1 KCNA

Generated on: 2026-01-13 15:04:14 **Version:** 1.0

1.1 Table of Contents

1.2 Overview

The **Kubernetes and Cloud Native Associate (KCNA)** exam demonstrates a user's foundational knowledge and skills in Kubernetes and the wider cloud native ecosystem.

1.3 Exam Overview

Detail	Information
Exam Format	Multiple Choice
Number of Questions	60
Duration	90 minutes
Passing Score	75%
Certification Validity	3 years
Cost	$250 USD
Retake Policy	1 free retake

1.4 Exam Domains & Weights

Domain	Weight
Kubernetes Fundamentals	46%
Container Orchestration	22%
Cloud Native Architecture	16%
Cloud Native Observability	8%
Cloud Native Application Delivery	8%

1.5 Prerequisites

- Basic understanding of Linux command line
- Familiarity with containers (Docker)
- General IT knowledge

1.6 Study Resources

1.6.1 Official Resources

- KCNA Exam Curriculum
- Kubernetes Documentation

- CNCF Landscape

1.6.2 Recommended Courses

- Kubernetes and Cloud Native Essentials (LFS250)
- Introduction to Kubernetes (LFS158)

1.6.3 Practice Resources

- Kubernetes Basics Tutorial
- Play with Kubernetes

1.7 Quick Navigation

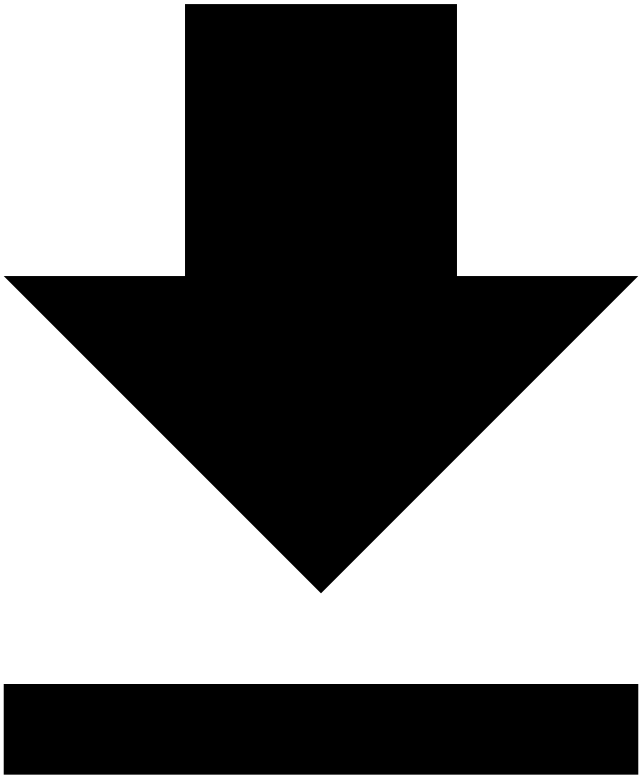
1.8 Exam Tips

1. **Understand the basics** - This is a foundational exam, focus on concepts
2. **Know the CNCF ecosystem** - Understand graduated, incubating, and sandbox projects
3. **Review Kubernetes architecture** - Control plane, worker nodes, and components
4. **Practice with kubectl** - Even though it's multiple choice, hands-on helps understanding
5. **Time management** - 90 seconds per question on average

1.9 Registration

Register for KCNA Exam

1.10 Kubernetes Fundamentals



Download PDF Version

This domain covers the core concepts of Kubernetes and represents the largest portion of the KCNA exam.

1.11 Kubernetes Architecture

1.11.1 Control Plane Components

Component	Description
kube-apiserver	Front-end for the Kubernetes control plane, exposes the Kubernetes API
etcd	Consistent and highly-available key-value store for cluster data
kube-scheduler	Watches for newly created Pods and assigns them to nodes
kube-controller-manager	

Component	Description
	Runs controller processes (Node, Job, EndpointSlice, ServiceAccount)
cloud-controller-manager	Embeds cloud-specific control logic

1.11.2 Node Components

Component	Description
kubelet	Agent that runs on each node, ensures containers are running in a Pod
kube-proxy	Network proxy that maintains network rules on nodes
Container Runtime	Software responsible for running containers (containerd, CRI-O)

1.12 Kubernetes Objects

1.12.1 Workload Resources

1.12.1.1 Pod

The smallest deployable unit in Kubernetes.

```yaml
apiVersion: v1
kind: Pod
metadata:
  name: nginx-pod
spec:
  containers:
  - name: nginx
    image: nginx:1.21
    ports:
    - containerPort: 80
```

1.12.1.2 Deployment

Manages ReplicaSets and provides declarative updates for Pods.

```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: nginx-deployment
spec:
  replicas: 3
  selector:
    matchLabels:
      app: nginx
```

```
  template:
    metadata:
      labels:
        app: nginx
    spec:
      containers:
      - name: nginx
        image: nginx:1.21
        ports:
        - containerPort: 80
```

1.12.1.3 ReplicaSet

Maintains a stable set of replica Pods running at any given time.

1.12.1.4 StatefulSet

Manages stateful applications with unique network identifiers and persistent storage.

1.12.1.5 DaemonSet

Ensures all (or some) nodes run a copy of a Pod.

1.12.1.6 Job & CronJob

- **Job**: Creates one or more Pods and ensures they successfully terminate
- **CronJob**: Creates Jobs on a repeating schedule

1.12.2 Service & Networking

1.12.2.1 Service Types

Type	Description
ClusterIP	Exposes the Service on a cluster-internal IP (default)
NodePort	Exposes the Service on each Node's IP at a static port
LoadBalancer	Exposes the Service externally using a cloud provider's load balancer
ExternalName	Maps the Service to a DNS name

```
apiVersion: v1
kind: Service
metadata:
  name: nginx-service
spec:
  selector:
    app: nginx
  ports:
  - port: 80
```

```
    targetPort: 80
  type: ClusterIP
```

1.12.2.2 Ingress

Manages external access to services, typically HTTP/HTTPS.

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

1.12.3 Configuration

1.12.3.1 ConfigMap

Stores non-confidential configuration data in key-value pairs.

```
apiVersion: v1
kind: ConfigMap
metadata:
  name: app-config
data:
  DATABASE_HOST: "mysql.default.svc.cluster.local"
  LOG_LEVEL: "info"
```

1.12.3.2 Secret

Stores sensitive information like passwords, tokens, or keys.

```
apiVersion: v1
kind: Secret
metadata:
  name: app-secret
type: Opaque
data:
  password: cGFzc3dvcmQxMjM=  # base64 encoded
```

1.12.4 Storage

1.12.4.1 PersistentVolume (PV)

A piece of storage in the cluster provisioned by an administrator.

1.12.4.2 PersistentVolumeClaim (PVC)

A request for storage by a user.

```yaml
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
  name: my-pvc
spec:
  accessModes:
    - ReadWriteOnce
  resources:
    requests:
      storage: 1Gi
  storageClassName: standard
```

1.13 Namespaces

Namespaces provide a mechanism for isolating groups of resources within a single cluster.

