

# Kubernetes Administration: Storage & Security Deep Dive

A comprehensive technical guide for Kubernetes administrators focused on volume management and security implementation.



# Agenda

## Part I: Storage Management

- Volume management principles
- Volume types and use cases
- Provisioning strategies
- Persistent Volumes architecture
- PVC workflow and implementation
- Storage Classes configuration

## Part II: Security Configuration

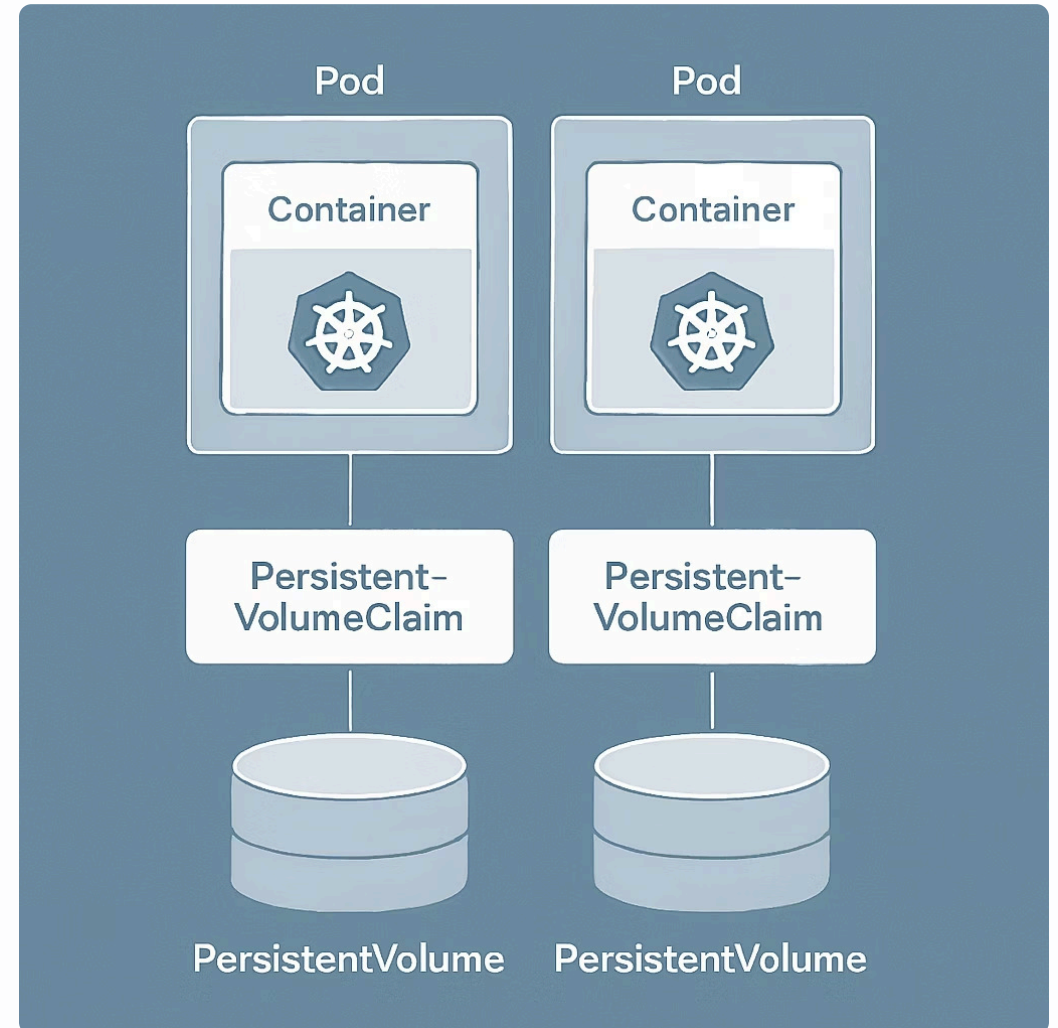
- Authentication mechanisms
- Authorization with RBAC
- Role-based access control implementation
- Security Context configuration
- Best practices and recommendations

# Kubernetes Storage: Core Concepts

Kubernetes storage abstraction enables containerized applications to access data persistently, beyond the container lifecycle.

Key objectives of Kubernetes storage:

- Data persistence across pod restarts
- Sharing data between containers
- Abstracting underlying storage implementations
- Automatic provisioning of storage resources



# Understanding Volume Management in K8s

Volumes in Kubernetes solve two critical problems for containerized applications:



