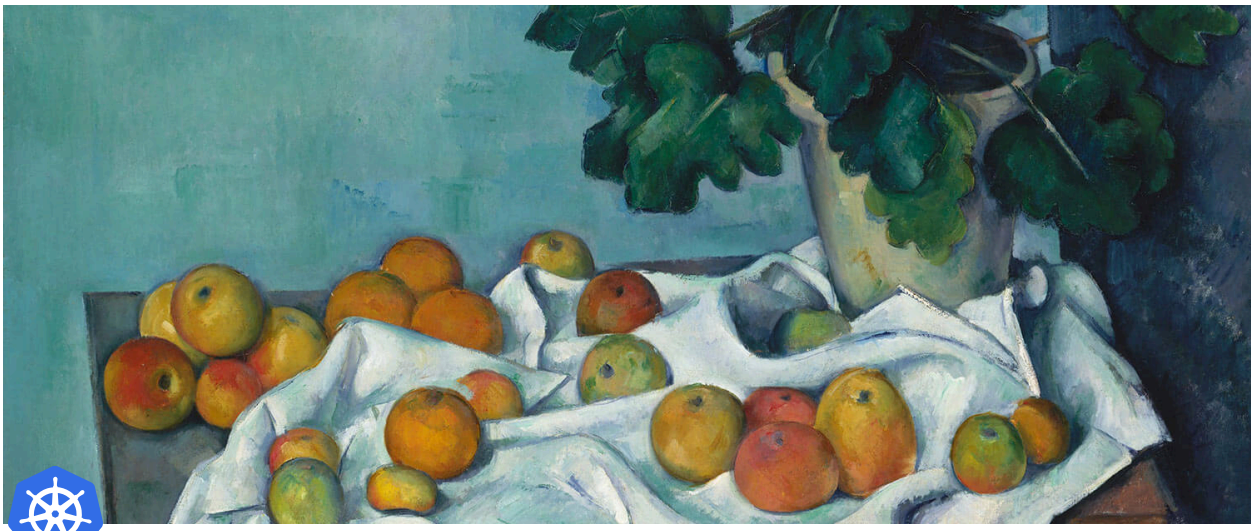


Intro to Kubernetes

KUBERNETES ARCHITECTURE SIMPLIFIED EXPLANATION

techopsexamples.com

1. KUBECTL → Lets you talk to Kubernetes

2. API SERVER → The brain that handles all requests

3. CONTROLLER MANAGER → Adjusts cluster resources as needed

4. SCHEDULER → Finds the best spot for workloads

5. KUBELET → Runs workloads on each machine

6. ETCD → Remembers everything about the cluster

7. KUBE PROXY → Directs traffic to the right workloads

8. POD → Where workloads actually run

9. CONTAINER RUNTIME → Runs the apps inside workloads

What Kubernetes Opens the Door To:

Category	What You Can Learn Next
GitOps	ArgoCD, Flux, Kustomize, Helm
Observability	Prometheus, Grafana, Loki, Jaeger
Service Mesh	Istio, Linkerd, Consul
Security	Pod security policies, network policies, mTLS, secret management
Chaos Engineering	LitmusChaos, Chaos Mesh
CI/CD	Kubernetes-native delivery pipelines (e.g., Tekton, Jenkins X)
Secret Management	Kubernetes Secrets, Sealed Secrets, Vault
Cloud & Multi-Cloud	AWS EKS, GCP GKE, Azure AKS, hybrid Kubernetes
Networking	Ingress controllers, CNI, DNS inside clusters
Platform Engineering	Building internal developer platforms (IDPs)
SRE/Resilience	Auto-scaling, self-healing, failover, resilience testing
Serverless	Knative, KEDA, FaaS on K8s
Cost Optimization	Cluster autoscaling, resource requests/limits tuning

Microservices & Monolithic Architectures

From Monolith to Microservices

Introduction

Many organizations start with a **monolithic architecture,** a single, unified application where all components are interconnected and run as one. As the application grows, monoliths can become **difficult to scale, maintain, and deploy**. This is where **microservices architecture** comes in.

What is a Monolith?

A **monolith** is a single application that contains all business logic, UI, and data access code.

All features run in one process and are tightly coupled.

Pros:

- Simple to develop (at the start)
- Easy to test and deploy as one unit
- Fewer operational concerns

Cons:

- Hard to scale individual components

- Long deployment cycles
- Difficult to adopt new technologies
- A single bug can affect the whole app

What Are Microservices?

Microservices are an architectural style where an application is broken into **smaller, independent services**, each responsible for a single business capability.

Pros:

- Each service can be developed, deployed, and scaled independently
- Technology agnostic (you can use different languages)
- Faster iteration and innovation
- Fault isolation (if one service fails, the rest can still work)

Cons:

- More complex infrastructure
- Requires DevOps maturity (CI/CD, observability, container orchestration)
- Harder debugging and tracing
- Network latency and communication overhead

Transition: From Monolith to Microservices

The transition is not a "big bang" rewrite. It's usually **incremental**, with a **"Strangler Fig"** pattern, gradually replacing parts of the monolith with microservices.

Steps to Migrate:

1. **Understand the Monolith:**

   Map out business capabilities and data domains.

2. **Identify Boundaries:**

   Break the system into logical domains using Domain-Driven Design (DDD).

3. **Extract Services:**

   Choose low-risk modules to extract first (e.g., authentication, user profile).

4. **Set up Communication:**

   Use REST, gRPC, or messaging (like Kafka or RabbitMQ) between services.

5. **Deploy and Observe:**

   Use CI/CD pipelines, logging, tracing, and monitoring (e.g., Prometheus, Grafana).

6. **Iterate:**

   Gradually migrate more parts and shrink the monolith.

Tools That Help

Purpose	Tools
Containers	Docker
Orchestration	Kubernetes
CI/CD	GitHub Actions, GitLab CI, Jenkins
Monitoring	Prometheus, Grafana
Tracing	Jaeger, OpenTelemetry
API Gateway	NGINX, Istio, Kong

Refactoring

Improving the Design of Existing Code

Refactoring is a disciplined technique for restructuring existing code without changing its external behavior. The goal is to improve the internal structure, making it easier to understand, more maintainable, and more adaptable to future changes. In essence, refactoring is about *cleaning up code,* not fixing bugs or adding features, but laying a stronger foundation for both.

Why Refactor?

1. **Improves Readability**: Clean code communicates intent. Refactoring eliminates duplication, simplifies logic, and improves naming, making the codebase easier to read and understand.
2. **Enhances Maintainability**: Well-structured code is easier to modify and debug.
3. **Reduces Technical Debt**: Over time, as features are added quickly under pressure, codebases can accumulate technical debt. Refactoring helps pay that back incrementally.
4. **Supports Scalability**: A well-factored system adapts better to change. It allows developers to scale applications confidently, knowing the architecture supports evolution.
5. **Facilitates Collaboration**: Consistent and clear code makes team collaboration smoother. Team members can onboard faster and contribute effectively.

Architectural Refactorings

These involve larger structural changes:

- **Modularization**
- **Decomposing Monoliths**

- **Introducing Layers or Services**
- **Event Sourcing / CQRS (in the context of design improvement)**

How to Refactor Effectively

1. **Write Tests First**: Before refactoring, ensure there are sufficient unit and integration tests to catch regressions.
2. **Make Small, Incremental Changes**: Big refactorings are risky. Do them in small steps, testing after each one.
3. **Use Version Control**: Commit frequently so you can roll back if something goes wrong.
4. **Automate Where Possible**: Use IDE tools and static analysis plugins to assist with common refactorings.
5. **Review and Collaborate**: Get feedback from peers to ensure the refactoring improves readability and doesn't introduce subtle issues.

Refactoring vs. Rewriting

- **Refactoring** improves code step-by-step without changing behavior.
- **Rewriting** involves discarding existing code and starting over, often with changed behavior and higher risk.

While rewriting might seem tempting, especially for legacy systems, it often introduces new bugs and delays. Refactoring is generally safer and more sustainable.

CNCF

What Are CNCF Projects?

The **Cloud Native Computing Foundation (CNCF)** is a part of the **Linux Foundation**, created to foster and sustain open-source technologies that support **cloud-native** architectures.

CNCF hosts a wide ecosystem of **open-source projects** that help build scalable, resilient, and observable cloud-native systems.

Cloud Native Defined

Key features:

- Containers
- Microservices
- Declarative APIs
- Continuous delivery

CNCF Project Maturity Levels

CNCF organizes projects into **three maturity levels**:

Level	Description
Sandbox	Early-stage, experimental projects with promise.
Incubating	Projects with active contributors and adoption, undergoing further testing.
Graduated	Mature, widely adopted projects with proven use in production environments.

Graduated CNCF Projects (Examples)

These are production-grade tools trusted across the industry:

Project	Purpose
Kubernetes	Container orchestration
Prometheus	Monitoring & alerting
Envoy	Cloud-native proxy / service mesh
Helm	Kubernetes package manager
Containerd	High-performance container runtime
etcd	Distributed key-value store used in Kubernetes
CoreDNS	Flexible DNS server for service discovery

Incubating & Sandbox Projects (Examples)

These are up-and-coming or maturing tools:

Project	Purpose
OpenTelemetry	Observability framework (metrics, logs, traces)
Argo	GitOps workflows and pipelines
Dapr	Distributed application runtime
Longhorn	Cloud-native distributed block storage
Cilium	Network security and observability using eBPF
Linkerd	Lightweight service mesh

> 🌱 Sandbox examples: WasmEdge, KubeVirt, KEDA, Chaos Mesh

Why CNCF Projects Matter

- **Vendor-neutral**: No vendor lock-in, supported by open communities.

- **Ecosystem-driven**: Interoperable tools that complement each other.
- **Scalable and resilient**: Built for modern cloud-scale applications.
- **Secure**: CNCF emphasizes security audits and best practices.

CNCF Landscape

The CNCF maintains a huge interactive landscape covering hundreds of projects across:

- App development
- CI/CD
- Networking & service mesh
- Storage
- Observability
- Orchestration
- Security

Kubernetes Overview

Kubernetes (often abbreviated as **K8s**) is an open-source platform used to **automate the deployment, scaling, and management of containerized applications**.

Simply put:

Kubernetes helps you run and manage Docker containers at scale.

Key Concepts:

- **Container Orchestration**: Kubernetes organizes multiple containers (such as those created with Docker) across several machines.
- **Automated Scaling**: It can automatically scale the number of containers up or down based on workload.
- **Self-healing**: If a container fails, Kubernetes replaces or restarts it automatically.
- **Load Balancing**: It distributes network traffic to keep your apps responsive.
- **Declarative Configuration**: You tell Kubernetes how your app should look, and it makes sure it stays that way.

Example Use Case:

If you have a web app built in Docker containers and suddenly get a lot of traffic, Kubernetes can:

- Automatically start more containers to handle the load.

- Stop extra containers when traffic goes down.
- Restart any failed containers.

Why Use Kubernetes?

- Handles complex container setups easily.
- Works across public clouds, private data centers, or your laptop.
- Widely adopted by companies for modern cloud-native applications.

Container Orchestration

What is Container Orchestration?

Container orchestration is the process of **automatically managing** a large number of containers (like Docker containers) across many servers. It handles tasks like:

- Starting and stopping containers
- Ensuring containers are running properly
- Distributing workloads across servers
- Scaling containers up or down based on demand
- Managing communication between containers

Why Do You Need Container Orchestration?

Imagine you're running an app with:

- A **web server**
- A **database**
- A **search engine**
- A **cache system**

Each run is in its container. As traffic grows, you might need:

- **More web server containers**
- Containers are spread across multiple machines
- Automatic replacement if one fails

Doing this manually gets **complex and error-prone**. That’s where **container orchestration tools**, like **Kubernetes**, come in.

What Kubernetes Does in Container Orchestration

Kubernetes automates:

Task	Description
Deployment	Spreads containers across machines (called nodes).
Scaling	Adds/removes containers when needed (e.g., more users = more containers).
Load Balancing	Sends traffic evenly to all containers.
Self-healing	Restarts failed containers or replaces broken ones.
Rolling Updates	Updates apps without downtime.

Real-World Analogy

Think of Kubernetes like an **air traffic controller** at a busy airport:

- It makes sure planes (containers) land and take off (start and stop) safely.
- Directs them to the right gates (servers).
- Handles delays or emergencies automatically.
- Keeps the whole airport running smoothly.

Orchestration Technologies

1. Docker Swarm → from Docker
2. Kubernetes → from Google
3. Mesos → from Apache

Kubernetes Architecture

Kubernetes follows a **client-server architecture** and is made up of two main components:

1. Control Plane (Master Components)

Manages the cluster and is responsible for global decisions (e.g., scheduling), detecting and responding to cluster events (e.g., starting up a new pod when a deployment's replicas field is unsatisfied).

- It needs to be running all the time at all costs. → We can create Control Plane node replicas configured in High Availability mode
- One control plane takes care of managing the cluster, while the rest stay in sync across all replicas
- This type of configuration adds resiliency to the cluster's control plane

Components of the Control Plane:

- **API Server (`kube-apiserver`)**
  - The front-end of the Kubernetes control plane.
  - All components interact with the cluster through this REST API.
  - It can scale horizontally
  - Highly configurable and customizable
- **Scheduler (`kube-scheduler`)**
  - Assigns newly created Pods to nodes based on resource availability and constraints.
  - Distributing work or containers across multiple nodes
  - It looks for newly created containers and assigns them to nodes
  - Highly configurable and customizable through scheduling policies, plugins, and profiles.
  - This component is extremely important and complex in a multi-node Kubernetes cluster.
- **Controller Manager (`kube-controller-manager`)**
  - Runs controllers (like the Node Controller, Replication Controller, etc.) that regulate the state of the system.
  - The brains behind orchestration
  - They are responsible for noticing and responding when nodes, containers, or endpoints go down, they make decisions to bring up new containers
  - Controllers are watch-loop processes that continuously run and compare the cluster's desired state
- **etcd Key Store**
  - A highly available key-value store that stores all cluster data.
  - It's the single source of truth for Kubernetes.
  - It only communicates with the API Server
  - etcd is written in the Go programming language. In Kubernetes, besides storing the cluster state, etcd is also used to store configuration details such as subnets, ConfigMaps, Secrets, etc.
- **Cloud Controller Manager** (optional)
  - Integrates Kubernetes with the cloud provider's API to manage resources like load balancers and storage.

---

2. Node (Worker Components)

These are the machines (VMs or physical servers) where your application workloads (pods) run.

Provide a running environment for client applications

- Applications are encapsulated in pods, controlled by the control plane agents running on the control plane node.
- Pods are scheduled on worker nodes

Components on Each Node:

- **kubelet**
  - An agent that runs on each node and ensures containers are running in a Pod.
  - Communicates with the API Server.
  - It runs on each node inside the cluster
- **kube-proxy**
  - Maintains network rules on nodes.
  - Handles network routing and load-balancing.
  - The network agent
  - Responsible for dynamic updates and maintenance of all networking rules
  - Also responsible for TCP, UDP, and SCTP stream forwarding or random forwarding across a set of Pod backends of an application
- **Container Runtime**
  - Software used to run containers.
  - Common runtimes: containerD, Docker, CRI-O.
- Add-ons
  - DNS
  - Dashboard
  - Monitoring
  - Logging
  - Device Plugins

Pods and Containers

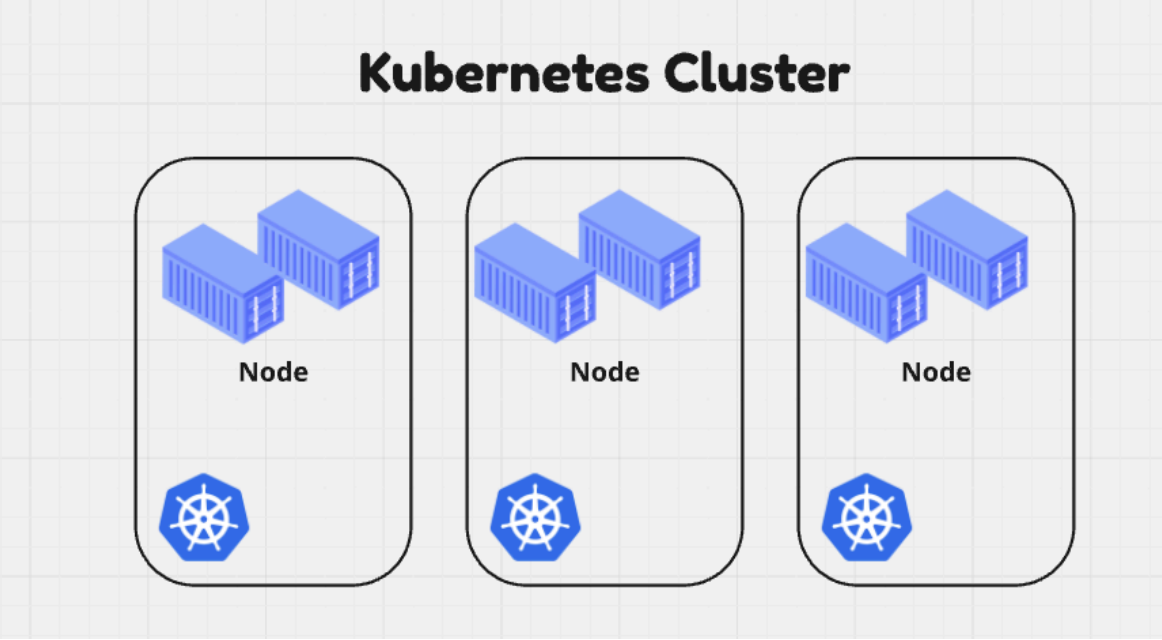
- A **Pod** is the smallest deployable unit in Kubernetes.
- It can host one or more **containers** that share storage, network, and a specification for how to run them.

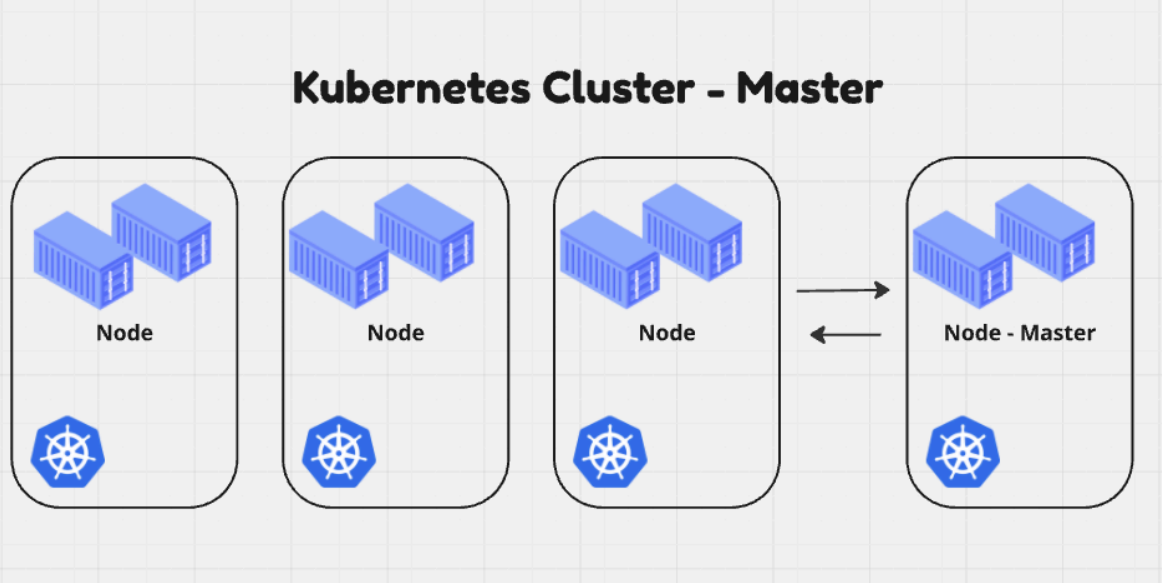
Flow Example

1. You `kubectl apply` a deployment YAML.
2. The **API Server** receives it.
3. The **Scheduler** places the Pod on a Node.
4. The **kubelet** on that Node pulls the container image and runs the Pod.
5. The **kube-proxy** ensures networking rules allow access to the Pod.
6. The **Controller Manager** ensures the Pod count remains as specified.

