Kubernetes: Container Orchestration Platform

Complete Guide to Container Orchestration and Kubernetes Architecture

Prepared By: Rashi Rana
Corporate Trainer

Why We Need an Orchestration Tool

Container Orchestration is the automated deployment, management, scaling, and networking of containers. As applications grow in complexity and scale, manual container management becomes impractical and error-prone.

Challenges with Manual Container Management

- **Scaling Issues:** Manually starting/stopping containers based on demand
- **Service Discovery:** Containers need to find and communicate with each other
- **Load Balancing:** Distributing traffic across multiple container instances
- **Health Monitoring:** Detecting and replacing failed containers
- **Rolling Updates:** Updating applications without downtime
- **Resource Management:** Efficiently utilizing CPU, memory, and storage
- **Configuration Management:** Managing secrets, configs across environments
- **Network Management:** Container-to-container communication

Benefits of Container Orchestration

Automated Scaling

High Availability

Automatically scale applications up or down based on demand, CPU usage, or custom metrics.

Ensure applications remain available by automatically replacing failed containers and distributing workloads.

Resource Optimization

Efficiently pack containers onto nodes to maximize resource utilization and minimize costs.

Simplified Deployment

Declarative configuration allows you to describe desired state rather than imperative steps.

Real-World Scenarios

- **E-commerce Platform:** Handle traffic spikes during sales events
- **Microservices Architecture:** Manage hundreds of interconnected services
- **CI/CD Pipelines:** Deploy applications across multiple environments
- **Multi-Cloud Deployments:** Run applications across different cloud providers

Prepared By: Rashi Rana
Corporate Trainer

Introduction to Kubernetes

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

What is Kubernetes?

- **Container Orchestrator:** Manages containerized applications across a cluster of machines
- **Declarative Platform:** You describe the desired state, Kubernetes makes it happen
- **Portable:** Runs on-premises, in the cloud, or in hybrid environments
- **Extensible:** Rich ecosystem of tools and extensions
- **Production-Ready:** Battle-tested by Google and thousands of organizations

Key Capabilities

Service Discovery & Load Balancing

Automatically expose containers and distribute

Storage Orchestration

Mount storage systems of your choice - local, cloud

network traffic for stable deployment.

providers, or network storage.

Automated Rollouts & Rollbacks

Deploy changes progressively and rollback if something goes wrong.

Self-Healing

Restart failed containers, replace containers, and kill unresponsive containers.

Secret & Configuration Management

Store and manage sensitive information and configuration separately from container images.

Horizontal Scaling

Scale applications up or down with simple commands or automatically based on CPU usage.

- **CNCF Projects:** Helm, Prometheus, Istio, Envoy, Fluentd
- **Cloud Providers:** EKS (AWS), GKE (Google), AKS (Azure)
- **Distributions:** OpenShift, Rancher, VMware Tanzu
- **Tools:** kubectl, kubeadm, kustomize, Skaffold

Prepared By: Rashi Rana
Corporate Trainer

Why Kubernetes (K8s)?

Business Drivers

Cost Efficiency

Better resource utilization, reduced infrastructure costs, and optimized cloud spending.

Faster Time to Market

Accelerated development cycles, automated deployments, and consistent environments.

Improved Reliability

Self-healing systems, automated failover, and built-in redundancy.

Vendor Independence

Avoid vendor lock-in with portable applications that run anywhere.

Technical Advantages

- **Immutable Infrastructure:** Consistent, reproducible deployments
- **Microservices Support:** Perfect platform for microservices architecture
- **DevOps Integration:** Seamless CI/CD pipeline integration
- **Multi-Cloud Strategy:** Deploy across multiple cloud providers
- **Container-Native:** Built specifically for containerized workloads
- **API-Driven:** Everything is programmable and automatable

Industry Adoption

> **Market Leadership:** Kubernetes has become the de facto standard for container orchestration, with 96% of organizations either using or evaluating Kubernetes (CNCF Survey 2023).

Success Stories

Company	Use Case	Benefits Achieved
Netflix	Global streaming platform	Handles 200M+ users, auto-scaling
Spotify	Music streaming service	1000+ microservices, faster deployments
Airbnb	Travel platform	Reduced deployment time from hours to minutes
Pinterest	Social media platform	Improved resource utilization by 80%

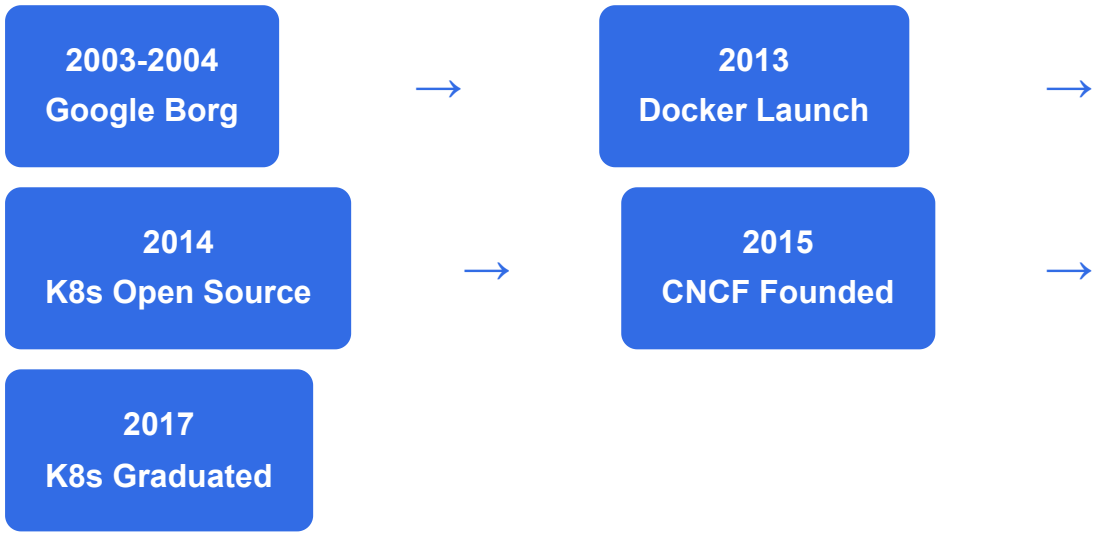
When to Choose Kubernetes

- **Microservices Architecture:** Managing multiple interconnected services
- **High Availability Requirements:** Mission-critical applications
- **Dynamic Scaling Needs:** Variable workloads and traffic patterns
- **Multi-Environment Deployments:** Dev, staging, production consistency
- **Team Collaboration:** Multiple teams working on different services

