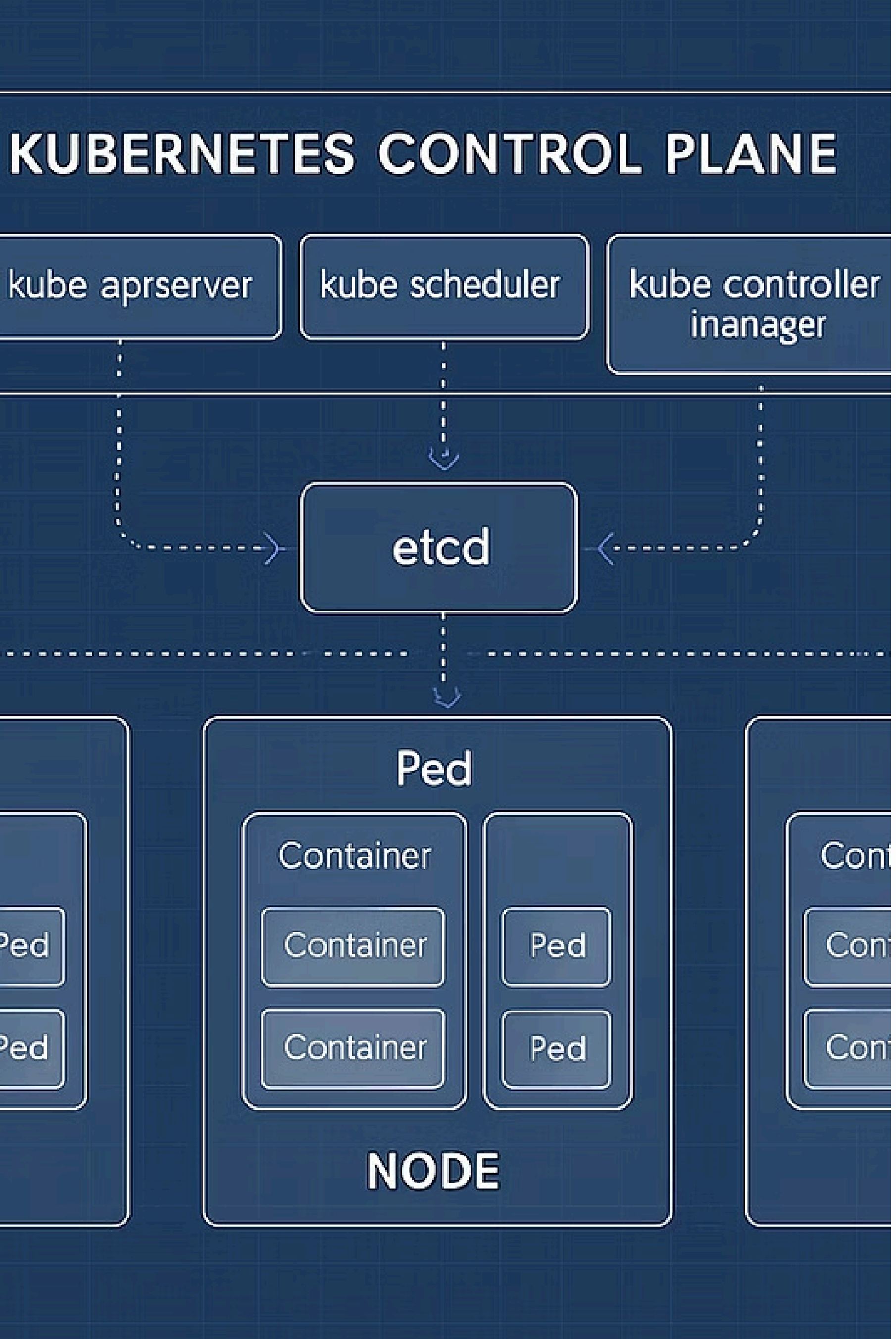


Advanced Kubernetes Administration: Scheduling, Lifecycle Management, and Configuration

A comprehensive technical guide for Kubernetes administrators

Agenda

Module 4: Scheduling

- Manual Scheduling
- Node Selector
- Taints and Tolerations

Module 5: Application Lifecycle

- Deployment Overview
- Deployment Strategies
- Practical Implementation

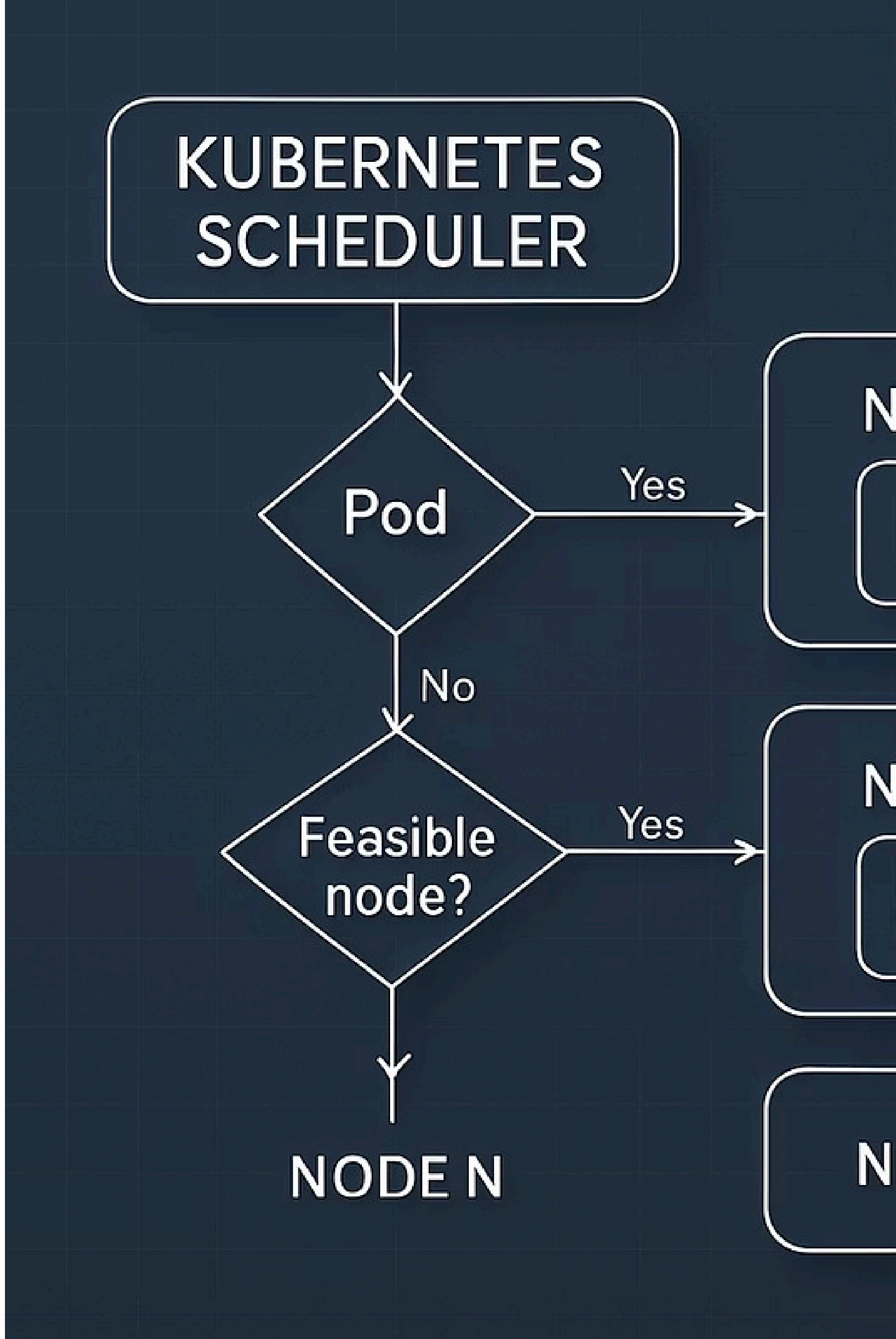
Module 6: Environment Variables

- Plain Key Environment Variables
- ConfigMaps
- Secrets
- Volumes & Environment Integration

Module 4: Scheduling

Pod Scheduling in Kubernetes

Understanding how pods are assigned to nodes and how administrators can control placement