Diagram

```
User → kubectl
        ↓
    [API Server] ↔ etcd
        ↓
[Scheduler]   [Controller Manager]
        ↓
     [Nodes]
        ↓
  [kubelet, kube-proxy, runtime]
        ↓
      [Pods]
        ↓
   [Containers]
```





Master → Another node with Kubernetes installed on it and configured as a Master, it watches over the nodes in the cluster and is responsible for the actual orchestration of containers on the worker node.s

Master Node (Control Plane)

The **master node** is the **brain** of the Kubernetes cluster.

Responsibilities:

- **Cluster management**
- **Scheduling Pods**
- **Maintaining cluster state**
- **Monitoring & reacting to changes**

Core Components on the Master Node:

Component	Purpose
`kube-apiserver`	Entry point for all Kubernetes commands (kubectl, dashboards).
`etcd`	Stores all cluster state and configuration (key-value store).
`kube-scheduler`	Decides which Node should run a Pod.
`kube-controller-manager`	Runs background controllers that manage cluster state.
`cloud-controller-manager` (optional)	Interacts with cloud providers like AWS or GCP.

📍 There can be **multiple master nodes** in a production cluster for **high availability**.

Worker Nodes (Cluster Nodes)

These are the machines (physical or virtual) that **run your applications** (Pods).

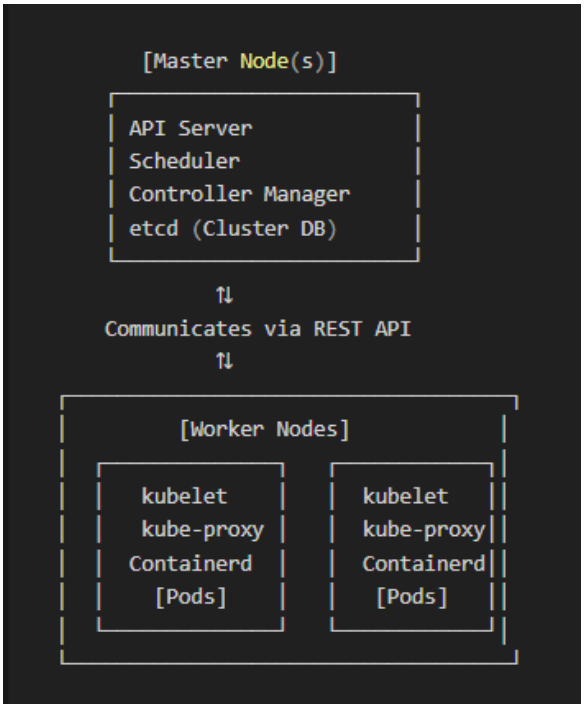
Responsibilities:

- Run containers inside Pods
- Communicate with the master node
- Report status and metrics

Components on Each Worker Node:

Component	Purpose
`kubelet`	Talks to the API Server; ensures containers are running in Pods.
`kube-proxy`	Handles networking and load balancing to Pods.
Container Runtime (e.g., containerd, Docker)	Actually runs containers.

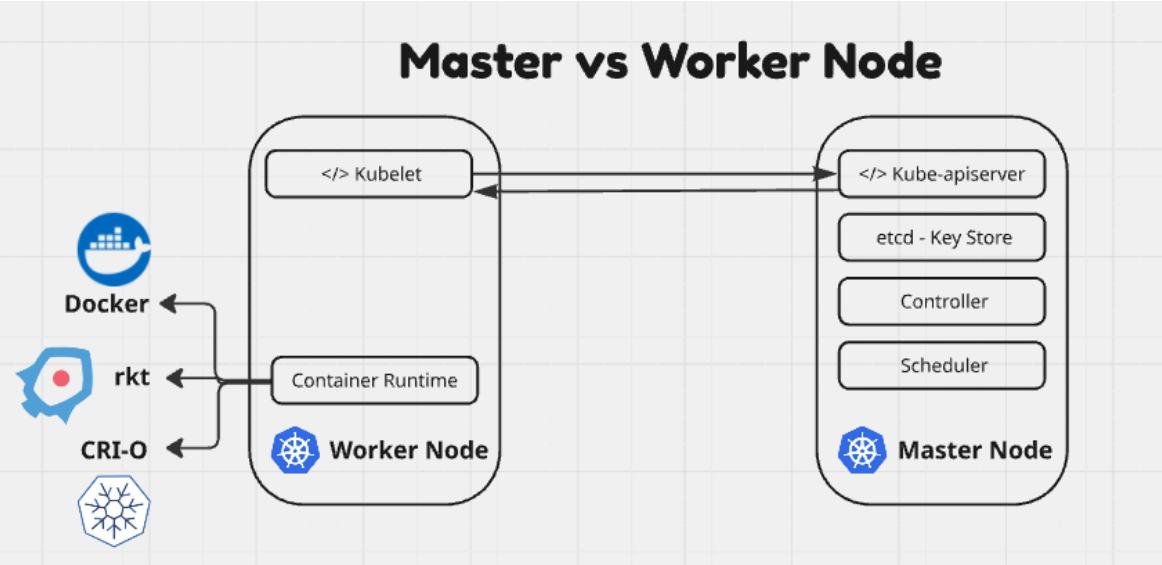
Visual Summary



Real-World Analogy

- **Master Node** = Air Traffic Control Tower (makes decisions)
- **Worker Nodes** = Airplanes (carry the applications, follow directions)

Master vs Worker Nodes



Networking Challenges

Networking in Kubernetes is powerful, but can be complex and full of challenges.

Common Networking Challenges in Kubernetes

1. Cluster Networking Complexity

- Kubernetes uses a flat, virtual network where each pod gets its IP.
- **Challenge**: Managing this at scale requires understanding **overlay networks**, **CNI plugins**, and **routing rules**.

2. CNI Plugin Limitations

- Kubernetes uses Container Network Interface (CNI) plugins (e.g., Calico, Flannel, Cilium).
- **Challenge**: Different plugins support different features (e.g., network policies, eBPF, IPv6).
- **Impact**: Selecting or misconfiguring a plugin can cause unpredictable behavior.

3. Service Discovery & Load Balancing

- Kubernetes services use **kube-proxy** and **iptables/ipvs** for internal load balancing.
- **Challenge**: Performance can degrade with many services or when using iptables in large clusters.

4. DNS Resolution Issues

- DNS (via CoreDNS) is used for service discovery.
- **Challenge**: Misconfigured DNS, overloaded CoreDNS pods, or DNS caching issues can cause intermittent service resolution failures.

5. Network Policies

- Network policies restrict traffic between pods for security.
- **Challenge**: They are not enforced by default and depend on the CNI plugin's support.
- **Common Pitfall**: Unintentionally blocking traffic or exposing services due to misconfigured policies.

6. Ingress Management

- Ingress controllers (like NGINX, Traefik) manage external HTTP(S) traffic to services.
- **Challenge**: Misconfigured Ingress resources or a lack of TLS setup can expose services or cause downtime.

7. Multi-Tenancy and Isolation

- Running multiple teams/apps securely in one cluster.
- **Challenge**: Enforcing strict network segmentation, especially without proper network policy or namespace isolation.

8. Egress Control

- Controlling outbound traffic from pods to external resources.
- **Challenge**: Kubernetes does not manage egress by default. Requires tools like egress gateways or firewalls.

9. Debugging Networking Issues

- **Challenge**: Tools like `kubectl`, `tcpdump`, or `netshoot` are needed, but many issues (e.g., dropped packets, DNS errors) are hard to diagnose without deep knowledge.

10. Cloud-Specific Networking

- Cloud providers implement networking differently (e.g., AWS VPC CNI vs. Azure CNI).
- **Challenge**: Networking issues often stem from cloud resource limits, NAT gateways, or IP exhaustion.

Kubernetes networking models and their challenges

1. Container-to-Container Communication Inside Pods

- Port conflicts can occur since containers must **not use the same ports**.
- Debugging is harder if logs and network tools are not uniformly installed across containers.
- Poorly designed sidecar or helper containers can affect the main app's network behavior.

2. **Pod-to-Pod Communication (Same Node and Across Nodes)**

- Requires a CNI plugin (e.g., Calico, Flannel, Cilium) to handle routing and connectivity.
- Pod IPs are **ephemeral;** they change if the pod is restarted or rescheduled.
- **Cross-node latency** or network segmentation can cause performance issues or dropped connections.

3. **Service-to-Pod Communication (Same Namespace and Cross-Namespace)**

- Service discovery via DNS can fail if CoreDNS is overloaded or misconfigured.
- **Cross-namespace communication** can be unintuitive or blocked by **network policies or RBAC**.
- Delays in pod readiness/liveness probes can lead to service routing to **unhealthy pods**.

4. **External-to-Service Communication**

- **NodePort** is insecure and not scalable.
- **LoadBalancer** can be expensive or limited by cloud provider quotas.
- **Ingress** setup can be complex and requires proper TLS configuration.
- External firewalls, DNS settings, or misconfigured ingress rules can block access.

Summary Table

Communication Type	Method	Key Challenge
Container-to-Container in Pod	`localhost` / shared network	Port conflicts, debugging
Pod-to-Pod (same/different nodes)	Direct pod IP	IP changes, CNI plugin behavior
Service-to-Pod (intra/inter namespace)	DNS + kube-proxy load balancing	DNS issues, network policies, cross-namespace
External-to-Service	NodePort / LoadBalancer / Ingress	Security, cost, ingress configuration

Kubectl

The Kubernetes Command Line Tool

`kubectl` is the **primary CLI tool** used to interact with a Kubernetes cluster. It lets you **manage and inspect cluster resources** by communicating with the **Kubernetes API Server**.

What `kubectl` Does

- Sends commands to the **API server**
- Retrieves logs, resource status, and cluster info
- Applies YAML configuration files
- Scales deployments, deletes resources, and more

Common `kubectl` Commands

General

Command	Description
`kubectl version`	Show client and server versions
`kubectl cluster-info`	Show cluster information
`kubectl config view`	Show current kubeconfig details

Working with Resources

Command	Description
`kubectl get pods`	List all pods
`kubectl get nodes`	List all nodes
`kubectl get services`	List services
`kubectl describe pod [pod-name]`	Detailed info about a pod
`kubectl logs [pod-name]`	View logs from a pod

Deployments & YAML

Command	Description
`kubectl apply -f [file.yaml]`	Apply a configuration file
`kubectl delete -f [file.yaml]`	Delete resources from file
`kubectl edit deployment [name]`	Edit a running deployment

`kubectl scale deployment [name] --replicas=3`	Scale to 3 pods

Debugging

Command	Description
`kubectl exec -it [pod-name] -- /bin/bash`	Get shell access to a pod
`kubectl port-forward [pod-name] 8080:80`	Access a pod locally
`kubectl top pods`	View pod resource usage (needs Metrics Server)

kubectl + YAML

YAML files define Kubernetes resources like Pods, Deployments, and Services.

Example:

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

Run it:

```
kubectl apply -f deployment.yaml
```

Tips

- Use `kubectl get all` to see everything in the current namespace.

- Add `A` or `-all-namespaces` to query all namespaces.
- Use `kubectl explain` to learn the schema of any resource.

CRI (Container Runtime Interface)

What is CRI?

CRI is an API **standard defined by Kubernetes** that allows the **Kubelet** to interact with various container runtimes (e.g., containerd, CRI-O, Docker shim).

Why does it exist?

Before CRI, Kubernetes only worked directly with Docker. CRI makes Kubernetes **runtime-agnostic**, meaning it can work with any container runtime that implements CRI.

How it works:

- The **Kubelet** uses the **CRI** to send commands such as "start pod", "stop container", and "pull image".
- The **container runtime** (e.g., containerd or CRI-O) receives and processes these requests.

Common CRI-compliant runtimes:

Runtime	Description
containerd	Lightweight, production-grade runtime (originally part of Docker)
CRI-O	Runtime optimized for Kubernetes, used in OpenShift
Docker shim	Deprecated adapter to use Docker with CRI

OCI (Open Container Initiative)

What is OCI?

OCI is a **standardization initiative** under the Linux Foundation aimed at ensuring compatibility across various container technologies.

What it provides:

1. **OCI Image Specification**
   - Defines how container images should be structured (layers, configs, manifests).
2. **OCI Runtime Specification**
   - Defines how to run a container (process, mounts, namespaces, etc.).

OCI-compliant tools:

Tool	Role
runc	Low-level runtime that follows OCI runtime spec (used by containerd, Docker, CRI-O)
Buildah	OCI-compliant image builder
Podman	OCI runtime and image tool

CRI vs OCI – Key Differences

Aspect	CRI	OCI
Scope	Kubernetes-specific runtime interface	Industry-wide container standards
Created By	Kubernetes project (CNCF)	Open Container Initiative (Linux Foundation)
Purpose	Abstract container runtimes for kubelet	Standardize images and runtimes
Example	CRI-O, containerd	runc, image format standards
Interface Type	API interface	Specification and implementation

Visualization

```
Kubelet
  ↓ (CRI)
Runtime (e.g., containerd, CRI-O)
  ↓ (OCI Runtime Spec)
runc
```

Summary

- **CRI**: Kubernetes plugs into different container runtimes.
- **OCI**: Standardizes how containers and images work across all tools and platforms.

What is containerd?

Containerd is a **high-performance container runtime** that manages the entire lifecycle of containers on a host system.

> It is CRI-compliant and OCI-compatible, making it a core part of the modern Kubernetes ecosystem.

Why containerd?

Originally a part of Docker, **containerd** was extracted to be a **standalone, reusable, and Kubernetes-friendly runtime**. It's now used by:

- **Docker** (under the hood)
- **Kubernetes** (via CRI plugin)
- **Amazon EKS**, **GKE**, **Azure AKS**, etc.

What Containerd does:

- Pulls container images from registries
- Manages container image storage
- Creates and runs containers using `runc` (OCI runtime)
- Handles networking, snapshots, and logging
- Implements the **Container Runtime Interface (CRI)** used by **kubelet**

---

containerd Architecture

```
[ Kubelet ]
    ↑↓ (CRI gRPC)
[ containerd ]
    ↑↓ (OCI)
[ runc ]
```

Key Components:

Component	Role
`containerd`	Main daemon/service
`containerd-shim`	Keeps containers alive if `containerd` crashes; separates container lifecycle
`runc`	Low-level runtime that starts containers (follows OCI spec)

Basic Usage (CLI with `ctr`)

Although `ctr` is mostly for debugging or advanced use (containerd is used **via Kubernetes**), here are a few examples:

```
# Check containerd version
containerd --version

# Pull an image
```

```
ctr image pull docker.io/library/nginx:latest

# Run a container
ctr run -d docker.io/library/nginx:latest mynginx

# List containers
ctr containers list
```

Features

- **Lightweight** and minimal dependencies
- **Built-in CRI plugin** for Kubernetes
- **OCI image & runtime compliant**
- **Integrated snapshotting & logging**
- Works with **namespaces**, **plugins**, and **gRPC APIs**

In Kubernetes

Most modern Kubernetes clusters (like those from GKE, EKS, and AKS) now **use containerd instead of Docker**.

> 📌 As of Kubernetes v1.24+, Docker is deprecated as a runtime. Use containerd or CRI-O.

Summary

Feature	containerd
Type	Container runtime
CRI-compliant?	Yes
OCI-compliant?	Yes (via `runc`)
Used in K8s?	Replaces Docker runtime
Maintained by	CNCF (Cloud Native Computing Foundation)

Pods

In Kubernetes, a **Pod** is the smallest and simplest unit in the Kubernetes object model that you can create or deploy.

What is a Pod?

A **Pod** represents a **single instance of a running process** in your cluster. It can contain **one or more containers** (usually one), which share:

- **Network** (IP address and port space)
- **Storage** (volumes)
- **Configuration** (environment variables, secrets)

Why Use Pods?

- To **run containers together** that need to share resources.
- To group **tightly coupled containers**, like a main app and its helper.

Example: A web server container and a sidecar container that handles logging.

Pod Lifecycle

1. **Pending** – Pod is accepted, but not yet scheduled to a node.
2. **Running** – Containers in the Pod are running.
3. **Succeeded** – All containers completed successfully.
4. **Failed** – One or more containers failed.
5. **Unknown** – Pod status can't be determined.

Pod Example (YAML)

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

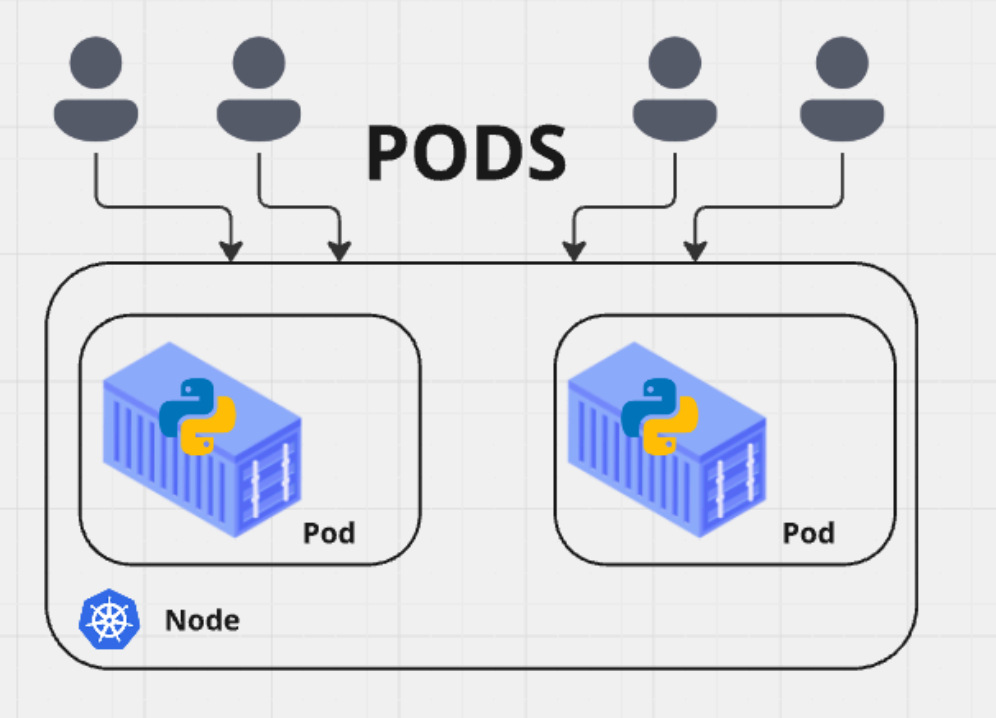
Pods Are Ephemeral

- Pods **do not self-heal;** if a pod dies, it won’t restart automatically.

- Use **Deployments, ReplicaSets, StatefulSets**, etc., to manage pod lifecycle and scaling.

Summary

Component	Description
Pod	The smallest deployable unit in Kubernetes
Contains	One or more containers
Shared	Network, storage, configuration
Managed by	Higher-level controllers like Deployments



💡 Pods have a one-to-one relationship with containers. To scale up, create new pods, and to scale down, delete existing pods.

💡 Do not add a container to an existing pod to scale your application

Multi-Container Pods

Multi-Container Pods in Kubernetes

A **multi-container pod** is a pod that runs **more than one container**, and all containers within the pod:

- **Share the same network namespace** (same IP & port space)
- **Can communicate via** `localhost`
- **Share storage volumes** (if defined)

Why Use Multi-Container Pods?

