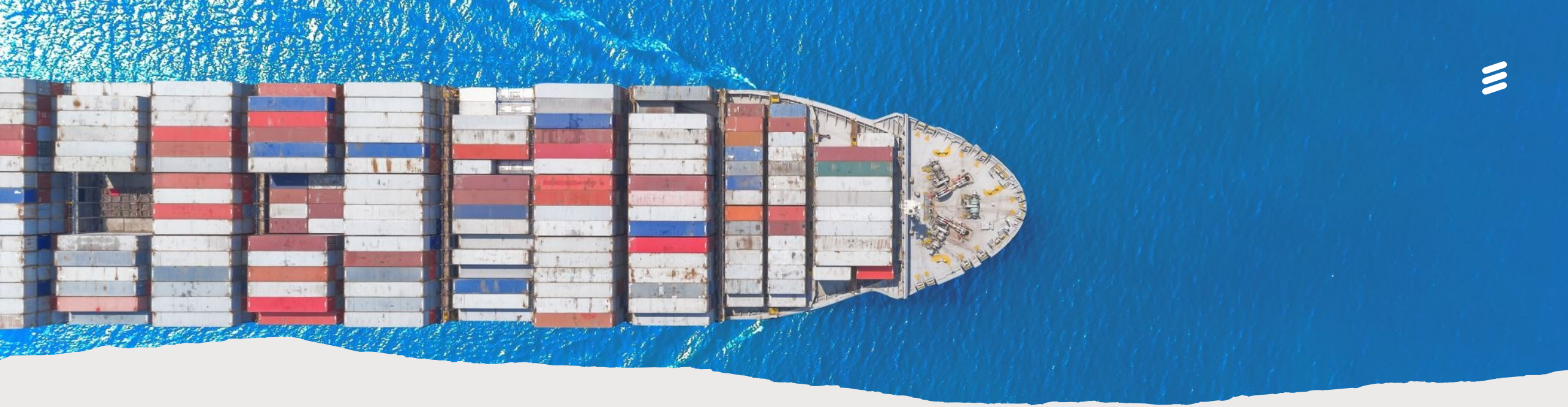
Kubernetes

- Krushnendu Jena
- Ashraf Shahzad
- Sandeep Kumar D
- Yogesh Sanjay Patil

Agenda

- Introduction
- What is k8s ? Why k8s ?
- Architecture
- Key Concepts
- Components
- Kubernetes resources



Introduction to Kubernetes

- The name of Kubernetes originates from Greek, meaning “helmsman” or “pilot”, and is the root of “governor” and “cybernetic”.
- Kubernetes, often abbreviated as K8s, is an open-source container orchestration platform designed to automate the deployment, scaling, and management of containerized applications.
- It was originally developed by Google and is now maintained by the Cloud Native Computing Foundation (CNCF).

VM

What is Kubernetes ? Why we need K8s ?

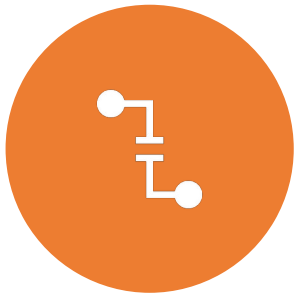
Container Orchestration: Kubernetes manages the deployment and scaling of containerized applications, ensuring they run reliably and consistently.

Scaling: It enables you to scale your applications up or down based on demand, ensuring optimal resource utilization.

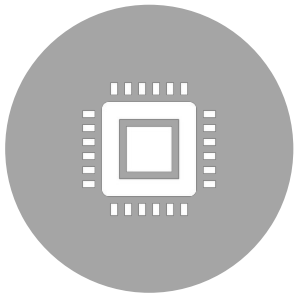
Automated Deployment and Rollouts: Kubernetes simplifies the process of deploying new versions of your application, allowing for smooth updates without downtime.

Load Balancing and Service Discovery: It handles traffic distribution across different instances of your application and provides mechanisms for service discovery.