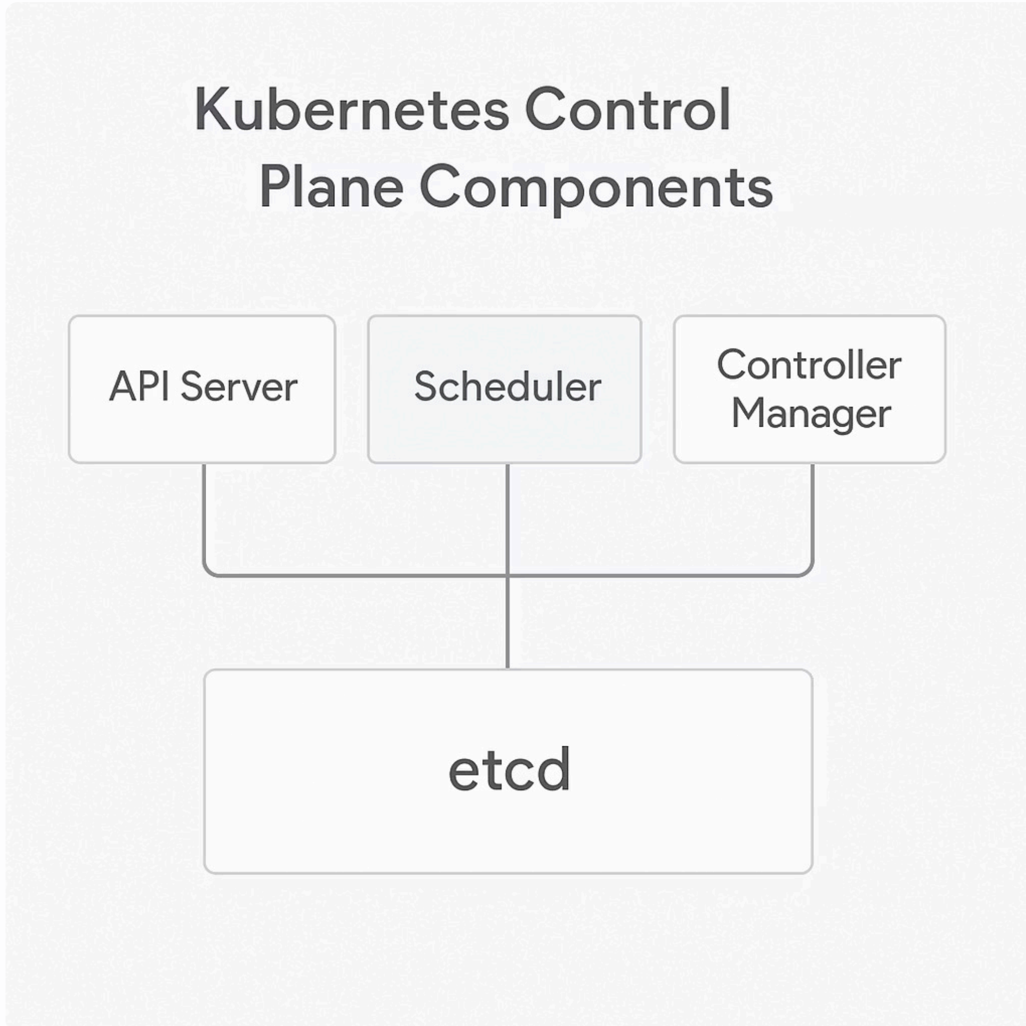


Manual Pod Scheduling

The Kubernetes Scheduler

The scheduler is responsible for deciding which worker node should run newly created pods.

When the scheduler is bypassed, administrators must manually specify the target node.



Manual Scheduling: Node Assignment

Manual Node Selection

To manually schedule a pod, set the **nodeName** field in the pod specification:

```
apiVersion: v1
kind: Pod
metadata:
  name: nginx-pod
spec:
  nodeName: worker-node-1  # Direct node assignment
  containers:
  - name: nginx
    image: nginx:1.19
    ports:
    - containerPort: 80
```

Considerations

- No health checks performed
- No resource availability checks
- Pod remains unscheduled if node is unavailable
- Cannot be changed after pod creation

Manual Scheduling: Node Selector

Using nodeSelector

More flexible than nodeName, selecting nodes based on labels:

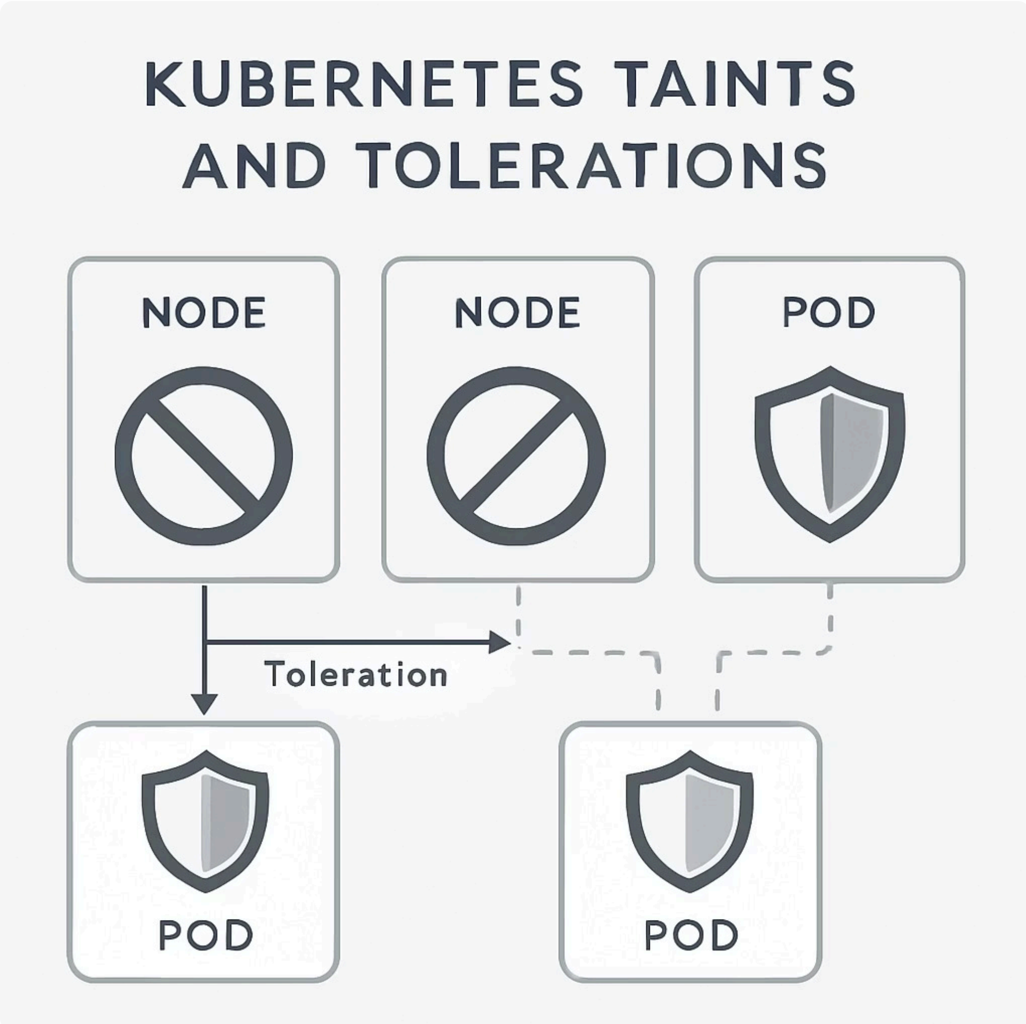
```
apiVersion: v1
kind: Pod
metadata:
  name: gpu-pod
spec:
  nodeSelector:
    gpu: "true"
    zone: "us-west"
  containers:
  - name: cuda-container
    image: nvidia/cuda:11.0
```

Labeling Nodes

```
# Add labels to nodes
kubectl label nodes worker-node-2 \
  gpu=true zone=us-west

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

The pod will only be scheduled on nodes matching **all** the specified labels.

Taints and Tolerations: Concept



Taints: Node Perspective

Taints are properties applied to nodes that repel certain pods.

Tolerations: Pod Perspective

Tolerations allow pods to schedule onto nodes with matching taints.

This mechanism enables nodes to control which pods should (or should not) be scheduled on them.

Taints: Implementation

Adding Taints to Nodes

```
# Syntax: kubectl taint nodes [node-name] [key]=[value]:[effect]

# Examples:
kubectl taint nodes worker-node-1 app=database:NoSchedule
kubectl taint nodes worker-node-2 dedicated=gpu:NoExecute
kubectl taint nodes worker-node-3 env=production:PreferNoSchedule
```

Taint Effects

NoSchedule

Pods will not be scheduled on the node unless they have a matching toleration.

PreferNoSchedule

The system will try to avoid placing pods without matching tolerations, but not guaranteed.

NoExecute

New pods without matching tolerations won't be scheduled AND existing pods without tolerations will be evicted.

Tolerations: Implementation

Adding Tolerations to Pods

```
apiVersion: v1
kind: Pod
metadata:
  name: database-pod
spec:
  tolerations:
  - key: "app"
    operator: "Equal"
    value: "database"
    effect: "NoSchedule"
  containers:
  - name: postgres
    image: postgres:13
```

Operators in Tolerations

- **Equal**: Matches when key/value are equal
- **Exists**: Matches when key exists (value not checked)

