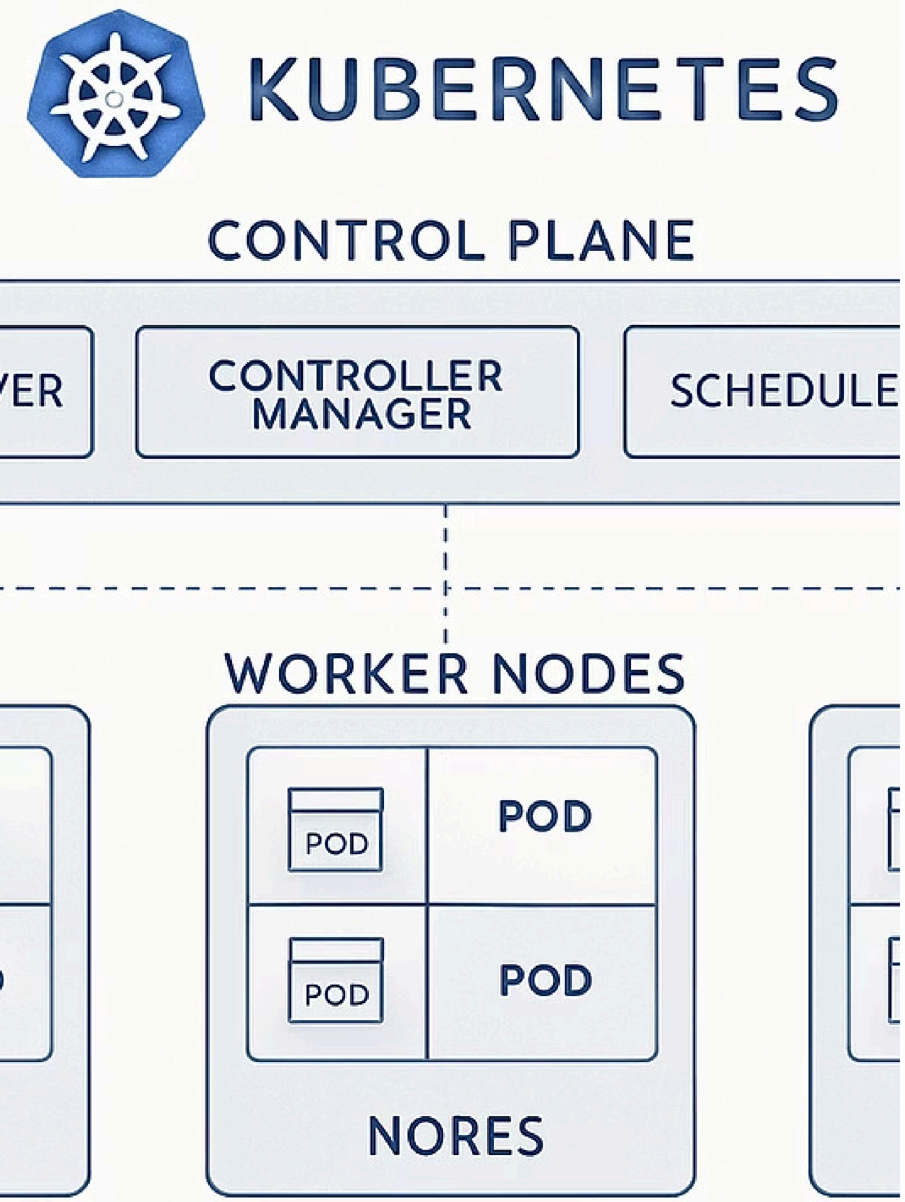


Kubernetes Administration: A Comprehensive Technical Guide

A technical deep dive into Kubernetes architecture, installation, configuration, and resource management for experienced administrators.

Course Agenda

1

Core Concepts

- Container Orchestration Fundamentals
- Kubernetes Architecture
- Control Plane Components

2

Installation & Configuration

- Cluster Design Principles
- Network Implementation
- Installation Validation

3

Resource Management

- Pod Lifecycle Management
- Workload Controllers
- Service Implementation

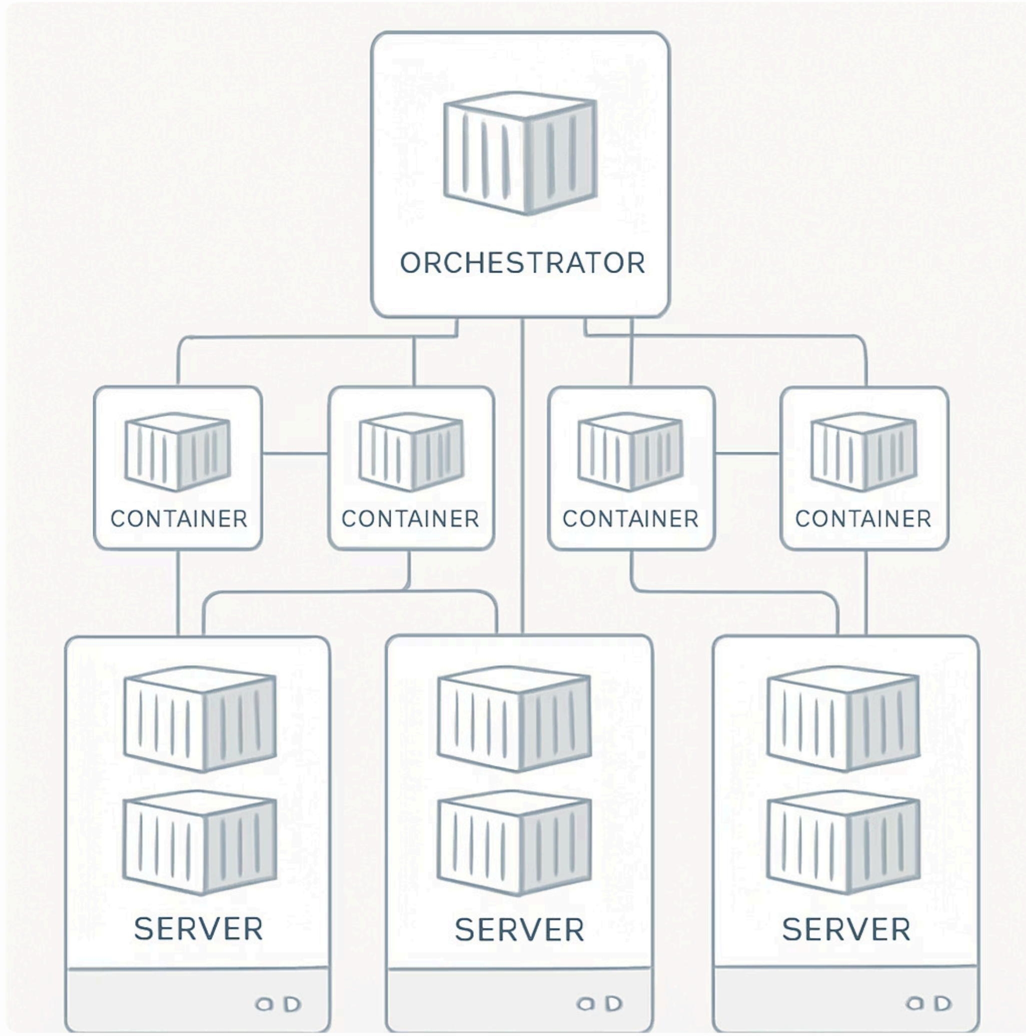
Module 1: Core Concepts

Understanding the fundamentals of container orchestration and the Kubernetes architecture

Container Orchestration: The Problem Space

Challenges of Container Deployment

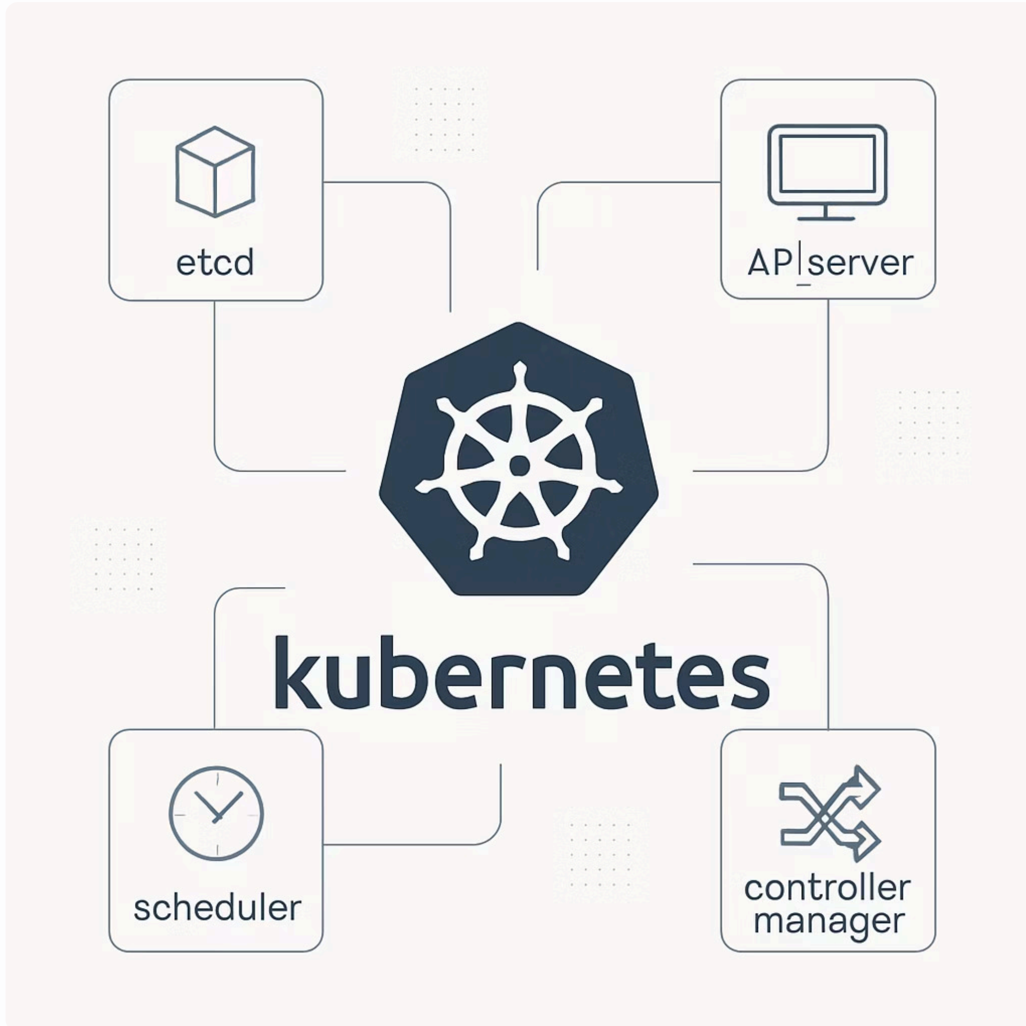
- Manual container deployment is error-prone
- Scaling containers requires significant overhead
- Service discovery becomes increasingly complex
- Load balancing across container instances
- Health monitoring and self-healing capabilities
- Rolling updates without downtime

