Cloud & Automation Class

Kubernetes for Beginners · Introduction

Mouna Rouini · Mariem Cherif · Skander Soltane · Adam Lahbib

Feb. 15, 2023 @ INSAT

Contents

1 Introduction

In this document, we will introduce you to the Kubernetes ecosystem and its main features. We will also explain the architecture of a Kubernetes cluster and its components. We will also explain the main concepts of Kubernetes and how to deploy a simple application on a Kubernetes cluster.

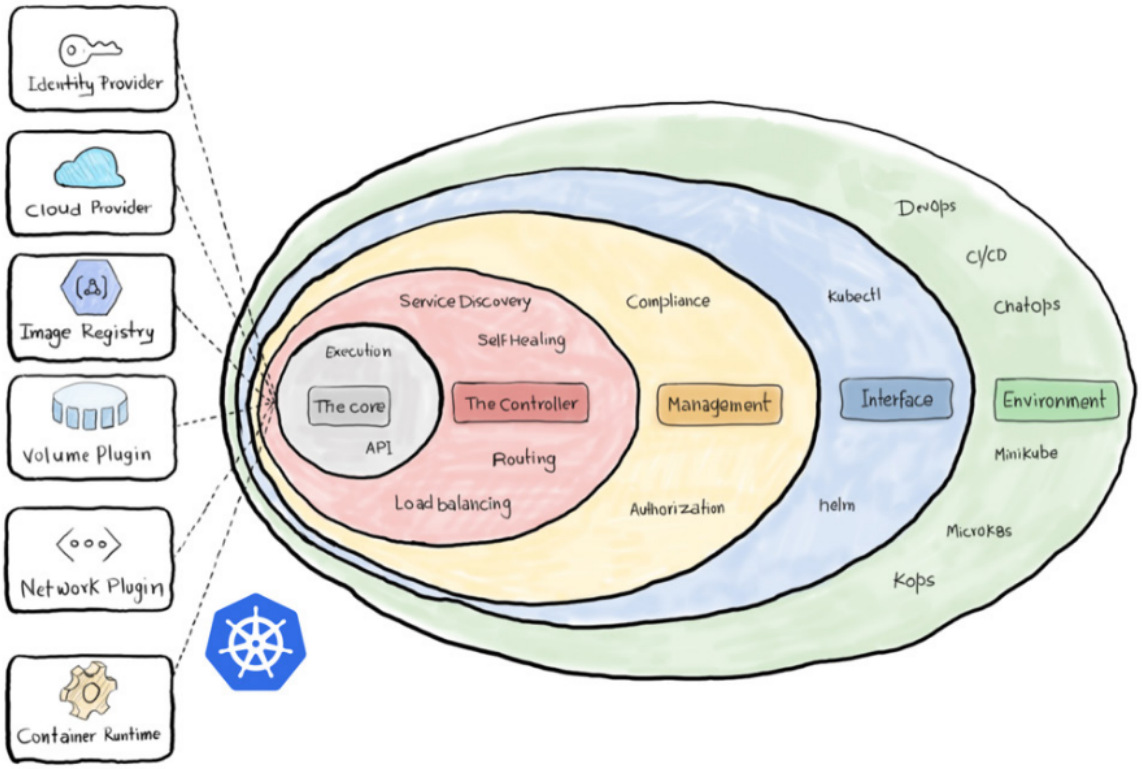
2 K8s or Kubernetes

Kubernetes is an open-source system for automating deployment, scaling, and management of containerized applications. It groups containers that make up an application into logical units for easy management and discovery. Kubernetes builds upon 15 years of experience of running production workloads at Google, combined with best-of-breed ideas and practices from the community.

3 The Architecture

Kubernetes is a container orchestration tool. Orchestration is another word for lifecycle management. A container orchestrator does many tasks, including:

- Container provisioning.
- Maintaining the state (and number) of running containers.
- Distribute application load evenly on the hardware nodes by moving containers from one node to the other.
- Load balancing among containers that host the same service.
- Handling container persistent storage.
- Ensuring that the application is always available even when rolling out updates.



3.1 Kubernetes Core Features

It is the most basic part of the whole system. It offers a number of RESTful APIs that enable the cluster to do its most basic operations. The other part of the core is execution. Execution involves a number of controllers like replication controller, replicaset, and deployments.