```
# Exists example (no value needed):
tolerations:
- key: "dedicated"
 operator: "Exists"
 effect: "NoSchedule"
```

Taints and Tolerations: Real-World Example

Common Use Cases

Dedicated Nodes

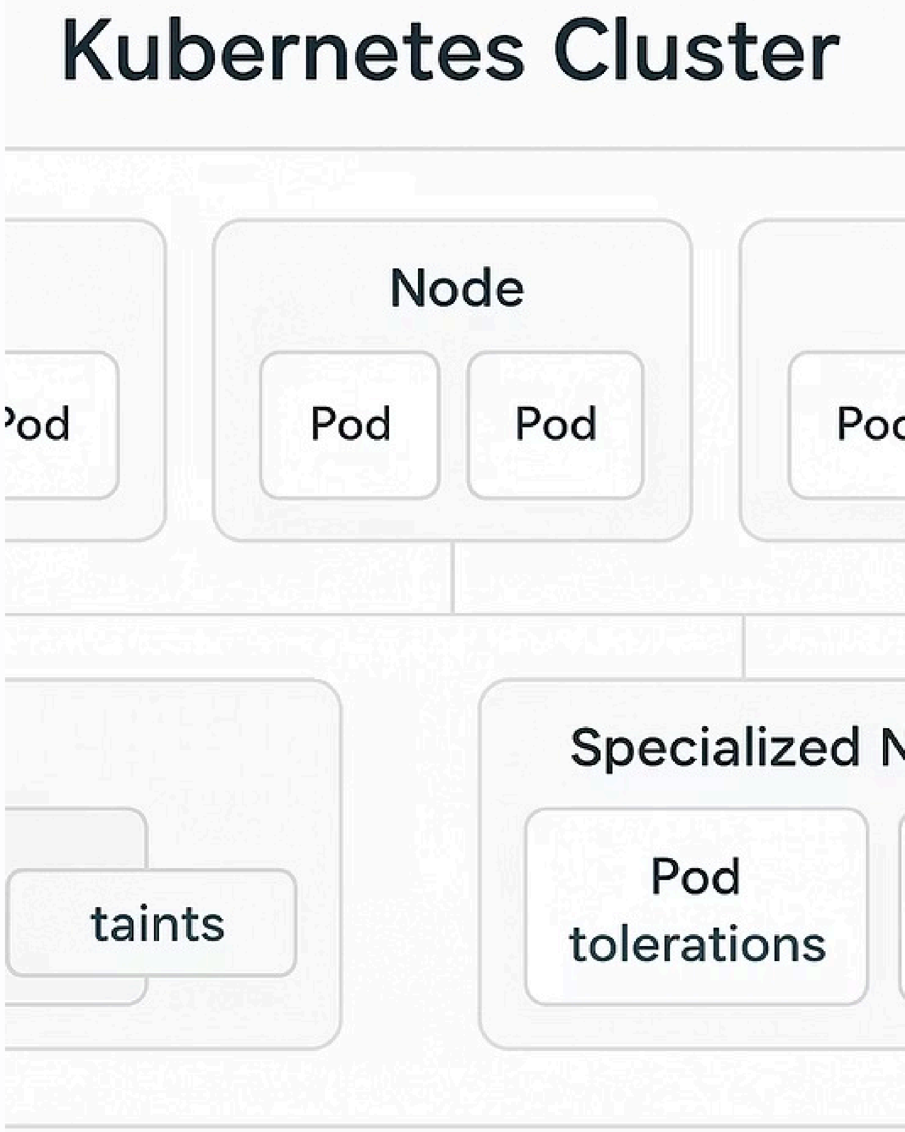
Reserve nodes for specific workloads (e.g., production-only or GPU-intensive applications)

Node Maintenance

Safely drain nodes by preventing new pods and gradually evicting existing ones

Special Hardware

Ensure only pods requiring special hardware (like GPUs) run on specialized nodes



Taint-Based Evictions

Node Conditions and Auto-Tainting

Kubernetes automatically adds taints to nodes with specific conditions:

- **node.kubernetes.io/not-ready**: Node is not ready
- **node.kubernetes.io/unreachable**: Node is unreachable from controller
- **node.kubernetes.io/memory-pressure**: Memory pressure on node
- **node.kubernetes.io/disk-pressure**: Disk pressure on node
- **node.kubernetes.io/network-unavailable**: Network unavailable
- **node.kubernetes.io/unschedulable**: Node is cordoned

Tolerating Node Problems

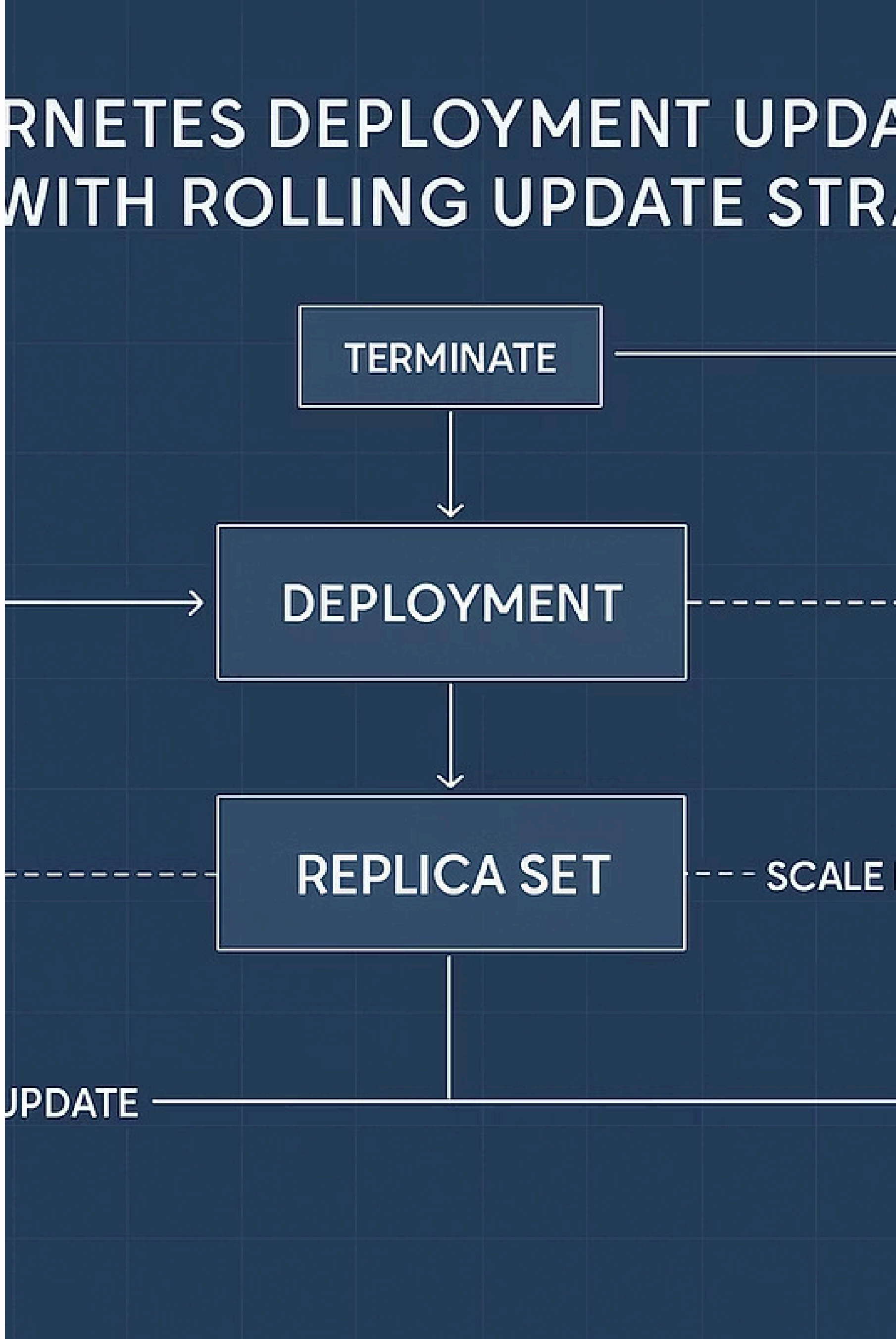
```
tolerations:
- key: "node.kubernetes.io/unreachable"
  operator: "Exists"
  effect: "NoExecute"
  tolerationSeconds: 300  # Evict after 5 minutes
```

The **tolerationSeconds** field defines how long a pod stays bound to a node with matching taint before being evicted.

Module 5: Application Lifecycle Management

Deployment Management

Managing application deployments, updates, and rollbacks in production Kubernetes environments

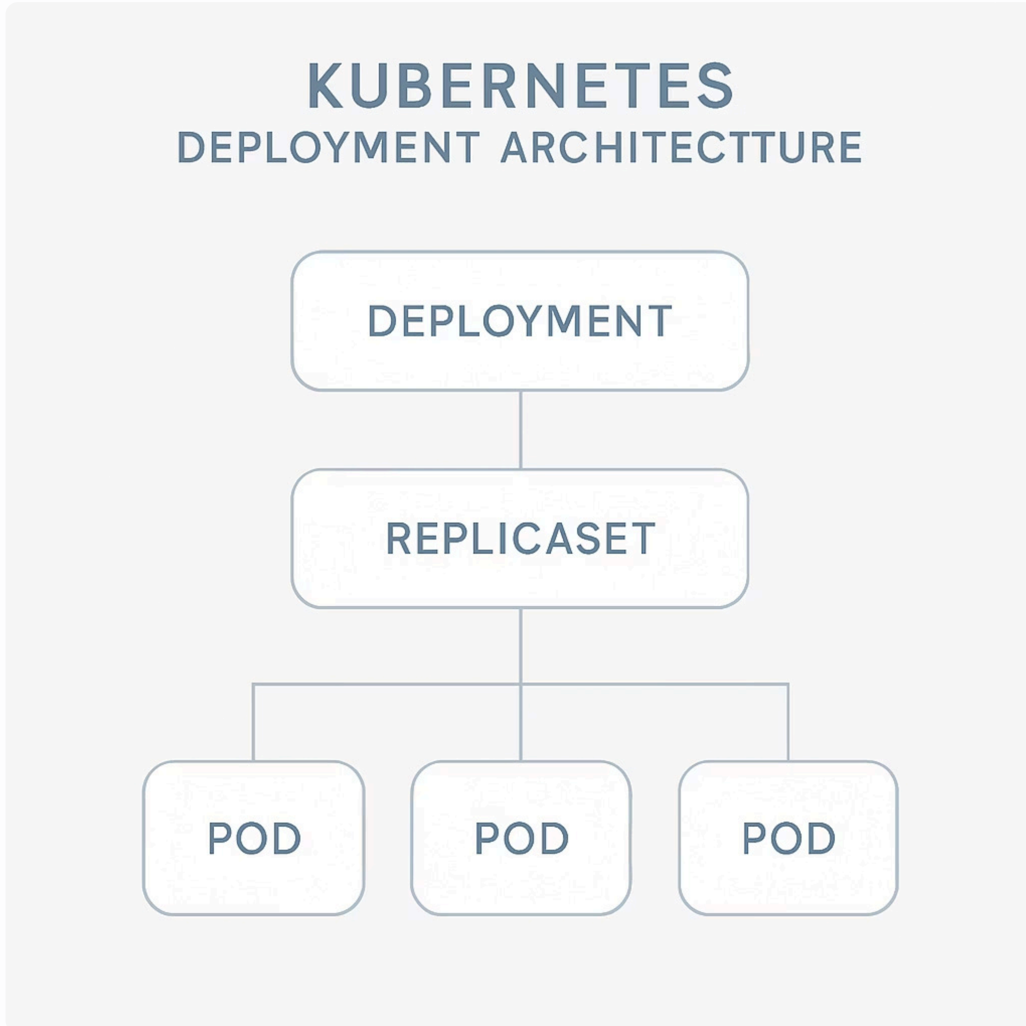


Deployment Overview

What is a Deployment?

A Deployment is a higher-level Kubernetes resource that:

- Manages ReplicaSets and Pods
- Provides declarative updates for applications
- Maintains deployment history for rollbacks
- Scales applications horizontally
- Ensures application availability during updates



Deployment Controller

How Deployments Work

The Deployment Controller continuously reconciles the actual state with the desired state by:

- Creating new ReplicaSets
- Scaling ReplicaSets up or down
- Monitoring rollout progress
- Cleaning up old ReplicaSets

Deployment Specification

