Docker, Kubernetes and Microservices Training - Open Source

Syed Wasiq Muhammad
Principal Consultant Development - Systems Limited
Cloud App Dev. & Mobility

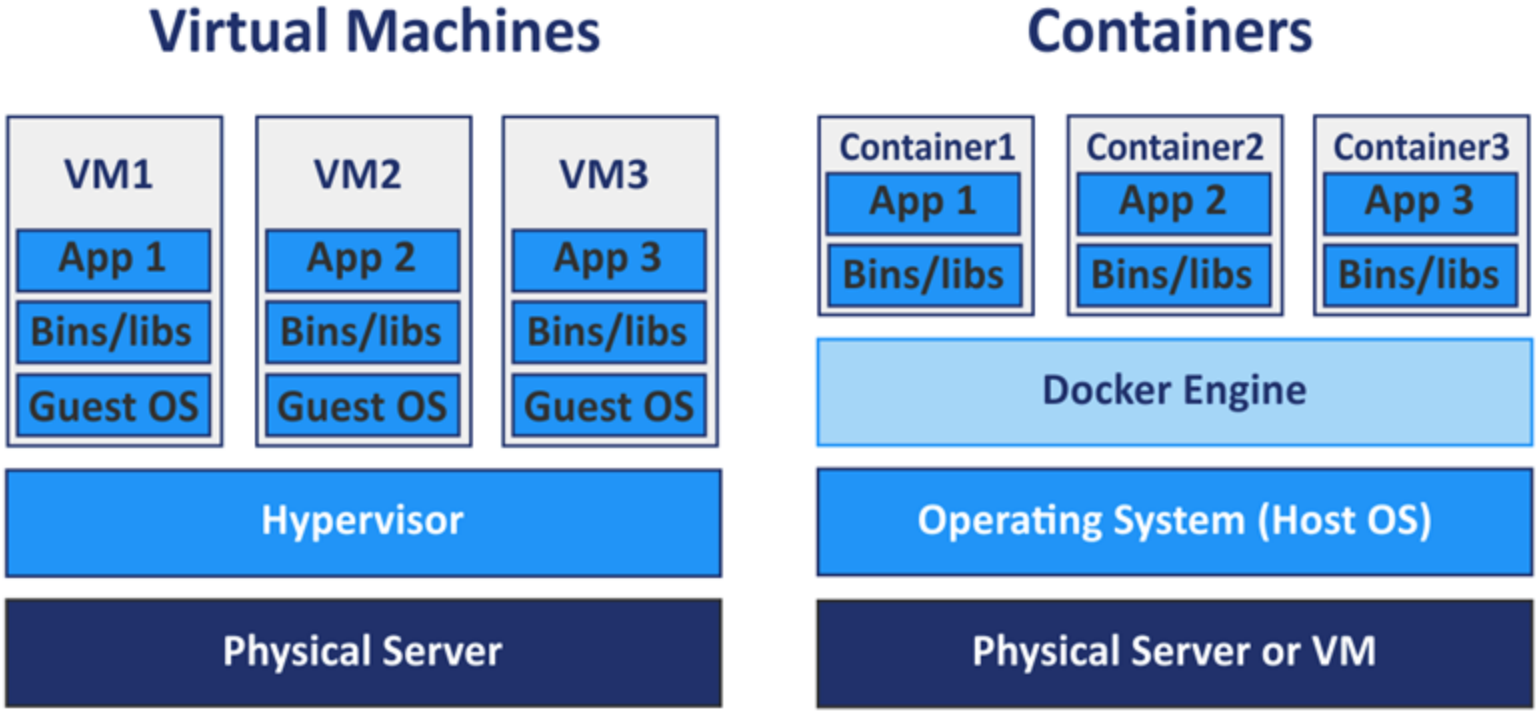
Ukasha Sohail
Jr Consultant - Systems Limited
Cloud App Dev. & Mobility

systems

Session # 4

- Overview
- Understanding Architecture and Components
- Replication Controller
- Replica Set
- Deployment
- Services
- Networking

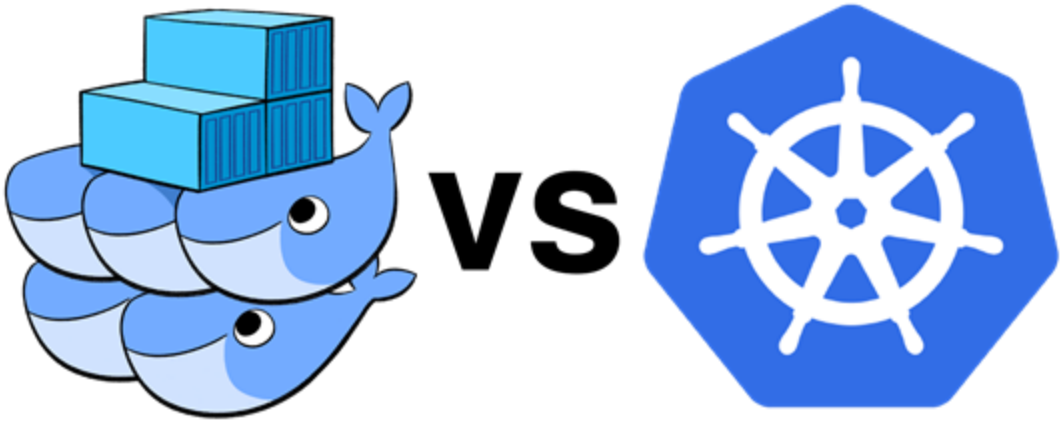
Why do you need Containers ?



Container Orchestration

The process of automatically deploying and managing containers is known as **Container Orchestration**

It automates tasks like deploying **containers**, **scaling** them up or down based on demand, **monitoring** their health, and ensuring they are always available and running efficiently.



Kubernetes

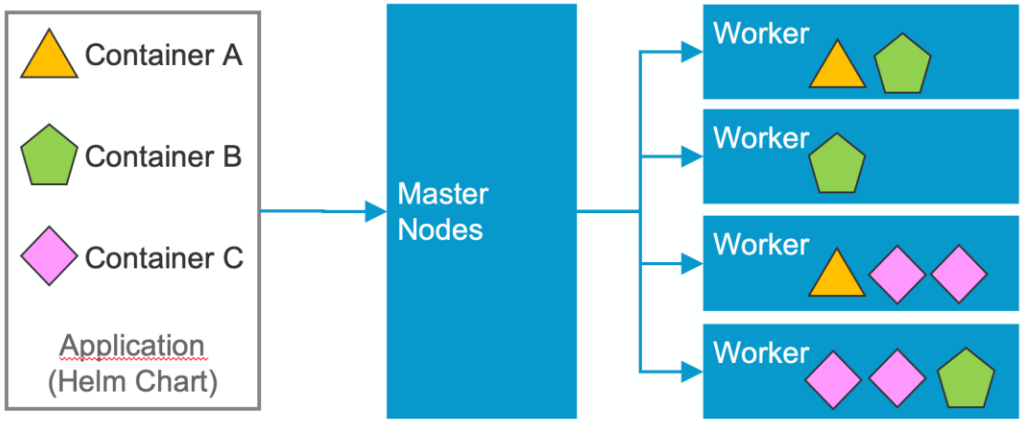
Kubernetes, often abbreviated as **K8s**, is an open-source container orchestration platform that was originally developed by **Google**. The project was started by Google engineers who were working on managing containers at a massive scale within their own infrastructure.