Prepared By: Rashi Rana
Corporate Trainer

History and Role in DevOps

Kubernetes History



Evolution Timeline

- **2003-2004:** Google develops Borg for internal container orchestration
- **2013:** Docker popularizes containerization technology
- **June 2014:** Google open-sources Kubernetes project
- **July 2015:** Cloud Native Computing Foundation (CNCF) established
- **2016:** Kubernetes 1.0 released, production-ready
- **2017:** Kubernetes becomes first CNCF graduated project
- **2018-Present:** Rapid adoption and ecosystem growth

Role in DevOps Culture

Infrastructure as Code

YAML manifests define infrastructure declaratively, version-controlled and reproducible.

Continuous Integration

Seamless integration with CI tools for automated testing and building.

Continuous Deployment

GitOps workflows enable automated deployments from Git repositories.

Monitoring & Observability

Built-in metrics, logging, and tracing capabilities for comprehensive monitoring.

DevOps Principles Enabled by Kubernetes

- **Collaboration:** Shared platform for Dev and Ops teams
- **Automation:** Self-healing, auto-scaling, and automated deployments
- **Measurement:** Rich metrics and monitoring capabilities
- **Sharing:** Reusable components and standardized practices

> **DevOps Impact:** Kubernetes has transformed how organizations approach application deployment, enabling true DevOps practices with infrastructure automation, continuous delivery, and collaborative workflows.

Prepared By: Rashi Rana

Kubernetes vs. Docker Swarm

Both Kubernetes and Docker Swarm are container orchestration platforms, but they differ significantly in complexity, features, and use cases.

Detailed Comparison

Aspect	Kubernetes	Docker Swarm
Complexity	High learning curve, complex setup	Simple to learn and deploy
Installation	Complex, multiple components	Built into Docker Engine
Scalability	Highly scalable (5000+ nodes)	Limited scalability (1000 nodes)
Load Balancing	Manual configuration required	Automatic load balancing
Auto-scaling	Horizontal Pod Autoscaler	No built-in auto-scaling
Rolling Updates	Advanced deployment strategies	Basic rolling updates
Networking	Complex but flexible	Simple overlay network

Aspect	Kubernetes	Docker Swarm
Storage	Persistent Volumes, Storage Classes	Basic volume support
Monitoring	Rich ecosystem (Prometheus, etc.)	Basic monitoring capabilities
Community	Large, active community	Smaller community

When to Choose Kubernetes

- **Complex Applications:** Microservices with multiple dependencies
- **Enterprise Requirements:** Advanced features, security, compliance
- **Multi-Cloud Strategy:** Vendor-neutral platform
- **Large Scale:** Hundreds or thousands of containers
- **Advanced Networking:** Complex networking requirements

When to Choose Docker Swarm

- **Simple Applications:** Monolithic or simple microservices
- **Quick Setup:** Need to get started quickly
- **Docker-Centric:** Already heavily invested in Docker
- **Small Teams:** Limited DevOps expertise
- **Basic Requirements:** Simple orchestration needs