```
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:1.19
 ports:
 - containerPort: 80
```

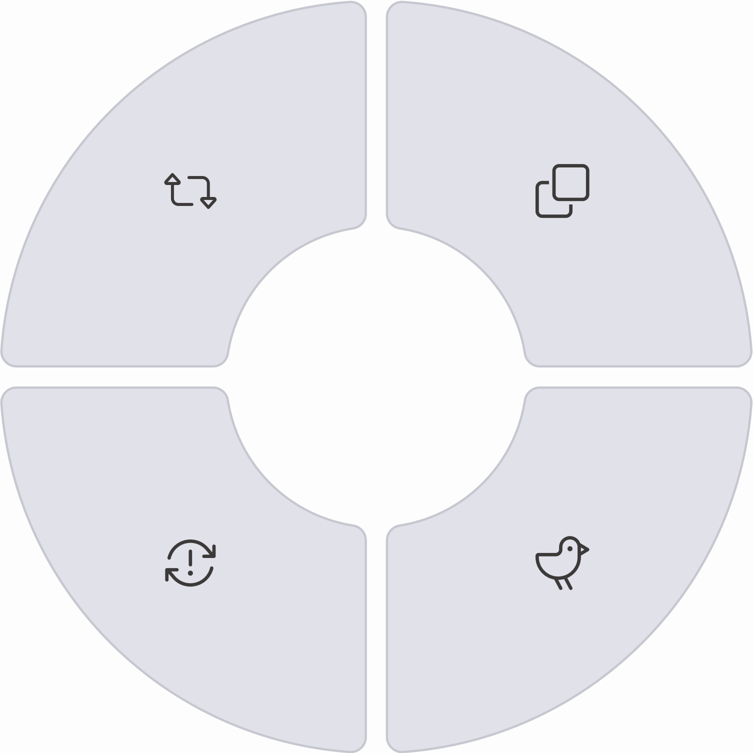
Deployment Strategies: Overview

Rolling Update

Default strategy that gradually replaces old pods with new ones

Blue/Green

Runs two identical environments, switching traffic when new version is ready



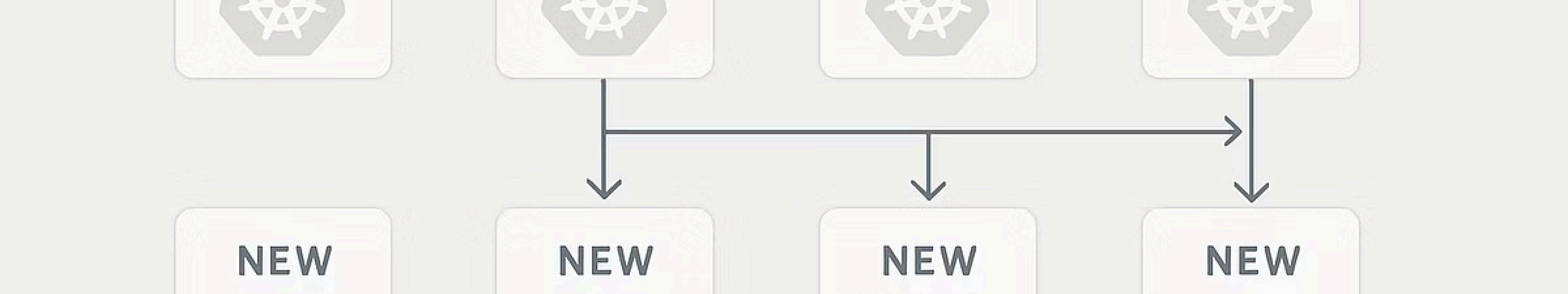
Recreate

Terminates all existing pods before creating new ones

Canary

Routes small percentage of traffic to new version for testing

Each strategy has different tradeoffs between availability, stability, and speed of updates.



Rolling Update Strategy

How Rolling Updates Work

1. New ReplicaSet is created for updated pods
2. New pods are gradually created in the new ReplicaSet
3. Old pods are gradually terminated from the old ReplicaSet
4. Process continues until all pods run the new version

Rolling Update: Configuration

Control Parameters