In 2014, Google open-sourced Kubernetes and donated it to the Cloud Native Computing Foundation (**CNCF**), a non-profit organization that fosters the development of cloud-native technologies.

Kubernetes is now supported on all public cloud service providers like **GCP**, **Azure** and **AWS** and the Kubernetes project is one of the top ranked projects in GitHub.

Distributed Architecture

Kubernetes operates in a **distributed manner**, meaning it can manage containers and applications across multiple **nodes** or **machines**. Each node in the Kubernetes cluster runs the Kubernetes runtime, which enables it to communicate and collaborate with other nodes.



CLUSTERS

A cluster is a set of nodes grouped together. This way even if one node fails you have your application still accessible from the other nodes. Moreover having multiple nodes helps in sharing load as well.

NODES

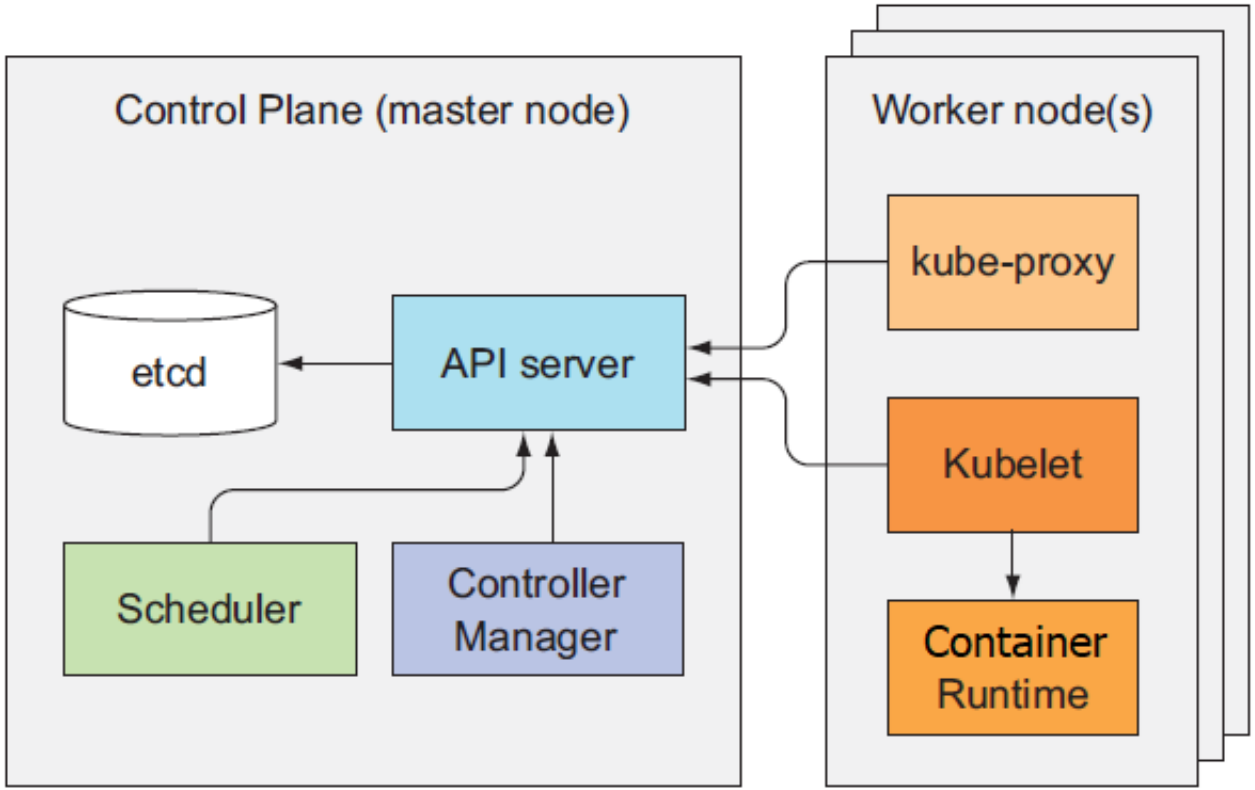
(MINIONS)

A node is a machine – physical or virtual – on which Kubernetes is installed. A node is a worker machine and this is were containers will be launched by Kubernetes.