## Container Ephemeral Storage

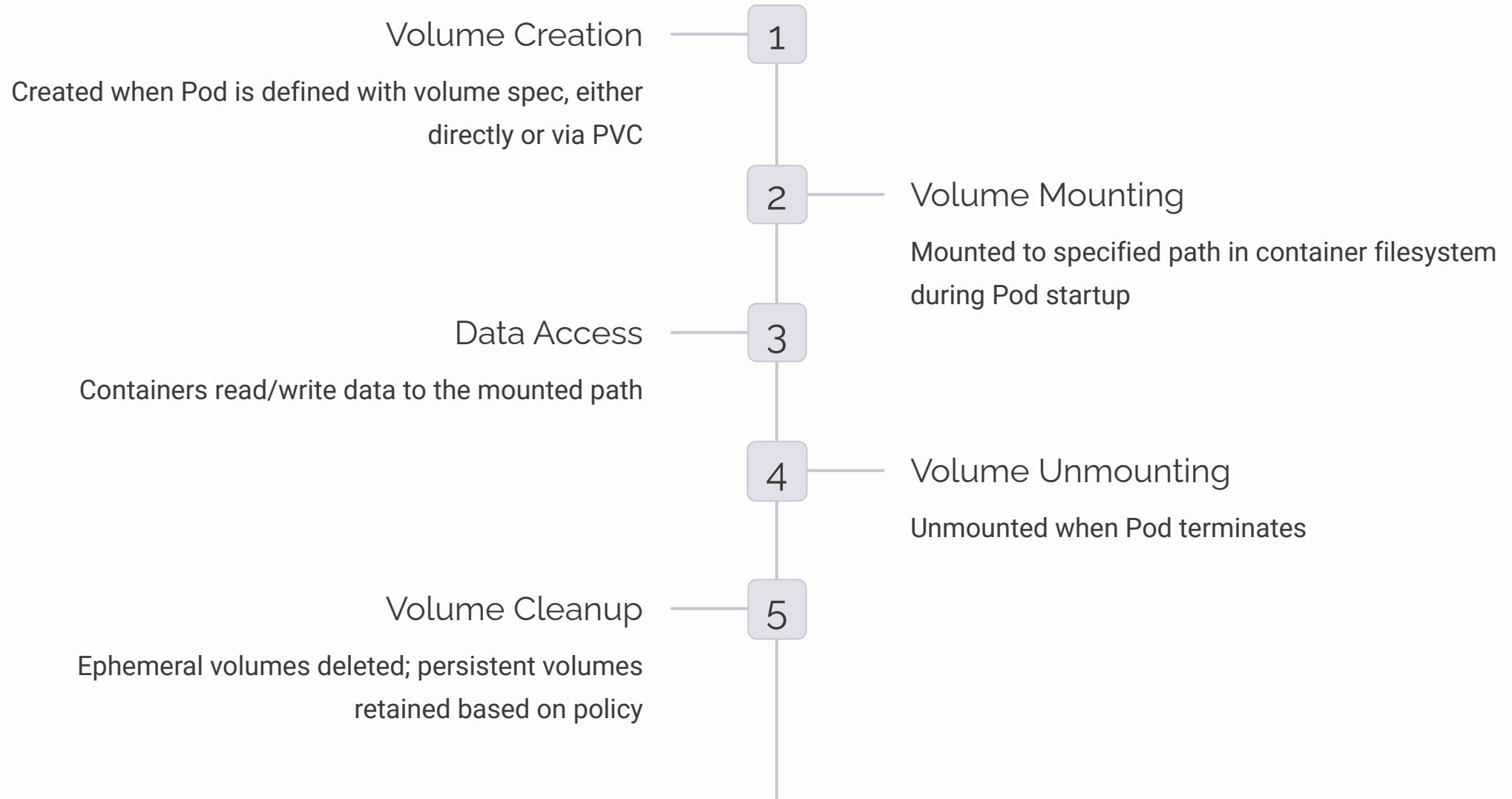
Files in a container are ephemeral, meaning when a container crashes or is terminated, the data is lost. Kubernetes restarts the container but with a clean state.



## Data Sharing Between Containers

Containers running in the same Pod often need to share files. Kubernetes volumes provide a file system that can be mounted by all containers in a Pod.

# Volume Lifecycle



# Types of Volumes

1

## Ephemeral Volumes

- emptyDir: Empty directory created when Pod is assigned to a node
- configMap: Inject configuration data into Pods
- secret: Inject sensitive data into Pods
- downwardAPI: Expose Pod and container fields to containers

2

## Persistent Volumes

- hostPath: Mounts file or directory from the host node's filesystem
- nfs: NFS share mounted into Pod
- csi: Container Storage Interface drivers for third-party storage
- awsElasticBlockStore, azureDisk, gcePersistentDisk: Cloud provider volumes

# emptyDir Volume Type

An emptyDir volume is first created when a Pod is assigned to a node, and exists as long as that Pod is running on that node.

Ideal for:

- Scratch space (e.g., for disk-based merge sorts)
- Checkpoint long computation for crash recovery
- Hold files retrieved from one container for use by another container

```
apiVersion: v1
kind: Pod
metadata:
  name: shared-logs
spec:
  containers:
    - name: log-producer
      image: busybox
      volumeMounts:
        - name: shared-volume
          mountPath: /logs
    - name: log-consumer
      image: nginx
      volumeMounts:
        - name: shared-volume
          mountPath: /var/log/nginx
  volumes:
    - name: shared-volume
      emptyDir: {}
```

# Cloud Provider Volume Types

AWS EBS  
(awsElasticBlockStore)

Mounts an Amazon Web Services (AWS) EBS Volume into a pod. Requires the pod to run on an AWS EC2 instance in the same region and availability zone as the EBS volume.

Azure Disk (azureDisk)

Mounts a Microsoft Azure Data Disk into a pod. Requires the pod to run on an Azure Virtual Machine in the same region as the disk.

Google Persistent Disk (gcePersistentDisk)

Mounts a Google Compute Engine (GCE) Persistent Disk into a pod. Requires the pod to run on a GCE VM in the same project and zone as the persistent disk.



Azure



**KUBERNETES**



# Storage Provisioning Methods

## Static Provisioning

Cluster administrator manually creates PersistentVolume resources with details of the actual storage. Requires pre-allocation of storage resources.



## Dynamic Provisioning

Storage volumes created on-demand based on PersistentVolumeClaim requests. Requires configured StorageClass with provisioner.

## CSI External Provisioner

Container Storage Interface allows third-party storage providers to implement custom provisioners without modifying Kubernetes core code.

# Static vs. Dynamic Provisioning

Feature	Static Provisioning	Dynamic Provisioning
Storage creation time	Before PVC creation	At PVC creation time
Admin involvement	Manual creation of each PV	One-time StorageClass setup
Storage utilization	Often leads to waste with pre-allocated resources	Just-in-time allocation, better utilization
Implementation complexity	Simple but labor-intensive	More complex setup, simpler operations
Use cases	Specialized storage needs, existing storage migration	General purpose applications, cloud deployments

# Persistent Volumes

A PersistentVolume (PV) is a cluster resource that represents a piece of storage that has been provisioned by an administrator or dynamically provisioned using StorageClasses.