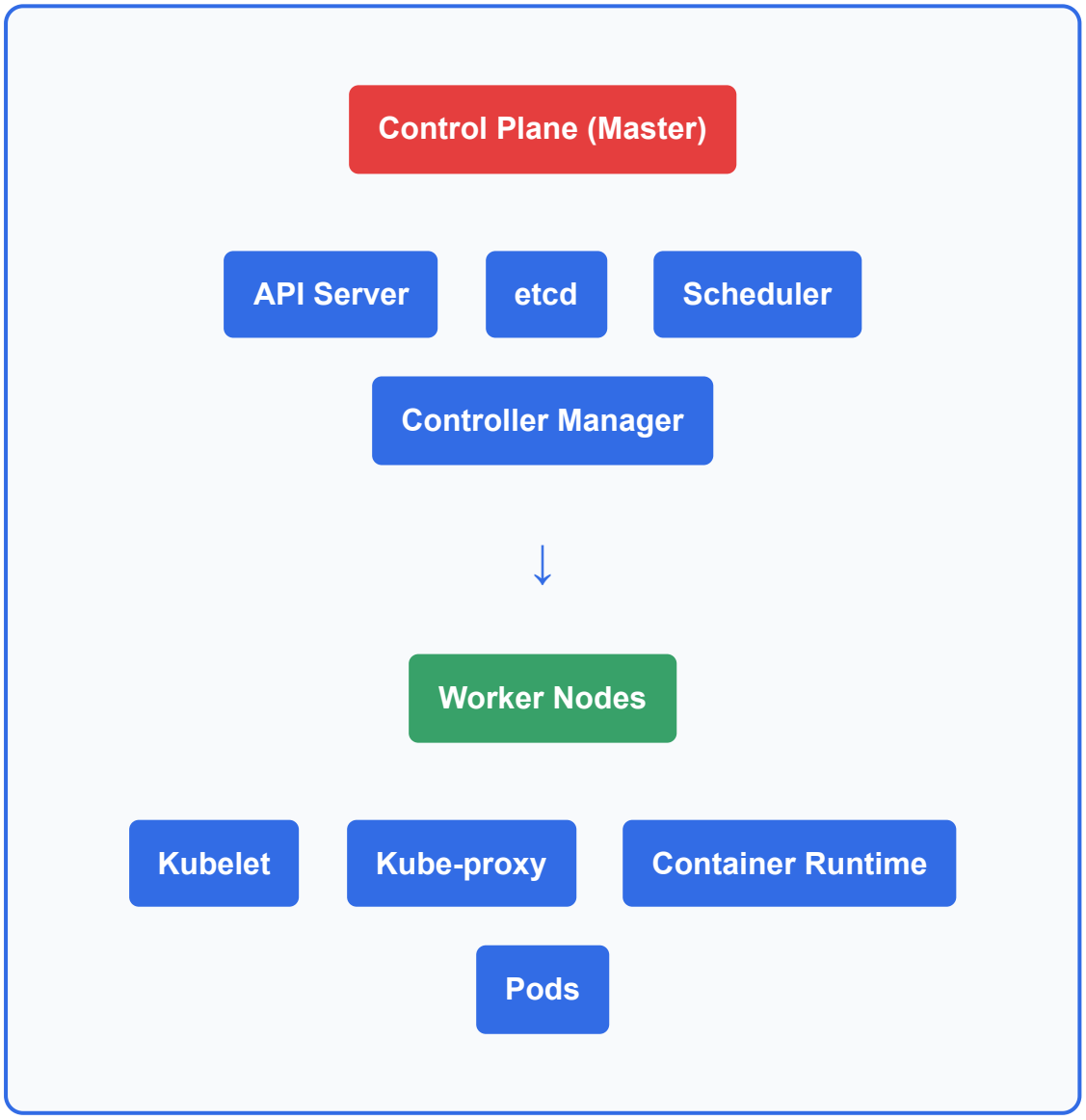
> **Market Reality:** While Docker Swarm is simpler, Kubernetes has become the industry standard due to its flexibility, features, and ecosystem support.

Prepared By: Rashi Rana
Corporate Trainer

Kubernetes Architecture Overview

Kubernetes Architecture follows a master-worker pattern where the control plane manages the cluster state and worker nodes run the actual workloads.

High-Level Architecture



Key Architectural Principles

Declarative Configuration

Describe desired state, Kubernetes ensures current state matches desired state.

API-Driven

All interactions happen through REST APIs, enabling automation and extensibility.

Controller Pattern

Controllers continuously monitor and reconcile actual state with desired state.

Distributed System

Designed for high availability and fault tolerance across multiple nodes.

Communication Flow

1. **User/CI System:** Submits YAML manifests to API Server
2. **API Server:** Validates and stores configuration in etcd
3. **Scheduler:** Assigns pods to appropriate worker nodes
4. **Kubelet:** Pulls container images and starts containers
5. **Controllers:** Monitor and maintain desired state

Cluster Components

- **Control Plane:** Manages cluster state and makes global decisions
- **Worker Nodes:** Run application workloads in containers
- **Add-ons:** DNS, Dashboard, Monitoring, Logging

- **Network:** Pod-to-pod and service-to-service communication

Prepared By: Rashi Rana
Corporate Trainer

Master and Worker Nodes

Control Plane (Master Node)

> The **Control Plane** is the brain of the Kubernetes cluster, responsible for making global decisions about the cluster and detecting and responding to cluster events.

Control Plane Responsibilities

- **Cluster State Management:** Maintain desired state of the cluster
- **API Gateway:** Single point of entry for all cluster operations
- **Scheduling:** Decide which node should run each pod
- **Resource Management:** Monitor and manage cluster resources
- **Policy Enforcement:** Apply security policies and resource quotas

Worker Node Overview

> **Worker Nodes** are the machines where your application containers actually run. Each worker node is managed by the control plane and contains the services necessary to run pods.

Worker Node Responsibilities

- **Pod Execution:** Run and manage application containers
- **Network Proxy:** Handle network routing for services

- **Container Runtime:** Pull images and run containers
- **Resource Monitoring:** Report node and pod status to control plane
- **Volume Management:** Mount and manage storage volumes

Node Types Comparison

Aspect	Control Plane (Master)	Worker Node
Primary Role	Cluster management and control	Run application workloads
Key Components	API Server, etcd, Scheduler, Controllers	Kubelet, Kube-proxy, Container Runtime
Workload Scheduling	Usually no application pods (can be configured)	Runs application pods
High Availability	Multiple masters for HA	Multiple workers for load distribution
Resource Requirements	Lower CPU/Memory for control tasks	Higher CPU/Memory for applications

Cluster Topology Examples

Single Master

1 Master + N Workers
Simple setup, single point of failure

High Availability

3+ Masters + N Workers
Production-ready, fault-tolerant

Development

All-in-one node
Master and worker on same machine

Edge Computing

Lightweight masters
Optimized for resource-constrained environments

Production Recommendation: Always use multiple master nodes (odd number: 3, 5, 7) for high availability in production environments.

Prepared By: Rashi Rana
Corporate Trainer

API Server

> The **API Server** is the central management entity and the only component that directly interacts with etcd. It serves as the front-end for the Kubernetes control plane.

