Kubernetes for App Developers (LFD459)

Introduction

- Objectives
- The Linux Foundation
- Linux Foundation Training
- Certification Programs and Digital Badging

Course Objectives

By the end of this course, you will be able to:

- Understand Kubernetes architecture and its core components.
- Deploy, manage, and scale containerized applications in Kubernetes.
- Work with Pods, Services, Deployments, and Volumes effectively.
- Understand security concepts, including authentication, RBAC, and policies.
- Use ConfigMaps, Secrets, and probes for application configuration and health checks.
- Troubleshoot, monitor, and expose applications inside and outside the cluster.

About The Linux Foundation

- The Linux Foundation (LF) is a non-profit consortium founded in 2000.
- It promotes open source innovation and protects the Linux ecosystem.
- Hosts hundreds of open source projects, including:
- Kubernetes, Cloud Native Computing Foundation (CNCF)
- Linux Kernel, Hyperledger, Node.js, OpenTelemetry, and more.

Training Philosophy

- Hands-on, practical, real-world focused.
- Created and maintained by active industry experts.
- Supported by major cloud providers (AWS, GCP, Azure).
- Designed to align with CNCF certifications.

Certification Programs and Digital Badging

Why Certification Matters:

- Validates your skills with recognized credentials.
- Demonstrates hands-on expertise, not just theory.
- Recognized globally by employers and the open source community.

Digital Badging:

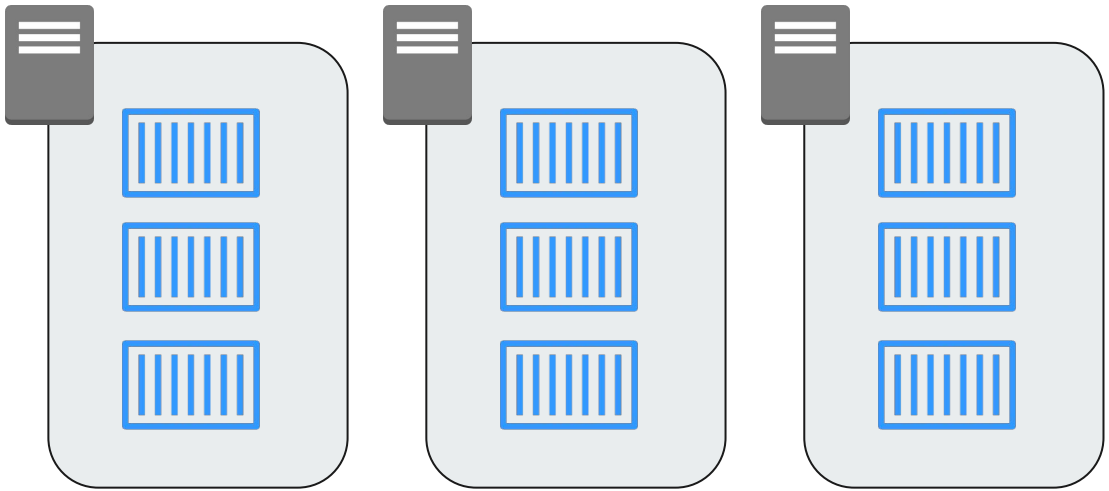
- Upon certification, you receive a verifiable digital badge.
- Share it on LinkedIn, GitHub, or your company profile.
- Each badge links to an online record of authenticity.

Kubernetes Architecture

- What Is Kubernetes?
- Components of Kubernetes
- Challenges
- The Borg Heritage
- Kubernetes Architecture
- Terminology
- Master Node
- Minion (Worker) Nodes
- Pods
- Services
- Controllers / Operators
- Single IP per Pod
- Networking Setup
- CNI Network Configuration File
- Pod-to-Pod Communication
- Cloud Native Computing Foundation
- Resource Recommendations
- Labs

What Is Kubernetes?

The Problem Before Kubernetes



- Manual container management and updates
- Difficult scaling and recovery
- Updates can caused downtime
- Managing multiple containers was becoming difficult and time-consuming.
- No centralized control or observability

Definition of Kubernetes

*"Kubernetes is an open-source system for **automating deployment**, **scaling**, and **management** of containerized applications."* – CNCF

Core Characteristics

- Declarative model – define what you want, not how (YAML/KYAML)
- Self-healing – restarts failed containers
- Scalability – add/remove replicas
- Rolling updates – deploy without downtime
- Service discovery – built-in DNS and load balancing

Components of Kubernetes

Control Plane vs Worker Plane

Layer	Responsibility	Key Components
Control Plane	Makes decisions, plans actions, maintains cluster state	API Server, Scheduler, Controller Manager, etcd, Cloud Controller Manager
Worker Plane	Executes workloads and reports their state	Kubelet, kube-proxy, Container Runtime

Challenges

Complexity of the Platform

- Kubernetes is **powerful but complex** — it spans networking, storage, scheduling, and security layers.
- Requires a deep understanding of **Linux**, **containers**, and **distributed systems**.
- Steep learning curve, especially for traditional infrastructure teams.

Security and Access Control

- Configuring RBAC correctly at scale is challenging.
- Managing secrets and enforcing Pod Security or Network Policies is often inconsistent.
- Multi-tenant environments require strong namespace isolation.
- Multi-cluster environments require plenty resources to manage them separately.

Networking

- Multiple abstraction layers: Pod → Service → Ingress → Mesh.
- Hard to debug when traffic routing fails or DNS doesn’t resolve.
- Different CNI implementations behave differently (Calico, Cilium, Flannel).

Storage and Stateful Workloads

- Stateless apps are easy; stateful workloads (databases, queues) require careful planning.
- PersistentVolumes, reclaim policies, and dynamic provisioning can be tricky.
- Cross-zone and backup consistency remain major operational concerns.

Resource Management

- Misconfigured resource requests cause throttling or overcommitment.
- Autoscaling requires proper metrics tuning (HPA, VPA, Cluster Autoscaler).
- Scheduler decisions may lead to node resource fragmentation.

Observability & Troubleshooting

- Requires multiple tools: Prometheus, Grafana, Loki, ELK, OpenTelemetry.
- Logs are ephemeral — need aggregation and retention strategies.
- Debugging distributed systems requires new mental models.

Operational Overhead

- Frequent version upgrades (~every 4 months).
- API deprecations and configuration drift.
- Balancing between **managed** (GKE/EKS/AKS) and **self-managed** clusters.

Organizational and Cultural Shift

- Requires a **DevOps** or **Platform Engineering** mindset.
- Developers must take ownership of deployment and runtime.
- Clear separation between platform and product teams is key.

The Borg Heritage

Google's Internal Evolution

- **Borg (2003)** – the original cluster management system used internally at Google.
- **Omega** – experimental reimplementation with better scalability and APIs.
- **Kubernetes (2014)** – open-source successor inspired by Borg, written in Go, donated to CNCF.

Key Design Philosophies from Borg

- Declarative model: describe what you want, not how to do it.
- Controllers maintain desired vs. actual state.
- Isolation and efficient scheduling across thousands of nodes.
- System self-healing and service discovery built-in.

Name and Symbolism

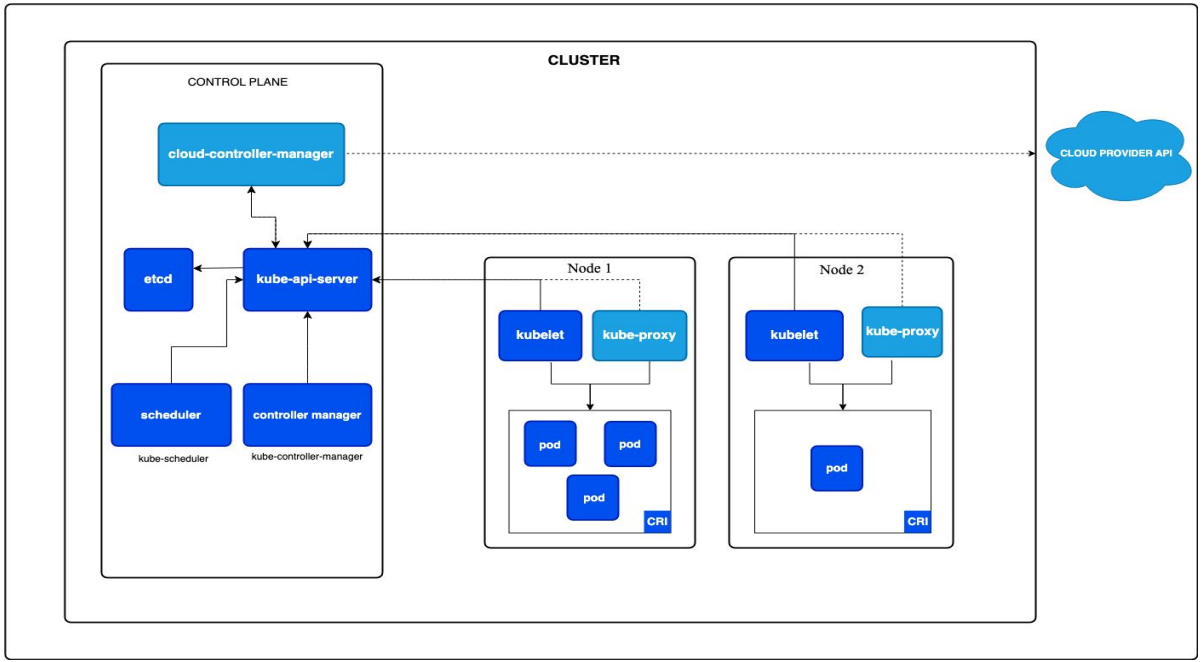
- Borg name origins from Star Trek
- Kubernetes means “helmsman” or “navigator” in Greek.
- The seven spokes in the Kubernetes logo reference Seven of Nine from Star Trek — an homage to Borg origins.

From Google to the World

- Kubernetes was released as open source in 2014.
- Donated to the Cloud Native Computing Foundation (CNCF).
- Now the de facto standard for container orchestration worldwide.

Kubernetes Architecture

Kubernetes Architecture



Terminology

Core Terminology

Term	Description	Example
Cluster	A set of machines managed by Kubernetes	Control + Worker nodes
Node	A single machine (virtual or physical)	Worker node
Pod	The smallest deployable unit	One or more containers
Service	Stable access point to Pods	ClusterIP, NodePort
Deployment	Controller that manages Pods and configuration	kubectl apply -f
Namespace	Logical resource grouping	dev, prod, staging

Master Node

Detailed Roles of Control Plane Components

API Server

- Acts as the **central communication** hub for the entire cluster.
- Exposes the **Kubernetes REST API**, which all components and clients (kubectl, controllers) use.
- Responsible for:
    - Validating and authenticating requests.
    - Persisting data in etcd.
    - Serving as the single source of truth for cluster state.

etcd

- A distributed **key–value database** storing the full cluster configuration and state.
- Keeps both **desired** and **current** state of all objects.
- Critical component – regular backups are mandatory.
- Used by API Server for every read/write operation.

Controller Manager

- Runs multiple internal controllers that constantly reconcile desired vs. actual state.
- Examples:
    - **Node Controller** – monitors node health.
    - **ReplicaSet Controller** – ensures correct number of Pod replicas.
    - **Deployment Controller** – manages rolling updates and rollbacks.
- If a Pod or Node disappears, the Controller Manager takes corrective action.

Scheduler

- Decides **which node** should host a newly created Pod.
- Evaluates:
    - Resource availability (CPU, memory).
    - Node taints and tolerations.
    - Pod affinity and anti-affinity rules.
- Once a node is chosen, the Scheduler communicates the decision to the API Server.

Cloud Controller Manager

- Integrates Kubernetes with **external cloud providers** (e.g., AWS, GCP, Azure).
- Manages:
    - Node lifecycle events (add/remove).
    - Load balancer provisioning.
    - Persistent volume integration.
- Makes Kubernetes cloud-aware while keeping Control Plane logic provider-agnostic.

Minion (Worker) Nodes

Detailed Roles of Worker Node Components

Kubelet

- Primary **node agent** that communicates directly with the API Server.
- Ensures all Pods assigned to the node are running and healthy.
- Core responsibilities:
  - Registers the node with the cluster.
  - Monitors Pod specs and reports status back to the Control Plane.
  - Restarts failed containers according to Pod definitions.
- Also handles **liveness** and **readiness** probes for health checking.

kube-proxy

- Maintains the network connectivity rules inside the node.
- Implements Service load balancing and Pod-to-Pod routing.
- Works in one of two modes:
  - **iptables mode** – creates NAT rules for routing.
  - **IPVS mode** – high-performance connection-based load balancing.
  - **nftables** – modern replacement for iptables using the Linux kernel's netfilter framework;
  - provides better performance, rule grouping, and scalability on new kernels.
- Ensures that traffic to a Service is evenly distributed among Pods.

Container Runtime

- Responsible for **running and managing containers** defined in Pods.
- Implements the **Container Runtime Interface** (CRI).
- Common implementations:
  - **containerd** – lightweight, high-performance runtime (default on most distros).
  - **CRI-O** – optimized for Kubernetes, used by Red Hat and OpenShift.
  - (Docker Engine deprecated since K8s v1.24)
- Handles:
  - Image pulling, container lifecycle.
  - Resource isolation via cgroups and namespaces.

Pods

Pod: The Smallest Deployable Unit

- The **smallest deployable and schedulable unit** in Kubernetes.
- Encapsulates one or more tightly coupled containers.
- Containers in the same Pod:
  - Share the same **network namespace** (same IP).
  - Can communicate via localhost.
  - Optionally share **storage volumes**.

Services

Service: Stable Network Identity

- Provides a **stable access endpoint** for a dynamic set of Pods.
- Abstracts Pod IP changes using a virtual IP and DNS name.
- Implements load balancing across replicas.
- Common types:
  - **ClusterIP** – internal cluster access (default).
  - **NodePort** – exposes on node ports (30000–32767).
  - **LoadBalancer** – integrates with external cloud LB.

Controllers / Operators

Controllers and Operators Overview

What Are Controllers?

- Controllers are **control loops** that continuously watch the cluster state and take action to move it toward the desired state.
- Every controller observes specific API resources (e.g., Pods, ReplicaSets, Nodes).
- When the actual state diverges, the controller makes changes through the API Server until both states match.

The Declarative Model

- Kubernetes works on a **desired vs. actual state** model:
  - You declare the desired state in YAML.
  - The API Server stores it in etcd.
  - Controllers reconcile continuously until the actual state matches.
- This loop is **idempotent** — applying the same manifest multiple times doesn't change the outcome.

Common Built-in Controllers

Deployment Controller

- Manages stateless workloads.
- Handles:
  - Replica management.
  - Rolling updates and rollbacks.
  - Pod template versioning (via ReplicaSets).
- Ensures no downtime during updates using rolling strategy or Recreate strategy.

StatefulSet Controller

- Manages **stateful** applications (DBs, caches, message queues).
- Ensures:
  - Predictable Pod names and ordering.
  - Stable persistent storage.
  - Graceful scaling (one Pod at a time).
- Used for workloads like PostgreSQL, RabbitMQ, Kafka.

DaemonSet Controller

- Ensures exactly **one Pod per Node** (or one per selected set of nodes).
- Typical use cases:
  - Log collectors (Fluentd, Filebeat).
  - Node monitoring agents (Prometheus Node Exporter).
  - Network proxies or firewalls.
- When a new node joins, the DaemonSet automatically schedules its Pod there.

Job and CronJob Controllers

- **Job**: Runs a Pod until it successfully completes once.
- **CronJob**: Schedules a Job to run at specified times (like a Linux cron).
- Common for:
  - Batch tasks.
  - Database backups.
  - Periodic cleanup scripts.

Custom Controllers and Operators

Why Custom Controllers Exist?

- Built-in controllers cover common patterns (stateless/stateful jobs, etc.).
- But modern systems (e.g., Prometheus, PostgreSQL, Kafka) require domain-specific automation.
- Custom controllers let developers extend Kubernetes' functionality through the same control-loop principle.

CustomResourceDefinition (CRD)

- Allows defining a new API type (custom resource).
- Example:

```yaml
apiVersion: myorg.io/v1
kind: Database
metadata:
  name: customers-db
spec:
  replicas: 2
  storage: 10Gi
```

- Once applied, Kubernetes recognizes kind: Database as a valid object.

Operator Pattern

- Combines:
  - A CustomResourceDefinition (new object type).
  - A Controller that knows how to manage it.
- Example: Prometheus Operator
  - Watches Prometheus and Alertmanager CRDs.
  - Creates StatefulSets, ConfigMaps, and Services automatically.
- Operators encode operational knowledge as code.

Controller Lifecycle and Flow

How Controllers Work

1. **Watch**: The controller continuously monitors the API Server for resource changes.
2. **Compare**: It compares the current (actual) state with the desired state.
3. **Act**: If they differ, it triggers actions (create/update/delete).
4. **Repeat**: This loop runs forever, maintaining equilibrium.

Behind the Scenes

- Controllers rely on informers (efficient event-based watchers).
- Use work queues to handle changes asynchronously.
- The model is event-driven and scalable across many controller instances.

Single IP per Pod

Core Networking Principle

- Every Pod in Kubernetes gets its **own unique IP** address.
- This IP is **shared among all containers within the Pod**.
- Containers inside a Pod communicate over *localhost*.
- Other Pods can reach this Pod directly via its IP — **no NAT, no port mapping**.

Why This Matters?

- Simplifies application development:
  - No need for complex container-to-container networking.
  - Containers can bind to the same ports (e.g., each Pod can expose port 80).
- Makes networking **predictable and consistent** across the cluster.
- Enables seamless **Pod-to-Pod** and **Service** communication.

Networking Setup

Goal of Kubernetes Networking

- Provide consistent communication between:
  - Pod ↔ Pod
  - Pod ↔ Service
  - External ↔ Cluster
- Ensure:
  - Every Pod has a routable IP.
  - No NAT between Pods.
  - DNS-based service discovery.

Networking Model Summary

- Each Pod has a unique IP → can reach every other Pod.
- Containers inside a Pod share the same network namespace.
- Services provide stable virtual IPs for dynamic Pods.
- NetworkPolicies control access when needed.

CNI

What Is CNI?

- **CNI (Container Network Interface)** is a specification that defines how container runtimes connect Pods to networks.
- It’s not a single plugin — it’s a standard API used by many plugins.
- When the Kubelet starts a Pod:
  - It calls the configured CNI plugin.
  - The plugin assigns an IP.
  - It sets up routing and interfaces inside the Pod namespace.