Key characteristics:

- Cluster-wide resource, not namespaced
- Lifecycle independent of any pod
- Managed by cluster administrators
- Represents actual storage in the infrastructure

```
apiVersion: v1
kind: PersistentVolume
metadata:
  name: pv-nfs-data
spec:
  capacity:
    storage: 10Gi
  accessModes:
    - ReadWriteMany
  persistentVolumeReclaimPolicy: Retain
  nfs:
    server: nfs-server.example.com
    path: "/exported/path"
```

# PersistentVolume Attributes

1

## Capacity

Storage size specified using "storage" attribute (e.g., 10Gi)

2

## Access Modes

- ReadWriteOnce (RWO): volume can be mounted as read-write by a single node
- ReadOnlyMany (ROX): volume can be mounted read-only by many nodes
- ReadWriteMany (RWX): volume can be mounted as read-write by many nodes
- ReadWriteOncePod (RWOP): volume can be mounted as read-write by a single pod

3

## Reclaim Policy

- Retain: Manual reclamation
- Delete: Associated storage asset deletion
- Recycle: Basic scrub (`rm -rf /volume/*`)

4

## Mount Options

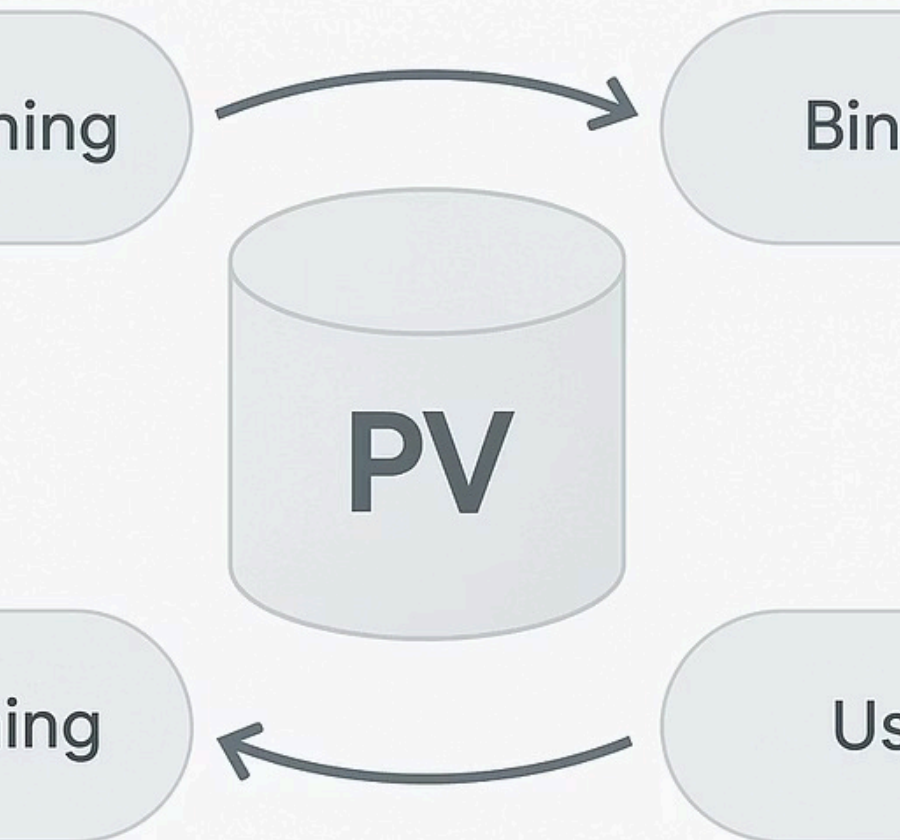
Specify additional mount options for when a PV is mounted on a node