What is Kubernetes ? Why we need K8s ?



Self-Healing: Kubernetes automatically restarts containers that fail and replaces them with new ones, ensuring high availability.

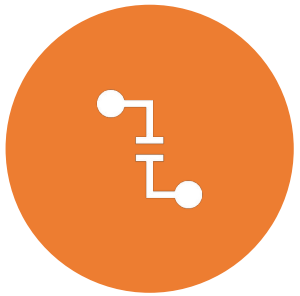


Resource Management: You can define the resources (CPU, memory, etc.) that each application or component can use, preventing resource contention.

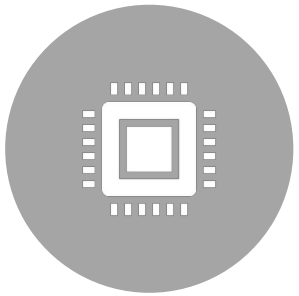
Declarative Configuration: Kubernetes allows you to define the desired state of your applications using YAML or JSON files, making it easy to version control and manage configurations.

Multi-Cloud and Hybrid Environments: It can manage applications across different cloud providers or in a combination of on-premises and cloud environments.

What is Kubernetes ? Why we need K8s ?



Automatic Bin Packing: Kubernetes automatically schedules containers based on their resource requirements and constraints, optimizing resource usage.



Horizontal Scaling: Kubernetes can scale applications up and down automatically based on metrics like CPU usage.

Automated Rollouts and Rollbacks: Kubernetes can manage deployments and ensure that only a certain number of pods are down while updating an application. It can roll back changes if something goes wrong.

Secret and Configuration Management: Kubernetes allows you to manage sensitive information and application configuration separately from the application code.