```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: frontend
spec:
  replicas: 5
  strategy:
    type: RollingUpdate
    rollingUpdate:
      maxSurge: 1        # Max pods above desired
      maxUnavailable: 1  # Max pods below desired
  # ... rest of deployment spec
```

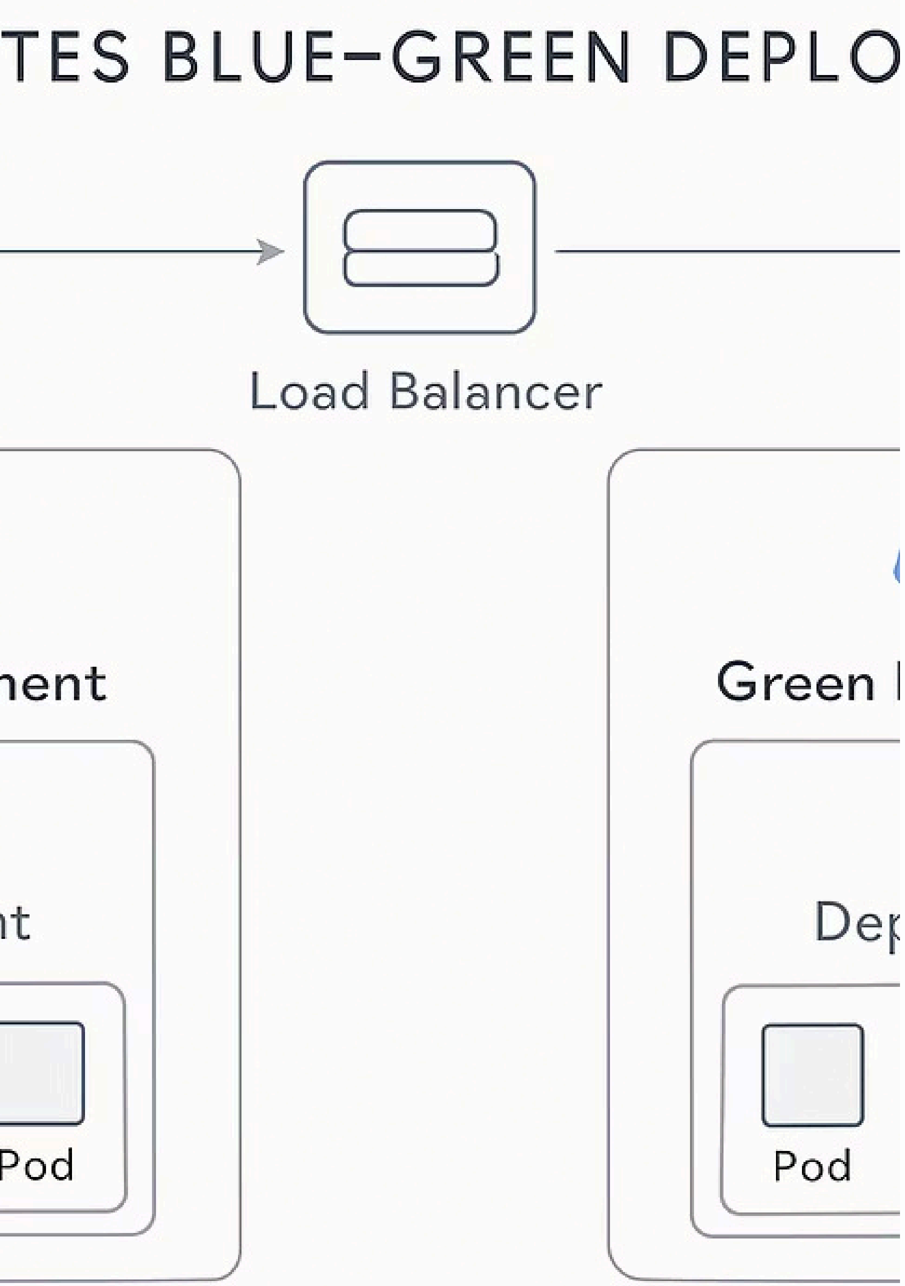
Impact of Parameters

maxSurge: How many pods can be created above the desired count

- Absolute number (e.g., 3) or percentage (e.g., 25%)
- Higher values = faster rollout but more resource usage

maxUnavailable: How many pods can be unavailable during update

- Absolute number or percentage
- Lower values = higher availability but slower rollout



Blue/Green Deployment Strategy

1. **Deploy "Green" Environment**

   Create a complete copy of the production environment with the new version

2. **Test "Green" Environment**

   Perform comprehensive testing on the new environment without affecting users

3. **Switch Traffic**

   Update service selector labels to direct traffic to the new environment

4. **Maintain "Blue" Environment**

   Keep old environment ready for immediate rollback if needed

Blue/Green: Implementation

1. Create Blue Deployment (Current Version)

```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: app-blue
  labels:
    app: myapp
    version: blue
spec:
  replicas: 3
  selector:
    matchLabels:
      app: myapp
      version: blue
  template:
    metadata:
      labels:
        app: myapp
        version: blue
    spec:
      containers:
      - name: myapp
        image: myapp:1.0
```

2. Create Service (Points to Blue)

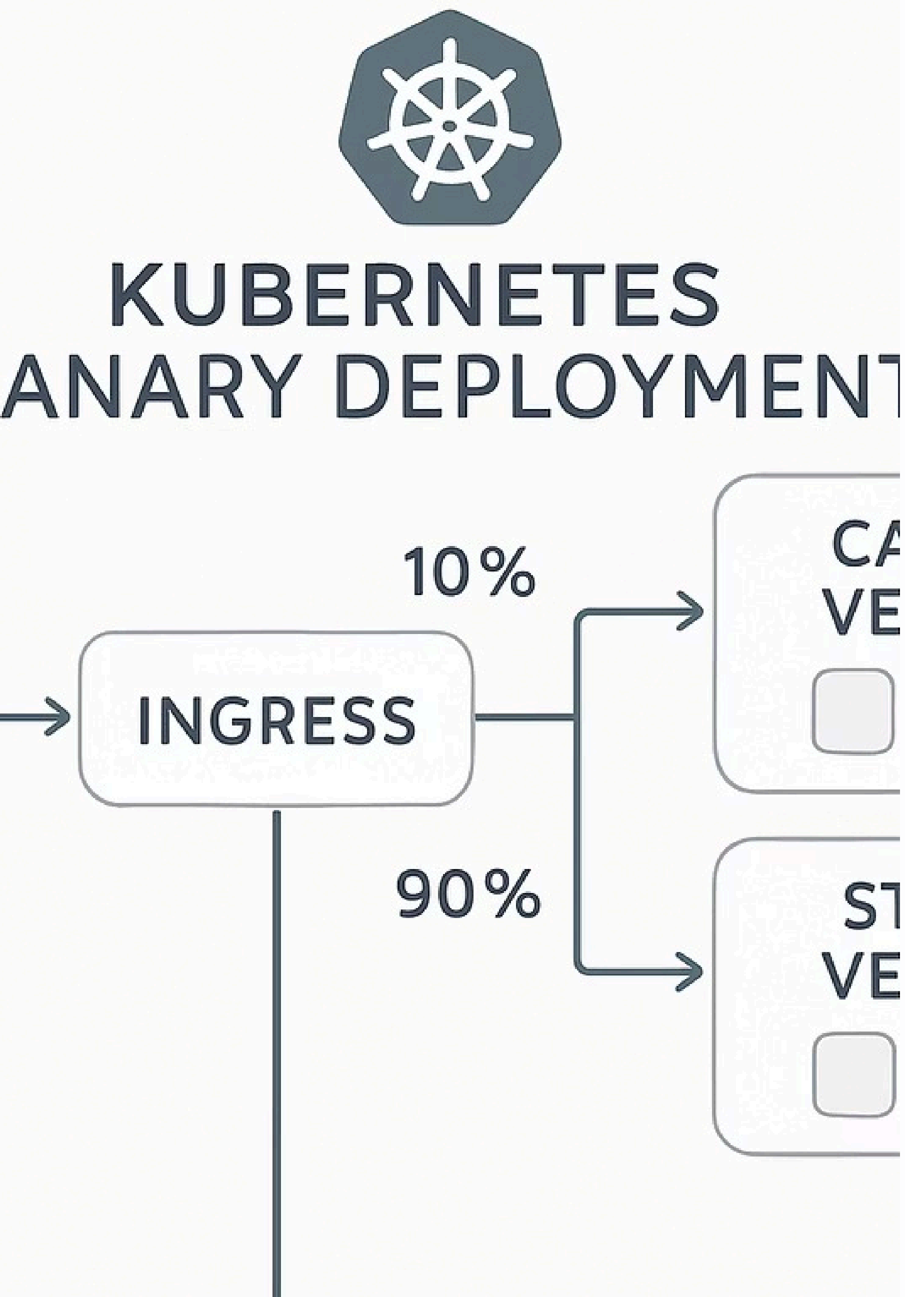
```yaml
apiVersion: v1
kind: Service
metadata:
  name: myapp-svc
spec:
  selector:
    app: myapp
    version: blue  # Points to blue version
  ports:
  - port: 80
    targetPort: 8080
```

3. Create Green Deployment (New Version)

Same as blue but with version: green and image: myapp:2.0

4. Switch Traffic to Green

Update service selector to version: green



Canary Deployment Strategy

How Canary Works

Named after the "canary in a coal mine" concept:

1. Deploy a small subset of new version pods
2. Route a percentage of traffic to the new version
3. Monitor for errors and performance issues
4. Gradually increase traffic to new version
5. Complete the rollout or rollback based on results

Benefits

- Early detection of issues in real production traffic
- Minimal user impact if problems occur
- Progressive confidence building
- Supports A/B testing scenarios

Ideal for high-traffic production systems where availability is critical.

Canary: Implementation

1. Stable Deployment

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: app-stable
spec:
  replicas: 9  # 90% of pods
  selector:
    matchLabels:
      app: myapp
      version: stable
  template:
    metadata:
      labels:
        app: myapp
        version: stable
    spec:
      containers:
      - name: myapp
        image: myapp:1.0
```

2. Canary Deployment

```
apiVersion: apps/v1
kind: Deployment
metadata:
 name: app-canary
spec:
 replicas: 1 # 10% of pods
 selector:
 matchLabels:
 app: myapp
 version: canary
 template:
 metadata:
 labels:
 app: myapp
 version: canary
 spec:
 containers:
 - name: myapp
 image: myapp:2.0
```

Canary: Service Configuration

3. Service (Matches Both Deployments)

```yaml
apiVersion: v1
kind: Service
metadata:
  name: myapp-svc
spec:
  selector:
    app: myapp  # Matches both stable and canary
  ports:
  - port: 80
    targetPort: 8080
```

The service sends traffic to all pods with the label app: myapp, which includes both versions.

