Kubernetes Overview

Yunus Emre Özdemir
Fahreddin Özcan

Introduction to Kubernetes

- What is Kubernetes?
- Why was Kubernetes created?
- Key benefits of using Kubernetes

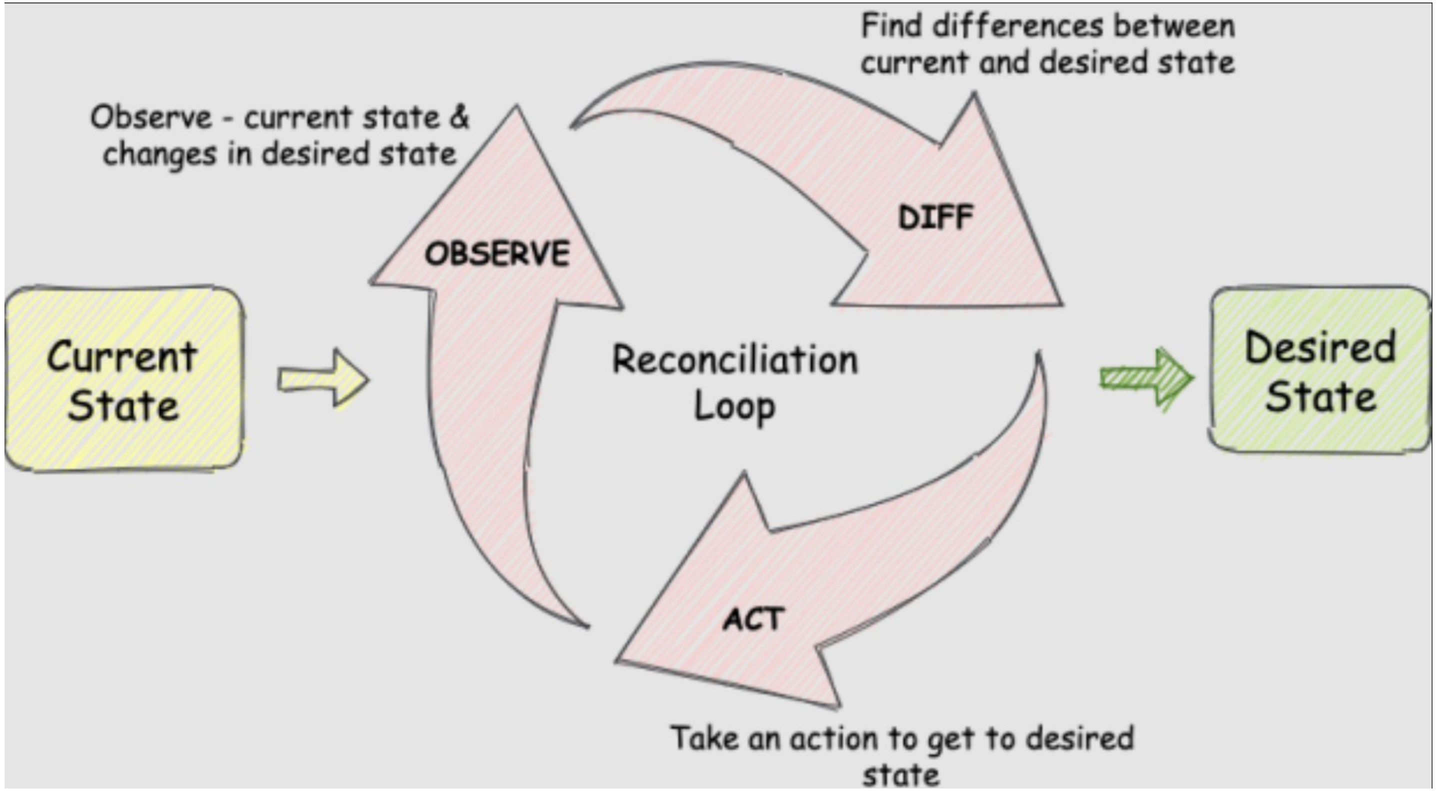
What is Kubernetes?

Kubernetes (often abbreviated as K8s) is an open-source container orchestration platform that automates the deployment, scaling, and management of containerized applications. Originally developed by Google and now maintained by the Cloud Native Computing Foundation (CNCF), Kubernetes has become the industry standard for managing microservices-based applications.

What is Kubernetes?

Kubernetes is not a mere orchestration system. In fact, it eliminates the need for orchestration. The technical definition of orchestration is execution of a defined workflow: first do A, then B, then C. In contrast, Kubernetes comprises a set of independent, composable control processes that continuously drive the current state towards the provided desired state. It shouldn't matter how you get from A to C. Centralized control is also not required. This results in a system that is easier to use and more powerful, robust, resilient, and extensible.

The Concept of Desired State in Kubernetes


Find differences between current and desired state
Observe - current state & changes in desired state
DIFF
OBSERVE
Current State
Reconciliation Loop
Desired State
ACT
Take an action to get to desired state

Evolution of Deployment Strategies

1️⃣ **Traditional Deployment (Physical Servers)**

- Applications ran directly on physical servers.
- No resource boundaries → One app could consume all resources.
- Scaling required adding more physical servers → **Expensive & inefficient.**

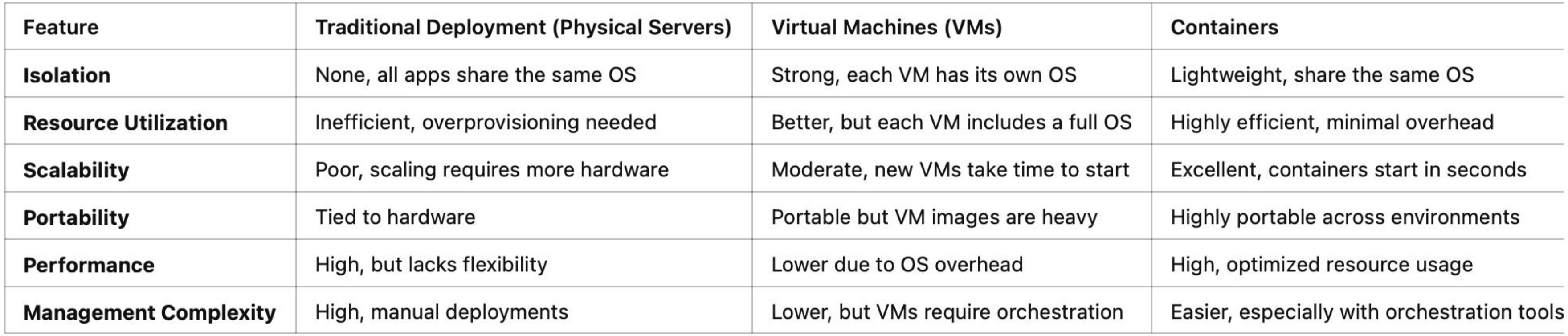
2️⃣ **Virtual Machines (VMs) Era**

- Introduced **virtualization** → Multiple VMs on a single physical server.
- **Improved resource utilization** & **isolation** between applications.
- Each VM runs its own OS, making it **heavier** than containers.