Multi-container pods are ideal when containers are **tightly coupled** and must **work together**.

They can be created and destroyed together

Common patterns:

Pattern	Description
Sidecar	Extends or enhances the main container (e.g., logging, monitoring, proxies)
Ambassador	Proxy to other networks or services
Adapter	Transforms or relays data between systems

Real-World Example: NGINX + Log Collector

```
apiVersion: v1
kind: Pod
metadata:
  name: multi-container-pod
spec:
  containers:
  - name: nginx
    image: nginx
    ports:
    - containerPort: 80
    volumeMounts:
    - name: shared-logs
      mountPath: /var/log/nginx

  - name: log-agent
    image: busybox
    command: ["sh", "-c", "tail -f /var/log/nginx/access.log"]
    volumeMounts:
    - name: shared-logs
      mountPath: /var/log/nginx

  volumes:
```

```
  - name: shared-logs
    emptyDir: {}
```

Key Concepts

Concept	Explanation
`emptyDir`	Temporary directory shared between containers (created when pod starts)
`volumeMounts`	Mount points for containers to read/write shared data
`localhost`	Containers can talk to each other directly via `localhost:<port>`

Best Practices

- Use multi-container pods **only when containers must be deployed together**.
- For loosely coupled components, prefer **separate pods** managed by a **Service**.
- Design containers with **separation of concerns;** each container should have a focused role.

Kubernetes Configurations

1. All-in-One Single-Node Installation

Description:

- A single node runs all Kubernetes components: control plane (API server, scheduler, controller manager), etcd, and worker components (kubelet, kube-proxy, container runtime).
- Tools like `minikube`, `kind`, and `k3s` are often used for this setup.

Use Case:

- Ideal for local development, learning, and testing lightweight applications.

Limitations:

- No fault tolerance—if the node fails, the entire cluster goes down.
- Resource contention between the control plane and workload containers.
- Not suitable for production environments.

2. Single-Control Plane and Multi-Worker Installation

Description:

- One node is dedicated to the control plane.
- Two or more nodes serve as workers, hosting application pods.

Use Case:

- Suitable for small-scale staging environments or non-critical workloads.

Limitations:

- The control plane is a single point of failure.
- If the control plane fails, existing workloads remain active, but new deployments, scaling, or recovery operations cannot be performed.

---

3. Single-Control Plane with Single-Node etcd, and Multi-Worker Installation

Description:

- Control plane with an external etcd instance.
- Multiple separate worker nodes handle application workloads.

Use Case:

- Slightly more robust than configuration #2, with clearly separated worker and control components.

Limitations:

- Etcd remains a single point of failure.
- No high availability or redundancy; cluster health depends on the single control/etcd node.

---

4. Multi-Control Plane and Multi-Worker Installation

Description:

- Multiple nodes host the control plane components, usually in an HA configuration.
- Worker nodes run workloads and scale independently.

Use Case:

- Appropriate for production-grade clusters requiring high availability and resilience.

Limitations:

- Increased complexity in setup and maintenance.
- Requires external or highly available etcd configuration.

---

5. Multi-Control Plane with Multi-Node etcd, and Multi-Worker Installation

Description:

- A highly available cluster with:
  - Three or more control plane nodes.
  - A distributed etcd cluster (typically 3 or 5 nodes).

- Multiple worker nodes for workloads.

Use Case:

- Enterprise-level production deployments require fault tolerance, scalability, and resilience.

Limitations:

- Highest complexity and cost in terms of infrastructure and operational overhead.
- Requires load balancers, secure etcd configuration, and robust monitoring.

Summary Table

Configuration	Control Plane Nodes	etcd Nodes	Worker Nodes	Suitable For	High Availability
All-in-One Single Node	1	1	1	Learning/Testing	No
Single-Control Plane, Multi-Worker	1	1	2+	Staging/Dev	No
Single-Control Plane, Single etcd, Multi-Worker	1	1	2+	Staging/Dev	Low
Multi-Control Plane, Multi-Worker	3+	External	2+	Production	Yes (partial)
Multi-Control Plane, Multi-etcd, Multi-Worker	3+	3+	2+	Enterprise/Production	Yes (full

---

Pod Deployment

Deploy a Pod Using a YAML File

Step 1: Create a YAML file (e.g., `pod.yaml`)

```
apiVersion: v1
kind: Pod
metadata:
  name: my-nginx-pod
spec:
  containers:
  - name: nginx-container
```

```
    image: nginx
    ports:
    - containerPort: 80
```

Step 2: Apply the YAML

```
kubectl apply -f pod.yaml
```

Step 3: Check Pod Status

```
kubectl get pods
kubectl describe pod my-nginx-pod
```

---

2. Deploy a Pod Using a `kubectl` Command

```
kubectl run my-nginx --image=nginx --restart=Never --port=80
```

- `-restart=Never` ensures it's just a Pod, **not a Deployment**.

Recommended Way: Use a Deployment

Why? Because Pods are **ephemeral,** if they crash, they don't restart automatically. A **Deployment** manages Pod replicas and self-healing.

Sample Deployment YAML

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: nginx-deployment
spec:
  replicas: 2
  selector:
    matchLabels:
      app: nginx
  template:
    metadata:
      labels:
        app: nginx
    spec:
```

```
    containers:
    - name: nginx
      image: nginx
      ports:
      - containerPort: 80
```

Deploy it:

```
kubectl apply -f deployment.yaml
kubectl get deployments
kubectl get pods
```

Minikube

What is Minikube?

Minikube is a **tool that lets you run a local Kubernetes cluster** on your laptop or workstation. It's ideal for **learning, development, and testing** Kubernetes workloads without needing access to a full cloud-based cluster.

Why Use Minikube?

- Learn Kubernetes locally without cloud costs.
- Develop and test apps in a real K8s environment
- Works offline
- Lightweight and fast to set up

Key Features

Feature	Description
Local Cluster	Runs a single-node Kubernetes cluster locally
Container Support	Supports Docker, containerd, or CRI-O runtimes
Addons	Built-in tools like dashboard, metrics-server, Ingress
Port Forwarding	Expose services to your host machine easily
Multi-Cluster	Can spin up multiple clusters with different profile

---

Use Cases

- Practice **Kubernetes YAML**
- Run **Helm charts locally**
- Build and test **CI/CD pipelines**
- Learn **Kubernetes controllers** (Deployments, Services, Ingress)

Limitations

- Not suitable for **production**
- Uses system resources (RAM/CPU) like a small VM
- Single-node only (for most use cases)

Minikube vs. Kind vs. k3s

Tool	Use Case
Minikube	Local full K8s setup for dev/testing
Kind	Kubernetes in Docker (great for CI/CD pipelines)
k3s	Lightweight K8s distro (great for edge or IoT)

1. Access the Kubernetes Cluster (kubectl)

After starting Minikube:

```
minikube start
kubectl get nodes
```

Use `kubectl` to interact with the cluster.

2. Access Services (Minikube service command)

To access a service exposed with `NodePort`:

```
minikube service <service-name>
```

This will open the default browser to the service endpoint.

Example:

```
kubectl expose deployment myapp --type=NodePort --port=8080
minikube service myapp
```

3. Access Minikube Dashboard (UI)

Launch the Kubernetes dashboard:

```
minikube dashboard
```

This opens a web-based dashboard to manage your cluster visually.

---

4. Get Minikube IP

Useful if accessing services manually:

```
minikube ip
```

Then access a service like this:

```
http://<minikube-ip>:<node-port>
```

Summary

Minikube = Your personal Kubernetes playground.

Installation Process on Ubuntu

Kubernetes Set Up Documentation

Installing local learning Kubernetes clusters:

1. **Minikube**
2. **Kind (Kubernetes IN Docker)**
3. **k3s (Lightweight Kubernetes)**
4. **MicroK8s**
5. **Docker Desktop (with Kubernetes enabled)**
6. **Rancher Desktop**

Kubernetes cluster bootstrapping

1. kubeadm

Purpose:

- Official tool from Kubernetes to **manually bootstrap** a cluster.

Key Features:

- Initializes the control plane and joins the worker nodes.
- Minimal automation – you manage networking, load balancers, etc.
- Best for **learning**, **custom setups**, **on-prem**, or **bare-metal**.

Pros:

- Lightweight, official, and well-documented.
- Fine-grained control over components.
- Works on most Linux environments.

Cons:

- No built-in infrastructure provisioning.
- Requires more manual work for HA clusters.

---

2. Kubespray

Purpose:

- Uses **Ansible** to deploy Kubernetes clusters on **any infrastructure** (cloud, bare-metal, hybrid).

Key Features:

- Automated provisioning of OS dependencies, kubeadm, networking, and more.
- Supports multi-node, multi-master HA configurations.

Pros:

- Flexible and infrastructure-agnostic.
- Great for **automated, repeatable setups**.
- Ideal for **on-prem** or **custom environments**.

Cons:

- More complex than kubeadm.
- Requires knowledge of Ansible.

---

3. kOps (Kubernetes Operations)

Purpose:

- **Production-grade tool** for bootstrapping, upgrading, and managing Kubernetes clusters on **AWS** (also supports GCP, DigitalOcean, etc.).

Key Features:

- Provisions both the infrastructure (via Terraform or cloud APIs) and the cluster.
- Built-in DNS, HA setup, secrets, and rolling upgrades.

Pros:

- End-to-end cluster management.
- Ideal for cloud-native Kubernetes on AWS.
- Automates everything, including upgrades.

Cons:

- Primarily tied to cloud providers.
- Limited on-prem support.

Summary

Tool	Best For	Infra Provisioning	OS Automation	HA Support	Skill Required
kubeadm	Learning, bare-metal setup	❌ No	❌ Manual	✅ Yes (manual)	Low–Medium (K8s-focused)
Kubespray	On-prem, hybrid, automation	❌ No	✅ Yes (Ansible)	✅ Yes	Medium–High (Ansible + K8s)
kOps	AWS-based production use	✅ Yes	✅ Yes	✅ Yes	Medium (Cloud + K8s)

Step 1: Download the latest release of kubectl

```
curl -LO "https://dl.k8s.io/release/$(curl -Ls https://dl.k8s.io/release/stable.txt)/bin/linux/amd64/kubectl"
```

Step 2: Make it executable

```
chmod +x kubectl
```

Step 3: Move to a directory in your PATH

```
sudo mv kubectl /usr/local/bin/
```

Step 4: Verify installation

```
kubectl version --client
```

---

2. Install Minikube on Ubuntu

Step 1: Download Minikube

```
curl -LO https://storage.googleapis.com/minikube/releases/latest/minikube-linux-amd64
```

Step 2: Install it

```
sudo install minikube-linux-amd64 /usr/local/bin/minikube
```

Step 3: Verify installation

```
minikube version
```

---

3. Start Minikube

Minikube requires a **hypervisor,** such as Docker, VirtualBox, or KVM. Assuming Docker is installed:

```
minikube start --driver=docker
```

You can also try:

```
minikube start
```

> 📝 Tip: If you don’t have Docker installed, run:

Install Docker on Ubuntu (Official Way)

🔷 Step 1: Update APT and install dependencies

```
sudo apt update
sudo apt install -y ca-certificates curl gnupg lsb-release
```

🔷 Step 2: Add Docker's official GPG key

```
sudo mkdir -p /etc/apt/keyrings

curl -fsSL https://download.docker.com/linux/ubuntu/gpg | \
  sudo gpg --dearmor -o /etc/apt/keyrings/docker.gpg
```

🔷 Step 3: Add Docker's stable repository

```
echo \
  "deb [arch=$(dpkg --print-architecture) signed-by=/etc/apt/keyrings/docker.gpg] \
  https://download.docker.com/linux/ubuntu \
  $(lsb_release -cs) stable" | \
  sudo tee /etc/apt/sources.list.d/docker.list > /dev/null
```

🔷 Step 4: Update APT again and install Docker

```
sudo apt update
sudo apt install -y docker-ce docker-ce-cli containerd.io docker-buildx-plugin docker-compose
-plugin
```

🔷 Step 5: Verify Docker

```
docker --version
```

🔷 Optional: Run Docker as a non-root user

```
sudo usermod -aG docker $USER
newgrp docker
```

Once done, you can start Minikube using the Docker driver:

```
minikube start --driver=docker
```

Check Minikube status, and if Kubectl is working.

```
minikune status

kubectl get nodes
```

Deploy an application and check the deployments.

```
kubectl create deployment hello-minikube1 --image=kicbase/echo-server:1.0

kubectl get deployments
```

Expose the deployment

```
kubectl expose deployment hello-minikube1 --type=LoadBalancer --port=8080
```

Get the URL of the service.

```
minikube service hello-minikube1 --url
```

Running pods

```
kubectl run nginx --image=nginx
```

Get pods

```
kubectl get pods
```

to get more information

```
kubectl describe pod nginx
```

The kubeconfig includes the API Server's endpoint server: https://192.168.99.100:8443 and the minikube user's client authentication key and certificate data.

```
kubectl config view
```

Install K3S - Lighter weight Cluster

1- Run the K3s Installer

```
curl -sfL https://get.k3s.io | sh -
```

This will:

- Install K3s
- Start a single-node Kubernetes cluster
- Set up systemd service to auto-start K3s
- Deploy `kubectl` as `k3s kubectl`

2- Use `kubectl`

By default, `kubectl` is bundled with K3s as:

```
sudo k3s kubectl get nodes
```

For convenience, create an alias:

```
echo "alias kubectl='sudo k3s kubectl'" >> ~/.bashrc
source ~/.bashrc
```

Now you can just use:

```
kubectl get pods -A
```

3- Test Rollout Commands

Example test:

```
kubectl create deployment my-app --image=nginx:1.14
kubectl rollout status deployment/my-app
kubectl set image deployment/my-app nginx=nginx:1.16
kubectl rollout undo deployment/my-app
```

Optional: View Cluster Info

```
kubectl cluster-info
kubectl get nodes
kubectl get deployments
```

Pro Tip: Use YAML

Try this deployment.yaml:

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: my-app
spec:
  replicas: 2
  selector:
    matchLabels:
      app: my-app
  template:
    metadata:
      labels:
        app: my-app
    spec:
      containers:
      - name: nginx
        image: nginx:1.14
```

Apply it:

```
kubectl apply -f deployment.yaml
```

Then trigger and test rollouts:

```
kubectl set image deployment/my-app nginx=nginx:1.16
kubectl rollout status deployment/my-app
```

Cleanup

To uninstall K3s:

```
sudo /usr/local/bin/k3s-uninstall.sh
```

Kubernetes Dashboard

What is the Kubernetes Dashboard?

The **Kubernetes Dashboard** is a web-based UI that lets you:

- Monitor cluster resources (nodes, pods, services)
- Deploy and manage applications
- View logs and troubleshoot
- Perform basic administrative tasks

It’s especially useful when you're new to Kubernetes or need visual feedback.

How to Launch the Kubernetes Dashboard with Minikube

1. **Start Minikube** (if not already running):

   ```
   minikube start
   ```

2. **Launch the Dashboard**:

   ```
   minikube dashboard
   ```

   This:

   - Starts the Dashboard (if not already running)
   - Opens it in your default web browser

   You’ll see a UI to explore nodes, namespaces, pods, deployments, etc.

```
minikube addons list
minikube addons enable metrics-server
minikube addons enable dashboard
minikube addons list
minikube dashboard
# if the browser doesnt open use:
minikube dashboard --url
```

APIs and kubectl proxy

What `kube-proxy` Does

`kube-proxy` handles **service networking** in Kubernetes. It enables stable service IPs and load-balances traffic across the correct backend Pods.

Key responsibilities:

1. **Implements Kubernetes Services**
   - It ensures traffic sent to a Service IP is forwarded to one of the correct backend Pods.
   - Uses either **iptables**, **ipvs**, or **userspace** mode to manage routing rules.
2. **Service Discovery & Load Balancing**
   - Watches the Kubernetes API for `Service` and `Endpoint` objects.
   - Maintains routing rules that forward requests to matching Pods.
3. **Handles Traffic Between Pods**
   - Works with the CNI (Container Network Interface) plugin to allow Pods on different nodes to communicate.

`kube-proxy` in k3s

In **k3s**, `kube-proxy` is replaced by a **lightweight implementation** (or even skipped in some CNI configurations like `flannel` + `host-gw`). It's built into the k3s binary and may use **iptables** or **IPVS** mode depending on how it's set up.

You can check how it's running using:

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

or check the process directly:

```
ps aux | grep kube-proxy
```

If it’s enabled, you can see its logs with:

```
kubectl logs -n kube-system -l k8s-app=kube-proxy
```

When to Think About `kube-proxy`

- If you're debugging network issues (e.g., a Service isn't routing correctly)
- If you're building custom CNI solutions or need high-performance routing
- If you're tuning for large clusters and want to switch to IPVS mode

---

API Examples You Can Try

1. List all Pods (default namespace)

```
curl http://localhost:8001/api/v1/namespaces/default/pods
```

2. List all Services (across all namespaces)

```
curl http://localhost:8001/api/v1/services
```

Stop the Proxy

To stop the proxy, just press `Ctrl+C` in the terminal where it's running.

YAML in Kubernetes

Basic YAML Structure in Kubernetes

```
apiVersion: v1
kind: Pod
metadata:
  name: my-pod
spec:
  containers:  #list or array
    - name: nginx #1st item in the list
      image: nginx:latest
```

Key Components:

- `apiVersion`: Specifies which version of the Kubernetes API to use.
- `kind`: The type of resource (e.g., Pod, Service, Deployment).
- `metadata`: Identifiers such as name, labels, and annotations.
- `spec`: The desired state/configuration of the resource.

To create it, we can use kubectl apply -f pod.yml (example)

Common Kubernetes Resources Defined via YAML

Resource	Kind	Purpose
Pod	`Pod`	Smallest deployable unit
Deployment	`Deployment`	Manages ReplicaSets and Pod rollouts
Service	`Service`	Exposes Pods to the network
ConfigMap	`ConfigMap`	Externalize configuration as key-values
Secret	`Secret`	Stores sensitive information
Ingress	`Ingress`	Manages external access (e.g., HTTP)
PersistentVolumeClaim	`PersistentVolumeClaim`	Request storage resources

Best Practices

- Always include **labels** for grouping and selecting resources.
- Use **comments** to annotate complex sections.
- Validate with `kubectl apply --dry-run=client -f file.yaml` before applying.
- Use **indentation (2 spaces)** consistently — YAML is indentation-sensitive.

YAML in Kubernetes – Practice Questions

Section 1: Basics

1. What does `apiVersion` represent in a Kubernetes YAML file?
2. What is the purpose of the `kind` field in a YAML definition?
3. Name three top-level fields (root keys) commonly found in a Kubernetes YAML file.
4. Why is indentation important in YAML?
5. What file extension is commonly used for Kubernetes resource definitions?

Section 2: Resource Definitions

1. What kind of Kubernetes resource is defined by this YAML?

```
apiVersion: v1
kind: Pod
metadata:
  name: busybox-pod
spec:
  containers:
    - name: busybox
```

```
    image: busybox
    command: ["sleep", "3600"]
```

2. Which field would you modify to change the container image used by a Pod?
3. In a Deployment YAML, which field defines how many Pod replicas should run?
4. What is the difference between a `ConfigMap` and a `Secret` in YAML?
5. How do you apply a YAML file in Kubernetes using `kubectl`?

Section 3: Troubleshooting and Validation

1. What command can you use to check if a YAML file is syntactically correct (client-side dry run)?
2. What will happen if you define a field with incorrect indentation in a YAML file?
3. Can you use multiple resources in a single YAML file? If yes, how?
4. How can you generate a YAML file from an existing Kubernetes resource?
5. True or False: You must always use double quotes for strings in YAML.