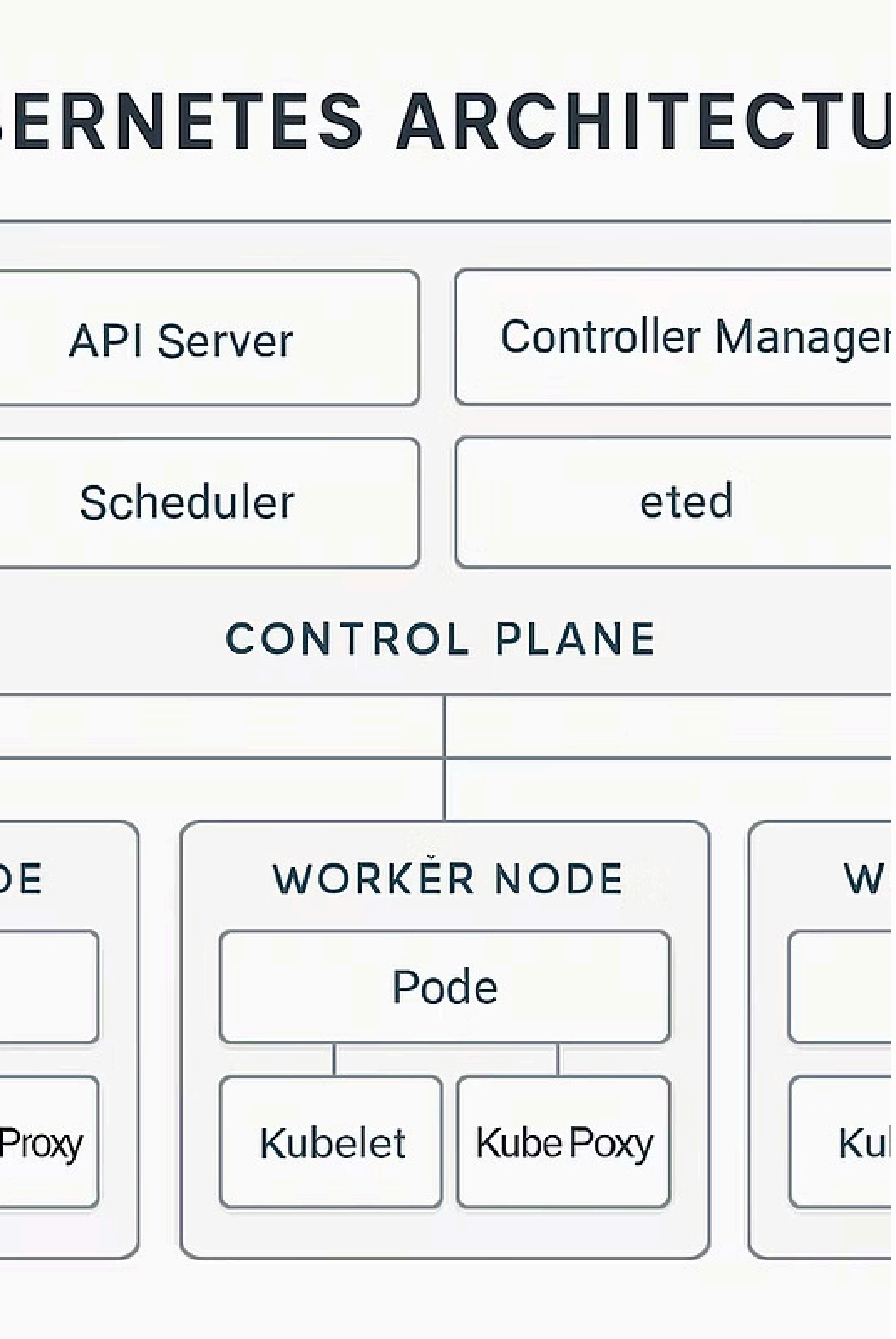


What is Kubernetes?

Kubernetes is an open-source platform designed to automate deploying, scaling, and operating application containers.

- Originally developed by Google, now maintained by CNCF
- Declarative configuration through YAML/JSON manifests
- Distributed system with control plane/worker architecture
- Self-healing mechanisms for fault tolerance
- Horizontal scaling with load balancing





Kubernetes Architecture: High Level

A distributed system with distinct control plane and worker node components that communicate to orchestrate containers at scale.

Control Plane Components

kube-apiserver

Front-end for the Kubernetes control plane exposing the Kubernetes API. Validates and processes API requests.

- RESTful API interface for cluster operations
- Authentication and authorization mechanisms
- Horizontal scaling capability for redundancy

etcd

Consistent and highly-available key-value store used as Kubernetes' backing store for all cluster data.

- Uses Raft consensus algorithm
- Stores cluster state and configuration
- Performance tuning critical for large clusters

kube-scheduler

Watches for newly created Pods with no assigned node, and selects a node for them to run on.

- Considers resource requirements, constraints
- Applies scheduling policies and affinity rules
- Uses scoring algorithm for optimal placement

Control Plane Components (Continued)

kube-controller-manager

Runs controller processes that regulate the state of the system.

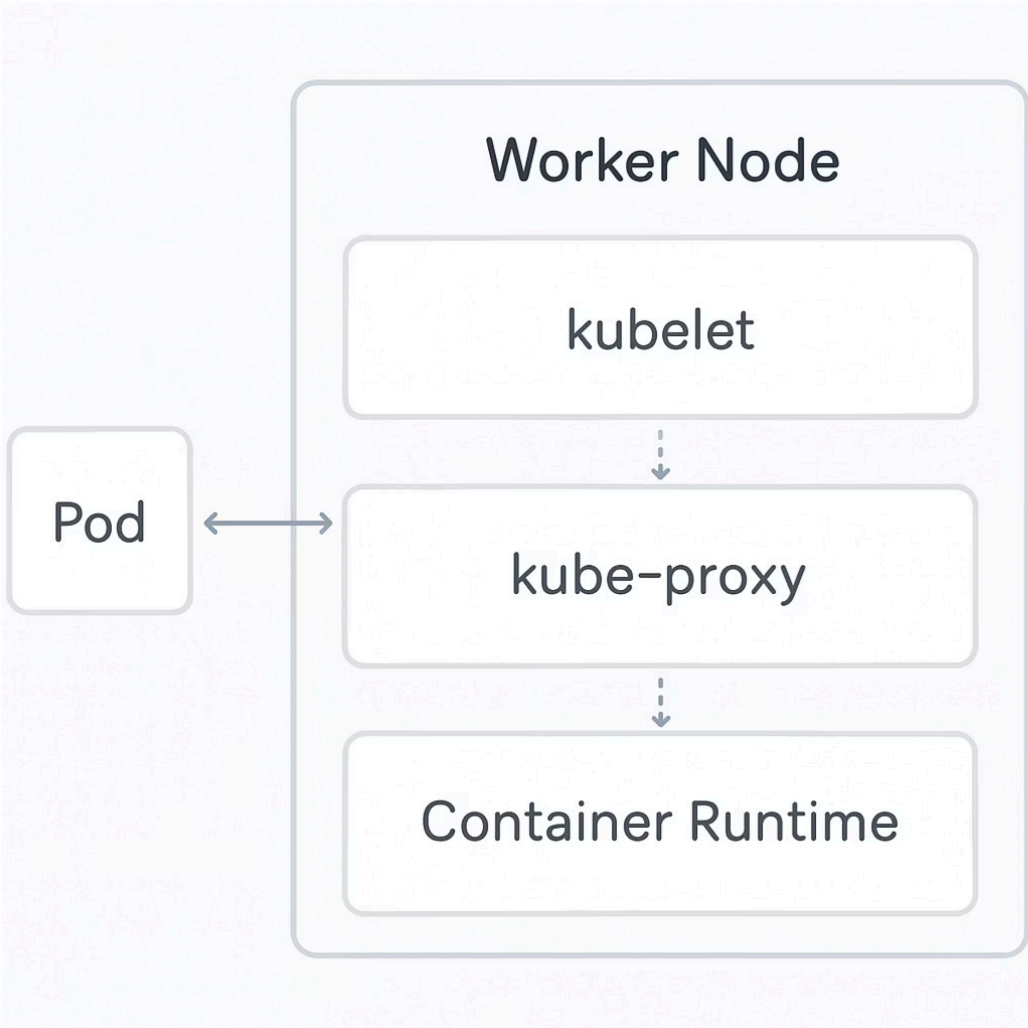
- Node Controller: Monitors node health
- Replication Controller: Maintains pod count
- Endpoints Controller: Populates endpoints
- Service Account & Token Controllers: Create accounts and API tokens

cloud-controller-manager

Links your cluster to your cloud provider's API, separating cloud-dependent from cloud-independent components.