It also includes the kubelet, which is the module responsible for communicating with the container runtime. The core is also responsible for contacting other layers (through kubelet) to fully manage containers.

3.2 CRI - Container Runtime Interface

Kubernetes uses Container Runtime Interface (CRI) to transparently manage your containers without necessarily having to know (or deal with) the runtime used.

3.3 The Network Plugin

A container orchestration system must be responsible for managing the network nexus where containers and services communicate. Kubernetes uses a library called Container Network Interface (CNI) as a medium between the cluster and various network providers. There are a number of network providers that can be used in Kubernetes.

3.4 The Volume Plugin

The volume Plugin is responsible for the volumes of containers and how to store the persistent volume storage. Kubernetes uses the CSI (Container Storage Interface) to interact with various storage plugins that are already available.

3.5 The Image Registry

Kubernetes must contact an image registry (whether public or private) to be able to pull images and spin out containers.

3.6 Cloud Provider

Kubernetes can be deployed on almost any platform and depends on the cloud provider APIs to perform scalability and resource provisioning tasks, such as provisioning load balancers, accessing cloud storage and utilizing the inter-node network, etc…

3.7 Identity Provider

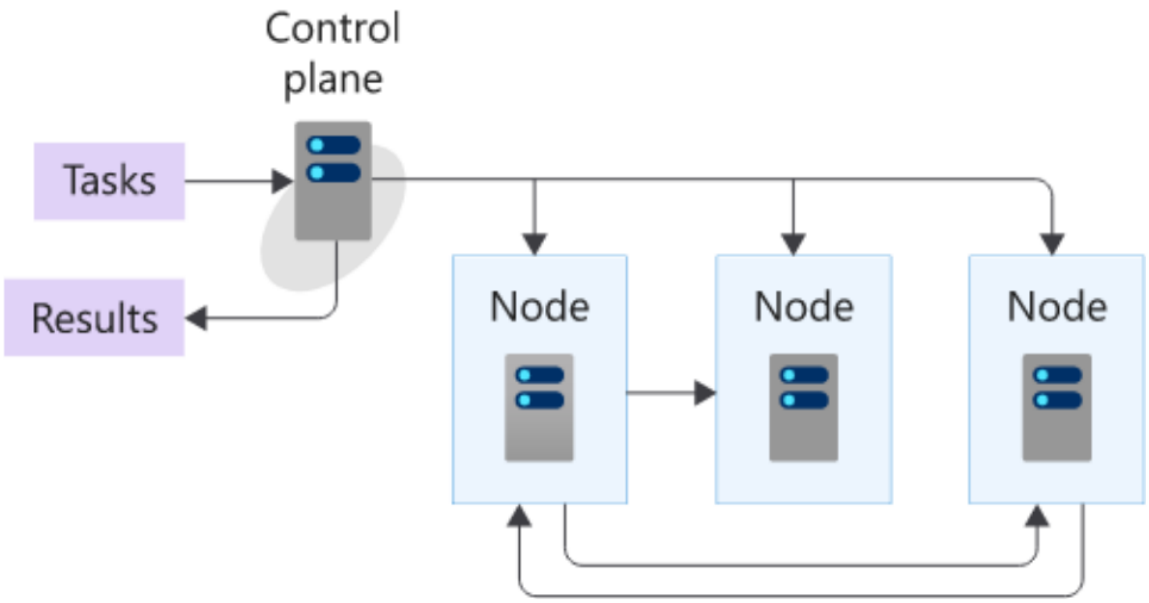
If you're provisioning a Kubernetes cluster in a small company with a small number of users, authentication won't be a big issue. You can create an account for each user and that's it. But, if you're working in a large enterprise, with thousands of developers, testers, security professionals, etc., then having to manually create an account for each person may turn into a nightmare.

You can use your own identity provider system to authenticate your users to the cluster as long as it uses OpenID connect.