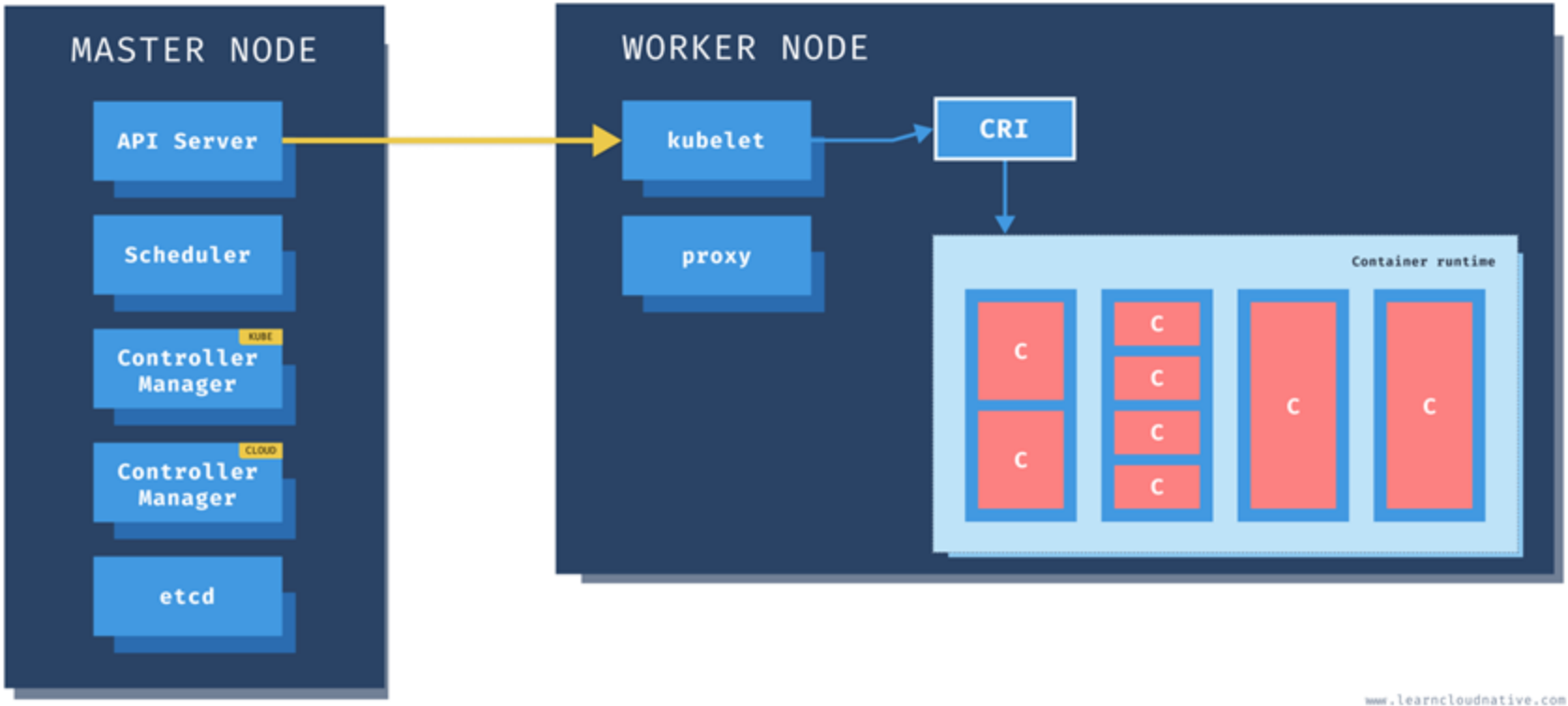
MASTER

The master is another node with Kubernetes installed in it, and is configured as a **Master**. The master watches over the nodes in the cluster and is responsible for the actual orchestration of containers on the worker nodes.

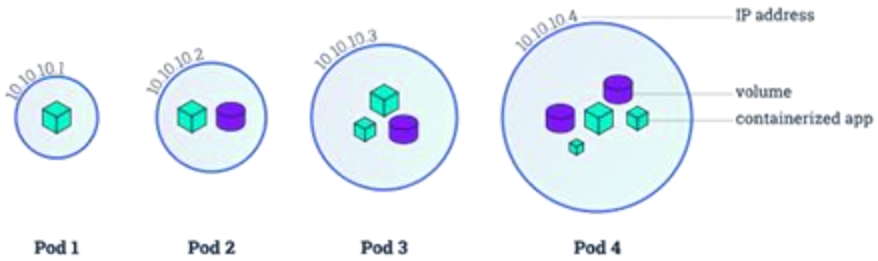
Components


Control Plane (master node)
Worker node(s)
kube-proxy
etcd
API server
Kubelet
Scheduler
Controller Manager
Container Runtime

Kubernetes


MASTER NODE
API Server
Scheduler
KUBE
Controller Manager
CLOUD
Controller Manager
etcd
WORKER NODE
kubelet
proxy
CRI
Container runtime
C
C
C
C
C
C
C
C
www.learncloudnative.com

POD



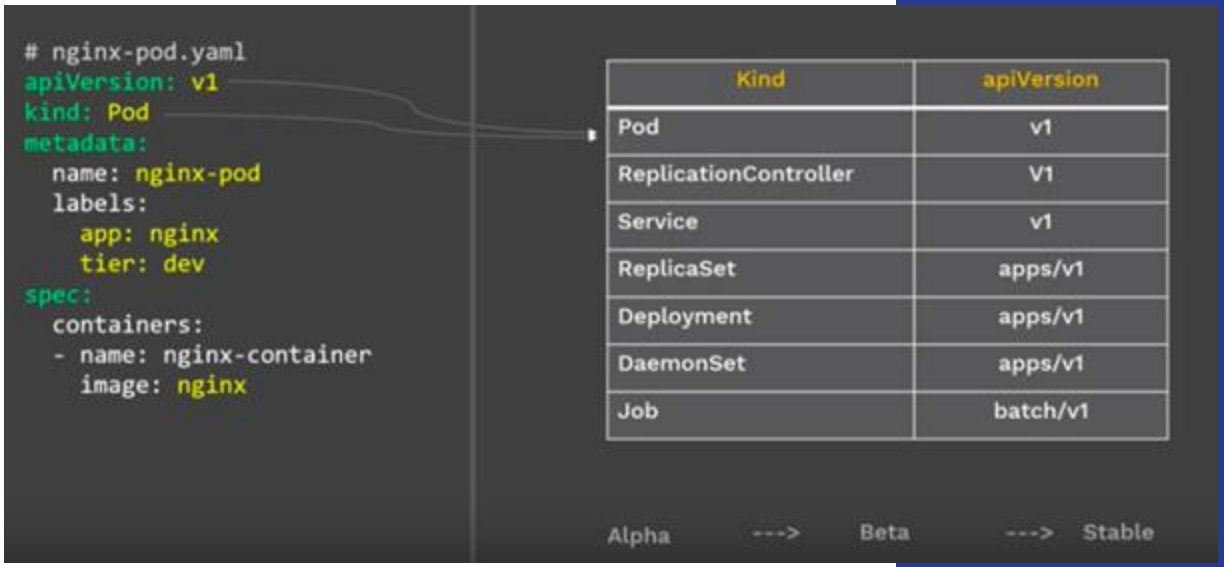
Kubernetes does not deploy containers directly on the worker nodes.

The containers are encapsulated into a Kubernetes object known as **PODs**.

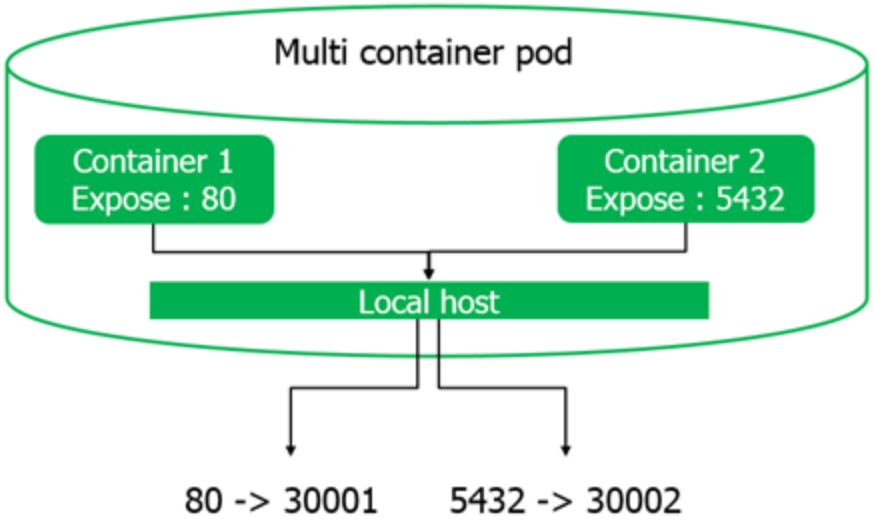
A POD is a single instance of an application. A POD is the smallest object, that you can create in Kubernetes.