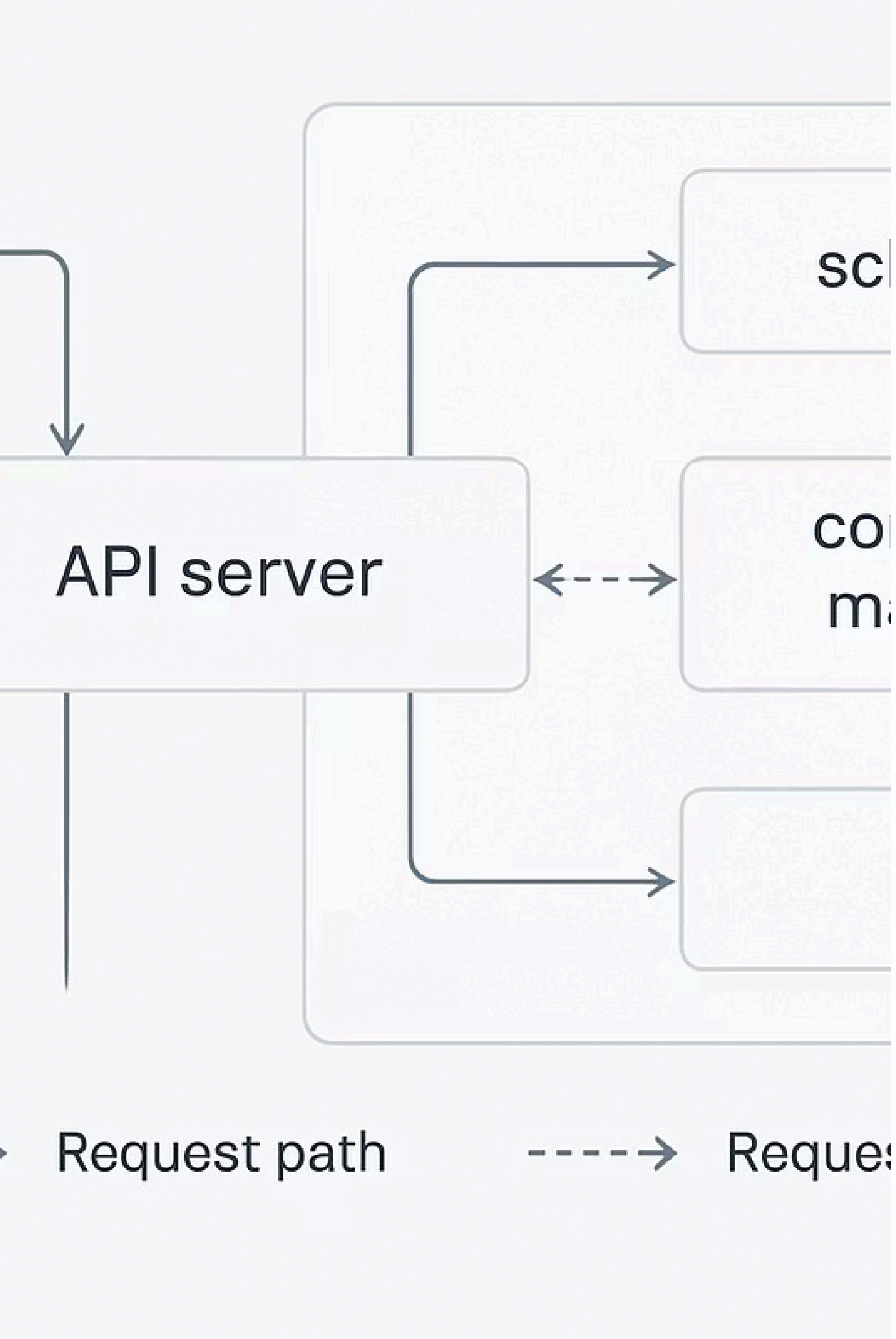
- Node Controller: Check cloud provider for node deletion
- Route Controller: Set up routes in cloud infrastructure
- Service Controller: Create, update, delete cloud load balancers

Worker Node Components



- **kubelet**: Agent that ensures containers are running in a Pod
- **kube-proxy**: Network proxy implementing the Kubernetes Service concept
- **Container Runtime**: Software responsible for running containers (containerd, CRI-O)

Worker nodes perform the actual container workloads and are managed by the control plane components.



Communication Flow in Kubernetes

All communication flows through the API server, which acts as the central hub for the entire cluster.

Kubernetes API Overview

API Structure

- REST-based API with JSON/YAML formats
- Resource versioning (alpha, beta, stable)
- API groups organize related resources
- Resource operations: GET, POST, PUT, DELETE, PATCH

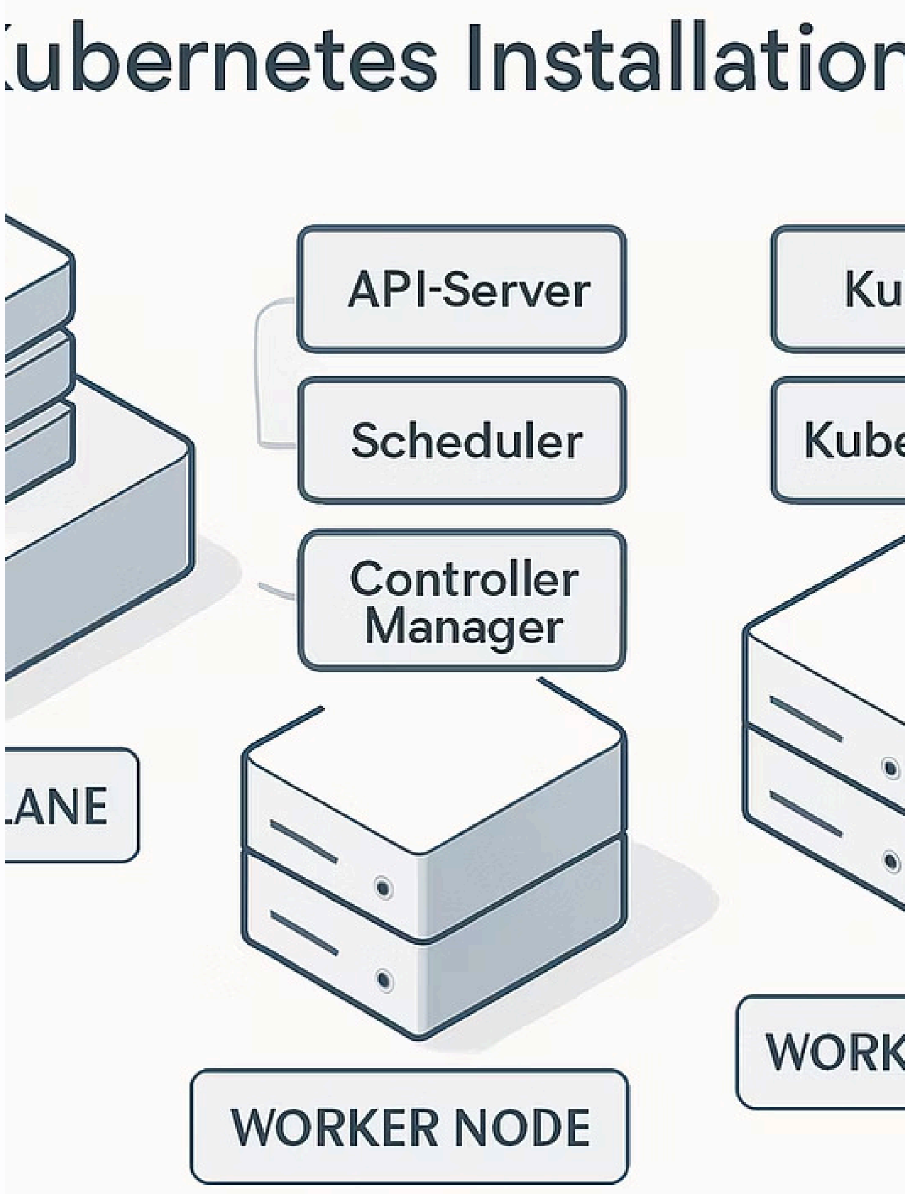
```
# Example API Path Structure
/api/v1/namespaces/default/pods

# API Groups Format
/apis/apps/v1/namespaces/default/deployments

# Common API Operations
kubectl get pods -n kube-system
kubectl describe pod nginx-pod
kubectl apply -f deployment.yaml
```

Module 2: Installation, Configuration & Validation

Designing and implementing a production-ready Kubernetes cluster



Cluster Design Considerations

High Availability

- Multi-master setup (minimum 3 control plane nodes)
- etcd quorum requirements (2n+1 nodes)
- Load balancer for API server
- Staggered upgrade process

Infrastructure Planning

- On-premises vs cloud provider
- Bare metal vs virtualized environments
- Node sizing and resource allocation
- Storage backends (local vs networked)

Security Architecture

- Network security policies
- RBAC implementation
- Secrets management strategy
- Runtime security (seccomp, AppArmor)

Production Cluster Deployment Options

Manual Deployment

Using kubeadm tool for step-by-step cluster creation

- Full control over configuration
- Complex, error-prone process
- Requires deep technical knowledge

Managed Services

Using cloud providers' managed Kubernetes (EKS, GKE, AKS)

- Simplified operations
- Built-in high availability
- Automatic version upgrades

Automated Installers

Using tools like kops, kubespray, or RKE

- Infrastructure as code approach
- Repeatable deployments
- Support for multiple environments

kubeadm Installation Process

Prepare Infrastructure

Set up servers, configure network, and install dependencies

```
# Install container runtime
apt-get update
apt-get install -y containerd.io

# Configure containerd
mkdir -p /etc/containerd
containerd config default > /etc/containerd/config.toml
systemctl restart containerd
```

Initialize Control Plane

Bootstrap the first control plane node

```
# Initialize the control plane
kubeadm init --control-plane-
endpoint="LOAD_BALANCER_DNS:LOAD_BALANCER_P
ORT" \
  --pod-network-cidr=192.168.0.0/16 \
  --upload-certs

# Configure kubectl for admin
mkdir -p $HOME/.kube
cp -i /etc/kubernetes/admin.conf $HOME/.kube/config
```

