Kubernetes provides you with a framework to run distributed systems resiliently

K8s > Docker (scalability)

Kubectl -> request -> api server

Nody control plane, worker node

Obraz zawierający tekst, zrzut ekranu, pismo odręczne, Czcionka

Opis wygenerowany automatycznie

Obraz zawierający tekst, pismo odręczne, zrzut ekranu, design

Opis wygenerowany automatycznie

Etcd – kv database

Obraz zawierający tekst, pismo odręczne, Czcionka, tablica suchościeralna biała

Opis wygenerowany automatycznie

Obraz zawierający tekst, Czcionka, pismo odręczne, linia

Opis wygenerowany automatycznie

Kubelet – pod management, communicates with api

Obraz zawierający tekst, Czcionka, zrzut ekranu, linia

Opis wygenerowany automatycznie

**1. Container Runtime Interface (CRI)**

The **CRI** defines how Kubernetes communicates with the underlying container runtime to manage containers. It standardizes operations like pulling images, starting/stopping containers, and collecting logs.

**Examples of CRI-Compatible Container Runtimes:**

* **containerd**: A lightweight runtime optimized for Kubernetes.
* **CRI-O**: A runtime designed specifically to run Open Container Initiative (OCI) containers.
* **Docker**: Previously used by Kubernetes, but Kubernetes now uses CRI-compatible runtimes directly.

**Why CRI?**

* To allow Kubernetes to work with a variety of container runtimes without being tightly coupled to one.

**2. Container Network Interface (CNI)**

The **CNI** defines how Kubernetes manages networking for Pods. It specifies how network configurations (like IP allocation, routing, etc.) are applied to Pods and how network plugins are implemented.

**Popular CNI Plugins:**

1. **Flannel**:
   * A simple and lightweight CNI implementation.
   * Provides a flat layer-3 network between Kubernetes nodes.
   * Commonly used for basic networking needs.
2. **Calico**:
   * A more advanced CNI plugin.
   * Supports networking and network policies using layer-3 and layer-4 capabilities.
   * Often chosen for its robust security features and scalability.
3. **Cilium**:
   * A CNI focused on security and observability.
   * Uses eBPF (extended Berkeley Packet Filter) for efficient packet processing.
   * Allows for advanced features like layer-7 (application-level) policies.

**Why CNI?**

* To abstract the networking layer in Kubernetes, enabling integration with various networking solutions.

**3. Container Storage Interface (CSI)**

The **CSI** standardizes how Kubernetes interacts with storage systems. It allows dynamic provisioning, management, and mounting of storage volumes for Pods.

**Popular CSI Implementations:**

1. **Longhorn**:
   * A lightweight, Kubernetes-native distributed storage system.
   * Provides high availability, snapshots, and backups.
   * Easy to set up and manage within Kubernetes.
2. **OpenEBS**:
   * A Kubernetes-native storage solution.
   * Focuses on providing containerized storage for stateful applications.
   * Offers features like dynamic volume provisioning, snapshots, and replication.

**Why CSI?**

* To abstract the storage layer, enabling Kubernetes to support multiple storage backends without being tied to any specific technology.

**Comparison and Role in Kubernetes:**

| **Interface** | **Purpose** | **Examples** |
| --- | --- | --- |
| **CRI** | Manages container runtimes. | containerd, CRI-O, Docker |
| **CNI** | Manages networking for Pods. | Flannel, Calico, Cilium |
| **CSI** | Manages storage provisioning. | Longhorn, OpenEBS |

**How They Work Together:**

* **CRI** handles container creation and management.
* **CNI** ensures that Pods have network connectivity and policies.
* **CSI** provides persistent storage for stateful workloads.

Together, these interfaces make Kubernetes flexible and capable of running on diverse underlying systems.

Obraz zawierający tekst, zrzut ekranu, Czcionka, design

Opis wygenerowany automatycznie

**What is kube-proxy?**

**kube-proxy** is a network component of Kubernetes that runs on every node in a cluster. Its primary role is to manage network communication for Pods, particularly routing and load balancing for Kubernetes Services. kube-proxy ensures that requests to Services are properly routed to the appropriate Pods, even as Pods are added, removed, or relocated across the cluster.