Default Namespaces: - `default` - Default namespace for objects with no other namespace - `kube-system` - For objects created by the Kubernetes system - `kube-public` - Readable by all users, reserved for cluster usage - `kube-node-lease` - For lease objects associated with each node

1.14 Labels and Selectors

1.14.1 Labels

Key-value pairs attached to objects for identification.

```yaml
metadata:
  labels:
    app: nginx
    environment: production
    tier: frontend
```

1.14.2 Selectors

Used to filter objects based on labels.

```yaml
selector:
  matchLabels:
    app: nginx
  matchExpressions:
  - key: environment
    operator: In
    values:
    - production
    - staging
```

1.15 Kubernetes API

1.15.1 API Groups

Group	Resources
core (v1)	Pods, Services, ConfigMaps, Secrets, Namespaces
apps/v1	Deployments, ReplicaSets, StatefulSets, DaemonSets
batch/v1	Jobs, CronJobs
networking.k8s.io/v1	Ingress, NetworkPolicy
rbac.authorization.k8s.io/v1	Roles, ClusterRoles, RoleBindings

1.15.2 API Versioning

- **Alpha (v1alpha1)**: Disabled by default, may be buggy
- **Beta (v1beta1)**: Enabled by default, well-tested
- **Stable (v1)**: Production-ready, backward compatible

1.16 kubectl Basics

1.16.1 Common Commands

```bash
# Get resources
kubectl get pods
kubectl get deployments
kubectl get services
kubectl get nodes

# Describe resources
kubectl describe pod <pod-name>
kubectl describe node <node-name>

# Create resources
kubectl apply -f manifest.yaml
```

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

# Delete resources
kubectl delete pod <pod-name>
kubectl delete -f manifest.yaml

# Logs and debugging
kubectl logs <pod-name>
kubectl exec -it <pod-name> -- /bin/bash

# Scaling
kubectl scale deployment nginx --replicas=5
```

1.17 Key Concepts to Remember

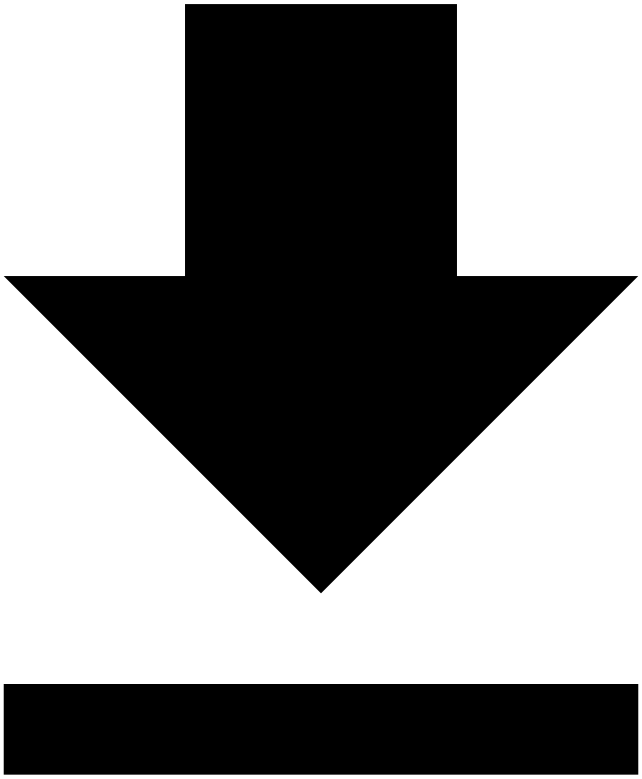
1. **Declarative vs Imperative**: Kubernetes prefers declarative configuration
2. **Desired State**: Controllers continuously work to match actual state to desired state
3. **Self-healing**: Kubernetes automatically replaces failed containers
4. **Horizontal Scaling**: Add more Pods to handle increased load
5. **Service Discovery**: Services provide stable endpoints for Pods
6. **Rolling Updates**: Deployments can update Pods without downtime

1.18 Practice Questions

1. What is the smallest deployable unit in Kubernetes?
2. Which component is responsible for scheduling Pods to nodes?
3. What is the default Service type in Kubernetes?
4. How do you store sensitive data in Kubernetes?
5. What is the purpose of a namespace?

Back to KCNA Overview | Next: Container Orchestration →

1.19 Container Orchestration



Download PDF Version

This domain covers container fundamentals and how Kubernetes orchestrates containerized workloads.

1.20 Container Fundamentals

1.20.1 What is a Container?

A container is a lightweight, standalone, executable package that includes everything needed to run a piece of software:

- Code
- Runtime
- System tools
- System libraries
- Settings

1.20.2 Container vs Virtual Machine

Aspect	Container	Virtual Machine
Size	Megabytes	Gigabytes
Startup	Seconds	Minutes
Isolation	Process-level	Hardware-level
OS	Shares host kernel	Full OS per VM
Resource Usage	Lightweight	Heavy

1.20.3 Container Runtime

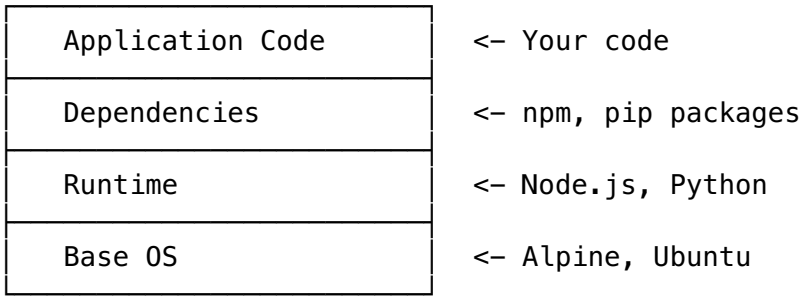
The container runtime is responsible for running containers. Kubernetes supports several runtimes through the Container Runtime Interface (CRI):

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

1.21 Container Images

1.21.1 Image Layers

Container images are built in layers:



1.21.2 Image Registries

Container images are stored in registries:

- **Docker Hub** - Public registry
- **Google Container Registry (GCR)**
- **Amazon Elastic Container Registry (ECR)**
- **Azure Container Registry (ACR)**
- **Harbor** - Open-source private registry

1.21.3 Image Naming Convention

```
registry/repository:tag
```

```
Examples:
docker.io/library/nginx:1.21
gcr.io/my-project/my-app:v1.0.0
my-registry.com/team/service:latest
```

1.22 Kubernetes Scheduling

1.22.1 How Scheduling Works

1. User creates a Pod
2. API Server stores Pod in etcd (status: Pending)
3. Scheduler watches for unscheduled Pods
4. Scheduler selects a suitable node
5. Scheduler updates Pod with node assignment
6. Kubelet on the node creates the container

1.22.2 Scheduling Factors

The scheduler considers:

- **Resource requests and limits**
- **Node selectors and affinity**
- **Taints and tolerations**
- **Pod topology spread constraints**
- **Available resources on nodes**

1.22.3 Node Selector

Simple way to constrain Pods to nodes with specific labels:

```yaml
apiVersion: v1
kind: Pod
metadata:
  name: nginx
spec:
  nodeSelector:
    disktype: ssd
  containers:
  - name: nginx
    image: nginx
```

1.22.4 Node Affinity

More expressive way to specify node constraints:

```yaml
apiVersion: v1
kind: Pod
metadata:
  name: nginx
spec:
```

```yaml
  affinity:
    nodeAffinity:
      requiredDuringSchedulingIgnoredDuringExecution:
        nodeSelectorTerms:
        - matchExpressions:
          - key: topology.kubernetes.io/zone
            operator: In
            values:
            - us-west-1a
            - us-west-1b
  containers:
  - name: nginx
    image: nginx
```

1.22.5 Taints and Tolerations

Taints allow nodes to repel certain Pods:

```
# Add taint to node
kubectl taint nodes node1 key=value:NoSchedule
```

Tolerations allow Pods to schedule on tainted nodes:

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

1.23 Resource Management

1.23.1 Resource Requests and Limits

```yaml
apiVersion: v1
kind: Pod
metadata:
  name: resource-demo
spec:
  containers:
  - name: app
    image: nginx
    resources:
      requests:
```

```
        memory: "64Mi"
        cpu: "250m"
      limits:
        memory: "128Mi"
        cpu: "500m"
```

- **Requests**: Minimum resources guaranteed
- **Limits**: Maximum resources allowed

1.23.2 Quality of Service (QoS) Classes

QoS Class	Condition
Guaranteed	Requests = Limits for all containers
Burstable	At least one container has requests < limits
BestEffort	No requests or limits specified

1.23.3 LimitRange

Sets default resource constraints for a namespace:

```
apiVersion: v1
kind: LimitRange
metadata:
  name: default-limits
spec:
  limits:
  - default:
      cpu: "500m"
      memory: "256Mi"
    defaultRequest:
      cpu: "100m"
      memory: "128Mi"
    type: Container
```

1.23.4 ResourceQuota

Limits total resource consumption in a namespace:

```
apiVersion: v1
kind: ResourceQuota
metadata:
  name: compute-quota
spec:
  hard:
    requests.cpu: "4"
    requests.memory: "8Gi"
    limits.cpu: "8"
    limits.memory: "16Gi"
    pods: "10"
```

1.24 Scaling

1.24.1 Horizontal Pod Autoscaler (HPA)

Automatically scales the number of Pods based on metrics:

```yaml
apiVersion: autoscaling/v2
kind: HorizontalPodAutoscaler
metadata:
  name: nginx-hpa
spec:
  scaleTargetRef:
    apiVersion: apps/v1
    kind: Deployment
    name: nginx
  minReplicas: 1
  maxReplicas: 10
  metrics:
  - type: Resource
    resource:
      name: cpu
      target:
        type: Utilization
        averageUtilization: 50
```

1.24.2 Vertical Pod Autoscaler (VPA)

Automatically adjusts resource requests and limits.

1.24.3 Cluster Autoscaler

Automatically adjusts the size of the Kubernetes cluster.

1.25 Key Concepts to Remember

1. **Containers share the host kernel** - Unlike VMs
2. **Images are immutable** - Changes create new layers
3. **Scheduler uses filtering and scoring** - To find the best node
4. **Resource requests affect scheduling** - Limits affect runtime
5. **QoS determines eviction priority** - BestEffort evicted first

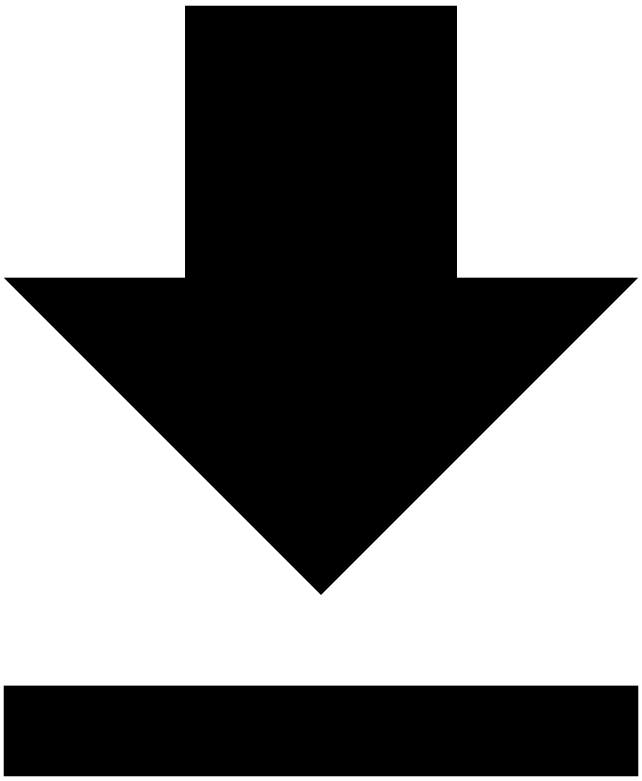
1.26 Practice Questions

1. What is the difference between a container and a virtual machine?
2. What is the role of the kube-scheduler?
3. How do taints and tolerations work together?
4. What is the difference between resource requests and limits?

5. What are the three QoS classes in Kubernetes?

← Previous: Kubernetes Fundamentals | Back to KCNA Overview | Next: Cloud Native Architecture →

1.27 Cloud Native Architecture



Download PDF Version

This domain covers cloud native principles, design patterns, and the CNCF ecosystem.

1.28 What is Cloud Native?

Cloud native technologies empower organizations to build and run scalable applications in modern, dynamic environments such as public, private, and hybrid clouds.

1.28.1 CNCF Definition

> "Cloud native technologies empower organizations to build and run scalable applications in modern, dynamic environments such as public, private, and hybrid clouds. Containers, service meshes, microservices, immutable infrastructure, and declarative APIs exemplify this approach."

1.29 Cloud Native Principles

1.29.1 The Twelve-Factor App

Factor	Description
1. Codebase	One codebase tracked in revision control
2. Dependencies	Explicitly declare and isolate dependencies
3. Config	Store config in the environment
4. Backing Services	Treat backing services as attached resources
5. Build, Release, Run	Strictly separate build and run stages
6. Processes	Execute the app as stateless processes
7. Port Binding	Export services via port binding
8. Concurrency	Scale out via the process model
9. Disposability	Maximize robustness with fast startup and graceful shutdown
10. Dev/Prod Parity	Keep development, staging, and production as similar as possible
11. Logs	Treat logs as event streams
12. Admin Processes	Run admin/management tasks as one-off processes

1.30 Microservices Architecture

1.30.1 Monolith vs Microservices

Aspect	Monolith	Microservices
Deployment	Single unit	Independent services
Scaling	Scale entire app	Scale individual services
Technology	Single stack	Polyglot
Team Structure	Large teams	Small, autonomous teams
Failure Impact	Entire app affected	Isolated failures

1.30.2 Microservices Characteristics

- **Single Responsibility**: Each service does one thing well

- **Independently Deployable**: Services can be deployed without affecting others
- **Decentralized Data**: Each service manages its own data
- **Smart Endpoints, Dumb Pipes**: Logic in services, simple communication
- **Design for Failure**: Services handle failures gracefully

1.31 CNCF Landscape

1.31.1 Project Maturity Levels

Level	Description	Examples
Graduated	Production-ready, widely adopted	Kubernetes, Prometheus, Envoy
Incubating	Growing adoption, maturing	Argo, Cilium, Flux
Sandbox	Early stage, experimental	New projects

1.31.2 Key CNCF Projects by Category

1.31.2.1 Container Runtime

- **containerd** - Industry-standard container runtime
- **CRI-O** - Lightweight runtime for Kubernetes

1.31.2.2 Orchestration

- **Kubernetes** - Container orchestration platform

1.31.2.3 Service Mesh

- **Istio** - Connect, secure, control, and observe services
- **Linkerd** - Ultralight service mesh
- **Envoy** - Edge and service proxy

1.31.2.4 Observability

- **Prometheus** - Monitoring and alerting
- **Grafana** - Visualization and dashboards
- **Jaeger** - Distributed tracing
- **OpenTelemetry** - Observability framework

1.31.2.5 CI/CD

- **Argo** - GitOps continuous delivery
- **Flux** - GitOps toolkit
- **Tekton** - Cloud-native CI/CD

1.31.2.6 Networking

- **Cilium** - eBPF-based networking
- **Calico** - Network policy engine
- **CoreDNS** - DNS server

1.31.2.7 Storage

- **Rook** - Storage orchestration
- **Longhorn** - Distributed block storage

1.31.2.8 Security

- **Falco** - Runtime security
- **OPA** - Policy engine
- **cert-manager** - Certificate management

1.32 Serverless

1.32.1 What is Serverless?

Serverless computing allows you to build and run applications without managing servers. The cloud provider automatically provisions, scales, and manages the infrastructure.

1.32.2 Serverless Platforms

- **AWS Lambda**
- **Google Cloud Functions**
- **Azure Functions**
- **Knative** (Kubernetes-native)

1.32.3 Function as a Service (FaaS)