Traffic Distribution

Traffic percentage is proportional to the number of pods:

- 9 stable pods (90% of traffic)
- 1 canary pod (10% of traffic)

To increase canary traffic, scale up the canary deployment and/or scale down the stable deployment.

For more precise traffic control, use service mesh solutions like Istio.

Deploying Applications as Deployments

Creating a Deployment

```
# Imperative command
kubectl create deployment nginx \
  --image=nginx:1.19 --replicas=3

# Or apply a YAML file
kubectl apply -f deployment.yaml
```

Common Operations

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

# Update image
kubectl set image deployment/nginx \
  nginx=nginx:1.20
```

Monitoring Deployments

```
# Get all deployments
kubectl get deployments

# Detailed deployment info
kubectl describe deployment nginx

# Check rollout status
kubectl rollout status deployment/nginx

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

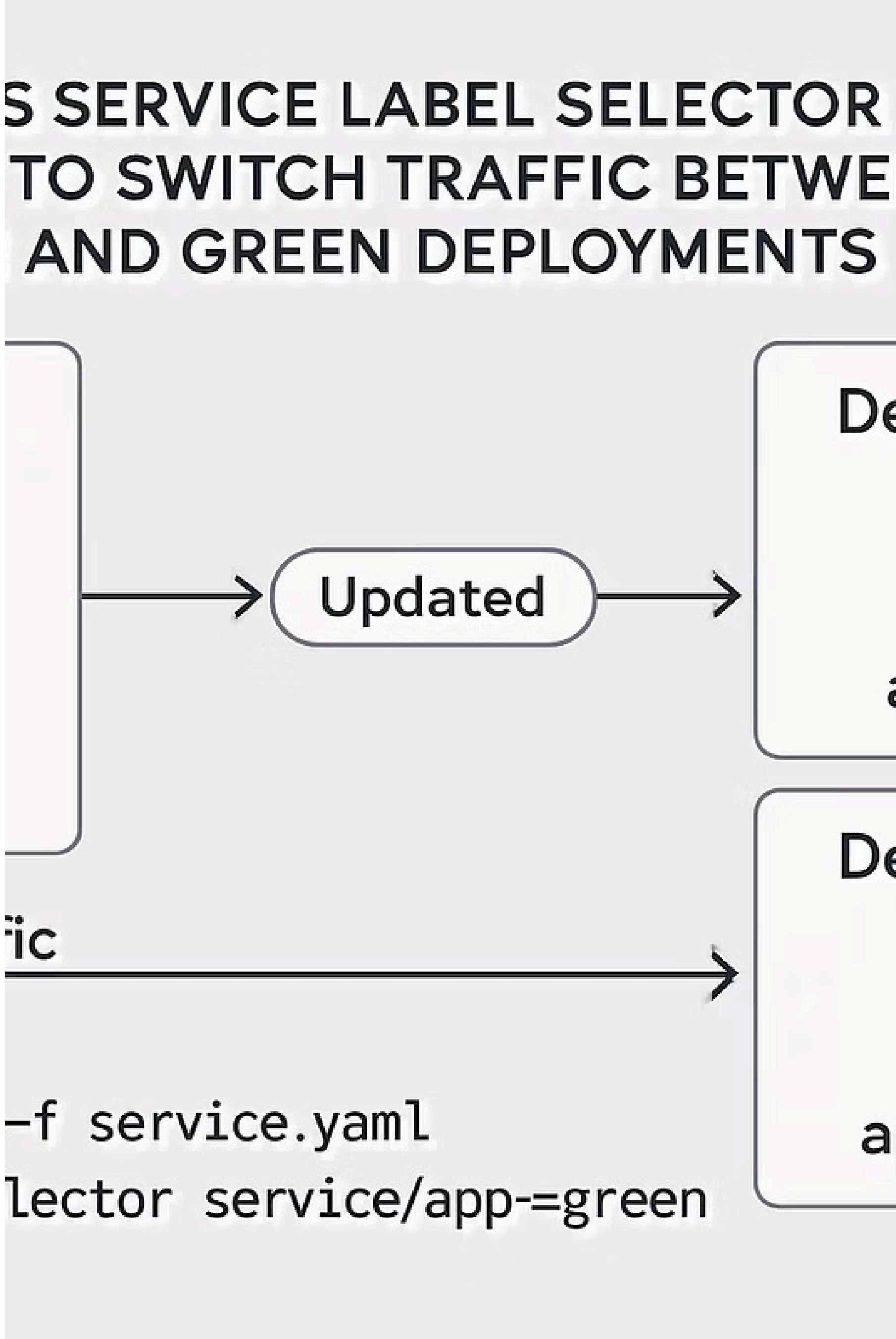
Managing Rollouts

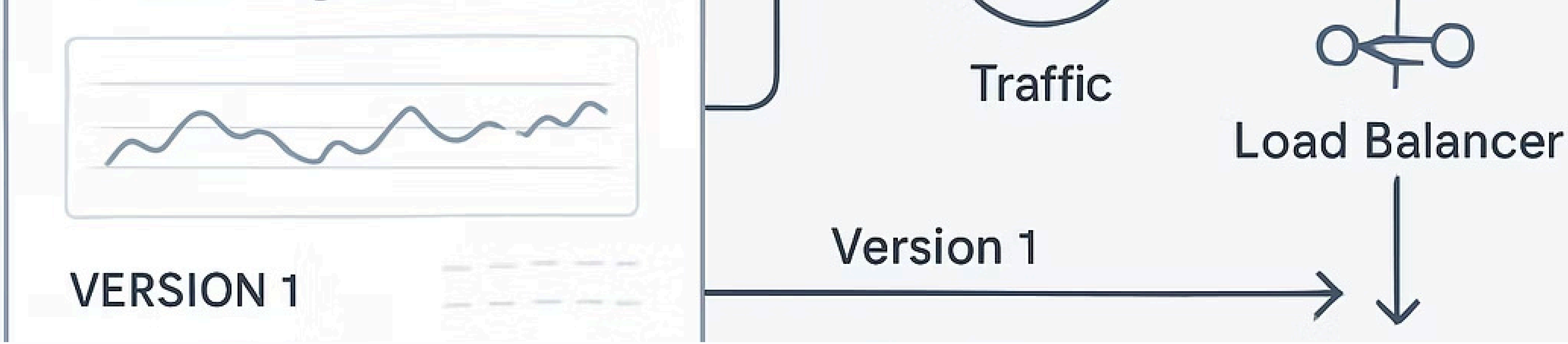
```
# Pause a rollout
kubectl rollout pause deployment/nginx

# Resume a rollout
kubectl rollout resume deployment/nginx

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

Implementing Blue/Green Deployments

Step-by-Step Process

1. Create "blue" deployment with the current application version
2. Create a service that selects the "blue" deployment using labels
3. Create "green" deployment with the new application version
4. Verify the "green" deployment functions correctly
5. Update the service selector to point to the "green" deployment
6. Once confirmed working, remove the "blue" deployment when ready





Implementing Canary Deployments

Implementation Approaches

Pod-based Canary

Use shared labels across deployments and adjust replica counts to control traffic percentage.

Service Mesh Canary

Use Istio or similar service mesh for precise traffic control based on HTTP headers, user IDs, etc.

Ingress-based Canary

Configure ingress controllers like NGINX or Traefik to split traffic between services.

Advanced Deployment Patterns

A/B Testing

Similar to canary but focuses on comparing features rather than testing stability:

- Deploy two versions simultaneously
- Route users to different versions based on criteria
- Collect metrics on user behavior
- Choose winning version based on data

Shadow Deployment

