Kubernetes

Kubernetes Fundamentals

Kubernetes Basics

What is Kubernetes?

Kubernetes is an **orchestrator**.
Kubernetes **orchestrates containerized cloud-native apps**.

Kubernetes is an open source **container management tool**.
It is a Go language-based, lightweight and portable application.

You can set up a Kubernetes cluster on a Linux-based OS to **deploy, manage, and scale** Docker **container applications on multiple hosts**.

Cloud-native application

A cloud-native application is a business application that is **made from a set of small independent services that communicate and form a useful application**.

Despite the name, cloud-native apps can also run on-premises.

Where did Kubernetes come from?

Kubernetes was **released by Google**.

It was **open sourced** in the summer of 2014 and handed over to the Cloud Native Computing Foundation (CNCF).

CNCF: https://www.cncf.io

What's in the Name?

The name Kubernetes (koo-ber-net-eez) comes from the Greek word meaning Helmsman – the person who steers a ship.

Often Kubernetes shortened to **K8s**

Original name: Seven of Nine (Borg from Star Trek)

Kubernetes and Docker

Kubernetes and Docker are **complementary technologies**.

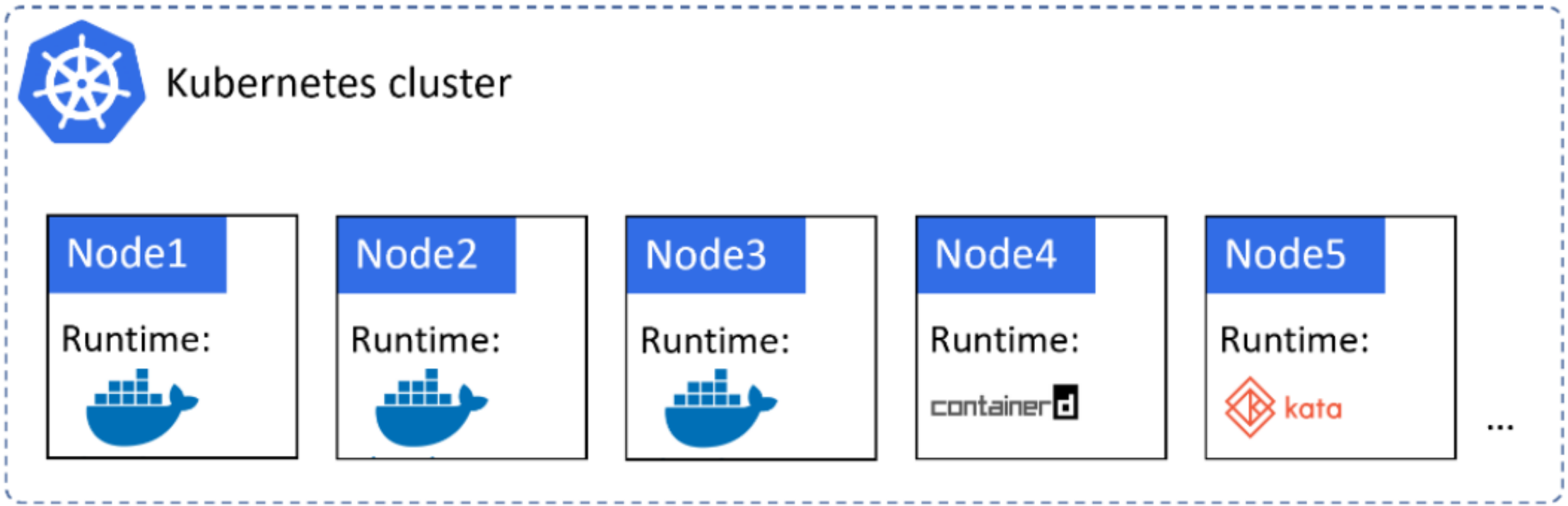
Docker is the low-level technology that starts and stops containers.

Kubernetes is the higher-level technology that deciding which nodes to run containers on, deciding when to scale up or down, and executing updates.

Container Runtime Interface (CRI)

Docker isn't the only container runtime that Kubernetes supports.

CRI is an abstraction layer that standardizes the way third-party container runtimes interface with Kubernetes. (example: gVisor, Kata)

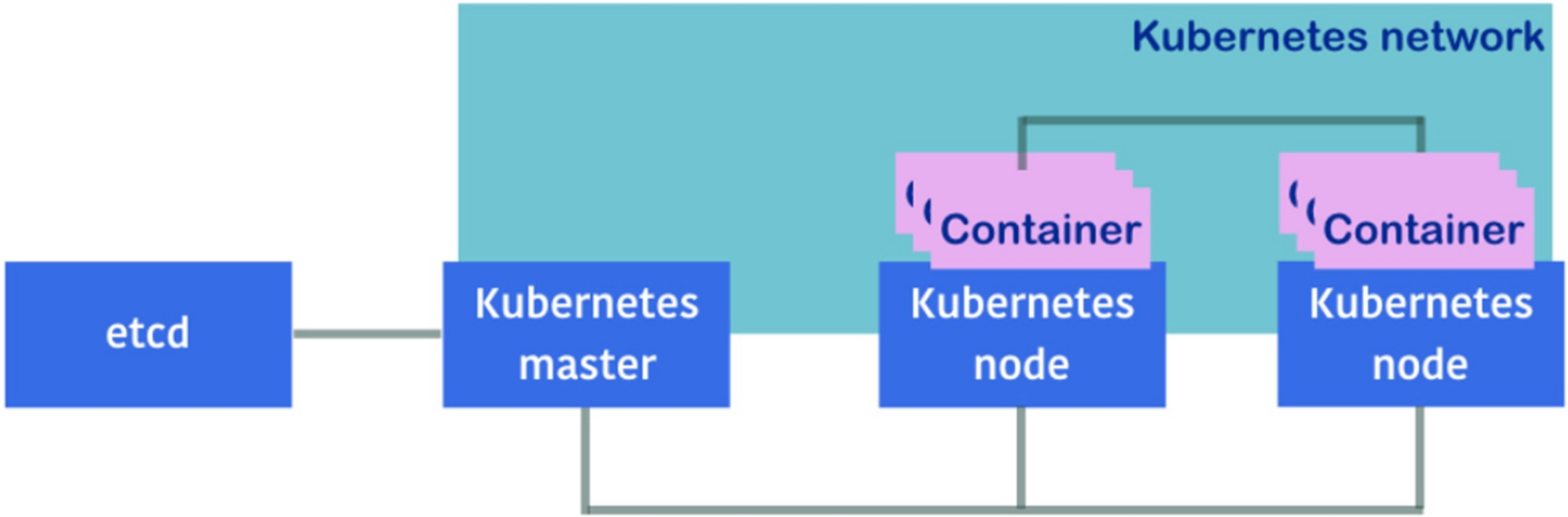

Kubernetes cluster
Node1
Runtime:
Node2
Runtime:
Node3
Runtime:
Node4
Runtime:
containerd
Node5
Runtime:
kata
...


kubernetes
VS
docker
SWARM
MESOS

Kubernetes components

Kubernetes is made up of the following components:

- Kubernetes master
- Kubernetes nodes
- etcd
- Kubernetes network


Kubernetes network
Container
Container
etcd
Kubernetes master
Kubernetes node
Kubernetes node

Masters and Nodes

A Kubernetes cluster is made of masters and nodes.

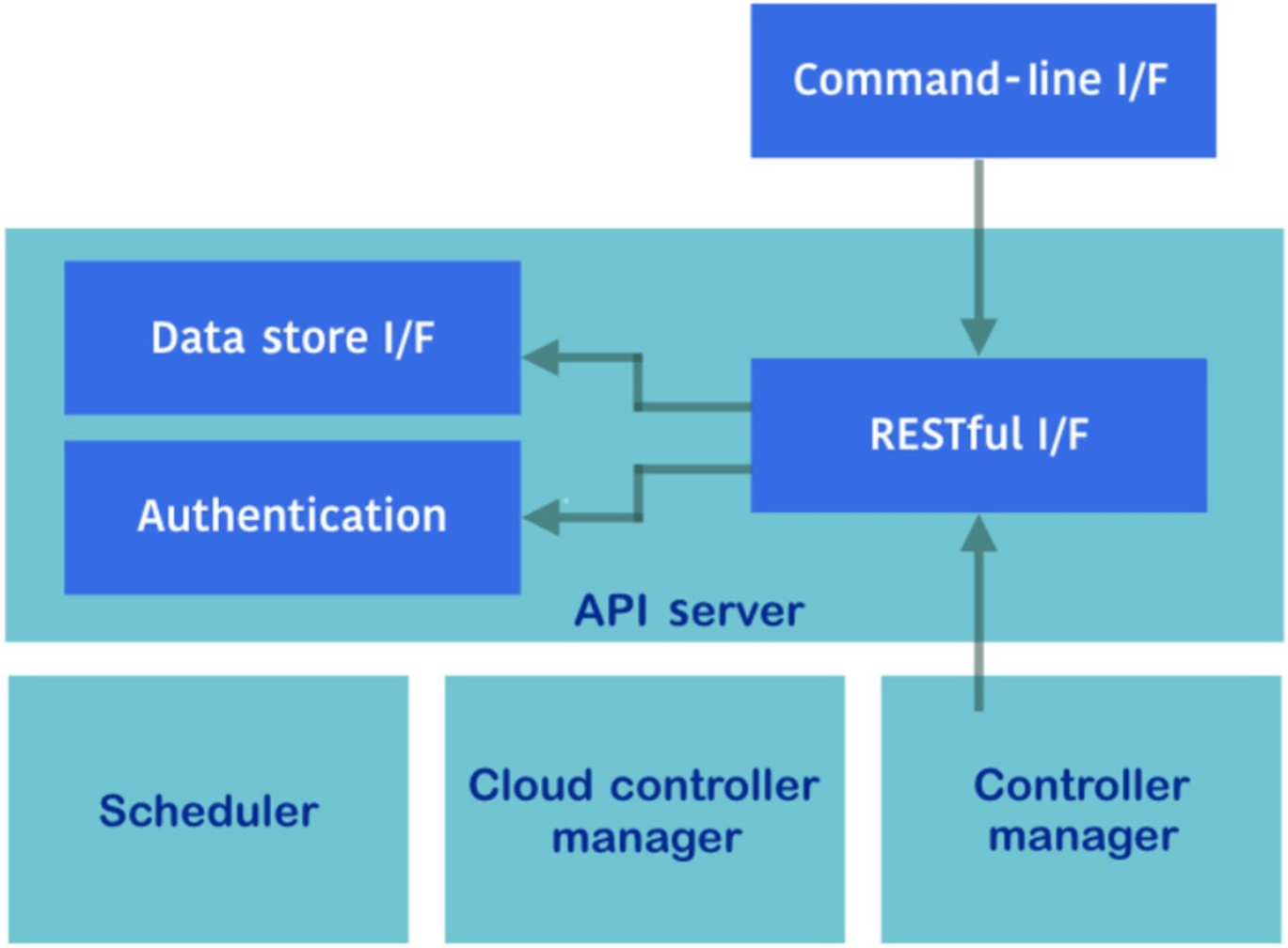
These are (Linux) hosts that can be VMs, bare metal servers in your data center, or instances in a private or public cloud.