5

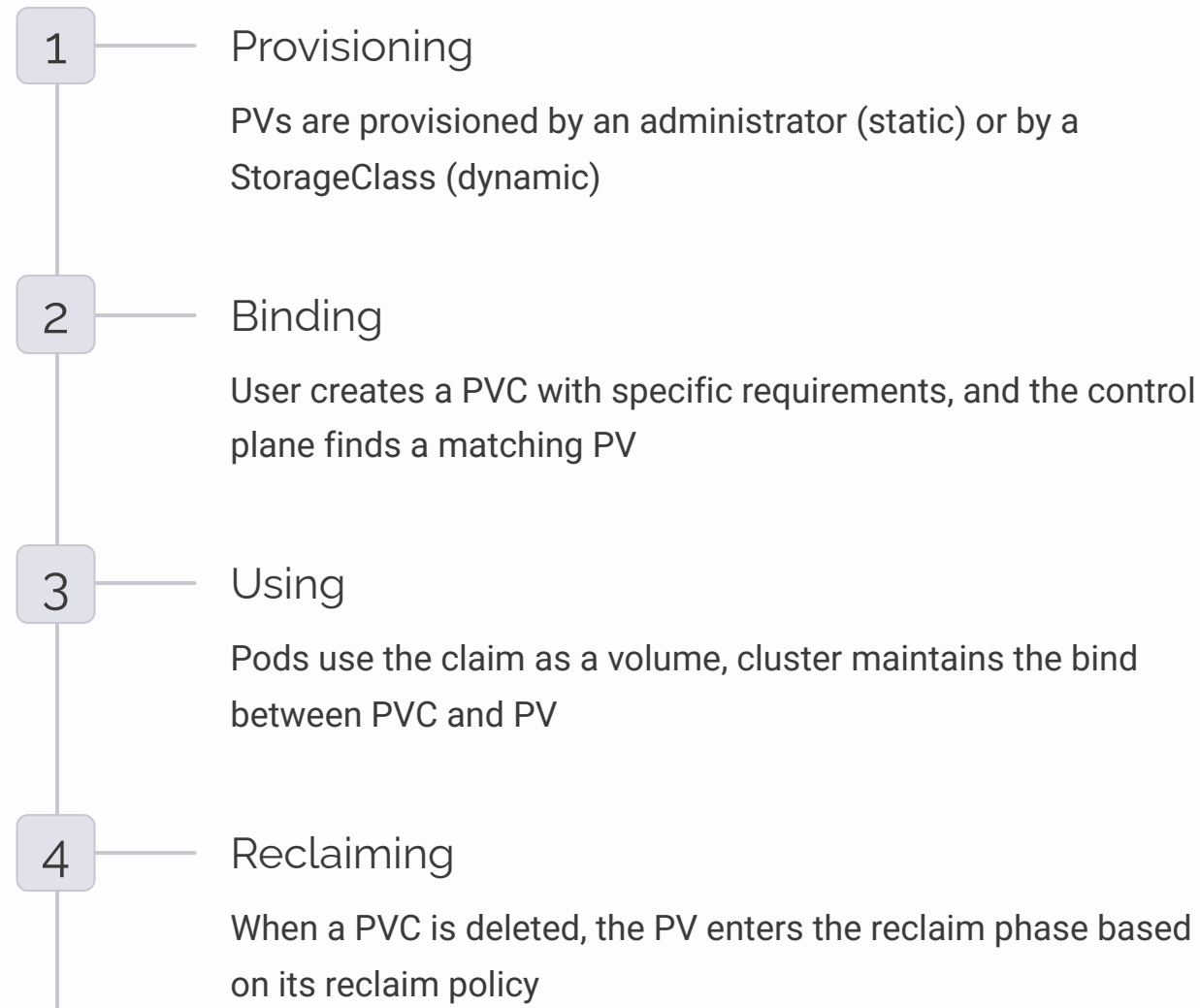
## Node Affinity

Constrains what nodes a volume can be accessed from

# Kubernetes Persistent Volume Lifecycle



## PersistentVolume Lifecycle



# Persistent Volume Claims

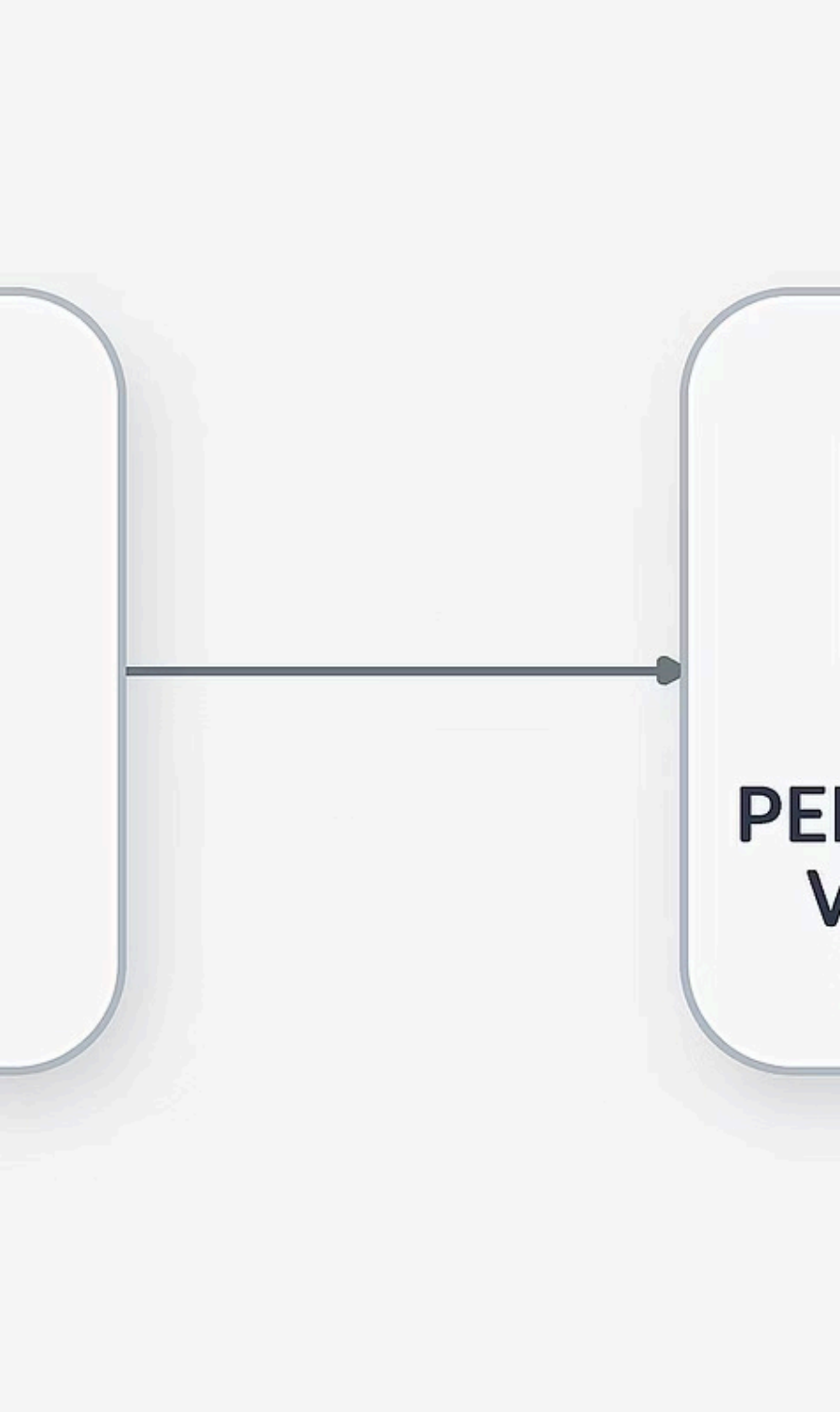
A PersistentVolumeClaim (PVC) is a request for storage by a user that can be fulfilled by a PersistentVolume.

Key characteristics:

- Namespaced resource, belongs to a specific namespace
- Represents a user's request for storage
- Can specify storage requirements (size, access mode)
- Can reference a specific StorageClass

```
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
  name: mysql-data-claim
  namespace: database
spec:
  accessModes:
    - ReadWriteOnce
  resources:
    requests:
      storage: 8Gi
      storageClassName: fast-ssd
```

# PVC to Pod Connection



```
apiVersion: v1
kind: Pod
metadata:
  name: database-pod
spec:
  containers:
    - name: database
      image: mysql:8.0
      volumeMounts:
        - mountPath: "/var/lib/mysql"
          name: mysql-data
      env:
        - name: MYSQL_ROOT_PASSWORD
          value: "password"
      volumes:
        - name: mysql-data
          persistentVolumeClaim:
            claimName: mysql-data-claim
```

# PVC Binding Process

## PVC Creation

User creates a PVC with specific requirements (size, access mode, storage class)

## Matching

Control plane searches for a PV that satisfies the PVC requirements

## Binding

Control plane binds the PVC to a matching PV (1:1 relationship)

## Dynamic Provisioning

If no matching PV exists and a storage class is specified, the system attempts to dynamically provision a new PV

## Pending State

If no matching PV exists and dynamic provisioning isn't possible, the PVC remains in a Pending state



# Understanding Storage Classes

A StorageClass provides a way for administrators to describe the "classes" of storage they offer. Different classes might map to quality-of-service levels, backup policies, or arbitrary policies determined by cluster administrators.