Key Responsibilities

- **API Gateway:** Single entry point for all REST operations
- **Authentication & Authorization:** Verify user identity and permissions
- **Admission Control:** Validate and mutate requests before processing
- **Data Validation:** Ensure resource definitions are correct
- **etcd Interface:** Only component that directly reads/writes to etcd
- **Watch Interface:** Notify clients about resource changes

API Server Features

RESTful API

Standard HTTP methods (GET, POST, PUT, DELETE) for resource operations

Resource Versioning

Multiple API versions (v1, v1beta1, v1alpha1) for backward compatibility

Custom Resources

Extend Kubernetes API with custom resource definitions (CRDs)

Aggregation Layer

Extend API server with additional APIs and services

API Server Workflow

1. **Request Reception:** Receive HTTP request from client (kubectl, kubelet, etc.)
2. **Authentication:** Verify client identity using certificates, tokens, or other methods
3. **Authorization:** Check if client has permission to perform the requested operation
4. **Admission Control:** Run admission controllers to validate/mutate the request
5. **Validation:** Validate resource schema and business logic
6. **Storage:** Store the resource in etcd or retrieve from etcd
7. **Response:** Return response to client with resource data or status

Common API Operations

```
# Get all pods in default namespace
GET /api/v1/namespaces/default/pods

# Create a new deployment
POST /apis/apps/v1/namespaces/default/deployments

# Update a service
```

```
PUT /api/v1/namespaces/default/services/my-service

# Delete a pod
DELETE /api/v1/namespaces/default/pods/my-pod

# Watch for changes to deployments
GET /apis/apps/v1/namespaces/default/deployments?
watch=true
```

Security Features

- **TLS Encryption:** All communication encrypted in transit
- **RBAC:** Role-based access control for fine-grained permissions
- **Service Accounts:** Identity for pods and services
- **Network Policies:** Control network traffic between pods

Prepared By: Rashi Rana
Corporate Trainer

Scheduler

The **Scheduler** is responsible for assigning pods to nodes based on resource requirements, constraints, and policies. It makes scheduling decisions but doesn't actually run the pods.

Scheduling Process

1. **Watch for Unscheduled Pods:** Monitor API server for pods without node assignment
2. **Filtering:** Find nodes that meet pod requirements (predicates)
3. **Scoring:** Rank suitable nodes based on priorities
4. **Selection:** Choose the best node for the pod
5. **Binding:** Update pod specification with selected node

Scheduling Factors

Resource Requirements

CPU, memory, storage requests and limits

Node Constraints

Node selectors, affinity, and anti-affinity rules

Quality of Service

Data Locality

Guaranteed, Burstable, and BestEffort classes

Schedule pods close to their data sources

Scheduling Constraints

Constraint Type	Description	Example Use Case
Node Selector	Simple key-value label matching	Schedule on GPU nodes
Node Affinity	Advanced node selection rules	Prefer SSD storage nodes
Pod Affinity	Schedule pods together	Co-locate web and cache pods
Pod Anti-Affinity	Schedule pods apart	Spread replicas across zones
Taints & Tolerations	Repel pods from nodes	Dedicated nodes for specific workloads

Custom Scheduling

- **Custom Schedulers:** Write your own scheduling logic
- **Scheduler Extenders:** Extend default scheduler with webhooks
- **Multiple Schedulers:** Run different schedulers for different workloads
- **Scheduler Profiles:** Configure different scheduling behaviors

Scheduling Example

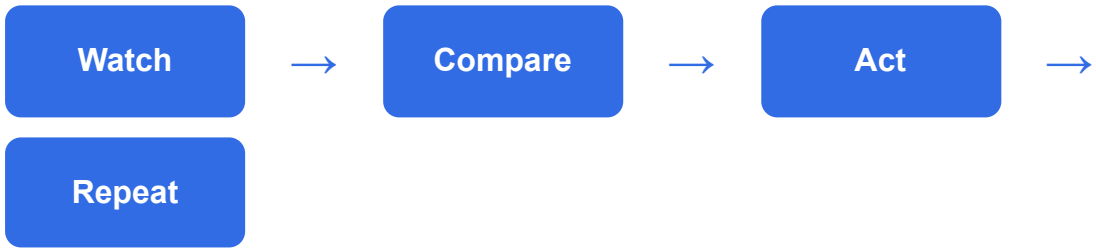
```
apiVersion: v1
kind: Pod
metadata:
  name: gpu-pod
spec:
  nodeSelector:
    accelerator: nvidia-tesla-k80
  affinity:
    nodeAffinity:
      requiredDuringSchedulingIgnoredDuringExecution:
        nodeSelectorTerms:
        - matchExpressions:
          - key: zone
            operator: In
            values: ["us-west1-a", "us-west1-b"]
  containers:
  - name: gpu-container
    image: tensorflow/tensorflow:latest-gpu
    resources:
      limits:
        nvidia.com/gpu: 1
```

Prepared By: Rashi Rana
Corporate Trainer

Controller Manager

The **Controller Manager** runs various controllers that regulate the state of the cluster. Controllers are control loops that watch the state of your cluster and make changes to move the current state toward the desired state.

Controller Pattern



Built-in Controllers

Deployment Controller

Manages ReplicaSets and rolling updates for Deployments

ReplicaSet Controller

Ensures specified number of pod replicas are running

Node Controller

Monitors node health and manages node lifecycle

Service Controller

Creates and manages load balancers for services

Endpoint Controller

Populates endpoint objects (joins Services & Pods)

Namespace Controller

Manages namespace lifecycle and cleanup

Controller Responsibilities

Controller	Watches	Actions
Deployment	Deployment objects	Create/update ReplicaSets, rolling updates
ReplicaSet	ReplicaSet objects	Create/delete pods to match desired count
DaemonSet	DaemonSet objects	Ensure pods run on all/selected nodes
Job	Job objects	Run pods to completion
CronJob	CronJob objects	Create jobs on schedule

Custom Controllers

- **Operators:** Application-specific controllers that extend Kubernetes
- **Custom Resource Definitions (CRDs):** Define new resource types
- **Controller Runtime:** Framework for building controllers
- **Kubebuilder:** SDK for building Kubernetes APIs and controllers

Controller Example: ReplicaSet

```
apiVersion: apps/v1
kind: ReplicaSet
metadata:
  name: nginx-replicaset
spec:
  replicas: 3
  selector:
    matchLabels:
      app: nginx
  template:
    metadata:
      labels:
        app: nginx
    spec:
      containers:
      - name: nginx
        image: nginx:1.21
        ports:
        - containerPort: 80
```

Key Concept: Controllers implement the declarative nature of Kubernetes by continuously working to achieve the desired state defined in your manifests.