Kubernetes master (control plane)

The Kubernetes master is the **main component** of the Kubernetes cluster. It serves several functionalities, such as the following:

- Authorization and authentication
- RESTful API
- Container deployment scheduler to Kubernetes nodes
- Scaling and replicating controllers
- Reading the configuration to set up a cluster

Kubernetes master daemons



API server (kube-apiserver)

The API server provides an HTTP- or HTTPS-based RESTful API, which is the **hub between Kubernetes components**, such as kubectl, the scheduler, the replication controller, the etcd data store, and the kubelet and kube-proxy, which runs on Kubernetes nodes.

Scheduler (kube-scheduler)

The scheduler helps to **choose which container runs on which nodes**.

It is striving to balance resource consumption to not place excessive load on any cluster node. It also takes user scheduling restrictions into account, such as (anti-)affinity rules.

Controller manager (kube-controller-manager)

The controller manager **performs cluster operations**:

- Manages Kubernetes nodes
- Creates and updates the Kubernetes internal information
- Attempts to change the current status to the desired status

etcd (cluster store)

etcd is the distributed **key-value data store**.

It can be accessed via the RESTful API to perform CRUD operations over the network.

Kubernetes uses etcd as the main data store (configuration and status).

If etcd becomes unavailable, apps running on the cluster should continue to work but updates to the cluster configuration will be halted.

Command-line interface (kubectl)

You can use the Kubernetes command-line interface, **`kubectl`**, to control the Kubernetes cluster. For example

`kubectl get cs` returns the status of each component.

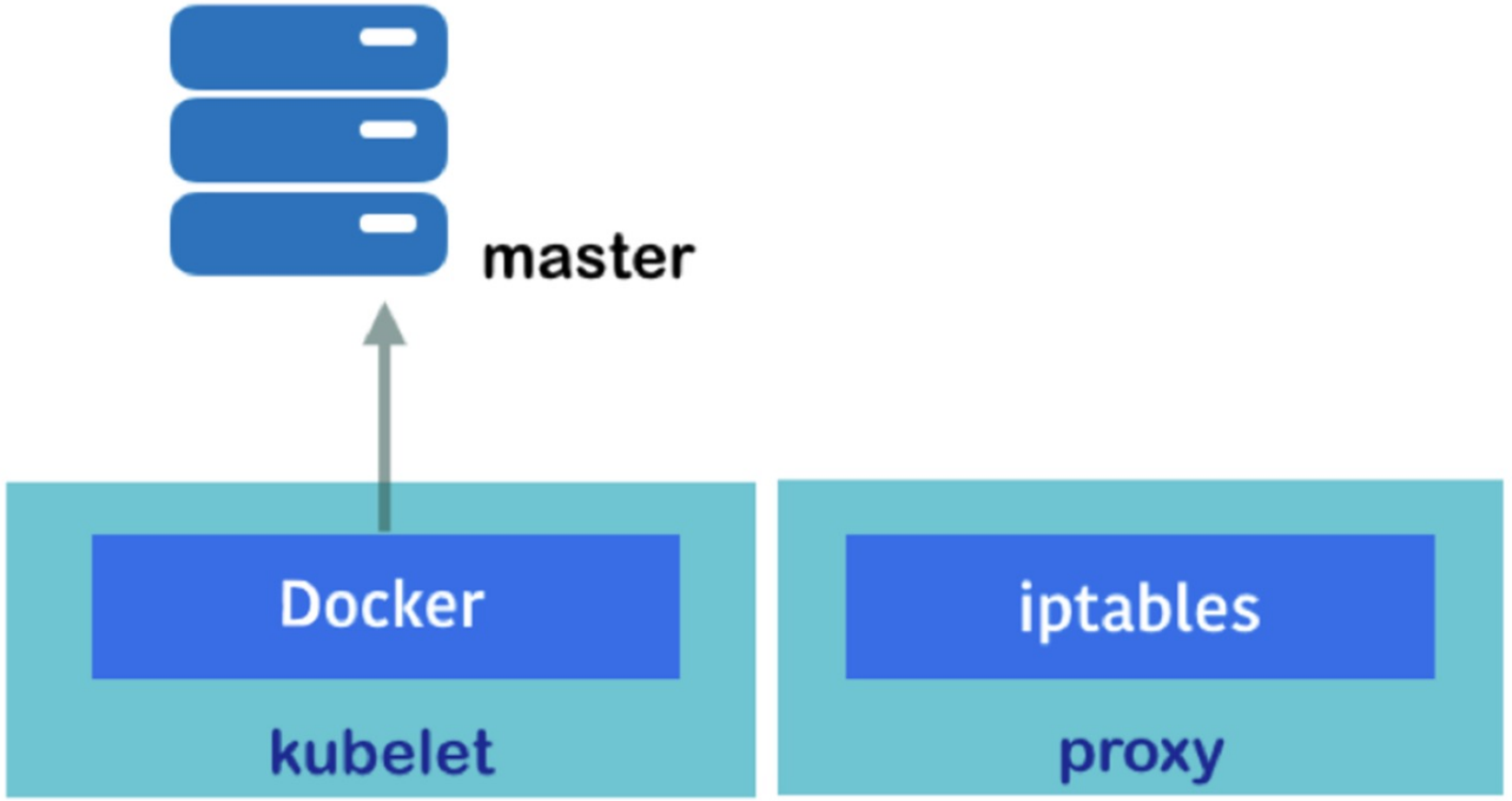
`kubectl get nodes` returns a list of Kubernetes nodes.

Kubernetes node

The Kubernetes node is a **slave node** in the Kubernetes cluster.
It is **controlled by the Kubernetes master** to run container applications using Docker (or rkt).

The node has two daemon processes, named **kubelet** and **kube-proxy**, to support its functionalities.

Kubernetes node (slave) daemons



Kubelet

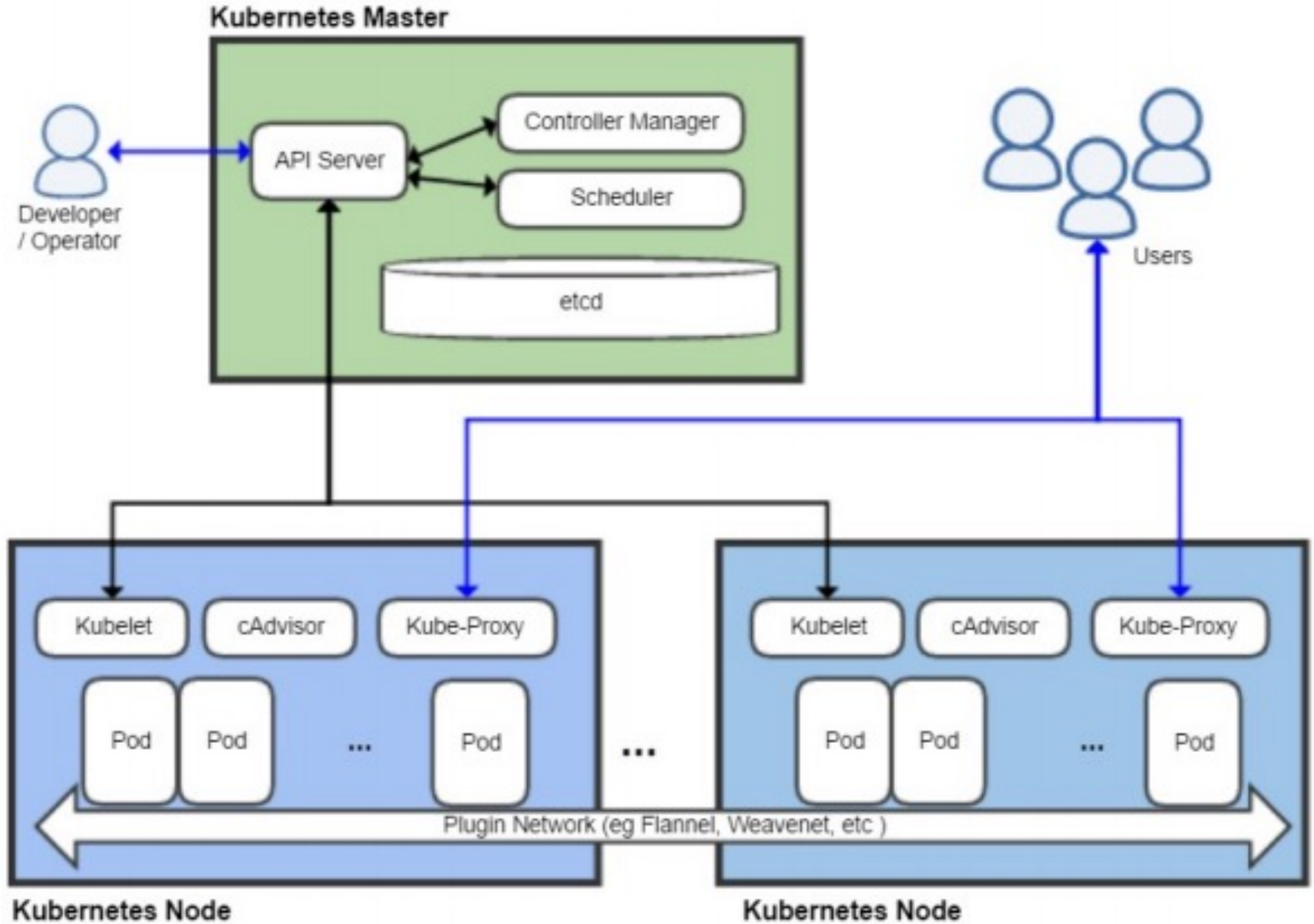
kubelet is the **main process** on the Kubernetes node that communicates with the Kubernetes master to handle the following operations:

- Periodically accesses the API controller to check and report
- Performs container operations
- Runs the HTTP server to provide simple APIs

Proxy (kube-proxy)

The proxy handles the **network proxy and load balancer** for each container.

It changes Linux **iptables rules** (nat table) to control TCP and UDP packets across the containers.


Kubernetes Master
Developer
/ Operator
API Server
Controller Manager
Scheduler
etcd
Users
Kubelet
cAdvisor
Kube-Proxy
Pod
Pod
...
Pod
...
Kubelet
cAdvisor
Kube-Proxy
Pod
Pod
...
Pod
Plugin Network (eg Flannel, Weavenet, etc)
Kubernetes Node
Kubernetes Node

cAdvisor

cAdvisor is an open source container **resource usage collector**.
Supports Docker containers natively.

Unlike most elements within Kubernetes that operate at the Pod level, cAdvisor **operates per node**.

Kubernetes DNS

Kubernetes cluster has an **internal DNS service** that is vital to operations.