**Key Functions of kube-proxy**

1. **Service Networking:**
   * kube-proxy ensures that Kubernetes Services (e.g., ClusterIP, NodePort, LoadBalancer) are reachable from within and outside the cluster.
2. **Connection Routing:**
   * It handles the routing of traffic to one of the Pods backing a Service based on the Service's configuration.
   * Load-balances traffic across all healthy Pods.
3. **Service Discovery:**
   * Uses Kubernetes Endpoints to maintain a list of Pods that back each Service.
   * Updates routing rules dynamically as the state of Pods and Endpoints changes.
4. **Network Protocol Handling:**
   * Supports TCP, UDP, and SCTP protocols for network communication.

**How kube-proxy Works**

When a Service is created in Kubernetes:

1. kube-proxy watches for changes to Services and Endpoints by interacting with the Kubernetes API server.
2. It configures the node’s networking to route traffic to the correct Pod(s) based on the Service definition.
3. Depending on the operating mode, kube-proxy sets up:
   * **IPTables**: Network rules using the iptables subsystem.
   * **IPVS (IP Virtual Server)**: More efficient, scalable, and feature-rich routing mechanism compared to iptables.
   * **Userspace**: An older, less efficient mode where kube-proxy proxies traffic itself.

**Operating Modes of kube-proxy**

1. **Userspace Mode:**
   * kube-proxy listens for traffic on each Service’s ClusterIP and forwards it to one of the Pods backing the Service.
   * Less efficient due to additional hops and latency.
2. **IPTables Mode (default):**
   * Uses Linux iptables rules to redirect traffic to Pods.
   * Scales better and is faster than Userspace mode.
3. **IPVS Mode:**
   * Uses IP Virtual Server (IPVS), built on Linux Netfilter.
   * Offers better performance and load-balancing algorithms.
   * Ideal for large-scale clusters.

**Key Concepts Related to kube-proxy**

1. **Service Types:**
   * **ClusterIP**: Internal traffic routing within the cluster.
   * **NodePort**: Exposes Services on a specific port of every node.
   * **LoadBalancer**: Integrates with cloud providers for external load balancing.
2. **Endpoints:**
   * kube-proxy uses the list of Endpoints (Pod IPs) backing a Service to route traffic.
3. **Selectors:**
   * kube-proxy identifies which Pods belong to a Service based on labels and selectors defined in the Service.

**Limitations of kube-proxy**

1. kube-proxy cannot enforce security or network policies; these are typically handled by CNI plugins like **Calico** or **Cilium**.
2. It relies on the underlying node’s networking capabilities (iptables, IPVS, etc.).
3. In very large clusters, kube-proxy may encounter performance bottlenecks if not configured optimally (e.g., iptables rules scaling).

Obraz zawierający tekst, pismo odręczne, zrzut ekranu, Czcionka

Opis wygenerowany automatyczniereplicaset, deployment, job,statefulset, daemonset – k8s objects

The **reconciliation loop** is a fundamental concept in Kubernetes and declarative infrastructure management. It refers to the process by which Kubernetes ensures that the **current state** of the system matches the **desired state** defined by the user. The reconciliation loop is a continuous feedback-driven mechanism that runs in Kubernetes controllers.

**Key Components of the Reconciliation Loop**

1. **Desired State:**
   * The user specifies what they want the system to look like (e.g., using YAML/JSON manifests for Pods, Deployments, Services, etc.).
   * This state is stored in the Kubernetes **etcd** database via the API server.
2. **Current State:**
   * The actual state of resources running in the cluster (e.g., what Pods exist, their health, the status of nodes, etc.).
3. **Controller:**
   * Each resource type in Kubernetes (e.g., Pods, Deployments, Nodes) has a corresponding **controller**.
   * The controller compares the desired state (from the API server) with the current state and takes actions to bring the system into alignment.
4. **Actions:**
   * The controller performs necessary actions (e.g., starting a new Pod, rescheduling a Pod, scaling replicas) to reconcile the two states.

**What is CCM (Cloud Controller Manager) in Kubernetes?**

The **Cloud Controller Manager (CCM)** is a component in Kubernetes responsible for integrating the Kubernetes cluster with the underlying cloud provider’s infrastructure. It allows Kubernetes to interact with cloud provider-specific APIs to manage resources like load balancers, storage volumes, and node instances.

Introduced as part of Kubernetes' effort to decouple cloud-specific logic from the core Kubernetes codebase, CCM enables Kubernetes to support multiple cloud providers through modular extensions.