Key functions:

- Enables dynamic provisioning of PersistentVolumes
- Abstracts underlying storage provider details
- Allows for different tiers of storage
- Simplifies storage management for end users

```
apiVersion: storage.k8s.io/v1
kind: StorageClass
metadata:
  name: fast-ssd
provisioner: kubernetes.io/gce-pd
parameters:
  type: pd-ssd
  replication-type: none
reclaimPolicy: Delete
allowVolumeExpansion: true
volumeBindingMode: Immediate
```

# Storage Class Parameters

## Provisioner

Determines what volume plugin is used for provisioning PVs. Internal provisioners are prefixed with "kubernetes.io" and CSI provisioners typically use their driver name.

## Parameters

Provisioner-specific parameters for volume creation. These vary by provisioner (e.g., for AWS EBS: type, iopsPerGB, encrypted, etc.)

## ReclaimPolicy

Controls what happens to PVs created by the StorageClass when their PVC is deleted. Default is Delete but can be set to Retain.

## VolumeBindingMode

Controls when volume binding and dynamic provisioning occur. Immediate (default) or WaitForFirstConsumer which delays binding until a pod using the PVC is created.

## AllowVolumeExpansion

Boolean flag that determines whether PVCs can be expanded after creation to request more storage.

# Volume Expansion

Kubernetes supports online volume expansion for certain volume types when enabled in the StorageClass.

Process:

1. StorageClass must have allowVolumeExpansion: true
2. Edit the PVC to request more storage
3. Storage expansion happens automatically
4. For some volume types, pod restart may be required

Note: Volume shrinking is not supported in Kubernetes

```
# Edit PVC to request more storage
kubectl edit pvc mysql-data-claim
```

```
# In the editor, change:
```

```
spec:
```

```
  resources:
```

```
    requests:
```

```
      storage: 8Gi
```

```
# To:
```

```
spec:
```

```
  resources:
```

```
    requests:
```

```
      storage: 15Gi
```

# Storage Administration Best Practices

## 1 Create Default StorageClass

Set up a default StorageClass for general workloads to enable easy dynamic provisioning

## 2 Implement Storage Tiers

Define multiple StorageClasses for different performance needs (e.g., fast-ssd, standard-hdd, high-iops)

## 3 Configure Resource Quotas

Set ResourceQuota for PVCs to limit storage consumption per namespace

## 4 Plan for Disaster Recovery

Implement backup solutions for PVs containing critical data

## 5 Use WaitForFirstConsumer Binding Mode

Especially in multi-zone clusters to ensure volume is created in the same zone as the pod

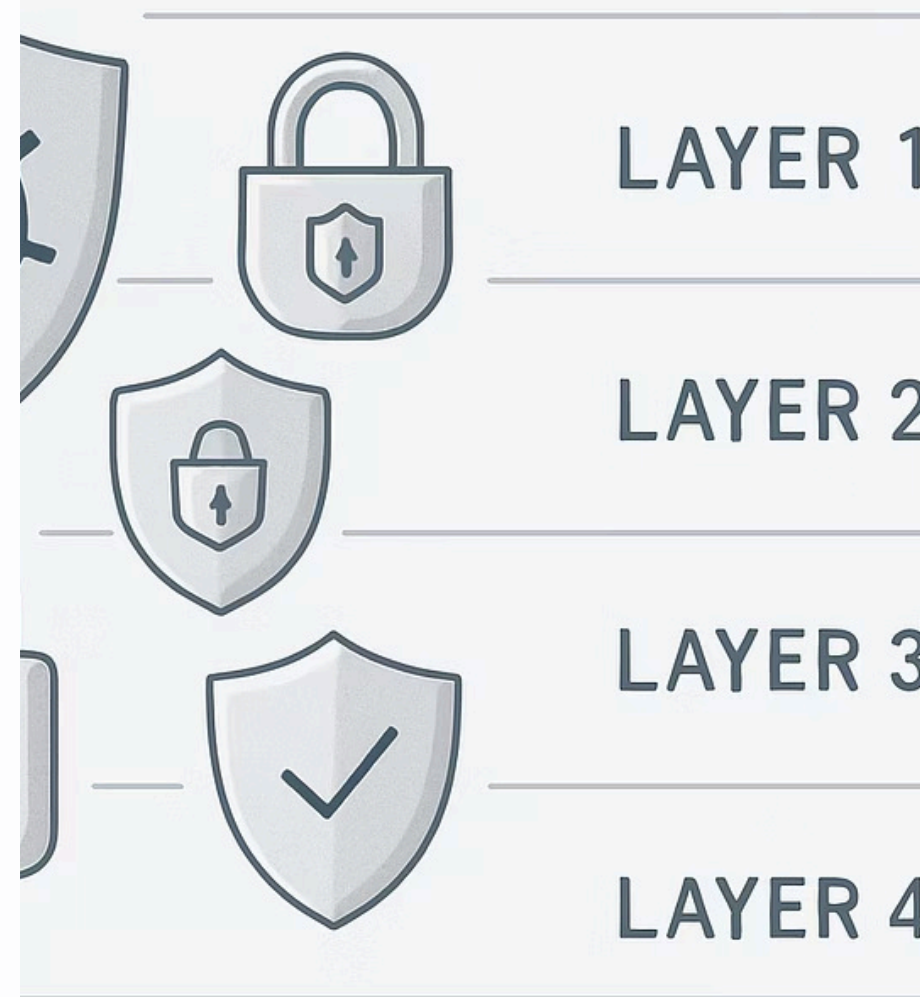
# Kubernetes Security: Overview

Kubernetes security is a multi-layered approach encompassing infrastructure security, cluster security, container security, and application security.

We'll focus on key administrative aspects:

- Authentication mechanisms
- Authorization with RBAC
- Role-based access control implementation
- Security Context configuration

## KUBERNETES SECURITY ARCHITECTURE



# Kubernetes Security Model



## Authentication

Who can access the cluster?

---



## Authorization

What actions can they perform?