Popular CNI Plugins

Plugin	Type	Highlights
Flannel	Overlay	Simple, lightweight, uses VXLAN.
Calico	Routed / Policy	Supports NetworkPolicy and BGP routing.
Cilium	eBPF	High performance, fine-grained security.
Weave Net	Overlay	Auto-discovers peers, easy to set up.
Canal	Hybrid	Combines Flannel + Calico (VXLAN + policies).

CNI Network Configuration File

CNI Configuration

- CNI plugins are defined in */etc/cni/net.d/* on **each node**.
- Typical file example:

```
{
 "cniVersion": "0.4.0",
  "name": "k8s-pod-network",
  "type": "calico",
  "ipam": {
    "type": "host-local",
    "ranges": [[{"subnet": "10.244.0.0/16"}]]
  }
}
```

- The CNI plugin is called whenever a Pod is created or deleted.

Pod-to-Pod Communication

How Pods Communicate

- Each node has a Pod CIDR range (e.g., 10.244.1.0/24).
- When a Pod is created, the node's CNI plugin:
  - Allocates an IP from that CIDR.
  - Creates a veth pair to the node's virtual bridge.
  - Adds routing entries for other nodes' CIDRs.
- Inter-node traffic is routed via the node's network interface.

Inter-Node Routing Example

```
Pod A (10.244.1.10) — veth — Node1 ————> Node2 — veth — Pod B (10.244.2.10)
```

- Node1 knows that traffic for 10.244.2.0/24 goes to Node2.
- Depending on the plugin:
  - Flannel → encapsulates packets in VXLAN.
  - Calico → uses native routing (BGP).
  - Cilium → uses eBPF datapath.

Cloud Native Computing Foundation

What Is the CNCF?

- The Cloud Native Computing Foundation (CNCF) is an open-source foundation under the Linux Foundation.
- Its mission:

  > *"To make cloud-native computing universal and sustainable."*

- CNCF hosts and governs open-source projects that enable containerization, microservices, and cloud-native infrastructure.

  https://landscape.cncf.io

Resource Recommendations

Primary Sources of Truth

- Kubernetes Documentation: https://kubernetes.io/docs
  - Versioned API reference, guides, and tutorials.
- CNCF Projects: https://cncf.io/projects
  - Lists all sandbox, incubating, and graduated projects.
- API Reference Browser: https://kubernetes.io/docs/reference

Interactive Labs and Learning Platforms

https://killercoda.com/

- Browser-based, interactive terminal environments.
- Dozens of official Kubernetes labs: Pods, Deployments, Services, RBAC, etc.
- Supports custom scenarios for instructors.

https://labs.play-with-k8s.com/

- Spins up temporary multi-node clusters in your browser.
- Great for testing networking, scheduling, and scaling.
- Sessions last ~4 hours — ideal for workshops or short labs.

Labs

Build

- Container Options
- Containerizing an Application
- Creating the Dockerfile
- Hosting a Local Repository
- Namespace
- Creating a Deployment
- Running Commands in a Container
- Multi-Container Pod
- readinessProbe
- livenessProbe
- Labs

Container Options

What Is a Container Runtime(recap)?

- A container runtime is the low-level engine that actually runs containers.
- It:
  - Pulls container images from registries.
  - Sets up isolation (namespaces, cgroups).
  - Manages networking and storage for containers.
  - Reports state to **Kubelet** via the **Container Runtime Interface (CRI)**.

The Docker Era

- **2013 – Docker revolutionizes containers**
  - Docker made Linux container technology easy and accessible.
  - Combined everything:
    - Build system (Dockerfile)
    - Image registry (Docker Hub)
    - Runtime (containerd under the hood)
  - Kubernetes was **originally built to work directly with Docker**.

How Kubernetes Used Docker

- Kubernetes talked to Docker through an internal adapter called **Dockershim**.
- Dockershim acted as a translator between the **Kubelet** and the **Docker runtime API**.
- This worked well — until the ecosystem evolved and standardized.

The Rise of CRI and OCI

Container Runtime Interface (CRI)

- Introduced by Kubernetes to standardize runtime communication.
- Defines two main gRPC services:
  - RuntimeService → start/stop Pods.
  - ImageService → manage container images.
- Any runtime that implements CRI can plug into Kubernetes.

The Rise of CRI and OCI

Open Container Initiative (OCI)

- Founded in 2015 by Docker, Google, Red Hat, and others.
- Defines industry standards for:
  - Image format (OCI Image Spec)
  - Runtime behavior (OCI Runtime Spec)
- Ensures interoperability between runtimes and registries.

Deprecation of Docker in Kubernetes

- Until Kubernetes v1.23, Dockershim was still built in.
- Starting with v1.24, Dockershim was completely removed.
- Kubernetes no longer talks to Docker directly — only to CRI-compliant runtimes.
- Under the hood, Docker itself also uses containerd, so your images still work!

CR Options

Runtime	Based On	Highlights
containerd	Extracted from Docker, CNCF project	Default in most modern clusters
CRI-O	Red Hat's Kubernetes-native runtime	Used in OpenShift
gVisor	Google's sandbox runtime	Enhanced isolation via user-space kernel
Kata Containers	Lightweight VMs	Extra security, used in finance/telco

Containerizing an Application

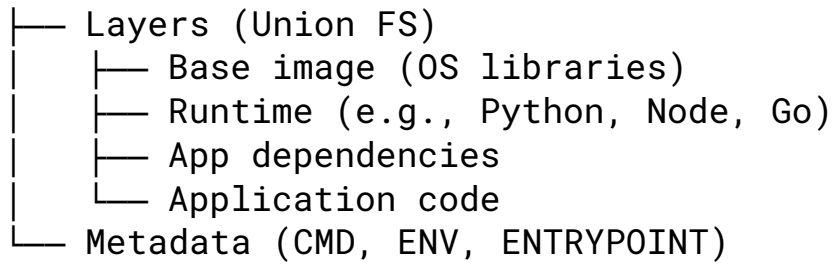
What Does “Containerizing” Mean?

- The process of packaging your application, its dependencies, and runtime into a single portable unit — a container image.
- Ensures your app runs the same way:
  - On a developer laptop,
  - In CI/CD pipelines,
  - In any Kubernetes cluster.
- A container is immutable — it can be recreated, replaced, or scaled without configuration drift.

How Containers Differ from Virtual Machines

Aspect	VM	Container
Isolation	Hardware-level (Hypervisor)	OS-level (Namespaces, cgroups)
Boot time	Minutes	Seconds
Size	GBs	MBs
Resources	Dedicated	Shared kernel
Portability	Limited	Very high (image-based)

Anatomy of a Container Image



- Built in layers — each instruction in a Dockerfile adds a new layer.
- Layers are cached and reused to speed up builds.
- Stored in registries (e.g., Docker Hub, GCR, GHCR).

Creating the Dockerfile

What Is a Dockerfile?

- A Dockerfile is a simple text file containing instructions to build a container image.
- Each instruction adds a new layer on top of the previous one.
- Docker uses these layers to assemble a reproducible image.

Example Dockerfile

```
# Base image
FROM python:3.12-slim
# Working directory inside the container
WORKDIR /app
# Copy dependencies first (for caching)
COPY requirements.txt .
# Install dependencies
RUN pip install --no-cache-dir -r requirements.txt
# Copy the application code
COPY . .
# Expose application port
EXPOSE 8080
# Define startup command
CMD ["python", "app.py"]
```

Dockerfile Instructions Overview

Core Instructions

Instruction	Instruction	Tip
FROM	Base image	Always use a tagged version, not latest
WORKDIR	Sets the working directory	Replaces cd
COPY / ADD	Sets the working directory	Prefer COPY — it's safer
RUN	Executes shell commands	Combine related commands in one layer
EXPOSE	Documents port usage	Does not actually publish the port
ENV	Sets environment variables	Use for config, not secrets
CMD	Default command	Can be overridden at runtime
ENTRYPOINT	Fixed entry command	Best for defining container purpose

Optimizing Dockerfiles

Use Multi-Stage Builds

```
FROM golang:1.22 AS builder
WORKDIR /src
COPY . .
RUN go build -o app

FROM gcr.io/distroless/base
COPY --from=builder /src/app /app
CMD ["/app"]
```

- Reduces image size
- Hides build tools from the final image
- Improves security

Keep Layers Minimal

- Combine commands to reduce layers:

```
RUN apt-get update && apt-get install -y curl && rm -rf /var/lib/apt/lists/*
```

- Avoid unnecessary files:

```
.dockerignore
.git
node_modules/
__pycache__/
```

Security Best Practices

- Never run as root:

```
RUN adduser --disabled-password appuser
USER appuser
```

- Use signed images or trusted registries.
- Scan images regularly:

```
trivy image myapp:v1
```

- Never hardcode secrets or tokens in the Dockerfile.

Common Mistakes to Avoid

- Using “latest” Tags
  - Non-deterministic builds → break CI/CD reproducibility.
- Installing Unnecessary Tools
  - Don’t install compilers, editors, or shells unless required.
- Caching Dependencies Incorrectly
  - Copy dependency files first to take advantage of caching.
- Not Cleaning Up
  - Leaving caches, temp files, or package lists increases image size.

Hosting a Local Repository

Why Do We Need a Registry?

- A registry is a storage and distribution system for container images.
- Every time Kubernetes creates a Pod, it must pull the image from a registry.
- Registries provide:
  - Centralized image storage.
  - Version control and tagging.
  - Security scanning and access control.
  - Caching and replication for faster deploys.

Public vs Private Registries

Type	Examples	Pros	Cons
Public	Docker Hub, GitHub Container Registry (GHCR)	Easy to use, free tiers	Limited privacy & control
Cloud-provider	GCR (Google), ECR (AWS), ACR (Azure)	IAM integration, scalable	Vendor lock-in
Private/Self-Hosted	Harbor, Nexus, Artifactory	Full control, on-premises security	Requires maintenance
Local	Localhost registry (:5000)	Quick testing, air-gapped	Not production-grade

Namespace

What Is a Namespace?

A Namespace in Kubernetes is a logical partition inside a cluster that groups and isolates resources.

- Think of it as a project or folder in your cluster.
- It allows multiple teams or environments to share a cluster without interfering with each other.
- Resource names must be unique within a Namespace, but can repeat across Namespaces.

Why Use Namespaces?

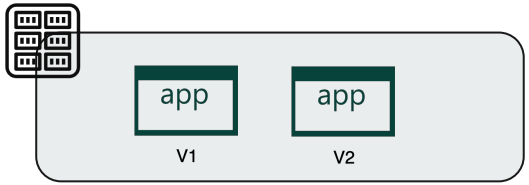
- Separate environments (dev, staging, prod).
- Manage multi-team clusters.
- Apply RBAC permissions per team.
- Enforce resource quotas and limits.

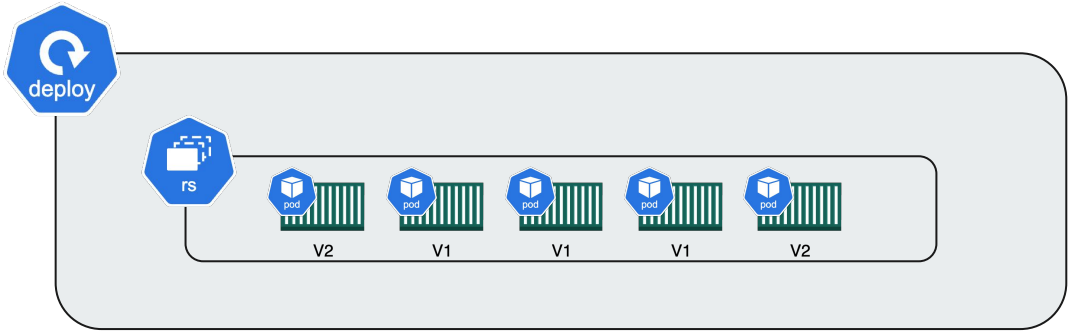
Namespace

```yaml
apiVersion: v1
kind: Namespace
metadata:
 name: dev
 labels:
   env: development
   team: workflows
```

Creating a Deployment

Deployment consideration





What Is a Deployment?

- A Deployment is a higher-level Kubernetes controller used to:
  - Run and manage multiple identical Pods.
  - Handle scaling up/down replicas.
  - Perform rolling updates and rollbacks automatically.
- It uses:
  - A ReplicaSet (which creates Pods).
  - Pod templates (defining the container spec).

Basic Deployment Example

```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: nginx-deploy
spec:
  replicas: 2
  selector:
    matchLabels:
      app: nginx
  template:
    metadata:
      labels:
        app: nginx
    spec:
      containers:
      - name: nginx
        image: nginx:1.27
        ports:
        - containerPort: 80
```

Running Commands in a Container

Why We Need to Execute Commands

- Kubernetes Pods are ephemeral — you can’t SSH into them.
- To debug or inspect a running container, use kubectl exec.
- It allows you to:
  - Check logs or temporary files inside the container.
  - Inspect environment variables and configuration.
  - Test connectivity from within the cluster.
  - Run admin commands like ps, netstat, curl, etc.

Basic Commands for Container Interaction

Inspecting Pods

```
kubectl get pods
kubectl get pods  <pod-name> -o yaml
kubectl describe pod <pod-name>
```

Getting Logs

```
kubectl logs <pod-name>
kubectl logs -f <pod-name>
kubectl logs <pod-name> -c <container-name>
```

Basic Commands for Container Interaction

Executing Commands

```
kubectl exec -it <pod-name> -- bash
kubectl exec -it <pod-name> -- sh
```

Run single commands

```
kubectl exec <pod-name> -- ls /app
kubectl exec <pod-name> -- cat /etc/os-release
```

Basic Commands for Container Interaction

Upload a file from local machine → container

```
kubectl cp ./config.yaml <pod-name>:/app/config.yaml
```

Download a file from container → local machine

```
kubectl cp <pod-name>:/app/logs/output.log ./output.log
```

Multi-Container Pod

What Is a Pod (Recap)

- Pod = The Smallest Deployable Unit in Kubernetes
- A Pod represents one or more containers that are scheduled together on the same Node.
- Kubernetes never runs containers directly — it always runs them inside a Pod.
- A Pod:
  - Defines shared resources for its containers.
  - Has one network namespace (shared IP, localhost).
  - Can define volumes for shared storage.

Pods with Multiple Containers

Yes — a Pod Can Have More Than One Container

- Each container in the Pod:
  - Runs in the same network namespace → shares IP and ports.
  - Can reach others via localhost:<port>.
  - Shares volumes for exchanging data.
  - Shares Pod-level configuration (env vars, labels, etc.).
- All containers:
  - Are started and stopped together.
  - Are scheduled on the same Node.
  - Share lifecycle — if the Pod dies, all containers die.

Shared Volume Between Containers

```yaml
apiVersion: v1
kind: Pod
metadata:
 name: writer-reader
spec:
 volumes:
 - name: shared-data
   emptyDir: {}
 containers:
 - name: writer
   image: busybox
   command:
     - sh
     - -c
     - >
       while true;
       do echo "$(date) - log entry" >> /data/log.txt;
       sleep 3;
       done
   volumeMounts:
   - name: shared-data
     mountPath: /data

 - name: reader
   image: busybox
   command:
     - sh
     - -c
     - >
       while true;
       do clear;
       echo "=== Reading from shared file ===";
       cat /data/log.txt;
       sleep 3;
       done
   volumeMounts:
   - name: shared-data
     mountPath: /data
```

readinessProbe

What Is a Readiness Probe?

- A readinessProbe tells Kubernetes when a container is ready to receive traffic.
- Even if the container is running (Running state), Kubernetes won't send traffic until the readiness check succeeds.
- The kubelet periodically runs the probe and:
  - Marks the Pod as ready in the Endpoints list when it passes.
  - Removes it from Service load-balancing when it fails.

Why We Need It?

Without a readinessProbe:

- The Service routes traffic to a Pod as soon as it starts.
- But the app may still be:
  - Initializing configuration.
  - Loading dependencies.
  - Waiting for an external service (DB, cache, etc.).

Example: HTTP Readiness Probe

```yaml
apiVersion: v1
kind: Pod
metadata:
 name: web-with-probe
spec:
 containers:
 - name: web
   image: nginx:1.27
   ports:
   - containerPort: 80
   readinessProbe:
     httpGet:
       path: /
       port: 80
     initialDelaySeconds: 5
     periodSeconds: 10
```

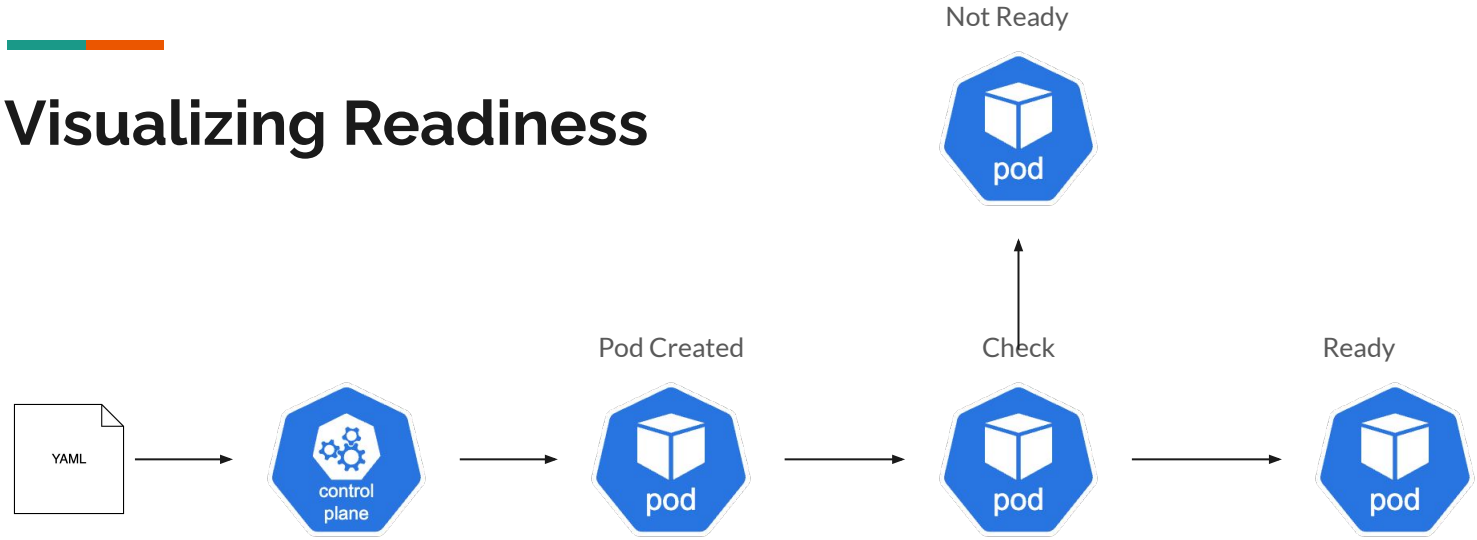
Example: Exec-Based Probe

```yaml
readinessProbe:
  exec:
    command:
      - cat
      - /tmp/ready
  initialDelaySeconds: 5
  periodSeconds: 3
```

Example: TCP Probe

```
readinessProbe:
 tcpSocket:
    port: 5432
  initialDelaySeconds: 10
  periodSeconds: 5
```

Visualizing Readiness



livenessProbe

What Is a Liveness Probe?