3️⃣ **Containerized Deployment Era**

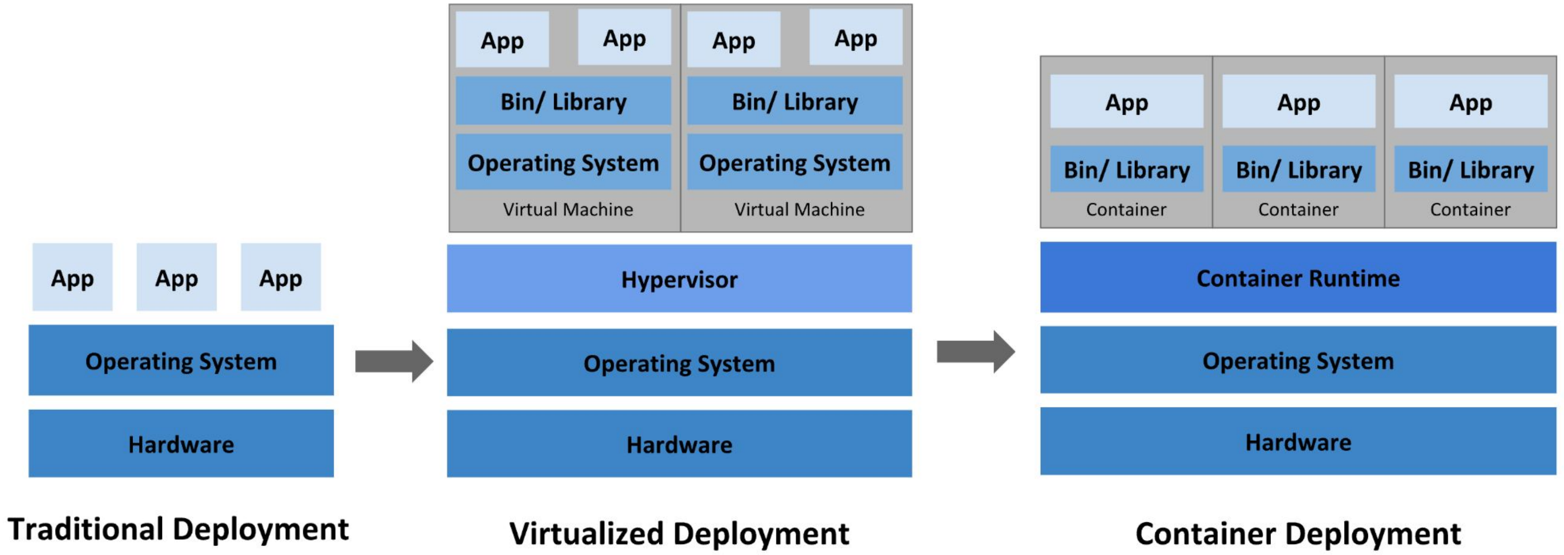
- Containers share the **host OS**, making them **lightweight** & **portable**.
- Faster startup, efficient scaling, and **cross-platform compatibility**.
- Ideal for **modern, cloud-native applications.**

Evolution of Deployment Strategies



Feature	Traditional Deployment (Physical Servers)	Virtual Machines (VMs)	Containers
Isolation	None, all apps share the same OS	Strong, each VM has its own OS	Lightweight, share the same OS
Resource Utilization	Inefficient, overprovisioning needed	Better, but each VM includes a full OS	Highly efficient, minimal overhead
Scalability	Poor, scaling requires more hardware	Moderate, new VMs take time to start	Excellent, containers start in seconds
Portability	Tied to hardware	Portable but VM images are heavy	Highly portable across environments
Performance	High, but lacks flexibility	Lower due to OS overhead	High, optimized resource usage
Management Complexity	High, manual deployments	Lower, but VMs require orchestration	Easier, especially with orchestration tools

Evolution of Deployment Strategies



Why Was Kubernetes Needed After Containers?

- Docker made containerization easy, but **managing containers at scale** required an orchestrator.
- Docker Swarm provided basic orchestration, but lacked **advanced networking, scaling, and multi-cluster support**.
- Kubernetes emerged as the **most powerful, flexible, and widely adopted** orchestration tool.

Google Borg & The Birth of Kubernetes

Google had been solving large-scale container management internally for years using a system called **Borg**. Borg allowed Google to efficiently manage workloads across thousands of servers. Inspired by Borg, Google engineers developed Kubernetes and open-sourced it in 2014.

Why Kubernetes?

Managing containers at scale is complex!

- How do we **deploy** thousands of containers?
- How do we **scale up and down** dynamically?
- How do we ensure **fault tolerance and load balancing**?
- How do we handle **networking, storage, and security**?

Key Benefits of Kubernetes

💡1️⃣Automated Deployment & Scaling

- Uses **YAML manifests** to define deployments.
- **Auto-scales** based on demand (Horizontal Pod Autoscaler).
- Easily **rolls out updates & rollbacks** if issues arise.

💡2️⃣Self-Healing & Fault Tolerance

- Automatically **restarts failed containers**.
- **Reschedules workloads** if a node goes down.
- **Detects unhealthy applications** and stops routing traffic to them.

Load Balancing & Service Discovery

⚖️3️⃣ **Intelligent Load Balancing**

- Evenly distributes traffic across multiple replicas.
- Uses **Service abstraction** to expose applications.

🔍4️⃣ **Built-in Service Discovery**

- No need for manual IP tracking!
- Kubernetes **assigns DNS names** to services automatically.

Portability & Multi-Cloud Support

☁️5️⃣ Works Anywhere

- Run the same application **on-prem, hybrid, or multi-cloud** environments.
- Kubernetes abstracts infrastructure, so applications **don’t depend on a specific cloud provider**.

🛠️6️⃣ Infrastructure as Code (IaC)

- Uses declarative **YAML configuration files** for reproducibility & automation.
- Easily integrates with **CI/CD pipelines** for automated deployments.

Security & Storage Management

🔒7️⃣ Enhanced Security

- Supports **RBAC (Role-Based Access Control)** for managing access.
- Uses **network policies** to define communication rules between services.