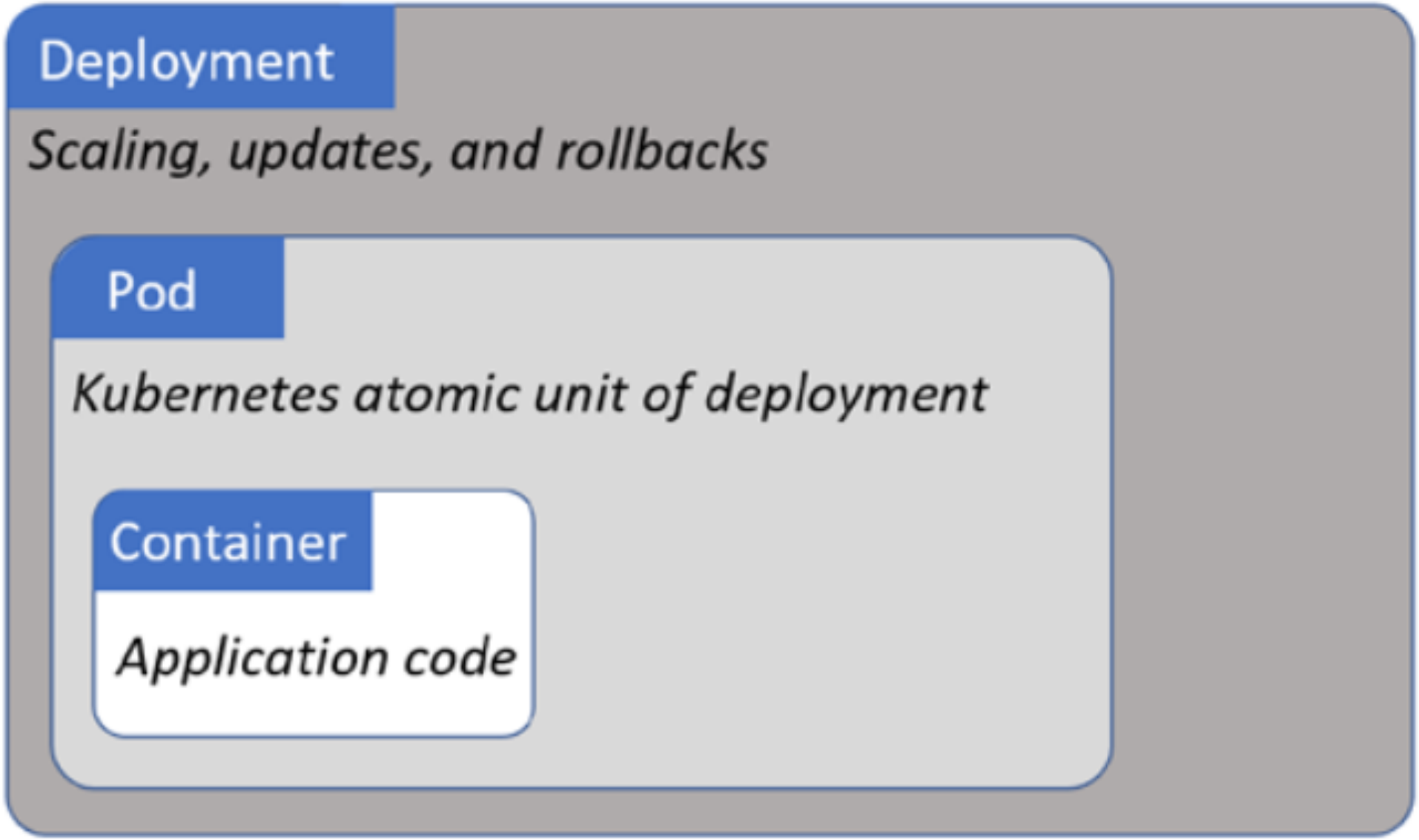
Every **new service is automatically registered** with the cluster's DNS so that all components in the cluster can find every Service by name.

Cluster DNS is based on CoreDNS (https://coredns.io/).

Kubernetes Principles

Packaging Apps

- Packaged as a container image
- Wrapped in a Kubernetes Pod
- Deployed via a declarative manifest file


Deployment
Scaling, updates, and rollbacks
Pod
Kubernetes atomic unit of deployment
Container
Application code

Pods

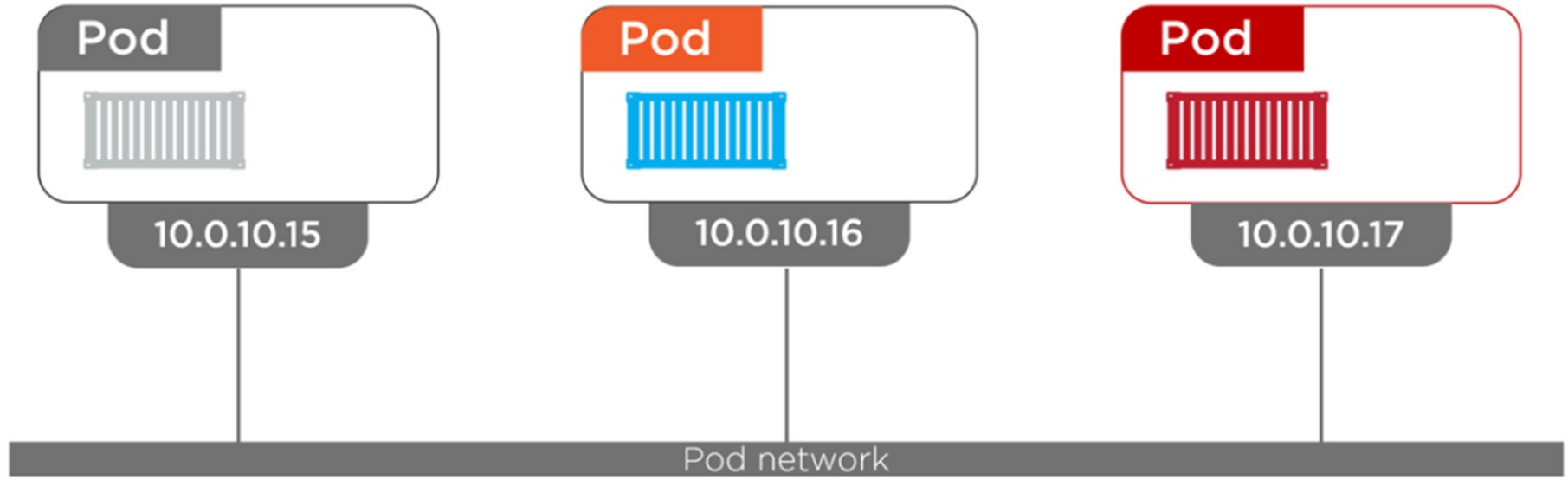
Pods is the **atomic unit** of scheduling in Kubernetes.
Containers must always run inside of Pods.

Pod comes from a pod of whales: group of whales

Pod is a construct for **running one or more containers**.
The simplest model is to run a single container per Pod.

Pod’s network

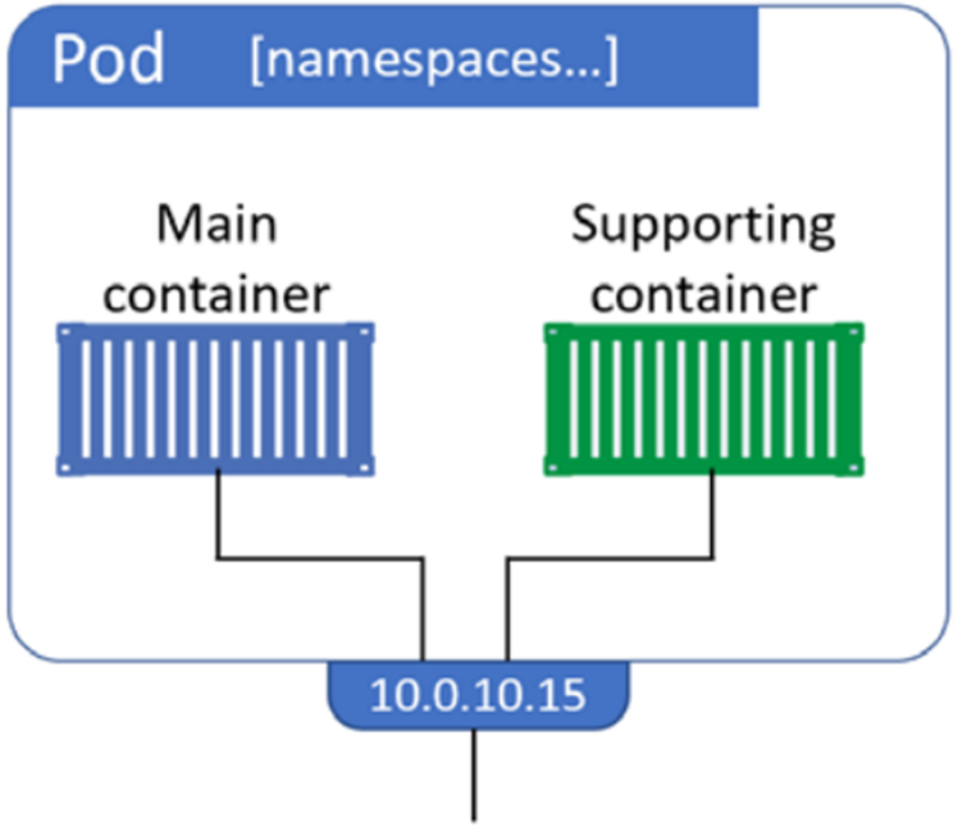
Every Pod gets its own routable IP, every Pod on the Pod network can talk directly to every other Pod.



Pod’s network

If you're running **multiple containers in a Pod, they all share the same environment (namespaces)**.

This includes the IPC, shared memory, volumes, network stack, etc.



Pods and cgroups

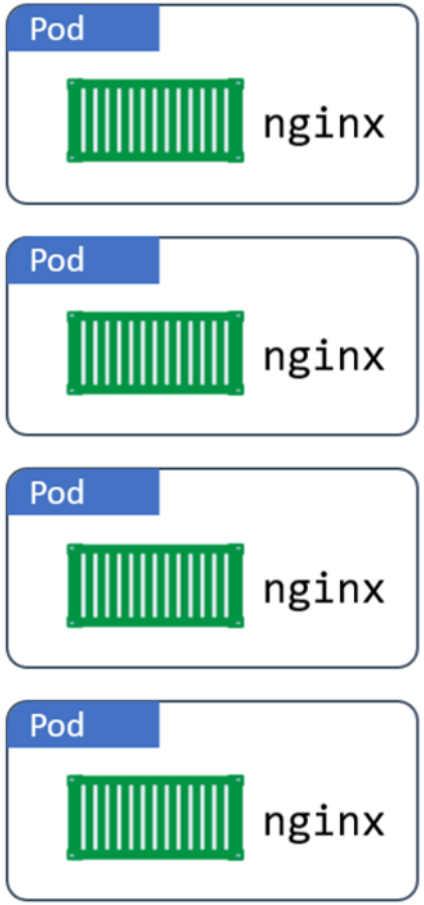
Control Groups (cgroups) prevent individual containers from consuming all of the available CPU, RAM, and Input/Output Operations Per Second (IOPS) on a node.

Individual containers have their own cgroup limits.

Pods as the Unit of Scaling

If you need **to scale** your app, you **add or remove Pods**.
You do not scale by adding more containers to an existing Pod.

Multi-container Pods are only for situations where two different, but complementary, containers need to share resources.



Pod lifecycle

Pods are **mortal**. Created, live, and die.

If Pod **die unexpectedly, Kubernetes starts a new one in its place**.

Deployments

We normally deploy Pods indirectly.

Deployment is a higher-level Kubernetes object that wraps around a particular Pod and **adds features such as scaling, zero-downtime updates, and versioned rollbacks**.