Prepared By: Rashi Rana
Corporate Trainer

etcd

etcd is a distributed, reliable key-value store used by Kubernetes to store all cluster data. It's the single source of truth for the entire cluster state.

Key Characteristics

- **Distributed:** Runs across multiple nodes for high availability
- **Consistent:** Uses Raft consensus algorithm for data consistency
- **Reliable:** Fault-tolerant with automatic leader election
- **Fast:** Optimized for read-heavy workloads
- **Secure:** Supports TLS encryption and authentication
- **Watchable:** Clients can watch for changes to keys

What etcd Stores

Cluster State

All Kubernetes objects: pods, services, deployments, etc.

Configuration Data

ConfigMaps, Secrets, and cluster configuration

Metadata

Network Information

Resource versions, timestamps, and ownership information

Service endpoints, network policies, and ingress rules

etcd Architecture

Component	Role	Description
Leader	Primary node	Handles all write operations and coordinates cluster
Followers	Secondary nodes	Replicate data from leader, can handle read operations
Raft Log	Transaction log	Ordered sequence of all state changes
State Machine	Data store	Current state derived from applying log entries

High Availability Setup

- **Odd Number of Nodes:** 3, 5, or 7 nodes for fault tolerance
- **Quorum:** Majority of nodes must be available (e.g., 2 out of 3)
- **Geographic Distribution:** Spread across availability zones
- **Backup Strategy:** Regular snapshots and disaster recovery

etcd Operations

```
# Check etcd cluster health
etcdctl cluster-health

# List all keys
```

```
etcdctl ls / --recursive

# Get a specific key
etcdctl get /registry/pods/default/my-pod

# Create a backup
etcdctl snapshot save backup.db

# Restore from backup
etcdctl snapshot restore backup.db
```

Performance Considerations

- **SSD Storage:** Use fast storage for better performance
- **Network Latency:** Low latency between etcd nodes
- **Resource Allocation:** Adequate CPU and memory
- **Monitoring:** Watch for slow operations and disk usage

> **Critical:** etcd is the most critical component. Always backup etcd data regularly and test restore procedures.

Prepared By: Rashi Rana
Corporate Trainer

Kubelet

> The **Kubelet** is the primary node agent that runs on each worker node. It's responsible for managing pods and containers on the node, ensuring they match the desired state.

Key Responsibilities

- **Pod Lifecycle Management:** Create, start, stop, and delete pods
- **Container Health Monitoring:** Perform health checks and restart failed containers
- **Resource Management:** Enforce resource limits and requests
- **Volume Management:** Mount and unmount volumes for pods
- **Node Status Reporting:** Report node and pod status to API server
- **Image Management:** Pull container images as needed

Kubelet Workflow

1. **Watch API Server:** Monitor for pod assignments to this node
2. **Pull Images:** Download required container images
3. **Create Containers:** Start containers using container runtime
4. **Monitor Health:** Perform liveness and readiness probes
5. **Report Status:** Send pod and node status to API server
6. **Handle Updates:** Apply configuration changes and restarts

Health Checks

Liveness Probe

Determines if container is running. Restarts container if probe fails.

Readiness Probe

Determines if container is ready to serve traffic. Removes from service if fails.

Startup Probe

Checks if container has started. Disables other probes until successful.

Probe Types

Probe Type	Method	Use Case
HTTP GET	HTTP request to container	Web applications, REST APIs
TCP Socket	TCP connection attempt	Database connections, TCP services
Exec	Execute command in container	Custom health check scripts

Kubelet Configuration

```
# Liveness probe example
apiVersion: v1
kind: Pod
metadata:
  name: liveness-http
spec:
  containers:
  - name: liveness
    image: nginx
    livenessProbe:
      httpGet:
        path: /healthz
        port: 8080
      initialDelaySeconds: 3
      periodSeconds: 3
    readinessProbe:
      httpGet:
        path: /ready
        port: 8080
      initialDelaySeconds: 5
      periodSeconds: 5
```

Resource Management

- **CPU Limits:** Throttle CPU usage when limit is reached
- **Memory Limits:** Kill container if memory limit exceeded
- **Storage Limits:** Monitor and enforce storage quotas
- **QoS Classes:** Guaranteed, Burstable, BestEffort

> **Key Point:** Kubelet is the bridge between Kubernetes control plane and the actual container runtime, ensuring desired state is maintained on each node.

Prepared By: Rashi Rana
Corporate Trainer

Kube-Proxy

Kube-proxy is a network proxy that runs on each node, implementing part of the Kubernetes Service concept by maintaining network rules and performing connection forwarding.

Primary Functions

- **Service Discovery:** Route traffic to appropriate pods
- **Load Balancing:** Distribute requests across pod replicas
- **Network Rules:** Maintain iptables/IPVS rules for services
- **Session Affinity:** Route requests from same client to same pod
- **Health Checking:** Remove unhealthy pods from rotation

Proxy Modes

iptables Mode

Default mode using iptables rules for traffic routing. Good performance for most use cases.

IPVS Mode

Uses Linux IPVS for better performance with large numbers of services.