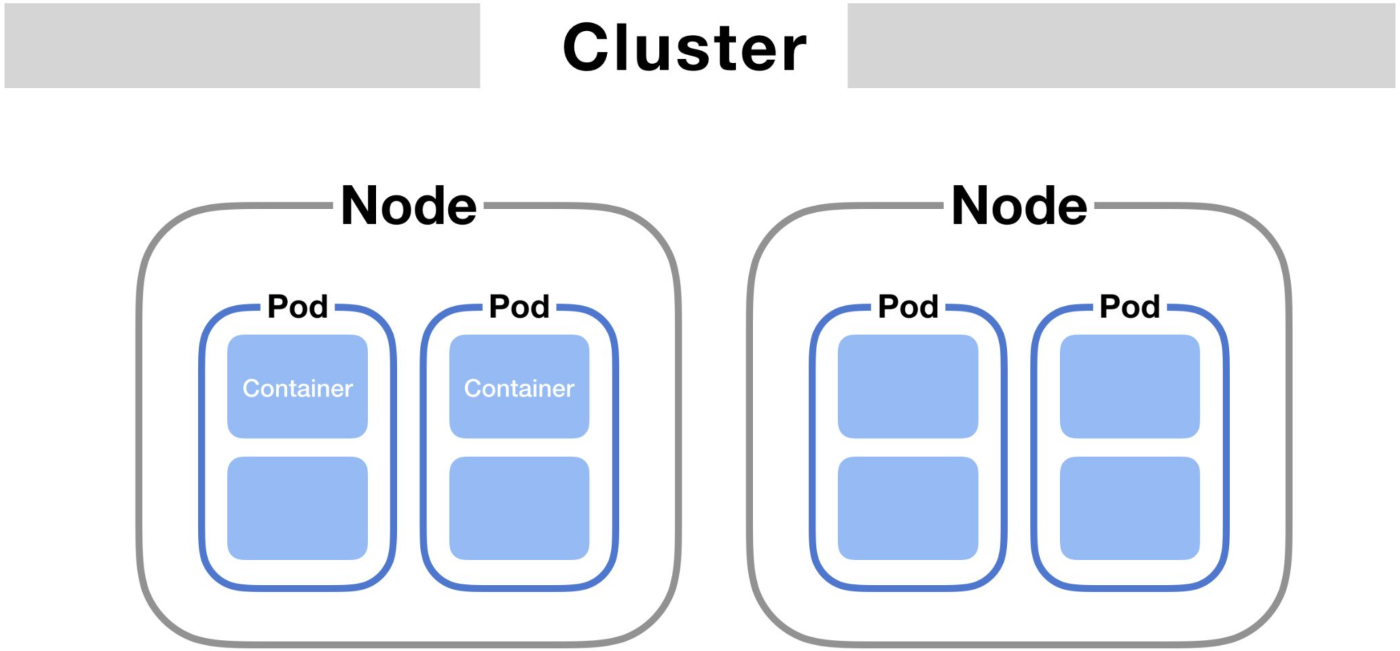
🗄️8️⃣ Persistent Storage Support

- Integrates with **Persistent Volumes (PV, PVC)** for stateful applications.
- Supports **cloud storage (AWS EBS, Google Persistent Disk, Azure Disk, etc.)**.

Core Kubernetes Concepts

- Pods
- Nodes & Clusters
- Services (ClusterIP, NodePort, LoadBalancer)
- Deployments & ReplicaSets
- Namespaces
- ConfigMaps & Secrets

Pods, Nodes & Clusters



Pods – The Building Block of Kubernetes

What is a Pod?

A **Pod** is the smallest deployable unit in Kubernetes. It represents one or more **containers** that share the same **network, storage, and lifecycle**.

How Does a Pod Relate to Containers?

- A Pod can run **one or multiple containers** that work together.
- Containers in a Pod **share the same IP address and port space**, allowing them to communicate via localhost.

They can also share **storage volumes**.

Example: Single vs. Multi-Container Pod

- **Single-Container Pod:** Most common use case, where a Pod wraps around a single application container.
- **Multi-Container Pod:** Useful when containers need to work closely together, like:
    - A **main app** and a **sidecar container** (e.g., a logging or proxy container).

Why Do We Need an Additional Abstraction?

Container Lifecycle Management

- Containers alone don’t have built-in self-healing.
- If a container crashes, Kubernetes uses Pods to restart them.

Networking Simplicity

- Containers within the same Pod communicate over localhost, avoiding complex networking setups.

Storage Sharing

- Containers in the same Pod can share persistent storage **without mounting external volumes** separately.

Deployment & Scaling

- Instead of managing individual containers, Kubernetes schedules and scales **Pods** as a unit.
- ReplicaSets and Deployments ensure multiple copies of a Pod can be managed efficiently.

Nodes & Clusters

What is a Node?

- A **Node** is a machine (VM or physical) that runs containerized workloads.
- It is managed by the **Kubernetes Control Plane**.

What is a Cluster?

- A **Cluster** is a collection of Nodes managed by Kubernetes.
- The **Control Plane** schedules workloads across Nodes.
- Nodes communicate with each other to ensure application availability.

Services & Pod Communication

What is a Service?

- An abstraction that **defines a logical set of Pods**.
- Provides a **stable network endpoint** (IP & DNS) for accessing these Pods.
- Implements **load balancing** and health checks.

Why are Services Necessary?

- **Ephemeral Pods:** Pod IPs can change upon restart or rescheduling.
- **Consistent Connectivity:** Services offer a persistent interface despite underlying changes.
- **Simplified Communication:** Facilitates reliable inter-Pod and external communication.

ClusterIP (Default) – Internal Communication

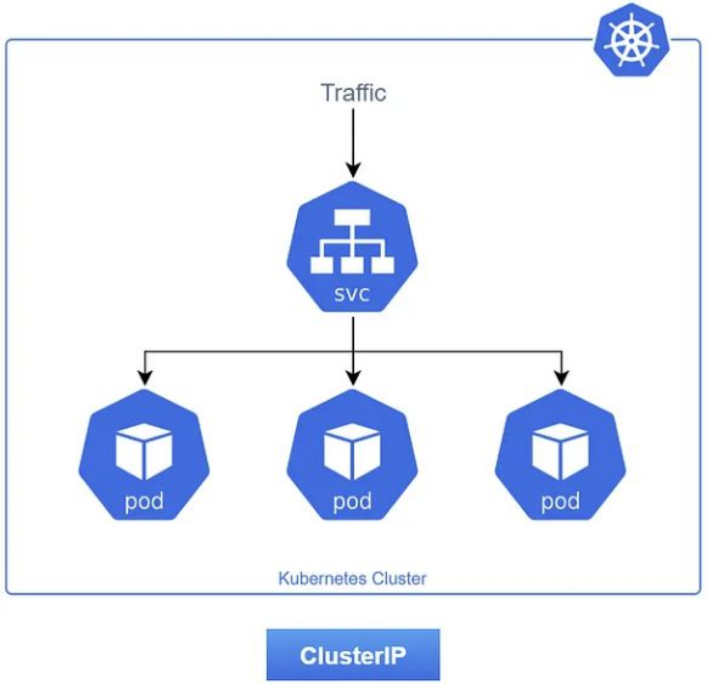
A **ClusterIP service** exposes the application **only within the cluster**. It is used for **internal communication** between microservices.

How it works?

- Kubernetes assigns a **stable, internal IP**.
- Other pods can reach the service using **DNS (service-name.namespace.svc.cluster.local)**.

Use Case:

✅ Backend microservices (e.g., API calling a database).



NodePort – Exposing a Service on Each Node

A **NodePort service** makes an application accessible **externally on a static port (30000–32767)** on **each node** in the cluster.

How it works?

- Every node in the cluster opens the **same port**.
- External users can access the service via:

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

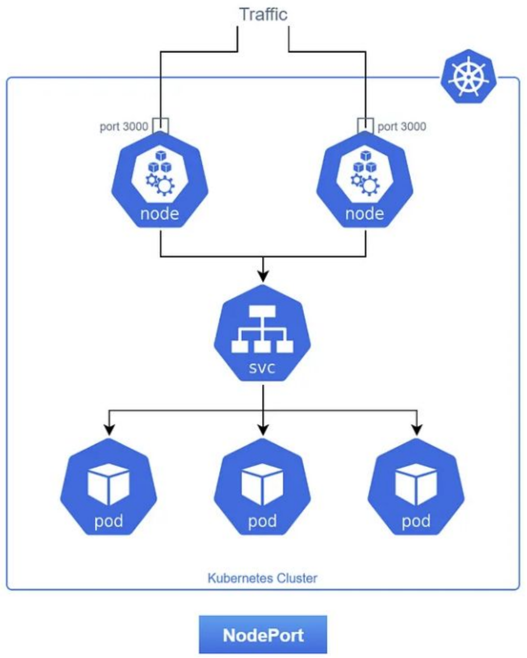
Use Case:

✅ Simple external access (not recommended for production).