Service

Services provide **reliable networking** for a set of Pods;

- **stable DNS name, IP address, and port,**
- **load-balance across a dynamic set of Pods.**

Services are objects in the Kubernetes API like Pods and Deployments

Setting up the Kubernetes cluster

Kubernetes learning enviroments

- Docker for Desktop (Windows, OSX)
  Ref.: https://www.docker.com/products/docker-desktop
- Play with Kubernetes (PWK)
  Ref.: https://labs.play-with-k8s.com
- Minikube
  Ref.: https://kubernetes.io/docs/setup/learning-environment/minikube
- MicroK8s
  Ref: https://microk8s.io

Kubernetes production enviroments

- kubeadm
  Ref.: https://kubernetes.io/docs/setup/production-environment/tools/kubeadm/install-kubeadm/
- Kubespray
  Ref.: https://kubernetes.io/docs/setup/production-environment/tools/kubespray/
- kops on AWS
  Ref.: https://kubernetes.io/docs/setup/production-environment/tools/kops/

Demonstration: Create a Kubernetes cluster

1) Creating of a Kubernetes cluster based on the trainer instructions.

2) Use **`kubectl`** command to get cluster's informations.

Kubernetes Objects

Kubernetes Objects

- Pod
- Service
- Volume
- Namespace

Object: Pod

A Pod is the basic execution unit of a Kubernetes application.
The **smallest** and simplest **unit** in the Kubernetes object model that you create or deploy.

Pods in a Kubernetes cluster can be used in two main ways:

- **Pods that run a single container**; wrapper around a single container.
- **Pods that run multiple containers** that need to work together. The Pod wraps these containers and storage resources together as a single manageable entity.

Object: Service

An abstract way to **expose an application** running on a set of Pods **as a network service**.

Kubernetes gives **Pods their own IP addresses and a single DNS name for a set of Pods, and can load-balance across them**.

ServiceTypes: ClusterIP, NodePort, LoadBalancer, ExternalName

Object: Volumes

More then one volume driver is allowed per Pod/Container.

There are several types of volumes to use:

- local,
- hostPath,
- nfs,
- iscsi,
- cephfs,
- azureDisk, azureFile (SMB),
- secret.

Object: Namespaces

Kubernetes supports **multiple virtual clusters** backed by the same physical cluster. These virtual clusters are called namespaces.

Namespaces are intended for use in environments with many users spread across multiple teams, or projects.
Namespaces are a way to divide cluster resources between multiple users (via resource quota).

Controllers

Kubernetes contains a number of **higher-level abstractions** called **Controllers**.

- ReplicaSet
- Deployment
- StatefulSet
- DaemonSet
- Job

Controller: ReplicaSet

A ReplicaSet's purpose is to **maintain a stable set of replica Pods running at any given time**. It is often used to guarantee the availability of a specified number of identical Pods.

Note: A Deployment is a higher-level concept that manages ReplicaSets and provides declarative updates to Pods along with a lot of other useful features. We recommend using Deployments instead of directly using ReplicaSets.

Controller: Deployments

A Deployment controller **provides declarative updates** for Pods and ReplicaSets.

The following are typical use cases for Deployments:

- Create a Deployment to rollout a ReplicaSet.
- Declare the new state of the Pods.
- Rollback to an earlier Deployment revision.
- Scale up the Deployment to facilitate more load.

Controller: StatefulSets

StatefulSet is used **to manage stateful applications**.
Unlike a Deployment, a **StatefulSet maintains a sticky identity for each of their Pods**.

StatefulSets are valuable 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.

Controller: DaemonSet

A DaemonSet **ensures that all (or some) Nodes run a copy of a Pod**.

Some typical uses of a DaemonSet are:

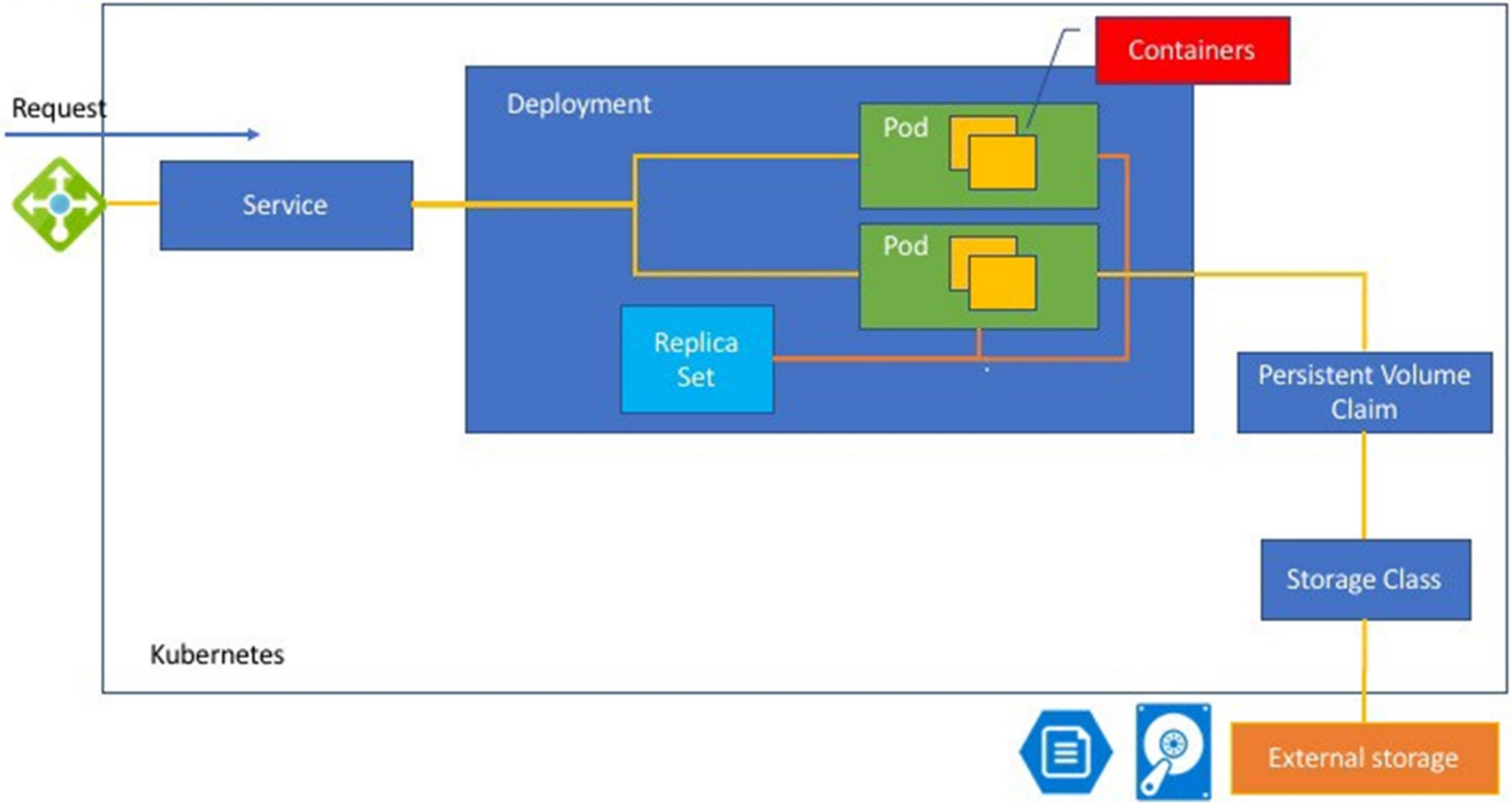
- Running a cluster storage daemon, such as glusterd, ceph, on each node.
- Running a logs collection daemon on every node, such as fluentd or logstash.
- Running a node monitoring daemon on every node.

Controller: Jobs, CronJob

A **Job creates one or more Pods and ensures that a specified number of them successfully terminate**.
When a specified number of successful completions is reached, the task (ie, Job) is complete. Deleting a Job will clean up the Pods it created.

Cron Job creates Jobs on a time-based schedule.
One CronJob object is like one line of a crontab (cron table) file. It runs a job periodically on a given schedule, written in Cron format.

Kubernetes Objects



Kubernetes network

If the container's network communication is only within **a single node**, you can **use Docker network** to discover the peer.

Along with **multiple nodes**, Kubernetes **uses an overlay network** or container network interface (CNI) to achieve multiple container communication.

The Kubernetes network model

Every Pod gets its own IP address.

- **Pods on a node can communicate with all pods on all nodes without NAT.**
- Agents on a node (e.g. system daemons, kubelet) can communicate with all pods on that node.

The Kubernetes CLI: `kubectl`

kubectl

kubectl is the **main Kubernetes command-line tool**.

Important to use a version that is no more than one minor version higher or lower than your cluster.

kubectl converts user-friendly commands into the JSON payload required by the API server.

Ref.: https://kubernetes.io/docs/reference/kubectl/overview

kubectl configuration file

the **`kubectl`** configuration file is called **`config`** and lives in **`$HOME/.kube`**.

It contains definitions for:
- Clusters
- Users
- Contexts

You can view your kubectl config using the **`kubectl config view`** command.

Lab: Use `kubectl` CLI

1) Use `kubectl` command in examples.

2) Install and use Lens the Kubernetes IDE. See: https://k8slens.dev/

Kubernetes Manifest files

PODs with YAML

pod-definition.yml

```yaml
apiVersion: v1
kind: Pod
metadata:
  name: myapp-pod
  labels:
    app: myapp
    type: front-end
spec:
  containers:
    - name: nginx-container
      image: nginx
```

```
kubectl create -f pod-definition.yml
```

Kind	Version
POD	v1
Service	v1
ReplicaSet	apps/v1
Deployment	apps/v1

Manifest files

Top-level resources:

- **apiVersion**
- **kind**
- **metadata**
- **spec**

apiVersion field

The apiVersion format is **<api-group>/<version>**

Core group omits the api-group part.

kind field

kind field tells Kubernetes the **type of object** being deployed.

Kind	Version
POD	v1
Service	v1
ReplicaSet	apps/v1
Deployment	apps/v1

metadata section

metadata section is where we attach a **name and labels**.

Name identify the object in the cluster.

Labels help us create couplings.

spec section

spec section is where we define any containers that will run in the Pod.

If this was a multi-container Pod, we'd define additional containers in the spec section.