POD -YAML

kubectl get pods
kubectl name nginx --image nginx
kubectl describe pod <name>

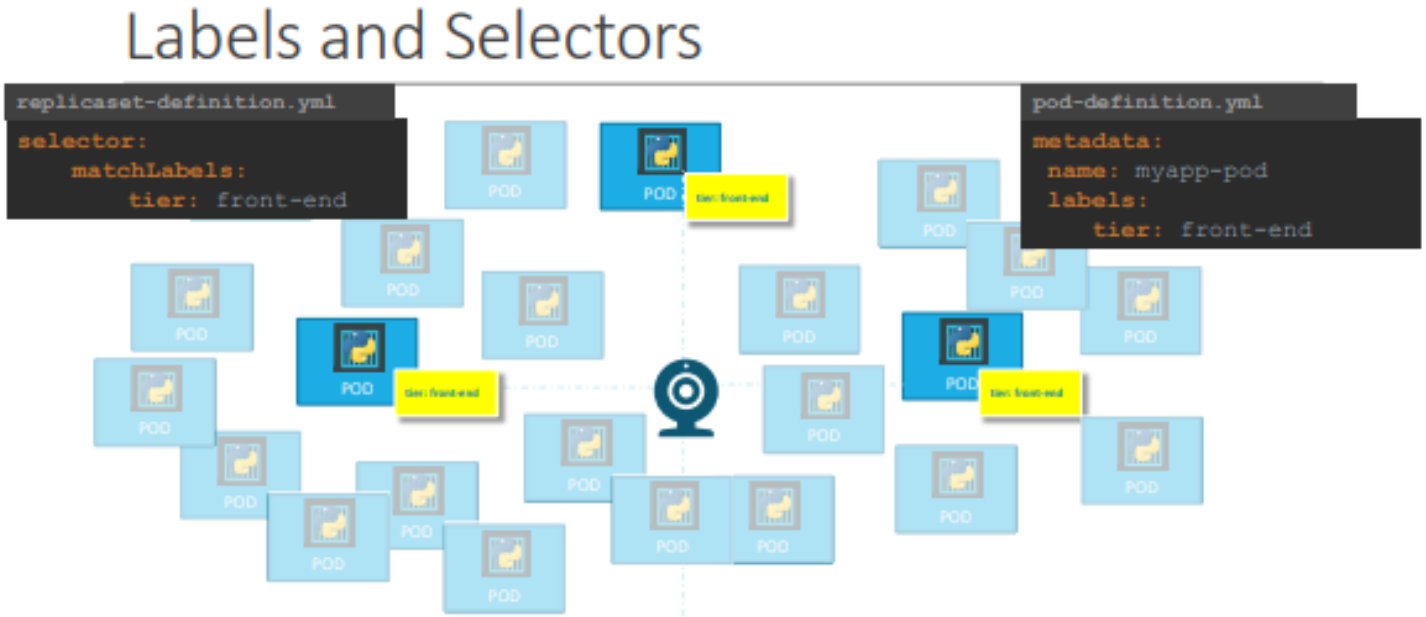


MULTI CONTAINER-POD

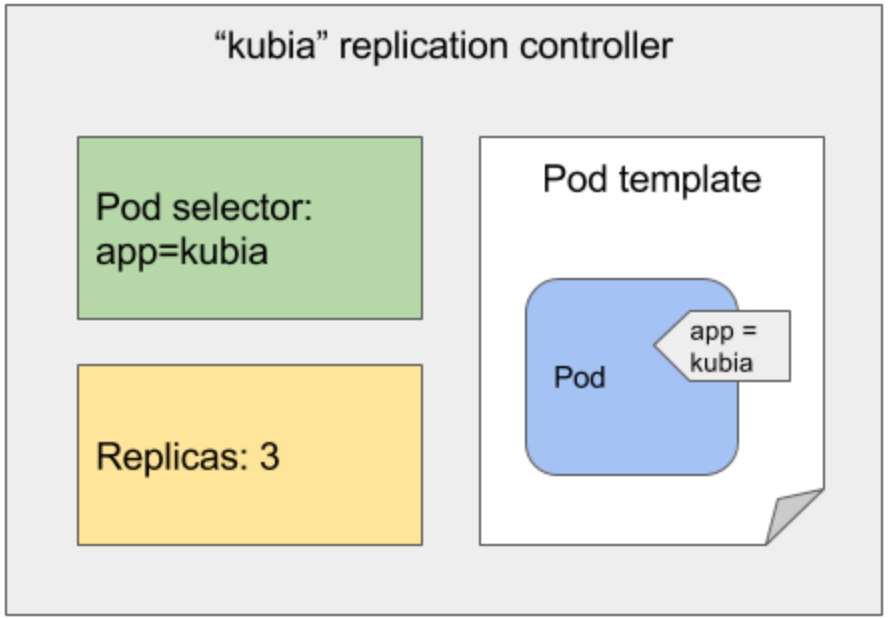


We can run multiple containers on a single pod. But the best practice is you have one container for one pod for application scalability

Why **Label** Or **Selector** ?