VM

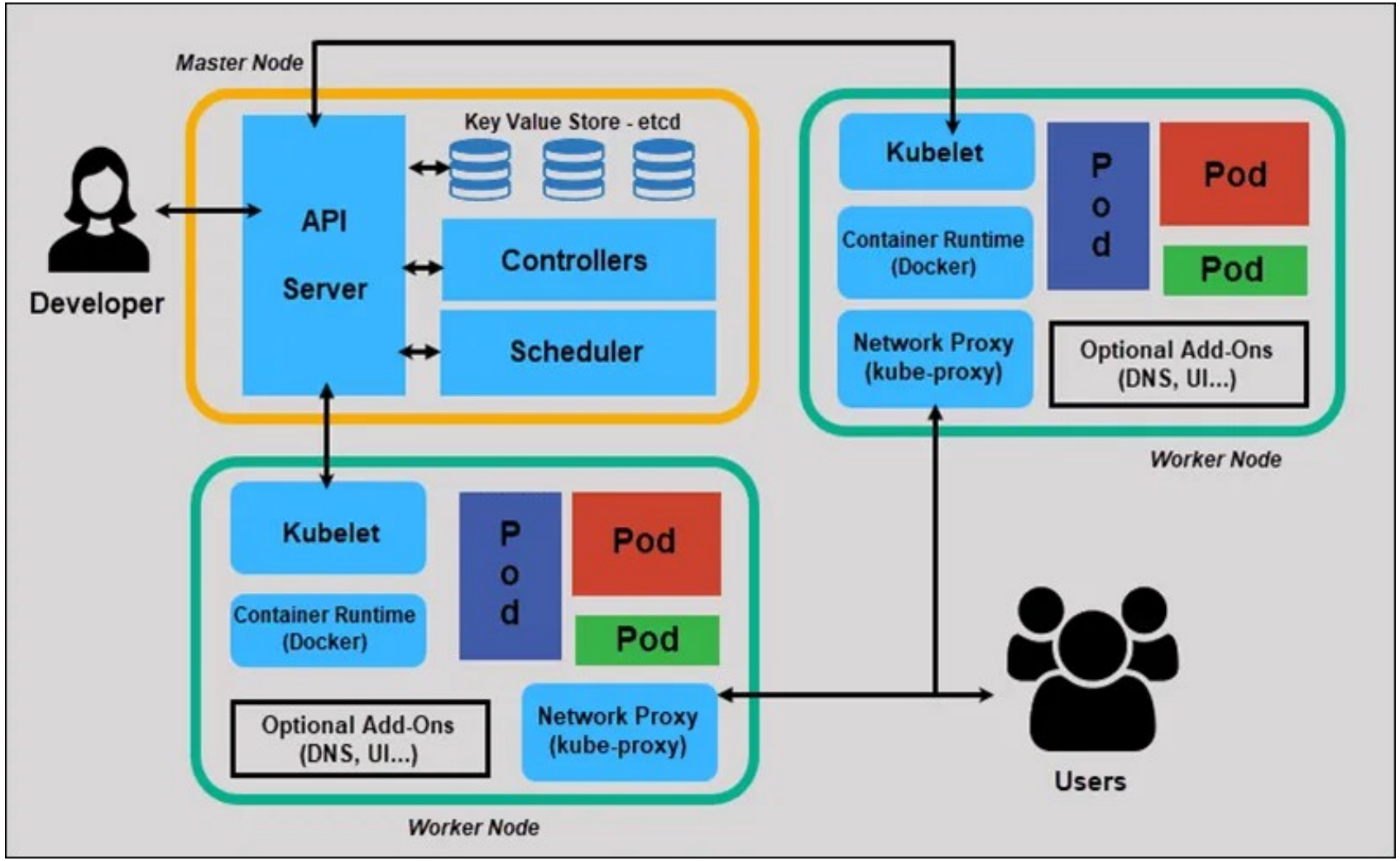
Kubernetes Benefits

- **Storage orchestration** Kubernetes allows you to automatically mount a storage system of your choice, such as local storages, public cloud providers, and more.
- **Automated rollouts and rollbacks** You can describe the desired state for your deployed containers using Kubernetes, and it can change the actual state to the desired state at a controlled rate. For example, you can automate Kubernetes to create new containers for your deployment, remove existing containers and adopt all their resources to the new container.
- **Secret and configuration management** Kubernetes store and manage sensitive information, such as passwords, OAuth tokens, and SSH keys. You can deploy and update secrets and application configuration without rebuilding your container images, and without exposing secrets in your stack configuration.
- **Batch execution** In addition to services, Kubernetes can manage your batch and CI workloads, replacing containers that fail, if desired.
- **Horizontal scaling** Scale your application up and down with a simple command, with a UI, or automatically based on CPU usage.
- **IPv4/IPv6 dual-stack** Allocation of IPv4 and IPv6 addresses to Pods and Services

D
G

VM

Kubernetes Architecture



Key Concepts:

Node:

A Node is like a computer in a network.

It's a machine, either a physical computer or a virtual machine, that does the actual work of running your applications.

Cluster:

A Kubernetes cluster is a set of nodes that run containerized applications. It includes the Master node for management and worker nodes for running applications.

Master Node:

The Master Node is like the "boss" or the "brain" of the Kubernetes cluster.

It's in charge of making decisions and managing the whole show.

Worker Node:

A Worker Node is like a "worker" in a team.

It's where the actual applications run.

Each Worker Node can run many applications in little containers.

VM

Master node component

API Server:

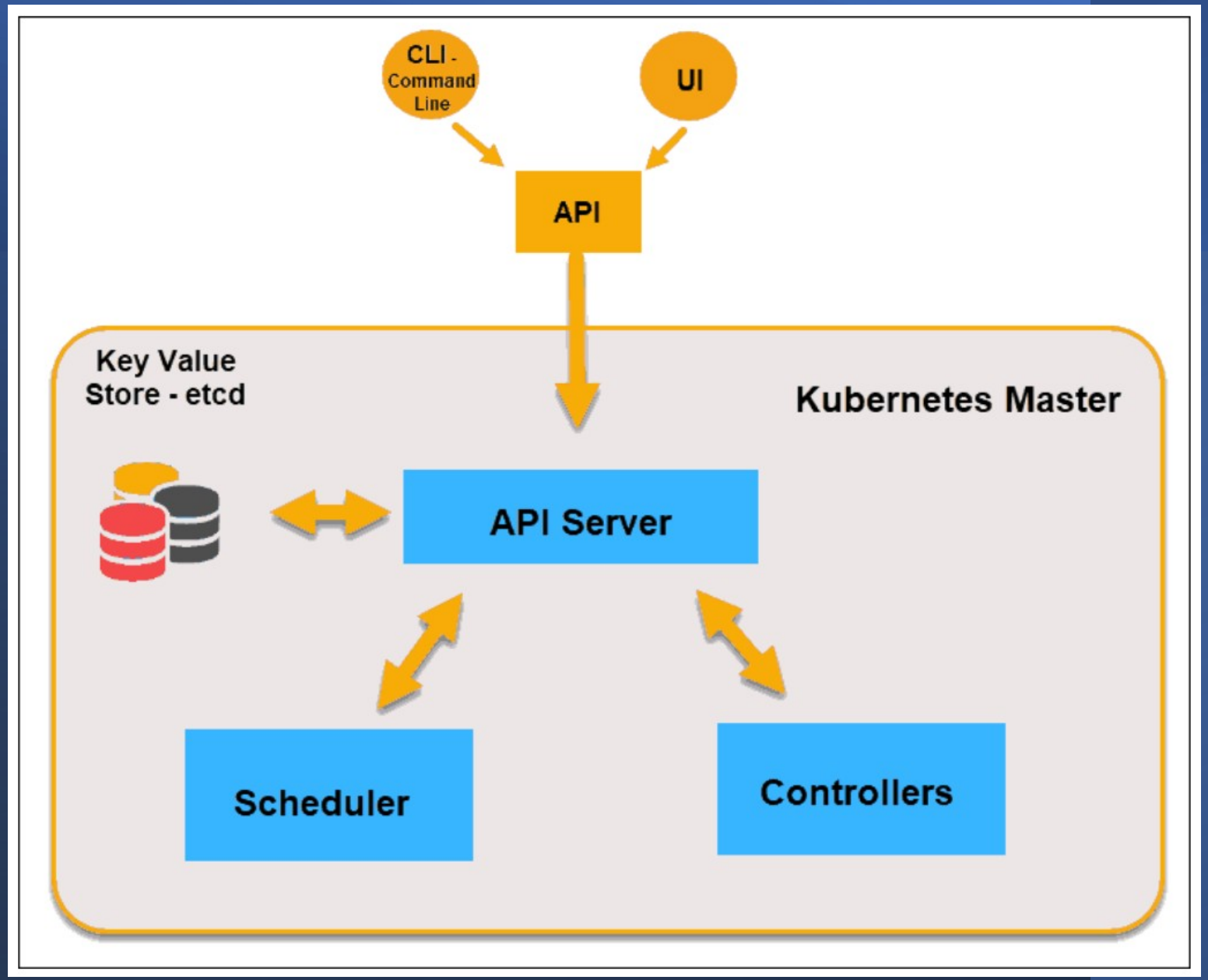
Scheduler:

Controller Manager:

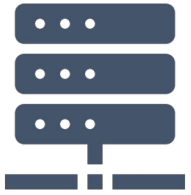
Etcd

Cloud Controller Manager (Optional):

VM


CLI - Command Line
UI
API
Key Value Store - etcd
Kubernetes Master
API Server
Scheduler
Controllers

Master node component



API Server:

Think of the API Server as the "communication hub" of the Master Node. It's like the front desk where everyone talks to each other.

It receives instructions from users and other components, and it tells the other parts of Kubernetes what to do.

Scheduler:

The Scheduler is like a "matchmaker" for pods and nodes. It looks at all the pods and decides which Worker Node each pod should run on.