Working with PODS

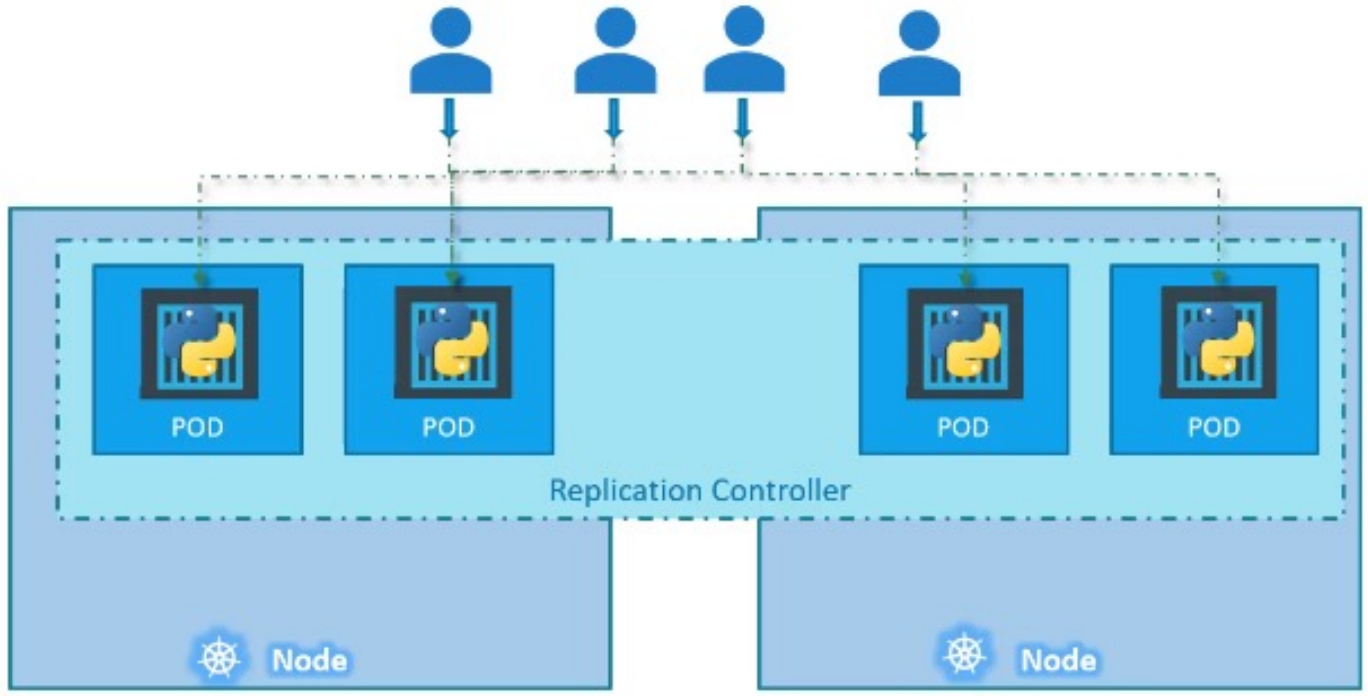
```
kubectl create –f pod-definition.yml

kubectl get pods

kubectl describe pod myapp-pod

kubectl delete –f pod-definition.yml
```

ReplicaSets (and Replication Controllers)

High Availability

ReplicaSets (and Replication Controllers)

Load Balancing & Scaling



Replication Controller

rc-definition.yml

```yaml
apiVersion: v1
kind: ReplicationController
metadata:
  name: myapp-rc
  labels:
      app: myapp
      type: front-end
spec:
  template:
    metadata:
      name: myapp-pod
      labels:
        app: myapp
        type: front-end
    spec:
      containers:
      - name: nginx-container
        image: nginx

  replicas: 3
```

pod-definition.yml

```yaml
apiVersion: v1
kind: Pod

metadata:
  name: myapp-pod
  labels:
    app: myapp
    type: front-end
spec:
  containers:
  - name: nginx-container
    image: nginx
```

```
> kubectl create -f rc-definition.yml
replicationcontroller "myapp-rc" created
```

```
> kubectl get replicationcontroller
NAME       DESIRED   CURRENT   READY     AGE
myapp-rc   3         3         3         19s
```

```
> kubectl get pods
NAME             READY     STATUS    RESTARTS   AGE
myapp-rc-4lvk9   1/1       Running   0          20s
myapp-rc-mc2mf   1/1       Running   0          20s
myapp-rc-px9pz   1/1       Running   0          20s
```

ReplicaSet

replicaset-definition.yml

```yaml
apiVersion: apps/v1
kind: ReplicaSet
metadata:
  name: myapp-replicaset
  labels:
      app: myapp
      type: front-end
spec:
  template:
    metadata:
      name: myapp-pod
      labels:
        app: myapp
        type: front-end
    spec:
      containers:
      - name: nginx-container
        image: nginx

  replicas: 3
  selector:
    matchLabels:
      type: front-end
```

pod-definition.yml

```yaml
apiVersion: v1
kind: Pod

metadata:
 name: myapp-pod
 labels:
    app: myapp
    type: front-end
spec:
  containers:
  - name: nginx-container
    image: nginx
```

```
> kubectl create -f replicaset-definition.yml
replicaset "myapp-replicaset" deleted
```

```
> kubectl get replicaset
NAME              DESIRED   CURRENT   READY     AGE
myapp-replicaset  3         3         3         19s
```

```
> kubectl get pods
NAME                      READY     STATUS    RESTARTS   AGE
myapp-replicaset-9ddl9    1/1       Running   0          45s
myapp-replicaset-9jtpx    1/1       Running   0          45s
myapp-replicaset-hq84m    1/1       Running   0          45s
```

ReplicaSet: Scale

```
> kubectl replace -f replicaset-definition.yml
```

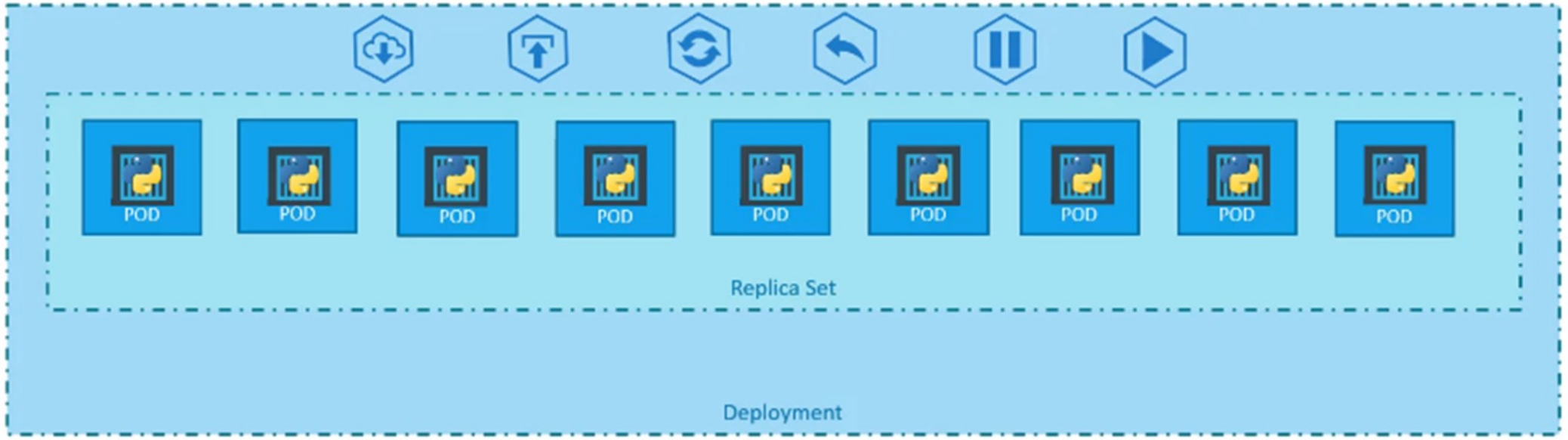
```
> kubectl scale --replicas=6 -f replicaset-definition.yml
```

```
> kubectl scale --replicas=6 replicaset myapp-replicaset
```

replicaset-definition.yml

```yaml
apiVersion: apps/v1
kind: ReplicaSet
metadata:
  name: myapp-replicaset
  labels:
      app: myapp
      type: front-end
spec:
  template:
    metadata:
      name: myapp-pod
      labels:
        app: myapp
        type: front-end
    spec:
      containers:
      - name: nginx-container
        image: nginx
  replicas: 6
  selector:
    matchLabels:
      type: front-end
```

Deployment



Deployment

```
> kubectl create -f deployment-definition.yml
deployment "myapp-deployment" created
```

```
> kubectl get deployments
NAME                DESIRED   CURRENT   UP-TO-DATE   AVAILABLE   AGE
myapp-deployment    3         3         3            3           21s
```

```
> kubectl get replicaset
NAME                           DESIRED   CURRENT   READY     AGE
myapp-deployment-6795844b58    3         3         3         2m
```

```
> kubectl get pods
NAME                                READY     STATUS    RESTARTS   AGE
myapp-deployment-6795844b58-5rbjl   1/1       Running   0          2m
myapp-deployment-6795844b58-h4w55   1/1       Running   0          2m
myapp-deployment-6795844b58-lfjhv   1/1       Running   0          2m
```

deployment-definition.yml

```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: myapp-deployment
  labels:
      app: myapp
      type: front-end
spec:
  template:
    metadata:
      name: myapp-pod
      labels:
        app: myapp
        type: front-end
    spec:
      containers:
      - name: nginx-container
        image: nginx
  replicas: 3
  selector:
    matchLabels:
      type: front-end
```

Rollout and Versioning

```
> kubectl rollout status deployment/myapp-deployment
Waiting for rollout to finish: 0 of 10 updated replicas are available...
Waiting for rollout to finish: 1 of 10 updated replicas are available...
Waiting for rollout to finish: 2 of 10 updated replicas are available...
Waiting for rollout to finish: 3 of 10 updated replicas are available...
Waiting for rollout to finish: 4 of 10 updated replicas are available...
Waiting for rollout to finish: 5 of 10 updated replicas are available...
Waiting for rollout to finish: 6 of 10 updated replicas are available...
Waiting for rollout to finish: 7 of 10 updated replicas are available...
Waiting for rollout to finish: 8 of 10 updated replicas are available...
Waiting for rollout to finish: 9 of 10 updated replicas are available...
deployment "myapp-deployment" successfully rolled out
```

```
> kubectl rollout history deployment/myapp-deployment
deployments "myapp-deployment"
REVISION  CHANGE-CAUSE
1         <none>
2         kubectl apply --filename=deployment-definition.yml --record=true
```

Deployment Strategy

Deployment: Apply

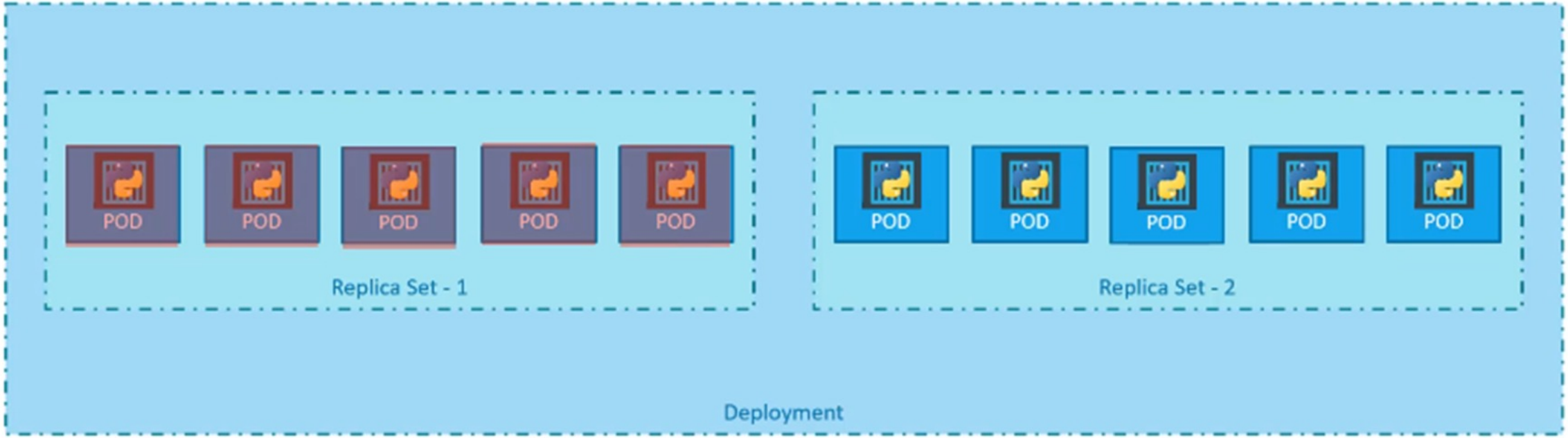
```
> kubectl apply -f deployment-definition.yml
deployment "myapp-deployment" configured
```

```
> kubectl set image deployment/myapp-deployment \
                  nginx=nginx:1.9.1
deployment "myapp-deployment" image is updated
```