4 Kubernetes Cluster

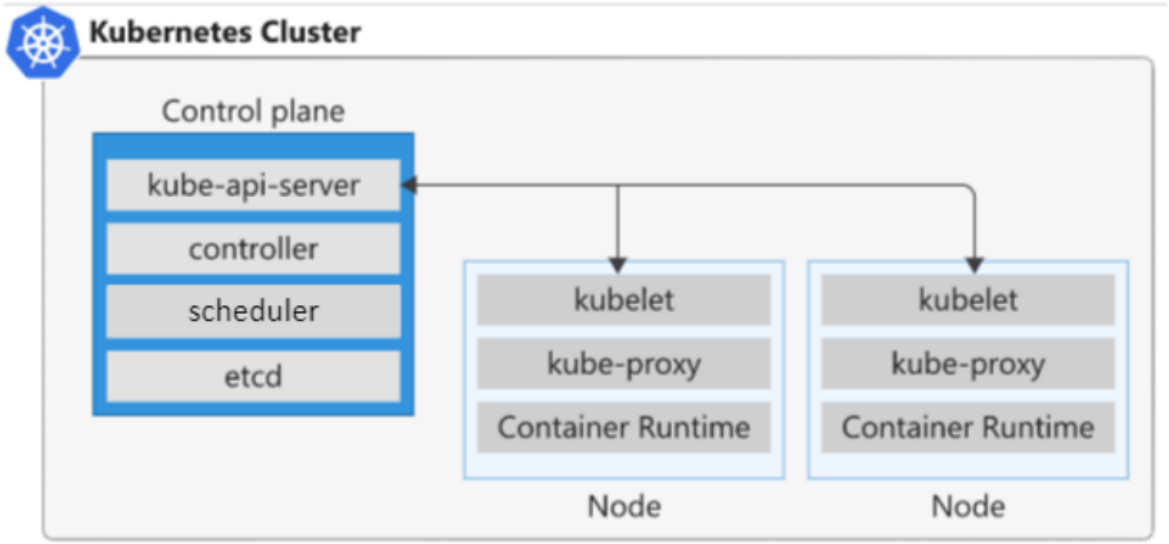
A cluster is a set of computers that you configure to work together and view as a single system. The computers configured in the cluster will typically do the same kinds of tasks. A cluster uses centralized software that's responsible for scheduling and controlling these tasks.

The computers in a cluster that run the tasks are called nodes, and the computers that run the scheduling software are called control planes.



5 Kubernetes Cluster Architecture

The control plane machine components include the Kubernetes API server, Kubernetes scheduler, Kubernetes controller manager, and ETCD. Kubernetes node components include a container runtime engine or docker, a Kubelet service, and a Kubernetes proxy service.



5.1 Control Plane

5.1.1 API Server

The front end of the Kubernetes control plane, the API Server supports updates, scaling, and other kinds of lifecycle orchestration by providing APIs for various types of applications.

Clients must be able to access the API server from outside the cluster, because it serves as the gateway, supporting lifecycle orchestration at each stage.

In that role, clients use the API server as a tunnel to pods, services, and nodes, and authenticate via the API server. This API exposes a RESTful API that you can use to post commands or YAML-based configuration.

5.1.2 Scheduler

The scheduler stores the resource usage data for each compute node; determines whether a cluster is healthy, and determines whether new containers should be deployed, and if so, where they should be placed.

The scheduler considers the health of the cluster generally alongside the pod's resource demands, such as CPU or memory. Then it selects an appropriate compute node and schedules the task, pod, or service, taking resource limitations or guarantees, data locality, and the quality of the service requirements.

5.1.3 Controller Manager

There are various controllers in a Kubernetes ecosystem that drive the states of endpoints, tokens and service accounts (namespaces), nodes, and replication (autoscaling).

The controller manager is a daemon that runs the Kubernetes cluster using several controller functions. It watches the objects it manages in the cluster as it runs the Kubernetes core control loops.

It observes them for their desired state and current state via the API server.

5.1.4 ETCD

Distributed and fault-tolerant, ETCD is an open-source, key-value store database that stores configuration data and information about the state of the cluster. Etcd stores the cluster state based on the Raft consensus algorithm.

This helps cope with a common problem that arises in the context of replicated state machines and involves multiple servers agreeing on values.

5.2 Worker Nodes

5.2.1 Kubelet

Each compute node includes a kubelet, an agent that communicates with the control plane to ensure the containers in a pod are running.

When the control plane requires a specific action that happens in a node, the Kubelet receives the pod specifications through the API server and executes the action.

It then ensures the associated containers are healthy and running.