- A livenessProbe tells Kubernetes whether the container is still healthy.
- If the probe fails repeatedly, Kubernetes will:
  - Kill the container.
  - Restart it automatically (according to restart policy).
- Helps self-heal applications that hang or deadlock.

Why Liveness Probes Matter

Without livenessProbe:

- A Pod may appear “Running” even if the app:
  - Is stuck in a deadlock.
  - Consumes 100% CPU but doesn’t respond.
  - Has crashed internally without exiting the process.
- Kubernetes won’t restart it — because from the outside it still looks alive.

Example: HTTP Liveness Probe

```yaml
apiVersion: v1
kind: Pod
metadata:
 name: web-with-probe
spec:
 containers:
 - name: web
   image: nginx:1.27
   ports:
   - containerPort: 80
   livenessProbe:
     httpGet:
       path: /
       port: 80
     initialDelaySeconds: 5
     periodSeconds: 10
     failureThreshold: 3
```

Example: Exec-Based Probe

```
readinessProbe:
  exec:
    command:
      - cat
      - /tmp/healthy
  livenessProbe: 5
  periodSeconds: 3
```

Example: TCP Probe

```
livenessProbe:
 tcpSocket:
    port: 5432
  initialDelaySeconds: 10
  periodSeconds: 5
```

Labs

Design

- Traditional Applications: Considerations
- Decoupled Resources
- Transience
- Flexible Framework
- Managing Resource Usage
- Using Label Selectors
- Multi-Container Pods
- Sidecar Container
- Adapter Container
- Ambassador
- Points to Ponder
- Jobs
- Labs

Traditional Applications: Considerations

Traditional Applications vs Cloud-Native

How We Used to Build Applications:

- Applications were monolithic — one binary, one process.
- Long-lived servers with static IPs and persistent state.
- Manual scaling: add servers → reconfigure load balancer.
- Failures handled by humans, not systems.

Example:

- A Java WAR deployed on Tomcat on a VM.
- If Tomcat crashed, someone restarted it.
- Logs written to disk, configuration in local files.

Public vs Private Registries

Aspect	Traditional App	Cloud-Native App
Deployment target	Static VMs or bare metal	Dynamic Pods & clusters
Scaling	Manual, vertical	Automatic, horizontal
Configuration	Files or environment	ConfigMaps, Secrets
State	Local filesystem	Externalized (DB, object store)
Failure handling	Manual restart	Self-healing (probes, controllers)
Networking	Static IPs	Dynamic Service discovery
Logging	Local log files	Centralized (stdout → collector)

Challenges When Migrating Legacy Apps

Common Issues:

- Apps assume the filesystem is persistent.
- Hardcoded IPs or hostnames.
- Startup time too long for container lifecycle.
- No health endpoints for probes.
- State tightly coupled with logic.

Migration requires:

- Separating state from logic.
- Adding probes (/ready, /live).
- Externalizing config and secrets.
- Using shared storage for persistence.
- Decoupling inter-service communication.

The 12-Factor App: Foundation of Cloud-Native Design

What Is the 12-Factor App?

- A methodology for building modern, portable, scalable applications.
- Originally defined by Heroku engineers in 2011 — still relevant today.
- Focus: simplicity, automation, and separation of concerns.
- Kubernetes is a perfect execution platform for 12-factor apps.

The 12 Factors (in short)

Factor	Goal / Kubernetes Connection
Codebase	One codebase per app — use GitOps / CI pipelines
Dependencies	Explicitly declare in Docker image (requirements.txt, package.json)
Config	Store config in environment variables → ConfigMaps / Secrets
Backing Services	Treat DB, queues as attached resources → externalize via Services
Build, Release, Run	Separate build from runtime → use CI/CD and immutable images
Processes	Run app as one or more stateless processes → use Deployments

The 12 Factors (in short)

Factor	Goal / Kubernetes Connection
Port Binding	Export services via port → Kubernetes Services / Ingress
Concurrency	Scale via process model → Horizontal Pod Autoscaler
Disposability	Fast startup & graceful shutdown → probes + SIGTERM handling
Dev/Prod Parity	Keep environments similar → same manifests / Helm / ArgoCD
Logs	Treat logs as event streams → write to stdout, collect via sidecar / FluentBit
Admin Processes	Run admin tasks as one-off Pods / Jobs / CronJobs

How Kubernetes Enforces 12-Factor Principles

12-Factor Principle	Kubernetes Mechanism	Result
Config as env vars	ConfigMap, Secret	Dynamic, environment-driven config
Stateless processes	Deployment, ReplicaSet	Replaceable, self-healing Pods
Disposability	Pod lifecycle + probes	Safe rolling updates, graceful restarts
Backing services	Service, Endpoint, PVC	Loose coupling between app and data
Logs as streams	kubectl logs, stdout	Centralized collection (ELK, Loki, etc.)
Admin tasks	Job, CronJob	Run once and exit cleanly

Design Principle: Immutable Infrastructure

In Kubernetes, You Don't Patch — You Replace

- Pods are immutable:
  - You never modify a running Pod.
  - Instead, create a new one with the new version.
- Deployments handle replacement safely (rolling updates).

Why it matters?:

- Predictability → every Pod starts from the same image.
- No configuration drift.
- Easier rollback and version control.

Decoupled Resources

Decoupled Resources: Designing for Independence

Core Idea

Modern, cloud-native applications should be modular and self-contained, but loosely coupled to their environment. Each component should focus on a single responsibility, while configuration, secrets, and data are managed externally.

Separation of Concerns in Kubernetes

Area	Decoupled With	Purpose
Configuration	ConfigMaps	Centralize runtime configuration (env vars, settings)
Sensitive data	Secrets	Securely provide credentials, tokens, and keys
Storage	PersistentVolumes / Claims	Store state outside of Pods; survive restarts
Networking	Services / Ingress	Abstract connectivity; stable endpoints
Execution	Pods / Deployments	Define runtime unit and update strategy

Logical Segmentation of Applications

Designing for Modularity

- Group workloads by role or function:
- *frontend*, *backend*, *worker*, *cache*, *db*
- Each group can scale, update, and deploy independently.
- Communicate via Services, not direct IPs.

Use Labels to Define Relationships

- *app=frontend*, *tier=backend*, *component=api*, *etc.*
- Labels and selectors provide dynamic, declarative grouping.

One Pod, One Purpose

- A Pod should represent a single logical process or function.
- Keep containers focused and composable:
  - e.g., one container for app logic, another for a side task like metrics or log collection.
- Don’t overload a single Pod with multiple responsibilities.

Decoupling Enables Flexibility

- Independent updates and scaling.
- Simpler troubleshooting — smaller, well-defined boundaries.
- Easier CI/CD — no need to rebuild everything for one change.
- Security isolation — least privilege per component.
- Enables microservice evolution over time.

Transience

Transience: Everything Is Temporary

- Pods are ephemeral — they can disappear at any time.
- Nodes can be drained, restarted, or rescheduled.
- Persistent state must live outside of Pods.
- Stability comes from controllers and declarative intent, not uptime.

Statelessness as a Survival Strategy

- Stateless Pods can be killed and recreated without losing data.
- Scaling up or down is instant — no migration needed.
- Easier blue-green and canary deployments.
- All stateful data (sessions, files, transactions) lives in external systems:
  - Databases
  - Object storage (S3, GCS)
  - Caches (Redis, Memcached)

How Kubernetes Handles Transience

Controller	Purpose	Behavior When Pod Fails
Deployment	Manages stateless Pods	Recreates Pod automatically
StatefulSet	Manages stateful apps	Recreates with stable identity
DaemonSet	One Pod per node	Recreates missing Pods per node
Job / CronJob	One-time or scheduled tasks	Reschedules until completion

Designing for Transient Environments

- **Don’t assume persistence** — no data on local disk.
- **Be restartable** — the app should recover gracefully.
- **Handle SIGTERM** — use graceful shutdown to save state or drain requests.
- **Avoid sticky sessions** — store user context externally.
- **Make startup fast** — readinessProbe should pass quickly.

Graceful Shutdown and Recovery

The Shutdown Lifecycle

1. Kubernetes sends a SIGTERM to the container.
2. terminationGracePeriodSeconds (default 30s) starts counting down.
3. App should stop accepting new requests and finish current work.
4. After the grace period — SIGKILL is sent if the app hasn't exited.

Best Practices

- Implement signal handling in your code.
- Use readinessProbe to remove Pod from Service before stopping.
- Store checkpoints or jobs externally.

From Pets to Cattle Mindset

Old World (Pets)

- Each server is special — manually maintained and monitored.
- Downtime = panic, SSH session, debugging.

Kubernetes World (Cattle)

- Instances are identical and disposable.
- If one fails — recreate automatically.
- Focus on systems, not servers.

Transience in Practice

- **Deployment rolls out** → old Pods deleted, new ones created.
- **Node drained** → Pods rescheduled elsewhere.
- **Crash** → kubelet restarts the container.
- **Config change** → Deployment recreates Pods automatically.

Flexible Framework

Kubernetes Is Not One Way to Run Apps

- It’s a toolbox of patterns for different workloads.
- You choose how your app behaves:
  - Persistent or transient
  - Singleton or replicated
  - Scheduled or continuous
- The framework adapts to your design — not the other way around.

Declarative Control, Flexible Execution

- You define what you want (desired state).
- Kubernetes uses different controllers to achieve it.
- Controllers provide patterns, not restrictions.

Why Flexibility Matters

- Some apps are stateless, others stateful — Kubernetes supports both.
- You can mix patterns within one system:
  - API with Deployment
  - Database with StatefulSet
  - Log collector with DaemonSet
  - Maintenance task as Job
- All under one API, one management plane.

Kubernetes as an Evolving Platform

From Built-Ins to Custom Automation

- The same framework lets you create your own controllers.
- Operators, CRDs, and automation loops extend flexibility beyond built-ins.
- Instead of scripting actions — you teach Kubernetes new behaviors.

Managing Resource Usage

Why Resource Management Matters?

- Containers share the same host resources (CPU, memory, I/O).
- Without limits, one process can starve others.
- Kubernetes enforces fairness and predictability by:
  - Declaring requests (what you need).
  - Setting limits (what you can’t exceed).

Requests and Limits Explained - Pod

```yaml
apiVersion: v1
kind: Pod
metadata:
 name: resource-demo
 labels:
   app: demo
spec:
 containers:
 - name: demo-container
   image: nginx:1.27
   ports:
   - containerPort: 80
   resources:
     requests:
       cpu: "250m"      # Minimum
guaranteed
       memory: "256Mi"
     limits:
       cpu: "500m"      # Maximum allowed
       memory: "512Mi"
```

Requests and Limits Explained - Deployment

```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
 name: resource-demo
spec:
 replicas: 3
 selector:
   matchLabels:
     app: demo
 template:
   metadata:
     labels:
       app: demo
   spec:
     containers:
     - name: demo-container
       image: nginx:1.27
       ports:
       - containerPort: 80
       resources:
         requests:
           cpu: "250m"
           memory: "256Mi"
         limits:
           cpu: "500m"
           memory: "512Mi"
```

How the Scheduler Uses Requests

- Scheduler only places Pods if a Node has enough free capacity for all requests.
- It doesn’t consider limits — only guarantees.
- Example:
- Node has 2 CPU.
- Pod requests 0.5 CPU → up to 4 such Pods can fit.

Quality of Service (QoS) Classes

QoS Class	Criteria	Behavior Under Pressure
Guaranteed	requests == limits for all containers	Highest priority — last to be killed
Burstable	Requests < Limits	Balanced — may be throttled
BestEffort	No requests or limits set	Lowest priority — first to be evicted

Namespace-Level Control: LimitRange

```yaml
apiVersion: v1
kind: LimitRange
metadata:
 name: namespace-limits
 namespace: dev
spec:
 limits:
 - default:
     cpu: 500m
     memory: 512Mi
   defaultRequest:
     cpu: 250m
     memory: 256Mi
   type: Container
```

Namespace-Level Control: ResourceQuota

```yaml
apiVersion: v1
kind: ResourceQuota
metadata:
 name: team-quota
 namespace: dev
spec:
 hard:
   requests.cpu: "2"
   requests.memory: "4Gi"
   limits.cpu: "4"
   limits.memory: "8Gi"
   pods: "20"
```

Balancing Cost and Performance

- Set requests = typical load, limits = max acceptable burst.
- Monitor usage — don’t guess.
- Use Vertical Pod Autoscaler (VPA) for dynamic tuning.
- Combine with Horizontal Pod Autoscaler (HPA) for scaling.
- Reserve resources in Nodes for system daemons via kube-reserved.

Ephemeral Storage — The Forgotten Resource

- Besides CPU and memory, Pods also consume temporary disk space.
- This includes:
  - /tmp directory inside the container
  - container logs
  - emptyDir volumes (without backing PV)
- Kubernetes lets you request and limit ephemeral storage the same way as CPU or memory.

Pod with Ephemeral Storage Limits

```yaml
apiVersion: v1
kind: Pod
metadata:
 name: ephemeral-demo
spec:
 containers:
 - name: writer
   image: busybox
   command: ["sh", "-c", "dd if=/dev/zero
of=/tmp/data.txt bs=1M count=50 && sleep
3600"]
   resources:
     requests:
       cpu: "100m"
       memory: "128Mi"
       ephemeral-storage: "100Mi"
     limits:
       cpu: "200m"
       memory: "256Mi"
       ephemeral-storage: "200Mi"
```

Using Label Selectors

What Are Labels?

- Key-value pairs attached to Kubernetes objects.
- Used to describe, group, or identify resources.
- Fully customizable — you decide the taxonomy.

```
labels:
  app: web
  tier: frontend
  env: prod
  team: workflows
```

What Are Selectors?

- Define which objects a higher-level resource should apply to.
- Example: A Deployment finds all Pods with matching labels.
- Works across many resource types: Deployments, NetworkPolicies, HPAs, RoleBindings, etc.

```
selector:
    matchLabels:
      app: web
      tier: frontend
```

Common Labeling Patterns

Key	Purpose	Example Value
app	Logical application name	checkout, api, frontend
tier	Functional layer	frontend, backend, cache
env	Environment	dev, staging, prod
team	Ownership / responsibility	workflows, infra
version	Versioning for rollouts	v1.2.4

Deployment Using Labels

```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
 name: web
 labels:
   app: web
   tier: frontend
spec:
 replicas: 3
 selector:
   matchLabels:
     app: web
     tier: frontend
 template:
   metadata:
     labels:
       app: web
       tier: frontend
   spec:
     containers:
     - name: web
       image: nginx:1.27
       ports:
       - containerPort: 80
```

Advanced Label Selectors

```
selector:
 matchExpressions:
 - key: env
   operator: In
   values:
   - staging
   - prod
 - key: tier
   operator: NotIn
   values:
   - cache
```

Operators:

In, NotIn, Exists, DoesNotExist

Use Labels for More Than Just Routing

Feature	How Labels Are Used
Services	Select which Pods receive traffic
Deployments / ReplicaSets	Match Pods to manage lifecycle
NetworkPolicies	Define allowed Pod-to-Pod communication
ResourceQuotas	Apply policies to label groups
Monitoring	Group metrics by label
RBAC / Access Control	Assign rules by labels via selectors
NodeSelector	Select dedicated nodes

Multi-Container Pods

Init Containers

What Are Init Containers?

- Special containers that run before regular application containers.
- Execute one-time setup tasks, such as:
  - Waiting for a dependency (DB, API).
  - Initializing configuration or data.
  - Applying schema migrations.
- Each init container must complete successfully before the next one (or main containers) start.

Why Use Init Containers?

Use Case	Example Purpose
Wait for service readiness	Poll database or API before app starts
Preload or copy data	Fetch files, certificates, or configs
Run migrations	Update DB schema before the app connects
Validate environment	Ensure required secrets or endpoints exist

Init Container Preparing Shared Data

```yaml
apiVersion: v1
kind: Pod
metadata:
 name: init-demo
spec:
 volumes:
 - name: shared-data
   emptyDir: {}
 initContainers:
 - name: init-fetch
   image: busybox
   command: ['sh', '-c', 'echo "Initializing
data..." > /data/info.txt']
   volumeMounts:
   - name: shared-data
     mountPath: /data
 containers:
 - name: app
   image: busybox
   command: ['sh', '-c', 'echo "App starting...";
cat /data/info.txt; sleep 3600']
   volumeMounts:
   - name: shared-data
     mountPath: /data
```

Real-World Use Cases

- **Database migration**: Run alembic upgrade head before web app starts.
- **Configuration injection**: Download templates from Git or S3.
- **Dependency wait**: Loop until curl -f http://db:5432 succeeds.
- **Security checks**: Verify TLS certs or secrets before use.

Sidecar Container

What Is a Sidecar?

- A secondary container inside the same Pod as the main application.
- Adds supporting functionality without modifying the main app code.
- Runs in parallel with the main container and shares:
  - Network namespace (can use localhost)
  - Volumes (for shared data or logs)

Why Use the Sidecar Pattern?

Purpose	Example Sidecar Functionality
Logging / Monitoring	Collect and forward app logs (FluentBit, Vector)
Security / Secrets	Fetch and renew credentials (Vault Agent)
Networking	Proxy or service mesh sidecar (Envoy, Istio)
Data sync	Sync files or configs (rsync, git-sync)
Metrics / Telemetry	Export app metrics (Prometheus sidecar)

Benefits and Trade-offs

Pros:

- Clean separation of concerns.
- Reuse sidecars across many apps.
- Easier upgrades of observability and security components.

Cons:

- Slightly higher resource usage (CPU, memory).
- Coupled lifecycle — if Pod restarts, both containers restart.
- Debugging can be more complex.

Adapter Container

What Is an Adapter Container?

- A secondary container inside the same Pod that transforms or normalizes data.
- Used when the main application outputs logs, metrics, or data in a non-standard format.
- The adapter translates it into a format that the rest of the platform understands.