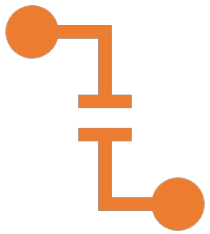
It tries to spread the pods evenly across all the nodes.

Controller Manager:

The Controller Manager is like the "supervisor" of the Master Node. It watches over everything and makes sure everything stays the way it should.

It checks if the right number of pods are running and takes action if something goes wrong.

Master node component



etcd:

Think of etcd as the "memory" of the Master Node. It's like a special storage place where Kubernetes keeps all its important information.

It's super reliable and ensures that even if the Master Node goes down, the information is safe and can be recovered.

Cloud Controller Manager (Optional):

If you're using a cloud provider, like AWS or Google Cloud, this component helps Kubernetes talk to that provider's services.

It's like a translator that helps Kubernetes understand the specific features of that cloud.

VM

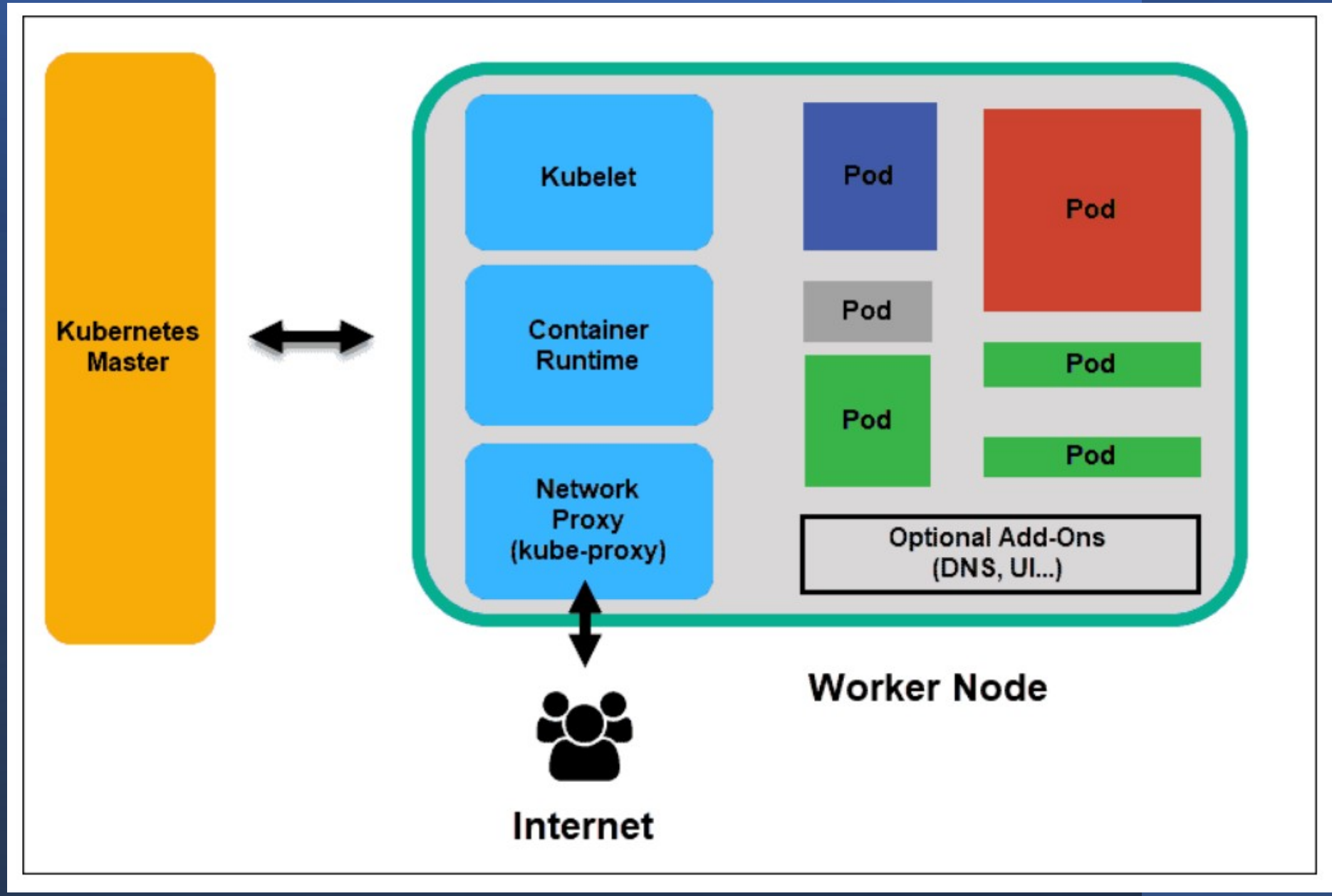
Worker node component

Kubelet

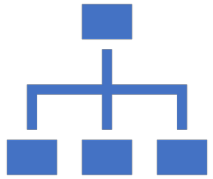
Kube Proxy:

Container Runtime:

VM


Kubernetes Master
Kubelet
Container Runtime
Network Proxy (kube-proxy)
Pod
Pod
Pod
Pod
Pod
Pod
Optional Add-Ons (DNS, UI...)
Worker Node
Internet

Worker node component



Kubelet:

Think of the Kubelet as the "task manager" of the Worker Node. It's responsible for making sure that the containers (pods) are running as they should.

It takes orders from the Master Node and makes sure the containers are up and healthy.

Kube Proxy:

The Kube Proxy is like a "traffic cop" for network traffic. It's in charge of making sure that the right pods receive the messages and requests.

It helps pods communicate with each other inside the cluster.

Container Runtime:

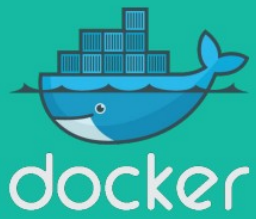
This is like the "engine" that runs the containers. It's the software that takes care of starting, stopping, and managing containers.

Common runtimes include Docker, containerd, and others.

VM

Kubernetes Resources

kubernetes



Core Kubernetes Resources

- **Pod**
- **Services**
- **Deployments**
- **ReplicaSets**
- **ConfigMaps:**
- **Secrets:**
- **Ingress**
- **PersistentVolumes (PV)**
- **PersistentVolumeClaims (PVC):**
- **Namespaces:**
- **StatefulSets**
- **DaemonSets:**

What is a pod

A Pod is the smallest unit of computing that can

be created and manage Kubernetes.

A Pod is a logical collection of one or more containers.

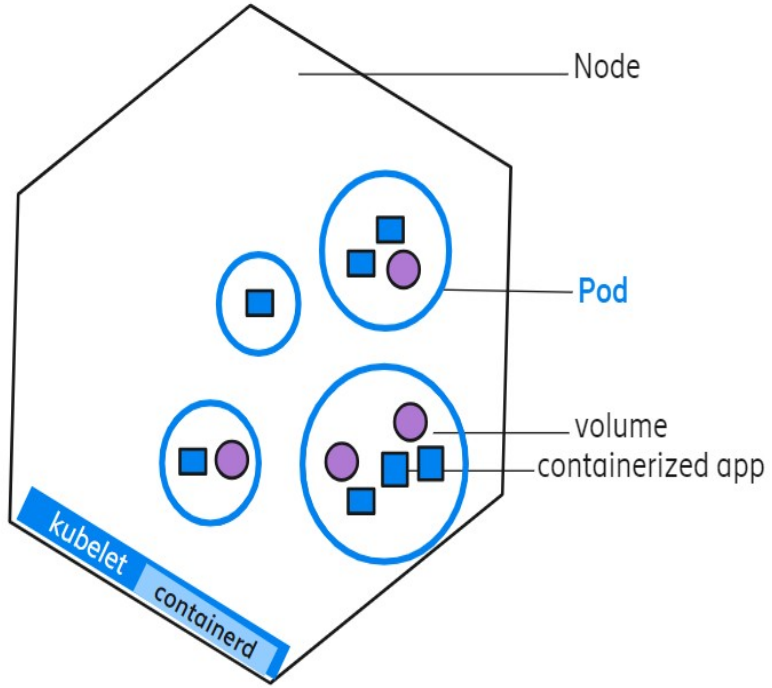
A Pod represents a single instance of the application.