```
Event → Function → Response

Examples:
- HTTP request triggers function
- Message queue triggers function
- Scheduled event triggers function
```

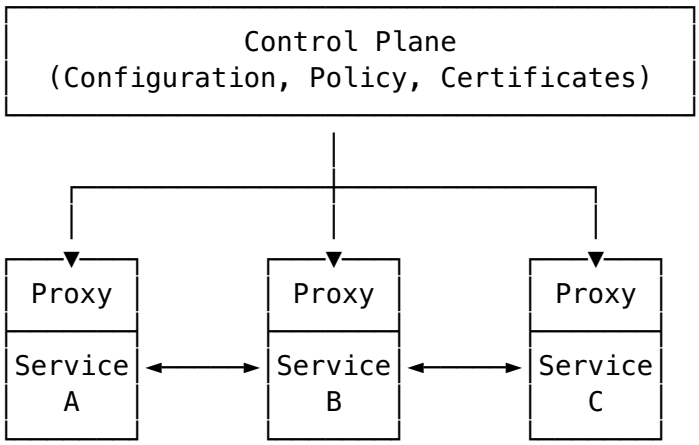
1.33 Service Mesh

1.33.1 What is a Service Mesh?

A dedicated infrastructure layer for handling service-to-service communication, providing:

- **Traffic Management**: Load balancing, routing
- **Security**: mTLS, authentication
- **Observability**: Metrics, tracing, logging

1.33.2 Service Mesh Architecture



1.33.3 Sidecar Pattern

A sidecar container runs alongside the main application container:

```yaml
apiVersion: v1
kind: Pod
metadata:
  name: app-with-sidecar
spec:
  containers:
  - name: app
    image: my-app:v1
  - name: sidecar-proxy
    image: envoy:v1.20
```

1.34 Cloud Native Design Patterns

1.34.1 Circuit Breaker

Prevents cascading failures by stopping requests to failing services.

1.34.2 Retry with Backoff

Automatically retries failed requests with increasing delays.

1.34.3 Bulkhead

Isolates components to prevent failures from spreading.

1.34.4 Sidecar

Deploys helper components alongside the main application.

1.34.5 Ambassador

Creates helper services that send network requests on behalf of a consumer.

1.35 Key Concepts to Remember

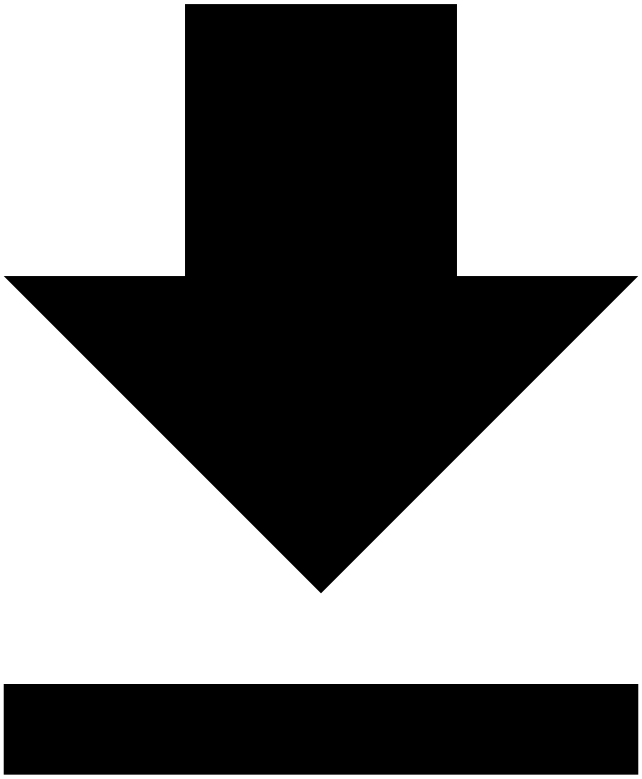
1. **Cloud native is about how applications are built**, not where they run
2. **Microservices enable independent deployment** and scaling
3. **CNCF projects have maturity levels**: Graduated, Incubating, Sandbox
4. **Service meshes handle cross-cutting concerns** like security and observability
5. **Serverless abstracts infrastructure** management completely

1.36 Practice Questions

1. What are the key characteristics of cloud native applications?
2. Name three graduated CNCF projects.
3. What is the difference between a monolith and microservices?
4. What does a service mesh provide?
5. What is the sidecar pattern?

← Previous: Container Orchestration | Back to KCNA Overview | Next: Cloud Native Observability →

1.37 Cloud Native Observability



Download PDF Version

This domain covers monitoring, logging, and tracing in cloud native environments.

1.38 Three Pillars of Observability

1.38.1 1. Metrics

Numeric measurements collected over time.

Examples: - CPU utilization - Memory usage - Request count - Error rate - Response latency

1.38.2 2. Logs

Timestamped records of discrete events.

Examples: - Application errors - Access logs - Audit logs - System events

1.38.3 3. Traces

Records of requests as they flow through distributed systems.

Examples: - Request path through microservices - Latency at each service - Error propagation

1.39 Prometheus

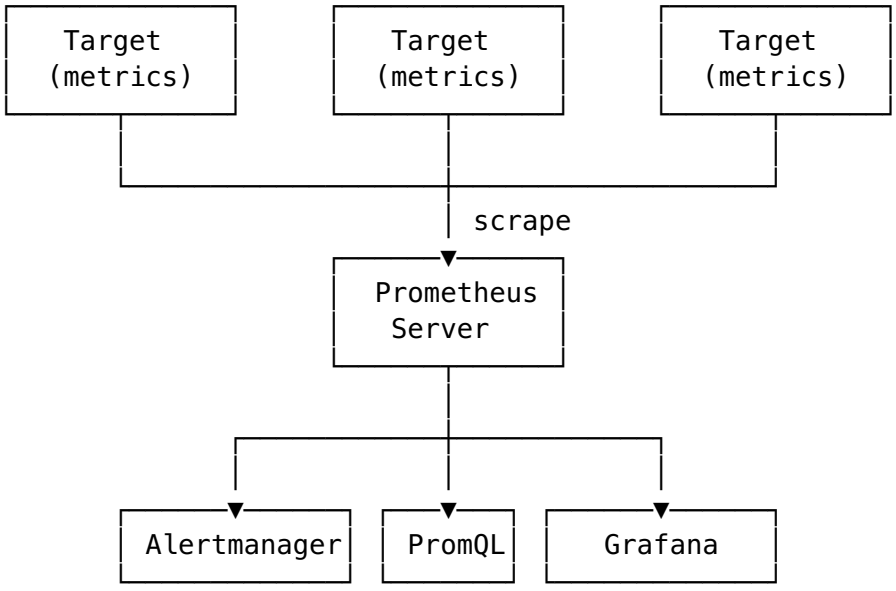
1.39.1 What is Prometheus?

Prometheus is an open-source monitoring and alerting toolkit, graduated from CNCF.

1.39.2 Key Features

- Multi-dimensional data model with time series
- PromQL query language
- Pull-based metrics collection
- Service discovery
- Alerting via Alertmanager

1.39.3 Architecture



1.39.4 Metric Types

Type	Description	Example
Counter	Cumulative, only increases	Total requests

Type	Description	Example
Gauge	Can go up or down	Current temperature
Histogram	Samples in buckets	Request duration
Summary	Similar to histogram with quantiles	Request duration

1.39.5 PromQL Examples

```
# CPU usage
rate(container_cpu_usage_seconds_total[5m])