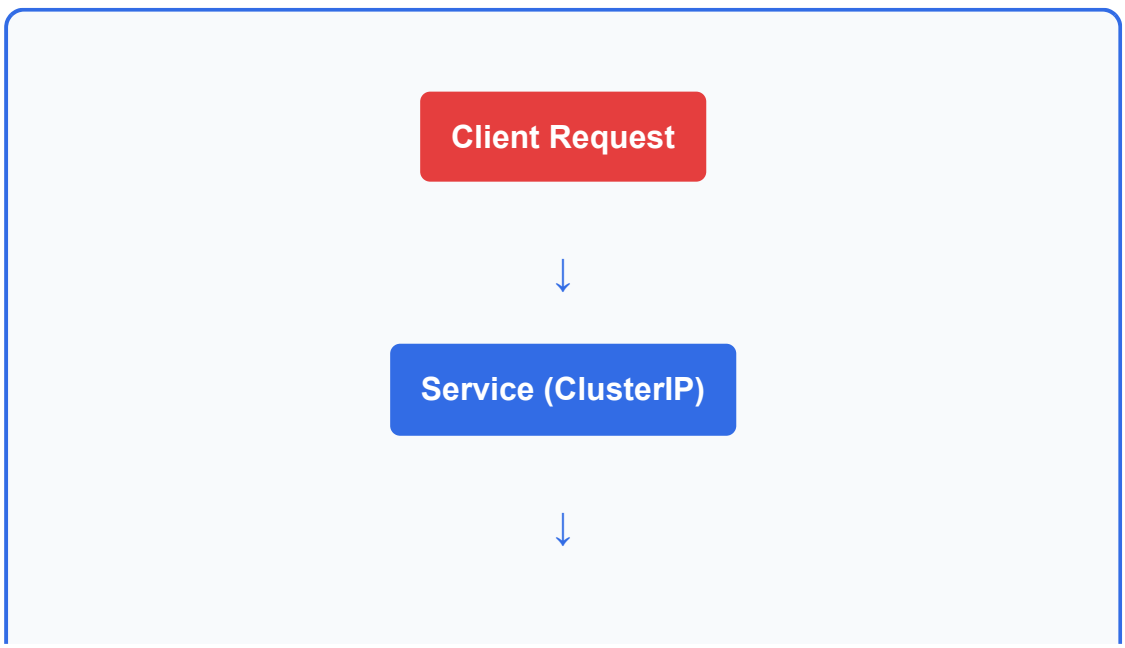
Userspace Mode

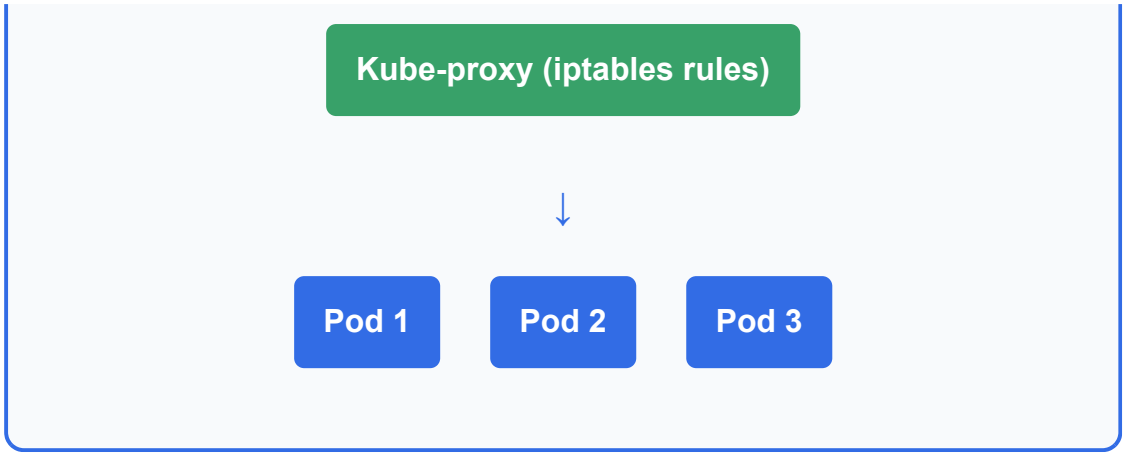
Legacy mode where kube-proxy acts as actual proxy. Slower but more compatible.

Service Types

Service Type	Description	Use Case
ClusterIP	Internal cluster communication	Microservice communication
NodePort	Expose service on node's IP	External access for testing
LoadBalancer	Cloud provider load balancer	Production external access
ExternalName	DNS CNAME record	External service integration

Traffic Flow Example





Service Configuration Example

```
apiVersion: v1
kind: Service
metadata:
  name: web-service
spec:
  selector:
    app: web
  ports:
  - protocol: TCP
    port: 80
    targetPort: 8080
  type: ClusterIP
  sessionAffinity: ClientIP # Optional: sticky sessions
```

Load Balancing Algorithms

- **Round Robin:** Default algorithm, distributes requests evenly
- **Session Affinity:** Route requests from same client to same pod
- **Least Connections:** Route to pod with fewest active connections (IPVS)
- **Weighted Round Robin:** Distribute based on pod weights (IPVS)

> **Important:** Kube-proxy only handles traffic routing. Actual load balancing logic is implemented by the service mesh or ingress

controller.

Prepared By: Rashi Rana
Corporate Trainer

Container Runtime

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

Container Runtime Interface (CRI)

- **Standardized API:** Common interface for different runtimes
- **Runtime Agnostic:** Kubernetes doesn't depend on specific runtime
- **Pluggable Architecture:** Easy to switch between runtimes
- **gRPC Protocol:** Communication between kubelet and runtime

Popular Container Runtimes

containerd

Industry standard, high-performance container runtime. Default in many Kubernetes distributions.

CRI-O

Lightweight runtime specifically designed for Kubernetes. OCI-compliant.