**Key Functions of CCM**

1. **Node Management:**
   * Manages cloud-specific details about nodes.
   * Detects and updates the status of cloud instances (e.g., marking nodes as "NotReady" if they are deleted in the cloud provider).
2. **Route Management:**
   * Configures routes in the cloud provider to ensure Pod-to-Pod communication across nodes.
   * Commonly used in environments where Kubernetes cannot assume full control of network routing (e.g., GCE, AWS).
3. **Service Management (Load Balancers):**
   * Creates and manages cloud load balancers for Kubernetes Services of type LoadBalancer.
   * Automatically configures external-facing IPs and backend endpoints.
4. **Persistent Volume Management:**
   * Dynamically provisions and attaches cloud storage volumes (e.g., EBS on AWS, Persistent Disks on GCP) to Kubernetes Pods using the Persistent Volume (PV) and Persistent Volume Claim (PVC) system.

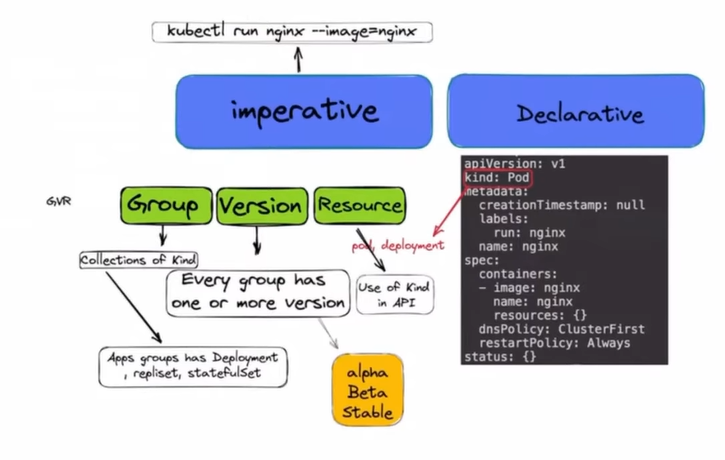
**Deployment of CCM**

The CCM can be deployed:

* **Separately as a standalone component.**
* **Bundled within the control plane**, though this is less common in newer setups.

YAML

$ for env variables



**Comparison: Imperative vs Declarative**

| **Feature** | **Imperative** | **Declarative** |
| --- | --- | --- |
| **Method** | Direct commands | Configuration files |
| **Use Case** | Ad-hoc, quick tasks | Long-term, automated, and repeatable |
| **Ease of Use** | Simple for one-off tasks | Requires setup but better for automation |
| **Persistence** | No record of intent | Persistent and version-controlled |
| **Reconciliation** | No reconciliation | Continuous reconciliation |
| **Examples** | kubectl create, kubectl delete | kubectl apply |

**When to Use Which Approach?**

1. **Imperative:**
   * Quick fixes or small-scale tasks.
   * Experimentation or development scenarios.
   * Temporary actions that don’t need to be saved.
2. **Declarative:**
   * Production environments where consistency is critical.
   * Long-term management of resources.
   * Integration with CI/CD pipelines or GitOps workflows.

Both approaches have their place in Kubernetes management. While imperative commands are useful for quick, on-the-fly tasks, the declarative approach is more robust and better suited for managing complex environments over time. Most production-grade workflows lean heavily on the declarative method for its scalability and repeatability.

kubectl get nodes

kubectl get pods -A

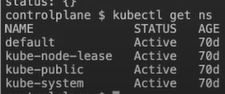
kubectl get pods -A -owide (too see node where the pod is running)