Why Use an Adapter?

Need	Adapter Role
App outputs custom logs	Parse and convert to JSON or structured format
App exposes proprietary metrics	Transform into Prometheus-compatible /metrics
App produces raw data files	Reformat, compress, or enrich before export
Legacy binary app	Add observability or monitoring without code changes

When to Use an Adapter vs a Separate Service

Use an Adapter (in the same Pod) when:

- Tight data coupling is required.
- The transformation depends on local files or app state.
- Both containers should always start and stop together.

Use a separate service when:

- Data processing can be asynchronous.
- Scaling needs differ from the app.
- Many apps send data to one processor.

Benefits and Trade-offs

Pros:

- No change to the main app's code.
- Adds observability or integration to legacy software.
- Keeps business logic clean and focused.

Cons:

- Shares resources (CPU, memory, disk) with the app.
- Both containers restart together.
- Debugging may be harder (two processes, one Pod).

Ambassador

What Is an Ambassador?

- A container in the same Pod that acts as a **proxy** or **gateway** for outbound connections.
- The main container connects to the ambassador over localhost.
- The ambassador handles communication with **remote services** — databases, APIs, or external networks.

Why Use an Ambassador?

Goal	Ambassador Responsibility
Simplify external connectivity	Hide complex service names or endpoints
Add encryption	Handle TLS/mTLS for legacy apps
Add authentication	Inject API keys or tokens
Route traffic	Proxy to different environments or regions
Bridge networks	Connect Kubernetes to on-prem or external systems

When to Use an Ambassador

Use when:

- The app shouldn't handle network or security details.
- You need a local proxy pattern (1:1 between app and gateway).
- Migrating legacy workloads to Kubernetes with minimal code changes.

Avoid when:

- The proxy must serve multiple apps.
- It needs independent scaling or management.
- It's large or complex — then it belongs as a standalone service.

Benefits and Trade-offs

Pros:

- Keeps app configuration simple (localhost).
- Adds encryption and auth transparently.
- Enables migration of legacy systems to Kubernetes.

Cons:

- Tightly coupled lifecycle — both restart together.
- Not suitable for shared or multi-tenant proxying.
- Adds a bit of complexity and resource overhead.

Points to Ponder

Designing for Kubernetes

- Kubernetes doesn’t fix bad architecture — it exposes it.
- You design for failure, not against it.
- Pods are ephemeral, Services are stable, and ConfigMaps/Secrets are the memory of your system.
- The key question isn’t “how to deploy?” but “what happens when it fails?”

Common Design Pitfalls

- Using multi-container Pods for unrelated services.
- Ignoring resource limits — one Pod starving the node.
- Hardcoding IPs, ports, or DNS names.
- Cramming setup logic inside the main app instead of an init container.
- Rebuilding legacy systems as-is without adopting declarative design.

Architecture Is a Journey

- Start simple, evolve gradually.
- Automate configuration before scaling workloads.
- Observe before optimizing.
- Every system grows more reliable through iteration — not perfection on day one.

Jobs

What Is a Job?

- A **controller** that runs Pods **until they successfully complete**.
- Guarantees that a defined number of Pods finish successfully.
- Retries automatically on failure (depending on configuration).
- Used for **batch processing, migrations, and one-time tasks**.

Simple Job

```yaml
apiVersion: batch/v1
kind: Job
metadata:
 name: job-demo
spec:
 template:
   spec:
     containers:
     - name: worker
       image: busybox
       command: ['sh', '-c', 'echo
"Processing..."; sleep 5; echo
"Done"']
     restartPolicy: Never
```

Controlling Job Behavior

Field	Purpose	Example
completions	Number of successful Pods required	completions: 3
parallelism	Number of Pods running at once	parallelism: 2
backoffLimit	Max retry attempts before failure	backoffLimit: 4
activeDeadlineSeconds	Total job timeout	activeDeadlineSeconds: 600

Parallel Job

```yaml
apiVersion: batch/v1
kind: Job
metadata:
 name: parallel-job
spec:
 completions: 4
 parallelism: 2
 template:
   spec:
     containers:
     - name: worker
       image: busybox
       command: ['sh', '-c', 'echo
"Task $(hostname)"; sleep 3']
     restartPolicy: Never
```

What Is a CronJob?

- Creates Jobs on a time-based schedule, like Linux cron.
- Each schedule execution spawns a new Job.
- Great for:
  - Backups
  - Reports
  - Database maintenance
  - Cleanup tasks

CronJob

```yaml
apiVersion: batch/v1
kind: CronJob
metadata:
 name: cronjob-demo
spec:
 schedule: "*/2 * * * *"   # Every 2
minutes
 jobTemplate:
   spec:
     template:
       spec:
         containers:
         - name: hello
           image: busybox
           command: ['sh', '-c', 'echo
"Hello from $(date)"']
         restartPolicy: OnFailure
```

Controlling Job Behavior

Field	Purpose
schedule	Cron expression (*/5 * * * *)
startingDeadlineSeconds	How long to wait if a schedule is missed
concurrencyPolicy	What to do if previous Job is still running (Allow, Forbid, Replace)
successfulJobsHistoryLimit / failedJobsHistoryLimit	How many old Jobs to keep

Best Practices

- Always set *restartPolicy*: *Never* or *OnFailure*.
- Add *backoffLimit* to prevent endless retries.
- Use proper *concurrencyPolicy* for overlapping schedules.
- Keep logs short-lived — use centralized log collection.
- Consider using dedicated namespaces for scheduled workloads.

Labs

Deployment Configuration

- Volumes Overview
- Introducing Volumes
- Volume Spec
- Volume Types
- Shared Volume Example
- Persistent Volumes and Claims
- Persistent Volume
- Persistent Volume Claim
- Dynamic Provisioning
- Secrets
- Using Secrets via Environment Variables
- Mounting Secrets as Volumes
- Portable Data with ConfigMaps
- Using ConfigMaps
- Deployment Configuration Status
- Scaling and Rolling Updates
- Deployment Rollbacks
- Labs

Volumes Overview

Why We Need Volumes

- Containers are ephemeral — when a Pod dies, its data disappears.
- Applications often need to store, share, or persist data.
- Kubernetes provides Volumes to handle data more predictably.

What Volumes Provide

Feature	Purpose
Persistence	Keep data beyond container lifetime
Sharing	Enable multiple containers to access the same data
Abstraction	Decouple app logic from underlying storage
Flexibility	Work with temporary or long-term storage backends

Volume Categories (At a Glance)

Category	Example Use Case
Ephemeral (e.g., emptyDir)	Temporary runtime data
Persistent (e.g., PV, PVC, StorageClass)	Long-lived application data
Host-based (e.g., hostPath)	Access to node-local files

Introducing Volumes

What Is a Volume in Kubernetes?

- A **directory accessible to containers** in a Pod.
- Mounted into one or more containers.
- Lives as long as the **Pod**, not the container.
- Can be used for:
- Sharing data between containers
- Storing temporary files
- Injecting configuration or secrets

Defining a Volume Inside a Pod

1. Define the volume under spec.volumes
2. Mount it inside one or more containers with volumeMounts

```
spec:
 volumes:
 - name: my-volume
   <volumeType>: {}
 containers:
 - name: app
   volumeMounts:
   - name: my-volume
     mountPath: /data
```

Volume Lifecycle

- Volume is created **when the Pod starts**.
- Destroyed **when the Pod is deleted**.
- Shared between all containers in that Pod.
- Data is independent from container restarts — but still lost if the Pod itself is recreated (unless it's a PersistentVolume).

Defining and Mounting a Volume

```
apiVersion: v1
kind: Pod
metadata:
 name: volume-intro
spec:
 volumes:
 - name: data
   emptyDir: {}
 containers:
 - name: writer
   image: busybox
   command: ["sh", "-c", "echo 'hello' >
/data/msg && sleep 3600"]
   volumeMounts:
   - name: data
     mountPath: /data
 - name: reader
   image: busybox
   command: ["sh", "-c", "cat /data/msg
&& sleep 3600"]
   volumeMounts:
   - name: data
     mountPath: /data
```

Volume Spec

Where Volumes Live in a Manifest

Every Pod defines two related sections:

```
spec:
 volumes:                # Defines available storage for the Pod
 containers:
 - name: app
   volumeMounts:         # Defines how that storage is used by a container
```

The volumes block = storage declaration
The volumeMounts block = where and how that storage appears in a container

The Volume Definition (spec.volumes)

Field	Description	Example
name	Logical name of the volume (must be unique in Pod)	data
emptyDir, hostPath, configMap, secret, persistentVolumeClaim	Type of volume and its parameters	emptyDir: {}
persistentVolumeClaim.claimName	Link to an existing PVC	pvc-demo

Mounting Volumes (spec.containers.volumeMounts)

Field	Purpose	Example
name	Reference to a volume declared under spec.volumes	data
mountPath	Path inside the container where the volume is attached	/var/data
readOnly	Mount volume as read-only (default: false)	true
subPath	Mount only a subdirectory of the volume	"logs"

Using subPath and readOnly

```yaml
apiVersion: v1
kind: Pod
metadata:
 name: volume-spec-demo
spec:
 volumes:
 - name: shared
   emptyDir: {}
 containers:
 - name: writer
   image: busybox
   command: ['sh', '-c', 'echo "hello" >
/data/msg && sleep 3600']
   volumeMounts:
   - name: shared
     mountPath: /data
 - name: reader
   image: busybox
   command: ['sh', '-c', 'cat /read/msg &&
sleep 3600']
   volumeMounts:
   - name: shared
     mountPath: /read
     readOnly: true
     subPath: msg
```

Volume Types

Common Built-in Volume Types

Type	Purpose	Persistence
emptyDir	Temporary shared storage between containers	Until Pod deletion
hostPath	Access node's filesystem path	Tied to the node
configMap	Mount configuration data	Recreated dynamically
secret	Mount sensitive data	Managed by K8s
persistentVolumeClaim	Connect to long-lived storage	Survives Pod lifecycle

Other Volume Types (for completeness)

Type	Purpose
projected	Combines multiple sources (e.g., Secret + ConfigMap)
downwardAPI	Exposes Pod metadata (name, namespace, labels) as files
gitRepo (deprecated)	Clones a Git repo into the Pod (replaced by init containers)
csi	Generic interface for external storage plugins

Shared Volume Example

Volumes Are Shared at the Pod Level

- All containers in a Pod can mount the **same volume**.
- This allows data sharing between processes — e.g., one writes, another reads.
- Practically **every volume type** (e.g., emptyDir, configMap, secret, persistentVolumeClaim) can be shared **within the same Pod**.

Sharing Between Pods

- Only **cluster-backed volumes** (e.g., PersistentVolume / PVC, CSI, NFS) can be shared across Pods.
- emptyDir, configMap, and secret are **Pod-scoped** and disappear when the Pod is gone.
- Persistent storage is required for **multi-Pod cooperation** — databases, file shares, or caches.

Persistent Volumes and Claims

Core Concept

- PersistentVolume (PV) - Represents actual storage in the cluster (disk, NFS, cloud volume, etc.)
- PersistentVolumeClaim (PVC) - A user’s request for storage (size, access mode)

Benefits of the PV/PVC Model

- **Separation of concerns:**
  - Admins manage storage infrastructure.
  - Developers just request capacity.
- **Portability:**
  - The same PVC spec works across environments (cloud, on-prem, etc.).
- **Scalability:**
  - Supports dynamic provisioning for automatic volume creation.

Persistent Volume

What Is a Persistent Volume?

- A **cluster-wide storage resource** managed by Kubernetes.
- Created by an **administrator** or automatically via **StorageClass**.
- Represents a real storage backend: disk, NFS, iSCSI, CSI driver, etc.
- Independent of Pods — persists beyond their lifecycle.

PV Anatomy (Spec Overview)

Field	Purpose
capacity	Defines available storage size (e.g., storage: 10Gi)
accessModes	Defines how Pods can access it (ReadWriteOnce, ReadOnlyMany, ReadWriteMany)
persistentVolumeReclaimPolicy	What happens when the claim is released (Retain, Delete, Recycle)
storageClassName	Identifies which StorageClass the PV belongs to
volumeMode	Mount mode (Filesystem or Block)
claimRef	(Optional) The PVC currently bound to it

Static Persistent Volume

```
apiVersion: v1
kind: PersistentVolume
metadata:
 name: pv-demo
spec:
 capacity:
   storage: 2Gi
 accessModes:
   - ReadWriteOnce
 persistentVolumeReclaimPolicy:
Retain
 storageClassName: “”
 hostPath:
   path: "/mnt/data"
```

Access Modes Explained

Mode	Meaning	Typical Use Case
ReadWriteOnce	One node can mount read-write	Databases, local disks
ReadOnlyMany	Many nodes read-only	Shared datasets, configs
ReadWriteMany	Many nodes read-write	NFS, distributed file systems

Reclaim Policy

Policy	Behavior
Retain	Keeps the data — manual cleanup required
Delete	Deletes the backend storage automatically
Recycle (deprecated)	Performs a simple cleanup and reuses the PV

Binding States of a PV

State	Description
Available	PV ready for a claim
Bound	PV attached to a PVC
Released	Claim deleted, PV not yet reclaimed
Failed	PV could not be recycled or reused

Filesystem vs Block Mode

Mode	Description
Filesystem	Default; mounts as a standard directory
Block	Exposes raw block device to container (no filesystem)

Persistent Volume Claim

What Is a PVC?

- A **request for storage** by a user or workload.
- Specifies desired **size, access mode**, and optionally **storage class**.
- Kubernetes automatically binds the PVC to a matching **PersistentVolume (PV)**.
- Once bound, the PVC can be mounted into Pods like any other volume.

PVC Structure Overview

Field	Purpose
accessModes	Desired access mode (ReadWriteOnce, ReadWriteMany, etc.)
resources.requests.storage	Requested capacity (e.g., 2Gi)
storageClassName	Defines which storage backend to use
volumeMode	Optional — Filesystem (default) or Block
volumeName	Optional — manually bind to a specific PV

Basic PersistentVolumeClaim

```yaml
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
 name: pvc-demo
spec:
 accessModes:
   - ReadWriteOnce
 resources:
   requests:
     storage: 2Gi
 storageClassName: “”
 volumeName: pv-demo
```

Pod Using a PVC

```
apiVersion: v1
kind: Pod
metadata:
 name: pvc-pod
spec:
 volumes:
 - name: storage
   persistentVolumeClaim:
     claimName: pvc-demo
 containers:
 - name: app
   image: nginx
   volumeMounts:
   - name: storage
     mountPath:
/usr/share/nginx/html
```

Dynamic Provisioning

Dynamic Provisioning: Storage on Demand

Problem?

- Traditionally, **admins had to create PVs manually**.
- PVCs waited in Pending state until a matching PV appeared.
- This didn't scale for large, multi-team clusters.

Solution?

- **Dynamic provisioning** lets Kubernetes **create PVs automatically** when a PVC is created.
- This is powered by the **StorageClass** resource.

What Is a StorageClass?

A StorageClass defines how Kubernetes should provision storage:

- Which provisioner/driver to use (e.g., Google PD, AWS EBS, CSI driver)
- What parameters to pass (disk type, performance tier, etc.)
- What reclaim policy to apply after use

StorageClass

```yaml
apiVersion: storage.k8s.io/v1
kind: StorageClass
metadata:
 name: fast
provisioner: kubernetes.io/gce-pd
parameters:
 type: pd-ssd
reclaimPolicy: Delete
volumeBindingMode: WaitForFirstConsumer
```

PVC with StorageClass

```yaml
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
 name: pvc-dynamic
spec:
 accessModes:
   - ReadWriteOnce
 resources:
   requests:
     storage: 5Gi
 storageClassName: fast
```

Access Modes Explained

Mode	Behavior
Immediate	PV is created as soon as PVC appears (default in older clusters)
WaitForFirstConsumer	PV is created only when a Pod using the PVC is scheduled (avoids zone mismatch)

Access Modes Explained

Aspect	Static Provisioning	Dynamic Provisioning
Who creates PV	Admin manually	Kubernetes automatically
When created	Before PVC	When PVC appears
Scale	Manual, limited	Scales with demand
Typical use	Legacy setups, NFS	Cloud storage, CSI drivers

Common Provisioners (CSI Drivers)

Environment	Provisioner Name
Google Cloud	pd.csi.storage.gke.io
AWS	ebs.csi.aws.com
Azure	disk.csi.azure.com
vSphere	csi.vsphere.vmware.com
OpenEBS	openebs.io/local

Secrets

What Is a Secret?

- A Kubernetes object that stores small amounts of sensitive data —
- such as passwords, tokens, SSH keys, or certificates.
- Designed to separate confidential information from container images and configuration.
- Mounted or injected at runtime (never baked into images).

Why Secrets Matter?

Without Secrets:

- Developers hardcode passwords or API keys in manifests.
- Anyone with access to a repository can see them.
- Revoking or rotating credentials becomes manual and error-prone.

With Secrets:

- Sensitive values stored securely in the cluster.
- Controlled access through **RBAC**.
- Easy to rotate, mount, or inject when needed.

How Kubernetes Stores Secrets

- Stored as **Base64-encoded** key-value pairs inside the etcd database.
- If etcd is encrypted (recommended), Secrets are stored **encrypted at rest**.
- Only the **API server** and authorized users can access them.

Secret Structure With Base64

```yaml
apiVersion: v1
kind: Secret
metadata:
 name: my-secret
type: Opaque
data:
 username: YWRtaW4=
 password: cGFzc3dvcmQ=
```

Secret Structure With string

```yaml
apiVersion: v1
kind: Secret
metadata:
 name: my-secret-string
type: Opaque
stringData:
 username: admin
 password: password
```

Access Modes Explained

Type	Purpose
Opaque	Default type for arbitrary key-value pairs
kubernetes.io/dockerconfigjson	Stores credentials for private container registries
kubernetes.io/tls	Holds TLS certificates and private keys
bootstrap.kubernetes.io/token	Used for node bootstrapping / kubeadm
kubernetes.io/service-account-token	Automatically created for service accounts

Creating Secrets

Option 1: From Literal Values

```
kubectl create secret generic db-secret \
  --from-literal=username=admin \
  --from-literal=password=s3cr3t
```

Option 2: From Files

```
kubectl create secret generic ssl-cert \
  --from-file=cert.pem \
  --from-file=key.pem
```

Option 3: From Manifest

```
kubectl apply -f secret.yaml
```

Secret Best Practices

- Encrypt etcd at rest (--encryption-provider-config).
- Restrict access using RBAC (e.g., get, list only for necessary roles).
- Avoid storing secrets in Git in plaintext.
- Rotate regularly — Secrets are immutable, so updates create new resource versions.
- Use tools like Sealed Secrets, External Secrets Operator, or Vault CSI for enhanced management.