🔹 Users can access: http://<node-ip>:30001

📌 **Limitations:**

- **Exposes service on all nodes**, even if the app runs on one.
- **Not suitable for production** (use LoadBalancer or Ingress instead).



LoadBalancer – Cloud-Managed External Access

A **LoadBalancer service** integrates with cloud provider **load balancers (AWS, Azure, GCP)** to expose a service externally.

How it works?

- The cloud provider assigns a **public IP address**.
- The Load Balancer routes traffic to the **NodePort** service.

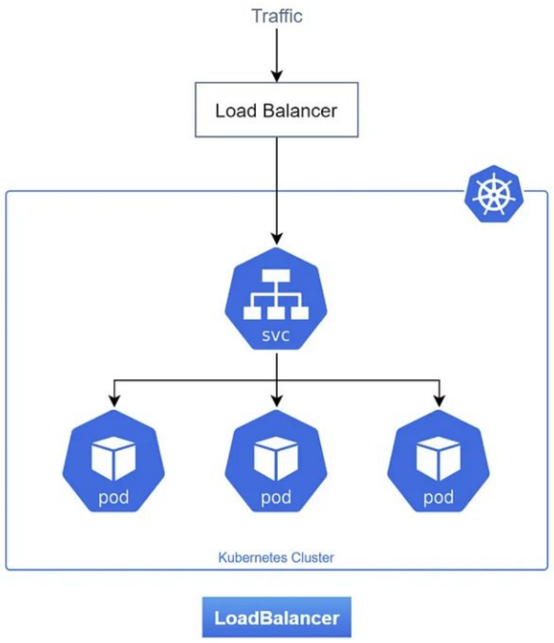
Use Case:

✅ **Production-ready** external access (e.g., public-facing web apps).

🔹 Users can access the app via a **public IP (assigned by the cloud provider).**

📌 **Limitations:**

- **Only works in cloud environments** (AWS, Azure, GCP).
- **Can be expensive** for high-traffic applications.



Ingress – Advanced HTTP Routing

While LoadBalancer services **work well for simple apps**, they **lack flexibility** (e.g., routing different URLs to different services).

🔹 **Ingress solves this** by providing an **HTTP(S) routing layer** in front of multiple services.

How it works?

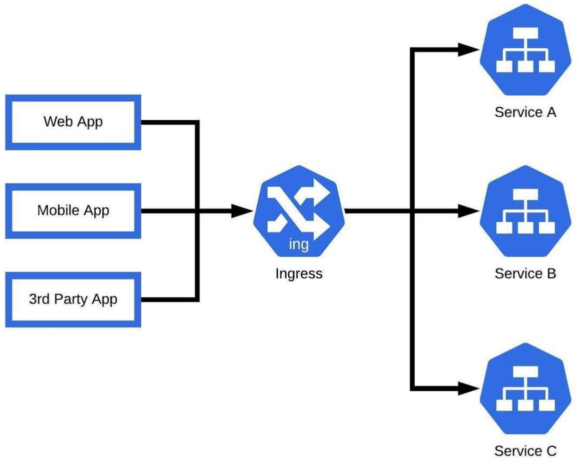
- Uses an **Ingress Controller** (e.g., NGINX, Traefik) to route requests based on **URLs or hostnames**.
- Enables **TLS termination (HTTPS)** for security.
- Supports **rewrite rules, redirects, and authentication**.

Use Case:

- ✅ Route multiple domains/subdomains to different services.
- ✅ Manage HTTPS with **TLS certificates**.

📌 **Why use Ingress?**

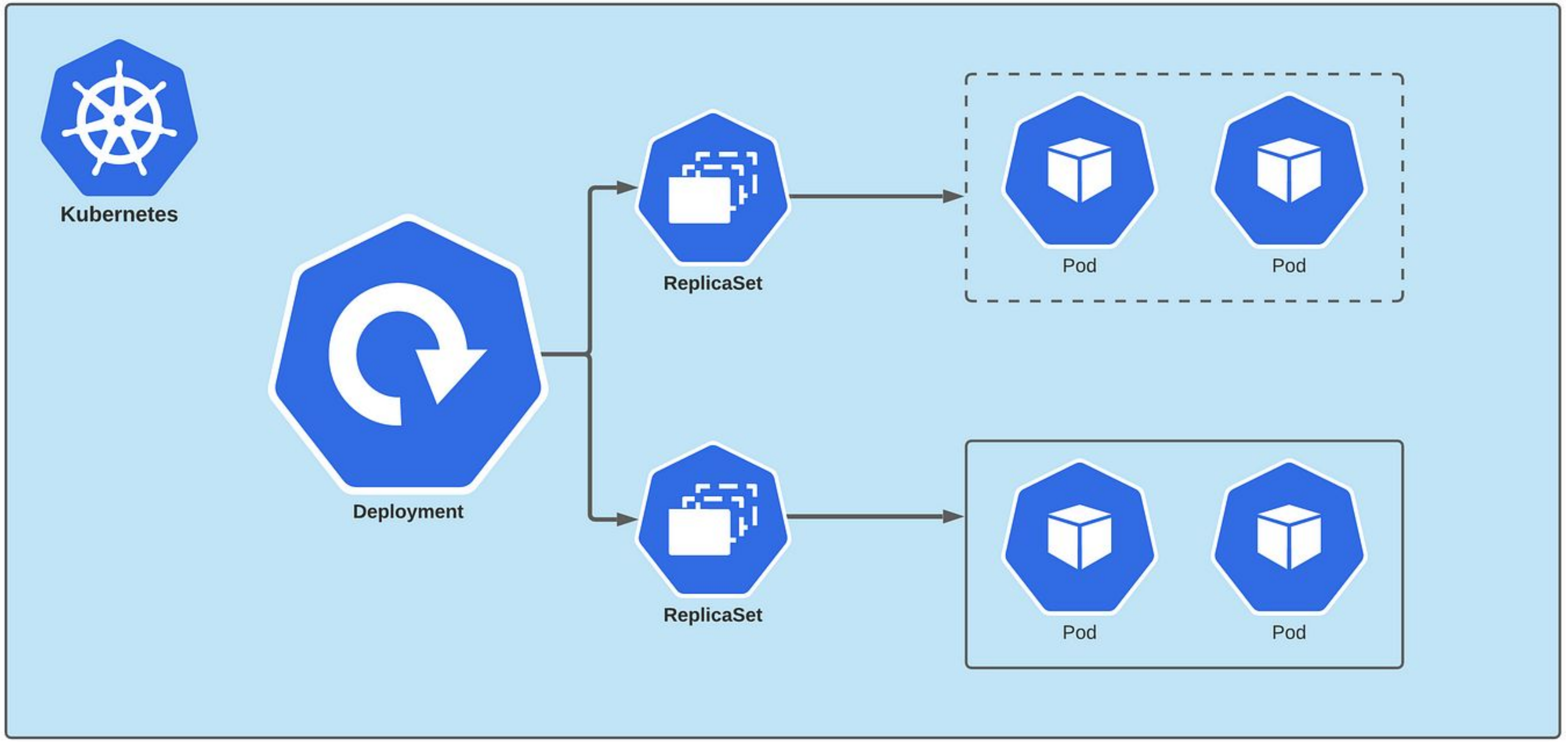
- ✅ **More powerful than LoadBalancer** (handles multiple services).
- ✅ **Optimized for HTTP/HTTPS traffic**.
- ✅ **Supports SSL/TLS** for secure connections.