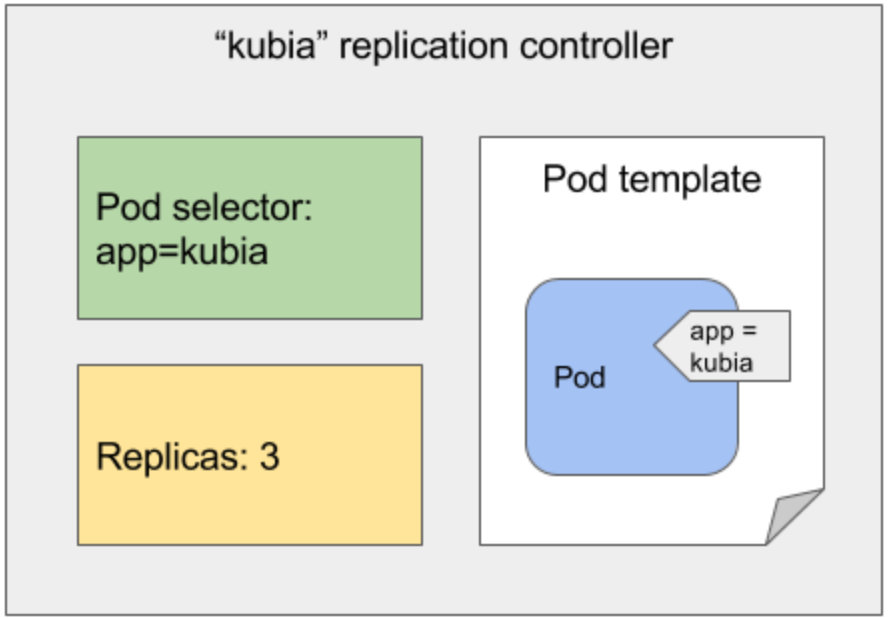


REPLICATION CONTROLLER



Replication Controller is a resource object that ensures a specified number of pod replicas are running at all times. It is one of the core controllers in Kubernetes and helps in maintaining the desired state of a pod set, automatically scaling the number of replicas up or down as needed.

REPLICATION CONTROLLER



we would like to have more than one instance or **POD** running at the same time.

That way if one fails we still have our application running on the other one.

The replication controller helps us run **multiple** instances of a single **POD** in the Kubernetes cluster thus providing **High Availability**

REPLICA SET

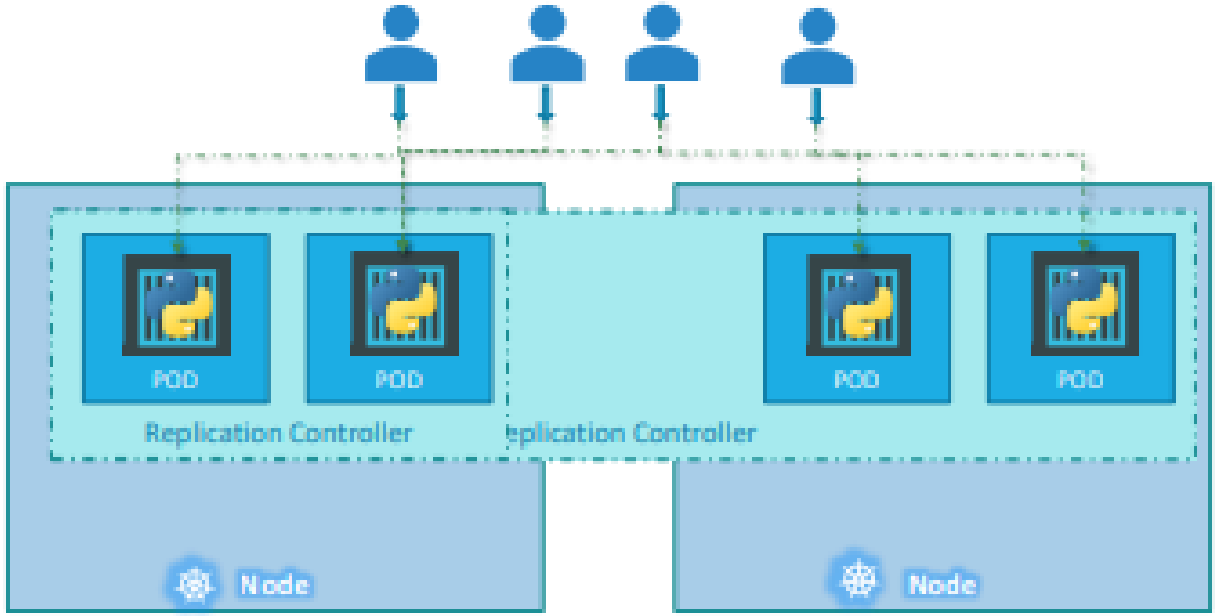
Difference between Controller and Replica SET

- POD before Creation
- Selector Label
- Consistent
- The selector is not a **REQUIRED** field in case of a replication controller

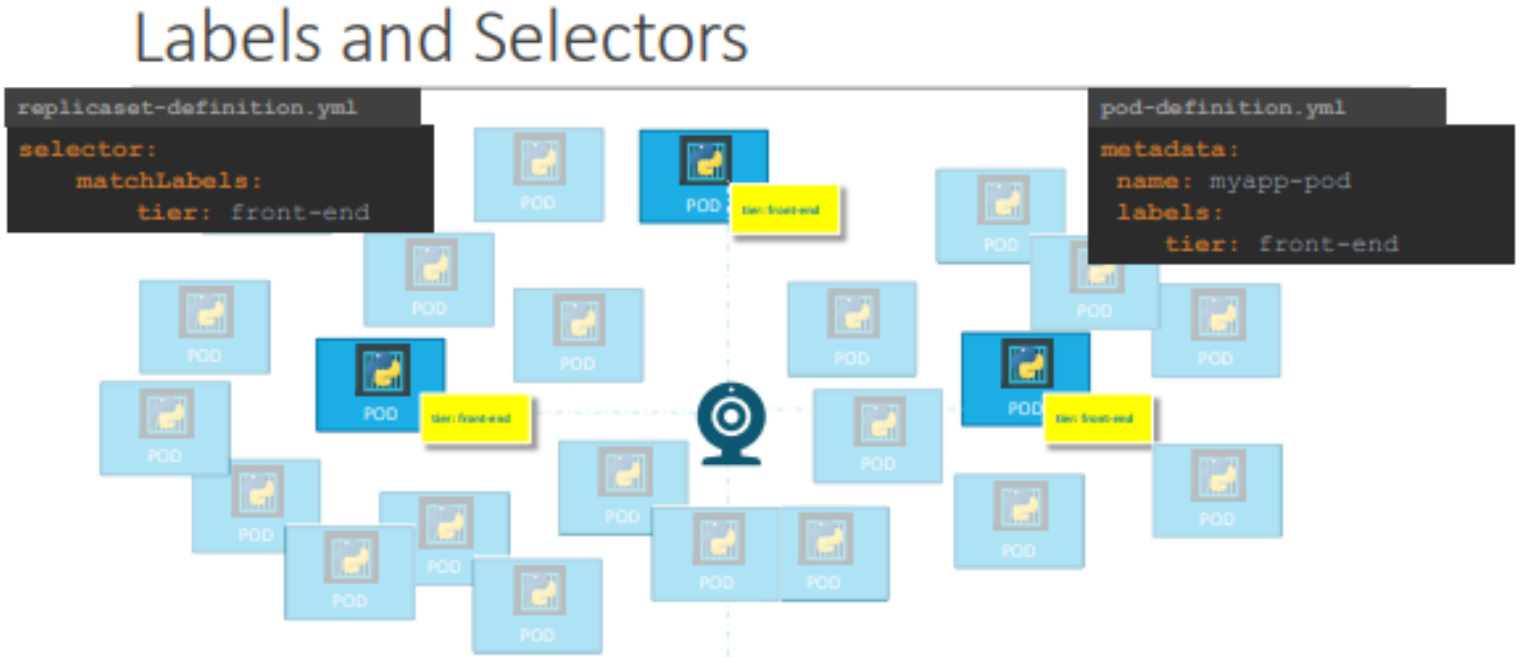
```yaml
apiVersion: apps/v1
kind: ReplicaSet
metadata:
  name: apache-rs
  labels:
    app: demo1
spec:
  replicas: 2
  selector:
    matchLabels:
      app: demo1
    matchExpressions:
      - {key: app, operator: In, values: [demo1]}
  template:
    metadata:
      labels:
        app: demo1
    spec:
      containers:
      - name: webserver
        image: lakshminp/apache:v1
        ports:
        - name: http
          containerPort: 8080
      restartPolicy: Always