How Kubernetes Delivers Secrets

1. As environment variables (available to the process)
2. As mounted volumes (available as files)

Using Secrets via Environment Variables

Why Use Environment Variables?

- Many applications already support environment-based configuration.
- Makes secrets accessible directly to the process (no filesystem mount needed).
- Perfect for small credentials like API keys, usernames, passwords.

Injecting a Single Secret Key as an Environment Variable

```yaml
apiVersion: v1
kind: Pod
metadata:
 name: env-secret-demo
spec:
 containers:
 - name: app
   image: busybox
   command: ['sh', '-c', 'echo
"User: $DB_USER, Pass: $DB_PASS" &&
sleep 3600']
   env:
   - name: DB_USER
     valueFrom:
       secretKeyRef:
         name: db-secret
         key: username
   - name: DB_PASS
     valueFrom:
       secretKeyRef:
         name: db-secret
         key: password
```

Injecting Entire Secret as Multiple Variables

```yaml
apiVersion: v1
kind: Pod
metadata:
 name: envfrom-secret-demo
spec:
 containers:
 - name: app
   image: busybox
   command: ['sh', '-c', 'env |
grep DB_ && sleep 3600']
   envFrom:
   - secretRef:
       name: db-secret
```

Best Practices for Environment Secrets

Do

- Use short-lived Pods or rolling updates after secret rotation.
- Prefer envFrom for grouped secrets, env for precise control.
- Keep secret names and variable prefixes consistent (e.g., DB_, AWS_).

Avoid

- Logging environment variables in your app (common leak!).
- Using overly broad RBAC rules (like get secrets for entire namespaces).
- Storing large binary data in Secrets — use volumes instead.

Mounting Secrets as Volumes

Why Use Volumes for Secrets?

- Some applications require **files**, not environment variables (e.g., TLS certs, SSH keys, service credentials).
- Mounting Secrets as volumes allows them to appear as **regular files** inside containers.
- Safer than embedding secrets in images or configs — access is controlled by Kubernetes.

Mounting Secrets into a Pod

```yaml
apiVersion: v1
kind: Pod
metadata:
 name: secret-vol-demo
spec:
 volumes:
 - name: creds
   secret:
     secretName: api-keys
 containers:
 - name: app
   image: busybox
   command: ["sh", "-c", "ls -l
/etc/creds && sleep 3600"]
   volumeMounts:
   - name: creds
     mountPath: /etc/creds
     readOnly: true
```

The Referenced Secret

```yaml
apiVersion: v1
kind: Secret
metadata:
 name: api-keys
type: Opaque
stringData:
 api-key: 12345-abcde
 api-token: ZXhhbXBsZS10b2tlbg==
```

Pod Mounting Only Selected Keys - Secret

```yaml
apiVersion: v1
kind: Secret
metadata:
 name: tls-secret
type: kubernetes.io/tls
data:
 tls.crt:
LS0tLS1CRUdJTiBDRVJUSUZJQ0FURS0t
LS0tCgpoZWxsbyB3b3JsZAo=    #
base64 encoded cert
 tls.key:
LS0tLS1CRUdJTiBSU0EgUFJJVkFURSBL
RVktLS0tLQoKc29tZSByYW5kb20ga2V5
Cg==
 ca.crt:
LS0tLS1CRUdJTiBDQSBURVNUIENFUlQt
LS0tLQoKZm9yIGRlbW8gcHVycG9zZXMK
```

Pod Mounting Only Selected Keys

```yaml
apiVersion: v1
kind: Pod
metadata:
 name: tls-reader
spec:
 volumes:
 - name: tls-certs
   secret:
     secretName: tls-secret
     items:
       - key: tls.crt
         path: server.crt
       - key: tls.key
         path: server.key
 containers:
 - name: web
   image: nginx:latest
   volumeMounts:
     - name: tls-certs
       mountPath: "/etc/nginx/ssl"
       readOnly: true
   ports:
     - containerPort: 443
```

How Secrets Are Actually Stored on the Node

- When a Pod using a Secret starts, the kubelet on the node:
  - Retrieves the Secret from the API server.
  - Writes it into its internal directory:

    `/var/lib/kubelet/pods/<pod-id>/volumes/kubernetes.io~secret/<secret-name>/`

  - Bind-mounts that directory into the container (e.g., /etc/creds).
- The data lives on the node's filesystem, not inside the container's memory.
- In some setups, kubelet may use tmpfs (RAM) for these mounts — but this is implementation-dependent, not guaranteed.

File Permissions and Customization

- Default file mode = 0644 (owner read/write, world read).
- You can make it stricter:

```
secret:
  secretName: db-secret
  defaultMode: 0400
```

- Files are always read-only inside containers.
- Changing permissions helps prevent accidental leaks from non-root users.

Automatic Updates

- When the underlying Secret is updated in Kubernetes:
  - kubelet detects the change,
  - updates the mounted files automatically (within ~1 minute).
- No Pod restart needed.
- Great for **cert rotation** or **API token refresh**.

Best Practices for Secret Volumes

Recommended:

- Use Secret volumes for TLS, SSH, and file-based credentials.
- Mount only required keys using items.
- Set restrictive file modes (e.g., 0400).
- Keep namespaces and Secrets scoped per app/service.
- Use RBAC to restrict read access to Secrets.

Avoid:

- Logging or copying Secret files to writable directories.
- Sharing the same Secret across unrelated applications.
- Assuming secrets are “hidden” — they’re visible inside Pods.

Portable Data with ConfigMaps

What is a ConfigMap?

- A Kubernetes object for storing **non-sensitive configuration data**.
- Key-value pairs such as:
  - feature flags,
  - log levels,
  - service URLs,
  - runtime tuning parameters.
- Lets you change behavior of an app **without rebuilding the container image**.

Why Not Just Put Config in the Image?

Hardcoding config in the image:

- Requires rebuilds for every environment (dev / test / prod).
- Makes debugging and rollback noisy.
- Leaks environment-specific details into artifacts.

Externalizing config with ConfigMaps:

- You ship **one image** to all environments.
- Each environment just has a different ConfigMap.
- You can roll back config without rolling back code.

ConfigMap - Key Values

```
apiVersion: v1
kind: ConfigMap
metadata:
 name: app-config
data:
 LOG_LEVEL: info
 FEATURE_X_ENABLED: "true"
 BACKEND_URL:
"http://backend.default.svc.clus
ter.local"
```

Multiline ConfigMap

```
apiVersion: v1
kind: ConfigMap
metadata:
 name: nginx-config
data:
 nginx.conf: |
   user  nginx;
   worker_processes  auto;

   events {
       worker_connections 1024;
   }

   http {
       server {
           listen 80;
           location / {
               root /usr/share/nginx/html;
               index index.html;
           }
       }
   }
```

Creating a ConfigMap

Option 1: From literal values

```
kubectl create configmap app-config \
  --from-literal=LOG_LEVEL=debug \
  --from-literal=FEATURE_X_ENABLED=true
```

Option 2: From file(s)

```
kubectl create configmap app-config \
  --from-file=config.yaml
```

Option 3: From YAML manifest

```
kubectl apply -f app-config.yaml
```

Access Modes Explained

Aspect	ConfigMap	Secret
Intended for	Non-sensitive config	Sensitive data (passwords, tokens)
Storage format	Plain text in etcd	Base64-encoded in etcd (can be encrypted at rest)
Access control	RBAC still applies	RBAC typically stricter
Typical usage	Feature flags, URLs, tunables	DB creds, API keys, TLS material

Why ConfigMaps Improve Portability

- You can promote the same image through dev → staging → prod.
- Only configuration objects differ between environments.
- You can version and review configuration in Git.
- Teams can roll out config changes without touching application code.

This is directly aligned with 12-factor principles (separate config from code).

Operational Notes

- ConfigMaps live in a **namespace**. They're not cluster-global.
- If you update a ConfigMap:
  - Pods that read it as ENV VARS do not automatically restart.
  - Pods that mount it as a volume can see updates on disk (similar to Secrets) — we'll cover that next.
- Access to ConfigMaps is still protected by RBAC — not everyone can get them by default.

Using ConfigMaps

Using ConfigMaps in Pods

1. Injecting configuration as **environment variables**
2. Mounting configuration as **files**

Using ConfigMaps as Environment Variables

Individual Keys

```yaml
apiVersion: v1
kind: Pod
metadata:
 name: cm-env-demo
spec:
 containers:
 - name: app
   image: busybox
   command: ["sh", "-c", "echo $LOG_LEVEL
$FEATURE_X_ENABLED && sleep 3600"]
   env:
     - name: LOG_LEVEL
       valueFrom:
         configMapKeyRef:
           name: app-config
           key: LOG_LEVEL
     - name: FEATURE_X_ENABLED
       valueFrom:
         configMapKeyRef:
           name: app-config
           key: FEATURE_X_ENABLED
```

Using ConfigMaps as Environment Variables

Inject All Keys at Once

```
apiVersion: v1
kind: Pod
metadata:
 name: cm-envfrom-demo
spec:
 containers:
 - name: app
   image: busybox
   command: ["sh", "-c", "env |
grep LOG && sleep 3600"]
   envFrom:
     - configMapRef:
         name: app-config
```

Using ConfigMaps as Files

```yaml
apiVersion: v1
kind: Pod
metadata:
 name: cm-vol-demo
spec:
 volumes:
 - name: app-config-vol
   configMap:
     name: nginx-config
 containers:
 - name: nginx
   image: nginx:latest
   volumeMounts:
     - name: app-config-vol
       mountPath: /etc/nginx/conf.d
       readOnly: true
```

Automatic Updates (Same Behavior as Secrets)

- When the Config Map is updated:
  - Pods using it **as a volume** will see changes automatically within ~1 minute.
  - Pods using it **as environment variables** will **not** see changes until restart.
- Good for dynamic configs (e.g., feature toggles, log levels).

Best Practices for ConfigMaps

Do:

- Keep ConfigMaps environment-specific (e.g. *app-config-dev*, *app-config-prod*).
- Combine related settings into one ConfigMap per app.
- Use GitOps and version control for managing ConfigMaps.
- Pair with Secrets for credentials (don’t mix sensitive and non-sensitive data).

Avoid:

- Putting huge binary data in ConfigMaps.
- Overwriting ConfigMaps used by running apps without testing.
- Using ConfigMaps as global storage — they are namespaced.

Deployment Configuration Status

Deployment Configuration Status

- A controller that ensures the desired number of Pod replicas are running and up-to-date.
- Defines:
  - The template for Pods (image, labels, environment, etc.),
  - The strategy for updating Pods,
  - The history of revisions (rollbacks possible).
- Handles creating, replacing, and deleting Pods automatically.

How Kubernetes Tracks Deployment State

- Every Deployment defines a desired state — how many replicas, which image, etc.
- The Kubernetes controller continuously compares it with the observed state.
- If differences are found:
  - new Pods are created,
  - old ones are terminated,
  - the status is updated in the API.

Deployment Status Conditions

Field	Meaning
availableReplicas	Number of ready Pods available to serve traffic
readyReplicas	Pods passing readiness checks
updatedReplicas	Pods created by the latest Deployment version
replicas	Desired total number of Pods
conditions	Overall health and rollout progress

Scaling and Rolling Updates

Why Scaling Matters?

- Kubernetes Deployments are designed to **scale horizontally** — by running more Pod replicas.
- Scaling improves **availability**, **resilience**, and **throughput**.
- Scaling up or down can be:
  - **manual** (operator-driven),
  - or **automatic** (via HorizontalPodAutoscaler).

Manual Scaling

Scaling via kubectl

```
kubectl scale deployment web --replicas=5
```

Automatic Scaling (HPA)

- Monitors metrics (e.g., CPU, memory, custom metrics).
- Automatically increases or decreases replicas based on load.
- Requires metrics server.

HorizontalPodAutoscaler

```yaml
apiVersion: autoscaling/v2
kind: HorizontalPodAutoscaler
metadata:
 name: web-hpa
spec:
 scaleTargetRef:
   apiVersion: apps/v1
   kind: Deployment
   name: web
 minReplicas: 2
 maxReplicas: 10
 metrics:
 - type: Resource
   resource:
     name: cpu
     target:
       type: Utilization
       averageUtilization: 70
```

How Kubernetes Updates Apps “Without Downtime”?

RollingUpdate Strategy

- Default Deployment strategy in Kubernetes.
- Updates Pods **incrementally**, not all at once.
- Maintains service availability during rollout.
- Controlled by:
  - maxSurge — how many extra Pods can run during update,
  - maxUnavailable — how many Pods can be temporarily down.

RollingUpdate

```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
 name: web
spec:
 replicas: 4
 strategy:
   type: RollingUpdate
   rollingUpdate:
     maxUnavailable: 1
     maxSurge: 1
 template:
   spec:
     containers:
     - name: app
       image: myapp:v2
```

Best Practices for Scaling and Updates

Do

- Use RollingUpdate as default for production workloads.
- Set readinessProbe properly to avoid serving traffic too early.
- Tune maxSurge and maxUnavailable based on SLA.
- Combine with HPA for elasticity.

Avoid

- Setting both maxSurge and maxUnavailable to high values — risk overload.
- Relying on Recreate in multi-tenant or high-availability systems.
- Forgetting to monitor rollout progress — updates can stall.

Deployment Rollbacks

Why Rollbacks Matter?

- Even with rolling updates, not all updates go smoothly:
  - readinessProbe never passes,
  - image is broken,
  - bad configuration causes crashes.
- Kubernetes stores Deployment history, letting you quickly roll back.

How Rollbacks Work Internally

- Each Deployment update creates **a new ReplicaSet** (revision).
- Kubernetes keeps old ReplicaSets (by default up to 10).
- A rollback simply:
  - pauses the current rollout,
  - reverts .spec.template to the previous ReplicaSet's spec,
  - updates d*eployment.kubernetes.io/revision*.

Checking Deployment History

View History:

```
kubectl rollout history deployment web
```

Example output:

```
deployment.apps/web
REVISION  CHANGE-CAUSE
1         Initial deploy
2         Updated image to v2
3         Updated config for feature X
```

Setting Change Causes (for History Clarity)

Adding Context with Annotations

```
kubectl annotate deployment web \
  kubernetes.io/change-cause="Updated image to v3 - bug fix"
```

- Appears in *rollout history* output.
- Helps track *why* each version was deployed.
- Very useful in GitOps or multi-team environments.

Performing a Rollback

Rollback to Previous Revision

```
kubectl rollout undo deployment web
```

Reverts to the previous ReplicaSet (e.g., from rev 3 → rev 2).

You can target a specific revision:

```
kubectl rollout undo deployment web --to-revision=1
```

Observing Rollback Progress

```
kubectl rollout status deployment web
```

- Works exactly like during a rollout.
- Kubernetes gradually replaces Pods with the previous version.
- You can watch Pods update live (kubectl get pods -w).

Controlling History Size

```
spec:
  revisionHistoryLimit: 5
```

- Controls how many old ReplicaSets are kept.
- Helps reduce resource usage.
- When limit is exceeded, oldest ReplicaSets are garbage-collected.

Pausing and Resuming Deployments

Pause Rollout

```
kubectl rollout pause deployment web
```

Resume Rollout

```
kubectl rollout resume deployment web
```

- Useful when testing partial updates or debugging rollout issues.
- You can make changes safely before resuming rollout progression.

Best Practices for Safe Rollbacks

Do:

- Always set *kubernetes.io/change-cause* for traceability.
- Use *progressDeadlineSeconds* to auto-fail bad rollouts.
- Keep a reasonable *revisionHistoryLimit* (5–10).
- Validate rollback results before declaring “success.”

Avoid:

- Manually deleting old ReplicaSets — breaks rollback chain.
- Rolling back without investigating the root cause.
- Forgetting to reapply new annotations after rollback (causes history gaps).

Labs

Security

- Security Overview
- Accessing the API
- Authentication
- Authorization
- ABAC
- RBAC
- RBAC Process Overview
- Admission Controller
- Security Contexts
- Pod Security Policies
- Network Security Policies
- Network Security Policy Example
- Default Policy Example
- Labs

Security Overview

What Does “Security” Mean in Kubernetes?

Kubernetes security covers multiple layers, not just user access.

It involves securing:

- The API Server — who can talk to the control plane.
- Workloads — what containers and Pods can do.
- The Network — which Pods can talk to each other.
- The Supply Chain — ensuring trusted images and manifests.

Deployment Status Conditions

Layer	Focus Area	Examples
API Access	Who can talk to the cluster	Authentication, RBAC
Workload	What Pods can do	SecurityContext, PodSecurity
Network	Who can talk to whom	NetworkPolicies
Runtime & Supply Chain	What runs and how	Image signing, Admission Controllers

Kubernetes Security Responsibilities

Developers:

- Avoid running containers as root
- Limit container privileges
- Store secrets safely

Cluster Operators:

- Manage authentication & RBAC
- Apply PodSecurity standards
- Enforce admission policies

Infrastructure / Platform Team:

- Secure API server and etcd
- Patch the control plane
- Configure audit logs and encryption

Threat Vectors in Kubernetes

Vector	Example Attack or Misuse
Misconfigured RBAC	Unauthorized access to resources
Privileged Containers	Container breakout to the host
Insecure Images	Malware or exposed secrets in images
Unrestricted Network	Lateral movement between namespaces
Unsecured API Access	Tokens or kubeconfigs leaked
Lack of Monitoring	Breach detection delays

Accessing the API

How clients talk to the API server

- Tools: kubectl, controllers, dashboards, CI agents, custom scripts.
- Connectivity options:
  - Direct to API server (https://<apiserver>:6443) — requires network access & auth.
  - Via kubectl proxy (local HTTP proxy to the API).
  - Via kubectl port-forward to reach a Pod that talks to API.
- Authentication methods (summary):
  - TLS client certificates
  - Bearer tokens (service account tokens, static tokens)
  - OpenID Connect (OIDC) / external identity providers
  - Webhook token authentication

Common API Endpoints You Can Curl

Examples of useful endpoints:

- GET /api → core API groups & versions.
- GET /apis → list of API groups.
- GET /api/v1/namespaces → list namespaces.
- GET /api/v1/namespaces/default/pods → pods in default ns.
- GET /version → server version.
- GET /healthz → basic health.
- GET /openapi/v2 → OpenAPI spec.