Answers

1. It specifies the version of the Kubernetes API to use for the resource.
2. It declares the type of resource being created (e.g., Pod, Service, Deployment).
3. `apiVersion`, `kind`, `metadata`, and `spec`.
4. YAML relies on indentation (spaces, not tabs) to define structure—incorrect indentation causes parsing errors.
5. `.yaml` or `.yml`
6. A **Pod**.
7. `spec.containers[].image`
8. `spec.replicas`
9. `ConfigMap` stores non-sensitive config data; `Secret` stores base64-encoded sensitive data.
10. `kubectl apply -f filename.yaml`
11. `kubectl apply --dry-run=client -f filename.yaml`
12. Kubernetes will fail to parse and reject the file with a syntax error.
13. Yes, separate them using `---` (YAML document separator).
14. `kubectl get <resource> <name> -o yaml`
15. **False** — Quotes are optional unless the string contains special characters or ambiguity.

Write YAML Practice Questions

1. Pod Definition

Write a YAML file to create a Pod named `nginx-pod` that runs an `nginx` container using the `nginx:latest` image.

---

2. Deployment

Write a Deployment YAML named `web-deployment` that manages 3 replicas of a container using the `httpd:2.4` image. The container should expose port 80.

---

3. Service

Write a Service YAML named `web-service` that exposes the `web-deployment` on port 80 using the ClusterIP type.

---

4. ConfigMap

Write a ConfigMap named `app-config` that stores the following key-value pairs:

- `APP_MODE`: `production`
- `APP_VERSION`: `1.2.3`

---

5. Secret

Write a Secret named `db-secret` that stores:

- `username`: `admin`
- `password`: `P@ssw0rd`

  (Encode the values in base64 manually or with `echo -n 'value' | base64`)

---

6. Ingress

Write an Ingress resource named `web-ingress` that routes traffic from `example.com` to the `web-service` on port 80.

---

7. PersistentVolumeClaim

Write a PersistentVolumeClaim named `my-pvc` that requests:

- Storage: 1Gi
- Access Mode: ReadWriteOnce
- Storage Class: standard

---

8. Multi-Document YAML

Create a multi-document YAML file that contains:

- A Pod named `multi-pod` running `alpine` and sleeping for 3600 seconds
- A ConfigMap named `multi-config` with a single key `mode: debug`

1. Pod Definition: nginx-pod

```
apiVersion: v1
kind: Pod
metadata:
  name: nginx-pod
spec:
  containers:
    - name: nginx
      image: nginx:latest
```

2. Deployment: web-deployment

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: web-deployment
spec:
  replicas: 3
  selector:
    matchLabels:
      app: web
  template:
    metadata:
      labels:
        app: web
    spec:
      containers:
        - name: httpd
          image: httpd:2.4
          ports:
            - containerPort: 80
```

3. Service: web-service

```
apiVersion: v1
kind: Service
```

```
metadata:
  name: web-service
spec:
  selector:
    app: web
  ports:
    - port: 80
      targetPort: 80
  type: ClusterIP
```

4. ConfigMap: app-config

```
apiVersion: v1
kind: ConfigMap
metadata:
  name: app-config
data:
  APP_MODE: production
  APP_VERSION: "1.2.3"
```

5. Secret: db-secret

```
apiVersion: v1
kind: Secret
metadata:
  name: db-secret
type: Opaque
data:
  username: YWRtaW4=       # base64 for 'admin'
  password: UEBzc3cwcmQ=   # base64 for 'P@ssw0rd'
```

6. Ingress: web-ingress

```
apiVersion: networking.k8s.io/v1
kind: Ingress
metadata:
  name: web-ingress
spec:
  rules:
    - host: example.com
      http:
```

```
      paths:
        - path: /
          pathType: Prefix
          backend:
            service:
              name: web-service
              port:
                number: 80
```

7. PersistentVolumeClaim: my-pvc

```
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
  name: my-pvc
spec:
  accessModes:
    - ReadWriteOnce
  resources:
    requests:
      storage: 1Gi
  storageClassName: standard
```

8. Multi-Document YAML (Pod + ConfigMap)

```
apiVersion: v1
kind: Pod
metadata:
  name: multi-pod
spec:
  containers:
    - name: alpine
      image: alpine
      command: ["sleep", "3600"]
---
apiVersion: v1
kind: ConfigMap
metadata:
  name: multi-config
data:
  mode: debug
```

1. What command creates a pod using a YAML file?

Answer:

```
kubectl apply -f <file-name>.yaml
```

---

2. How do you view all running pods in the cluster?

Answer:

```
kubectl get pods
```

---

3. How do you see detailed information about a pod named `webapp`?

Answer:

```
kubectl describe pod webapp
```

---

4. What does the `STATUS` column in `kubectl get pods` show?

Answer:

It shows the **current lifecycle state** of the pod, such as:

- `Pending`
- `Running`
- `CrashLoopBackOff`
- `ImagePullBackOff`

---

5. Which section in a pod YAML file specifies the container image?

Answer:

```
spec:
  containers:
    - name: mycontainer
      image: nginx
```

---

6. How do you delete a pod named `testpod`?

Answer:

```
kubectl delete pod testpod
```

7. If you delete a pod that was created by a Deployment, what happens?

Answer:

The Deployment will **automatically recreate** the pod to maintain the desired state.

8. How can you check which container image a pod is using?

Answer:

```
kubectl get pod <pod-name> -o jsonpath="{.spec.containers[*].image}"
```

9. How do you see all pods across all namespaces?

Answer:

```
kubectl get pods --all-namespaces
```

10. How do you get the Kubernetes version (server + client)?

Answer:

```
kubectl version --short
```

11. How do you expose a pod on port 80 as a NodePort service?

Answer:

```
kubectl expose pod <pod-name> --type=NodePort --port=80
```

12. What's the default location of the `kubectl` config file on Linux?

Answer:

```
/home/<user>/.kube/config
```

Core Kubernetes Building Blocks (Essentials You MUST Know)

Kubernetes is built around a set of fundamental components that work together to manage containerized applications.

We declare it in the **spec** section

Here are the **main building blocks**, categorized by purpose:

1. Basic Objects

Resource	Description
Pod	Smallest deployable unit — wraps one or more containers
Service	Provides a stable network endpoint for a set of pods
Namespace	Logical partitioning of the cluster (like folders)

2. Workload Controllers

Controller	Description
ReplicaSet	Ensures a set number of pod replicas are running
Deployment	Manages ReplicaSets; allows rolling updates and rollbacks
StatefulSet	For stateful apps (like databases), maintains pod identity
DaemonSet	Ensures a pod runs on every (or selected) node
Job	Runs a task once to completion
CronJob	Runs jobs on a schedule (like cron tasks)

3. Networking

Component	Description
Service	Exposes pods internally or externally
Ingress	Routes HTTP/HTTPS traffic to services
NetworkPolicy	Controls traffic between pods

4. Configuration & Secrets

Resource	Description
ConfigMap	Stores config data (e.g. env vars, app settings)
Secret	Stores sensitive data (passwords, tokens) securely

5. Storage

Resource	Description
Volume	Temporary storage for a pod
PersistentVolume (PV)	Abstraction of storage provisioned by admin
PersistentVolumeClaim (PVC)	Request for storage by a user/pod

6. Cluster Components

Component	Description
Node	A machine (VM or physical) in the cluster
Kubelet	Agent that runs on each node and manages pods
kube-apiserver	Central access point for all Kubernetes commands
Controller Manager	Manages controllers like Deployments, Jobs
Scheduler	Assigns pods to nodes
etcd	Key-value store for all cluster data (backbone of Kubernetes state)

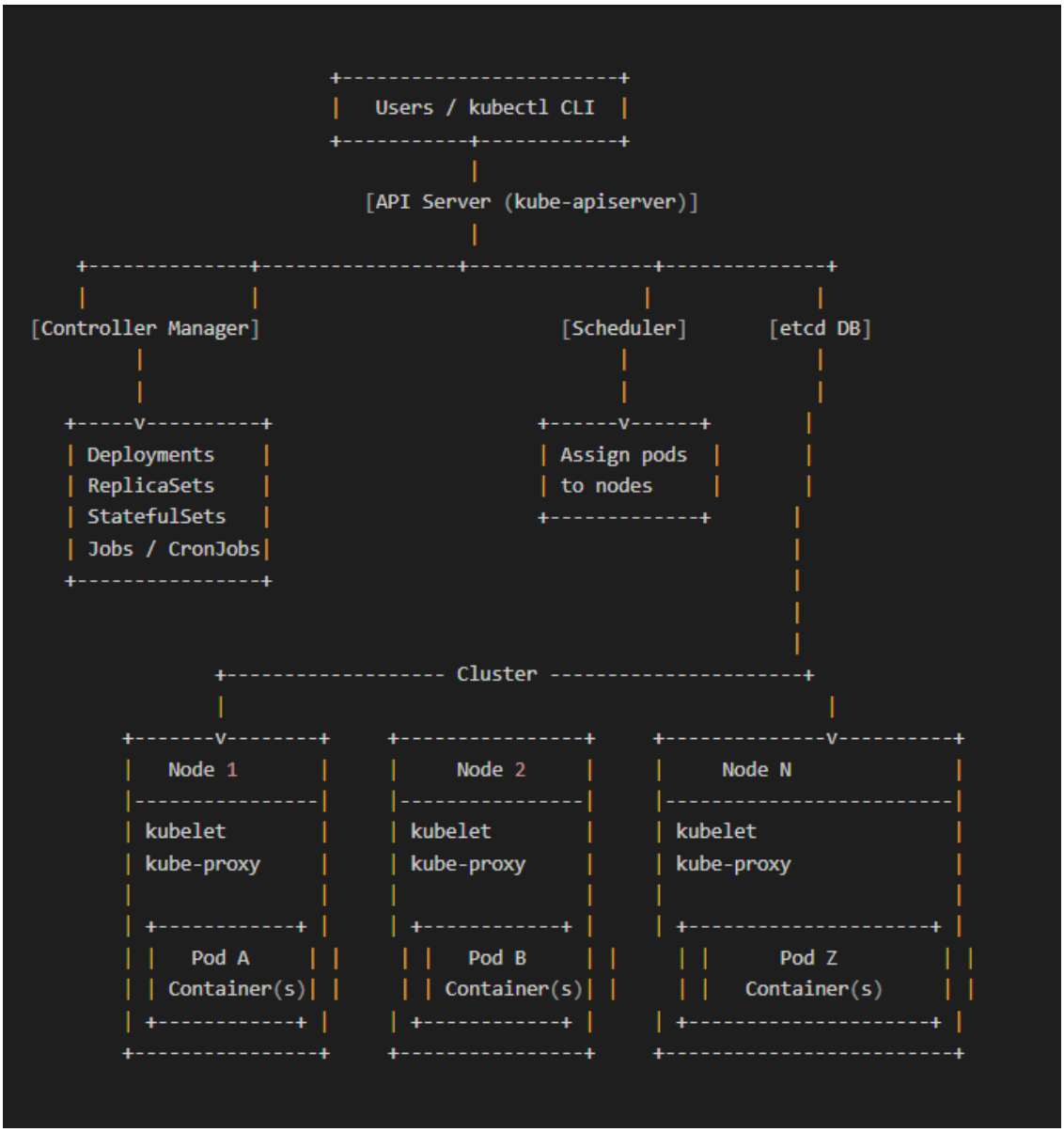
Visual

```
App Level:
  └── Pod → Container(s)
    └── Managed by → Deployment / ReplicaSet / StatefulSet / Job
      └── Exposed by → Service → (optionally Ingress)

Configuration:
  └── ConfigMap, Secret

Storage:
  └── Volume, PVC, PV

Infrastructure:
  └── Node → Kubelet, kube-proxy
    └── Controlled by → API Server, Scheduler, etcd
```



Kubernetes Namespace

- **What is it?**

  A way to divide cluster resources among users or teams. It provides a scope for resource names and helps organize them.

- **Why use it?**
  - Avoid naming conflicts (e.g., two apps named `backend` in different namespaces)
  - Isolate environments (dev, staging, prod)
  - Manage access control and resource limits per team/project

- **Default Namespaces:**
  - `default` – for user-created resources
  - `kube-system` – for Kubernetes system components
  - `kube-public` – publicly accessible data
  - `kube-node-lease` – node heartbeat info
- **Common Commands:**

```
kubectl create namespace <name>    # Create a namespace
kubectl get namespaces             # List all namespaces
kubectl get pods -n <namespace>    # View pods in a namespace
kubectl config set-context --current --namespace=<name>  # Set default namespace
```

- **Not All Resources Belong to a Namespace:**

  Cluster-level resources like `Nodes`, `PersistentVolumes`, and `StorageClasses` are not namespaces.

Kubernetes Labels

What are Labels?

Labels are **key-value pairs** attached to Kubernetes objects (like Pods, Services, Deployments) to organize and select subsets of objects.

They help you **group resources** by environment, version, role, or any custom criteria.

Label Syntax Example:

```
metadata:
  labels:
    app: nginx
    env: production
    tier: frontend
```

Key Features of Labels

- **Selector-based filtering:** Use label selectors to target specific resources.
- **Organize resources:** Group by environment (`env=dev`), app (`app=myapp`), etc.
- **Decouple configuration:** Independent of object structure; easy to add/update.

Common `kubectl` Commands with Labels

1. **Apply a label to a resource:**

```
kubectl label pods my-pod app=nginx
```

2. **List resources with labels:**

```
kubectl get pods --show-labels
```

3. **Filter using label selector:**

```
kubectl get pods -l app=nginx
```

Or multiple labels:

```
kubectl get pods -l 'app=nginx,env=production'
```

4. **Override existing label (force):**

```
kubectl label pods my-pod app=redis --overwrite
```

5. **Remove a label:**

```
kubectl label pods my-pod app-
```

Label Selectors

Used in Deployments, Services, etc., to match pods:

Example in a Service:

```
spec:
  selector:
    app: nginx
    tier: frontend
```

This service will route traffic to all pods labeled `app=nginx` and `tier=frontend`.

Best Practices

- Use meaningful keys like `app`, `env`, `version`.
- Keep labels consistent across environments.
- Don’t store sensitive or large data in labels.

Replication Controller

A **ReplicationController** in Kubernetes is a legacy object used to ensure that a specified number of **pod replicas** are running at any given time. It was one of the earliest constructs for managing replication in Kubernetes, but has largely been replaced by **Deployments**, which offer more advanced features.

It can't replicate if there is only 1 pod that failed

Purpose of ReplicationController

- **Ensures availability**: Always maintains the desired number of pod replicas.
- **Self-healing**: If a pod crashes or is deleted, it creates a new one to replace it.
- **Scaling & Load Balancing**: You can manually scale the number of replicas up or down.

Key Components

Here's a basic example of a ReplicationController YAML:

```
apiVersion: v1
kind: ReplicationController
metadata: # for the replication controller
  name: nginx-controller
spec: #for controller
  replicas: 3
  selector: # not required
    app: nginx
  template: #defining the pod
    metadata: #for the pod
      name: nginx-myapp
      labels:
        app: nginx
    spec: # for pod
      containers:
      - name: nginx
        image: nginx:latest
        ports:
        - containerPort: 80

# to see the replication, you run the command
# kubectl get replicationcontroller
```

How It Works

1. You define a `replicas` value (e.g., 3).

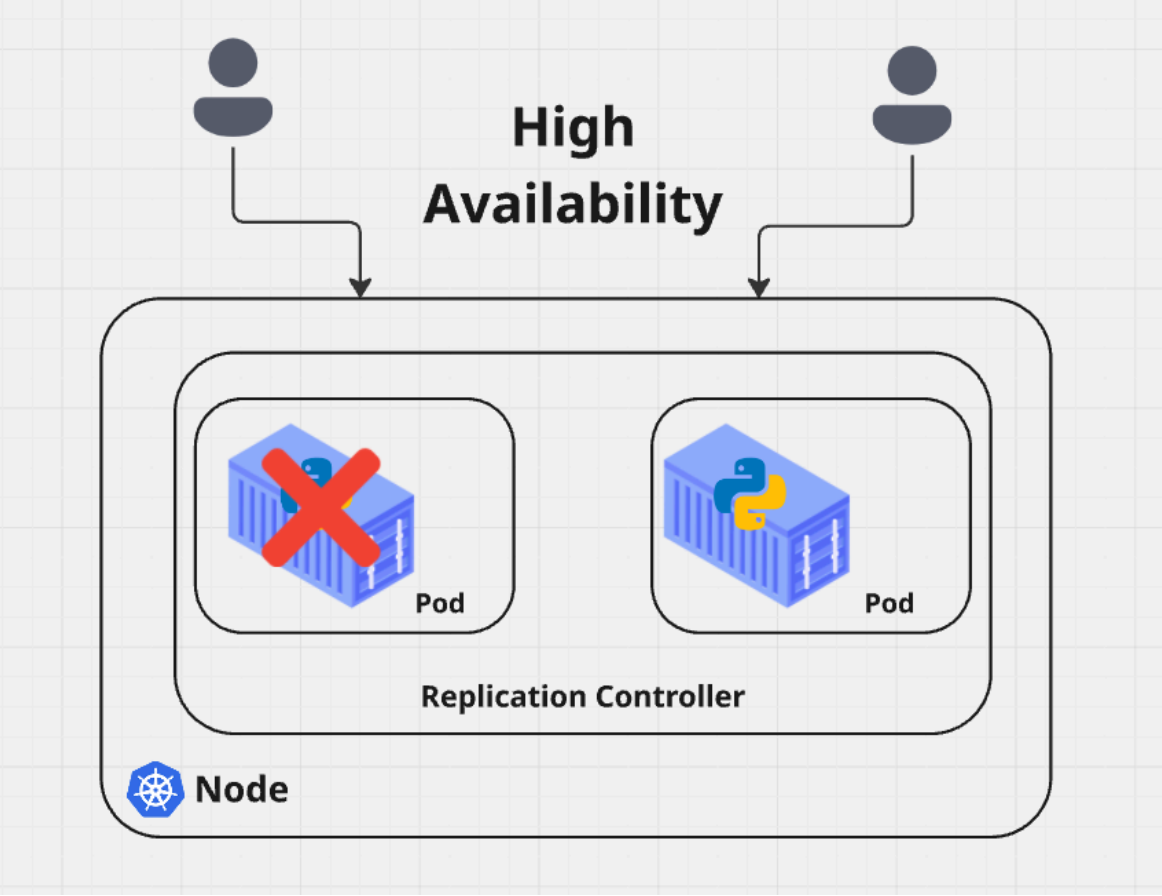
2. Kubernetes ensures there are always 3 pods matching the specified `selector`.
3. If a pod fails, it creates another to maintain the count.
4. You can **scale** by editing the `replicas` count.

Limitations (vs. Deployment)

- No **rolling updates or rollbacks**.
- Less control over versioning and lifecycle.
- Limited adoption in modern Kubernetes environments.

Use Cases

- Still useful for **basic** replication needs.
- Good for learning Kubernetes concepts.
- But in production, use a **Deployment** instead.



ReplicaSet in Kubernetes

A **ReplicaSet** is a Kubernetes object that ensures a specified number of **identical Pods** are running at all times. It is the **successor to ReplicationController** and adds more powerful **label selector** capabilities.

Key Differences from ReplicationController

Feature	ReplicationController	ReplicaSet
Label Selector Matching	Basic equality	Set-based + equality
Modern Usage	Legacy	Used by Deployments
Rolling Updates	False	True (via Deployment)

> ⚠️ You rarely create a ReplicaSet directly. Instead, you use a Deployment, which manages ReplicaSets under the hood to provide rolling updates, rollbacks, and version control.

Example ReplicaSet YAML