deployment-definition.yml

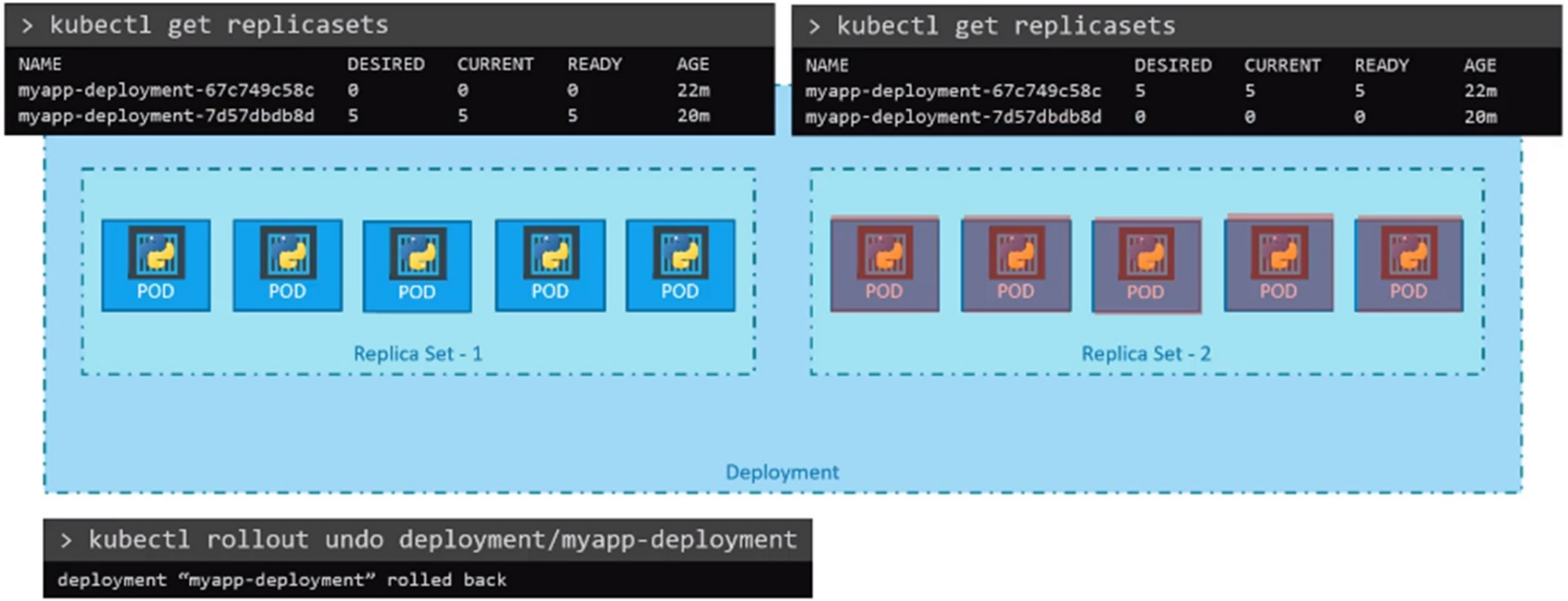
```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: myapp-deployment
  labels:
      app: myapp
      type: front-end
spec:
  template:
    metadata:
      name: myapp-pod
      labels:
         app: myapp
         type: front-end
    spec:
      containers:
      - name: nginx-container
        image: nginx:1.7.1
  replicas: 3
  selector:
    matchLabels:
       type: front-end
```

Deployment: Upgrade




```
> kubectl get replicasets
NAME                          DESIRED   CURRENT   READY   AGE
myapp-deployment-67c749c58c   0         0         0       22m
myapp-deployment-7d57dbdb8d   5         5         5       20m
```

Deployment: Rollback

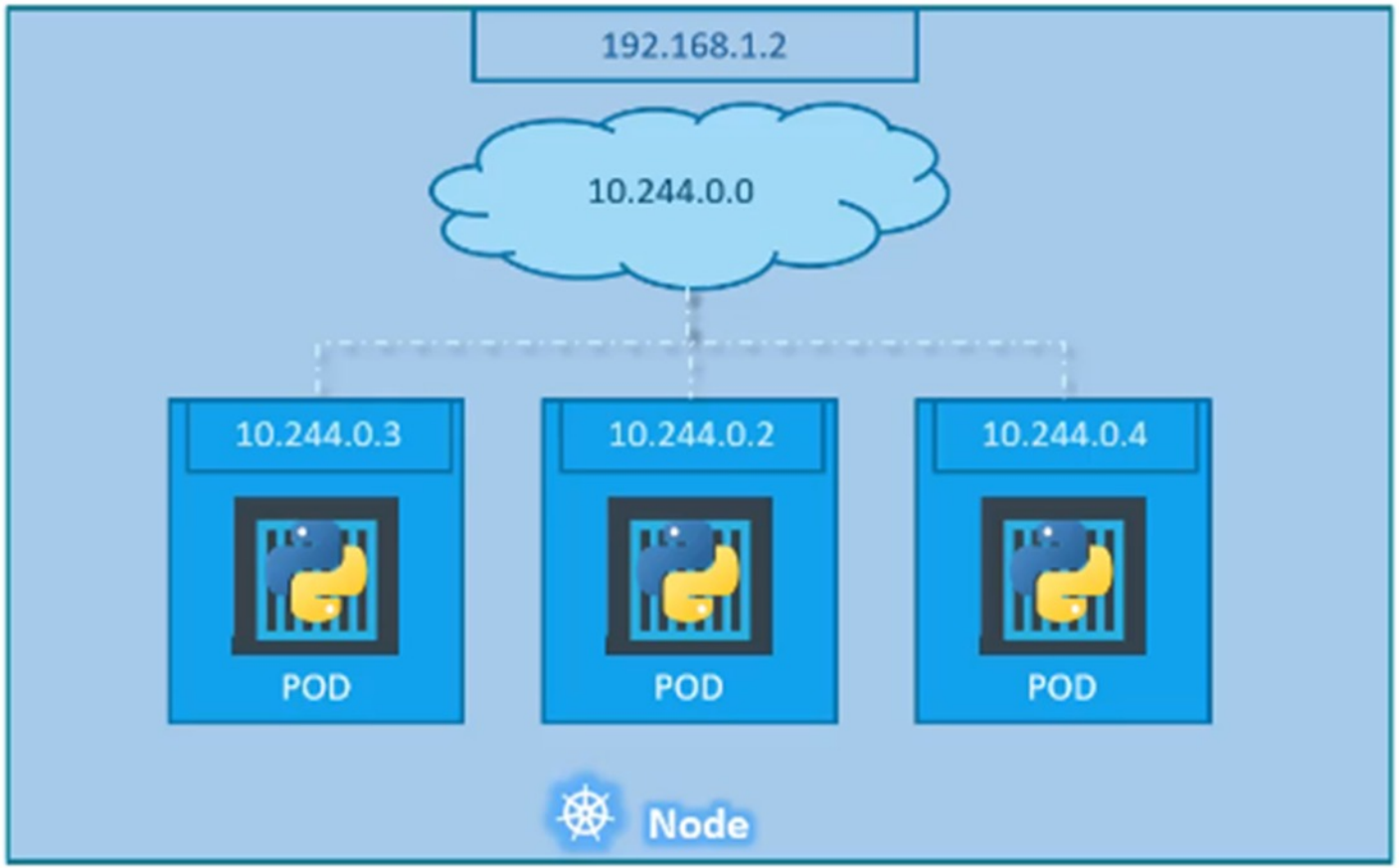


Lab: Deployments

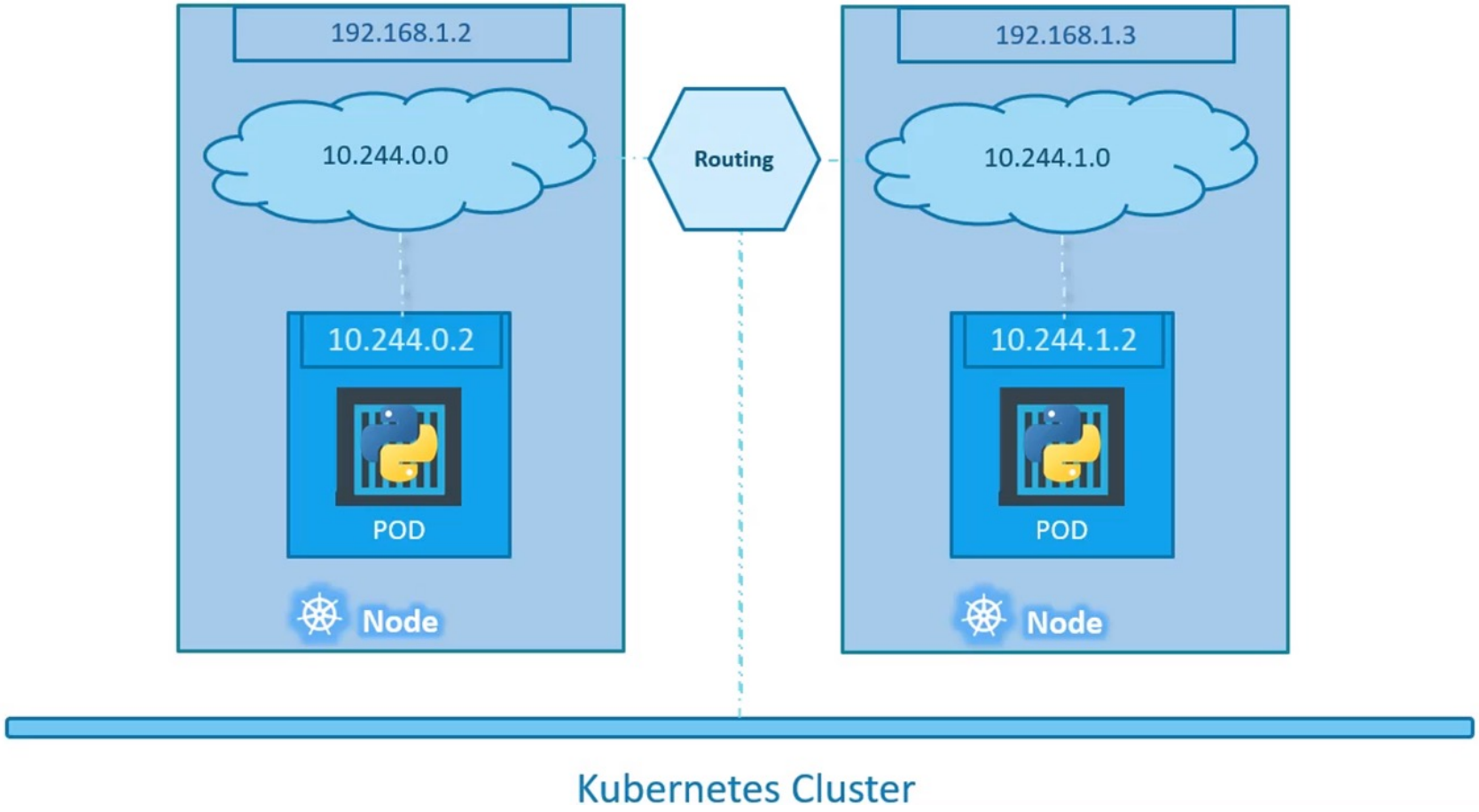
1) Create your deployment.
2) Upgrade the application to an other version.
3) Rollback the application.