📌 For production:

- Use **Ingress** for web apps (HTTP/HTTPS).
- Use **LoadBalancer** for direct cloud access.
- Use **ClusterIP** for microservices inside the cluster.

Deployments & Replicasets



Deployments & Replicasets

ReplicaSet:

Purpose:

- Ensures a specified number of identical Pods are running.

Core Functionality:

- Self-healing by replacing failed Pods.
- Maintains consistent Pod count.

Deployment:

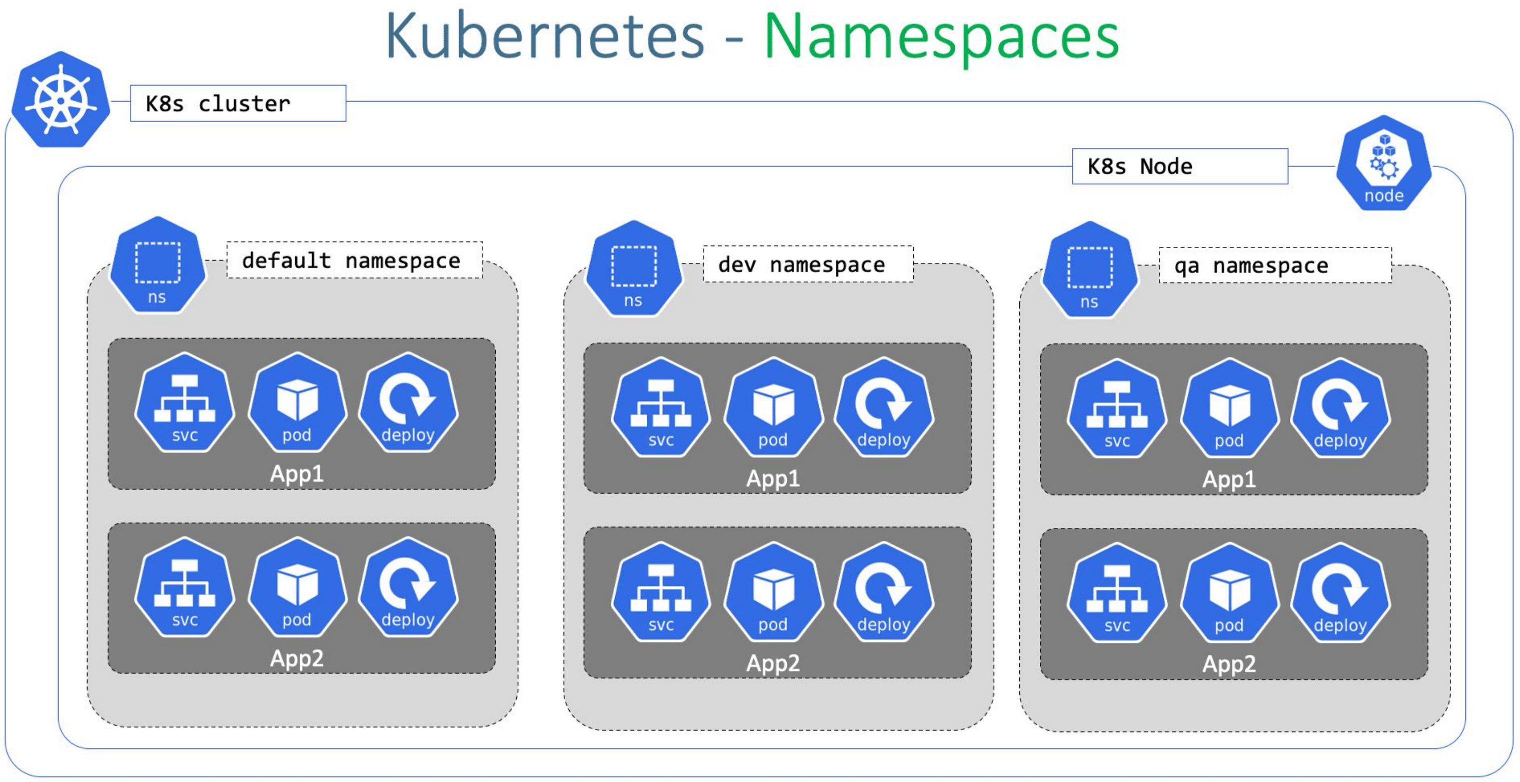
Purpose:

- Provides declarative, higher-level management for updating and scaling applications.

Core Functionality:

- Manages ReplicaSets under the hood.
- Enables **rolling updates** and **rollbacks**.
- Simplifies version control and lifecycle management of applications.

Namespaces – Organizing Resources



Namespaces – Organizing Resources

📂 Why Use Namespaces?

- In large clusters, we need to **isolate and organize resources**.
- Kubernetes provides multiple **logical clusters within a single physical cluster**.

🛠️ Key Features:

✅ Separate resources for **teams, environments (dev/prod), or projects**.

✅ Helps enforce **resource limits** (CPU, memory quotas).

✅ Built-in namespaces: default, kube-system, kube-public.

Difference Between a Namespace and a Cluster

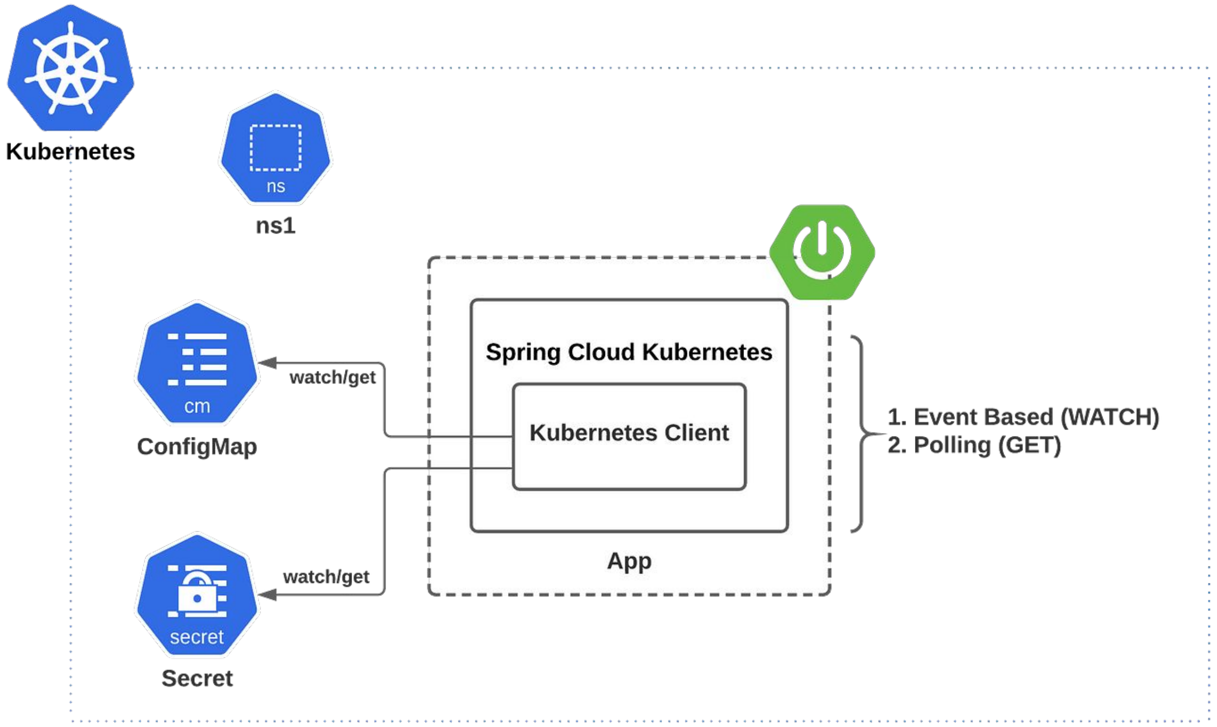
Cluster:

- A **Cluster** is the complete Kubernetes environment—a collection of physical or virtual machines (nodes) managed by the Kubernetes control plane.
- It encompasses the entire infrastructure where containerized applications run.
- The control plane schedules workloads, manages networking, storage, and overall health across all nodes.

Namespace:

- A **Namespace** is a **logical partition** within a cluster that helps to organize and isolate resources.
- It allows teams, projects, or environments (e.g., development, staging, production) to coexist on the same cluster without resource name collisions.
- Namespaces enable the application of resource quotas and policies at a granular level within the same cluster.

ConfigMaps & Secrets



ConfigMaps – Managing Configuration Settings

What is a ConfigMap?

- A Kubernetes object that stores **non-confidential configuration data**.
- Decouples configuration from container images.

Benefits

- **Centralized Management:** Update configuration without rebuilding images.
- **Flexible Consumption:** Can be used as environment variables, command-line arguments, or mounted files.
- **Easy Updates:** Changes to a ConfigMap can be reflected in running applications (depending on how they're consumed).

Secrets – Securely Storing Sensitive Data

What is a Secret?

- A Kubernetes object designed for **storing sensitive information** (e.g., passwords, tokens, keys).
- Data is usually **base64 encoded**.

Differences from ConfigMap

- **Security:** Secrets are handled with stricter access controls and are meant for confidential data.
- **Usage:** While ConfigMaps are for general configuration, Secrets are for information that **must remain protected**.
- **Management:** Can be encrypted at rest (depending on cluster configuration) for enhanced security.

Storage in Kubernetes

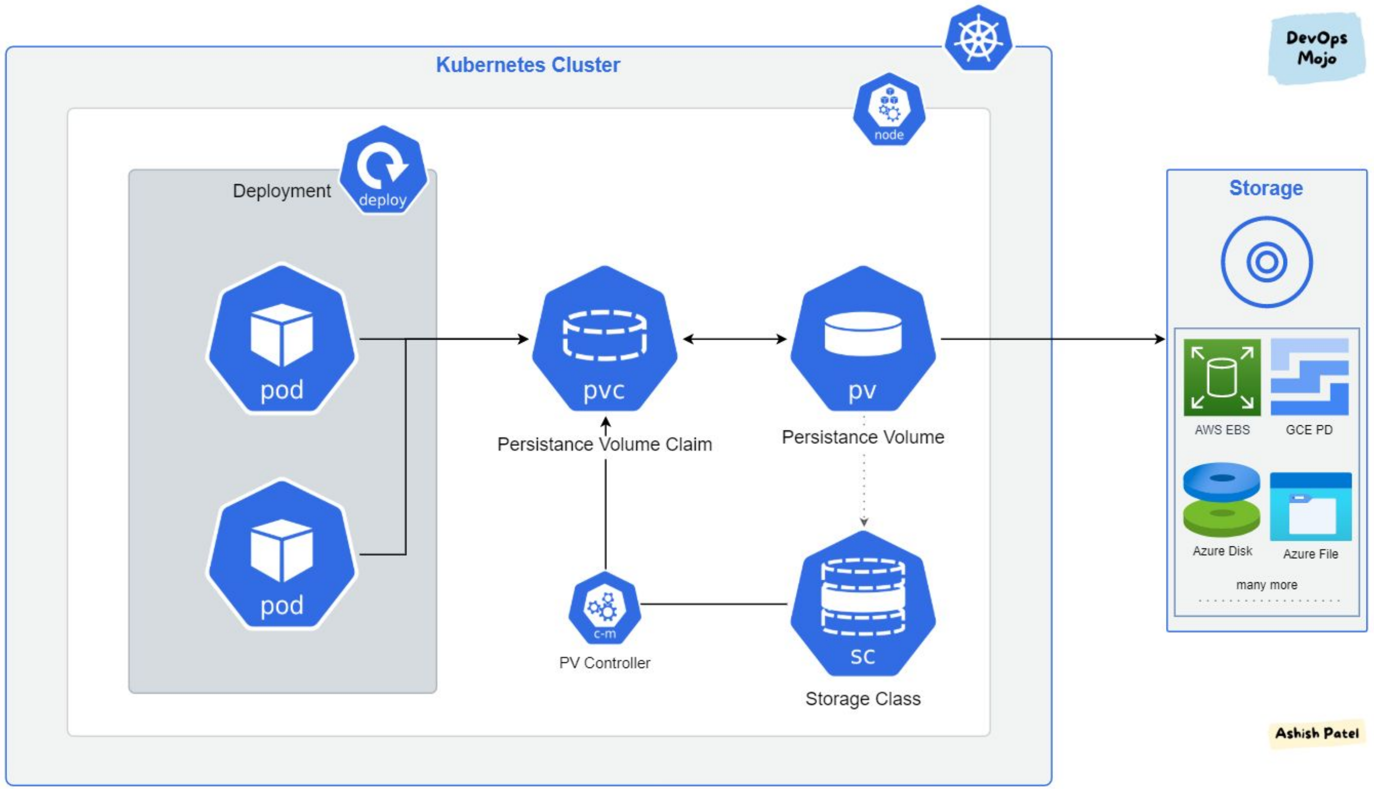
- Persistent Volumes (PVs) & Persistent Volume Claims (PVCs)
- Storage Classes
- StatefulSets for stateful applications

Storage in Kubernetes – Overview

📦 Why is storage important?

- Containers are **ephemeral** – data inside a Pod is lost when the Pod restarts.
- Applications often need **persistent storage** (databases, logs, etc.).
- Kubernetes provides **Persistent Volumes (PVs), Persistent Volume Claims (PVCs), Storage Classes, and StatefulSets** for managing storage efficiently.

Storage in Kubernetes – Overview



DevOps Mojo

Ashish Patel

Persistent Volumes (PV)

Persistent Volumes are a way to abstract and represent physical or networked storage resources in a cluster. They serve as the “backend” storage configuration in a Kubernetes cluster. PVs are crucial for the following reasons:

1. **Resource Abstraction**

   PVs abstract the underlying storage, making it easier to manage storage resources independently of the applications that use them.

2. **Resource Management**

   PVs enable administrators to allocate and manage storage resources, allowing for better utilization and optimization of storage hardware.

3. **Data Persistence**

   PVs ensure data persistence, even if pods or containers are recreated or rescheduled. This is vital for stateful applications and databases.