```

Load Balancing & Scaling



SCALE


Labels and Selectors
replicaset-definition.yml
selector:
matchLabels:
tier: front-end
pod-definition.yml
metadata:
name: myapp-pod
labels:
tier: front-end
POD

COMMANDS

Scale

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

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

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

TYPE NAME

replicaset-definition.yml

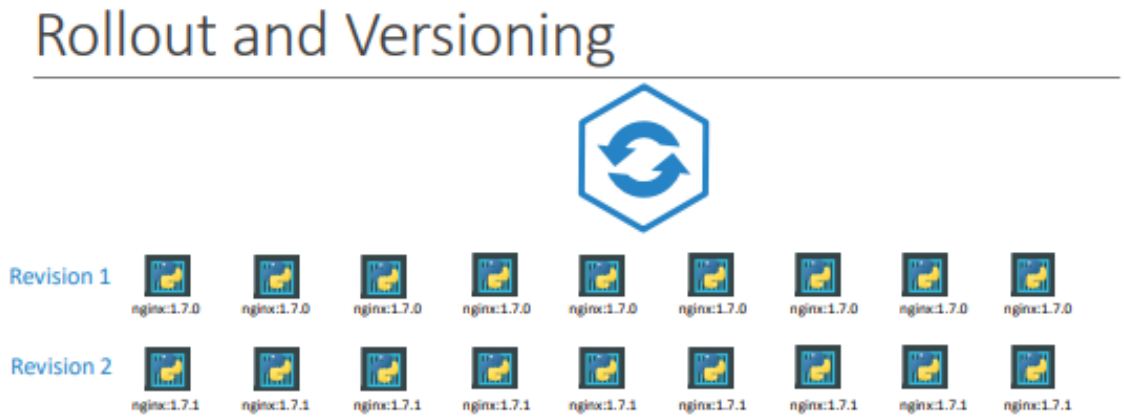
```yaml
apiVersion: apps/v1
kind: ReplicaSet
metadata:
  name: myapp-replicaset
  labels:
      app: myapp
      type: front-end
spec:
  template:
    metadata:
      name: myapp-pod
      labels:
        app: myapp
        type: front-end
    spec:
      containers:
      - name: nginx-container
  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

Deployments

Do not want to upgrade all of them at once

Rolling Updates
upgrade them one after the other. You would like to be able to rollBACK the changes that were recently carried out. Finally, say for example you would like to make multiple changes to your environment

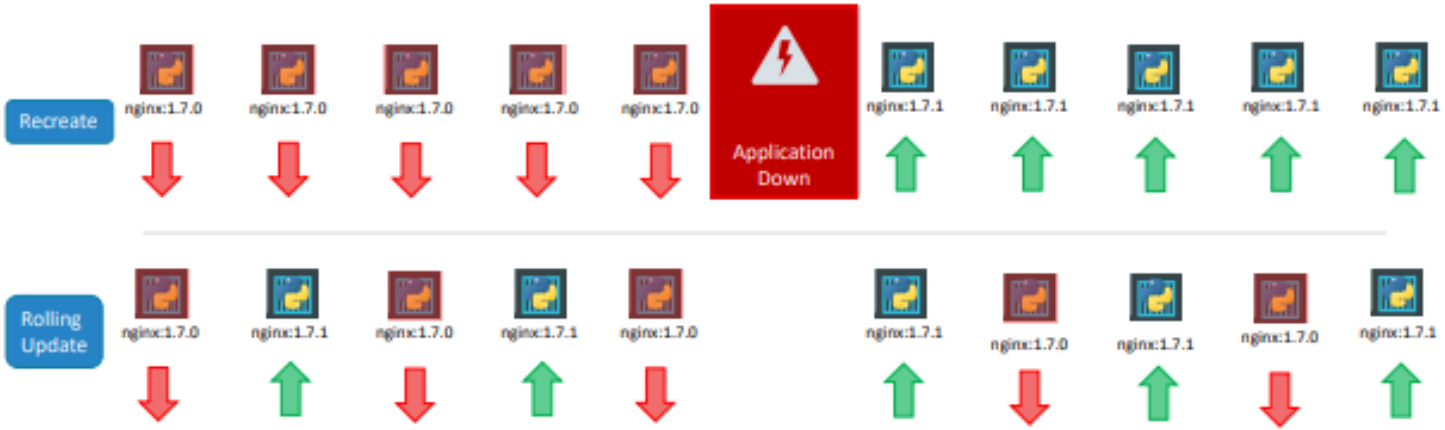
Deployment which is a kubernetes object that comes **higher** in the hierarchy.



Deployments Strategy

Recreate Strategy
Delete all pods
then create
Application
inaccessible

Rolling Updates
One by one update
It is the default
Deployment
Strategy



Summarize Commands

Create

```
> kubectl create -f deployment-definition.yml
```

Get

```
> kubectl get deployments
```

Update

```
> kubectl apply -f deployment-definition.yml
```

```
> kubectl set image deployment/myapp-deployment nginx=nginx:1.9.1
```

Status