Access via kubectl proxy

Start proxy (local, no TLS hassles)

```
kubectl proxy --port=8001
# then from same machine:
curl http://127.0.0.1:8001/api
```

Why use it

- Uses your current kubectl kubeconfig/auth.
- No need to manage certs or tokens in the curl command.
- Good for interactive exploration and demos.

Direct curl to API Server (with Bearer token)

1. Obtain a token (example for a ServiceAccount)

```
# Create SA

kubectl create sa demo-sa -n default
# For newer kubectl: create token
kubectl create token demo-sa -n default
# OR (older clusters): get secret token name then decode
SECRET=$(kubectl get sa/demo-sa -n default -o jsonpath='{.secrets[0].name}')
kubectl get secret "$SECRET" -n default -o go-template='{{ index .data "token" }}' | base64 -d
```

2. Curl the API (skip TLS verification only for demos)

```
APISERVER="https://your-apiserver:6443"
TOKEN="<the-token>"
curl -k -H "Authorization: Bearer ${TOKEN}" \
 -H "Accept: application/json" \
 "${APISERVER}/api/v1/namespaces/default/pods"
```

Direct curl to API Server (with client certs)

```
curl --cert /path/client.crt --key /path/client.key \
  --cacert /path/ca.crt \
  "https://your-apiserver:6443/api"
```

When to use:

- Clusters issued client certificates for users/clients.
- Mutual TLS gives strong authentication but requires cert lifecycle management.

Practical Tips & Best Practices

- **Avoid putting tokens on the shell command line** (shell history). Use files or kubectl proxy.
- Prefer **short-lived tokens** (service account tokens via kubectl create token or OIDC).
- Use **--cacert** (do not use -k) in production to validate server certs.
- Use kubectl proxy for local debugging to avoid exposing tokens.
- Audit API access: enable audit logs in API server to see who called what.
- Limit scopes: give service accounts minimal RBAC permissions (least privilege).

ServiceAccount

What Is a ServiceAccount?

A ServiceAccount (SA) is an identity used by processes running inside the cluster to authenticate to the Kubernetes API.

- Every Pod runs under a ServiceAccount context.
- ServiceAccounts are meant for machines, Pods, or controllers — not human users.
- They’re essential for fine-grained access control via RBAC.

How ServiceAccounts Work

- Each ServiceAccount has an associated token (JWT).
- The token is mounted inside the Pod at:

  `/var/run/secrets/kubernetes.io/serviceaccount/token`

- When the Pod contacts the API server, it uses this token as:

  `Authorization: Bearer <token>`

- This mechanism authenticates the Pod and determines its permissions.

ServiceAccount

```yaml
apiVersion: v1
kind: ServiceAccount
metadata:
 name: app-sa
 namespace: dev
```

Service Account attached to pod

```yaml
apiVersion: v1
kind: Pod
metadata:
 name: demo-pod
 namespace: dev
spec:
 serviceAccountName: app-sa
 containers:
 - name: app
   image: busybox
   command: ["sh", "-c", "cat
/var/run/secrets/kubernetes.io/s
erviceaccount/token && sleep
3600"]
```

ServiceAccount

What is Authentication in Kubernetes?

- Authentication = proving your identity to the API server.
- Every API request must include credentials.
- If authentication fails → 401 Unauthorized (request rejected immediately).
- If authentication succeeds → request moves on to authorization.

Types of Clients That Authenticate

- Human users
  - cluster admins
  - developers / operators
  - CI/CD engineers
- Service accounts (in-cluster identities)
  - Pods talking to the API
  - Controllers / operators / system components
- Cluster components
  - kubelet on each node
  - controller-manager, scheduler, etc.

How Authentication Works

1. Client sends request to API server with some credential (token, client cert, etc.).
2. The API server runs through one or more authenticators:
    a. Token authenticator
    b. Client certificate authenticator
    c. Webhook / OIDC authenticator
    d. Anonymous (if allowed)
3. If any authenticator accepts → request is considered authenticated.
4. Server assigns a user and groups to that request.

User Accounts vs Service Accounts

User Accounts:

- Represent humans.
- Not stored as Kubernetes objects — there is no User resource.
- Often managed externally:
  - OIDC / SSO (e.g. Dex, Keycloak, Azure AD, etc.)
  - Client certs issued by cluster admin
  - Static tokens in kubeconfig (not recommended long-term)

Service Accounts:

- Represent workloads / Pods.
- Are real Kubernetes objects: kind: ServiceAccount.
- Automatically mounted into Pods as tokens (so the Pod can call the API).
- Used heavily by controllers / operators.

Authorization

Are You Allowed?

- Authorization = permission check after authentication.
- Evaluates who can do what on which resource in which namespace.
- If denied → 403 Forbidden.

How Authorization Works

1. Request reaches API Server (already authenticated).
2. The server sends the request to one or more authorizers:
    a. RBAC
    b. ABAC
    c. Webhook
    d. Node authorizer (for kubelets)
3. If any authorizer approves → the request is allowed.
4. If none approve → 403 Forbidden.

Authorization Modes

Mode	Description	Status
RBAC (Role-Based Access Control)	Assigns roles to users or groups	Standard
ABAC (Attribute-Based Access Control)	Uses JSON policy files with conditions	Legacy
Webhook	External service makes the decision	Advanced / custom
Node Authorizer	Special authorizer for kubelet API access	Internal only

ABAC

What Is ABAC?

- ABAC = Attribute-Based Access Control.
- Uses rules based on attributes of:
  - the user (identity),
  - the requested action (verb),
  - the target object (resource, namespace).
- Authorization is granted when all attributes match a defined policy.

How ABAC Works in Kubernetes

1. Request arrives at the API Server (already authenticated).
2. Server compares the request against static policy files on disk (--authorization-policy-file).
3. Each policy file contains JSON documents describing allowed actions.
4. If a rule matches → request allowed; otherwise → 403 Forbidden.

ABAC Policy File Example

```
{"apiVersion": "abac.authorization.kubernetes.io/v1beta1", "kind": "Policy", "spec":
{"user": "alice", "namespace": "*", "resource": "*", "apiGroup": "*"}}

{"apiVersion": "abac.authorization.kubernetes.io/v1beta1", "kind": "Policy", "spec":
{"user": "bob", "namespace": "projectCaribou", "resource": "pods", "readonly": true}}

{"apiVersion": "abac.authorization.kubernetes.io/v1beta1", "kind": "Policy", "spec":
{"group": "system:authenticated", "readonly": true, "nonResourcePath": "*"}}

{"apiVersion": "abac.authorization.kubernetes.io/v1beta1", "kind": "Policy", "spec":
{"group": "system:unauthenticated", "readonly": true, "nonResourcePath": "*"}}
```

Enabling ABAC (for legacy clusters)

```
kube-apiserver \
  --authorization-mode=ABAC \
  --authorization-policy-file=/etc/kubernetes/policy.jsonl
```

Advantages and Limitations of ABAC

Advantages

- Simple and explicit.
- No need for additional objects in etcd.
- Works offline (just local file).

Limitations

- Requires API Server restart to apply changes.
- Not scalable — no namespace delegation or team self-service.
- No inheritance, no grouping, no audit metadata.
- Hard to audit or manage in GitOps workflows.

Why ABAC Was Replaced by RBAC

Reasons for the Shift

- Static files didn't fit dynamic cloud-native workflows.
- Enterprises needed namespace-scoped, group-based, auditable control.
- GitOps and automation required API-native access rules.

RBAC introduced:

- declarative YAML objects (Role, RoleBinding, etc.),
- in-cluster storage in etcd,
- instant updates without restarting API Server.

RBAC

The Modern Kubernetes Authorization Model

- RBAC = Role-Based Access Control
- Introduced to make authorization dynamic, API-native, and manageable at scale.
- Works by defining roles (what actions are allowed) and bindings (who gets those roles).

RBAC Core Concepts

Object	Scope	Purpose
Role	Namespaced	Defines allowed verbs on specific resources
ClusterRole	Cluster-wide	Same as Role but applies to all namespaces
RoleBinding	Namespaced	Grants Role to a user/group/service account
ClusterRoleBinding	Cluster-wide	Grants ClusterRole to a user/group/service account

RBAC in Action: The 3 Questions

Each RBAC decision answers:

1. **Who** — the subject (user, group, or service account)
2. **Can perform what** — verbs like get, create, delete
3. **On which resources** — e.g. pods, deployments, secrets

Typical Verbs in RBAC

get, list, watch, create, update, patch, delete, deletecollection, impersonate

1. Read-only: get, list, watch
2. Write: create, update, patch, delete
3. Special verbs:
    a. impersonate (rare; lets one user act as another)
    b. escalate (for admins managing RBAC itself)

Namespaced Role and RoleBinding

```yaml
apiVersion: rbac.authorization.k8s.io/v1
kind: Role
metadata:
 name: pod-reader
 namespace: default
rules:
 - apiGroups: [""]
   resources: ["pods"]
   verbs: ["get", "list", "watch"]
---
apiVersion: rbac.authorization.k8s.io/v1
kind: RoleBinding
metadata:
 name: read-pods
 namespace: default
subjects:
 - kind: ServiceAccount
   name: demo-sa
   namespace: default
roleRef:
 kind: Role
 name: pod-reader
 apiGroup: rbac.authorization.k8s.io
```

ClusterRole and ClusterRoleBinding

```yaml
apiVersion: rbac.authorization.k8s.io/v1
kind: ClusterRole
metadata:
 name: cluster-pod-reader
rules:
 - apiGroups: [""]
   resources: ["pods"]
   verbs: ["get", "list", "watch"]
---
apiVersion: rbac.authorization.k8s.io/v1
kind: ClusterRoleBinding
metadata:
 name: cluster-read-pods
subjects:
 - kind: User
   name: alice@example.com
   apiGroup: rbac.authorization.k8s.io
roleRef:
 kind: ClusterRole
 name: cluster-pod-reader
 apiGroup: rbac.authorization.k8s.io
```

RBAC Core Concepts

Name	Scope	Purpose
cluster-admin	Cluster-wide	Full access to everything (superuser)
admin	Namespaced	Manage most resources in one namespace
edit	Namespaced	Read/write most resources, not RBAC
view	Namespaced	Read-only access

RBAC Best Practices

Do:

- Use namespace-scoped Roles whenever possible.
- Group permissions logically (per app/team).
- Keep roles small and specific (few resources per rule).
- Audit and review ClusterRoleBindings regularly.
- Store RBAC manifests in Git for version control.

Avoid:

- Assigning cluster-admin unless absolutely necessary.
- Using * for resources or verbs (verbs: ["*"]).
- Letting service accounts share credentials between Pods.

RBAC Process Overview

RBAC Decision Flow: Step by Step

1. Authentication → API server identifies the caller (user / service account / group).
2. Authorization (RBAC phase) → API server checks if that identity is allowed to perform the requested action.
3. Evaluation chain:
    a. Collect all Roles & RoleBindings from the request's namespace.
    b. Collect all ClusterRoles & ClusterRoleBindings (global).
    c. Combine all rules that match:
    d. *user* or any of the user's groups, or
    e. serviceaccount:<namespace>:<name>.
4. Check if any rule allows the requested:
    a. verb (e.g. get, list, create),
    b. resource,
    c. namespace (if applicable).
5. If found → allow.
6. If not → 403 Forbidden.

RBAC Is Additive, Not Subtractive

- Permissions accumulate — more RoleBindings = more access.
- There is no “deny” rule in RBAC.
- To restrict access:
  - remove bindings, or
  - use Admission Controllers or OPA Gatekeeper for explicit deny logic.

Admission Controller

Admission Controllers: The Final Gate

- Admission Controllers are plugins in the API Server.
- They intercept requests after authentication & authorization, but before persistence.
- Used to modify, validate, or reject API requests.

Two Main Types

Type	Purpose	Example Use Case
Mutating Admission Controller	Modifies objects before they're stored	Inject sidecar, add default labels, add resource limits
Validating Admission Controller	Approves or denies requests based on policy	Enforce naming conventions, block privileged Pods

Two Main Types

Controller	Purpose
NamespaceLifecycle	Prevents deleting active namespaces
LimitRanger	Enforces resource request/limit policies
ResourceQuota	Ensures namespace quotas aren’t exceeded
PodSecurity (replaces PodSecurityPolicy)	Enforces Pod security standards (restricted, baseline, privileged)
ServiceAccount	Auto-injects ServiceAccount tokens into Pods
DefaultStorageClass	Automatically assigns default StorageClass
MutatingAdmissionWebhook	Calls custom webhook for mutation
ValidatingAdmissionWebhook	Calls custom webhook for validation

Admission Controller Workflow

- Request is authenticated and authorized
- Mutating controllers run first e.g., add defaults, inject labels, or mutate fields
- Validating controllers run next e.g., check if object violates security or quota
- If approved → object is persisted in etcd

Why Custom Webhooks?

- Built-in controllers handle generic use cases.
- You can extend behavior for your organization (security, compliance, naming).

Two webhook types:

1. MutatingAdmissionWebhook → modifies incoming objects.
2. ValidatingAdmissionWebhook → enforces rules.

Validating Webhook Example

```yaml
apiVersion: admissionregistration.k8s.io/v1
kind: ValidatingWebhookConfiguration
metadata:
 name: deny-privileged
webhooks:
 - name:
deny-privileged.policies.example.com
   admissionReviewVersions: ["v1"]
   sideEffects: None
   rules:
     - apiGroups: [""]
       apiVersions: ["v1"]
       operations: ["CREATE", "UPDATE"]
       resources: ["pods"]
   clientConfig:
     service:
       namespace: webhook-system
       name: webhook-server
       path: /validate
```

Webhook Response Example

```
{
 "apiVersion": "admission.k8s.io/v1",
 "kind": "AdmissionReview",
 "response": {
   "uid": "1a2b3c",
   "allowed": false,
   "status": {
     "message": "Privileged pods are not allowed"
   }
 }
}
```

Best Practices for Admission Control

Do:

- Keep Admission Controllers enabled for security (NamespaceLifecycle, ResourceQuota, LimitRanger, PodSecurity).
- Use validating webhooks for custom policy enforcement.
- Test webhooks in a staging cluster before rollout.
- Ensure webhook servers are **highly available** (API requests depend on them).
- Combine with **OPA Gatekeeper** or **Kyverno** for policy as code.

Avoid:

- Disabling admission controllers for convenience.
- Using webhooks that have latency or unavailability — they block all API writes.
- Forgetting to version webhooks when API versions change.

Security Contexts

Security Contexts: Local Pod-Level Security

- A SecurityContext defines privilege and access settings for a Pod or container.
- Enforces security inside the node, complementing RBAC and Network Policies.
- You can apply it at:
  - Pod level (spec.securityContext)
  - or per container (containers[].securityContext)

Two Main Types

Scope	Example Field	Description
Pod-level	fsGroup, runAsUser, runAsGroup, supplementalGroups	Affects all containers in the Pod
Container-level	privileged, allowPrivilegeEscalation, readOnlyRootFilesystem, capabilities	Affects only that container

Basic SecurityContext (Container-Level)

```yaml
apiVersion: v1
kind: Pod
metadata:
 name: restricted-pod
spec:
 containers:
 - name: app
   image: busybox
   command: ["sh", "-c", "id &&
sleep 3600"]
   securityContext:
     runAsUser: 1000
     runAsGroup: 3000
     privileged: false
     allowPrivilegeEscalation:
false
     readOnlyRootFilesystem: true
```

Shared SecurityContext

```yaml
apiVersion: v1
kind: Pod
metadata:
 name: pod-level-security
spec:
 securityContext:
   runAsNonRoot: true
   runAsUser: 2000
   fsGroup: 3000
 containers:
 - name: api
   image: nginx
 - name: sidecar
   image: busybox
   command: ["sh", "-c", "ls -l /shared"]
   volumeMounts:
     - name: shared-data
       mountPath: /shared
 volumes:
   - name: shared-data
     emptyDir: {}
```

Linux Capabilities

- Fine-grained kernel privileges that can be granted or removed.
- Instead of full root (UID 0), capabilities give minimal required powers.
- Example:
  - CAP_NET_ADMIN → manage network interfaces
  - CAP_SYS_TIME → set system clock
  - CAP_SYS_ADMIN → broad admin privileges (avoid!)

Linux Capabilities

```
securityContext:
 capabilities:
   drop: ["ALL"]

securityContext:
 capabilities:
   add: ["NET_BIND_SERVICE"]
```

Privileged Containers

```
securityContext:
  privileged: true
```

- Gives container almost full root access to the host.
- Can access /dev, load kernel modules, modify networking.
- Should be reserved only for trusted infrastructure components (e.g., CNI plugins, device drivers).

allowPrivilegeEscalation

```
securityContext:
  allowPrivilegeEscalation: false
```

- Prevents a process from gaining more privileges than its parent.
- Even if binary has setuid bit, it won't escalate.

readOnlyRootFilesystem

```
securityContext:
  readOnlyRootFilesystem: true
```

- Root filesystem is mounted read-only.
- Application must write to /tmp or mounted volumes.
- Great for preventing tampering, persistence, and malware activity.

runAsNonRoot / runAsUser / runAsGroup

```yaml
securityContext:
  runAsNonRoot: true
  runAsUser: 1001
  runAsGroup: 1001
```

- Ensures container isn't started as UID 0.
- If the image defaults to root, the Pod will fail to start.
- Recommended for all application workloads.

fsGroup and supplementalGroups

```yaml
securityContext:
  fsGroup: 2000
  supplementalGroups: [3000, 4000]
```

- fsGroup: ensures volumes are writable by a specific group.
- supplementalGroups: adds secondary groups for container processes.
- Especially useful when sharing PVCs between Pods.

Practical Guidelines

Do:

- Use runAsNonRoot everywhere.
- Drop all capabilities by default.
- Use readOnlyRootFilesystem for stateless apps.
- Enforce limits with PodSecurity (next section).
- Test containers with kubectl exec and id to verify effective UID/GID.

Avoid:

- Running privileged containers (unless infrastructure).
- Leaving allowPrivilegeEscalation enabled.
- Relying on image defaults for user/group IDs.

Pod Security Policies/Pod Security Standards

Pod Security in Kubernetes

- Kubernetes needs a way to **enforce security constraints** on Pods,
- even when users define arbitrary manifests.
- Example goals:
  - Prevent privileged Pods.
  - Ensure containers run as non-root.
  - Control hostPath mounts and capabilities.