Production traffic is duplicated to the new version without affecting users:

- Deploy new version alongside current version
- Copy (mirror) real traffic to new version
- Monitor how new version handles real traffic
- No user impact as responses from new version are discarded

These patterns typically require service mesh implementations like Istio or Linkerd.

Deployment Best Practices

Health Checks

Implement readiness and liveness probes to ensure proper traffic routing and automatic recovery.

Handle State

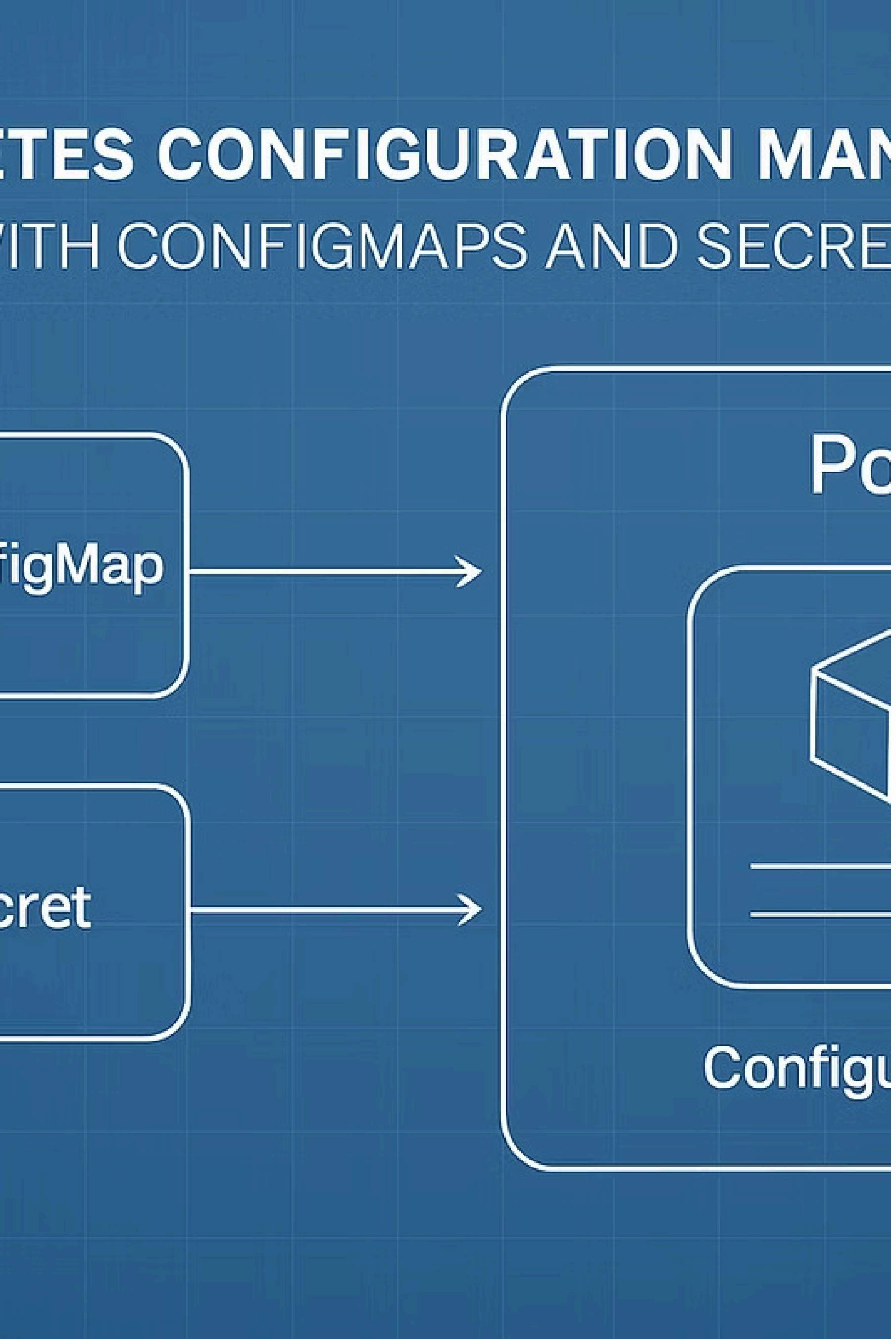
Ensure deployments handle database migrations and state transitions gracefully between versions.

Monitoring

Set up comprehensive monitoring for both old and new versions during the deployment process.

Version Control

Keep all deployment configurations in version control and implement GitOps workflows.



Module 6: Environment Variables

Configuration Management

Configuring applications in Kubernetes using environment variables and external configuration

Plain Key Environment Variables

Direct Definition in Pod Spec

```yaml
apiVersion: v1
kind: Pod
metadata:
  name: env-demo
spec:
  containers:
  - name: env-demo-container
    image: nginx
    env:
    - name: ENVIRONMENT
      value: "production"
    - name: LOG_LEVEL
      value: "INFO"
    - name: MAX_CONNECTIONS
      value: "100"
```

Usage Considerations

- Values are stored directly in the pod definition
- Values are always strings (quotes optional)
- Numbers and booleans are converted to strings
- Changes require pod recreation
- Not suitable for secrets or shared configurations
- Cannot be updated centrally

Environment Variables: ValueFrom

Dynamic Sources

Environment variables can be populated from various sources:

- ConfigMaps
- Secrets
- Pod fields
- Container resources

Pod Field References

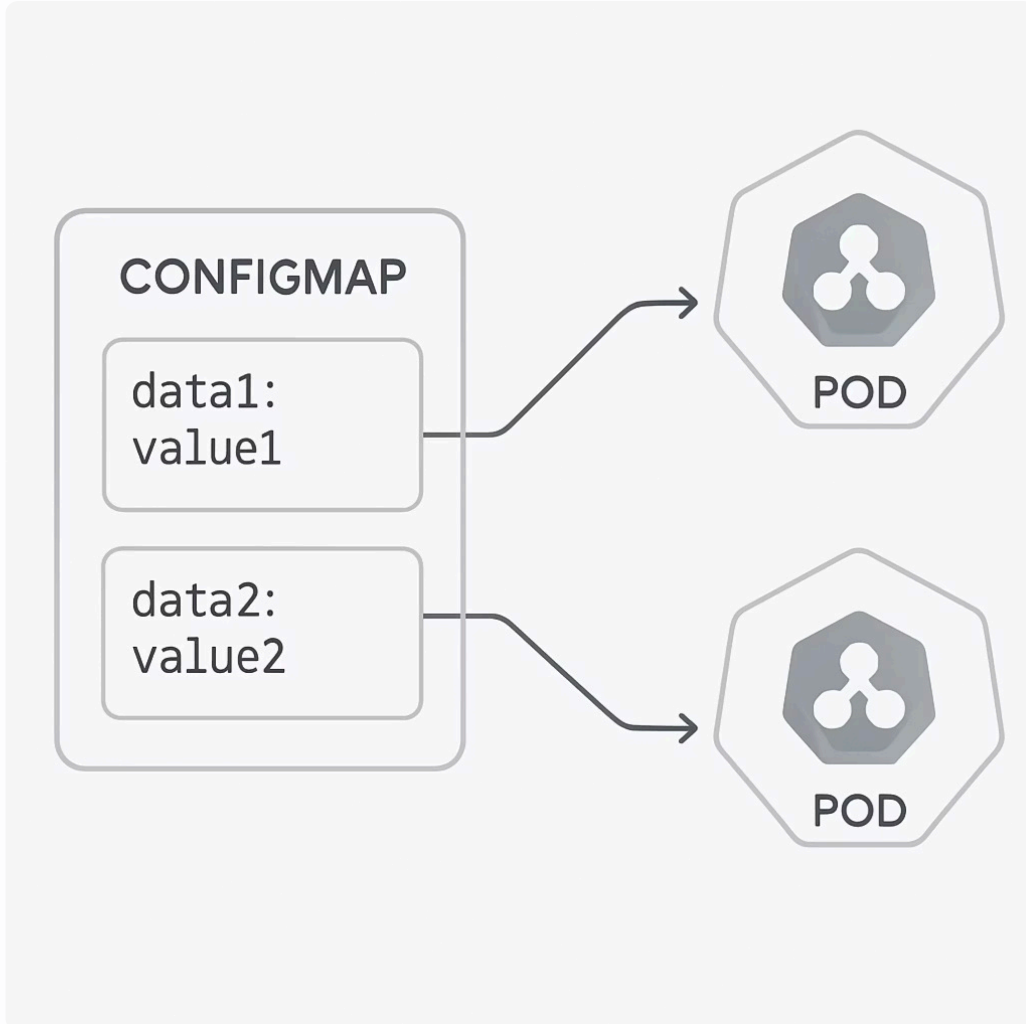
```
env:
- name: POD_NAME
 valueFrom:
 fieldRef:
 fieldPath: metadata.name
- name: POD_IP
 valueFrom:
 fieldRef:
 fieldPath: status.podIP
- name: NODE_NAME
 valueFrom:
 fieldRef:
 fieldPath: spec.nodeName
```

ConfigMaps: Overview

What are ConfigMaps?

ConfigMaps decouple configuration from pod specifications:

- Store non-confidential configuration data
- Key-value pairs or configuration files
- Can be updated without rebuilding container images
- Reusable across multiple pods
- Can be modified independently of pods



Creating ConfigMaps

Imperative Creation

