs Kustomize

Use Helm When:

- Need package management and distribution
- Complex templating requirements
- Want to use existing Helm charts
- Need dependency management
- Require release management features

Use Kustomize When:

- Managing multiple environments
- Want simple configuration overlays
- Prefer template-free approach
- Working with standard Kubernetes YAML
- Need built-in kubectl integration **Common Patterns**

Environment Management with Kustomize

```
# Development kubectl apply -k
./overlays/development/
```

```
# Staging kubectl apply -k
./overlays/staging/
```

```
# Production kubectl apply -k
./overlays/production/
```

Application Distribution with Helm

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

# Install application helm install
myapp mycompany/webapp

# Upgrade application helm upgrade myapp
mycompany/webapp --version 2.0.0
```

Key Commands

Summary

Helm Commands # Repository

```
management helm repo add
<name> <url> helm repo
update helm search repo
<term>
```

Release management

```
helm install <name> <chart>
helm upgrade <name> <chart>
helm rollback <name> <revision>
helm uninstall <name> helm list
```

Chart development helm

```
create <name> helm template
<name> <chart> helm package
<chart>
```

Kustomize Commands

```
# Build and apply kubectl
kustomize <directory> kubectl
apply -k <directory>
kubectl delete -k <directory>
```

Validation kubectl apply -k <directory> --

dry-run=client Best Practices

Helm Best Practices

- Use semantic versioning for charts
- Include resource limits and probes

- Organize values logically
- Test templates before releasing
- Document chart usage

Kustomize Best Practices

- Keep base configuration minimal
- Use meaningful directory structure
- Apply consistent labeling
- Use overlays for environment differences
- Version control all configurations