PodSecurityPolicy (PSP): The Old Model

- A cluster-level resource (PodSecurityPolicy) defining allowed settings:
  - privilege escalation
  - host networking / ports
  - volume types
  - SecurityContext fields
- Admission Controller PodSecurityPolicy enforced these rules.
- Each ServiceAccount or user needed explicit permission to use a PSP.

PodSecurityPolicy

```
apiVersion: policy/v1beta1
kind: PodSecurityPolicy
metadata:
 name: restricted
spec:
 privileged: false
 runAsUser:
   rule: MustRunAsNonRoot
 seLinux:
   rule: RunAsAny
 fsGroup:
   rule: MustRunAs
   ranges: [{min: 1, max: 65535}]
```

Deprecation of PodSecurityPolicy

Status:

- PSP deprecated in **Kubernetes 1.21**
- Completely **removed in 1.25**
- Replaced by the new **Pod Security Admission (PSA)** system implementing **Pod Security Standards (PSS)**

Reasons for removal:

- Complex RBAC interactions (users had to be bound to specific PSPs).
- Difficult to predict which PSP would apply.
- Limited extensibility (no fine-grained exceptions).
- Admission webhook alternatives (OPA Gatekeeper, Kyverno) became preferred.

Pod Security Standards (PSS)

- A simplified, **namespace-level** model enforcing predefined security profiles.
- Built into Kubernetes Admission Controllers (no CRD needed).
- Defines three **standard levels** of security:
  - **Privileged** – unrestricted (for system namespaces).
  - **Baseline** – minimizes privilege escalation risks.
  - **Restricted** – enforces strong security isolation.

Two Main Types

Level	Use Case	Characteristics
Privileged	System namespaces, CNI, CSI, monitoring agents	Full host access, all capabilities allowed
Baseline	Common apps that need basic permissions	No privilege escalation, limited host access
Restricted	High-security workloads, multi-tenant clusters	Non-root, read-only, no host access

Enforcing PSS on a Namespace

```
kubectl label namespace dev \
  pod-security.kubernetes.io/enforce=restricted \
  pod-security.kubernetes.io/audit=baseline \
  pod-security.kubernetes.io/warn=baseline
```

- enforce → hard-blocks noncompliant Pods.
- audit → logs violations to audit logs.
- warn → prints warnings to users during apply.

Complementing PSS with Tools

- Use **OPA Gatekeeper** or **Kyverno** to define custom rules like:
  - image registry allow-lists,
  - specific label enforcement,
  - mandatory annotations,
  - custom SecurityContext validation.

Network Security Policies

Why Network Security Matters in Kubernetes

By Default:

- All Pods in a cluster can freely communicate with all others.
- There's no built-in isolation between namespaces or apps.

This leads to potential issues:

- Data leaks between namespaces.
- Compromised Pods can scan or attack others.
- No "zero-trust" segmentation by default.

What Is a NetworkPolicy?

- A **NetworkPolicy** is a namespaced resource that defines **how Pods are allowed to communicate** with:
  - other Pods,
  - namespaces,
  - or external endpoints.

Enforced by the CNI plugin

- NetworkPolicies are just rules; enforcement depends on your CNI (e.g., Calico, Cilium, Weave, etc.).
- Not all CNIs support them — always check compatibility!

Basic NetworkPolicy Structure

```yaml
apiVersion: networking.k8s.io/v1
kind: NetworkPolicy
metadata:
 name: deny-all
 namespace: dev
spec:
 podSelector: {}
 policyTypes:
 - Ingress
 - Egress
```

Two Main Types

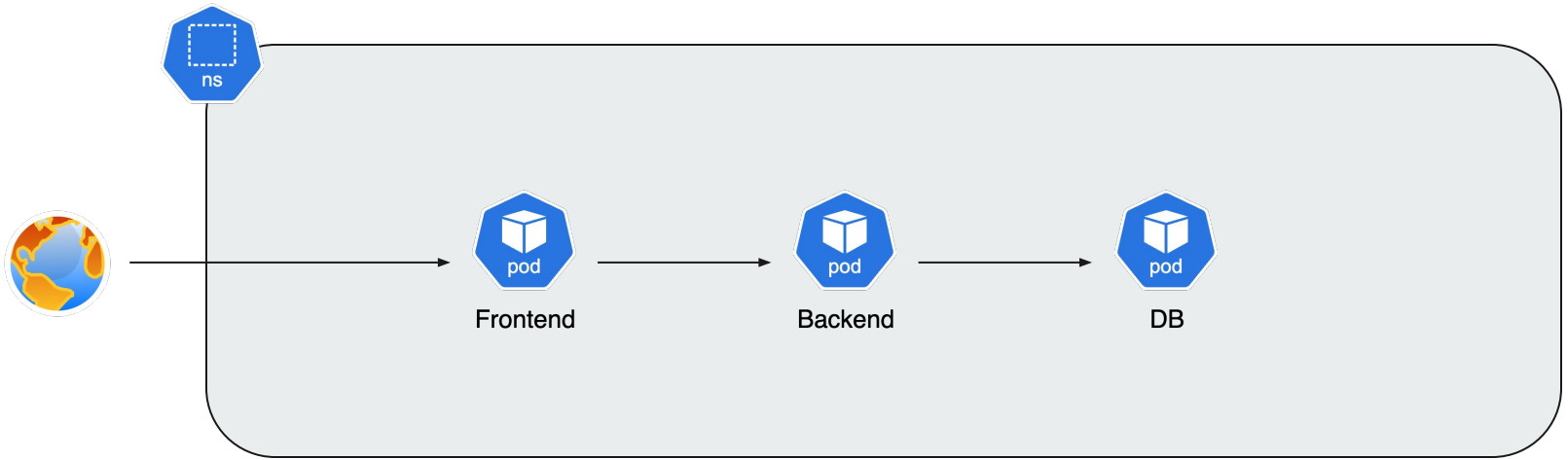
Field	Purpose
podSelector	Selects target Pods the policy applies to
policyTypes	Defines traffic direction (Ingress, Egress)
ingress	List of allowed inbound connections
egress	List of allowed outbound connections
from / to	Define sources/destinations (Pods, Namespaces, IP blocks)
ports	Limit by port and protocol

How Policies Combine

- Multiple NetworkPolicies can apply to the same Pod.
- All allow rules are **additive** (union).
- Default deny happens only if **any policy** selects a Pod and doesn't allow specific traffic.
- No "deny" policy — only allow rules.

Network Security Policy Example

Network Security Policy Example: 3-Tier App



Network Security Policy Example: 3-Tier App

- Frontend can accept only traffic from outside on port 80
- Backend accept only connection from Frontend on port 8080
- Backend should connect only do DNS and DB
- DB accept connection only from backend on Port 5432

Allow frontend to RECEIVE traffic from the outside

```yaml
apiVersion: networking.k8s.io/v1
kind: NetworkPolicy
metadata:
 name: allow-public-to-frontend
 namespace: prod
spec:
 podSelector:
   matchLabels:
     app: frontend
 policyTypes:
 - Ingress
 ingress:
 - from:
     - ipBlock:
         cidr: 0.0.0.0/0
   ports:
     - protocol: TCP
       port: 80
```

Allow frontend → backend

```yaml
apiVersion: networking.k8s.io/v1
kind: NetworkPolicy
metadata:
 name: allow-frontend-to-backend
 namespace: prod
spec:
 podSelector:
   matchLabels:
     app: backend
 policyTypes:
 - Ingress
 ingress:
 - from:
     - podSelector:
         matchLabels:
           app: frontend
   ports:
     - protocol: TCP
       port: 8080
```

Allow backend → db

```yaml
apiVersion: networking.k8s.io/v1
kind: NetworkPolicy
metadata:
 name: allow-backend-to-db
 namespace: prod
spec:
 podSelector:
   matchLabels:
     app: db
 policyTypes:
 - Ingress
 ingress:
 - from:
     - podSelector:
         matchLabels:
           app: backend
   ports:
     - protocol: TCP
       port: 5432
```

Allow Controlled Egress

```yaml
apiVersion: networking.k8s.io/v1
kind: NetworkPolicy
metadata:
 name: backend-egress
 namespace: prod
spec:
 podSelector:
   matchLabels:
     app: backend
 policyTypes:
 - Egress
 egress:
 - to:
     - podSelector:
         matchLabels:
           app: db
   ports:
     - protocol: TCP
       port: 5432
 - to:
     - namespaceSelector: {}
   ports:
     - protocol: UDP
       port: 53
```

Default Policy Example

Deny All Ingress, Allow All Egress

```yaml
apiVersion: networking.k8s.io/v1
kind: NetworkPolicy
metadata:
 name: default-deny-ingress
spec:
 podSelector: {}
 policyTypes:
 - Ingress
```

Deny All Egress, Allow All Ingress

```yaml
apiVersion: networking.k8s.io/v1
kind: NetworkPolicy
metadata:
 name: default-deny-egress
spec:
 podSelector: {}
 policyTypes:
 - Egress
```

Labs

Exposing Applications

- Service Types
- Services Diagram
- Service Update Pattern
- Accessing an Application with a Service
- Service without a Selector
- ClusterIP
- NodePort
- LoadBalancer
- ExternalName
- Ingress Resource
- Ingress Controller
- Service Mesh
- Labs

Service Types

Why Services Exist?

- Pods are ephemeral — they come and go, get recreated, their IPs change.
- Kubernetes Services give applications stable endpoints and built-in load balancing.
- Depending on where you need to expose your app, Kubernetes offers several Service types.

Four Main Service Types

Type	Visibility	Purpose	Typical Use Case
ClusterIP	Internal only (within cluster)	Default service type for inter-Pod communication	Backend APIs, databases, internal microservices
NodePort	Exposed on every Node's IP & static port	Simple external access without a LoadBalancer	Local testing, dev clusters
LoadBalancer	Exposed publicly through a cloud LB	Automatic external load balancing	Production-grade public endpoints
ExternalName	DNS alias to an external hostname	Integrates external systems into cluster DNS	Legacy or hybrid connectivity

Service Update Pattern

What It Service Update Pattern?

- Describes **how Services react to Pod lifecycle changes**.
- Ensures clients always reach **only healthy**, **ready** backend Pods.
- Works automatically — no need for manual reconfiguration.

Accessing an Application with a Service

Accessing an Application with a Service (Overview)

- Enable stable and discoverable communication between Pods and Services.
- Access applications **without caring about** Pod IPs.
- Test connectivity easily from within or outside the cluster.

DNS-Based Service Discovery

- Managed by CoreDNS (previously kube-dns).
- Every Service gets its own A record → points to the ClusterIP.
- Pods in the same namespace can even use the short name:
  `http://<service-name>:8080`
- Other namespaces require the full FQDN
  `http://<service-name>.<namespace>.svc.cluster.local:8080`

Service without a Selector

Service Without a Selector — Concept

- A Service that doesn’t automatically select Pods.
- You define backends manually via Endpoints (legacy) or EndpointSlices (modern).
- Useful when:
  - You want to point a Service to external systems (DBs, APIs).
  - You integrate with workloads outside the cluster.
  - You manage backend IPs dynamically via a controller.

Service Without a Selector

```yaml
apiVersion: v1
kind: Service
metadata:
 name: legacy-db
 namespace: prod
spec:
 ports:
 - port: 5432
   targetPort: 5432
```

Manual Endpoints (Deprecated)

```yaml
apiVersion: v1
kind: Endpoints
metadata:
 name: legacy-db
 namespace: prod
subsets:
 - addresses:
     - ip: 10.20.30.40
   ports:
     - port: 5432
```

EndpointSlices (Recommended)

```yaml
apiVersion: discovery.k8s.io/v1
kind: EndpointSlice
metadata:
 name: legacy-db-slice
 namespace: prod
 labels:
   kubernetes.io/service-name:
legacy-db
addressType: IPv4
ports:
 - name: db
   port: 5432
endpoints:
 - addresses: ["10.20.30.40"]
 - addresses: ["10.20.30.41"]
```

Four Main Service Types

Use Case	Example
Connect to an external database	legacy-db → on-prem PostgreSQL
Expose a non-Kubernetes workload	VM-based backend in same network
Proxy to external API	external-api-svc → api.partner.com
Custom controller updates endpoints dynamically	Service Mesh or DNS integration

ClusterIP

ClusterIP — The Default Service Type

- It’s the **default Service type** in Kubernetes.
- Used for **service-to-service** communication (e.g. frontend → backend).
- Not reachable from outside unless combined with Ingress, NodePort, or LoadBalancer.

ClusterIP

```yaml
apiVersion: v1
kind: Service
metadata:
 name: backend-svc
 namespace: prod
spec:
 type: ClusterIP
 selector:
   app: backend
 ports:
   - name: http
     port: 8080
     targetPort: 8080
```

ClusterIP in Practice

Common Use Cases:

- Internal communication between microservices.
- APIs consumed by other in-cluster components.
- Databases or message brokers used by applications.
- Monitoring / telemetry pipelines.

Not For:

- External access (from outside cluster).
- Direct internet exposure.

NodePort

What Is NodePort?

- Makes a Service reachable from **outside** the cluster.
- Automatically creates an **underlying ClusterIP Service**.
- Traffic sent to <NodeIP>:<NodePort> is routed to backend Pods.

NodePort

```yaml
apiVersion: v1
kind: Service
metadata:
 name: frontend-svc
 namespace: prod
spec:
 type: NodePort
 selector:
   app: frontend
 ports:
   - port: 80
     targetPort: 8080
     nodePort: 30080
```

When to Use NodePort

Good For:

- Development, testing, and simple clusters.
- On-prem or local environments without cloud LoadBalancers.
- When using your own external reverse proxy (e.g., NGINX, HAProxy, Traefik).

Not Ideal For:

- Production-grade public exposure (no health checks, no autoscaling).
- Environments with dynamic IPs or limited node access.
- Load distribution is not automatic across multiple Nodes — you need an external LB in front.

LoadBalancer

What Is a LoadBalancer Service?

- Extends ClusterIP + NodePort with an external IP.
- Automatically provisions a load balancer in supported environments (GCP, AWS, Azure).
- Forwards external traffic → NodePorts → ClusterIP → backend Pods.

LoadBalancer

```yaml
apiVersion: v1
kind: Service
metadata:
 name: public-api
 namespace: prod
spec:
 type: LoadBalancer
 selector:
   app: api
 ports:
   - port: 443
     targetPort: 8443
```

When to Use LoadBalancer?

Best For:

- Exposing public APIs or web apps to the internet.
- Production clusters running on a managed cloud (GKE, EKS, AKS).
- Combining with Ingress for advanced routing and TLS termination.

Be Cautious With:

- Cost — each LB instance may incur hourly and traffic charges.
- Static IP persistence — deleting the Service releases the IP unless reserved.
- Cloud provider rate limits or provisioning delays.

ExternalName

What Is ExternalName?

- Kubernetes DNS resolves the Service name directly to the external hostname.
- No ClusterIP, no kube-proxy, no Endpoints.
- Great for **connecting in-cluster applications to external services** — e.g., managed databases, APIs, or legacy systems.

ExternalName

```yaml
apiVersion: v1
kind: Service
metadata:
 name: external-db
 namespace: prod
spec:
 type: ExternalName
 externalName: db.example.com
```

Limitations and Considerations

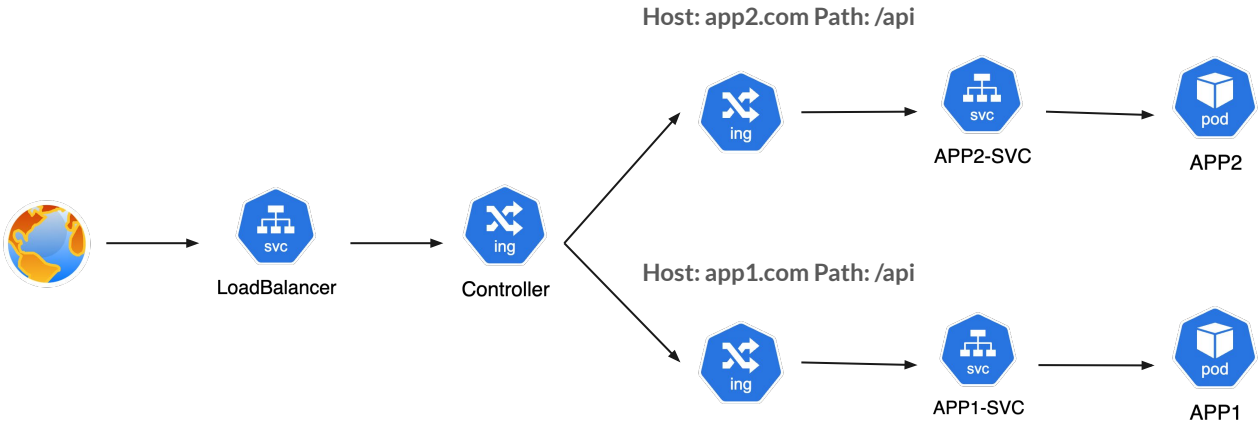
- Works only with **DNS-resolvable names**, not raw IPs.
- Does not support ports — you connect to whatever the external host uses.
- No Endpoints, readinessProbe, or health checks.
- No network policy enforcement — traffic bypasses Kubernetes networking.
- For more advanced control, consider using a normal ClusterIP with manually defined EndpointSlices.

Ingress Resource

What Is an Ingress?

- An Ingress is a Kubernetes API object that manages external HTTP/S access to Services inside the cluster.
- It defines Layer 7 routing rules (based on host, path, or both).
- Requires an Ingress Controller to implement these rules.
- Works with ClusterIP Services — not directly with Pods or NodePorts.

Ingress Architecture Overview



Ingress Example

```yaml
apiVersion: networking.k8s.io/v1
kind: Ingress
metadata:
 name: web-ingress
 annotations:
   nginx.ingress.kubernetes.io/rewrite-target: /
spec:
 rules:
 - host: myapp.example.com
   http:
     paths:
     - path: /app1
       pathType: Prefix
       backend:
         service:
           name: app1-svc
           port:
             number: 80
     - path: /app2
       pathType: Prefix
       backend:
         service:
           name: app2-svc
           port:
             number: 80
```

Ingress Controller

What Is an Ingress Controller?

- An Ingress Controller is the actual implementation that watches Ingress resources and configures a reverse proxy or load balancer accordingly.
- It’s a running component inside your cluster (usually a Deployment + Service).
- Reads Ingress manifests via the Kubernetes API.
- Configures routing rules dynamically (host/path/TLS).
- Handles load balancing, TLS termination, redirects, rewrites, etc.