4. **Access Control**

PVs define access modes (e.g., ReadWriteOnce, ReadOnlyMany, ReadWriteMany) and security settings for how pods can access the storage, ensuring data integrity and security.

Persistent Volume Claims (PVC)

Persistent Volume Claims act as requests for storage by pods. They are used by developers to specify their storage requirements. Here’s why PVCs are crucial:

1. **Resource Request**

   PVCs allow developers to request storage resources without needing to know the underlying infrastructure details, making it easier to scale applications.

2. **Dynamic Provisioning**

   When configured with a Storage Class, PVCs can dynamically provision storage resources based on predefined policies, simplifying the provisioning process.

3. **Isolation**

   PVCs isolate storage-related concerns from application code, improving maintainability and portability of applications.

Storage Classes – Dynamic Provisioning

Storage Classes are an abstraction layer over the underlying storage infrastructure. They define the properties and behavior of PVs dynamically provisioned from them. Storage Classes are valuable for the following reasons:

1. **Dynamic Provisioning**

   They enable automatic provisioning of PVs with specified characteristics, such as storage type (e.g., SSD, HDD), access mode, and other parameters, simplifying storage management.

2. **Resource Optimization**

   Storage Classes facilitate the utilization of different storage resources based on application requirements, ensuring that workloads receive the right storage configuration.

3. **Scaling and Automation**

   They allow administrators to set up policies and rules for storage allocation, promoting scalability and automation in storage management.

StatefulSets – Managing Stateful Applications

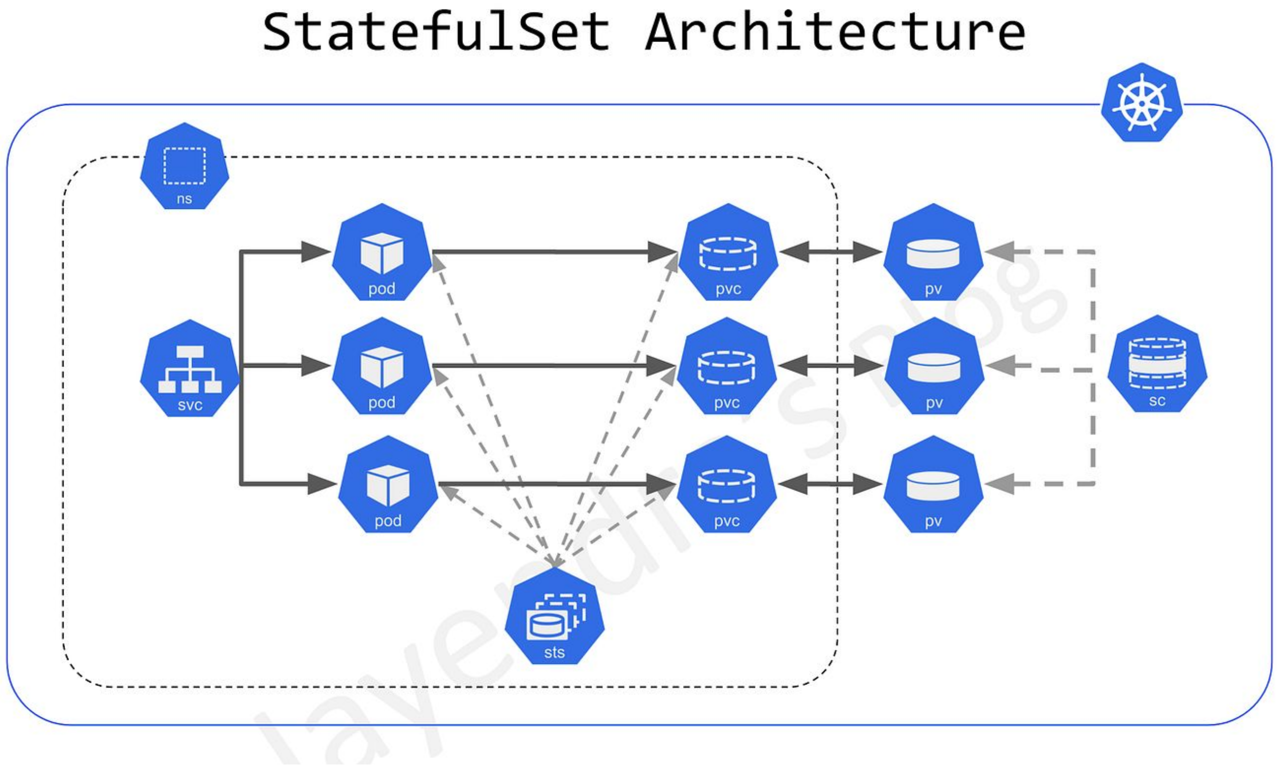
📌 **Why do we need StatefulSets?**

- **Deployments & ReplicaSets are best for stateless apps.**
- **Databases (MySQL, PostgreSQL, Redis, etc.) require stable identities & persistent storage.**

StatefulSets ensure that Pods:

- **Get stable network identities (Persistent Hostnames).**
- **Use stable storage (Persistent Volumes are not deleted).**
- **Scale in an ordered manner (pods are created/deleted one at a time).**

StatefulSets – Managing Stateful Applications

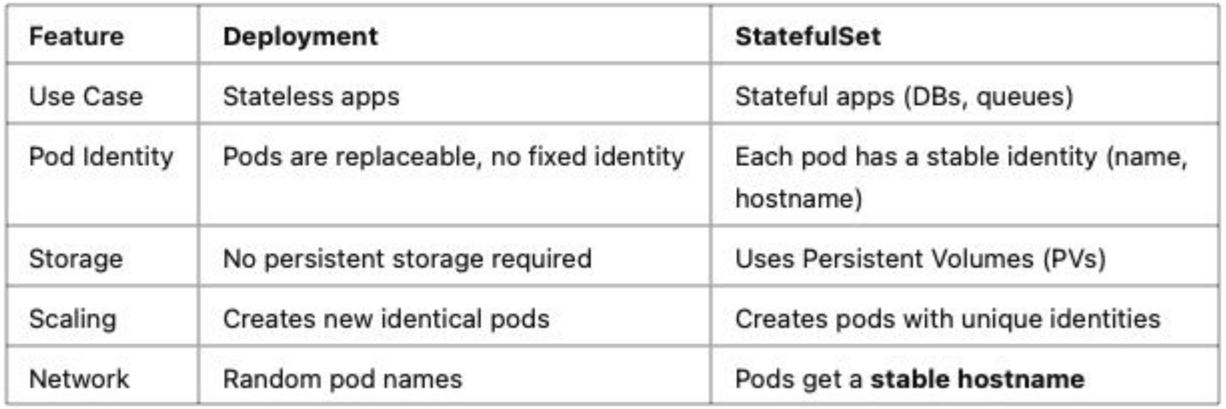


Stateful Applications in Kubernetes

1️⃣ Difference Between Stateless and Stateful Applications

- **Stateless Applications**: Do **not** maintain any persistent state between restarts (e.g., web servers, APIs).
- **Stateful Applications**: Require **persistent storage** (e.g., databases like MySQL, PostgreSQL, Kafka, Elasticsearch).