Docker Engine

Popular but deprecated in Kubernetes 1.24+. Uses

Kata Containers

Secure runtime using lightweight VMs for better isolation.

dockershim compatibility layer.

Runtime Comparison

Runtime	Performance	Security	Ecosystem
containerd	High	Good	Excellent
CRI-O	High	Good	Good
Docker	Medium	Good	Excellent
Kata	Medium	Excellent	Limited

Runtime Responsibilities

- **Image Management:** Pull, store, and manage container images
- **Container Lifecycle:** Create, start, stop, and delete containers
- **Resource Isolation:** CPU, memory, and network isolation
- **Storage Management:** Handle container filesystems and volumes
- **Network Setup:** Configure container networking
- **Security:** Apply security policies and constraints

OCI Standards

Runtime Specification

Defines how to run containers from filesystem bundles

Image Specification

Defines container image format and metadata

Distribution Specification

Defines how to distribute container images

Runtime Configuration

```
# Check current runtime
kubectl get nodes -o wide

# Runtime configuration in kubelet
--container-runtime=remote
--container-runtime-
endpoint=unix:///var/run/containerd/containerd.sock

# containerd configuration
/etc/containerd/config.toml
```

Security Features

- **Namespace Isolation:** Process, network, and filesystem isolation
- **Cgroups:** Resource limiting and accounting
- **Seccomp:** System call filtering
- **AppArmor/SELinux:** Mandatory access control
- **User Namespaces:** Root privilege isolation

> **Trend:** containerd has become the most popular runtime due to its performance, stability, and direct integration with Kubernetes.

Prepared By: Rashi Rana
Corporate Trainer

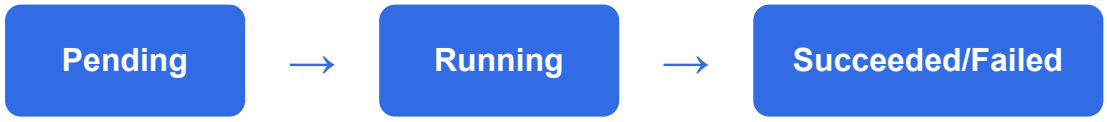
Understanding Pods

> A **Pod** is the smallest deployable unit in Kubernetes. It represents a group of one or more containers that share storage, network, and a specification for how to run the containers.

Pod Characteristics

- **Shared Network:** All containers share the same IP address and port space
- **Shared Storage:** Containers can share volumes for data exchange
- **Atomic Unit:** Containers in a pod are scheduled together
- **Ephemeral:** Pods are mortal and can be created/destroyed
- **Single Node:** All containers in a pod run on the same node

Pod Lifecycle



Pod Phases

Phase	Description	Next Steps
Pending	Pod accepted but not yet running	Scheduling, image pulling

Phase	Description	Next Steps
Running	At least one container is running	Normal operation
Succeeded	All containers terminated successfully	Job completion
Failed	At least one container failed	Restart or cleanup
Unknown	Pod state cannot be determined	Node communication issues

Pod Patterns

Single Container

Most common pattern - one application container per pod

Sidecar

Helper container alongside main application (logging, monitoring)

Ambassador

Proxy container for external service communication

Adapter

Transform container output to standard format

Pod Specification Example

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

```
    name: multi-container-pod
    labels:
      app: web
  spec:
    containers:
    - name: web-server
      image: nginx:1.21
      ports:
      - containerPort: 80
      volumeMounts:
      - name: shared-data
        mountPath: /usr/share/nginx/html
    - name: log-collector
      image: fluentd:latest
      volumeMounts:
      - name: shared-data
        mountPath: /var/log
    volumes:
    - name: shared-data
      emptyDir: {}
```

Pod Networking

- **Cluster IP:** Each pod gets unique IP address
- **Port Sharing:** Containers share network namespace
- **Localhost Communication:** Containers can communicate via localhost
- **DNS Resolution:** Automatic service discovery

- **Single Responsibility:** One main application per pod
- **Stateless Design:** Store state externally
- **Resource Limits:** Define CPU and memory limits
- **Health Checks:** Implement liveness and readiness probes
- **Graceful Shutdown:** Handle SIGTERM signals properly

Prepared By: Rashi Rana
Corporate Trainer

Types of Deployments

Kubernetes provides several deployment strategies to manage application updates, each with different characteristics for availability, resource usage, and rollback capabilities.

Deployment Strategies

Rolling Update

Default strategy. Gradually replace old pods with new ones. Zero downtime deployment.

Recreate

Terminate all old pods before creating new ones. Causes downtime but uses fewer resources.

Blue-Green

Run two identical environments. Switch traffic instantly between versions.

Canary

Gradually shift traffic from old to new version. Test with subset of users first.

Rolling Update Configuration

```
apiVersion: apps/v1
kind: Deployment
metadata:
```

```
  name: nginx-deployment
spec:
  replicas: 3
  strategy:
    type: RollingUpdate
    rollingUpdate:
      maxUnavailable: 1 # Max pods that can be unavailable
      maxSurge: 1 # Max pods above desired replica count
  selector:
    matchLabels:
      app: nginx
  template:
    metadata:
      labels:
        app: nginx
    spec:
      containers:
      - name: nginx
        image: nginx:1.21
        ports:
        - containerPort: 80
```

Deployment Strategy Comparison

Strategy	Downtime	Resource Usage	Rollback Speed	Risk
Rolling Update	None	Medium	Medium	Low
Recreate	Yes	Low	Fast	Medium
Blue-Green	None	High (2x)	Instant	Medium
Canary	None	Medium	Fast	Low

Kubernetes Workload Types

Deployment

Stateless applications with rolling updates and scaling capabilities

StatefulSet

Stateful applications with stable network identities and persistent storage

DaemonSet

Run one pod per node (logging agents, monitoring, network plugins)