```
apiVersion: apps/v1 # different from RC it has apps/v1 not v1
kind: ReplicaSet
metadata:
  name: nginx-replicaset
spec:
  replicas: 3
  selector: #required
    matchLabels:
      type: front-end
  template: # spec for pod
    metadata:
      labels:
        app: nginx
        type: front-end
    spec:
      containers:
      - name: nginx
        image: nginx:latest
        ports:
        - containerPort: 80
```

How It Works

1. `replicas`: The desired number of Pod copies.

2. `selector`: Matches the Pods that the ReplicaSet will manage.
3. `template`: The Pod template used to create new Pods.

If a Pod fails or is deleted, the ReplicaSet automatically creates a new one.

When to Use

- **Direct use**: Rare; for advanced scenarios or learning.
- **Common use**: Managed by a **Deployment** object, which is the best practice in production.

Related Concepts

- **Deployment:** manages ReplicaSets and enables updates/rollbacks.
- **ReplicaSet:** Maintains Pod replicas.
- **Pod:** Smallest deployable unit.

The Selector is a major difference between ReplicationController and ReplicaSet

Labels and Selectors in Kubernetes

Labels

Labels are **key-value pairs** attached to Kubernetes objects like Pods, Nodes, Services, etc.

Purpose:

- Organize and group objects
- Enable selection/filtering for operations like deployments, scaling, and monitoring

Example:

```
labels:
  app: nginx
  env: production
```

Selectors

Selectors are used to **filter and match resources** based on their labels.

What type of operations can we use with Selectors to group objects with a specific Label?

Equality-based, Set-based

Two types:

1. **Equality-based** (used by ReplicationController):

```
selector:
  app: nginx
```

→ Matches objects where `app = nginx`

2. **Set-based** (used by ReplicaSet and beyond):

```
selector:
  matchLabels:
    app: nginx
  matchExpressions:
  - key: env
    operator: In
    values:
    - production
    - staging
```

Use in Practice

- **ReplicaSets** use selectors to find Pods to manage.
- **Services** use selectors to send traffic to the right Pods.
- **Deployments** match Pods via selectors and update them during rollouts.

Summary:

- Labels = metadata
- Selectors = filters using that metadata

When scaling, we can update the replica number in the YAML file, then replace it using this command

```
kubectl replace -f filenamereplica.yml
```

Or, we can change it in this command

```
kubectl scale --replicas=6 -f filenamereplica.yml
```

Commands Summary

```
kubectl create -f replicafile.yml
```

```
kubectl get replicaset

kubectl delete replicaset replicafile

kubectl replace -f replicafile.yml

kubectl scale --replicas=6 -f replicafile.yml

kubectl edit replicaset replicafile

kubectl scale replicaset replicafile --replicas=s
```

Deployments in Kubernetes

What is a Deployment?

A **Deployment** is a Kubernetes object used to manage a group of identical pods using a ReplicaSet. It provides declarative updates, making it easier to roll out, update, scale, and roll back applications.

The DeploymentController is a part of the control plane node's controller manager

Key Features

- **Creates and manages ReplicaSets**
- Supports **rolling updates** to ensure zero downtime
- **Rollbacks** to previous versions if something goes wrong
- Easy **scaling** of the number of pods
- Ensures **self-healing** by replacing failed or deleted pods

Basic Deployment YAML

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: nginx-deployment
spec:
  replicas: 3
  selector:
    matchLabels:
      app: nginx
```

```
  template:
    metadata:
      labels:
        app: nginx
    spec:
      containers:
      - name: nginx
        image: nginx:1.25
        ports:
        - containerPort: 80
```

Common Commands

Create or update a Deployment

```
kubectl apply -f deployment.yaml
```

View Deployments

```
kubectl get deployments
```

Describe a Deployment

```
kubectl describe deployment nginx-deployment
```

Scale a Deployment

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

Roll back to the previous revision

```
kubectl rollout undo deployment nginx-deployment
```

Get all objects

```
kubectl get all
```

Check rollout status

```
kubectl rollout status deployment nginx-deployment
```

Rolling Update in Kubernetes

A **rolling update** is a deployment strategy that **updates pods incrementally** to the new version **without downtime**. Kubernetes replaces old pods with new ones **one at a time**, ensuring that some version of the app is always running.

It is the default setting

There is another strategy called recreate - this causes application downtime

---

How it works

When you update a **Deployment** (for example, change the container image), Kubernetes:

1. Creates a new ReplicaSet with the updated spec.
2. Gradually increases the number of pods in the new ReplicaSet.
3. Simultaneously scales down the old ReplicaSet.
4. Maintains the desired number of total replicas throughout.

Example: Update a Deployment

Assume this Deployment is already applied:

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

To trigger a rolling update, change the image version (e.g., from `v1` to `v2`) and reapply:

```
kubectl set image deployment/myapp-deployment myapp=myapp:v2
```

Kubernetes will:

- Start 1 new pod with `v2`

- Terminate 1 old pod with `v1`
- Repeat until all pods are updated

Control Rolling Update Speed

You can configure these fields in the deployment spec:

```
strategy:
  type: RollingUpdate
  rollingUpdate:
    maxUnavailable: 1
    maxSurge: 1
```

- `maxUnavailable`: Maximum number of pods that can be **unavailable** during the update.
- `maxSurge`: Maximum number of **extra** pods that can be created temporarily during the update.

Check Status

```
kubectl rollout status deployment myapp-deployment
```

Rollback if Something Goes Wrong

```
kubectl rollout undo deployment myapp-deployment
```

Kubernetes `rollout` Command

The `kubectl rollout` command is used to **manage the rollout status and history** of Deployments (and other resources like DaemonSets and StatefulSets).

Common `kubectl rollout` Commands

1. Check the rollout status

Shows progress of a rolling update:

```
kubectl rollout status deployment <deployment-name>
```

Example:

```
kubectl rollout status deployment myapp-deployment
```

---

2. Undo a rollout (rollback)

Reverts the Deployment to the previous working version:

```
kubectl rollout undo deployment <deployment-name>
```

Example:

```
kubectl rollout undo deployment myapp-deployment
```

3. Undo to a specific revision

```
kubectl rollout undo deployment <deployment-name> --to-revision=2
```

4. View rollout history

Shows all previous revisions:

```
kubectl rollout history deployment <deployment-name>
```

5. Pause a rollout

Prevents Kubernetes from continuing with an update (useful for manual steps):

```
kubectl rollout pause deployment <deployment-name>
```

6. Resume a paused rollout

```
kubectl rollout resume deployment <deployment-name>
```

To **add rollback** support to your Kubernetes Deployment, you don't need to add anything special to the YAML — **rollback is built-in** for Deployments. However, here's how to **enable, trigger, and manage rollbacks** effectively.

1. Default Deployment YAML (Rollback-ready)

Kubernetes automatically stores **revision history** for Deployments by default.

Example:

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: myapp-deployment
```

```
spec:
  replicas: 3
  selector:
    matchLabels:
      app: myapp
  template:
    metadata:
      labels:
        app: myapp
    spec:
      containers:
      - name: myapp
        image: myapp:v1
```

---

2. Trigger a New Rollout

Change the image:

```
kubectl set image deployment/myapp-deployment myapp=myapp:v2
```

Check rollout:

```
kubectl rollout status deployment myapp-deployment
```

---

3. Roll Back to Previous Version

Rollback:

```
kubectl rollout undo deployment myapp-deployment
```

Rollback to specific revision:

```
kubectl rollout undo deployment myapp-deployment --to-revision=2
```

View history:

```
kubectl rollout history deployment myapp-deployment
```

---

4. Optional: Limit Revision History (cleanup)

To limit how many old versions Kubernetes keeps (to save resources):

```
spec:
  revisionHistoryLimit: 3
```

Add that under the `spec` of your Deployment to retain only the last 3 versions.

Summary

What Are Rollout Commands in Kubernetes?

Rollout commands help you **manage and monitor the updates** (aka rollouts) of your **Kubernetes Deployments** (or StatefulSets, DaemonSets).

When you update a deployment (like changing the image version), Kubernetes **rolls out** the new version **gradually** to ensure stability.

Common Rollout Commands

1. `kubectl rollout status`

- **Checks** the progress of a rollout.
- Example:

```
kubectl rollout status deployment/my-app
```

2. `kubectl rollout history`

- **Shows** the revision history of the deployment.
- Example:

```
kubectl rollout history deployment/my-app
```

3. `kubectl rollout undo`

- **Rolls back** to a previous revision if something goes wrong.
- Example:

```
kubectl rollout undo deployment/my-app
```

- Rollback to a specific revision:

```
kubectl rollout undo deployment/my-app --to-revision=2
```

4. `kubectl set image`

- **Triggers a rollout** by changing the image.
- Example:

```
kubectl set image deployment/my-app my-container=my-image:2.0
```

Rollout Workflow Roadmap

1. Start with a Deployment

You create a deployment (e.g., `my-app`) that defines replicas, container images, etc.

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: my-app
spec:
  replicas: 3
  template:
    spec:
      containers:
      - name: my-container
        image: my-image:1.0
```

---

2. Update the Deployment

You update the image (or another property), either by editing the YAML or using:

```
kubectl set image deployment/my-app my-container=my-image:2.
```

This triggers a **new rollout**.

---

3. Monitor the Rollout

Check the rollout status to monitor if the new pods are ready:

```
kubectl rollout status deployment/my-app
```

---

4. Check History

See the history of deployments:

```
kubectl rollout history deployment/my-app
```

5. Undo if Needed

If there's an issue, roll back:

```
kubectl rollout undo deployment/my-app
```

Summary Table

Command	Purpose
`kubectl set image`	Start a new rollout
`kubectl rollout status`	Monitor progress
`kubectl rollout history`	See version history
`kubectl rollout undo`	Roll back if needed

Kubernetes Networking

An IP Address is assigned to a pod, not like Docker, it is assigned to a container

Key Concepts

1. Pod-to-Pod Communication

- **Every pod gets its IP address.**
- Pods can communicate with each other **across nodes** without NAT (Network Address Translation).
- Kubernetes assumes a **flat network space,** all pods can talk to all other pods directly.

2. Container-to-Container Communication (Within a Pod)

- Containers in the same pod share the **same network namespace**, IP, and port space.
- They communicate via `localhost`.

3. Service Networking

- A **Service** is a stable abstraction over a set of pods.
- Kubernetes assigns a **virtual IP (ClusterIP)** to each service.
- Types of services:

- `ClusterIP` – Default. Only accessible inside the cluster.
- `NodePort` – Exposes the service on a static port on each node.
- `LoadBalancer` – Provisions an external IP via a cloud provider.
- `ExternalName` – Maps a service to an external DNS name.

---

4. DNS in Kubernetes

- Kubernetes runs a DNS pod (`CoreDNS`) that automatically creates DNS records for services and pods.
- So, services can talk to each other using **DNS names**, like:

```
http://my-service.my-namespace.svc.cluster.local
```

---

5. Network Policies

- Used to control **traffic flow** at the IP address or port level.
- Define rules for:
  - **Ingress** (incoming) traffic to pods
  - **Egress** (outgoing) traffic from pods
- These are enforced by **network plugins** (e.g., Calico, Cilium).

---

6. CNI Plugins (Container Network Interface)

- Kubernetes uses the CNI standard to implement network functionality.
- Popular CNI plugins:
  - **Calico** – L3 routing, network policies
  - **Flannel** – Simple overlay network
  - **Weave** – Encrypted mesh networking
  - **Cilium** – eBPF-powered, secure, and observable

Summary

Component	Purpose
Pod IP	Unique per pod; direct pod-to-pod communication
Services	Stable IP/DNS to expose and balance across pods
DNS	Name resolution for services and pods
Network Policy	Secure/limit traffic between pods

CNI Plugin	Implements the network stack

DaemonSets in Kubernetes

A **DaemonSet** ensures that a specific pod runs **on every node** (or a selected set of nodes) in a Kubernetes cluster. This is useful for deploying background system-level services that need to be present on each node.

What DaemonSets Are Used For

Common use cases for DaemonSets include:

- Log collection agents (e.g., Fluentd, Logstash)
- Monitoring agents (e.g., Prometheus Node Exporter)
- Security agents (e.g., Falco, antivirus software)
- Storage daemons (e.g., GlusterFS, Ceph)

How DaemonSets Work

When you create a DaemonSet:

- The Kubernetes controller ensures **one pod is scheduled on every node**.
- As new nodes are added to the cluster, the DaemonSet controller automatically **adds the pod to those nodes**.
- If a node is removed, the pod on that node is also removed.

You can also use **node selectors, labels, taints, or tolerations** to restrict a DaemonSet to only run on specific nodes.

Example DaemonSet YAML

```
apiVersion: apps/v1
kind: DaemonSet
metadata:
  name: node-exporter
  namespace: monitoring
spec:
  selector:
    matchLabels:
      app: node-exporter
  template:
    metadata:
      labels:
```

```
      app: node-exporter
  spec:
    containers:
    - name: node-exporter
      image: prom/node-exporter:v1.0.1
      ports:
      - containerPort: 9100
```

Key Properties

Field	Description
`spec.selector`	Defines how the DaemonSet matches pods
`spec.template`	Describes the pod to run on each node
`spec.updateStrategy`	Can control how rolling updates are done for DaemonSets

Managing DaemonSets

Useful `kubectl` commands:

- View all DaemonSets:

  ```
  kubectl get daemonsets -A
  ```

- Describe a specific DaemonSet:

  ```
  kubectl describe daemonset <name> -n <namespace>
  ```

- Delete a DaemonSet:

  ```
  kubectl delete daemonset <name>
  ```

Difference Between DaemonSet and Deployment

Feature	DaemonSet	Deployment
Scheduling	One pod per node	Any number of pods on any node
Use Case	Node-specific tasks	Application workloads
Node Addition	Automatically adds new pods	Does not auto-adjust

Here's a clear breakdown of **Authentication, Authorization, and Admission Control** in Kubernetes, often referred to as the **3 As of API security**:

1. Authentication (Who are you?)

- **Purpose:** Verifies the **identity** of the user or service requesting the Kubernetes API server.
- **Examples of authentication methods:**
  - **Certificates** (client TLS certs)
  - **Bearer tokens** (including service account tokens)
  - **OIDC** (OAuth2/OpenID Connect providers like Google, Azure AD)
  - **Webhook tokens** (custom external auth systems)
- **Result:** If valid, Kubernetes associates the request with a `User` or `ServiceAccount`.
  - Normal users: they are managed outside the Kubernetes cluster via standalone services
  - Service accounts: Enable communication for the in-cluster process through the API server to perform various operations.

---

2. Authorization (What are you allowed to do?)

- **Purpose:** Determines **whether the authenticated identity is allowed** to perform a specific action on a resource.
- **Mechanisms supported:**
  - **RBAC** (Role-Based Access Control) – the most commonly used
  - **ABAC** (Attribute-Based Access Control) – legacy
  - **Webhook authorization** – for custom logic
- **Example:**

  "Is `user1` allowed to delete pods in the `dev` namespace?"

---

3. Admission Control (Should this request be allowed in the current context?)

- **Purpose:** Acts on a request **after authentication and authorization**, but **before it is persisted** to etcd.
- specify granular access control policies
- **Admission controllers** can:
  - Deny requests that don't meet certain criteria
  - Mutate or validate the request
  - Enforce policies (e.g., no `latest` tag in container images)
- **Examples of admission controllers:**

- `NamespaceLifecycle`
- `LimitRanger`
- `PodSecurity`
- `ValidatingAdmissionWebhook` / `MutatingAdmissionWebhook`

Request Flow Summary:

```
Client Request
  ↓
Authentication → Authorization → Admission Control → API Server executes request (if approved)
```

Summary Table:

Stage	Function	Example
Authentication	Verifies identity	Token, certificate, OIDC
Authorization	Checks permissions	RBAC: Can `alice` read pods?
Admission Control	Enforces rules and policies	Deny pods with `latest` image tag

Services in Kubernetes

In Kubernetes, **Services** are a fundamental abstraction used to enable communication between different components within a cluster and to expose applications to the outside world when needed.

It logically groups the pods via Labels and Selectors

Labels and Selectors use a key-value pair format

Summary: Services enable communication within and outside of the cluster

Why Services Are Important

- **Pod IPs are not permanent**: Since Pods can be recreated at any time, their IP addresses may change, making direct communication unreliable.
- **Consistent network identity**: Services provide a stable IP and DNS name to access a group of Pods.
- **Load balancing**: Services distribute traffic across the targeted Pods, ensuring high availability and reliability.

Types of Services

1. **ClusterIP** (default)
    - **Purpose**: Exposes the Service on an internal IP within the cluster.
    - **Accessibility**: Only reachable from within the cluster.
    - **Use case**: Communication between internal components, such as frontend Pods calling backend APIs.
2. **NodePort**
    - **Purpose**: Exposes the Service on a static port on each Node's IP.
    - **Accessibility**: Reachable externally via `<NodeIP>:<NodePort>`.
    - **Use case**: Useful for simple external access in development or small-scale setups.
    - Notes:
        - External access to the node
3. **LoadBalancer**
    - **Purpose**: Provision an external load balancer (usually provided by cloud platforms) to expose the Service.
    - **Accessibility**: External clients can access the service via a public IP.
    - **Use case**: Common in production for exposing web apps or APIs.
4. **ExternalName**
    - **Purpose**: Maps the Service to a DNS name defined outside the cluster.
    - **Accessibility**: Does not create a traditional Service proxy but allows internal access to external services.
    - **Use case**: Useful for connecting to managed databases or APIs.

Key Concepts

- **Selectors**: Most Services use labels to select the Pods they route traffic to.
- **Endpoints**: Kubernetes automatically maintains a list of endpoints (Pod IPs) behind a Service.
- **DNS**: Kubernetes automatically creates a DNS entry for each Service, making it easy for other components to discover and connect to it using names like `service-name.namespace.svc.cluster.local`.

---

1. Example: `ClusterIP` Service

```
apiVersion: v1
kind: Service
metadata:
  name: my-web-app-service
  labels:
```

```
    app: my-web-app
spec:
  selector:
    app: my-web-app
  ports:
    - protocol: TCP
      port: 80        # Port exposed by the service
      targetPort: 8080 # Port the Pods are actually listening on
```

In this example:

- Pods are labeled with `app: my-web-app`.
- The Service listens on port `80` and forwards traffic to port `8080` on the selected Pods.

---

2. Example: `NodePort` Service

```
apiVersion: v1
kind: Service
metadata:
  name: my-web-app-nodeport
spec:
  type: NodePort
  selector:
    app: my-web-app
  ports:
    - port: 80
      targetPort: 8080
      nodePort: 30080  # Exposes the service on each Node at this port
```

This allows access from outside the cluster using `http://<NodeIP>:30080`.

---

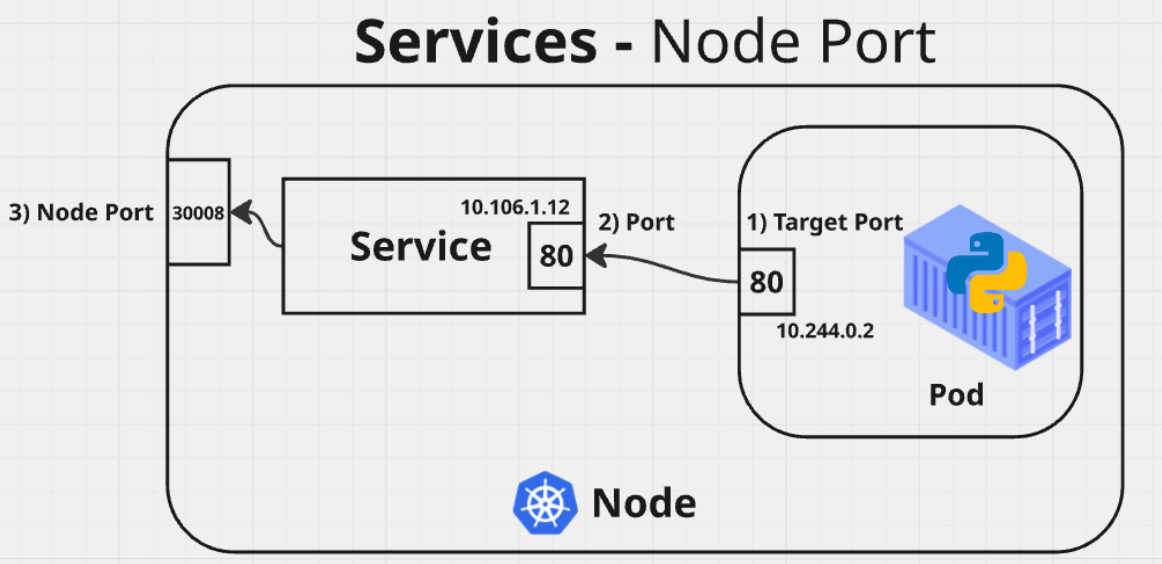
3. Example: `LoadBalancer` Service (Cloud Providers)

```
apiVersion: v1
kind: Service
metadata:
  name: my-web-app-loadbalancer
spec:
  type: LoadBalancer
  selector:
    app: my-web-app
  ports:
```

```
  - port: 80
    targetPort: 8080
```

This will request that a cloud provider provision a load balancer and assign a public IP address.