Networking in Kubernetes

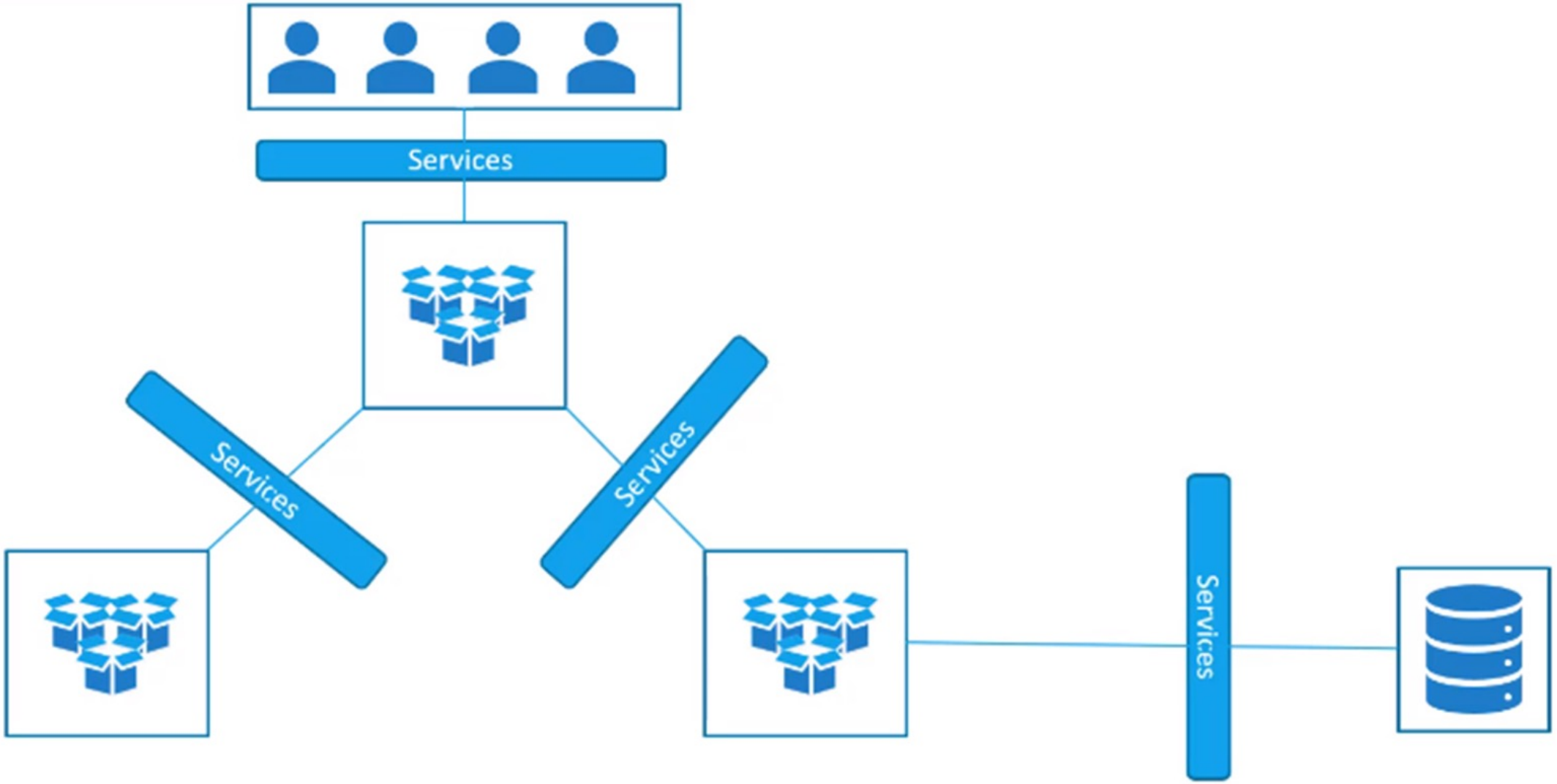
Single Node



Cluster Networking



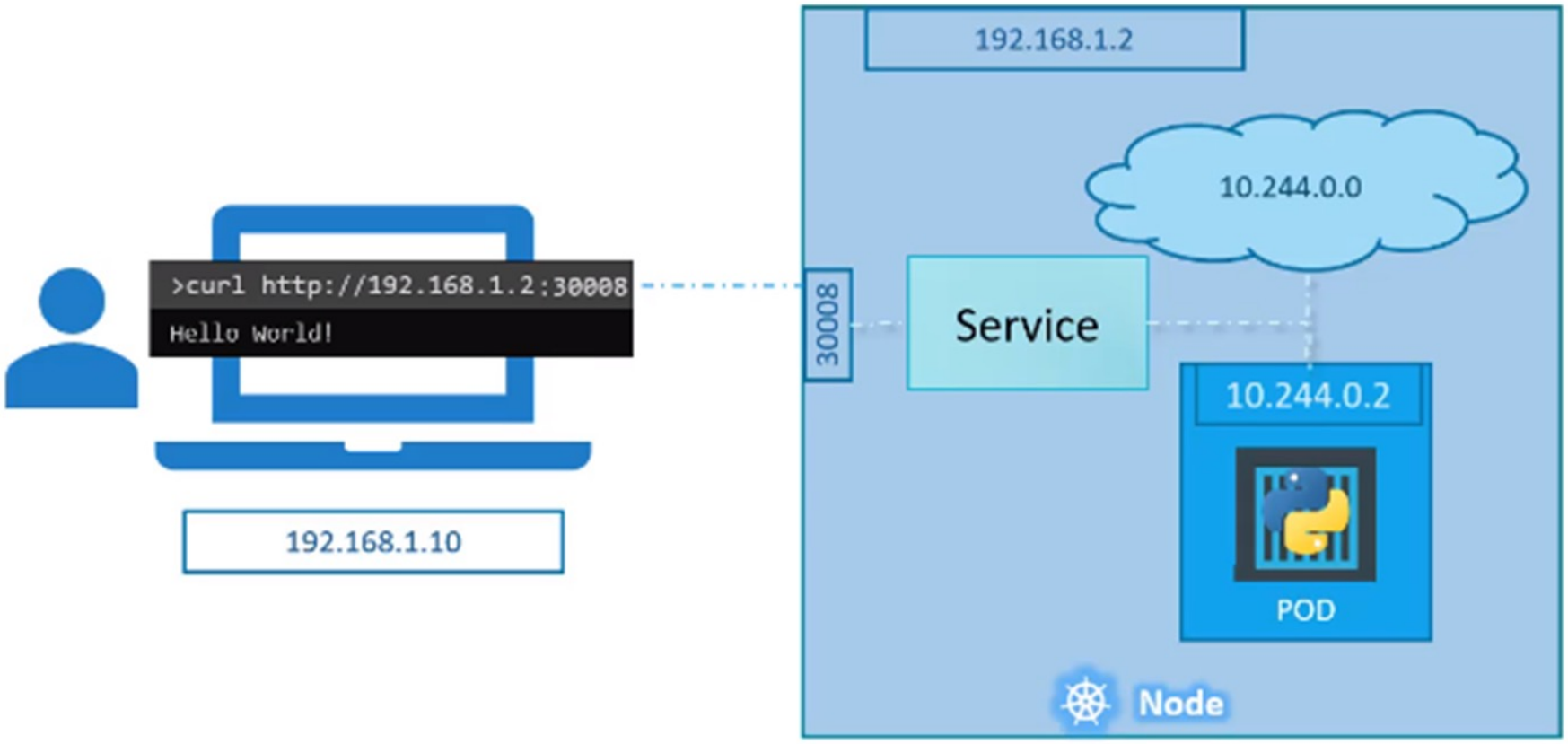
Services



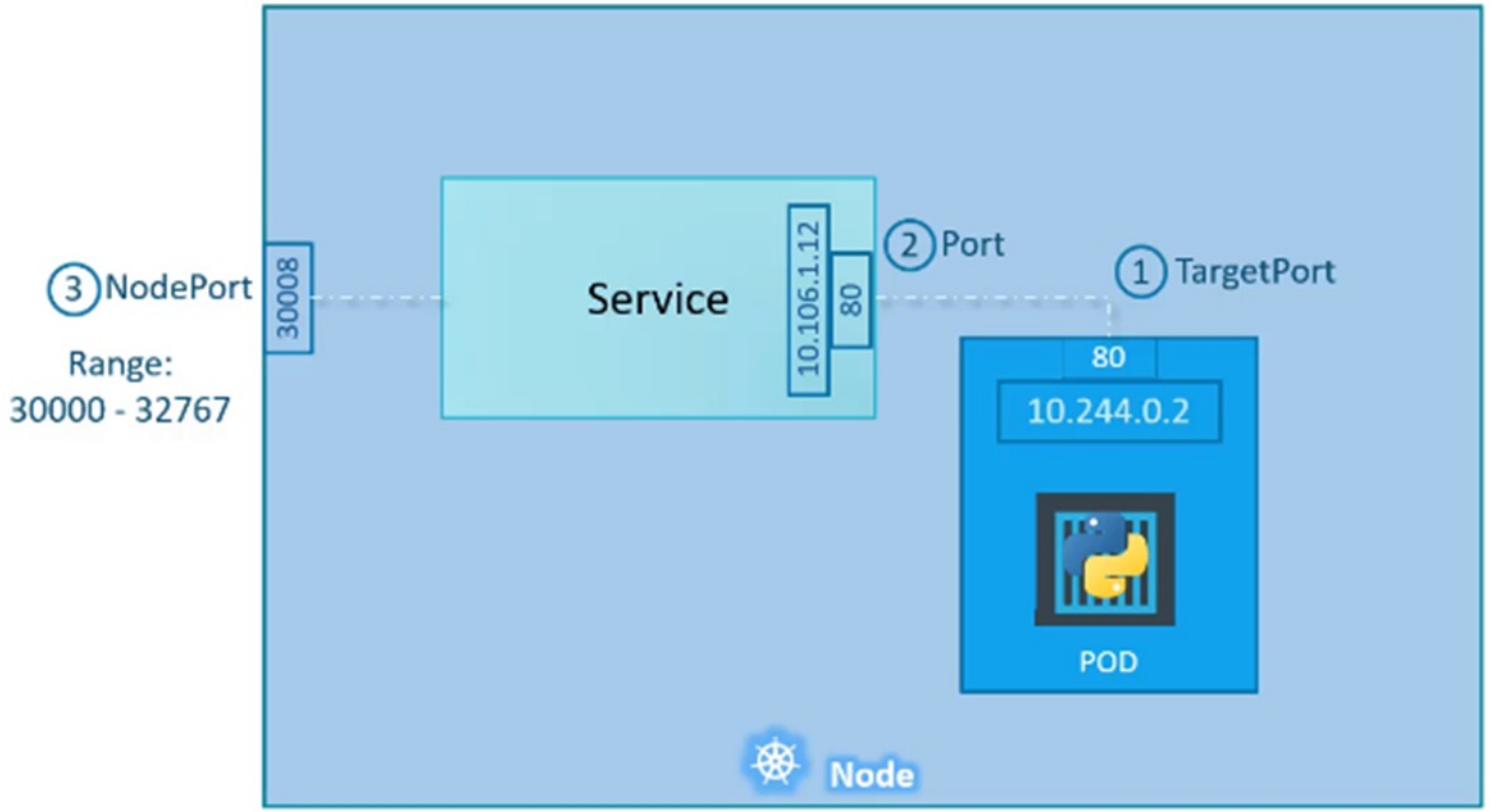
Services Types

- **NodePort**: Exposes the Service on each Node's IP at a static port (the NodePort). A ClusterIP Service, to which the NodePort Service routes, is automatically created.
- **ClusterIP**: Exposes the Service on a cluster-internal IP. Choosing this value makes the Service only reachable from within the cluster.
- **LoadBalancer**: Exposes the Service externally using a cloud provider's load balancer.
- **ExternalName**: Maps the Service to the contents of the externalName field (e.g. foo.bar.example.com), by returning a CNAME record with its value.