# Memory usage
container_memory_usage_bytes

# HTTP request rate
rate(http_requests_total[5m])

# Error rate
rate(http_requests_total{status=~"5.."}[5m]) /
rate(http_requests_total[5m])

# 95th percentile latency
histogram_quantile(0.95,
rate(http_request_duration_seconds_bucket[5m]))
```

1.40 Grafana

1.40.1 What is Grafana?

Grafana is an open-source visualization and analytics platform.

1.40.2 Key Features

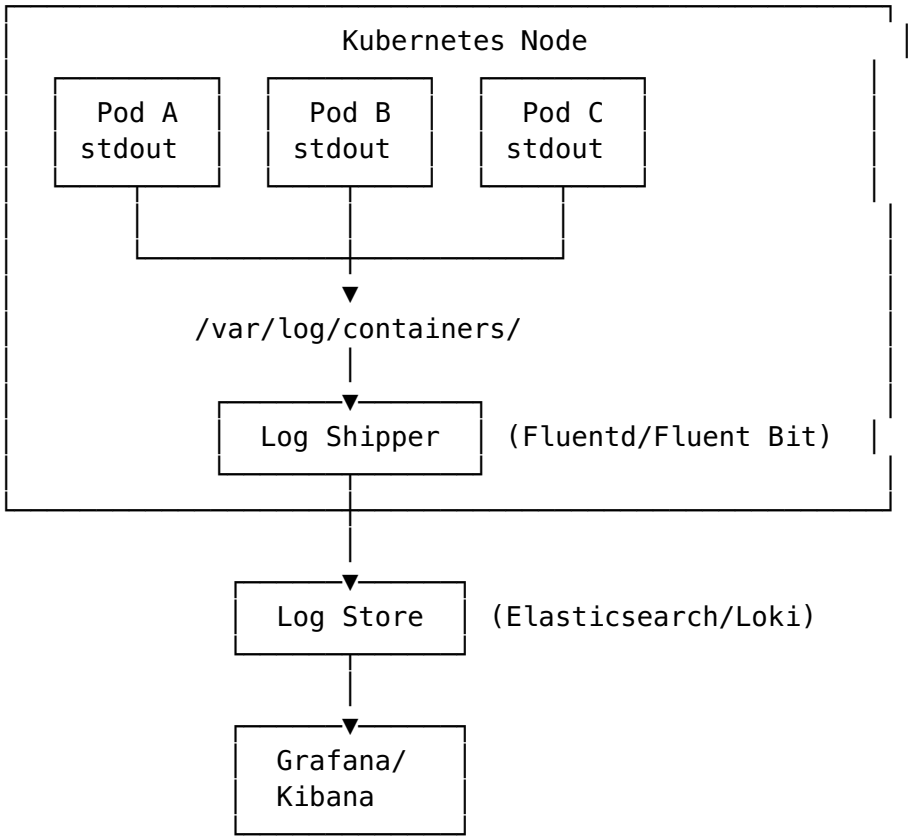
- Dashboard creation and sharing
- Multiple data source support
- Alerting capabilities
- Annotations
- Templating

1.40.3 Common Panels

- **Graph**: Time series visualization
- **Stat**: Single value display
- **Gauge**: Visual gauge
- **Table**: Tabular data
- **Heatmap**: Distribution over time

1.41 Logging

1.41.1 Logging Architecture in Kubernetes



1.41.2 Logging Tools

Tool	Description
Fluentd	Open-source data collector (CNCF Graduated)
Fluent Bit	Lightweight log processor
Elasticsearch	Search and analytics engine
Loki	Log aggregation system by Grafana
Kibana	Visualization for Elasticsearch

1.41.3 Kubernetes Logging Commands

```
# View pod logs
kubectl logs <pod-name>

# Follow logs
kubectl logs -f <pod-name>

# Logs from specific container
kubectl logs <pod-name> -c <container-name>
```

```
# Previous container logs
kubectl logs <pod-name> --previous

# Logs with timestamps
kubectl logs <pod-name> --timestamps
```

1.42 Distributed Tracing

1.42.1 What is Distributed Tracing?

Distributed tracing tracks requests as they flow through multiple services, helping identify:

- Performance bottlenecks
- Error sources
- Service dependencies

1.42.2 Tracing Concepts

Concept	Description
Trace	End-to-end journey of a request
Span	Single operation within a trace
Context	Metadata propagated between services

1.42.3 Trace Example