Deploy Network Plugin

Install CNI plugin for pod networking

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

# Verify nodes are Ready
kubectl get nodes
```

Join Worker Nodes

Add worker nodes to the cluster

```
# Run on each worker node
kubeadm join
LOAD_BALANCER_DNS:LOAD_BALANCER_PORT \
 --token TOKEN \
 --discovery-token-ca-cert-hash SHA256:HASH
```

Networking Solutions Comparison

CNI Plugin	Features	Performance	Use Cases
Calico	BGP-based routing, advanced policy, eBPF dataplane	High (eBPF), Medium (standard)	Production environments requiring network policy
Flannel	Simple overlay, VXLAN encapsulation	Medium	Basic deployments with minimal configuration
Cilium	eBPF-based, layer 3-7 visibility, advanced policy	Very High	Security-focused environments, microservices
Weave Net	Encrypted networking, multi-cloud mesh	Medium	Multi-cloud environments requiring encryption

Network Solution Configuration: Calico

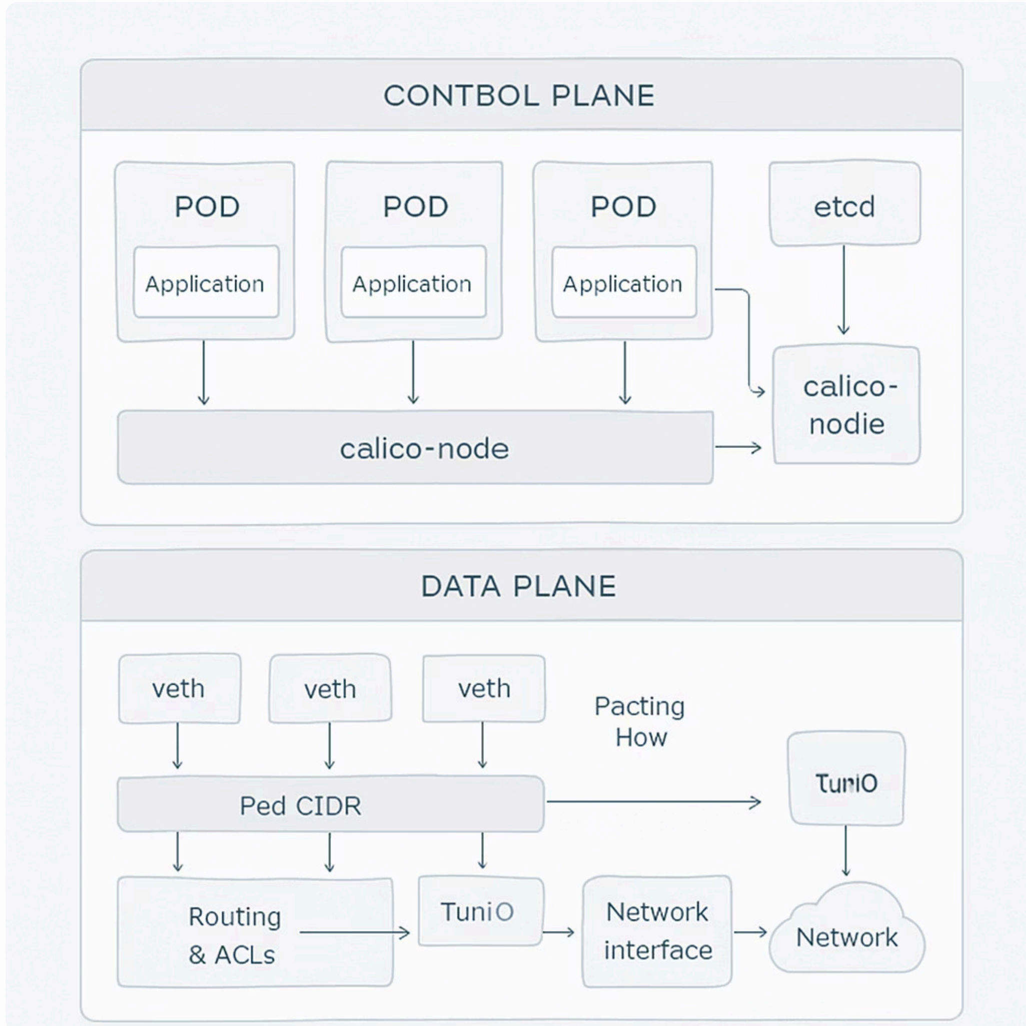
Installation Steps

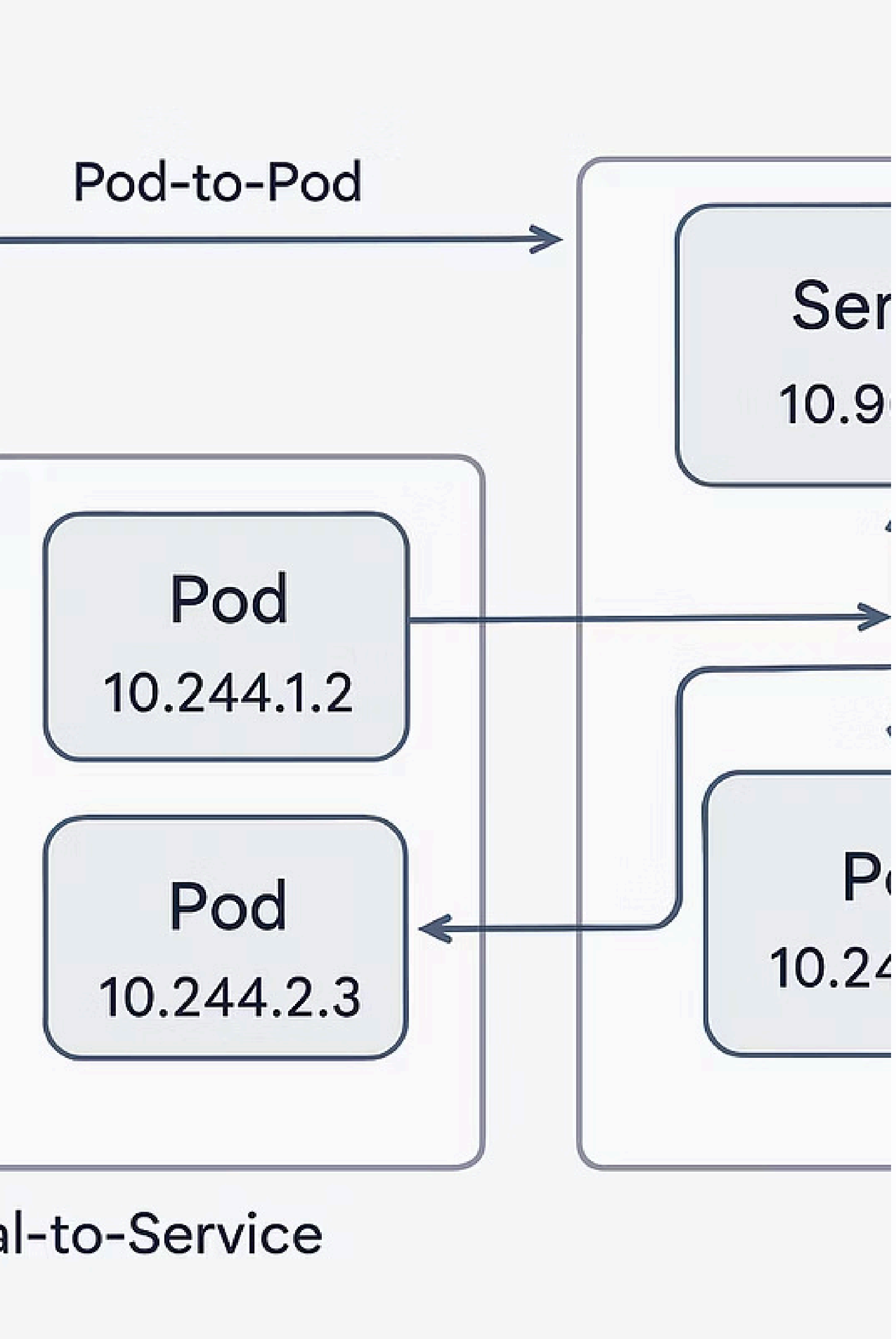
```
# Install Calico using manifest
kubectl apply -f
https://docs.projectcalico.org/manifests/calico.yaml

# Verify Calico pods are running
kubectl get pods -n kube-system -l k8s-app=calico-node
```

Advanced Configuration

```
# Apply custom IP pool
kubectl apply -f - <
```





Kubernetes Network Model

- Every Pod gets its own IP address
- Pods on a node can communicate with all pods on all nodes without NAT
- Agents on a node can communicate with all pods on that node
- CNI plugins implement this model in different ways

Cluster Validation with kubectl

1 Verify Node Status

```
kubectl get nodes
kubectl describe node
```

Check for Ready status and capacity/allocatable resources

2 Check Control Plane Health

```
kubectl get pods -n kube-system
kubectl get componentstatuses
```

Verify all control plane components are running

3 Test Core Functionality

```
# Deploy test pod
kubectl run nginx --image=nginx
kubectl get pods
kubectl port-forward nginx 8080:80
```

Validate pod scheduling and network connectivity

4 Verify API Access