```
# From literal values
kubectl create configmap app-config \
  --from-literal=DB_HOST=mysql \
  --from-literal=DB_PORT=3306

# From file
kubectl create configmap nginx-conf \
  --from-file=nginx.conf

# From env file
kubectl create configmap env-config \
  --from-env-file=config.env
```

Declarative Creation

```
apiVersion: v1
kind: ConfigMap
metadata:
 name: app-config
data:
 # Simple key-value pairs
 DB_HOST: "mysql"
 DB_PORT: "3306"

 # Configuration files as multi-line strings
 app.properties: |
 service.name=MyApp
 service.type=ClusterIP
 max.retry.count=5
```

Using ConfigMaps: Environment Variables

Single Environment Variables

```
env:
- name: DATABASE_HOST
  valueFrom:
    configMapKeyRef:
      name: app-config
      key: DB_HOST
- name: DATABASE_PORT
  valueFrom:
    configMapKeyRef:
      name: app-config
      key: DB_PORT
```

All ConfigMap Keys as Environment Variables

```
# Import all keys from a ConfigMap
envFrom:
- configMapRef:
    name: app-config
```

Each key in the ConfigMap becomes an environment variable in the container.

Keys that are invalid environment variable names (containing special characters) are skipped.

Secrets: Overview

What are Secrets?

Secrets store and manage sensitive information:

- Passwords, tokens, keys
- Similar to ConfigMaps but for sensitive data
- Base64 encoded (not encrypted by default)
- Separate resource type for access control
- Can be mounted as files or environment variables

Creating and Using Secrets

Creating Secrets

```
# Imperative creation
kubectl create secret generic db-creds \
  --from-literal=username=admin \
  --from-literal=password=s3cr3t

# Declarative YAML (values must be base64 encoded)
apiVersion: v1
kind: Secret
metadata:
  name: db-creds
type: Opaque
data:
  username: YWRtaW4=  # base64 encoded "admin"
  password: czNjcjN0  # base64 encoded "s3cr3t"
```

Using Secrets as Environment Variables

```
env:
- name: DB_USERNAME
 valueFrom:
 secretKeyRef:
 name: db-creds
 key: username
- name: DB_PASSWORD
 valueFrom:
 secretKeyRef:
 key: password
 name: db-creds

# Or all keys at once
envFrom:
- secretRef:
 name: db-creds
```

Secret Types

Opaque (generic)

Default type for arbitrary user-defined data

kubernetes.io/service-account-token

Service account credentials

kubernetes.io/dockerconfigjson

Docker registry credentials (.dockerconfigjson)

kubernetes.io/tls

TLS certificates (tls.crt and tls.key)

kubernetes.io/ssh-auth

SSH credentials

kubernetes.io/basic-auth

Basic authentication credentials (username and password)

Typed secrets have validation and specific expected keys.

Using Environment Variables and Volumes

Environment Variables Limitations

- Cannot be updated without pod restart
- Visible in process list and logs
- Entire value must fit in environment variable
- Not suitable for large configurations

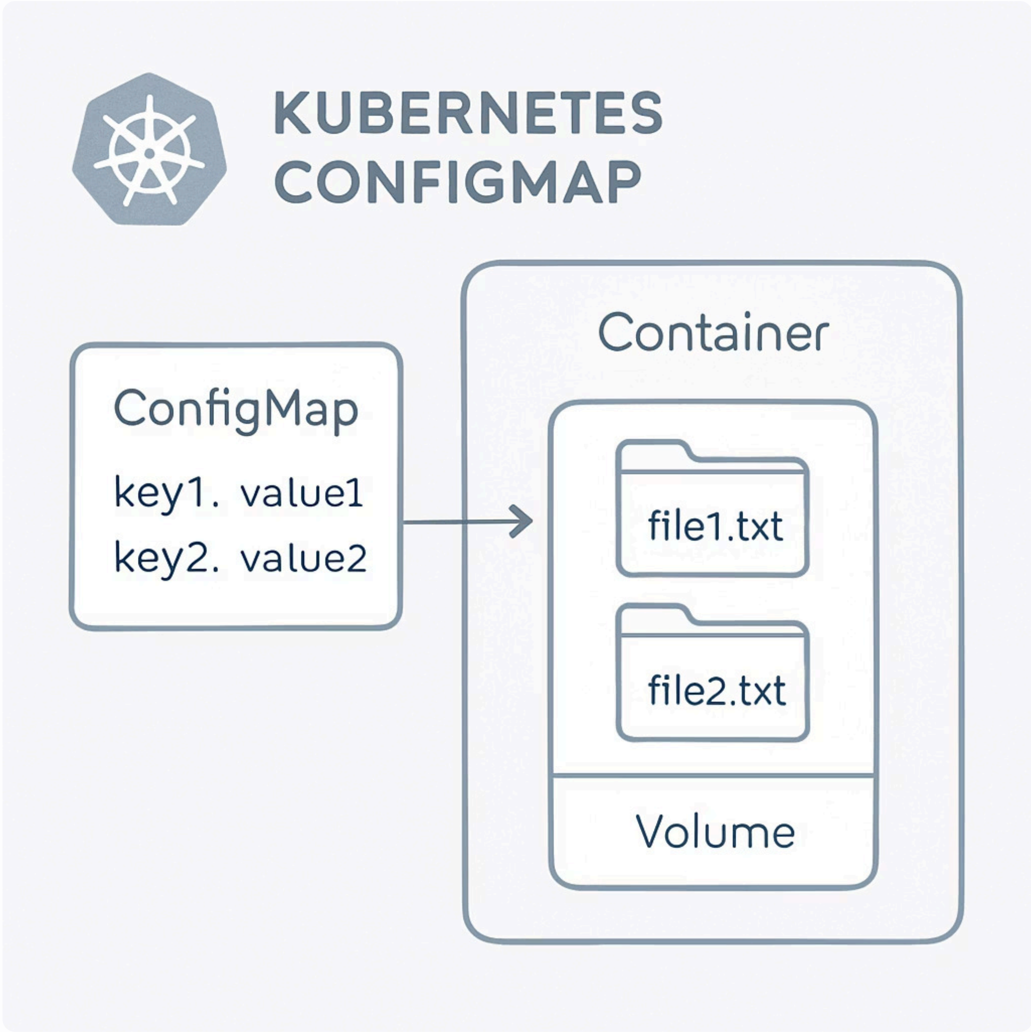
Volume Mounts for Configuration

- ConfigMaps and Secrets can be mounted as volumes
- Each key becomes a file in the mounted directory
- Updates to ConfigMaps/Secrets propagate to volumes
- Better for larger configurations
- More secure for sensitive data

ConfigMap Volume Mounts

Mounting ConfigMaps as Volumes

```yaml
apiVersion: v1
kind: Pod
metadata:
  name: config-volume-pod
spec:
  containers:
  - name: app
    image: nginx
    volumeMounts:
    - name: config-volume
      mountPath: /etc/config
  volumes:
  - name: config-volume
    configMap:
      name: app-config
      # Optional: specify items to mount
      items:
      - key: nginx.conf
        path: nginx.conf
      - key: app.properties
        path: application.properties
```



Each key from the ConfigMap becomes a file in the mounted directory with the value as its content.

Secret Volume Mounts

Mounting Secrets as Volumes

```
apiVersion: v1
kind: Pod
metadata:
  name: secret-volume-pod
spec:
  containers:
  - name: app
    image: nginx
    volumeMounts:
    - name: secret-volume
      mountPath: /etc/secrets
      readOnly: true  # Best practice
  volumes:
  - name: secret-volume
    secret:
      secretName: db-creds
      # Optional: file permissions
      defaultMode: 0400  # Read-only by owner
```

Security Considerations

- Always mount secrets as read-only
- Set restrictive file permissions (defaultMode)
- Consider using specific paths rather than entire secrets
- For production, use external secret management systems:
  - HashiCorp Vault
  - AWS Secrets Manager
  - Azure Key Vault
  - GCP Secret Manager

Key Takeaways

Scheduling

Control pod placement with manual scheduling, node selectors, and taints/tolerations

Deployment Strategies

Choose appropriate strategies (rolling, blue/green, canary) based on application requirements

Configuration Management

Separate configuration from application code using ConfigMaps and Secrets

Best Practices

Apply proper monitoring, health checks, and security controls for production-grade Kubernetes