kubectl run nginx --image=nginx

kubectl run demo --image=nginx --dry-run=client -oyaml

Namespaces

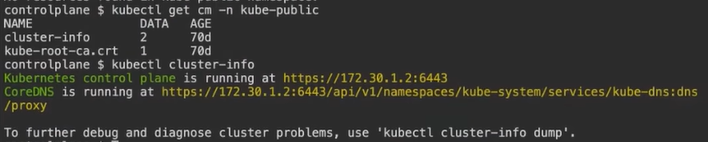
kubectl get ns



kubectl get pods -n kube-system reserved for the system

kubectl get cm -n kube-public

kubectl cluster-info



kubectl get lease -n kube-node-lease



Obraz zawierający tekst, zrzut ekranu, Czcionka, diagram

Opis wygenerowany automatycznie

Namespaces makes environment isolated, segregated, versioned

kubectl api-resources - list of all apis

kubectl api-resources –namespaced=false i.e get nodes, ged podes

kubectl create deploy pod-name --image=nginx

kubectl create deploy pod-name --image=nginx -n dev

kubectl get ns

kubectl describe ns dev

kubectl create ns demo --dry -run=client -oyaml

Obraz zawierający tekst, zrzut ekranu, Czcionka

Opis wygenerowany automatycznie

Kubectl delete ns testing

kubectl config set-context --current --namespace=dev

Labels (key value in metadata section) and selectors

Obraz zawierający tekst, zrzut ekranu, menu, Czcionka

Opis wygenerowany automatycznie

Obraz zawierający tekst, Czcionka, linia, pismo odręczne

Opis wygenerowany automatycznie

kubectl run nginx --image=nginx

kubectl create deploy nginx --image=nginx

kubectl label pod nginx app=demo

kubectl get pods -l run=nginx

kubectl get pods -l 'app in (demo,nginx)'

kubectl describe pod

Obraz zawierający tekst, zrzut ekranu

Opis wygenerowany automatycznie

kubectl logs -f pod name

kubectl get pods

kubectl delete pod

kubectl exec -it podname --sh

kubectl get pods -owide

kubectl delete pod --force

pod lifecycle:  
Obraz zawierający tekst, Czcionka, pismo odręczne, linia

Opis wygenerowany automatycznie

Init container  
Obraz zawierający tekst, zrzut ekranu, Czcionka, design

Opis wygenerowany automatycznie

Vi init

kubectl apply -f init

kubectl exec -it init-demo1 --sh

curl localhost

pod will always get single IP, container will share ip (sidecar, envoy, proxy)

kubectl apply -f multiple-init-container.yaml

kubectl logs -f multiple-init-container.yaml -c nginx-container

kubectl exec -it multi-container -c nginx-container -- curl localhost

Probes

Obraz zawierający tekst, zrzut ekranu, Czcionka, oprogramowanie

Opis wygenerowany automatycznie

kubectl apply -f https://github.com/kubernetes-sigs/metrics-server/releases/latest/download/components.yaml add- --kubelet-insecure-tls

Obraz zawierający tekst, zrzut ekranu, Czcionka

Opis wygenerowany automatycznie

Wget https://github.com/kubernetes-sigs/metrics-server/releases/latest/download/components.yaml

Kubectl apply -f components.yaml.1

Kubectl create -f rr.yaml

Kubectl get pods

Obraz zawierający tekst, Czcionka, zrzut ekranu, czarne

Opis wygenerowany automatycznie

Kubectl top pods

Deployments

**What Are Kubernetes Deployments?**

A **Deployment** in Kubernetes is a higher-level resource that manages the lifecycle of **Pods** and ensures they run reliably and consistently in the cluster. Deployments allow you to describe the desired state of your application and ensure that Kubernetes automatically maintains it.

**Key Features of Deployments**

1. **Declarative Management:**
   * Define your application's desired state (e.g., number of replicas, container images) in a YAML/JSON manifest.
2. **Scaling:**
   * Easily scale applications up or down by adjusting the number of replicas.
3. **Rolling Updates:**
   * Automatically roll out updates to Pods incrementally to avoid downtime.
4. **Rollback Support:**
   * Roll back to a previous state if an update fails or causes issues.
5. **Self-Healing:**
   * Automatically replaces failed Pods to maintain the desired state.

**How Deployments Work**

1. **Deployment Specification:**
   * You define a Deployment in a YAML or JSON file.
   * It includes details like the number of replicas, container images, labels, and selectors.
2. **ReplicaSet Management:**
   * The Deployment creates and manages a **ReplicaSet**, which ensures the desired number of Pods are running.
3. **Pod Management:**
   * The ReplicaSet creates and maintains the Pods.
4. **Continuous Reconciliation:**
   * Kubernetes continuously monitors the Deployment and ensures the current state matches the desired state.

**Deployment vs. ReplicaSet vs. Pod**

| **Resource** | **Purpose** |
| --- | --- |
| **Pod** | Smallest deployable unit in Kubernetes. Runs containers. |
| **ReplicaSet** | Ensures a specified number of identical Pods are running. |
| **Deployment** | Manages ReplicaSets, enabling declarative updates, scaling, and rollbacks. |

Obraz zawierający tekst, Czcionka, zrzut ekranu, menu

Opis wygenerowany automatycznie

In Kubernetes, **deployment strategies** determine how changes to a Deployment are rolled out to the cluster. The choice of strategy impacts how updates affect availability, downtime, and user experience. Kubernetes primarily supports two deployment strategies: **Rolling Update** and **Recreate**, but you can implement custom strategies for advanced scenarios.