```
kubectl auth can-i --list
kubectl api-resources
```

Confirm proper API access and available resources

Troubleshooting Installation Issues

Network Issues

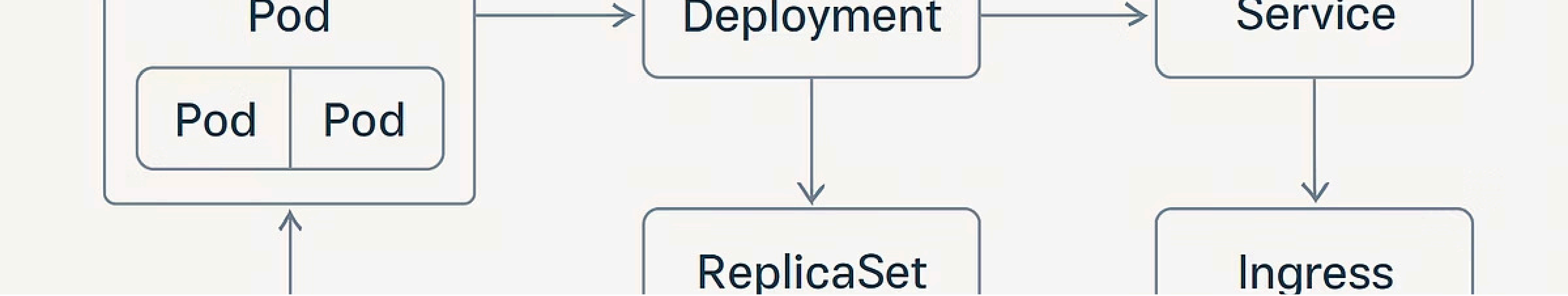
- Check firewall rules for required ports
- Verify network plugin is correctly installed
- Test pod-to-pod communication
- Troubleshoot with `tcpdump` and `ping`

Certificate Issues

- Check certificate expiration with `kubeadm certs check-expiration`
- Renew certificates with `kubeadm certs renew all`
- Verify CA trust chain is intact

Node Issues

- Check kubelet logs: `journalctl -u kubelet`
- Verify kubelet configuration
- Check system requirements (CPU, memory, kernel parameters)



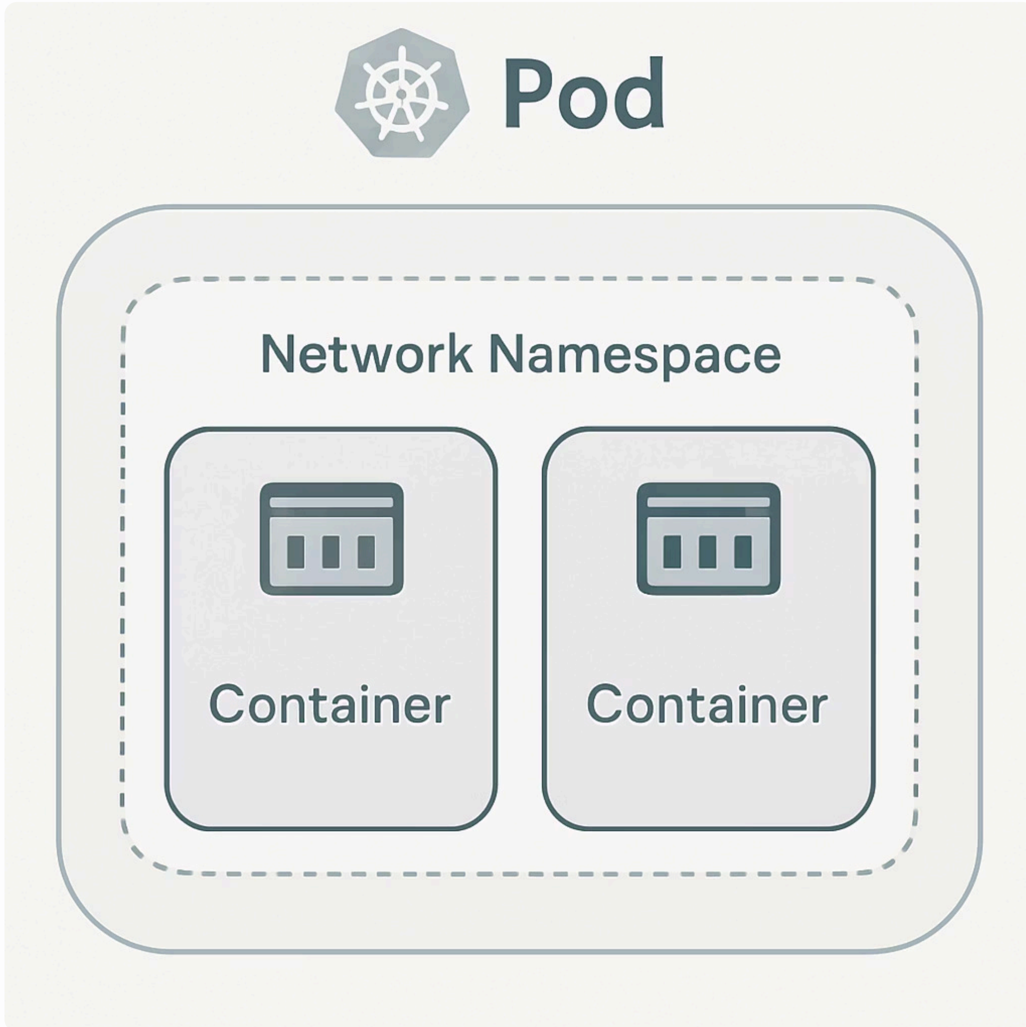
Module 3: Creating Kubernetes Resources

Understanding and implementing core Kubernetes resources for application deployment

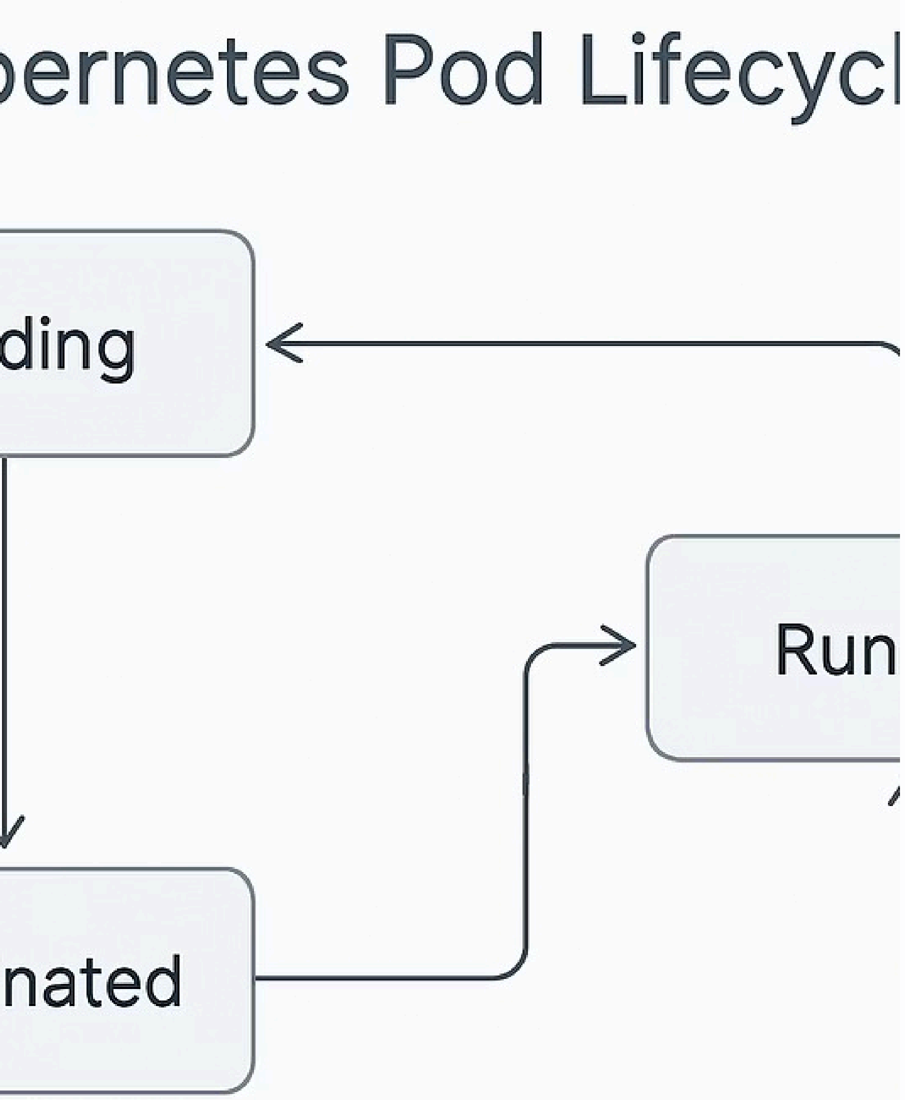
Pods: The Atomic Unit

Pod Characteristics

- Smallest deployable unit in Kubernetes
- Contains one or more containers
- Shares network namespace (same IP address)
- Can share storage volumes
- Ephemeral by design - not self-healing




```yaml
apiVersion: v1
kind: Pod
metadata:
 name: nginx-pod
spec:
 containers:
 - name: nginx
 image: nginx:1.19
 ports:
 - containerPort: 80
```



Pod Lifecycle States

- **Pending**: Pod accepted but containers not yet created
- **Running**: Pod bound to node, all containers created, at least one running
- **Succeeded**: All containers terminated successfully
- **Failed**: All containers terminated, at least one failed
- **Unknown**: State cannot be determined

Multi-Container Pod Patterns

Sidecar Pattern

Enhances the main container with additional functionality

- Log collectors
- File synchronizers
- Configuration updaters

```
containers:
- name: web
  image: nginx
- name: log-collector
  image: fluentd
```

Ambassador Pattern

Proxies network connections to the main container

- Database proxies
- Connection pooling
- TLS termination

```
containers:
- name: app
  image: app
- name: redis-ambassador
  image: redis-proxy
```

Adapter Pattern

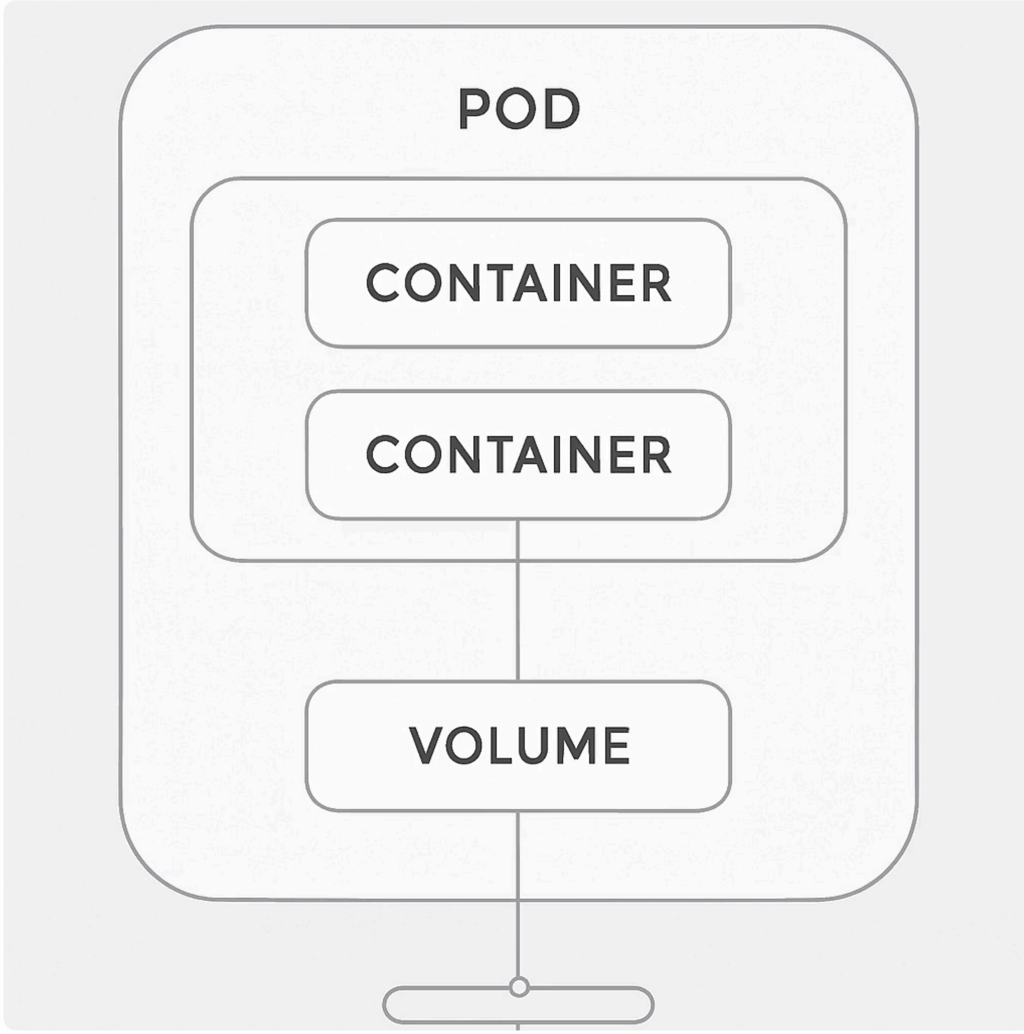
Standardizes output from the main container

- Metric exporters
- Format converters
- API normalizers