5.2.2 Kubeproxy

The Kube-proxy component is responsible for local cluster networking and runs on each node.

It ensures that each node has a unique IP address. It also implements rules to handle routing and load balancing of traffic by using IP tables and IPVS. This proxy doesn't provide DNS services by itself.

A DNS cluster add-on based on CoreDNS is recommended and installed by default.

5.2.3 Container Runtime

The container runtime is the underlying software that runs containers on a Kubernetes cluster.

The runtime is responsible for fetching, starting, and stopping container images.

Kubernetes supports several container runtimes, including but not limited to Docker, RKT, CRI-O, Containerd, and Frakti.

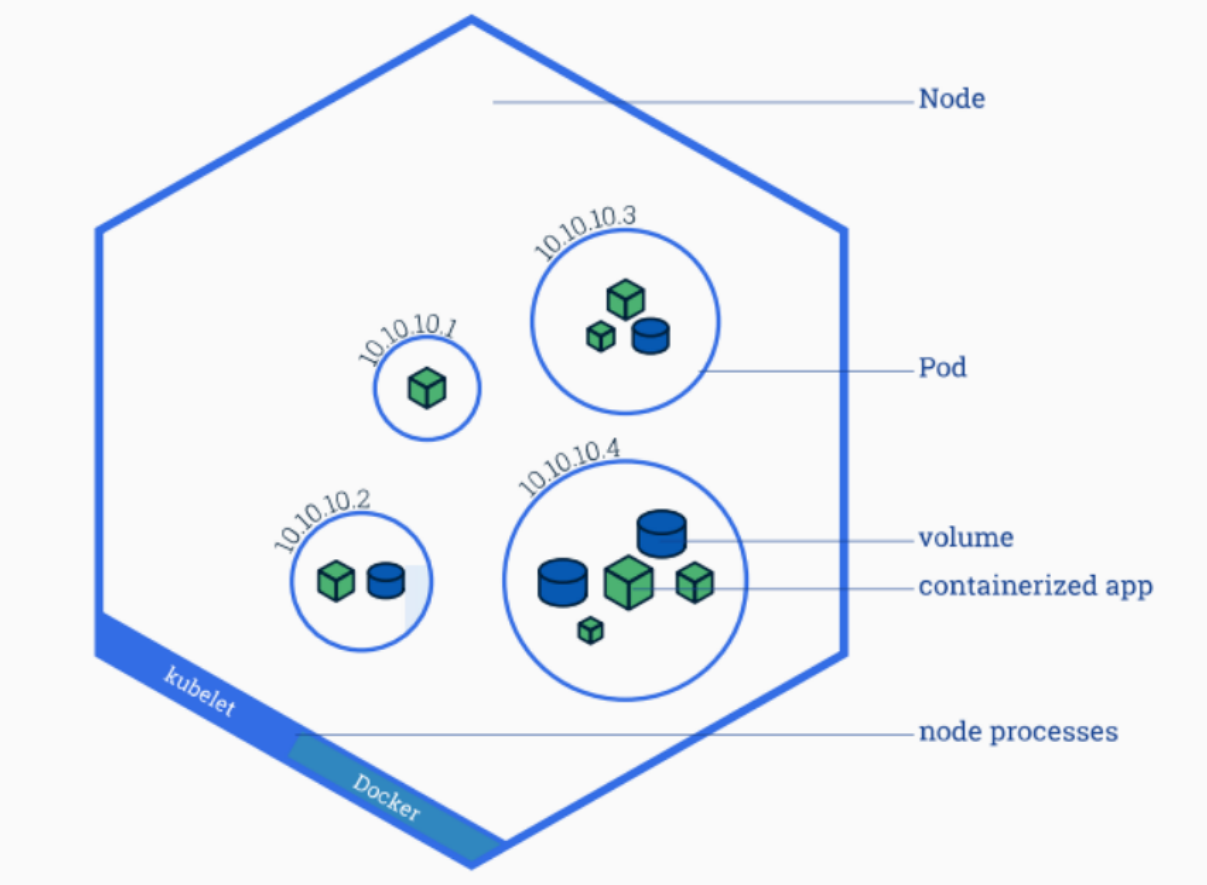
6 Pods, StatefulSets, ReplicaSets, and Deployments

6.1 Pod

6.1.1 Definition

A pod is the smallest unit in a Kubernetes cluster. A pod may contain one or more containers. Pods can be scheduled (in Kubernetes terminology, scheduling means deploying) to a node.