---



## Admission Control

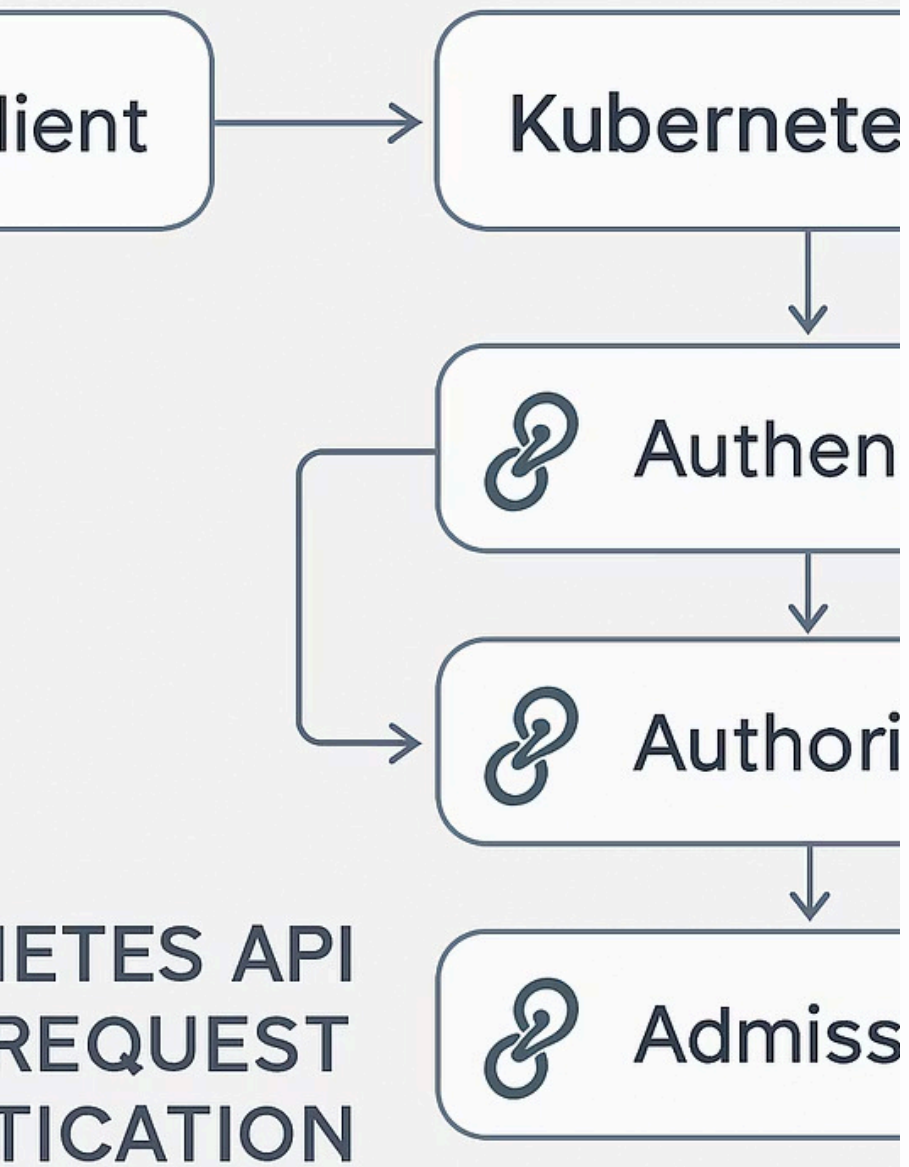
What modifications are required during request processing?

---



## Pod/Container Security

How are workloads isolated and secured?



# Understanding Kubernetes Authentication

Kubernetes uses a request processing chain to authenticate API requests:

1. TLS establishment between client and API server
2. Authentication plugins examine request for user identity
3. System authenticates as anonymous user if all plugins fail
4. API server doesn't provide an identity management system - relies on external sources
5. Every request must be authenticated or rejected

# Authentication Methods

1

## X.509 Client Certificates

- TLS client certificates for authentication
- Client cert's CN field becomes username
- Organization fields become user's group memberships
- Requires certificate management infrastructure

2

## Static Token File

- API server reads bearer tokens from a file
- Each line contains token, username, and user ID
- Simple but requires API server restart to update

3

## Service Account Tokens

- Automatically managed accounts
- Mounted into pods at known location
- JWT tokens signed by kube-apiserver

4

## OpenID Connect

- OAuth 2.0 extension adding ID token
- Integrates with external identity providers
- Supports SSO across multiple systems



# Kubernetes Users: Two Types

## Normal Users

- Managed externally to Kubernetes
- May come from corporate directory (LDAP, IAM)
- Authentication handled by certificate authorities, OAuth, etc.
- Cannot be created via API calls
- Used by humans and external processes

## Service Accounts

- Managed by Kubernetes API
- Bound to specific namespaces
- Created automatically or via API
- Credentials stored as Secrets
- Mounted to pods for application use
- Used by processes running in pods

# Service Account Management

Service accounts are namespaced Kubernetes resources that provide identities for processes running in pods.

Key operations:

- Creation and configuration
- Token management and rotation
- Assignment to pods
- Permissions management via RBAC

```
# Create a service account
```

```
kubectl create serviceaccount jenkins -n ci-cd
```

```
# View service account details
```

```
kubectl describe serviceaccount jenkins -n ci-cd
```

```
# Create token for service account
```

```
kubectl create token jenkins -n ci-cd
```

```
# Assign to pod
```

```
apiVersion: v1
```

```
kind: Pod
```

```
metadata:
```

```
  name: jenkins-agent
```

```
spec:
```

```
  serviceAccountName: jenkins
```

```
  containers:
```

```
    - name: jenkins-agent
```

```
      image: jenkins/agent:latest
```

# Kubernetes Authorization

After a request is authenticated, it must be authorized. Kubernetes supports multiple authorization modules:

## Node Authorization

Special-purpose authorizer that grants permissions to kubelets based on the pods they are scheduled to run.

## Attribute-Based Access Control (ABAC)

Grants rights based on policies that combine attributes. Requires API server restart to update policies.

## Role-Based Access Control (RBAC)

Regulates access based on roles assigned to users. Most commonly used and flexible approach.

## Webhook

HTTP callbacks to external REST services for authorization decisions. Enables integration with external systems.