```
containers:
- name: app
  image: app
- name: prometheus-adapter
  image: prom-adapter
```

Pod Configuration: Volume Mounts

```yaml
apiVersion: v1
kind: Pod
metadata:
  name: volume-pod
spec:
  containers:
  - name: container1
    image: nginx
    volumeMounts:
    - name: shared-data
      mountPath: /usr/share/nginx/html
  - name: container2
    image: debian
    volumeMounts:
    - name: shared-data
      mountPath: /data
    command: ["/bin/sh"]
    args: ["-c", "echo Hello > /data/index.html && sleep
3600"]
  volumes:
  - name: shared-data
    emptyDir: {}
```



Volumes enable data sharing between containers and provide persistence beyond container lifecycle.

Pod Configuration: Resource Requests and Limits

Resource Management

- **Requests**: Minimum resources guaranteed to the container
- **Limits**: Maximum resources the container can use
- CPU measured in cores or millicores (m)
- Memory measured in bytes (Ki, Mi, Gi)
- Affects scheduling decisions and QoS class

```yaml
apiVersion: v1
kind: Pod
metadata:
  name: resource-pod
spec:
  containers:
  - name: app
    image: nginx
    resources:
      requests:
        memory: "64Mi"
        cpu: "250m"
      limits:
        memory: "128Mi"
        cpu: "500m"
```

QoS Classes: Guaranteed (requests=limits), Burstable (requests

Labels and Selectors

Labels

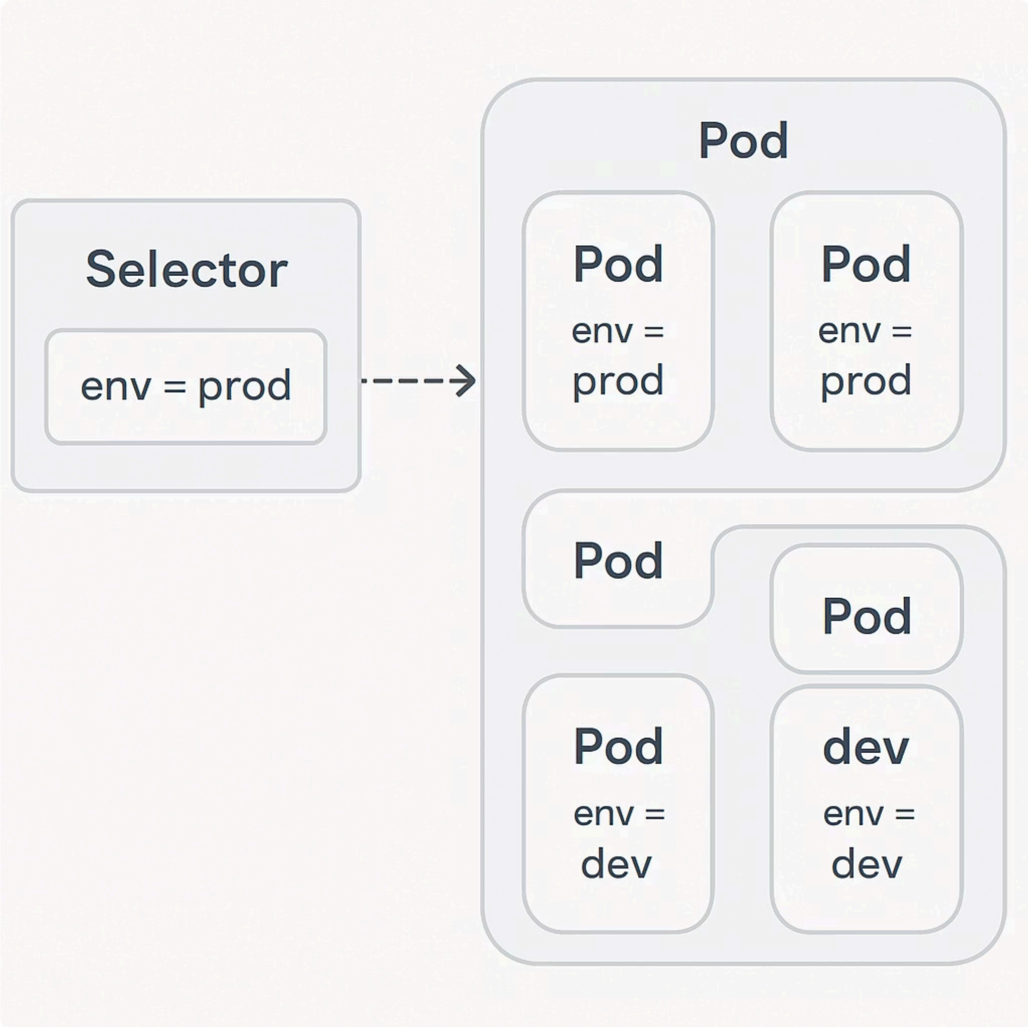
Key-value pairs attached to objects for identification and organization.

```
metadata:
  labels:
    app: nginx
    tier: frontend
    environment: production
    version: v1.0.0
```

Label Selectors

```
# Equality-based
kubectl get pods -l
environment=production,tier=frontend

# Set-based
kubectl get pods -l 'environment in (production,staging)'
```



Labels enable horizontal selection across resources while annotations provide non-identifying metadata.

Managing Labels in Practice

1 Adding Labels

```
# Add label to existing pod
kubectl label pod nginx-pod tier=frontend

# Add label to multiple pods
kubectl label pods -l app=nginx
environment=production

# Add label in manifest
metadata:
  labels:
    app: nginx
```

2 Updating Labels

```
# Overwrite existing label
kubectl label --overwrite pod nginx-pod
tier=backend

# Remove a label
kubectl label pod nginx-pod tier-
```

3 Selecting with Labels

```
# List pods with specific labels
kubectl get pods -l 'environment in (production),tier
notin (frontend)'

# Count pods matching selector
kubectl get pods -l app=nginx --no-headers | wc -l
```

4 Label Best Practices

- Use consistent naming conventions
- Consider reverse-DNS notation for custom labels
- Label for multiple dimensions (app, env, tier, version)
- Avoid putting non-identifying info in labels (use annotations)

Replication Controllers vs ReplicaSets

Replication Controller (Legacy)

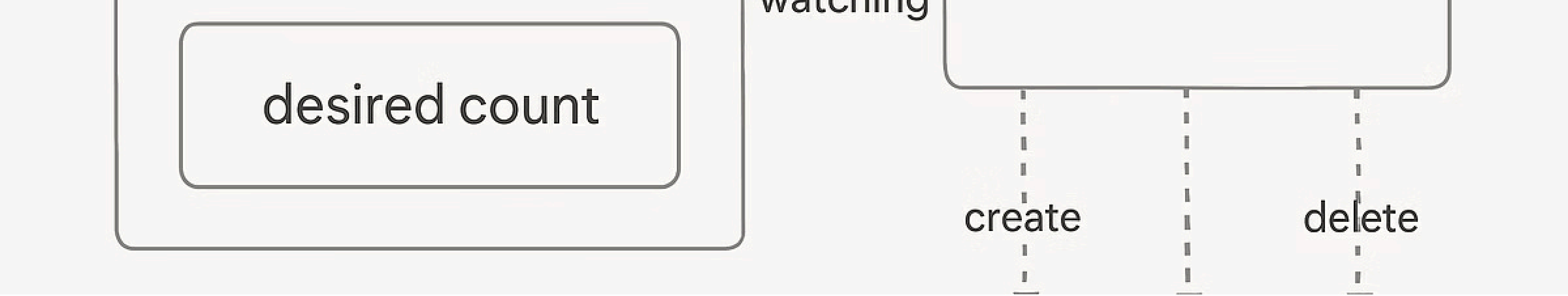
- Original replication implementation
- Maintains specified number of pod replicas
- Basic selector support (equality only)
- Gradually being replaced by ReplicaSets

```
apiVersion: v1
kind: ReplicationController
metadata:
  name: nginx-rc
spec:
  replicas: 3
  selector:
    app: nginx
  template:
    metadata:
      labels:
        app: nginx
    spec:
      containers:
      - name: nginx
        image: nginx
```

ReplicaSet (Recommended)

- Next-generation replication controller
- Supports set-based selectors
- More expressive label queries
- Used by Deployments behind the scenes

```
apiVersion: apps/v1
kind: ReplicaSet
metadata:
 name: nginx-rs
spec:
 replicas: 3
 selector:
 matchLabels:
 app: nginx
 matchExpressions:
 - {key: tier, operator: In, values: [frontend]}
 template:
 metadata:
 labels:
 app: nginx
 tier: frontend
 spec:
 containers:
 - name: nginx
 image: nginx
```



How ReplicaSets Work

- ReplicaSet controller continuously monitors pod count
- If actual count < desired: Creates new pods from template
- If actual count > desired: Deletes excess pods
- Uses owner references to track its pods
- Provides self-healing mechanism at the pod level

ReplicaSet Operations

1 Create and Inspect