A pod is a self-contained logical process, with an isolated environment. It has its own IP address, storage, hostname, etc. However, a pod can host more than one container. So, a pod can be thought of as a container of containers.



We can manage and create objects in our Kubernetes cluster using Kubectl or YAML files. Kubectl is the client tool that is used to send API requests to K8s.

6.1.2 Creating a Pod Definition with YAML

```yaml
apiVersion: v1
kind: Pod
metadata:
  name: webserver
spec:
  containers:
  - name: webserver
    image: nginx:latest
    ports:
    - containerPort: 80
```

Creating the Pod with Kubectl:

```
root@localhost:~# kubectl apply -f pod001.yaml
pod/webserver created
root@localhost:~# kubectl get pods
NAME            READY       STATUS       RESTARTS         AGE
webserver        1/1        Running        0               19m
```

6.2 ReplicaSet

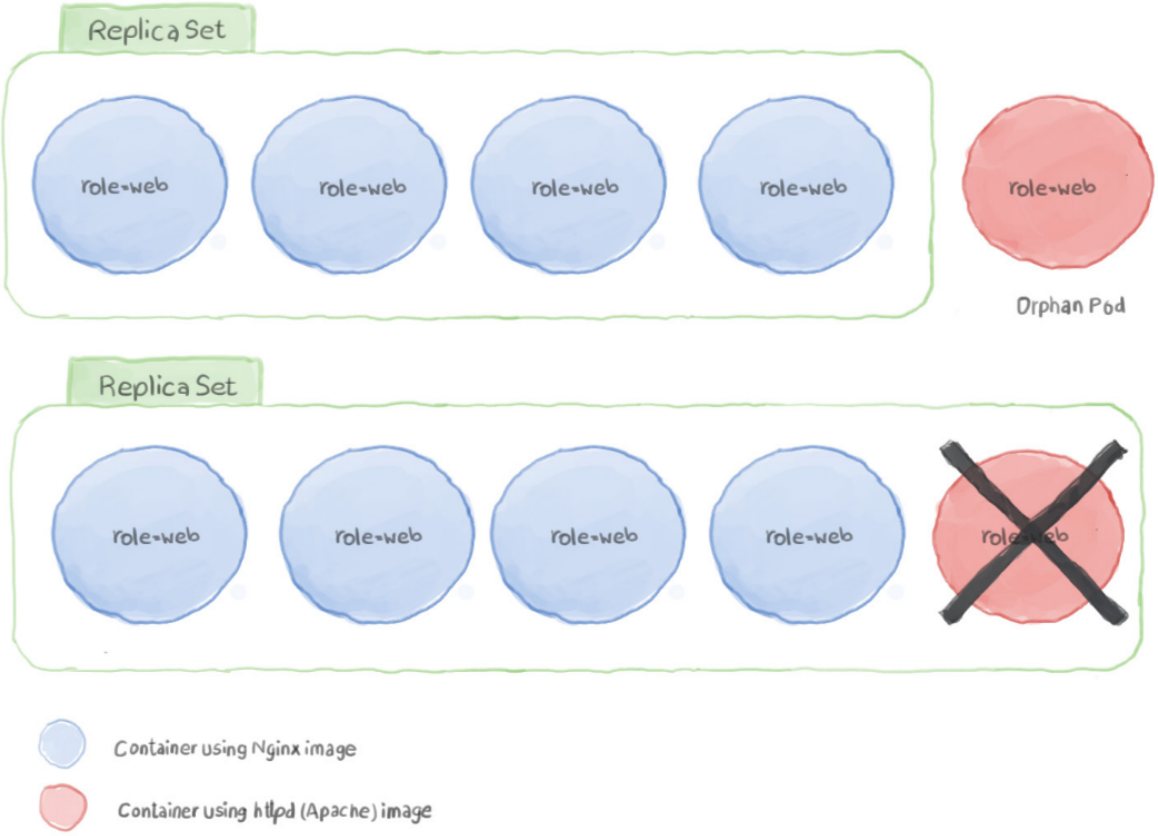
6.2.1 Definition

A ReplicaSet is a process that runs multiple instances of a Pod and keeps the specified number of Pods constant. Its purpose is to maintain the specified number of Pod instances running in a cluster at any given time to prevent users from losing access to their application when a Pod fails or is inaccessible.