Common Ingress Controllers

Controller	Description	Best For
NGINX Ingress Controller	Most popular, stable, flexible	General workloads, production
Traefik	Lightweight, auto-discovers routes	Simple setups, observability
HAProxy Ingress	High performance, fine-grained control	High-load environments
Istio Gateway / Envoy	Part of service mesh	Advanced traffic management
Kong / Ambassador / Contour	API gateway solutions	Enterprise, API exposure
Cloud Provider Controllers	GCP, AWS, Azure native LBs	Managed environments

IngressClass

```
apiVersion: networking.k8s.io/v1
kind: IngressClass
metadata:
 name: nginx
spec:
 controller:
k8s.io/ingress-nginx
```

Ingress

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

Service Mesh

What Is a Service Mesh?

A **Service Mesh** is an infrastructure layer that manages **service-to-service** communication — adding features like **security**, **observability**, **and traffic control** without modifying application code.

- Provides a dedicated network layer for internal service traffic.
- Operates at Layer 7 (HTTP/gRPC), similar to an internal Ingress.
- Typically implemented as a set of sidecar proxies running alongside Pods.

How a Service Mesh Works

- Each Pod includes a sidecar proxy injected automatically.
- All inbound/outbound traffic passes through this proxy.
- The proxies communicate via a control plane (e.g., Istiod, Linkerd Control).
- Applications are unaware — no code changes required.

Common Ingress Controllers

Component	Role
Data Plane	Sidecar proxies handling actual traffic (e.g., Envoy)
Control Plane	Central management that configures the proxies
Telemetry System	Collects metrics, logs, and traces
Security Layer	Manages certificates, mTLS, identity, and authorization

Service Mesh Features

Traffic Management:

- Fine-grained routing (e.g., 90% v1, 10% v2)
- Retries, circuit breakers, fault injection
- Intelligent load balancing

Security:

- Mutual TLS (mTLS) between services
- Certificate rotation and workload identity
- Policy-based access control

Observability:

- Built-in metrics (latency, error rates)
- Distributed tracing (Jaeger, Zipkin)
- Structured logs for each request

When to Use a Service Mesh

Use a Service Mesh When:

- You need zero-trust networking (mTLS everywhere).
- You operate many microservices and want consistent policies.
- You require advanced traffic control (A/B, canary, retry, timeout).
- You need deep observability for troubleshooting and SLOs.

Avoid / Postpone When:

- Your cluster is small (<10 services).
- You don’t yet need advanced routing or telemetry.
- You can achieve goals via simpler means (Ingress + NetworkPolicy).

Labs

Troubleshooting

- Troubleshooting Overview
- Basic Troubleshooting Steps
- Ongoing (Constant) Change
- Basic Troubleshooting Flow: Pods
- Basic Troubleshooting Flow: Node and Security
- Basic Troubleshooting Flow: Agents
- Monitoring
- Logging Tools
- Monitoring Applications
- System and Agent Logs
- Conformance Testing
- More Resource
- Labs

Troubleshooting Overview

Why Troubleshooting Matters

Kubernetes is a distributed system — failures happen often and in many layers:

- Pod scheduling
- Networking
- Configuration errors
- Permissions
- Resource limits

Common Troubleshooting Layers

Layer	Examples of Issues	Primary Tools
Application	Crashes, misconfiguration, bad probes	kubectl logs, kubectl exec, readinessProbe
Pod / ReplicaSet	Pending Pods, ImagePull errors	kubectl describe pod, kubectl get events
Node / Kubelet	Disk pressure, taints, scheduling issues	kubectl get nodes, journalctl -u kubelet
Network	Service unreachable, DNS failures	kubectl exec curl, kubectl get svc, nslookup
Cluster Control Plane	API unresponsive, etcd sync issues	kubectl get componentstatuses, system logs

Common Categories of Problems

Category	Description	Example Symptom
Scheduling	Pod can't be placed on a node	Pending state
Image Pull	Container image not found or unauthorized	ErrImagePull
CrashLoopBackOff	App repeatedly starts and fails	Wrong entrypoint or missing dependency
Readiness / Liveness	Health check fails	Pod marked Unready
Networking	Services not reachable	DNS or CNI issue
RBAC / Secrets	Forbidden or missing credentials	403 Forbidden errors
Resource Limits	Pod evicted or throttled	OOMKilled, CPU throttling

Basic Troubleshooting Steps

`Troubleshooting Steps

1. **Observe** – What's happening? Identify symptoms.
2. **Describe** – Collect context with kubectl describe and kubectl get events.
3. **Check Logs** – Application and container logs (kubectl logs).
4. **Validate Configuration** – YAML specs, probes, image names, namespaces.
5. **Test Communication** – Use kubectl exec, curl, or nslookup inside a Pod.
6. **Inspect Nodes** – Check node status, resources, and kubelet logs.
7. **Confirm Permissions** – Use kubectl auth can-i.
8. **Fix and Verify** – Apply fixes, re-test, and document cause.

Ongoing (Constant) Change

Basic Troubleshooting Flow: Pods

Pod Troubleshooting Flow

When a Pod isn’t Running, it’s usually due to:

- Scheduling problems
- Container image errors
- Application startup failures
- Probe configuration issues
- Resource limits or permissions

Pod Stuck in Pending

- No available nodes (unschedulable or tainted).
- Insufficient CPU/memory resources.
- Node selectors or affinity rules too restrictive.
- PVC not yet bound (waiting for volume).

ImagePullBackOff / ErrImagePull

- Image name or tag typo.
- Private registry without credentials.
- DockerHub rate limiting.

CrashLoopBackOff

- Application crashes on startup.
- Wrong entrypoint or missing dependency.
- Misconfigured environment variables or secrets.
- Failing liveness probe (container restarted repeatedly).

OOMKilled or Resource Pressure

- Container exceeded memory limit.
- Node memory pressure.
- Incorrect requests and limits configuration.

Readiness / Liveness Probe Failures

- Wrong probe path or port.
- App not ready yet when probe runs.
- Timeout too short or missing startupProbe.

ConfigMap / Secret Mount Issues

- ConfigMap or Secret doesn’t exist.
- Wrong key reference.
- Missing volume mount path.

Volume Mount Failures

- PersistentVolumeClaim not bound.
- StorageClass misconfiguration.
- Access mode conflict (e.g. RWO volume used by multiple Pods).

Network / DNS Problems

- CoreDNS not running or misconfigured.
- NetworkPolicy blocking traffic.
- Wrong Service name or port.

Pod Evicted

- Node out of disk, memory, or CPU.
- Eviction due to QoS and scheduling pressure.

Basic Troubleshooting Flow: Node and Security

Troubleshooting Nodes and Security

- Node availability and scheduling
- Kubelet health and logs
- Resource pressure and taints
- RBAC misconfigurations
- API access and authentication

Node Not Ready

- Kubelet stopped or crashed.
- Network plugin (CNI) malfunctioning.
- Disk or memory pressure.
- Control plane can’t reach the node (firewall or network split).

Node Has Disk or Memory Pressure

- Node is out of disk space or memory.
- Pods with low priority being evicted.

Pods Not Scheduled on a Node

- Taints on nodes preventing scheduling.
- Resource requests too high.
- Node affinity or selector too restrictive.

Kubelet Registration Issues

- Expired kubelet certificate.
- Clock skew between node and API server.
- Network connectivity flapping.

RBAC — Access Denied

- Missing or incorrect Role/ClusterRole.
- RoleBinding not applied or bound to wrong ServiceAccount.

Unauthorized / API Access Fails

- API Server down or crashed.
- Wrong kubeconfig context.
- Firewall or network blocking port 6443.

Admission Controller Blocking Changes

- ValidatingWebhook or MutatingWebhook rejecting Pod creation.
- Policy engine (OPA Gatekeeper, Kyverno) enforcing rules.

Basic Troubleshooting Flow: Agents

What Are Kubernetes Agents?

Kubernetes “agents” are the background components that run cluster logic — both on control-plane nodes and worker nodes.

These include:

- **kubelet** — manages Pods on each node
- **kube-proxy** — maintains networking rules
- **controller-manager** — ensures desired state
- **scheduler** — assigns Pods to nodes
- **CoreDNS** — handles service discovery
- **etcd** — key-value store for all cluster data

Kubelet Issues

- Node marked NotReady.
- Pods fail to start or get stuck in ContainerCreating.

Kube-Proxy Problems

- Services not reachable (curl to ClusterIP fails).
- Endpoints exist, but requests time out.

CoreDNS Failures

- Pods can’t resolve other Services (nslookup fails).
- Applications can’t connect using service names.

Scheduler Not Assigning Pods

- Pods remain in Pending, even with healthy nodes.
- Wrong selector or taint

Controller Manager Not Reconciling

- Deployments stuck with old replicas.
- CronJobs, Jobs, or ReplicaSets not updating.

etcd Performance or Health Issues

- API calls extremely slow or timing out.
- Errors like:
  `etcdserver: request timed out`

Control Plane Connectivity

- kubectl intermittent timeouts or slow responses.
- “Connection refused” or “context deadline exceeded.”

Monitoring

Monitoring Overview

Monitoring in Kubernetes means collecting metrics, logs, and events from the cluster, nodes, and workloads — to ensure reliability and performance.

Monitoring helps you:

- Detect problems early.
- Understand resource consumption.
- Analyze performance trends.
- Debug issues post-incident.

Three Pillars of Observability

Pillar	What It Means	Example Tools
Metrics	Numerical performance data (CPU, memory, latency)	Metrics Server, Prometheus, Grafana
Logs	Structured/unstructured event text output	Fluentd, Loki, Elasticsearch, kubectl logs
Traces	End-to-end request visibility across services	Jaeger, OpenTelemetry, Tempo

Logging Tools

Logging in Kubernetes (Overview)

Logging in Kubernetes means collecting, storing, and analyzing log output from applications, Pods, and system components.

Logs are the most direct way to understand:

- What your app is doing.
- Why it failed.
- How it interacts with other components.

Monitoring Applications

Application Monitoring Overview

Application monitoring in Kubernetes focuses on understanding how deployed workloads behave — performance, health, and dependencies.

You can monitor apps at multiple layers:

- Pod-level (probes, resource usage)
- Service-level (latency, error rates)
- Application-level (custom business metrics)

System and Agent Logs

System and Agent Logs Overview

System and agent logs provide insights into the internal workings of Kubernetes — including node health, networking, scheduling, and control-plane operations.

You'll analyze logs from:

- Node-level components (kubelet, kube-proxy, containerd)
- Control-plane components (kube-apiserver, scheduler, controller-manager, etcd)
- Cluster services (CoreDNS, metrics-server, etc.)

Conformance Testing

What Is Conformance Testing?

Conformance testing verifies that a Kubernetes cluster implementation behaves according to the official CNCF (Cloud Native Computing Foundation) specifications.

- Ensures consistency and interoperability between Kubernetes distributions (EKS, GKE, AKS, OpenShift, etc.).
- Confirms that all core APIs and workloads operate as defined by upstream Kubernetes.
- Required for Certified Kubernetes status.

Why Conformance Testing Matters

- Ensures vendor neutrality — workloads behave the same across environments.
- Helps detect API regressions or incompatible changes after upgrades.
- Provides confidence in custom or self-managed clusters.
- Required for Kubernetes Certified Service Provider (KCSP) or Certified Distribution status.

Sonobuoy — The Official Conformance Test Tool

Sonobuoy is the official CNCF tool for running Kubernetes conformance tests.

- Developed by VMware Tanzu under CNCF.
- Runs end-to-end (E2E) tests to validate Kubernetes behavior.
- Generates detailed reports (JSON and tarball) with test results.
- Works with any Kubernetes 1.11+ cluster.

Running a Conformance Test with Sonobuoy

Installation

```
curl -L https://github.com/vmware-tanzu/sonobuoy/releases/latest/download/sonobuoy_$(uname
-s)_amd64.tar.gz | tar xz
sudo mv sonobuoy /usr/local/bin
```

Run Tests

```
sonobuoy run --mode=certified-conformance
```

Check Status

```
sonobuoy status
```

Retrieve Results

```
sonobuoy retrieve .
tar xzf *.tar.gz
```

Conformance Coverage Areas

Category	What It Tests
Core API Behavior	Pods, Services, Deployments, ConfigMaps
Scheduling	Pod placement and taints/tolerations
Networking	DNS, Service discovery, CNI communication
Storage	PVC provisioning and volume mounting
Security	ServiceAccounts, RBAC, Secrets
Scalability	Replication, rolling updates, and state reconciliation

More Resource

More Resources

- **Taints & Tolerations** — keep Pods off certain nodes unless allowed.
- **Node Affinity** — tell Pods where they should (or must) run.
- **Topology Spread Constraints** — keep replicas evenly distributed.
- **Helm** — package, version, and deploy apps as charts.
- **Kustomize** — environment-specific overlays without templates.

Taints & Tolerations

Taints & Tolerations — Concept

Definition

- **Taint (node)**: repels Pods that don't tolerate it.
- **Toleration (pod)**: allows a Pod to land on a tainted node.

Effects

- NoSchedule — scheduler won't place Pods unless they tolerate.
- PreferNoSchedule — soft hint; scheduler avoids placing Pods.
- NoExecute — evicts existing Pods that don't tolerate; blocks new ones.

Common Use Cases

- **Dedicated nodes** (e.g., dedicated=infra:NoSchedule)
- **GPU pools** (only ML workloads tolerate nvidia.com/gpu taint)
- **Maintenance/cordon-like control** (NoExecute to clear a node)
- **Spot/preemptible pools** (only best-effort workloads tolerate them)

Examples (CLI & YAML)

```
kubectl taint nodes node1 dedicated=infra:NoSchedule
```

```
apiVersion: v1
kind: Pod
metadata: { name: log-shipper }
spec:
 tolerations:
 - key: "dedicated"
   operator: "Equal"
   value: "infra"
   effect: "NoSchedule"
 containers:
 - name: flb
   image: cr.fluentbit.io/fluent/fluent-bit:2.2
```

Best Practices (Taints/Tolerations)

Do

- Label + taint special pools (infra, gpu, io-optimized).
- Pair with Node Affinity to pull Pods where they belong.
- Document taint keys/values in a cluster contract.

Avoid

- Catch-all tolerations on every Pod.
- Using Exists indiscriminately (defeats taints' purpose).

Node Affinity

Node Affinity — Concept

Definition:

Pod-side rules that constrain or prefer nodes by node labels.

Types:

- requiredDuringSchedulingIgnoredDuringExecution (**hard**)
- preferredDuringSchedulingIgnoredDuringExecution (**soft, weighted**)

Hard Affinity

```yaml
affinity:
 nodeAffinity:
   requiredDuringSchedulingIgnoredDuringExecution:
     nodeSelectorTerms:
     - matchExpressions:
       - key: disktype
         operator: In
         values: ["ssd"]
```

Soft Affinity

```yaml
affinity:
 nodeAffinity:
   preferredDuringSchedulingIgnoredDuringExecution:
   - weight: 80
     preference:
       matchExpressions:
       - key: topology.kubernetes.io/zone
         operator: In
         values: ["europe-west1-a"]
```

Topology Spread Constraints

Topology Spread Constraints

Rules that keep replicas evenly distributed across topology domains (nodes/zones/regions).

Why

- Avoid single-node/zone failure for an entire Deployment.
- Balance load & failure domains.
- Complement affinity/anti-affinity.

Core Fields

Field	Meaning
topologyKey	Node label key (e.g., kubernetes.io/hostname, topology.kubernetes.io/zone)
maxSkew	Max difference in Pod count between domains
whenUnsatisfiable	DoNotSchedule (strict) or ScheduleAnyway (best-effort)
labelSelector	Which Pods count toward balancing

Even Spread Across Nodes

```
spec:
 replicas: 6
 selector:
     { matchLabels: { app: web } }
 template:
   metadata: { labels: { app: web } }
   spec:
     topologySpreadConstraints:
     - maxSkew: 1
       topologyKey: kubernetes.io/hostname
       whenUnsatisfiable: DoNotSchedule
       labelSelector: { matchLabels: { app: web
} }
     containers:
     - name: nginx
       image: nginx:1.27
```

Best Practices

- Start with one constraint (hostname or zone), maxSkew: 1.
- Use DoNotSchedule for HA services; ScheduleAnyway for batch.
- Remember: DaemonSets are excluded by default.
- Keep labelSelector aligned with your Deployment’s labels.

Helm

Helm — Concept & Value

What

- Packages Kubernetes manifests into charts (templated YAML).
- Manages install/upgrade/rollback with versioned releases.
- Centralizes configuration in values.yaml (and -f overrides.yaml).

Why

- Reuse, share, and version complex deployments (Prometheus, Ingress, etc.).
- Consistent rollouts across environments/teams.
- First-class fit for CI/CD and GitOps.

Chart Structure

```
mychart/
  ├── Chart.yaml          # name, version, description, apiVersion
  ├── values.yaml         # defaults
  ├── templates/          # *.yaml.gotmpl (Go templates)
  └── charts/             # sub-charts (dependencies)
```

Basic Workflow (CLI)

```
# add repo & search
helm repo add bitnami https://charts.bitnami.com/bitnami
helm search repo nginx

# install with overrides
helm install web bitnami/nginx -f values-prod.yaml

# inspect / diff / upgrade / rollback
helm get values web
helm upgrade web bitnami/nginx -f values-new.yaml
helm rollback web 1
```

Best Practices

Do

- Keep templates minimal; push logic into values.yaml.
- Use semver and changelogs; document breaking changes.
- Validate with helm lint and helm template.
- Pin sub-chart versions in Chart.yaml (dependencies:).

Avoid

- Overusing conditionals/loops — hard to maintain.
- Embedding secrets directly in values (use External Secrets/Sealed Secrets).

Kustomize

Kustomize — Concept

What

- Patch & compose raw YAML using overlays, no templates.
- Natively supported: kubectl apply -k.

Why

- Clean environment layering (dev/stage/prod).
- Deterministic, YAML-in/YAML-out.
- Great for teams preferring no templating language.

Directory Layout & Command

```
base/
 deployment.yaml
 service.yaml
 kustomization.yaml
overlays/
 staging/
   kustomization.yaml
   patch-replicas.yaml
 prod/
   kustomization.yaml
   patch-resources.yaml
```

Best Practices (Kustomize)

Do

- Keep base minimal and reusable; put env deltas into overlays.
- Prefer strategic merges for readability; use JSON6902 for precision.
- Leverage commonLabels, namePrefix, images to avoid copy-paste.

Avoid

- Deep overlay stacks (hard to reason about).
- Mixing Kustomize & Helm in the same tree unless clearly separated.

Labs

TEF

TEF

https://sube.nobleprog.com/pl/open-tef/38303