```
Trace ID: abc123
├── Span: API Gateway (10ms)
│   └── Span: Auth Service (5ms)
├── Span: Order Service (50ms)
│   ├── Span: Database Query (20ms)
│   └── Span: Payment Service (25ms)
└── Span: Notification Service (15ms)
```

1.42.4 Tracing Tools

Tool	Description
Jaeger	Distributed tracing platform (CNCF Graduated)
Zipkin	Distributed tracing system
OpenTelemetry	Observability framework (CNCF)

1.43 OpenTelemetry

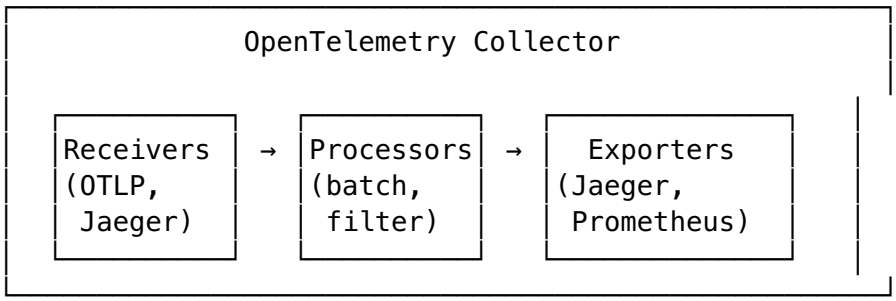
1.43.1 What is OpenTelemetry?

OpenTelemetry is a collection of tools, APIs, and SDKs for instrumenting, generating, collecting, and exporting telemetry data (metrics, logs, traces).

1.43.2 Components

- **API**: Defines how to generate telemetry
- **SDK**: Implements the API
- **Collector**: Receives, processes, and exports data
- **Exporters**: Send data to backends

1.43.3 OpenTelemetry Collector



1.44 Cost Management

1.44.1 Observability Costs

- **Storage**: Metrics, logs, and traces consume storage
- **Compute**: Processing and querying data
- **Network**: Data transfer between components

1.44.2 Cost Optimization

- Set appropriate retention periods
- Use sampling for high-volume traces
- Aggregate metrics where possible
- Filter unnecessary logs

1.45 Key Concepts to Remember

1. **Three pillars**: Metrics, Logs, Traces
2. **Prometheus uses pull-based** metrics collection
3. **PromQL** is the query language for Prometheus
4. **OpenTelemetry** unifies observability instrumentation

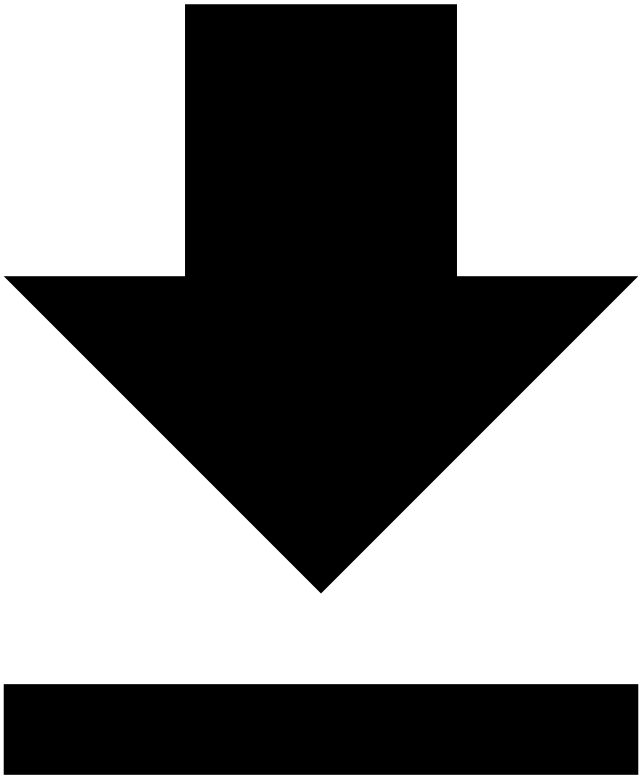
5. **Jaeger and Zipkin** are popular tracing tools

1.46 Practice Questions

1. What are the three pillars of observability?
2. What is the difference between a Counter and a Gauge in Prometheus?
3. What is the purpose of distributed tracing?
4. Name two CNCF graduated observability projects.
5. What does OpenTelemetry provide?

← Previous: Cloud Native Architecture | Back to KCNA Overview | Next: Cloud Native Application Delivery →

1.47 Cloud Native Application Delivery



Download PDF Version

This domain covers CI/CD, GitOps, and application deployment strategies in cloud native environments.

1.48 CI/CD Fundamentals

1.48.1 Continuous Integration (CI)

Automatically building and testing code changes.

Key Practices:

- Frequent code commits
- Automated builds
- Automated testing
- Fast feedback loops

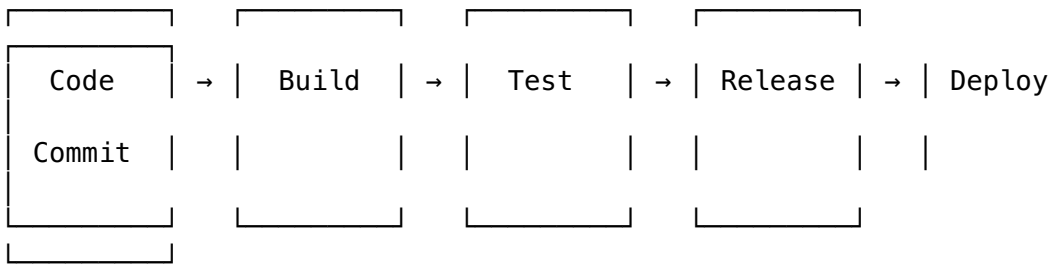
1.48.2 Continuous Delivery (CD)

Automatically deploying code changes to staging/production.

Key Practices:

- Automated deployments
- Environment parity
- Rollback capabilities
- Release automation

1.48.3 CI/CD Pipeline Stages



1.49 GitOps

1.49.1 What is GitOps?

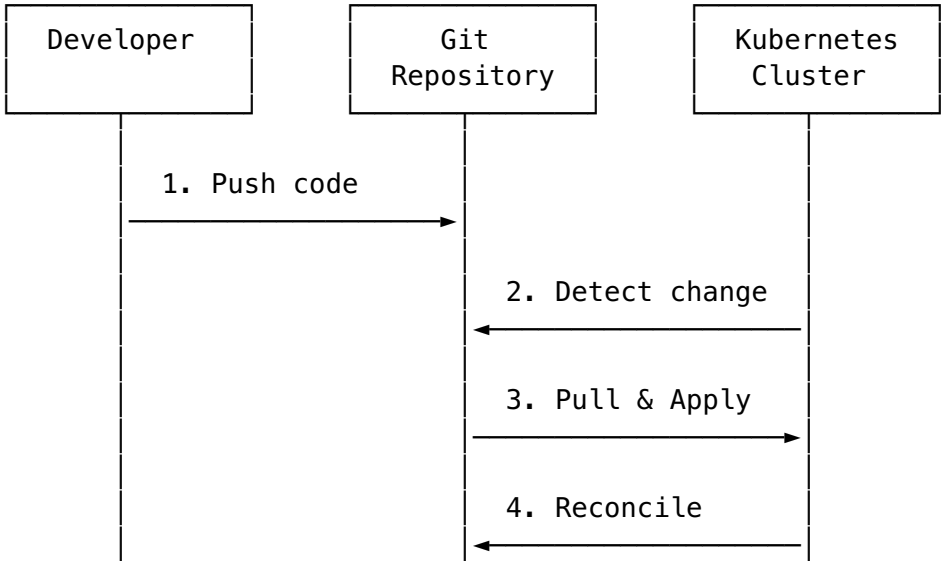
GitOps is a way of implementing Continuous Deployment for cloud native applications using Git as the single source of truth.

1.49.2 GitOps Principles

1. **Declarative**: System state is described declaratively

2. **Versioned**: Desired state is stored in Git
3. **Automated**: Changes are automatically applied
4. **Reconciled**: Software agents ensure actual state matches desired state

1.49.3 GitOps Workflow



1.49.4 GitOps Tools

Tool	Description
Argo CD	Declarative GitOps CD for Kubernetes (CNCF)
Flux	GitOps toolkit for Kubernetes (CNCF)
Jenkins X	CI/CD for Kubernetes

1.50 Argo CD

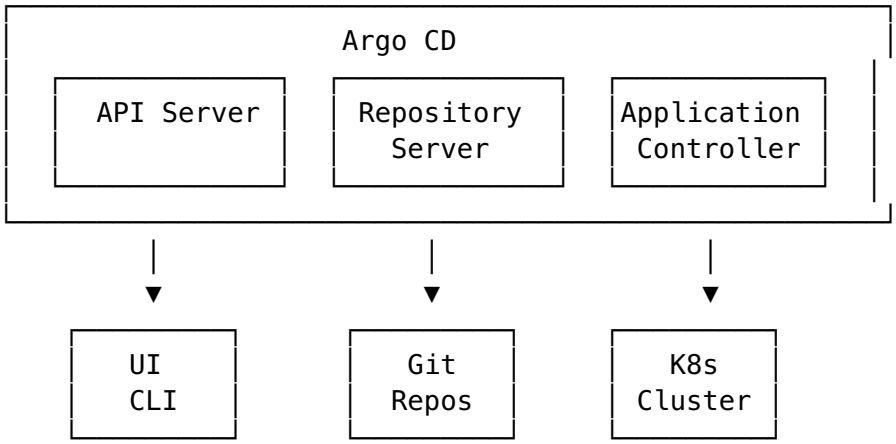
1.50.1 What is Argo CD?

Argo CD is a declarative, GitOps continuous delivery tool for Kubernetes.

1.50.2 Key Features

- Automated deployment of applications
- Support for multiple config management tools (Kustomize, Helm, Jsonnet)
- SSO integration
- Rollback/roll-anywhere
- Health status analysis
- Web UI and CLI

1.50.3 Argo CD Architecture



1.50.4 Argo CD Application

```yaml
apiVersion: argoproj.io/v1alpha1
kind: Application
metadata:
  name: my-app
  namespace: argocd
spec:
  project: default
  source:
    repoURL: https://github.com/org/repo.git
    targetRevision: HEAD
    path: manifests
  destination:
    server: https://kubernetes.default.svc
    namespace: my-app
  syncPolicy:
    automated:
      prune: true
      selfHeal: true
```

1.51 Flux

1.51.1 What is Flux?

Flux is a set of continuous and progressive delivery solutions for Kubernetes.

1.51.2 Flux Components

Component	Description
Source Controller	Manages sources (Git, Helm, OCI)
Kustomize Controller	Reconciles Kustomize resources

Component	Description
Helm Controller	Manages Helm releases
Notification Controller	Handles events and alerts
Image Automation	Updates container images

1.52 Helm

1.52.1 What is Helm?

Helm is the package manager for Kubernetes.

1.52.2 Key Concepts

Concept	Description
Chart	Package of pre-configured Kubernetes resources
Release	Instance of a chart running in a cluster
Repository	Collection of charts
Values	Configuration for a chart

1.52.3 Helm Commands

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

# Search charts
helm search repo nginx

# Install chart
helm install my-release bitnami/nginx

# Upgrade release
helm upgrade my-release bitnami/nginx

# Rollback
helm rollback my-release 1

# Uninstall
helm uninstall my-release

# List releases
helm list
```

1.52.4 Helm Chart Structure

```
my-chart/
├── Chart.yaml          # Chart metadata
```

```
├── values.yaml         # Default configuration
├── templates/          # Kubernetes manifests
│   ├── deployment.yaml
│   ├── service.yaml
│   └── _helpers.tpl
└── charts/             # Dependencies
```

1.53 Deployment Strategies

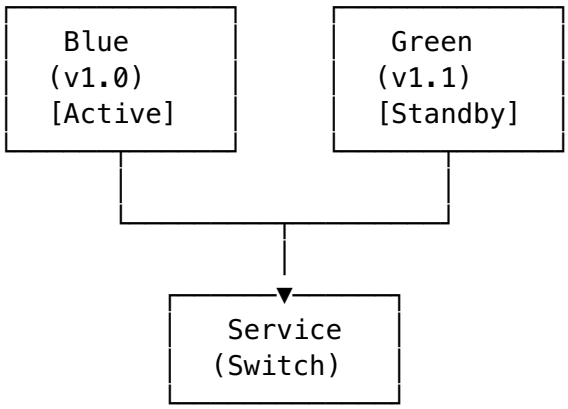
1.53.1 Rolling Update

Gradually replaces old pods with new ones.

```yaml
spec:
  strategy:
    type: RollingUpdate
    rollingUpdate:
      maxSurge: 25%
      maxUnavailable: 25%
```