Services - Node Port



Kubernetes Object Hierarchy

1. Pod

- The smallest deployable unit in Kubernetes.
- You can define a Pod directly, but it's rarely used in production because Pods aren't self-healing (if a Pod dies, it won't be recreated automatically).

2. Deployment

- A higher-level object that **manages Pods** for you.
- Automatically creates a **ReplicaSet** behind the scenes to ensure the desired number of Pods are running.
- Supports rolling updates and rollbacks.
- **You typically define a Deployment, not a Pod, directly.**

3. ReplicaSet

- Ensures that a specified number of Pod replicas are running at all times.

- Usually not created manually — handled by Deployments.
- You don't need to write YAML for this unless you're doing something very specific.

4. Service

- A stable network endpoint that allows access to one or more Pods.
- You create this separately, but **you link it to your Pods using label selectors** (e.g., `app: my-app`).

Typical Workflow

1. **Create a Deployment** (defines how many Pods, what container image, etc.)
2. **Create a Service** (exposes the Pods managed by the Deployment)
3. (Optional) Use Ingress for more advanced routing (e.g., path-based or domain-based routing)

Example Overview

1. **Deployment YAML** (creates Pods with label `app: my-app`)
2. **Service YAML** (selects Pods with label `app: my-app` to expose them)

You don't *need* to create Pods, ReplicaSets, and Deployments all separately — define the Deployment and let it handle the rest. Then add a Service on top to expose it.

Here's a full example of a **Deployment + Service** in Kubernetes using YAML. This setup deploys a basic web app (e.g., Nginx) and exposes it internally using a `ClusterIP` service.

1. Deployment + Service (YAML Example)

```
# deployment.yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: my-nginx-deployment
spec:
  replicas: 3
  selector:
    matchLabels:
      app: my-nginx
  template:
    metadata:
      labels:
        app: my-nginx
    spec:
      containers:
        - name: nginx
```

```
        image: nginx:latest
        ports:
          - containerPort: 80
---
# service.yaml
apiVersion: v1
kind: Service
metadata:
  name: my-nginx-service
spec:
  selector:
    app: my-nginx
  ports:
    - port: 80
      targetPort: 80
  type: ClusterIP
```

What This Does

- **Deployment**:
  - Launches 3 replicas of `nginx:latest`.
  - Labels the Pods with `app: my-nginx`.
- **Service**:
  - Selects Pods with the label `app: my-nginx`.
  - Routes traffic to port 80 inside the Pods.
  - Exposes a stable internal endpoint (ClusterIP) accessible within the cluster.

How to Apply

```
kubectl apply -f deployment.yaml
kubectl apply -f service.yaml
```

Then check:

```
kubectl get pods
kubectl get svc #svc = service
```

To access a Kubernetes Service (like the Nginx example) running on **k3s on an EC2 instance**, you'll need to expose the service to the outside world and make sure the EC2 instance allows incoming traffic on the required port.

1. Use NodePort or LoadBalancer

If your current Service is type `ClusterIP`, it's only accessible *inside* the cluster. You'll need to change the Service type to `NodePort` (or `LoadBalancer` if using a cloud LB).

Update your service like this:

```
apiVersion: v1
kind: Service
metadata:
  name: my-nginx-service
spec:
  selector:
    app: my-nginx
  ports:
    - port: 80
      targetPort: 80
      nodePort: 30080  # Exposes the service on this port
  type: NodePort
```

Apply it:

```
kubectl apply -f service.yaml
```

---

2. Get the EC2 Instance's Public IP

In your terminal:

```
curl http://checkip.amazonaws.com
```

Or from the AWS EC2 console, find the **public IPv4 address** of your EC2 instance.

---

3. Allow Inbound Traffic in EC2 Security Group

Make sure your EC2 instance's **security group** has an inbound rule that allows TCP traffic to port `30080`.

- Go to the **EC2 Dashboard > Instances > [your instance]**
- Click on the **Security group** linked to your instance
- Add a rule:
  - **Type**: Custom TCP
  - **Port range**: 30080
  - **Source**: 0.0.0.0/0 (or your IP for restricted access)

---

4. Access the Application in Browser or Curl

Now you can access your service:

```
http://<your-ec2-public-ip>:30080
```

You should see the Nginx welcome page.

---

kube-proxy

In Kubernetes, `kube-proxy` is a network component that runs on every node in the cluster. Its primary job is to enable communication between services and the pods they expose. It listens to the Kubernetes API for updates to `Service` and `Endpoint` objects, and sets up the necessary networking rules to forward traffic to the appropriate backend pods.

`kube-proxy` abstracts the complexity of pod networking by handling the routing of traffic based on service definitions. It can operate in different modes, like `iptables` or `ipvs`, to efficiently manage and load-balance traffic across pods.

Traffic Policies in Kubernetes Services

Kubernetes Services support **traffic policies** that influence how traffic is routed to backend Pods. These policies are especially useful when you need to control load balancing behavior across a distributed cluster.

1. ExternalTrafficPolicy

This setting applies to `NodePort` and `LoadBalancer` Services and controls how **external traffic** (coming from outside the cluster) is handled.

Options:

- `Cluster` (default)
  - Traffic is routed to any healthy pod in the cluster, regardless of which node the traffic arrived on.
  - **Pros**: Better load distribution.
  - **Cons**: The client source IP is not preserved (it's replaced by the node IP).
- `Local`
  - Only routes traffic to pods running on the same node the request was received on.
  - **Pros**: Preserves the original client source IP.
  - **Cons**: If there are no local pods on a node, traffic is dropped.

Example:

```
spec:
  type: LoadBalancer
```

```
  externalTrafficPolicy: Local
```

2. InternalTrafficPolicy (introduced in Kubernetes v1.26)

This setting controls how traffic from **within the cluster** is distributed across pods behind a `ClusterIP` or `LoadBalancer` service.

Options:

- `Cluster` (default)
  - Internal traffic can be sent to any pod in the cluster.
- `Local`
  - Internal traffic is only sent to pods on the same node.
  - Useful for reducing cross-node traffic or keeping latency low.

Example:

```
spec:
  internalTrafficPolicy: Local
```

Summary

- `kube-proxy` ensures service traffic is properly routed within a Kubernetes cluster.
- `externalTrafficPolicy` and `internalTrafficPolicy` provide fine-tuned control over how traffic flows, which can be important for performance, latency, and preserving source IPs.
- External traffic policy can be handled by Load Balancer and Node Port
- Internal traffic policy can be handled by Load Balancer and Cluster IP

Service Discovery

- Environment Variables
  - *kubelet* daemon on that node adds a set of environment variables in the pod for all active services
- DNS
  - add-on in Kubernetes
  - it creates a DNS Record for each service

Multi-Port Services in Kubernetes

A **Multi-Port Service** in Kubernetes is a Service that exposes **more than one port** on the same group of Pods. This is useful when your application exposes multiple ports for different purposes, for

example, a web server exposing HTTP on port 80 and HTTPS on port 443, or a database exposing a data port and a management port.

When to Use a Multi-Port Service

- An app with **multiple containers**, each exposing different ports.
- A single container that listens on **multiple ports**, each for a different function (e.g., API, metrics, admin UI).
- You want to expose and route to multiple ports from a **single service** definition instead of creating separate Services.

YAML Example: Multi-Port Service

Here's an example where a Deployment runs an application that exposes port 80 for HTTP and 9090 for metrics:

```
apiVersion: v1
kind: Service
metadata:
  name: multi-port-service
spec:
  selector:
    app: my-app
  ports:
    - name: http
      protocol: TCP
      port: 80
      targetPort: 8080
    - name: metrics
      protocol: TCP
      port: 9090
      targetPort: 9090
  type: ClusterIP
```

How It Works

- The `selector` targets Pods with the label `app: my-app`.
- The service exposes:
  - Port **80**, which routes to container port **8080** (e.g., main web traffic).
  - Port **9090**, which routes to container port **9090** (e.g., Prometheus metrics).
- Clients in the cluster can connect to either port through the same Service.

You can access the ports inside the cluster like:

```
http://multi-port-service.default.svc.cluster.local:80
http://multi-port-service.default.svc.cluster.local:9090
```

Things to Remember

- Always name your ports when defining multiple — it's **required** when using features like Ingress or advanced routing.
- Both `port` and `targetPort` can be the same or different based on your container configuration.
- You can define services with multiple protocols too (e.g., TCP and UDP), but this is rare and must be done carefully.

Liveness in Kubernetes

Liveness probes in Kubernetes are used to check whether a container is **still running properly**. If a container becomes **unresponsive or stuck**, even if it's technically still running, Kubernetes will automatically **restart** it based on the liveness probe's failure results.

Why Use a Liveness Probe?

A container might:

- Be running but stuck in an infinite loop or a deadlock
- Have failed internally while the process still exists
- Become unresponsive due to memory issues or exceptions

Liveness probes help Kubernetes **detect and recover** from these situations without human intervention.

Types of Liveness Probes

Kubernetes supports three types of liveness probes:

Type	Description
HTTP GET	Kubernetes sends an HTTP GET request to a specific path. If the response code is between 200 and 399, the container is considered alive.
Exec Command	Runs a command inside the container. If the command exits with code `0`, the container is considered alive.
TCP Socket	Tries to establish a TCP connection to the specified port. If the connection is successful, the container is alive.

Example: HTTP Liveness Probe

```
livenessProbe:
  httpGet:
    path: /healthz
    port: 8080
  initialDelaySeconds: 10
  periodSeconds: 5
  failureThreshold: 3
```

Key Fields:

- `initialDelaySeconds`: How many seconds to wait before performing the first check
- `periodSeconds`: How often to perform the check
- `failureThreshold`: How many failures before the container is restarted

Liveness vs Readiness

- **Liveness probe**: Indicates if the container should be restarted
- **Readiness probe**: Indicates if the container is ready to receive traffic

---

Startup Probes

The latest addition to the Probes family is the Startup Probe. This probe is specifically designed for legacy applications that may require additional time to initialize fully. Its purpose is to delay the Liveness and Readiness probes, providing enough time for the application to complete its initialization process

---

Kubernetes in Cloud Environments

Why Kubernetes in the Cloud?

- **Scalability**: Automatically adjusts based on traffic.
- **High Availability**: Manages failures at the application and infrastructure levels.
- **Portability**: Consistent environment across clouds and on-prem.
- **Microservices-Friendly**: Built for distributed, container-based architectures.
- **DevOps Integration**: Works well with CI/CD pipelines, observability tools, and GitOps workflows.

Kubernetes on AWS, Amazon EKS (Elastic Kubernetes Service)

Key Features:

- **Fully managed** control plane.
- Integrated with **IAM**, **VPC**, **CloudWatch**, and **ALB/NLB**.
- Supports **Fargate** for serverless container execution.
- EBS and EFS support for persistent volumes.

Pros:

- Deep integration with the AWS ecosystem.
- Scalable and secure.
- Optional self-managed nodes or managed node groups.

Use Cases:

- Hybrid workloads using **Outposts**.
- Financial services leveraging compliance-ready environments.

Kubernetes on GCP, Google Kubernetes Engine (GKE)

Key Features:

- Google’s native Kubernetes service (Google created Kubernetes).
- Autopilot mode for **fully managed nodes and operations**.
- Strong built-in observability (Stackdriver, Cloud Logging/Monitoring).
- Native CI/CD integration with **Cloud Build** and **Artifact Registry**.

Pros:

- Best-in-class Kubernetes performance.
- Simplified node management with **GKE Autopilot**.
- Smart autoscaling and updates.

Use Cases:

- AI/ML workloads due to TensorFlow, GPUs, and TPU support.
- Global applications leveraging Google's network.

Kubernetes on Azure, Azure Kubernetes Service (AKS)

Key Features:

- Managed control plane with optional **Azure Container Instances (ACI)** for burstable workloads.

- Native integrations with **Azure DevOps**, **Azure Monitor**, and **Application Gateway Ingress Controller**.
- **Azure Arc** for hybrid/multi-cloud Kubernetes management.

Pros:

- Strong developer tooling (Visual Studio Code + Azure CLI).
- Easy setup and scaling.
- Integration with Microsoft identity services and networking.

Use Cases:

- Enterprises already using the Microsoft stack (AD, .NET, Windows containers).
- Hybrid deployments using **Azure Stack** and **Arc**.

Comparison

Feature	AWS EKS	GCP GKE	Azure AKS
Control Plane	Managed	Fully managed	Managed
Auto Scaling	Node & Pod	Node & Pod	Node & Pod
Best For	AWS workloads	Performance, automation	Microsoft ecosystem
Serverless	Fargate	Autopilot	ACI
Hybrid Cloud	Outposts	Anthos	Azure Arc

Real-World Use Cases

- **Multi-cloud Resilience**: Deploying the same Kubernetes cluster across AWS and GCP for HA.
- **Cost Optimization**: Using preemptible instances in GKE or Spot Instances in EKS.
- **Regulatory Compliance**: Using AKS in Azure Government for US FedRAMP environments.

Kubernetes Volume Management

In Kubernetes, **volume management** handles how containers access and store data that persists beyond their lifecycle. Since containers are **ephemeral** (they disappear when restarted or deleted), volumes provide a way to persist data, share files between containers, and integrate with storage systems.

---

1. What is a Volume in Kubernetes?

A **volume** in Kubernetes is a directory accessible to containers in a pod. Unlike container-local storage, a volume's lifecycle is tied to the **pod**, not the individual container.

2. Types of Volumes

a. Ephemeral Volumes (short-lived)

- **emptyDir**: Temporary storage; cleared when the pod is deleted.
- **configMap**, **secret**: Inject configuration or secrets as files.
- **downwardAPI**: Expose pod or container metadata.

b. Persistent Volumes (PVs) (long-lived)

- Designed to persist beyond the pod's lifecycle.
- Managed via the **PersistentVolume (PV)** and **PersistentVolumeClaim (PVC)** system.

3. Persistent Volume (PV) and Persistent Volume Claim (PVC)

PersistentVolume (PV)

- A **resource in the cluster** representing a piece of storage.
- Can be backed by local disk, NFS, cloud storage (EBS, Azure Disk, GCP PD), etc.

PersistentVolumeClaim (PVC)

- A **request for storage** by a pod.
- Pods don't use PVs directly—they use PVCs to claim the storage they need.
- Access Methods:
  - ReadWriteOnce
  - ReadOnlyMany
  - ReadWriteMany
  - ReadWriteOncePod

Example:

```
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
  name: my-pvc
spec:
  accessModes:
    - ReadWriteOnce
  resources:
```

```
    requests:
      storage: 5Gi
```

Then in a pod:

```
volumes:
  - name: my-volume
    persistentVolumeClaim:
      claimName: my-pvc
```

---

4. Access Modes

- **ReadWriteOnce (RWO)**: Mounted as read-write by a single node.
- **ReadOnlyMany (ROX)**: Mounted as read-only by many nodes.
- **ReadWriteMany (RWX)**: Mounted as read-write by many nodes.

---

5. Volume Plugins/Storage Classes

Kubernetes supports several volume backends:

- **Cloud storage**: AWS EBS, GCP Persistent Disk, Azure Disk
- **Network storage**: NFS, iSCSI, Ceph
- **Local storage**: HostPath, Local PV
- **CSI drivers**: Container Storage Interface (modern, portable storage plugin system)

You can define **StorageClasses** for dynamic provisioning:

```
apiVersion: storage.k8s.io/v1
kind: StorageClass
metadata:
  name: fast
provisioner: kubernetes.io/aws-ebs
parameters:
  type: gp2
```

---

6. Dynamic vs Static Provisioning

- **Static**: Admin manually creates PVs.
- **Dynamic**: PVC automatically creates a PV using a `StorageClass`.

---

7. Volume Lifecycle

- PV exists independently in the cluster.
- PVC is a user request.
- A controller binds them together.
- When the pod is deleted, the PVC may be retained or deleted based on the **reclaim policy**:
  - **Retain**
  - **Delete**
  - **Recycle** (deprecated)

---

8. Best Practices

- Use **StorageClasses** for flexibility and automation.
- Set the correct **access modes** according to the workload.
- Monitor volume usage and reclaim policies to avoid orphaned storage.
- Prefer **CSI drivers** for future-proof and cross-cloud compatibility.

Container Storage Interface (CSI)

The **Container Storage Interface (CSI)** is a standard that enables container orchestration systems like **Kubernetes** to expose arbitrary storage systems to containers in a consistent and portable way.

Before CSI, storage integration in Kubernetes was done through in-tree plugins, which were tightly coupled with the Kubernetes codebase and difficult to maintain or extend. CSI addresses this by providing a **vendor-neutral API** that allows storage providers to develop plugins independently of Kubernetes, improving flexibility, security, and maintainability.

Key Features of CSI:

- **Standardized API**: Defines a common interface between container orchestrators and storage systems.
- **Decoupled Architecture**: Storage providers can build and maintain their own CSI drivers without modifying the Kubernetes core code.
- **Portability**: Enables consistent storage operations across different environments—on-premises, cloud, or hybrid.
- **Extensible**: Supports a wide range of storage solutions, including block, file, and object storage.

How CSI Works in Kubernetes:

1. **CSI Driver**: A storage provider implements a CSI-compliant driver as a set of gRPC services running in the cluster.

2. **CSI Sidecar Containers**: Helper containers run alongside the driver to handle tasks like volume mounting, attaching, and health monitoring.
3. **Kubernetes Controller and Kubelet**: Communicate with the CSI driver via standard CSI RPC calls to manage storage lifecycle operations such as provisioning, attaching, and mounting volumes.

Common Use Cases:

- Dynamic provisioning of persistent volumes
- Mounting cloud provider storage (e.g., AWS EBS, Azure Disk, GCP PD)
- Integrating enterprise storage systems (e.g., NetApp, Dell EMC, Portworx)
- Supporting stateful applications like databases in Kubernetes

---

In summary, the **CSI standard simplifies and streamlines storage management in Kubernetes**, making it easier for developers and operators to consume storage from a variety of backend systems consistently and reliably.

ConfigMaps

Use Cases:

- Set environment variables for applications.
- Externalize app configuration files.
- Share configuration across multiple Pods.
- Provide command-line arguments to applications at runtime.

Mounting as a Volume:

You can mount a ConfigMap as a file inside a container.

```
volumeMounts:
  - name: config-volume
    mountPath: /etc/config
volumes:
  - name: config-volume
    configMap:
      name: my-config
```

Each key in the ConfigMap becomes a file under `/etc/config`.

Updating a ConfigMap:

If you update a ConfigMap, **running Pods don't automatically reload** the values. You must restart the Pod or implement logic to watch for changes (like a sidecar pattern or reloader).