Service: NodePort



Service: NodePort



service-definition.yml

```yaml
apiVersion: v1
kind: Service
metadata:
    name: myapp-service

spec:
    type: NodePort
    ports:
     - targetPort: 80
       port: 80
       nodePort: 30008
    selector:
       app: myapp
       type: front-end
```

Service Definition: NodePort

service-definition.yml

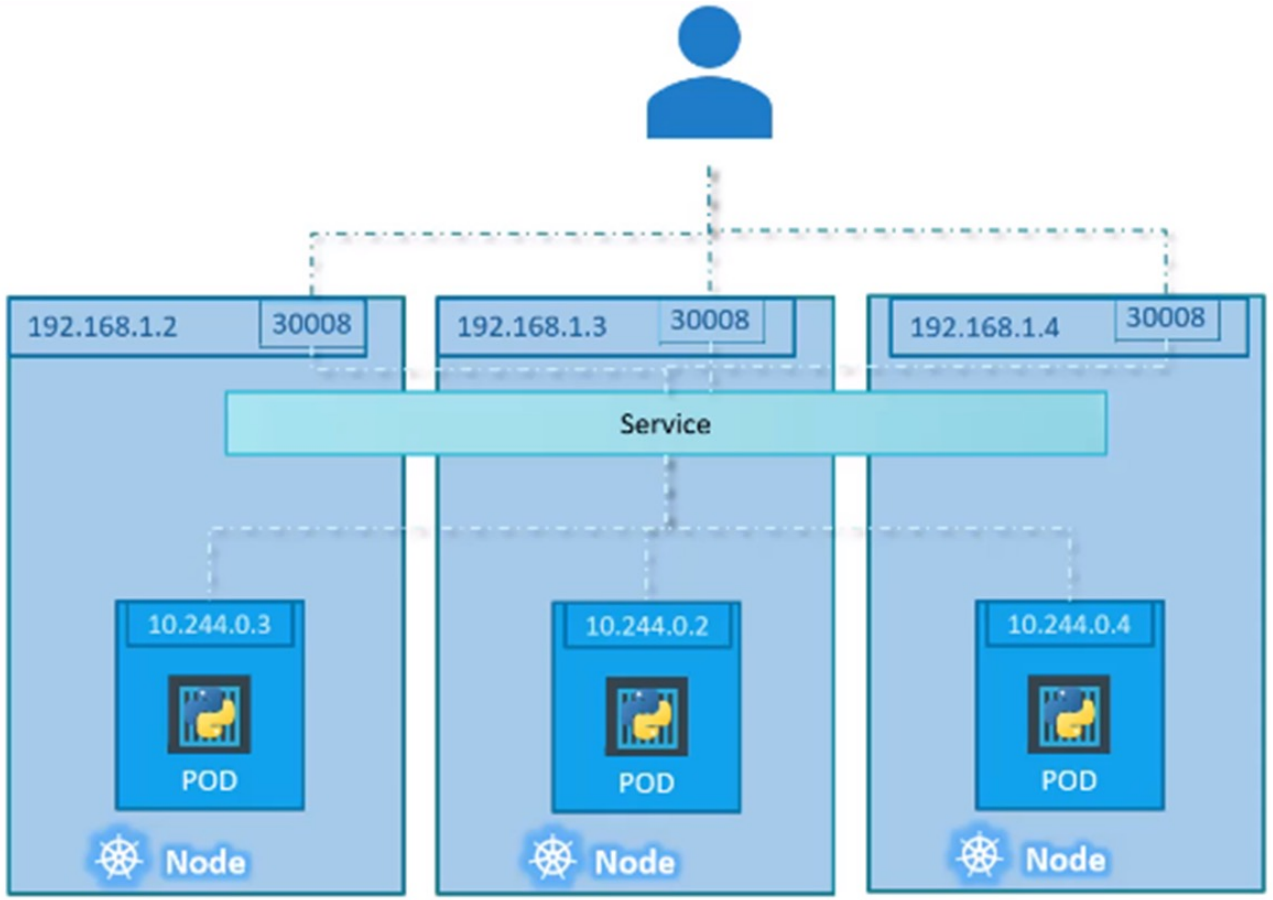
```yaml
apiVersion: v1
kind: Service
metadata:
    name: myapp-service

spec:
    type: NodePort
    ports:
     - targetPort: 80
       port: 80
       nodePort: 30008
    selector:
       app: myapp
       type: front-end
```

```
> kubectl create -f service-definition.yml
service "myapp-service" created
```

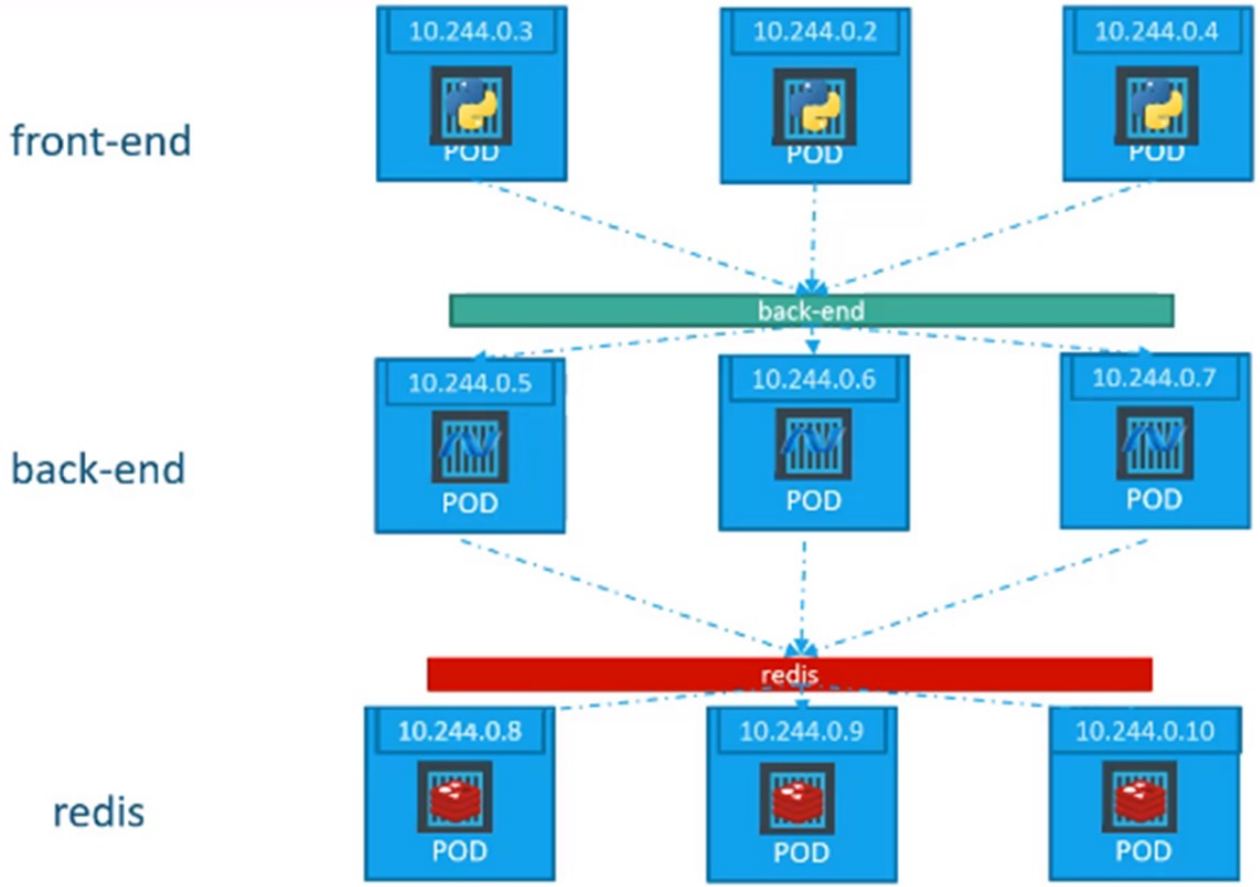
```
> kubectl get services
NAME            TYPE        CLUSTER-IP       EXTERNAL-IP   PORT(S)        AGE
kubernetes      ClusterIP   10.96.0.1        <none>        443/TCP        16d
myapp-service   NodePort    10.106.127.123   <none>        80:30008/TCP   5m
```

```
> curl http://192.168.1.2:30008
<html>
<head>
<title>Welcome to nginx!</title>
<style>
    body {
        width: 35em;
        margin: 0 auto;
        font-family: Tahoma, Verdana, Arial, sans-serif;
    }
</style>
</head>
<body>
```

Service: NodePort



Service: ClusterIP



Service: ClusterIP

service-definition.yml

```yaml
apiVersion: v1
kind: Service
metadata:
    name: back-end

spec:
    type: ClusterIP
    ports:
     - targetPort: 80
       port: 80

    selector:
       app: myapp
       type: back-end
```

```
> kubectl create -f service-definition.yml
service "back-end" created
```

```
> kubectl get services
NAME         TYPE        CLUSTER-IP       EXTERNAL-IP   PORT(S)    AGE
kubernetes   ClusterIP   10.96.0.1        <none>        443/TCP    16d
back-end     ClusterIP   10.106.127.123   <none>        80/TCP     2m
```

Lab: Services

1) Create a NodePort service for your deployment

2) Check the service from outside of the cluster.

Kubernetes Web UI (Dashboard)

Dashboard is a web-based Kubernetes user interface. You can use Dashboard to deploy containerized applications to a Kubernetes cluster, troubleshoot your containerized application, and manage the cluster resources.

Ref.: https://github.com/kubernetes/dashboard

Lab: Install Kubernetes Web UI

1) Install Kubernetes Web UI
   https://github.com/kubernetes/dashboard

2) Create a sample user and get the token

3) Change the service type from ClusterIP to NodePort
   https://github.com/kubernetes/dashboard/tree/master/docs/user/accessing-dashboard

Kubernetes Metrics server

Metrics server is responsible for collecting resource metrics from kubelets and exposing them in Kubernetes Apiserver through Metrics API.

Ref.: https://github.com/kubernetes-sigs/metrics-server

Lab: Install Kubernetes Metrics server

1) Install Kubernetes Metrics server
   https://github.com/kubernetes-sigs/metrics-server

2) Check Kubernetes Web UI

Kubernetes + Compose = Kompose

Kompose is a conversion tool for all things compose (namely Docker Compose) to container orchestrators.

- kompose convert
- kompose up
- kompose down

Ref.: https://kubernetes.io/docs/tasks/configure-pod-container/translate-compose-kubernetes
Ref.: http://kompose.io