**1. Rolling Update (Default)**

**How It Works:**

* Updates Pods incrementally by replacing old Pods with new ones.
* Ensures a mix of old and new Pods during the update process, avoiding downtime.

**Key Features:**

* **Zero Downtime:** Ensures some Pods remain available during the update.
* **Configurable Parallelism:**
  + You can control how many Pods are updated simultaneously (maxSurge and maxUnavailable).

**Configuration Example:**

yaml

Skopiuj kod

apiVersion: apps/v1

kind: Deployment

metadata:

name: my-app

spec:

replicas: 5

strategy:

type: RollingUpdate

rollingUpdate:

maxSurge: 2 # Number of additional Pods to create during the update

maxUnavailable: 1 # Number of Pods that can be unavailable during the update

template:

spec:

containers:

- name: my-app

image: my-app:v2

**When to Use:**

* Ideal for most scenarios where high availability is critical.
* Suitable for stateless applications.

**2. Recreate**

**How It Works:**

* Deletes all existing Pods of the Deployment before creating new ones.
* Ensures no overlap between old and new versions.

**Key Features:**

* **Simpler Update Process:** No need for parallel updates or additional Pods.
* **Downtime:** All Pods are terminated before new ones start.

**Configuration Example:**

yaml

Skopiuj kod

apiVersion: apps/v1

kind: Deployment

metadata:

name: my-app

spec:

replicas: 5

strategy:

type: Recreate

template:

spec:

containers:

- name: my-app

image: my-app:v2

**When to Use:**

* Suitable for applications that cannot run multiple versions simultaneously (e.g., apps that rely on a shared database schema).
* Works for stateful or tightly coupled applications.

**3. Custom Deployment Strategies**

For advanced use cases, you can implement custom deployment strategies using external tools like **Argo Rollouts**, **Flagger**, or by scripting your own sequence of events.

**Examples of Custom Strategies:**

1. **Blue-Green Deployment:**
   * Deploys the new version (blue) alongside the existing version (green).
   * Traffic is switched to the new version only after validation.
   * Ensures zero downtime with full rollback capability.
2. **Canary Deployment:**
   * Gradually shifts traffic from the old version to the new version.
   * Monitors the performance and stability of the new version before increasing traffic.
3. **A/B Testing:**
   * Similar to canary, but focuses on splitting traffic between different versions for testing specific features or user segments.
4. **Shadow Deployment:**
   * Sends a copy of production traffic to the new version without affecting real users.
   * Used to test the new version under real-world conditions without serving traffic.

**Comparison of Deployment Strategies**

| **Strategy** | **Downtime** | **Parallel Versions** | **Use Cases** |
| --- | --- | --- | --- |
| **Rolling Update** | None | Yes | Stateless apps, high availability required. |
| **Recreate** | Yes | No | Stateful apps, schema-dependent apps. |
| **Blue-Green** | None | Yes | Critical apps with full rollback capability. |
| **Canary** | Minimal | Yes | Gradual rollouts with monitoring/validation. |
| **Shadow** | None | Yes | Testing new versions without affecting users. |

**Best Practices for Choosing Deployment Strategies**

1. **Understand Application Requirements:**
   * Stateless vs. stateful, database dependencies, or specific version constraints.
2. **Monitor Metrics:**
   * Use monitoring tools like Prometheus, Grafana, or Kubernetes Events to observe updates.
3. **Use Probes:**
   * Configure **readiness** and **liveness** probes to ensure only healthy Pods serve traffic.
4. **Plan Rollbacks:**
   * Ensure rollback mechanisms are in place for failures (e.g., kubectl rollout undo).
5. **Automate Testing:**
   * Use CI/CD pipelines with integration and performance tests during deployments.

Choosing the right deployment strategy depends on the application, business requirements, and tolerance for downtime or risk. Kubernetes’ built-in strategies, combined with tools for custom strategies, provide flexibility to meet various needs.