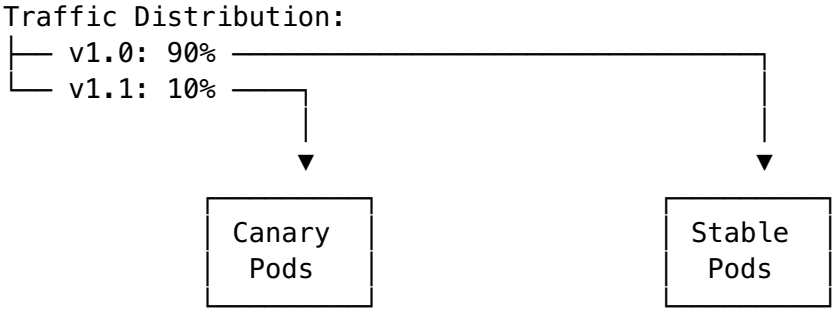
1.53.2 Blue-Green Deployment

Two identical environments, switch traffic between them.



1.53.3 Canary Deployment

Gradually shift traffic to new version.



1.53.4 A/B Testing

Route traffic based on specific criteria (headers, cookies).

1.54 Application Configuration

1.54.1 Kustomize

Kubernetes native configuration management.

```
base/
├── deployment.yaml
├── service.yaml
└── kustomization.yaml

overlays/
├── dev/
│   └── kustomization.yaml
├── staging/
│   └── kustomization.yaml
└── prod/
    └── kustomization.yaml
```

```yaml
# kustomization.yaml
apiVersion: kustomize.config.k8s.io/v1beta1
kind: Kustomization
resources:
  - deployment.yaml
  - service.yaml
namePrefix: dev-
namespace: development
```

1.55 Key Concepts to Remember

1. **GitOps uses Git as single source of truth** for infrastructure
2. **Argo CD and Flux** are popular GitOps tools
3. **Helm** is the package manager for Kubernetes
4. **Rolling updates** are the default deployment strategy
5. **Canary deployments** allow gradual traffic shifting

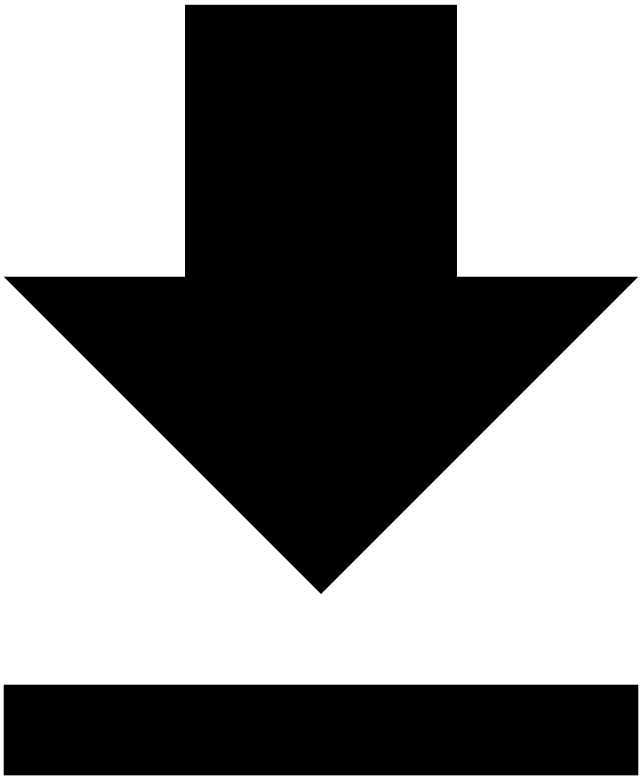
1.56 Practice Questions

1. What is GitOps and what are its core principles?
2. What is the difference between Argo CD and Flux?
3. What is a Helm chart?
4. Describe the difference between blue-green and canary deployments.
5. What is Kustomize used for?

← Previous: Cloud Native Observability | Back to KCNA Overview

---

1.57 Sample Practice Questions



Download PDF Version

> **Disclaimer**: These are sample practice questions created for study purposes only. They are NOT actual exam questions and are designed to help you test your understanding of KCNA concepts. Real exam questions may differ in format and content.

1.58 Instructions

- Each question has one correct answer unless otherwise specified
- Try to answer without looking at the solutions first
- Review the explanations to understand the concepts better

1.59 Section 1: Kubernetes Fundamentals (46%)

1.59.1 Question 1.1

What is the smallest deployable unit in Kubernetes?

1. Container
2. Pod
3. Deployment
4. ReplicaSet

Show Answer

Answer: B) Pod

A Pod is the smallest deployable unit in Kubernetes. It can contain one or more containers that share storage and network resources.

1.59.2 Question 1.2

Which control plane component is responsible for storing all cluster data?

1. kube-apiserver
2. kube-scheduler
3. etcd
4. kube-controller-manager

Show Answer

Answer: C) etcd

etcd is a consistent and highly-available key-value store used as Kubernetes' backing store for all cluster data.

1.59.3 Question 1.3

What is the default Service type in Kubernetes?

1. NodePort
2. LoadBalancer
3. ClusterIP
4. ExternalName

Show Answer

Answer: C) ClusterIP

ClusterIP is the default Service type. It exposes the Service on a cluster-internal IP, making it only reachable from within the cluster.

1.59.4 Question 1.4

Which component runs on every node and ensures containers are running in a Pod?

1. kube-proxy
2. kubelet
3. kube-scheduler
4. container runtime

Show Answer

Answer: B) kubelet

The kubelet is an agent that runs on each node in the cluster. It ensures that containers are running in a Pod as specified.

1.59.5 Question 1.5

What Kubernetes object would you use to store non-confidential configuration data?

1. Secret
2. ConfigMap
3. PersistentVolume
4. ServiceAccount

Show Answer

Answer: B) ConfigMap

ConfigMaps are used to store non-confidential data in key-value pairs. Secrets are used for sensitive data.

1.59.6 Question 1.6

Which namespace contains Kubernetes system components?

1. default
2. kube-public
3. kube-system
4. kube-node-lease

Show Answer

Answer: C) kube-system

The kube-system namespace contains objects created by the Kubernetes system, such as kube-dns, kube-proxy, and other system components.

1.59.7 Question 1.7

What does a ReplicaSet ensure?

1. Pods are scheduled on specific nodes

2. A specified number of pod replicas are running at any given time
3. Pods have persistent storage
4. Pods can communicate with external services

Show Answer

Answer: B) A specified number of pod replicas are running at any given time

A ReplicaSet's purpose is to maintain a stable set of replica Pods running at any given time.

1.59.8 Question 1.8

Which API version is used for Deployments?

1. v1
2. apps/v1
3. batch/v1
4. networking.k8s.io/v1

Show Answer

Answer: B) apps/v1

Deployments use the apps/v1 API group. The core v1 API is used for Pods, Services, ConfigMaps, etc.

1.60 Section 2: Container Orchestration (22%)

1.60.1 Question 2.1

What is the main difference between a container and a virtual machine?

1. Containers are larger than VMs
2. Containers share the host OS kernel
3. VMs are faster to start
4. Containers provide hardware-level isolation

Show Answer

Answer: B) Containers share the host OS kernel

Containers share the host operating system's kernel, making them lightweight and fast to start. VMs include a full OS and provide hardware-level isolation.

1.60.2 Question 2.2

Which Kubernetes object ensures that all (or some) nodes run a copy of a Pod?