In order for a ReplicaSet to work, it needs to know which pods it will manage so that it can restart the failing ones or kill the unneeded. It also requires understanding how to create new pods from scratch in case it needs to spawn new ones.

A ReplicaSet uses labels to match the pods that it will manage. It also needs to check whether the target pod is already managed by another controller.

When you create a pod with the same tag/label that matches the ReplicaSet you created, the ReplicaSet will have full control of it.



6.2.2 Creating a ReplicaSet Definition with YAML

The ReplicaSet YAML file definition is similar to the Pod definition file, it needs apiVersion, kind, metadata, etc. In addition, it needs a template value where you write the definition of the pod that will be controlled by the ReplicaSet, and don't forget the selector that will match every pod that has the same value with this ReplicaSet.

Every pod created by the ReplicaSet will have a unique name generated randomly by the ReplicaSet.

Furthermore, you can scale up and down the number of pods dynamically either with the kubectl utility or by editing the definition file.

```
apiVersion: apps/v1
kind: ReplicaSet
metadata:
   name: web
   labels:
     role: web
spec:
```

```
    selector:
      matchLabels:
         role: web
   template:
      metadata:
         labels:
            role: web
          spec:
              containers:
              - name: nginx
                image: nginx
```

Creating the ReplicaSet with Kubectl:

```
root@localhost:~# kubectl apply -f replica000.yaml
replicaset/web created
root@localhost:~# kubectl get pods
NAME           READY        STATUS              RESTARTS    AGE
web-6n9cj       0/1    ContainerCreating           0       15s
web-7kqbm       0/1    ContainerCreating           0       15s
web-9src7       0/1    ContainerCreating           0       16s
web-fvxzf       0/1    ContainerCreating           0       15s
```