Kubectl create deploy nginx --image=nginx

Kubectl get rs

Kubectl get pods

Obraz zawierający tekst, zrzut ekranu, menu, Czcionka

Opis wygenerowany automatycznie

Name-replicasetidwithhash-deploymenthash

Kubectl scale deploy nginx --replicas 5

Kubectl edit deploy nazwa -> make changes on a fly

Kubectl set image deployment/nginx nginx=nginx:1.15.2 –record

Kubectl describe deploy nginx

Obraz zawierający tekst, zrzut ekranu, menu, Czcionka

Opis wygenerowany automatycznie

Kubectl rollout status deploy nginx  
Obraz zawierający tekst, Oprogramowanie multimedialne, Czcionka, zrzut ekranu

Opis wygenerowany automatycznie

kubectl rollout history deployment nginx

kubectl rollout undo deployment nginx --to-revision 2

kubectl delete deploy nginx --force

**STATEFULSETS**

Obraz zawierający tekst, zrzut ekranu, Czcionka, linia

Opis wygenerowany automatycznie

**What Are StatefulSets in Kubernetes?**

**A StatefulSet is a Kubernetes resource designed to manage stateful applications that require persistent data storage and stable network identities. Unlike Deployments, which are suited for stateless applications, StatefulSets are specifically designed to handle workloads where maintaining the identity and state of each Pod is critical.**

**Key Features of StatefulSets**

1. **Stable Network Identity:**
   * **Each Pod gets a unique, predictable hostname based on its order (e.g., my-app-0, my-app-1).**
   * **Hostnames persist across Pod rescheduling.**
2. **Persistent Storage:**
   * **Each Pod can be assigned a unique, persistent storage volume.**
   * **Volumes are retained even if the Pod is deleted.**
3. **Ordered Deployment and Scaling:**
   * **Pods are created, updated, and deleted in a specific order.**
   * **Ensures predictable startup and shutdown sequences.**
4. **Graceful Updates:**
   * **Updates are applied sequentially, ensuring application stability during changes.**

**When to Use StatefulSets**

**StatefulSets are ideal for applications that:**

* **Require persistent data (e.g., databases, message queues).**
* **Depend on stable network identities (e.g., Zookeeper, Kafka, Elasticsearch).**
* **Need ordered, sequential deployment and scaling.**

**Components of a StatefulSet**

1. **Pods:**
   * **Managed Pods have stable identities and persistent storage.**
   * **Pods are named statefulset-name-ordinal (e.g., my-app-0).**
2. **Headless Service:**
   * **A Headless Service (with no ClusterIP) is used to enable direct DNS resolution for each Pod.**
3. **PersistentVolumeClaim (PVC):**
   * **Ensures each Pod has its own persistent storage.**

**Key Behaviors of StatefulSets**

1. **Pod Creation:**
   * **Pods are created sequentially (my-app-0, my-app-1, my-app-2).**
2. **Pod Deletion:**
   * **Pods are deleted in reverse order (my-app-2 → my-app-1 → my-app-0).**
3. **Scaling:**
   * **New Pods are added in order, and extra Pods are deleted in reverse order.**
4. **Storage Retention:**
   * **Even if a Pod is deleted, its PersistentVolume remains intact and is reattached to the new Pod with the same ordinal.**

**Differences Between StatefulSets and Deployments**

| **Feature** | **StatefulSet** | **Deployment** |
| --- | --- | --- |
| **Use Case** | **Stateful applications** | **Stateless applications** |
| **Pod Identity** | **Stable, predictable (e.g., my-app-0)** | **Dynamic (e.g., random Pod name)** |
| **Storage** | **Unique and persistent per Pod** | **Shared or ephemeral** |
| **Scaling/Updates** | **Ordered, sequential** | **Parallel** |
| **Headless Service** | **Required** | **Optional** |

**How StatefulSets Handle Persistent Data**

1. **PersistentVolumeClaim Templates:**
   * **StatefulSets automatically create PersistentVolumeClaims (PVCs) for each Pod.**
   * **Each Pod gets its own unique volume.**
2. **Volume Reattachment:**
   * **When a Pod is rescheduled, it retains its volume, ensuring no data loss.**
3. **Use Case:**
   * **For databases like MySQL, Cassandra, or Redis, where consistent storage is crucial.**

**Best Practices for StatefulSets**

1. **Use Headless Services:**
   * **Ensure proper DNS resolution for Pods.**
2. **Configure Persistent Storage:**
   * **Define volumeClaimTemplates for unique, persistent storage.**
3. **Monitor Resource Usage:**
   * **Set resource requests and limits to ensure Pods are stable.**
4. **Understand the Ordered Nature:**
   * **Be aware that scaling and updates are sequential and may take time.**