```
# Create ReplicaSet
kubectl apply -f replicaset.yaml

# View ReplicaSets
kubectl get rs

# Get details
kubectl describe rs nginx-rs
```

2 Scaling

```
# Imperative scaling
kubectl scale rs nginx-rs --replicas=5

# Declarative scaling (modify yaml and apply)
spec:
  replicas: 5
```

3 Update Pod Template

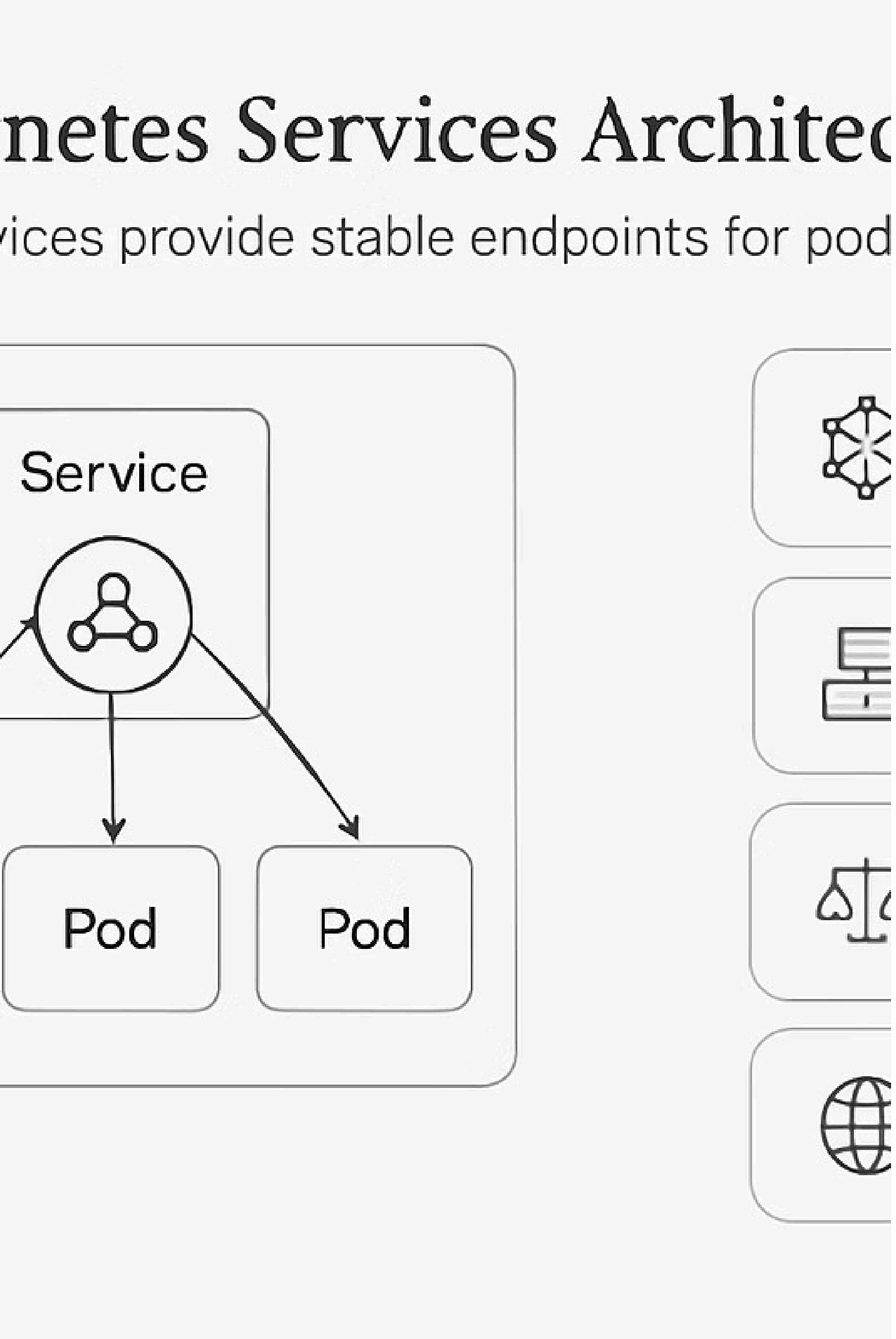
```
# Edit ReplicaSet
kubectl edit rs nginx-rs

# Note: Updating template only affects new pods,
# not existing ones
```

4 Delete with Options

```
# Delete ReplicaSet but keep pods (--cascade=false)
kubectl delete rs nginx-rs --cascade=false

# Delete ReplicaSet and all pods
kubectl delete rs nginx-rs
```



Understanding Kubernetes Services

Services provide stable network endpoints for pods, enabling discovery and load balancing regardless of pod lifecycles.

Service Types

ClusterIP

Default service type that exposes the service on an internal IP within the cluster.

- Only accessible within cluster
- Load balances across pod endpoints
- Ideal for internal communication

```
apiVersion: v1
kind: Service
metadata:
  name: my-service
spec:
  selector:
    app: MyApp
  ports:
  - port: 80
    targetPort: 9376
```

NodePort

Exposes the service on each node's IP at a static port.

- Builds on ClusterIP
- Accessible externally via :
- Port range: 30000-32767

```
apiVersion: v1
kind: Service
metadata:
  name: my-service
spec:
  type: NodePort
  selector:
    app: MyApp
  ports:
  - port: 80
    targetPort: 9376
    nodePort: 30007
```

LoadBalancer

Provisions an external load balancer in cloud providers.

- Builds on NodePort
- Integrates with cloud load balancers
- Provides single external IP

```
apiVersion: v1
kind: Service
metadata:
  name: my-service
spec:
  type: LoadBalancer
  selector:
  app: MyApp
  ports:
  - port: 80
  targetPort: 9376
```

Service Discovery Mechanisms

Environment Variables

Kubernetes automatically injects service information as environment variables into pods.

```
# For service named "redis-master" in port 6379
REDIS_MASTER_SERVICE_HOST=10.0.0.11
REDIS_MASTER_SERVICE_PORT=6379
REDIS_MASTER_PORT=tcp://10.0.0.11:6379
REDIS_MASTER_PORT_6379_TCP=tcp://10.0.0.11:6379
REDIS_MASTER_PORT_6379_TCP_PROTO=tcp
REDIS_MASTER_PORT_6379_TCP_PORT=6379
REDIS_MASTER_PORT_6379_TCP_ADDR=10.0.0.11
```

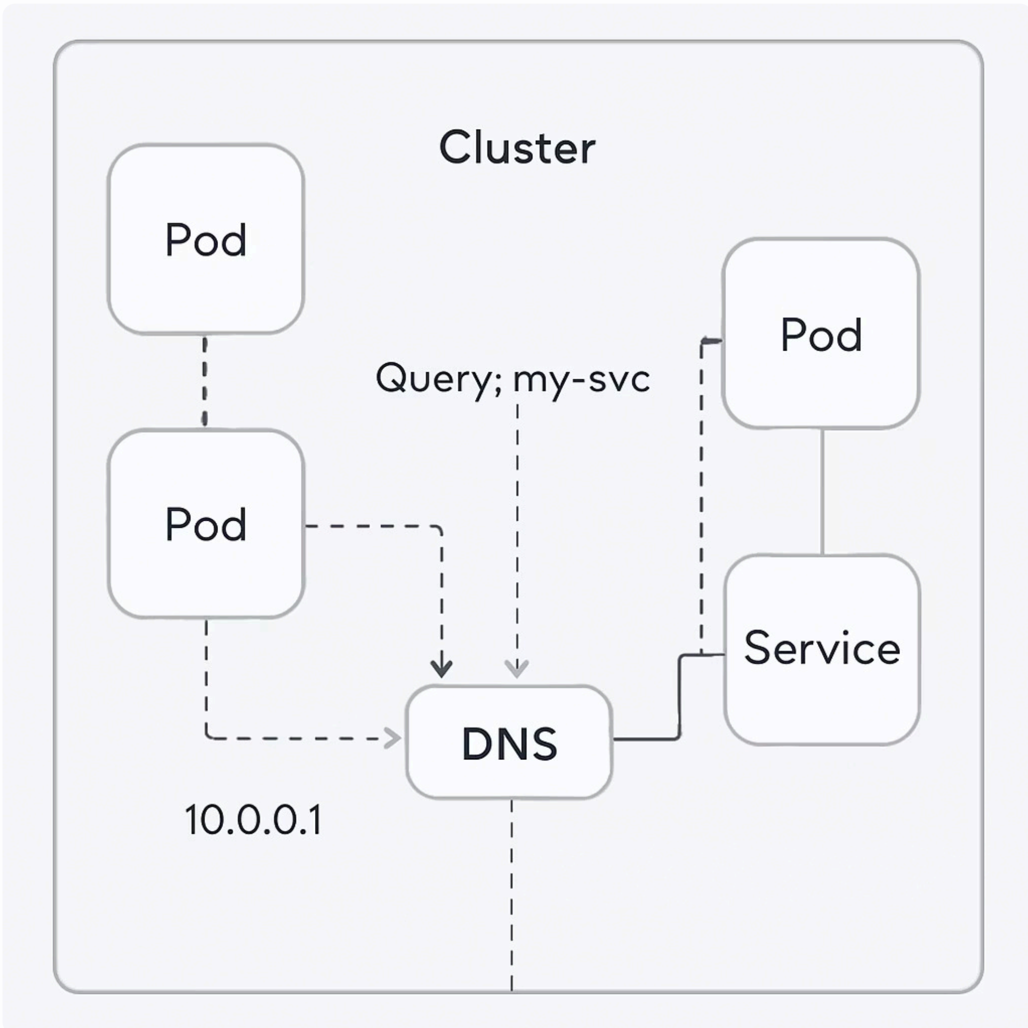
DNS Resolution

Kubernetes DNS automatically creates records for services.

```
# Format for services
..svc.cluster.local

# Examples
myservice.default.svc.cluster.local
redis-master.default.svc.cluster.local:6379

# Shortened forms in same namespace
myservice
myservice.default
```

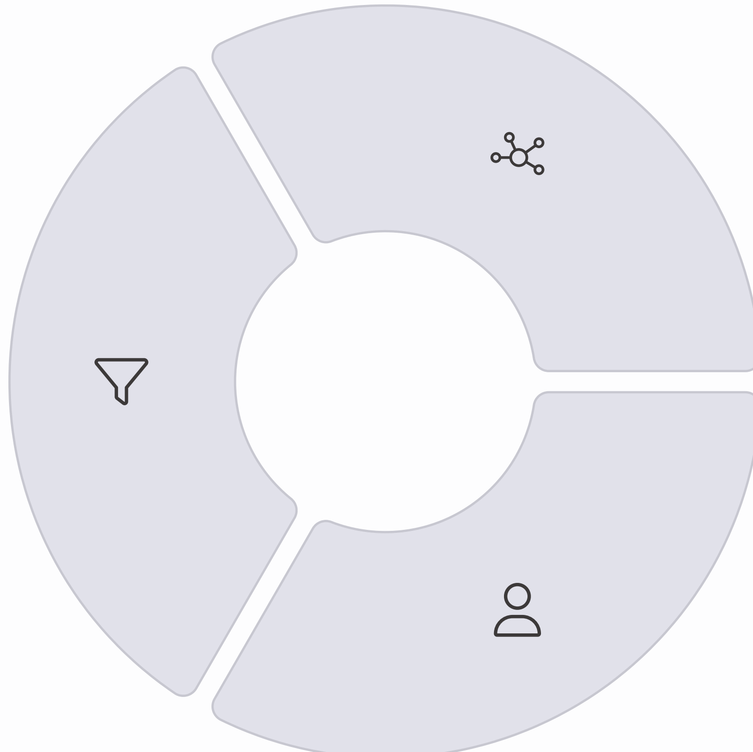


Service Implementation: kube-proxy Modes

iptables Mode

Default mode using iptables rules for traffic redirection.

- Lower overhead than userspace
- Random selection for load balancing
- Connection fails if selected pod is down



IPVS Mode

High-performance mode using Linux IPVS for traffic routing.

- Better performance for large clusters
- More load balancing algorithms
- Requires IPVS kernel modules

Userspace Mode

Legacy mode where kube-proxy acts as a userspace proxy.

- Higher latency
- Round-robin load balancing
- Automatic retry on connection failure

Service Configuration: Advanced Options

Session Affinity