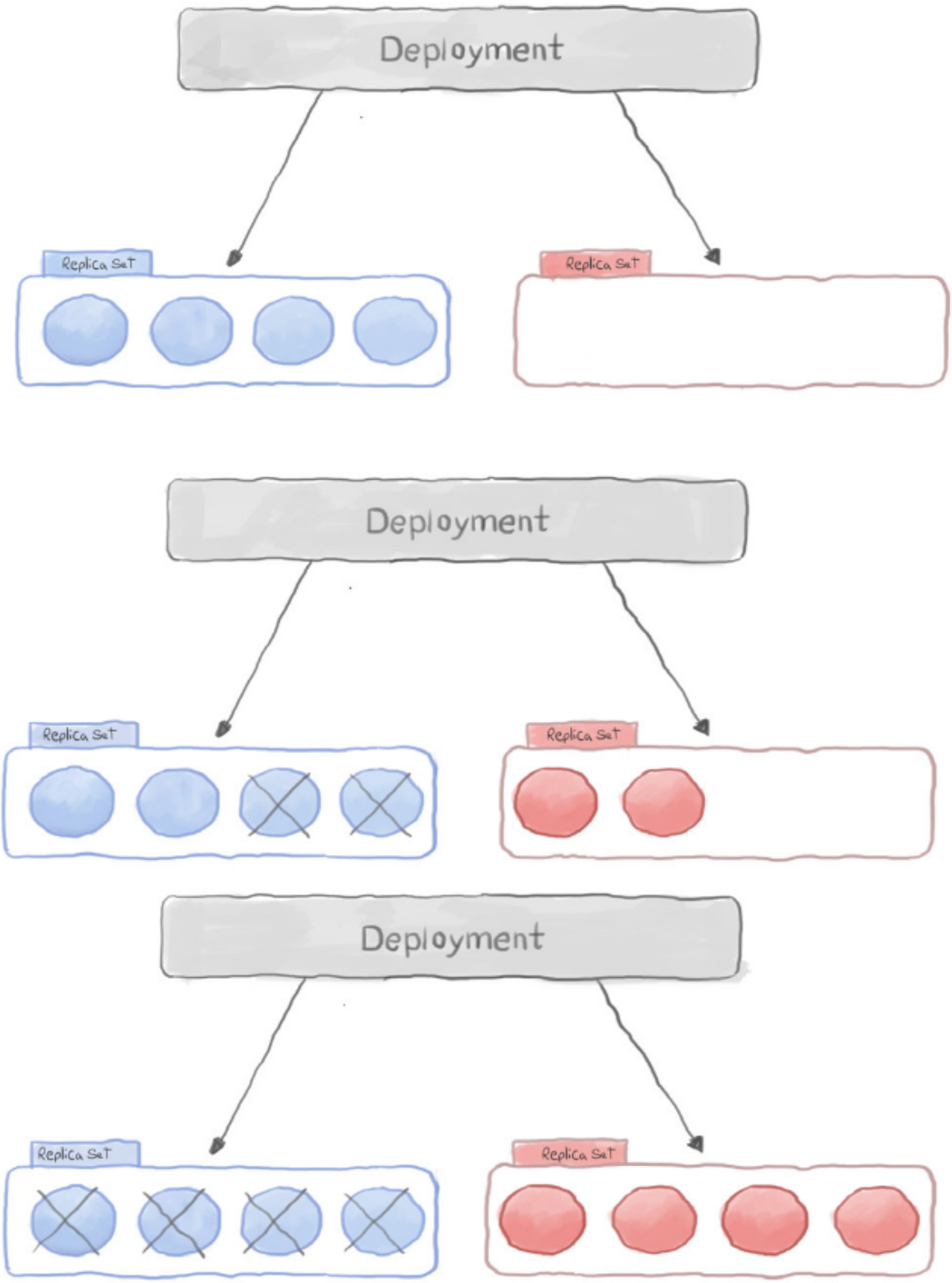
6.3 Deployment

6.3.1 Definition

Kubernetes Deployments are one of the most potent controllers you can use. They not only maintain a specified number of pods but also ensure that any updates you want to make do not cause downtime. Behind the scenes, Deployments use ReplicaSets to manage pods.

Deployments support rollover updates in which you can interrupt an ongoing deployment update and instruct the Deployment controller to start the new update immediately without causing an application outage. Kubernetes maintains a list of the recent deployments.

You can use this list to roll back an update and you can also choose a specific deployment to roll back to by specifying its revision number. You can use Deployments to scale up, or down, the number of pods it is managing.


Deployment
Replica Set
Replica Set
Deployment
Replica Set
Replica Set
Deployment
Replica Set
Replica Set

6.3.2 Creating a Deployment Definition with YAML

```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
    name: apache
    labels:
       role: webserver
spec:
   replicas: 2
   selector:
      matchLabels:
         role: webserver
   template:
      metadata:
         labels:
            role: webserver
      spec:
          containers:
          - name: frontend
            image: httpd
            ports:
            - containerPort: 80
```

Creating the Deployment with Kubectl:

```
root@localhost:~# kubectl apply -f dep000.yaml
Deployments/apache Created
root@localhost:~# kubectl get
root@localhost:~# kubectl get pods
NAME            READY          STATUS          RESTARTS     AGE
apache-6n9cj     0/1     ContainerCreating        0         15s
apache-7kqbm     0/1     ContainerCreating        0         15s
```