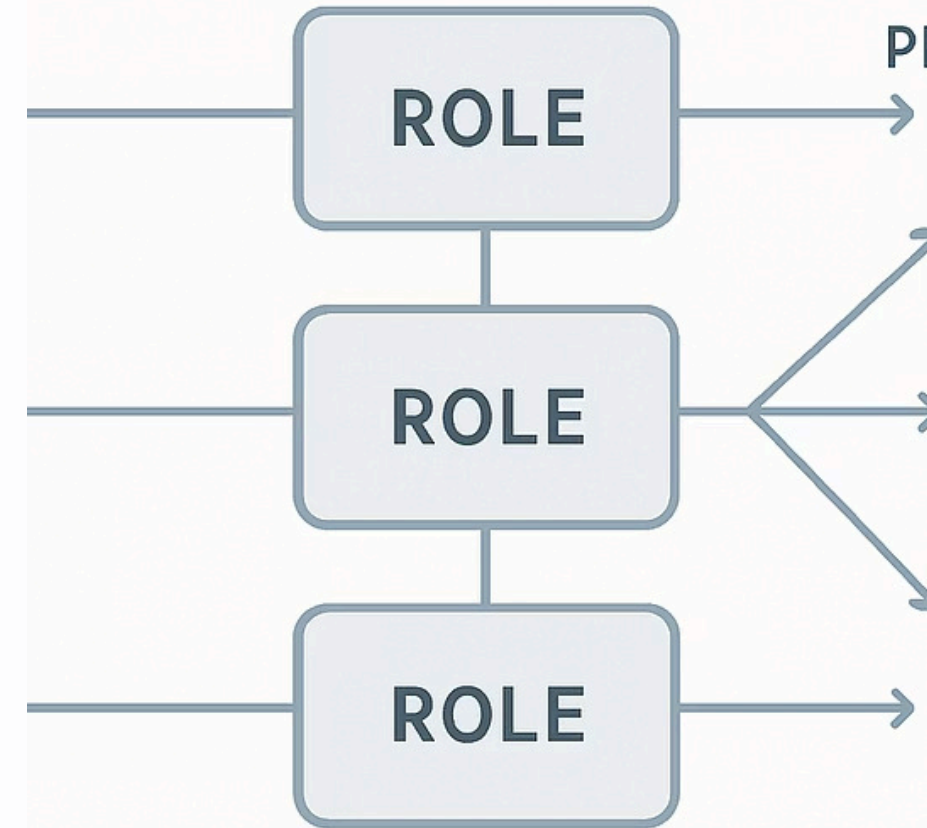
# Role-Based Access Control (RBAC)

RBAC is a method of regulating access to resources based on the roles of individual users within an organization.

Core concepts:

- Roles define permissions
- Subjects (users, groups, service accounts) are bound to roles
- Permissions are additive - there are no "deny" rules
- RBAC is enabled by default in Kubernetes v1.6+

## KUBERNETES RBAC MODEL



# RBAC API Objects

## Role

Namespace-scoped permission set that defines rules for resources within a specific namespace.

## ClusterRole

Cluster-wide permission set that defines rules for resources across the entire cluster.

## RoleBinding

Namespace-scoped binding that grants permissions defined in a Role or ClusterRole to subjects within a namespace.

## ClusterRoleBinding

Cluster-wide binding that grants permissions defined in a ClusterRole to subjects across all namespaces.

# Role and ClusterRole

Both Role and ClusterRole contain rules that represent a set of permissions. Permissions are purely additive (there are no "deny" rules).

A rule specifies:

- API Groups (e.g., apps, batch, networking.k8s.io)
- Resources (e.g., pods, deployments, services)
- Verbs (e.g., get, list, watch, create, update, delete)

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

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

# RoleBinding and ClusterRoleBinding

RoleBinding and ClusterRoleBinding bind subjects to roles.

Subjects can be:

- User accounts (string)
- Groups (string)
- Service accounts (namespace + name)

Key differences:

- RoleBinding: Namespace-scoped, grants permissions within specific namespace
- ClusterRoleBinding: Cluster-wide, grants permissions across all namespaces

```
apiVersion: rbac.authorization.k8s.io/v1
kind: RoleBinding
metadata:
  name: read-pods
  namespace: default
subjects:
- kind: User
  name: jane
  apiGroup: rbac.authorization.k8s.io
- kind: ServiceAccount
  name: jenkins
  namespace: ci-cd
roleRef:
  kind: Role
  name: pod-reader
  apiGroup: rbac.authorization.k8s.io
```

# Role vs ClusterRole: When to Use

Role	ClusterRole
Permissions for resources in a specific namespace	Cluster-wide resources (nodes, PVs)
Namespace-specific access to namespaced resources	Non-resource endpoints (/healthz)
Application-specific permissions	Resources across all namespaces
Team-specific access controls	Standardized permissions for common roles

Note: You can use a ClusterRole in a RoleBinding to reuse common role definitions across namespaces.



# Default ClusterRoles

1

cluster-admin

Super-user access that allows any action on any resource. When used in a ClusterRoleBinding, it gives full control over every resource in the cluster and in all namespaces.

2

admin

Allows admin access within a namespace. Intended to be granted within a namespace using a RoleBinding. Allows read/write access to most resources but not resource quotas or the namespace itself.

3

edit

Allows read/write access to most objects in a namespace but doesn't allow viewing or modifying roles or role bindings. Intended for developers working in their project's namespace.

4

view

Read-only access to most objects in a namespace. Cannot view secrets (as they contain sensitive information) or roles. For users who need to see resources but not modify them.

# RBAC Best Practices

## 1 Principle of Least Privilege

Grant only the permissions necessary for users and services to perform their functions, nothing more.

## 2 Use Groups for Management

Assign roles to groups rather than individual users for easier administration. Map external directory groups to Kubernetes groups.

## 3 Namespace Isolation

Use namespaces to isolate resources and implement separate RoleBindings within each namespace.

## 4 Avoid Direct cluster-admin Use

Limit cluster-admin privileges to a small number of administrators and emergency accounts.

## 5 Regularly Audit Permissions

Periodically review RBAC assignments and remove unnecessary permissions.