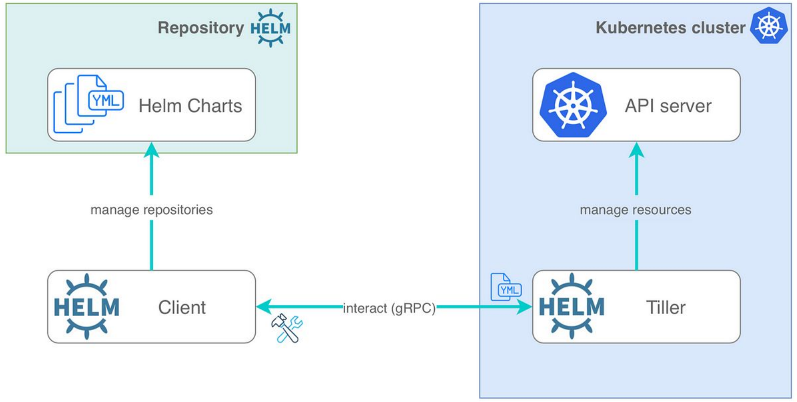
2️⃣ Deployment vs. StatefulSet



Feature	**Deployment**	**StatefulSet**
Use Case	Stateless apps	Stateful apps (DBs, queues)
Pod Identity	Pods are replaceable, no fixed identity	Each pod has a stable identity (name, hostname)
Storage	No persistent storage required	Uses Persistent Volumes (PVs)
Scaling	Creates new identical pods	Creates pods with unique identities
Network	Random pod names	Pods get a **stable hostname**

Helm & Kubernetes

Package Management



1️⃣ What is Helm?

- **"The Package Manager for Kubernetes"**
- Similar to apt (Linux) or npm (JavaScript) but for Kubernetes.
- Deploys applications using **Helm Charts** (templates for Kubernetes resources).

2️⃣ Benefits of Helm

✅ **Simplifies Deployments** (One command to deploy complex applications).

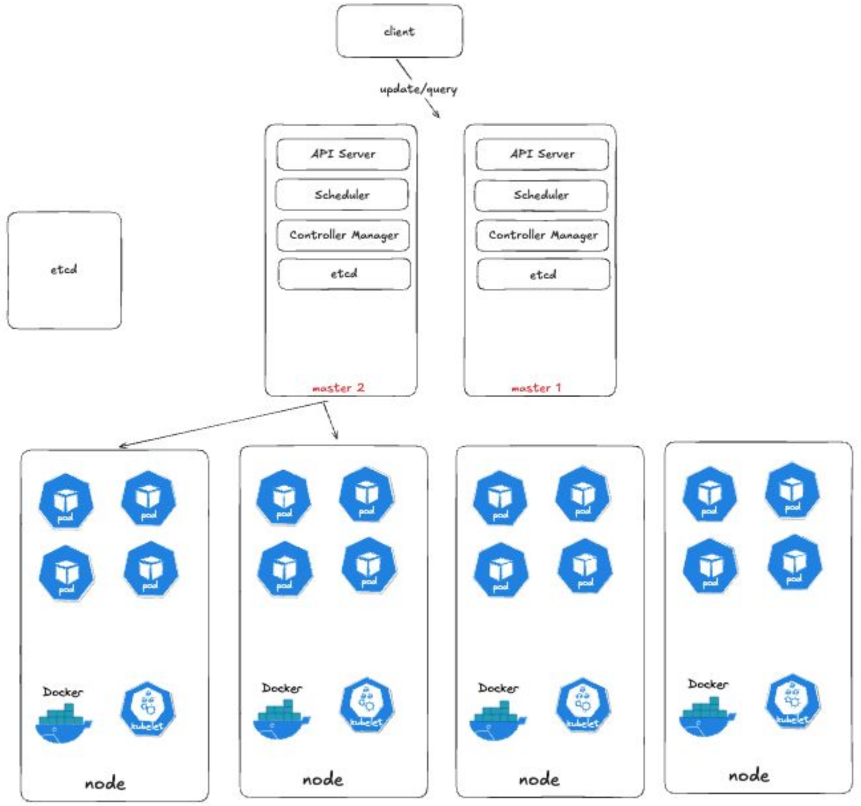
✅ **Version Control for Configurations** (Rollback to previous versions).

✅ **Parameterization** (Different environments can use the same template with different values).

Kubernetes Architecture

components of a cluster

- control plane (master)
    - multiple master for high availability
    - **api server, scheduler, controller manager, etcd**
- worker nodes
    - multiple, for workload distribution
    - **container runtime, kubelet, kube-proxy**



API Server: The Cluster Gateway

- primary entry point
- authentication and authorization
- validation and processing
- communication hub

How does API server handles requests?

1. authentication, authorization

2. admission control(modification, validation)

3. schema validation

4. persistance

kubectl and other control mechanisms

- kubectl: the CLI power tool
  - primary CLI for cluster management
  - supports declarative and imperative approaches
  - a lot of plugins (kubectx, kubens, stern, k9s…….)
- What can i do with kubectl?
  - resource management: create, get, describe, delete
  - cluster operations: logs, exec, port-forward
  - context management: multiple cluster access
- Can I interact with my cluster outside of kubectl?
  - dashboard: web-based UI
  - client libraries: official SDKs for various languages

Scheduler: the workload placer

- Determines optimal pod placement
- How does it decide?
  - filtering
  - scoring
- Which factors are considered?
  - resource requirements and availability
  - hardware/software constraints
  - node utilization
  - affinity- anti affinity spec
  - deadlines
  - Can I customize the factors? yes, by rules and policies

Controller Manager: the state keeper

- ensures **actual state** matches **desired state**
- implements control loops for cluster resources
- handles failures and recoveries

Some examples:

- Deployment controller
- ReplicaSet controller
- Node controller
- Endpoint controller

The Concept of Desired State in Kubernetes

Desired State Defined:

- The **desired state** is a declarative description (typically in YAML or JSON) of how you want your cluster and its workloads to behave.
- It includes specifications like the number of replicas, container images, configurations, and policies.

How Kubernetes Achieves It:

- The **Kubernetes control plane** continuously monitors the cluster to compare the **actual state** with the **desired state**.
- **Controllers** (such as Deployment or ReplicaSet controllers) are responsible for reconciling differences:
  - If a pod crashes or deviates from the defined state, the controller automatically creates, replaces, or scales resources until the actual state matches the desired state.
  - This **self-healing** mechanism ensures reliability and resilience across the cluster.

etcd: the data store

- more than simple storage
    - distributed kv store
    - source of truth for cluster state
    - supports cluster configurations
- some critical features
    - strong consistency guarantees
    - automatic failover
    - backup and restore capabilities
- best practices:
    - external etcd cluster instead of stacked etcd
    - regular backups
    - high availability setup
    - performance monitoring

Kubelet: the node commander

- pod lifecycle manager
  - makes sure containers are running in pods
  - handles volume mounts
  - reports status
- container health management
  - monitoring and restarting containers
  - some probe checks: liveness, readiness, startup
- resource management
  - resource limits
  - local storage
  - container logging

container runtime

- image/registry management
- container lifecycle
- resource isolation
- security
  - image verification
  - runtime isolation
  - resource constraints
- containerd
- runc

kube-proxy: the network orchestrator

- service abstraction
- load balancing
- network policy enforcement
- zero-downtime deployments

container networking

iptables rules control traffic between c-c, h-c and c-e

veth (virtual ethernet pairs)

NAT allows containers with private IP addresses to communicate with external networks

port mapping exposes container services through iptables

overlay networks, bridge network