Job

Run pods to completion for batch processing or one-time tasks

CronJob

Run jobs on schedule (backups, reports, cleanup tasks)

Deployment Commands

```
# Create deployment
kubectl create deployment nginx --image=nginx:1.21

# Update deployment image
kubectl set image deployment/nginx nginx=nginx:1.22

# Check rollout status
kubectl rollout status deployment/nginx

# View rollout history
kubectl rollout history deployment/nginx
```

```
# Rollback to previous version
kubectl rollout undo deployment/nginx

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

Advanced Deployment Patterns

- **A/B Testing:** Split traffic between versions for testing
- **Feature Flags:** Control feature rollout without deployment
- **Progressive Delivery:** Automated canary with metrics-based decisions
- **Shadow Deployment:** Mirror production traffic to new version

Prepared By: Rashi Rana
Corporate Trainer

Stateful vs Stateless Applications

Understanding the difference between **stateful** and **stateless** applications is crucial for choosing the right Kubernetes workload type and deployment strategy.

Stateless Applications

Characteristics

- No persistent data storage
- Each request is independent
- Easily replaceable
- Horizontally scalable

Examples

- Web servers (Nginx, Apache)
- API gateways
- Microservices
- Load balancers

Kubernetes Resource

- Deployment
- ReplicaSet
- Service
- Horizontal Pod Autoscaler

Benefits

- Easy to scale
- Fast recovery
- Simple deployment
- Cloud-native friendly

Stateful Applications

Characteristics

- Persistent data storage
- Stable network identity
- Ordered deployment
- Data consistency requirements

Examples

- Databases (MySQL, PostgreSQL)
- Message queues (Kafka)
- Distributed systems (Elasticsearch)
- File systems

Kubernetes Resource

- StatefulSet
- Persistent Volumes
- Headless Services
- Storage Classes

Challenges

- Complex scaling
- Data migration
- Backup/restore
- Network dependencies

Detailed Comparison

Aspect	Stateless	Stateful
Data Storage	No persistent data	Requires persistent storage
Pod Identity	Interchangeable pods	Unique, stable identities
Scaling	Easy horizontal scaling	Complex, often manual scaling
Deployment Order	Parallel deployment	Sequential, ordered deployment

Aspect	Stateless	Stateful
Network	Any IP address	Stable network identifiers
Recovery	Fast, simple restart	Complex data recovery

StatefulSet Example

```
apiVersion: apps/v1
kind: StatefulSet
metadata:
  name: mysql
spec:
  serviceName: mysql
  replicas: 3
  selector:
    matchLabels:
      app: mysql
  template:
    metadata:
      labels:
        app: mysql
    spec:
      containers:
      - name: mysql
        image: mysql:8.0
        env:
        - name: MYSQL_ROOT_PASSWORD
          value: "password"
        volumeMounts:
        - name: mysql-storage
          mountPath: /var/lib/mysql
  volumeClaimTemplates:
  - metadata:
      name: mysql-storage
    spec:
      accessModes: ["ReadWriteOnce"]
      resources:
        requests:
          storage: 10Gi
```

Best Practices

Stateless Design

- Store state externally
- Use caching layers
- Implement health checks
- Design for failure

Stateful Management

- Plan data migration
- Implement backups
- Monitor data consistency
- Use persistent volumes

Architecture Recommendation: Design applications to be stateless when possible. Use external data stores and caching layers to maintain state outside the application containers.

Prepared By: Rashi Rana
Corporate Trainer

Summary and Next Steps

Key Concepts Covered

- Why container orchestration is essential for modern applications
- Kubernetes architecture and core components
- Master node components: API Server, Scheduler, Controller Manager, etcd
- Worker node components: Kubelet, Kube-proxy, Container Runtime
- Understanding Pods as the fundamental unit
- Different deployment strategies and workload types
- Stateful vs Stateless application patterns

Kubernetes Benefits Recap

Operational Excellence

Automated deployments, self-healing, and consistent environments

Scalability

Horizontal and vertical scaling based on demand

Portability

Run anywhere - on-premises, cloud, or hybrid environments

Developer Productivity

Focus on code, not infrastructure management

Learning Path Forward

1. **Hands-on Practice:** Set up local Kubernetes cluster (minikube, kind, or Docker Desktop)
2. **Basic Operations:** Learn kubectl commands and YAML manifest creation
3. **Networking:** Understand Services, Ingress, and Network Policies
4. **Storage:** Work with Persistent Volumes and Storage Classes
5. **Configuration:** Master ConfigMaps, Secrets, and environment management
6. **Monitoring:** Implement logging, metrics, and observability
7. **Security:** Learn RBAC, Pod Security Standards, and best practices
8. **Production:** High availability, backup strategies, and troubleshooting

Recommended Tools and Resources

- **Local Development:** Docker Desktop, minikube, kind, k3s
- **Cloud Platforms:** EKS (AWS), GKE (Google), AKS (Azure)
- **Package Management:** Helm charts for application deployment

- **GitOps:** ArgoCD, Flux for continuous deployment
- **Monitoring:** Prometheus, Grafana, Jaeger for observability
- **Security:** Falco, OPA Gatekeeper for policy enforcement

Next Session Topics

- **Hands-on Labs:** Setting up your first Kubernetes cluster
- **kubectl Mastery:** Essential commands and operations
- **YAML Manifests:** Writing and managing Kubernetes resources
- **Services & Networking:** Exposing applications and service discovery
- **Storage Solutions:** Persistent data management

> **Remember:** Kubernetes is a journey, not a destination. Start with the basics, practice regularly, and gradually build your expertise. The container orchestration skills you develop will be invaluable in modern cloud-native environments.

Prepared By: Rashi Rana
Corporate Trainer