# Understanding Security Context

A security context defines privilege and access control settings for pods and containers. It allows you to control:

- User and group IDs
- Privileged or unprivileged execution
- Linux capabilities
- SELinux context
- AppArmor profiles
- Seccomp profiles
- AllowPrivilegeEscalation flag
- Read-only root filesystem

```
apiVersion: v1
kind: Pod
metadata:
  name: security-context-demo
spec:
  securityContext:
    runAsUser: 1000
    runAsGroup: 3000
    fsGroup: 2000
  containers:
    - name: secure-container
      image: busybox
      command: ["sh", "-c", "sleep 1h"]
      securityContext:
        allowPrivilegeEscalation: false
        readOnlyRootFilesystem: true
      capabilities:
        drop: ["ALL"]
        add: ["NET_BIND_SERVICE"]
```

# Pod vs Container Security Context

Setting	Pod Level	Container Level
runAsUser/runAsGroup	Yes	Yes (overrides Pod)
fsGroup	Yes	No
supplementalGroups	Yes	No
seLinuxOptions	Yes	Yes (overrides Pod)
capabilities	No	Yes
privileged	No	Yes
readOnlyRootFilesystem	No	Yes
allowPrivilegeEscalation	No	Yes

# Linux Capabilities in Kubernetes

Linux capabilities break down the privileges of the root user into distinct units that can be independently added or removed.

By default, Docker containers run with a reduced set of capabilities. In Kubernetes, you can:

- Add capabilities beyond the default set
- Drop capabilities from the default set
- Drop all capabilities and only add specific ones

Common capabilities:

- NET\_BIND\_SERVICE: Bind to ports below 1024
- NET\_ADMIN: Configure network interfaces
- SYS\_ADMIN: Perform system administration tasks
- SYS\_TIME: Modify system clock
- CHOWN: Change file ownership
- DAC\_OVERRIDE: Bypass file read/write/execute permission checks
- KILL: Bypass permission checks for sending signals

# Pod Security Context Best Practices

## 1 Run as Non-Root User

Set `runAsUser` to a non-zero value and use `runAsNonRoot: true` to prevent containers from running as root

## 2 Use Read-Only Root Filesystem

Set `readOnlyRootFilesystem: true` to prevent modifications to the container filesystem

## 3 Drop Unnecessary Capabilities

Drop "ALL" capabilities and only add those specifically required by the application

## 4 Prevent Privilege Escalation

Set `allowPrivilegeEscalation: false` to prevent processes from gaining more privileges than their parent

## 5 Use Pod Security Standards

Implement pod security standards (Privileged, Baseline, Restricted) using admission controllers

# Pod Security Standards

## Privileged

Unrestricted policy, providing the widest possible level of permissions. Allows for privilege escalation, hostPath volumes, and running as any user.

## Baseline

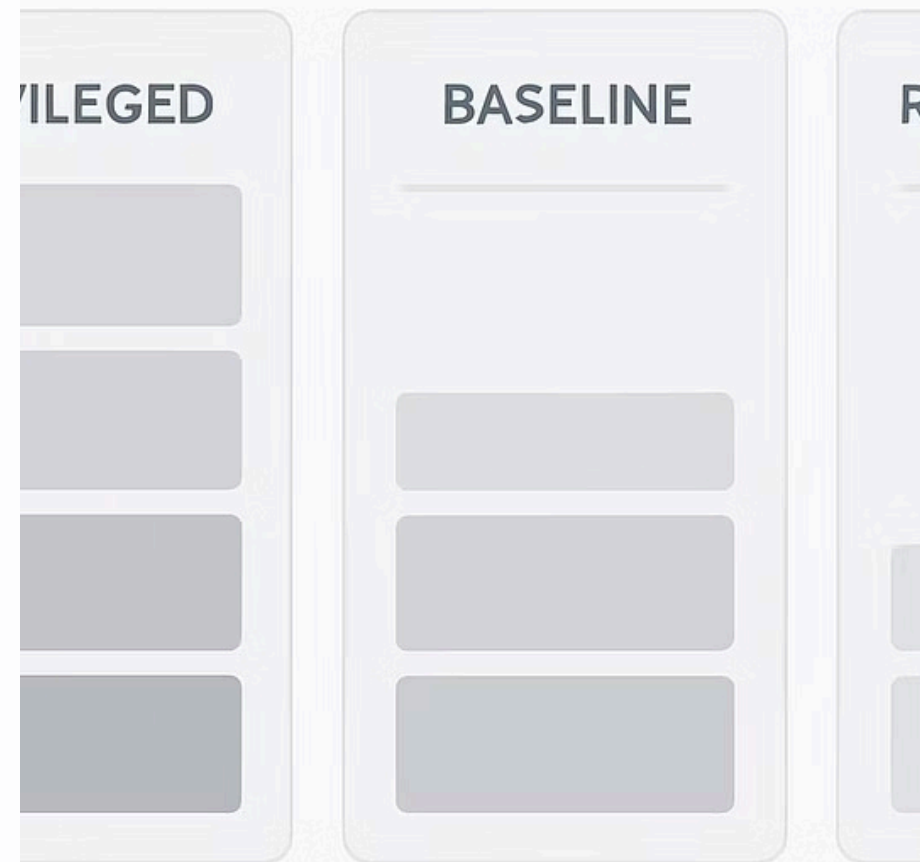
Minimally restrictive policy which prevents known privilege escalations. Allows the default (minimally specified) Pod configuration.

## Restricted

Heavily restricted policy, following current Pod hardening best practices. Requires running as non-root, prevents privilege escalation, and restricts capabilities.

## Kubernetes Pod Security

### Levels Comparison



# Key Takeaways: Storage & Security

## Storage

- Volume management provides persistent storage for containers
- PersistentVolumes abstract the underlying storage infrastructure
- PersistentVolumeClaims allow users to request storage resources
- StorageClasses enable dynamic provisioning and tiered storage options
- Proper storage configuration is critical for stateful applications

## Security

- Authentication verifies user identity through multiple methods
- RBAC provides fine-grained access control based on roles
- Roles define permissions, bindings connect users to roles
- Security Contexts configure pod and container security settings
- Follow principle of least privilege for all components