```
> kubectl rollout status deployment/myapp-deployment
```

```
> kubectl rollout history deployment/myapp-deployment
```

Rollback

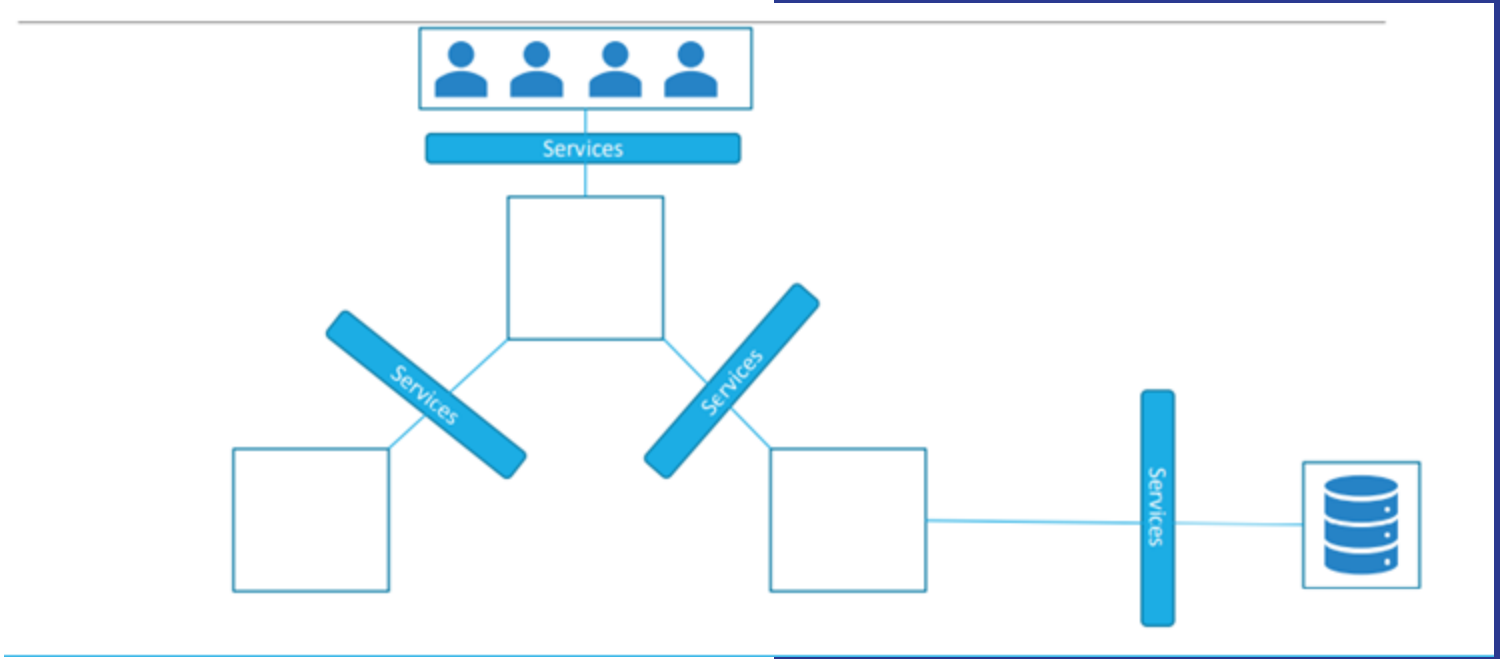
```
> kubectl rollout undo deployment/myapp-deployment
```

SERVICES

Service is an abstraction that provides a **consistent and reliable** way to expose an application running on a set of pods (replicas) to other services within the cluster or to external clients.

It acts as a **stable endpoint** to access your application, even if the underlying pods are scaled up, down, or moved around in the cluster.

SERVICES



SERVICES

1.Cluster IP: This is the default service type. It exposes the service on an internal IP address within the cluster, making it accessible only from within the cluster.

2.Node Port: This service type exposes the service on a static port on each node's IP address, allowing external clients to **access the service using the node's IP** and the Node Port.

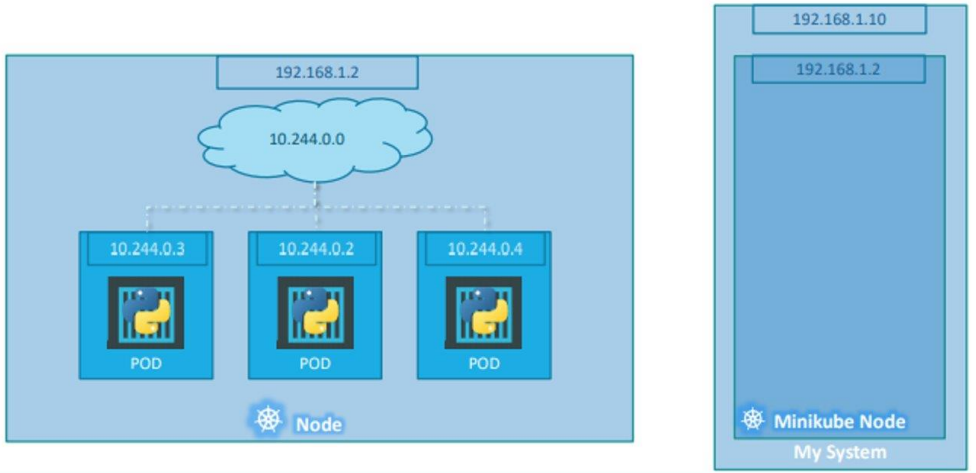
3.LoadBalancer: This service type provisions an external load balancer (supported by cloud providers) and assigns a fixed public IP to the service. This allows external clients to access the service through the load balancer.

4.ExternalName: This service type allows you to map a service to an external DNS name. It does not provide any load balancing or proxying; it simply provides a DNS alias to an external name.

Networking

Networking refers to how the pods and services within a **cluster communicate** with each other and with **external resources.** Kubernetes provides a robust networking model that allows seamless connectivity and communication between various components in the cluster.

- IP Address is assigned to a POD



Networking

Pod Networking: Pods are the smallest deployable units in Kubernetes. Each pod has its own unique IP address, which is reachable within the cluster. Containers within a pod share the same network namespace, enabling them to communicate with each other using **localhost**.

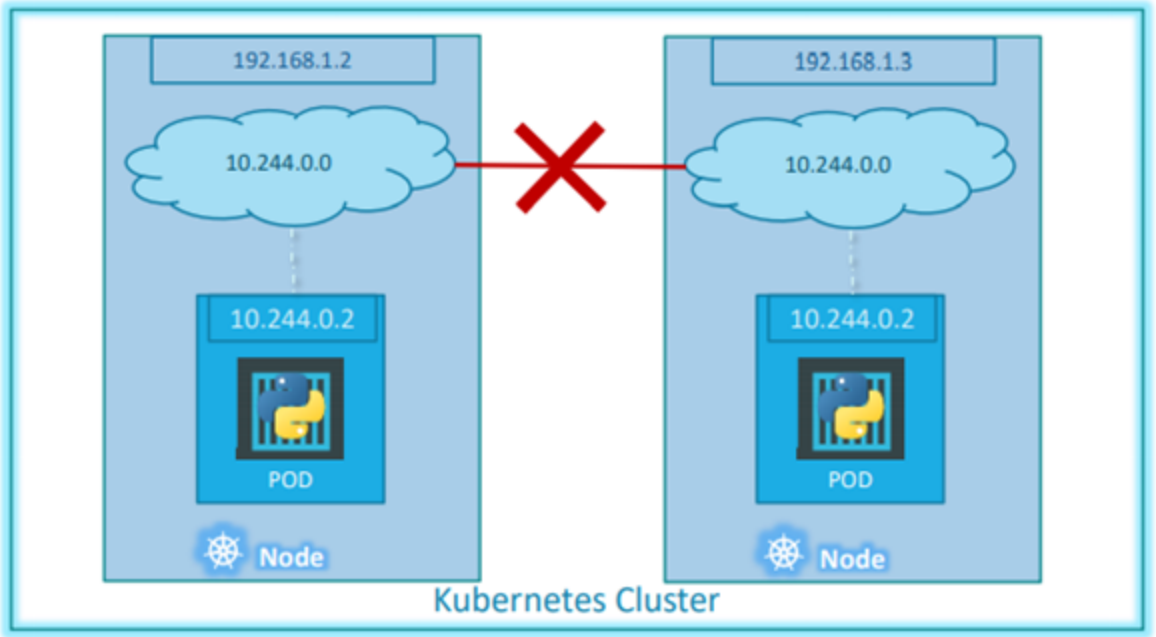
Service Networking: Services abstract away the underlying pods and provide a **stable IP address** and **DNS name** for accessing the pods. Services act as **load balancers**, distributing network traffic to the pods associated with the service. The service IP and DNS name are used by other services or external clients to communicate with the application.