Create a ConfigMap

```
kubectl create configmap my-config \
--from-literal=key1=value1 \
--from-literal=key2=value2

#get configmap
kubectl get configmaps my-config -o yaml
```

Definition Manifest file

```
apiVersion: v1
kind: ConfigMap
metadata:
  name: myfirstconfig
 data:
  text1: company1
  text2: hello
  company: example technology
```

Create using the following command

```
kubectl create -f customer-configmap.yaml
```

Use a ConfigMap inside a Pod as environment variables

Step 1: Create a ConfigMap

You can create a ConfigMap either via YAML or `kubectl`.

Option A: Using YAML

```
apiVersion: v1
kind: ConfigMap
metadata:
  name: app-config
data:
  LOG_LEVEL: debug
  APP_MODE: production
```

Save this as `configmap.yaml`, then apply it:

```
kubectl apply -f configmap.yaml
```

Option B: Using kubectl directly

```
kubectl create configmap app-config \
  --from-literal=LOG_LEVEL=debug \
  --from-literal=APP_MODE=production
```

Step 2: Use a ConfigMap in a Pod as Environment Variables

Here's how to inject the entire ConfigMap or specific keys as environment variables.

Option A: Use all keys in the ConfigMap

```
apiVersion: v1
kind: Pod
metadata:
  name: configmap-env-pod
spec:
  containers:
    - name: myapp
      image: busybox
      command: ["sh", "-c", "env && sleep 3600"]
      envFrom:
        - configMapRef:
            name: app-config
```

This will inject:

```
LOG_LEVEL=debug
APP_MODE=production
```

Option B: Use specific keys from the ConfigMap

```
apiVersion: v1
kind: Pod
metadata:
  name: configmap-key-pod
spec:
```

```
  containers:
    - name: myapp
      image: busybox
      command: ["sh", "-c", "env && sleep 3600"]
      env:
        - name: LOG_LEVEL
          valueFrom:
            configMapKeyRef:
              name: app-config
              key: LOG_LEVEL
        - name: MODE
          valueFrom:
            configMapKeyRef:
              name: app-config
              key: APP_MODE
```

Now the environment will include:

```
LOG_LEVEL=debug
MODE=production
```

Tips

- You can view the environment variables inside the Pod using:

  ```
  kubectl exec configmap-env-pod -- printenv
  ```

- If a key is missing in the ConfigMap, the Pod **will fail to start** unless you use `optional: true`.

Use a ConfigMap inside a Pod as a volume

This is useful when your app expects configuration as files rather than environment variables.

Step 1: Create the ConfigMap

Let's say we want to create a config file with some settings.

ConfigMap with file-like data:

```
apiVersion: v1
kind: ConfigMap
metadata:
  name: app-config
```

```
data:
  app.properties: |
    LOG_LEVEL=debug
    APP_MODE=production
```

Apply it:

```
kubectl apply -f configmap.yaml
```

This will create a ConfigMap with one key `app.properties` whose value is a multi-line string.

Step 2: Mount the ConfigMap as a Volume in a Pod

```
apiVersion: v1
kind: Pod
metadata:
  name: configmap-volume-pod
spec:
  containers:
    - name: myapp
      image: busybox
      command: ["sh", "-c", "cat /etc/config/app.properties && sleep 3600"]
      volumeMounts:
        - name: config-volume
          mountPath: /etc/config
  volumes:
    - name: config-volume
      configMap:
        name: app-config
```

What this does:

- Mounts the ConfigMap `app-config` as a volume at `/etc/config`.
- The key `app.properties` becomes a file named `/etc/config/app.properties`.
- Inside the container, this file will contain:

  ```
  LOG_LEVEL=debug
  APP_MODE=production
  ```

Each key becomes a separate file

If your ConfigMap looks like this:

```
data:
  LOG_LEVEL: debug
  APP_MODE: production
```

Then `/etc/config/LOG_LEVEL` and `/etc/config/APP_MODE` will be **two separate files**, each containing just their value.

Best Practices

- Only mount what's needed — for security and clarity.
- Combine this with init containers if you need to pre-process config files.
- Use defaultMode to set permissions (e.g., `defaultMode: 0400`).

Secrets

Use Cases:

- Database usernames and passwords.
- OAuth tokens.
- SSH keys, TLS certificates.
- External API keys (e.g., Stripe, Twilio).

Mounting as a Volume:

Secrets can also be mounted as files:

```
volumeMounts:
  - name: secret-volume
    mountPath: /etc/secrets
volumes:
  - name: secret-volume
    secret:
      secretName: my-secret
```

Security Considerations:

- **Base64 is *not* encryption** — it's just encoding.
- Use **encryption at rest** via `EncryptionConfiguration` and enable it in the kube-apiserver.
- Use **RBAC** to tightly control access to Secrets.
- Avoid logging secrets accidentally (e.g., printing env vars).

- In cloud environments (like AWS/EKS), integrate with KMS (Key Management Service) to auto-encrypt secrets.

How to create Kubernetes Secrets from literal values and how to define them using a YAML manifest.

---

1. Create Secrets from Literal Values (CLI)

You can use `kubectl` to create a Secret quickly from the command line using `--from-literal`.

Example:

```
kubectl create secret generic db-secret \
  --from-literal=username=admin \
  --from-literal=password=Pa55w0rd!
```

This creates a Secret named `db-secret` with two key-value pairs:

- `username: admin`
- `password: Pa55w0rd!`

To view the Secret:

```
kubectl get secret db-secret -o yaml
```

To decode it:

```
kubectl get secret db-secret -o jsonpath="{.data.username}" | base64 -d
```

---

2. Define the Secret Using a YAML Manifest

Here's the declarative way to define the same Secret:

```
apiVersion: v1
kind: Secret
metadata:
  name: db-secret
type: Opaque
data:
  username: YWRtaW4=        # "admin"
  password: UGE1NXcwcmQh    # "Pa55w0rd!" (base64 encoded)
```

Encode values to base64:

To get the base64-encoded version of values:

```
echo -n "admin" | base64        # Outputs: YWRtaW4=
echo -n "Pa55w0rd!" | base64     # Outputs: UGE1NXcwcmQh
```

Apply it:

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

Tip: Use `stringData` for plain-text (auto-encoded)

Kubernetes will automatically base64-encode values in `stringData`. Easier for editing and readability:

```
apiVersion: v1
kind: Secret
metadata:
  name: db-secret
type: Opaque
stringData:
  username: admin
  password: Pa55w0rd!
```

---

how to **use Kubernetes Secrets inside Pods,** both:

- As **environment variables**
- As a **mounted volume**

Step 1: Create the Secret

We'll start with a basic secret with two keys: `username` and `password`.

Option A: Using a YAML manifest

```
apiVersion: v1
kind: Secret
metadata:
  name: db-secret
type: Opaque
stringData:
  username: admin
  password: Pa55w0rd!
```

Apply it:

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

Step 2A: Use Secret as Environment Variables in a Pod

```
apiVersion: v1
kind: Pod
metadata:
  name: secret-env-pod
spec:
  containers:
    - name: app
      image: busybox
      command: ["sh", "-c", "echo USER=$DB_USER && echo PASS=$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
```

Inside the container:

You'll get:

```
DB_USER=admin
DB_PASS=Pa55w0rd!
```

Step 2B: Use Secret as a Volume in a Pod

```
apiVersion: v1
kind: Pod
metadata:
  name: secret-vol-pod
spec:
  containers:
```

```
    - name: app
      image: busybox
      command: ["sh", "-c", "cat /etc/secrets/username && cat /etc/secrets/password && sleep 36
00"]
      volumeMounts:
        - name: secret-vol
          mountPath: /etc/secrets
          readOnly: true
  volumes:
    - name: secret-vol
      secret:
        secretName: db-secret
```

Inside the container:

- `/etc/secrets/username` → contains `admin`
- `/etc/secrets/password` → contains `Pa55w0rd!`

Each key becomes a file in the mounted directory.

Security Tips

- Mounted secrets are stored in memory (tmpfs) by the Kubelet.
- Mark secrets as `readOnly: true` to avoid accidental changes.
- Use `defaultMode` to set file permissions:

  ```
  defaultMode: 0400
  ```

Example: Using Both in a Pod

Here's how you can use both a ConfigMap and a Secret in a Pod:

```
apiVersion: v1
kind: Pod
metadata:
  name: myapp
spec:
  containers:
    - name: app
      image: myapp:latest
      env:
        - name: LOG_LEVEL
```

```
        valueFrom:
          configMapKeyRef:
            name: my-config
            key: LOG_LEVEL
        - name: DB_PASSWORD
          valueFrom:
            secretKeyRef:
              name: my-secret
              key: password
      volumeMounts:
        - name: config-vol
          mountPath: /etc/config
        - name: secret-vol
          mountPath: /etc/secrets
  volumes:
    - name: config-vol
      configMap:
        name: my-config
    - name: secret-vol
      secret:
        secretName: my-secret
```

Best Practices

Recommendation	ConfigMap	Secret
Separate config from code	✅	✅
Avoid hardcoding sensitive data	❌	✅
Restrict access using RBAC	✅	✅
Rotate data regularly	🚫 (manual)	✅
Use Git for versioning	✅	❌ (not recommended)

Ingress

What is Ingress?

An Ingress is an API object in Kubernetes that manages external access to services within a cluster, typically via HTTP/HTTPS. It allows you to define rules for routing traffic based on hostnames or URL paths.

It does not expose arbitrary ports or protocols and generally works with an Ingress Controller like NGINX, Traefik, or cloud-specific controllers (e.g., GCP, AWS ALB).

Users do not directly connect to the Service; they reach the Ingerss endpoint and from there the request is directed to the Service

---

Why Use Ingress?

Ingress provides a centralized way to manage external HTTP(S) traffic into your Kubernetes services. Without Ingress, each service would need its own LoadBalancer or NodePort, which can be inefficient and hard to maintain.

Key Components

1. Ingress Resource

This defines how traffic should be routed.

Example:

```
apiVersion: networking.k8s.io/v1
kind: Ingress
metadata:
  name: example-ingress
spec:
  rules:
    - http:
        paths:
          - path: /app
            pathType: Prefix
            backend:
              service:
                name: app-service
                port:
                  number: 80
          - path: /api
            pathType: Prefix
            backend:
              service:
                name: api-service
                port:
                  number: 8080
```

2. Ingress Controller

The controller implements the rules defined in the Ingress resource. Popular options include:

- NGINX Ingress Controller

- Traefik
- Istio
- AWS ALB Ingress Controller
- Google Cloud Ingress

You must install an Ingress controller before you can use Ingress resources.

It is also known as:

- Controllers
- Ingres Proxy
- Service Proxy
- Reverse Proxy

minikube Nginx Ingress Default Controller

```
minikube addons enable ingress
```

Types of Ingress Rules

A. Simple Fanout

Route traffic to different services based on URL path.

```
IP   → /service1 → Service1
     → /service2 → Service2
```

B. Name-based Virtual Hosting

Route traffic based on the hostname.

Example:

```
rules:
  - host: app.example.com
    http:
      paths:
        - path: /
          pathType: Prefix
          backend:
            service:
              name: app-service
              port:
```

```
            number: 80
  - host: blog.example.com
    http:
      paths:
        - path: /
          pathType: Prefix
          backend:
            service:
              name: blog-service
              port:
                number: 80
```

C. TLS Termination

You can secure your Ingress by configuring TLS using a Kubernetes Secret.

Example:

```
spec:
  tls:
    - hosts:
        - app.example.com
      secretName: tls-secret
  rules:
    ...
```

How to Deploy Ingress

Step-by-step:

1. **Deploy Services** – Ensure your services are up and running.
2. **Install Ingress Controller** – For example, NGINX:

   ```
   helm repo add ingress-nginx <https://kubernetes.github.io/ingress-nginx>
   helm install ingress-nginx ingress-nginx/ingress-nginx
   ```

3. **Create Ingress Resource**:

   ```
   kubectl apply -f ingress.yaml
   ```

Example: Full Setup

Service Definitions

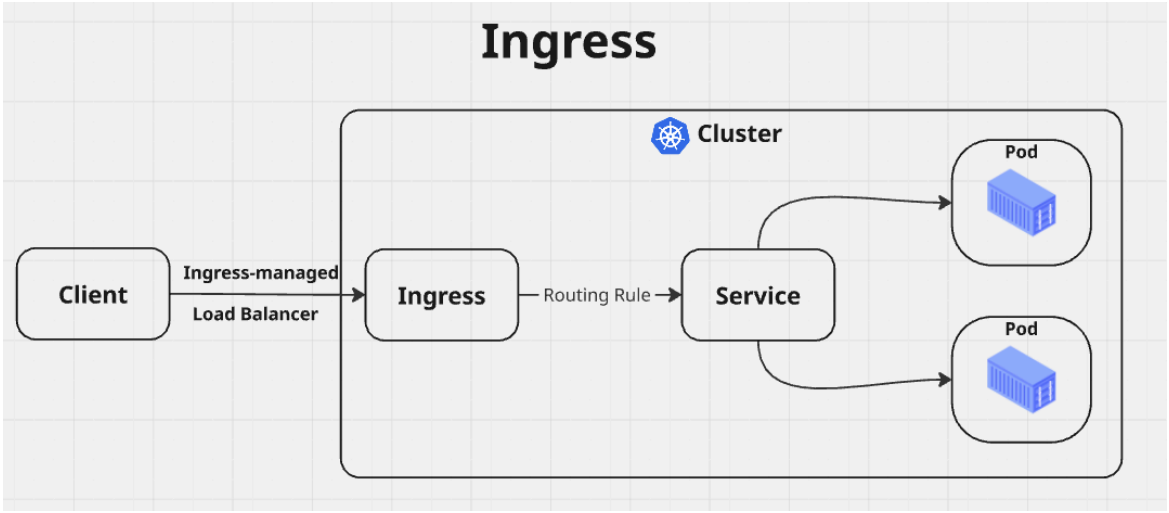
```
# app-service.yaml
apiVersion: v1
kind: Service
metadata:
  name: app-service
spec:
  selector:
    app: my-app
  ports:
    - protocol: TCP
      port: 80
      targetPort: 3000
```

```
# api-service.yaml
apiVersion: v1
kind: Service
metadata:
  name: api-service
spec:
  selector:
    app: my-api
  ports:
    - protocol: TCP
      port: 8080
      targetPort: 5000
```

Ingress

```
# ingress.yaml
apiVersion: networking.k8s.io/v1
kind: Ingress
metadata:
  name: multi-path-ingress
spec:
  rules:
    - http:
        paths:
          - path: /app
            pathType: Prefix
            backend:
```

```
      service:
        name: app-service
        port:
          number: 80
    - path: /api
      pathType: Prefix
      backend:
        service:
          name: api-service
          port:
            number: 8080
```



Annotations

Annotations are key-value pairs that can be attached to Kubernetes objects such as Pods, Services, Ingresses, Deployments, and more. They are used to store metadata or configuration information that is not intended for identifying or selecting resources.

Unlike **labels**, which are used for organizing and selecting resources, **annotations** are meant for storing arbitrary, often non-identifying, metadata.

Purpose of Annotations

Annotations serve several purposes:

- Store additional information about an object (e.g., build version, release name, git commit hash)
- Provide configuration options specific to controllers or third-party tools

- Enable customization of behavior on a per-object basis

They are particularly useful when you need to pass instructions to controllers or operators that manage your workloads.

Syntax

Annotations are defined under the `metadata.annotations` field in a resource definition:

```
metadata:
  name: my-ingress
  annotations:
    nginx.ingress.kubernetes.io/rewrite-target: /
    example.com/custom-annotation: "true"
```

Common Use Cases

1. Ingress Controllers

Annotations are widely used with Ingress resources to control how traffic is routed or handled by the controller.

Example using NGINX Ingress Controller:

```
annotations:
  nginx.ingress.kubernetes.io/rewrite-target: /
  nginx.ingress.kubernetes.io/ssl-redirect: "true"
  nginx.ingress.kubernetes.io/canary: "true"
  nginx.ingress.kubernetes.io/canary-weight: "20"
```

These annotations tell the NGINX Ingress Controller to:

- Rewrite the URL path
- Redirect HTTP to HTTPS
- Enable canary deployment
- Route 20% of traffic to this Ingress

2. Service Configuration

Some cloud providers use annotations to configure services of type `LoadBalancer`.

Example (AWS ELB):

```
annotations:
  service.beta.kubernetes.io/aws-load-balancer-backend-protocol: tcp
```

```
    service.beta.kubernetes.io/aws-load-balancer-type: nlb
```

This tells Kubernetes to create a Network Load Balancer instead of a Classic Load Balancer and use TCP for backend communication.

3. Build and Deployment Tools

Tools like Helm, ArgoCD, Jenkins X, etc., often use annotations to track deployment metadata.

Example:

```
annotations:
  helm.sh/hook: pre-install
  argocd.argoproj.io/sync-wave: "1"
```

4. Custom Metadata

You can also define your own custom annotations for documentation or internal tooling:

```
annotations:
  owner: dev-team
  last-updated: "2025-04-05"
  changelog: <https://example.com/changelog/v1.2.0>
```

Key Differences Between Labels and Annotations

Feature	Labels	Annotations
Purpose	Identify and select resources	Attach metadata or configuration
Format	Key-value pairs	Key-value pairs
Selectors	Can be used in label selectors	Not used in selectors
Length & Format	Limited character set and length	No restrictions on format or length
Usage	Organizing resources	Customization, documentation, integrations

---

Best Practices

- Use labels for grouping and selecting resources.
- Use annotations for extended metadata or integration-specific settings.
- Avoid putting sensitive data in annotations unless necessary.
- Be consistent with naming conventions, especially for custom annotations (use reverse domain notation like `example.com/my-annotation`).

Quota and Limits Management in Kubernetes

In Kubernetes, **resource quotas** and **limit ranges** are used to manage resource consumption (CPU and memory) across namespaces. These mechanisms help ensure that no single application or user can consume excessive resources, which could impact the stability and performance of the cluster.

---

1. Resource Quotas (`ResourceQuota`)

A `ResourceQuota` restricts the total amount of resources that can be consumed within a namespace. It applies at the **namespace level** and limits:

- Total number of objects (e.g., Pods, Services)
- Total CPU and memory usage for all containers in the namespace
- Storage requests

Example: ResourceQuota Definition

```
apiVersion: v1
kind: ResourceQuota
metadata:
  name: dev-quota
  namespace: development
spec:
  hard:
    pods: "20"
    services: "10"
    cpu: "2"
    memory: "4Gi"
    persistentvolumeclaims: "5"
```

This quota ensures that the `development` namespace cannot exceed:

- 20 Pods
- 10 Services
- 2 CPU cores total
- 4 GB of memory total
- 5 Persistent Volume Claims

View Resource Quota Usage

```
kubectl get resourcequota -n development
```

---

2. Limit Ranges (`LimitRange`)

A `LimitRange` defines **minimum and maximum** constraints on resource usage **per container or Pod**, within a namespace. It helps enforce consistency and prevents misconfigurations.

It can specify:

- Minimum and maximum CPU/memory for containers
- Default requests and limits if not specified

Example: LimitRange Definition

```
apiVersion: v1
kind: LimitRange
metadata:
  name: mem-limit-range
  namespace: development
spec:
  limits:
    - default:
        cpu: "1"
        memory: "512Mi"
      defaultRequest:
        cpu: "0.5"
        memory: "256Mi"
      type: Container
```

This configuration:

- Sets a default request of 0.5 CPU and 256Mi memory per container
- Sets a default limit of 1 CPU and 512Mi memory per container
- Applies only to containers (`type: Container`)

View Limit Ranges

```
kubectl get limitranges -n development
```

---

3. How Resource Quotas and Limit Ranges Work Together

Feature	ResourceQuota	LimitRange
Scope	Namespace	Namespace
Purpose	Controls total resource usage in a namespace	Controls individual container/Pod resource usage

Enforces	Total limits across all resources	Min/max/default values per container or Pod
Can Be Used With	LimitRange	ResourceQuota

Together, they provide a layered approach to resource management:

- **LimitRange** ensures that individual containers don't ask for too much (or too little).
- **ResourceQuota** ensures that the overall namespace doesn't consume more than its fair share.