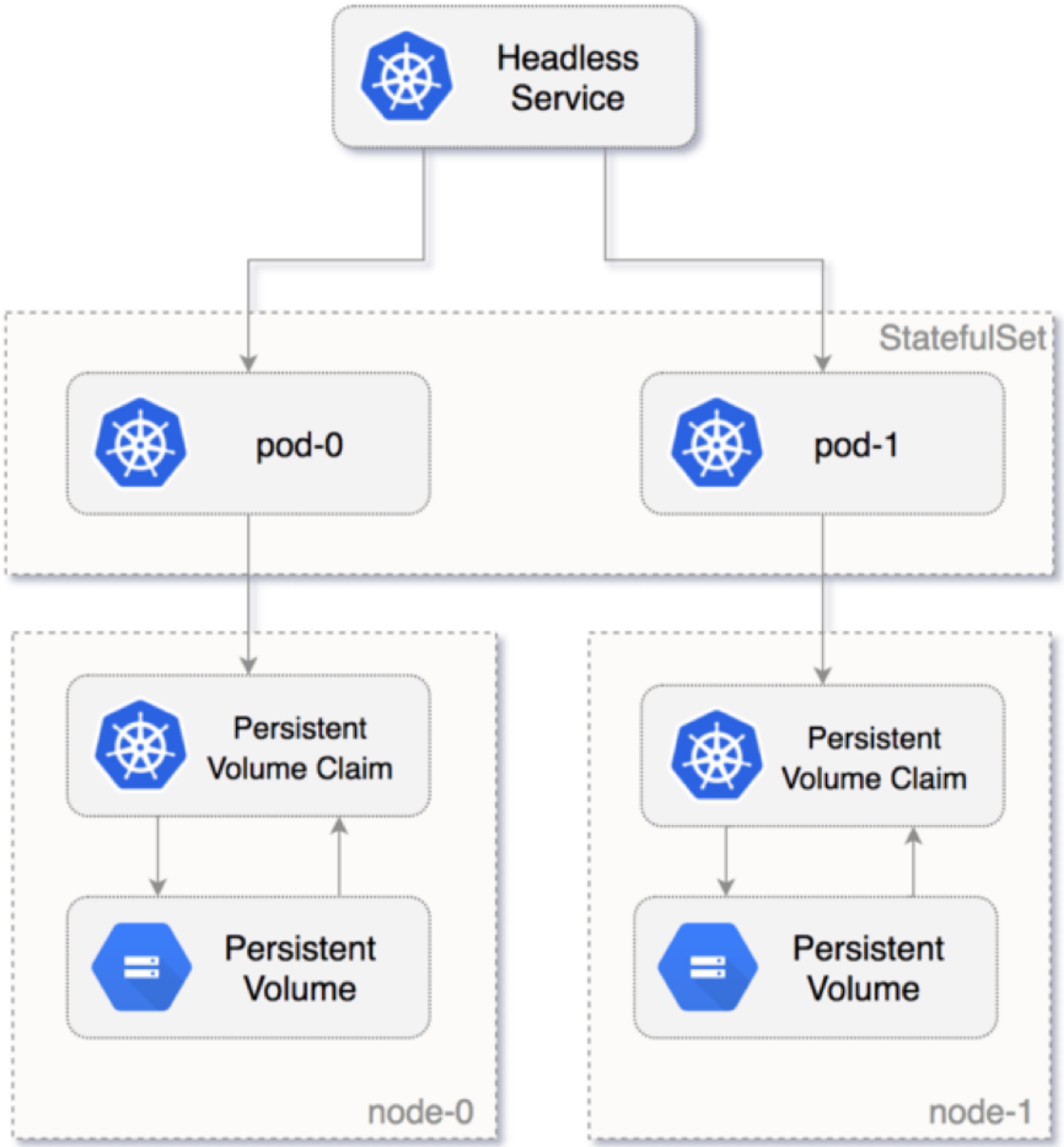
6.4 StatefulSet

6.4.1 Definition

StatefulSets are used to manage stateful applications. StatefulSets are similar to Deployments, but they are more suitable for stateful applications. StatefulSets provide guarantees about the ordering and uniqueness of Pods. StatefulSets are useful for applications that require one or more of the following.

- Stable, unique network identifiers.
- Stable, persistent storage.

- Ordered, graceful deployment and scaling.
- Ordered, automated rolling updates.
- Ordered, automated rollbacks.
- Ordered, automated replacement of unhealthy replicas.
- Ordered, automated replacement of unresponsive replicas.
- Ordered, automated replacement of deleted replicas.



6.4.2 Creating a StatefulSet Definition with YAML

```yaml
apiVersion: apps/v1
kind: StatefulSet
metadata:
  name: web
spec:
    serviceName: "nginx"
    replicas: 3
    selector:
        matchLabels:
        app: nginx
    template:
        metadata:
        labels:
            app: nginx
        spec:
        terminationGracePeriodSeconds: 10
        containers:
        - name: nginx
            image: nginx:1.7.9
            ports:
            - containerPort: 80
            name: web
```

Creating the StatefulSet with Kubectl:

```
root@localhost:~# kubectl apply -f statefulset000.yaml
statefulset.apps/web created
root@localhost:~# kubectl get pods
NAME            READY          STATUS          RESTARTS     AGE
web-0           0/1     ContainerCreating        0          15s
web-1           0/1     ContainerCreating        0          15s
web-2           0/1     ContainerCreating        0          15s
```