Cluster Networking: Cluster networking ensures that pods and services can communicate across different nodes in the cluster. Kubernetes assigns a **unique IP subnet to each node,** and all the pods on a node can communicate with each other using this internal network. Additionally, the Kubernetes **control plane components** use cluster networking for communication and coordination.

Network Plugins: Kubernetes supports various network plugins, also known as Container Network Interfaces (CNI), which implement the networking and communication logic. These plugins handle tasks like setting up network interfaces, IP allocation, and routing between pods. Popular CNI plugins include Calico, Flannel, Weave, Cilium, and more.

Cluster Networking

- All containers/PODs can communicate to one another without NAT
- All nodes can communicate with all containers and vice-versa without NAT



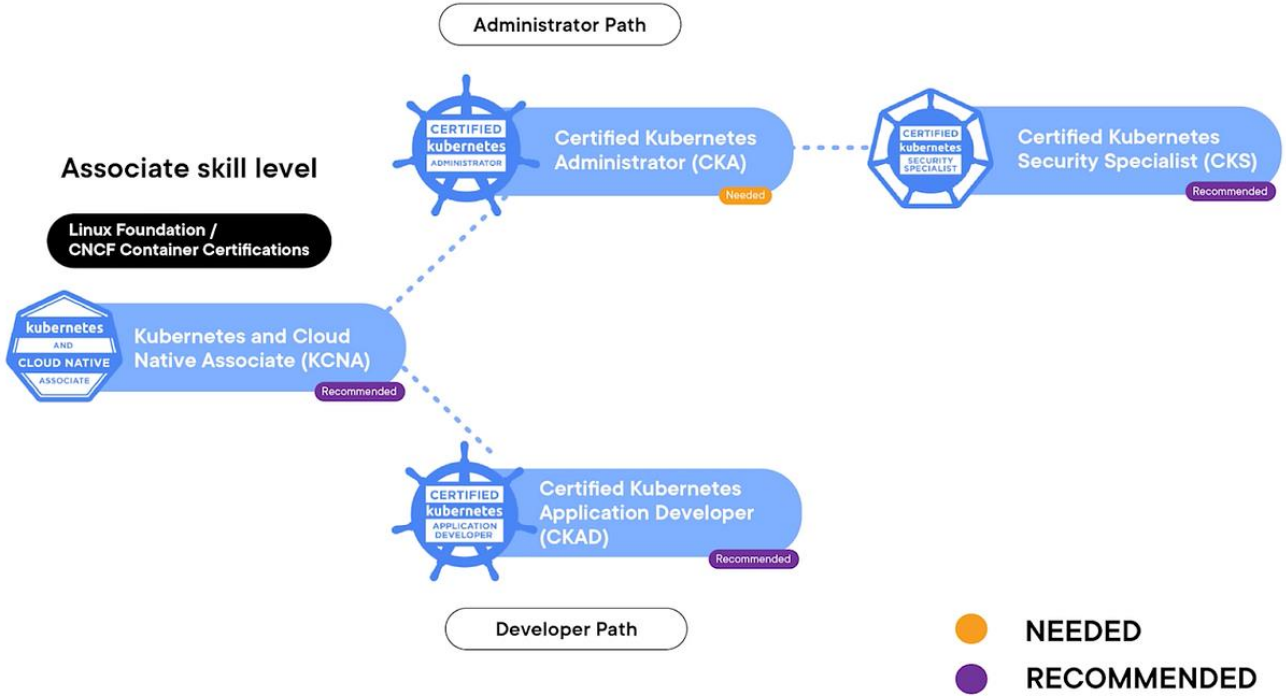

CISCO


cilium


flannel


vmware
NSX

Certification Track



Assignment # A

Add a microservice called **data-service** with no HTTP endpoint but just a MQ receiver.

Send payload received in POST requests of **billing-service, shipping-service** and **users-service** to **data-service** via MQ and further send all messages received by data-service to webhook-service using MQ.

Assignment # B

Objective: The objective of this assignment is to create a simple Redis pod and a Redis deployment in a Kubernetes cluster. The Redis pod will run a single instance of Redis, and the Redis deployment will ensure that the Redis application is continuously available by maintaining a specified number of replicas.

Instructions:

Create a Redis Pod:

- Create a YAML file named **redis-pod.yaml**.
- In the YAML file, define a Kubernetes Pod that runs a Redis container.
- Use the official Redis image from Docker Hub (**redis:latest**).
- Name the pod as **redis-pod**.
- Configure the container to listen on port **6379**.
- Add appropriate labels to the pod to identify it as a Redis pod.

Create a Redis Deployment:

- Create a YAML file named **redis-deployment.yaml**.
- In the YAML file, define a Kubernetes Deployment for Redis.
- Use the same Redis container image (**redis:latest**) as used in the previous step.
- Name the deployment as **redis-deployment**.
- Set the number of replicas to **3**.
- Configure the container to listen on port **6379**.
- Add appropriate labels to the deployment to identify it as a Redis deployment.