4. Monitoring Resource Usage

You can monitor current usage against quotas using:

```
kubectl describe resourcequota <quota-name> -n <namespace>
```

Example Output:

```
Name:          dev-quota
Namespace:     development
Resource       Used  Hard
--------       ----  ----
cpu            1     2
memory         768Mi 4Gi
pods           3     20
services       1     10
```

5. Best Practices

- Always define **ResourceQuotas** in multi-tenant clusters.
- Use **LimitRanges** to enforce defaults and prevent resource starvation.
- Monitor usage regularly and adjust quotas as needed.
- Combine with **Horizontal Pod Autoscaler (HPA)** for dynamic scaling under defined limits.
- Educate developers about setting proper resource requests and limits.

Autoscaling in Kubernetes

HPA, VPA, and Cluster Autoscaler

Autoscaling in Kubernetes allows you to automatically adjust resources based on current workload demands. This helps improve application performance while optimizing resource usage and cost.

There are three main types of autoscalers:

1. Horizontal Pod Autoscaler (HPA)

What it does:

HPA automatically scales the number of **Pod replicas** in a Deployment, ReplicaSet, or StatefulSet based on observed metrics like CPU utilization or custom metrics (e.g., HTTP requests per second).

Use case:

Scale out your application when demand increases (e.g., during traffic spikes), and scale in when demand decreases.

Example: HPA Definition (based on CPU)

```
apiVersion: autoscaling/v2
kind: HorizontalPodAutoscaler
metadata:
  name: nginx-hpa
  namespace: default
spec:
  scaleTargetRef:
    apiVersion: apps/v1
    kind: Deployment
    name: nginx-deployment
  minReplicas: 2
  maxReplicas: 10
  metrics:
    - type: Resource
      resource:
        name: cpu
        target:
          type: Utilization
          averageUtilization: 50
```

This will:

- Keep at least 2 Pods running
- Scale up to 10 Pods if CPU usage exceeds 50%
- Automatically adjusts the number of replicas every few minutes

Commands

```
# Create HPA using kubectl
kubectl autoscale deployment nginx-deployment --cpu-percent=50 --min=2 --max=10
```

```
# View HPA status
kubectl get hpa
```

2. Vertical Pod Autoscaler (VPA)

What it does:

VPA automatically adjusts the **CPU and memory requests** for containers in a Pod. It ensures that Pods get the right amount of resources they need to run efficiently.

Use case:

Right-size container resource requests to avoid over-provisioning or under-provisioning.

> ⚠️ Note: VPA typically recreates Pods to apply new resource settings. This means some downtime can occur unless used with rolling updates or eviction strategies.

Components of VPA

- **Recommender:** Analyzes historical and current resource usage to recommend values.
- **Updater:** Evicts Pods so that the recommender's suggestions can be applied.
- **Admission Controller:** Applies the recommended resource limits when new Pods are created.

Example: VPA Configuration

```
apiVersion: autoscaling.k8s.io/v1
kind: VerticalPodAutoscaler
metadata:
  name: my-app-vpa
spec:
  targetRef:
    apiVersion: "apps/v1"
    kind:       Deployment
    name:       my-app-deployment
  updatePolicy:
    updateMode: "Auto"
```

This configuration tells VPA to automatically update the resource requests for `my-app-deployment` based on recommendations.

3. Cluster Autoscaler

What it does:

Cluster Autoscaler automatically adjusts the **number of nodes** in a Kubernetes cluster based on the resource needs of pending Pods.

Use case:

Add more worker nodes when there is not enough capacity to schedule new Pods, and remove unused nodes when they're no longer needed.

> ✅ Works only with clusters managed by cloud providers (like AWS ASG, GKE, AKS) or tools like Kops or KubeSphere.

How it works:

- If a Pod cannot be scheduled due to insufficient resources, the Cluster Autoscaler adds a node to the node pool.
- If a node is underutilized and all its Pods can be moved elsewhere, the node is removed from the cluster.

Enabling Cluster Autoscaler

This depends on your environment:

- **AWS**: Use Auto Scaling Groups with kubelet integration.
- **GCP (GKE)**: Enable autoscaling when creating or editing a node pool.
- **Azure (AKS)**: Enable autoscaling via the Azure portal or CLI.
- **On-premises**: Not directly supported; consider alternatives like Karpenter.

Comparison Table

Feature	Horizontal Pod Autoscaler (HPA)	Vertical Pod Autoscaler (VPA)	Cluster Autoscaler
Scales	Number of Pod replicas	Container resource requests	Number of nodes
Based On	CPU/Memory usage, custom metrics	Historical usage patterns	Pending Pods / Node utilization
Scope	Workload level (Deployment, ReplicaSet)	Workload level (Pod containers)	Cluster level
Impact	Increases/decreases app capacity	Optimizes per-container resource usage	Adds/removes infrastructure
Downtime	None (if using rolling updates)	Possible (Pod recreation)	Possible (node drain)

Best Practices

- Use **HPA + Cluster Autoscaler** together to handle both workload and infrastructure scaling.
- Use **VPA** carefully in production due to Pod restarts; combine with disruption budgets.

- Monitor metrics regularly to tune autoscaling thresholds.
- Combine autoscaling with proper resource requests and limits to ensure efficiency.
- Test autoscaling behavior under load before deploying to production.

Jobs and Cronjobs in Kubernetes

In Kubernetes, **Jobs** and **CronJobs** are used to manage **batch workloads,** tasks that run to completion rather than running continuously like web servers.

---

1. Job

A **Job** creates one or more **Pods** and ensures that a specified number of them successfully terminate. It is useful for **one-time or retryable batch tasks**, such as:

- Running a data migration script
- Performing a database backup
- Executing a machine learning model training job

Key Features of Jobs:

- Guarantees that **one or more Pods complete successfully**
- Supports parallel execution (via `parallelism`)
- Can retry failed Pods up to a limit (`backoffLimit`)

Example: Job Definition

```
apiVersion: batch/v1
kind: Job
metadata:
  name: pi-calculation-job
spec:
  template:
    spec:
      containers:
        - name: pi
          image: perl
          command: ["perl", "-Mbignum=bpi", "-wle", "print bpi(2000)"]
      restartPolicy: OnFailure
  backoffLimit: 4
  completions: 1
  parallelism: 1
```

This Job runs a Perl container to calculate π to 2000 decimal places.

Commands

```
# Apply the job
kubectl apply -f job.yaml

# View jobs
kubectl get jobs

# View pods created by the job
kubectl get pods

# Check logs of a completed pod
kubectl logs <pod-name>
```

---

configuration options available when defining a **Job** in Kubernetes:

---

1. `parallelism`

- **What it does:**

  Specifies the **maximum number of Pods** that can run **simultaneously** for this Job.

- **Default value:**

  If not set, defaults to **1**.

- **Use case:**

  Useful for parallelizing work — for example, processing multiple files or records concurrently.

```
parallelism: 5
```

> This means up to 5 Pods will be created and run at the same time.

---

2. `completions`

- **What it does:**

  Sets the **number of successful Pod completions** required before the Job is considered complete.

- **Default value:**

  If not set, defaults to **1**.

- **Use case:**

For sequential or parallel batch tasks where you want a fixed number of successes.

```
completions: 3
```

> The Job will only succeed after 3 Pods have completed successfully.

---

3. `activeDeadlineSeconds`

- **What it does:**

  Defines the **maximum amount of time** (in seconds) that a Job can be active before it is automatically terminated and marked as failed.

- **Use case:**

  Prevents long-running Jobs from consuming resources indefinitely.

```
activeDeadlineSeconds: 600
```

> The Job must finish within 10 minutes or it will be stopped.

---

4. `backoffLimit`

- **What it does:**

  Specifies the **number of retries** before marking the Job as **failed** due to repeated failures.

- **Default value:**

  If not set, defaults to **6**.

- **Use case:**

  Useful for handling transient errors (e.g., network issues).

```
backoffLimit: 3
```

> The Job will retry failed Pods up to 3 times before giving up.

---

5. `ttlSecondsAfterFinished`

- **What it does:**

  Defines the **time-to-live (TTL)** in seconds for a **completed or failed Job**, after which it will be automatically deleted.

- **Use case:**

Clean up finished Jobs automatically instead of manually deleting them.

```
ttlSecondsAfterFinished: 86400
```

> The Job will be deleted automatically 24 hours (86400 seconds) after completion or failure.

Example: Full Job YAML with All Options

```
apiVersion: batch/v1
kind: Job
metadata:
  name: process-data-job
spec:
  template:
    spec:
      containers:
        - name: data-processor
          image: my-data-processor:latest
          command: ["sh", "-c", "echo Processing data && exit 0"]
      restartPolicy: OnFailure
  parallelism: 3
  completions: 5
  activeDeadlineSeconds: 600
  backoffLimit: 3
  ttlSecondsAfterFinished: 86400
```

Summary Table

Field	Description	Default
`parallelism`	Max number of pods running at once	1
`completions`	Number of successful pod completions needed	1
`activeDeadlineSeconds`	Max duration the job can be active	Not set
`backoffLimit`	Max number of retries before marking job failed	6
`ttlSecondsAfterFinished`	Time before automatic cleanup of finished job	Not set

---

2. CronJob

A **CronJob** is used to run **Jobs on a repeating schedule**, similar to the Unix `cron` utility.

It is ideal for recurring tasks such as:

- Daily backups
- Hourly log cleanup
- Periodic health checks

Key Features of CronJobs:

- Scheduled using standard **cron syntax**
- Can specify a time zone (since v1.21)
- Controls how many instances can be active at once
- Optionally suspends scheduled executions

Example: CronJob Definition

```
apiVersion: batch/v1
kind: CronJob
metadata:
  name: daily-backup-cronjob
spec:
  schedule: "0 0 * * *"  # Runs every day at midnight
  timeZone: "UTC"
  jobTemplate:
    spec:
      template:
        spec:
          containers:
            - name: backup
              image: alpine
              command:
                - /bin/sh
                - -c
                - echo "Running daily backup at $(date)"
          restartPolicy: OnFailure
  successfulJobsHistoryLimit: 3
  failedJobsHistoryLimit: 1
  concurrencyPolicy: Forbid
  suspend: false
```

This CronJob runs a simple shell command every day at midnight UTC.

Commands

```
# Apply the cronjob
kubectl apply -f cronjob.yaml

# List cronjobs
kubectl get cronjobs

# List triggered jobs
kubectl get jobs

# List pods from triggered jobs
kubectl get pods
```

Comparison Table

Feature	Job	CronJob
Purpose	Run a task to completion	Run a task on a schedule
Execution Type	One-time or manual	Repeating (scheduled)
Scheduling	Not scheduled	Uses cron expressions
Runs Automatically	No (unless created via CronJob)	Yes (based on schedule)
Use Case	Ad-hoc batch processing	Periodic maintenance or batch tasks

Best Practices

- Always set appropriate values for `backoffLimit`, `completions`, and `parallelism` in Jobs.
- Use `successfulJobsHistoryLimit` and `failedJobsHistoryLimit` to control job retention.
- Use `concurrencyPolicy` in CronJobs to prevent overlapping executions:
  - `Allow`: Allow concurrent runs
  - `Forbid`: Do not allow concurrent runs
  - `Replace`: Replace the currently running job
  - `Forbid` is often safest for idempotent jobs
- Set `suspend: true` temporarily to disable a CronJob without deleting it.

StatefulSets in Kubernetes

A **StatefulSet** is a Kubernetes controller used to manage **stateful applications** — applications that require stable, unique network identifiers and persistent storage.

Unlike **Deployments**, which are designed for **stateless applications** (where Pods are interchangeable), StatefulSets ensure that each Pod has:

- A **sticky identity** (e.g., `pod-0`, `pod-1`, etc.)
- **Persistent storage** that remains associated with the Pod even if it gets rescheduled
- Ordered deployment, scaling, and deletion

When to Use StatefulSets

Use a StatefulSet when your application requires:

- Stable hostnames and network identities
- Persistent data storage per instance
- Ordered deployment or updates (e.g., master-slave setups)

Examples:

- Databases like MySQL, PostgreSQL (in clustered mode)
- Distributed data stores like ZooKeeper, etcd, Cassandra

Security Contexts and Pod Security Admission (PSA)

Security Contexts

A **Security Context** defines privilege and access control settings for a **Pod or individual container**. It helps enforce security policies at runtime.

Common Settings in a Security Context:

- Run as a specific user (`runAsUser`)
- Run as non-root user (`runAsNonRoot`)
- Allow/disallow privileged mode (`privileged`)
- Drop or add Linux capabilities (`capabilities`)
- SELinux or AppArmor profiles
- Read-only root filesystem (`readOnlyRootFilesystem`)

Example:

```
securityContext:
  runAsUser: 1000
```

```
runAsNonRoot: true
readOnlyRootFilesystem: true
```

You can define this at the **Pod level** or per **container**.

Pod Security Admission (PSA)

PSA is a built-in Kubernetes admission controller that enforces security policies on how Pods are configured. It replaces the older **Pod Security Policy (PSP)** mechanism.

PSA Policies

It offers three **predefined levels** of enforcement:

1. **Privileged** – Unrestricted, mostly for system Pods.
2. **Baseline** – Minimally restrictive, allows default Pod behavior.
3. **Restricted** – Follows strict security best practices (e.g., must run as non-root).

Enabling PSA

You enable it by labeling namespaces:

```
kubectl label ns default pod-security.kubernetes.io/enforce=baseline
```

This tells Kubernetes to enforce **baseline policy** on all Pods in the `default` namespace.

Summary

Feature	Description
Security Context	Defines how a Pod/container runs (user, capabilities, etc.)
Pod Security Admission	Cluster-level enforcement of secure Pod configurations

They work together to help you run safer, more secure workloads in Kubernetes.

Network Policies

What it is:

A Kubernetes resource that controls **network traffic between Pods** and optionally to external networks.

Purpose

- Restrict communication between Pods
- Improve security by applying the **principle of least privilege**

- Prevent unauthorized access to services like databases or APIs

Key Features

- Define which Pods can talk to each other using **labels**
- Control **ingress** (incoming) and **egress** (outgoing) traffic
- Apply policies selectively to specific namespaces or Pods

Example Policy

```
apiVersion: networking.k8s.io/v1
kind: NetworkPolicy
metadata:
  name: db-policy
spec:
  podSelector:
    matchLabels:
      app: database
  ingress:
    - from:
        - podSelector:
            matchLabels:
              app: backend
  policyTypes:
    - Ingress
```

This allows only Pods with label `app: backend` to connect to Pods labeled `app: database`.

Requirements

- A **network plugin** that supports NetworkPolicy (e.g., Calico, Cilium, Weave Net)
- Enabled in the cluster (not all plugins enforce policies by default)

Default Behavior

- If no NetworkPolicy exists, **all Pods can communicate freely**
- Once a policy applies to a Pod, **only allowed traffic is permitted**

Monitoring, Logging, and Troubleshooting

Metrics Server

- Lightweight tool that collects basic CPU/memory metrics from nodes and Pods.
- Required for `kubectl top` and **Horizontal Pod Autoscaler (HPA)**.
- Easy to install and ideal for cluster-level resource monitoring.

Prometheus

- Full-featured monitoring system for collecting and querying custom metrics.
- Widely used with Kubernetes for detailed application and infrastructure monitoring.
- Works with Grafana for dashboards and Alertmanager for alerts.

Logging

- Container logs can be viewed using:

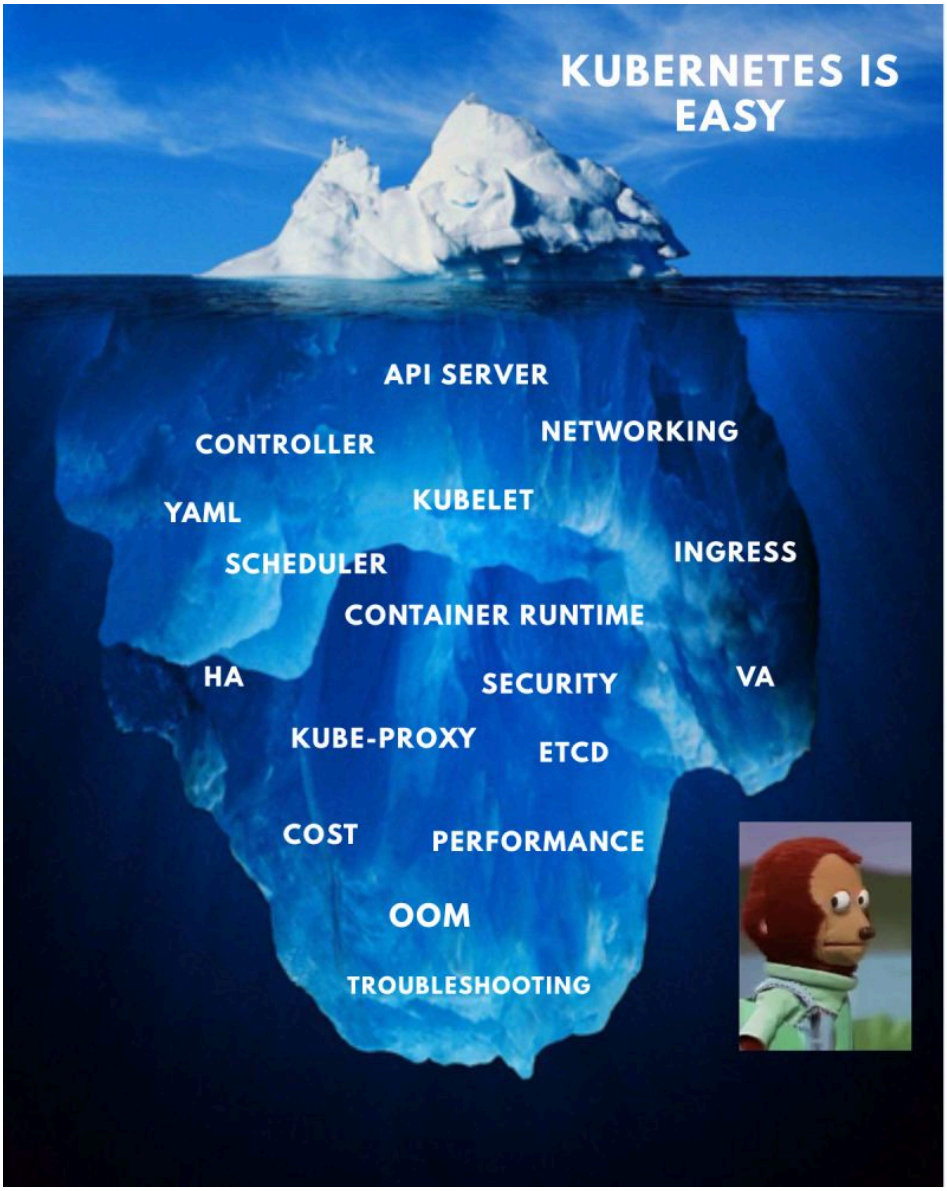
```
kubectl logs <pod-name>
```

- For centralized logging, tools like **Fluentd**, **Loki**, or **ELK Stack** are used.

Troubleshooting Tools

- `kubectl describe pod <pod-name>` – Check events and status
- `kubectl logs <pod-name>` – View container logs
- `kubectl get events` – See recent cluster events
- `kubectl exec -it <pod-name> -- sh` – Enter a running container

THE END


KUBERNETES IS EASY
API SERVER
CONTROLLER
NETWORKING
YAML
KUBELET
SCHEDULER
INGRESS
CONTAINER RUNTIME
HA
SECURITY
VA
KUBE-PROXY
ETCD
COST
PERFORMANCE
OOM
TROUBLESHOOTING