1. Deployment
2. StatefulSet

3. DaemonSet
4. ReplicaSet

Show Answer

Answer: C) DaemonSet

A DaemonSet ensures that all (or some) nodes run a copy of a Pod. Common use cases include log collectors and monitoring agents.

1.60.3 Question 2.3

What is the purpose of resource requests in Kubernetes?

1. To limit the maximum resources a container can use
2. To guarantee minimum resources for scheduling
3. To autoscale pods based on usage
4. To define storage requirements

Show Answer

Answer: B) To guarantee minimum resources for scheduling

Resource requests specify the minimum amount of resources a container needs. The scheduler uses this to decide which node to place the Pod on.

1.60.4 Question 2.4

Which QoS class is assigned to a Pod when requests equal limits for all containers?

1. BestEffort
2. Burstable
3. Guaranteed
4. Standard

Show Answer

Answer: C) Guaranteed

A Pod is classified as Guaranteed when every container has memory and CPU limits that equal their requests.

1.60.5 Question 2.5

What mechanism allows Pods to be scheduled on nodes with specific taints?

1. Node selectors
2. Node affinity
3. Tolerations
4. Pod priority

Show Answer

Answer: C) Tolerations

Tolerations are applied to Pods and allow (but do not require) the Pods to schedule onto nodes with matching taints.

1.61 Section 3: Cloud Native Architecture (16%)

1.61.1 Question 3.1

Which of the following is a CNCF graduated project?

1. Argo
2. Flux
3. Prometheus
4. Cilium

Show Answer

Answer: C) Prometheus

Prometheus is a CNCF graduated project. Graduated projects are considered stable and production-ready with wide adoption.

1.61.2 Question 3.2

What does the “12-factor app” methodology primarily address?

1. Security best practices
2. Building software-as-a-service applications
3. Container image optimization
4. Network configuration

Show Answer

Answer: B) Building software-as-a-service applications

The 12-factor app methodology provides best practices for building modern, scalable, maintainable software-as-a-service applications.

1.61.3 Question 3.3

What is a key characteristic of microservices architecture?

1. Single deployment unit
2. Shared database for all services
3. Services can be deployed independently
4. Monolithic codebase

Show Answer

Answer: C) Services can be deployed independently

Microservices are independently deployable services that communicate over well-defined APIs. Each service can be developed, deployed, and scaled independently.

1.61.4 Question 3.4

What does a service mesh primarily provide?

1. Container runtime
2. Service-to-service communication management
3. Persistent storage
4. CI/CD pipelines

Show Answer

Answer: B) Service-to-service communication management

A service mesh handles service-to-service communication, providing features like traffic management, security (mTLS), and observability.

1.61.5 Question 3.5

What is the sidecar pattern in Kubernetes?

1. Running multiple replicas of the same container
2. Running a helper container alongside the main application container
3. Running containers on dedicated nodes
4. Running containers with elevated privileges

Show Answer

Answer: B) Running a helper container alongside the main application container

The sidecar pattern involves running a helper container in the same Pod as the main application to provide supporting functionality like logging, proxying, or configuration.

1.62 Section 4: Cloud Native Observability (8%)

1.62.1 Question 4.1

What are the three pillars of observability?

1. CPU, Memory, Disk
2. Logs, Metrics, Traces
3. Pods, Services, Deployments
4. Authentication, Authorization, Admission

Show Answer

Answer: B) Logs, Metrics, Traces

The three pillars of observability are Logs (event records), Metrics (numeric measurements), and Traces (request flow through systems).

1.62.2 Question 4.2

Which Prometheus metric type can only increase?

1. Gauge
2. Counter
3. Histogram
4. Summary

Show Answer

Answer: B) Counter

A Counter is a cumulative metric that can only increase (or reset to zero on restart). It's used for values like total requests or errors.

1.62.3 Question 4.3

What is the primary purpose of distributed tracing?

1. Storing application logs
2. Tracking requests across multiple services
3. Monitoring CPU usage
4. Managing secrets

Show Answer

Answer: B) Tracking requests across multiple services

Distributed tracing tracks requests as they flow through multiple services, helping identify performance bottlenecks and errors.

1.62.4 Question 4.4

Which CNCF project provides a unified observability framework for metrics, logs, and traces?

1. Prometheus
2. Jaeger
3. OpenTelemetry
4. Fluentd

Show Answer

Answer: C) OpenTelemetry

OpenTelemetry provides a single set of APIs, libraries, and agents to collect metrics, logs, and traces from applications.

1.63 Section 5: Cloud Native Application Delivery (8%)

1.63.1 Question 5.1

What is GitOps?

1. Using Git for version control
2. Using Git as the single source of truth for declarative infrastructure
3. A Git hosting service
4. A CI/CD tool

Show Answer

Answer: B) Using Git as the single source of truth for declarative infrastructure

GitOps uses Git repositories as the source of truth for defining the desired state of infrastructure and applications, with automated synchronization.

1.63.2 Question 5.2

What is a Helm chart?

1. A monitoring dashboard
2. A package of pre-configured Kubernetes resources
3. A network diagram
4. A security policy

Show Answer

Answer: B) A package of pre-configured Kubernetes resources

A Helm chart is a collection of files that describe a related set of Kubernetes resources, allowing for easy deployment and management.

1.63.3 Question 5.3

Which deployment strategy gradually shifts traffic from the old version to the new version?

1. Rolling update
2. Blue-green deployment
3. Canary deployment
4. Recreate deployment

Show Answer

Answer: C) Canary deployment

Canary deployment gradually shifts traffic to the new version, allowing you to test with a small percentage of users before full rollout.

1.63.4 Question 5.4

What is the default deployment strategy in Kubernetes?

1. Recreate
2. Blue-green
3. Canary
4. Rolling update

Show Answer

Answer: D) Rolling update

Rolling update is the default deployment strategy in Kubernetes. It gradually replaces old Pods with new ones, ensuring zero downtime.

1.63.5 Question 5.5

Which tool is a CNCF project for GitOps continuous delivery?

1. Jenkins
2. Argo CD
3. GitHub Actions
4. CircleCI

Show Answer

Answer: B) Argo CD

Argo CD is a CNCF project that provides declarative, GitOps continuous delivery for Kubernetes.

1.64 Scenario-Based Questions

1.64.1 Scenario 1

You need to deploy an application that requires exactly one instance running on every node in your cluster for log collection.

Question: Which Kubernetes object should you use?

1. Deployment with node affinity
2. StatefulSet
3. DaemonSet
4. ReplicaSet with pod anti-affinity

Show Answer

Answer: C) DaemonSet

DaemonSet ensures that a copy of a Pod runs on all (or selected) nodes. This is ideal for node-level services like log collectors, monitoring agents, or network plugins.

1.64.2 Scenario 2

Your application needs to maintain stable network identities and persistent storage across Pod restarts.

Question: Which Kubernetes object is most appropriate?

1. Deployment
2. StatefulSet
3. DaemonSet
4. Job

Show Answer

Answer: B) StatefulSet

StatefulSet is designed for stateful applications that require stable network identities, persistent storage, and ordered deployment/scaling.

1.64.3 Scenario 3

You want to expose your application to external traffic and need the cloud provider to provision a load balancer.

Question: Which Service type should you use?

1. ClusterIP
2. NodePort
3. LoadBalancer
4. ExternalName

Show Answer

Answer: C) LoadBalancer

LoadBalancer Service type exposes the Service externally using a cloud provider's load balancer. It automatically provisions an external load balancer.

1.65 Study Tips

1. **Understand core concepts** - Know the difference between Pods, Deployments, Services, etc.
2. **Practice with kubectl** - Hands-on experience helps solidify concepts
3. **Know the CNCF landscape** - Understand graduated, incubating, and sandbox projects
4. **Review Kubernetes architecture** - Understand control plane and node components
5. **Study the 12-factor app** - Know the principles for cloud native applications

---

← Back to KCNA Overview