```yaml
apiVersion: v1
kind: Service
metadata:
  name: my-service
spec:
  selector:
    app: MyApp
  sessionAffinity: ClientIP
  sessionAffinityConfig:
    clientIP:
      timeoutSeconds: 10800
  ports:
  - port: 80
    targetPort: 9376
```

Multiple Ports

```yaml
apiVersion: v1
kind: Service
metadata:
  name: my-service
spec:
  selector:
    app: MyApp
  ports:
  - name: http
    port: 80
    targetPort: 8080
  - name: https
    port: 443
    targetPort: 8443
```

External Traffic Policy

```yaml
apiVersion: v1
kind: Service
metadata:
  name: my-service
spec:
  selector:
    app: MyApp
  externalTrafficPolicy: Local
  type: LoadBalancer
  ports:
  - port: 80
    targetPort: 9376
```

Headless Services

```yaml
apiVersion: v1
kind: Service
metadata:
 name: my-service
spec:
 clusterIP: None
 selector:
 app: MyApp
 ports:
 - port: 80
 targetPort: 9376
```

ExternalName Services

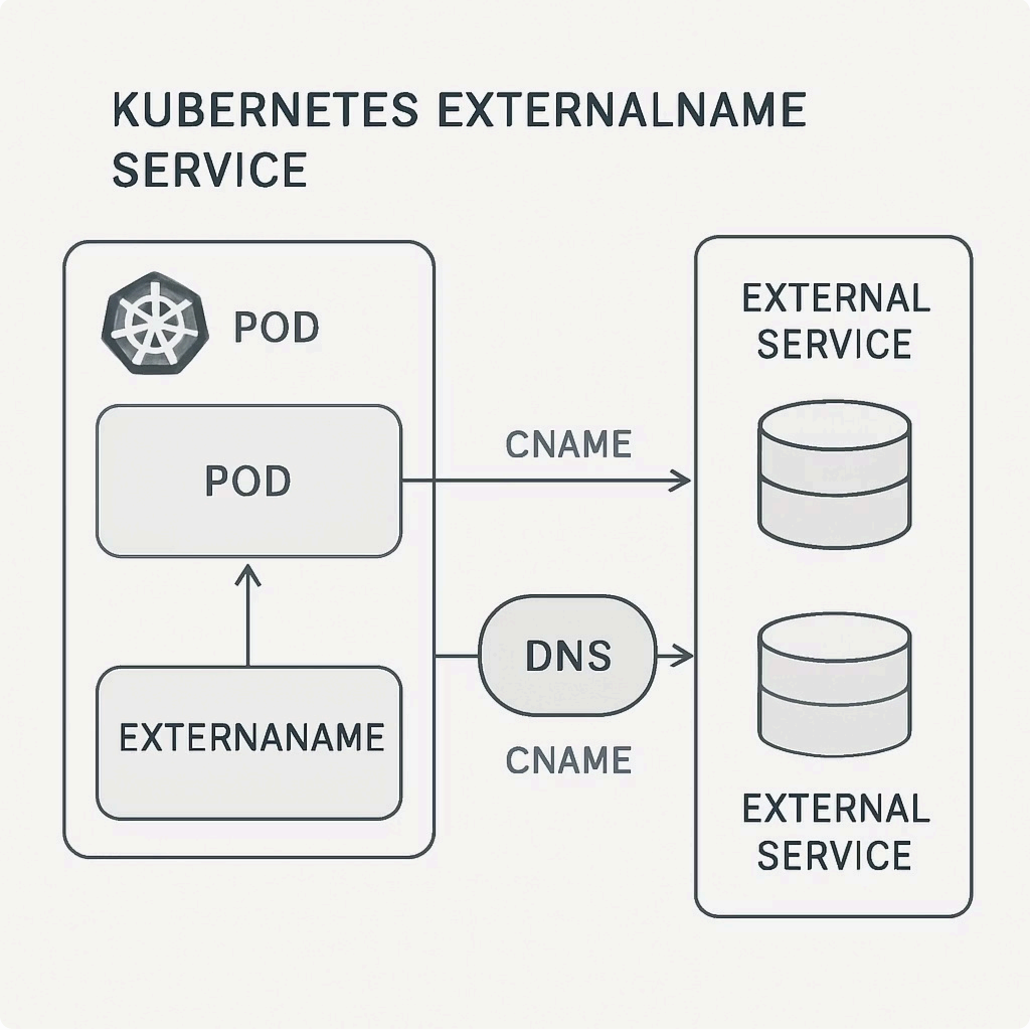
Service without Selectors

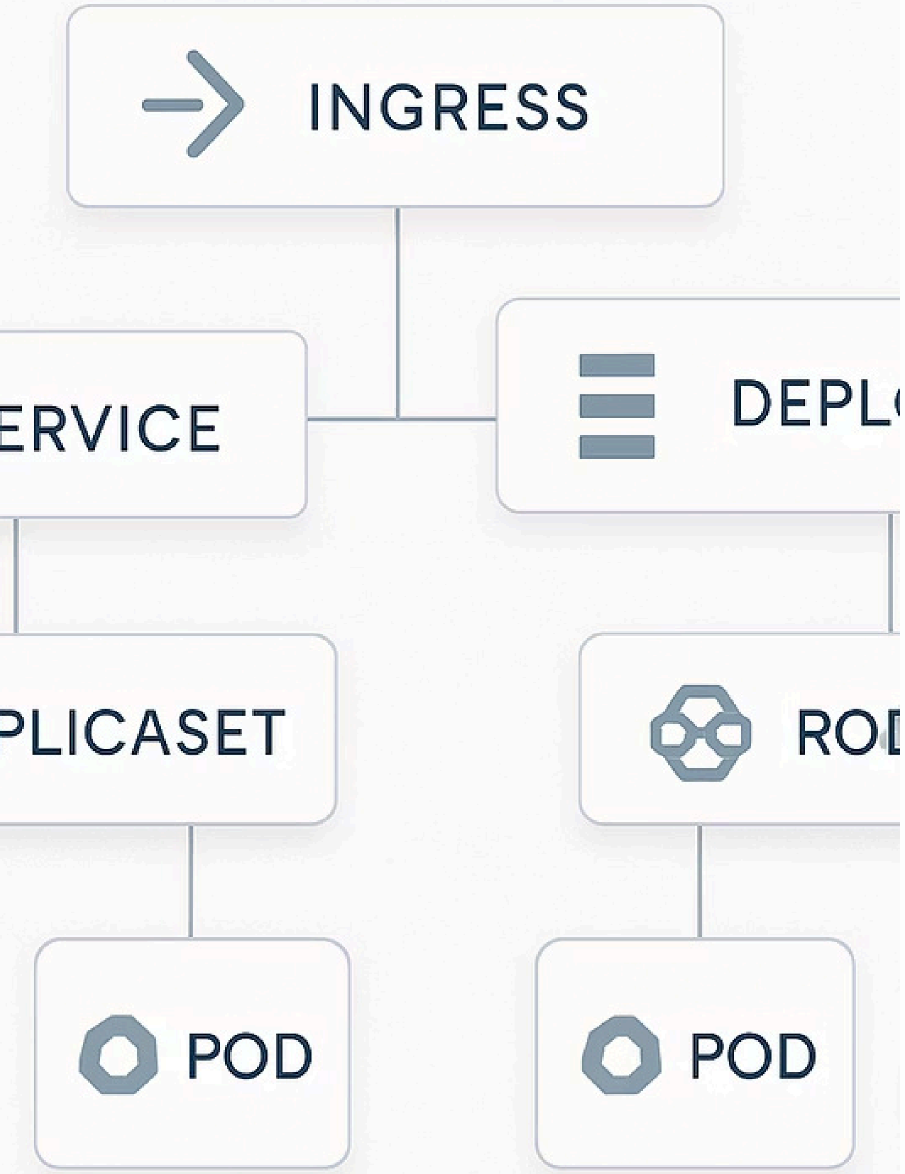
Maps a service to an external DNS name rather than to pods.

- Acts as a CNAME record in cluster DNS
- No proxying or load balancing
- Useful for integrating external services
- Provides abstraction for external dependencies

```
apiVersion: v1
kind: Service
metadata:
  name: my-database
  namespace: prod
spec:
  type: ExternalName
  externalName: database.example.com
```

Applications can connect to `my-database.prod.svc.cluster.local` which resolves to `database.example.com`.





Putting It All Together: Deployment Architecture

This diagram shows how all the components we've covered work together in a complete application deployment.

Kubernetes Resources: Best Practices

Resource Organization

- Use namespaces for logical separation
- Implement consistent labeling strategy
- Group related resources in the same manifest
- Use ConfigMaps and Secrets for configuration

Reliability Patterns

- Configure resource requests and limits
- Implement readiness and liveness probes
- Set appropriate pod disruption budgets
- Use anti-affinity for high availability

Security Considerations

- Apply principle of least privilege with RBAC
- Implement network policies for pod isolation
- Use security contexts to harden containers
- Scan images for vulnerabilities

Next Steps in Your Kubernetes Journey

Advanced Workloads

StatefulSets, DaemonSets, Jobs and CronJobs

Security Hardening

RBAC, NetworkPolicies, PodSecurityPolicies

Storage Solutions

PersistentVolumes, StorageClasses, CSI

Advanced Networking

Ingress Controllers, Service Mesh, NetworkPolicies

Monitoring & Logging

Prometheus, Grafana, ELK Stack, Loki

Thank you for attending this technical deep dive into Kubernetes. Apply these patterns in your environment to build resilient, scalable container infrastructure.