Pods in general , not stateful.

(The intention is that misbehaving

Pods shall be replaced.)

The basic scheduling unit in Kubernetes are pod.



DG

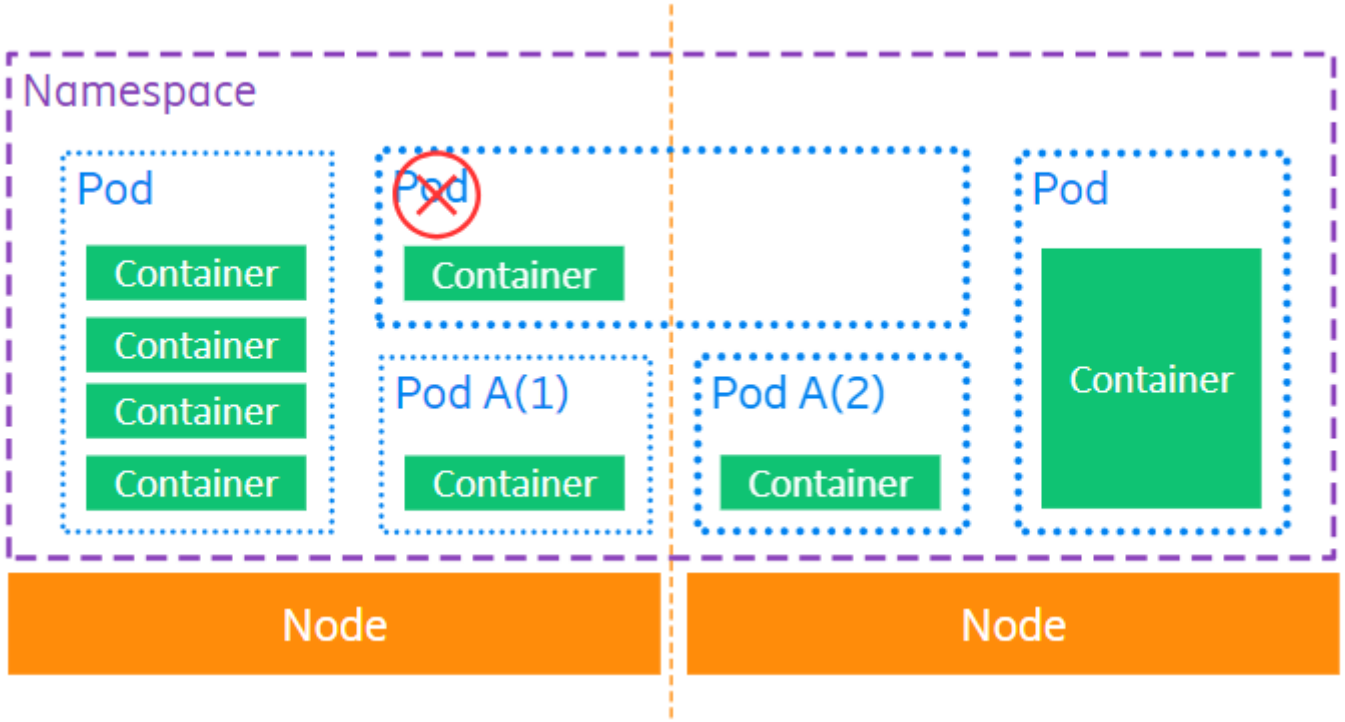
Kubernetes Pod

A Pod runs on a node.

A Pod cannot be stretched across multiple nodes.

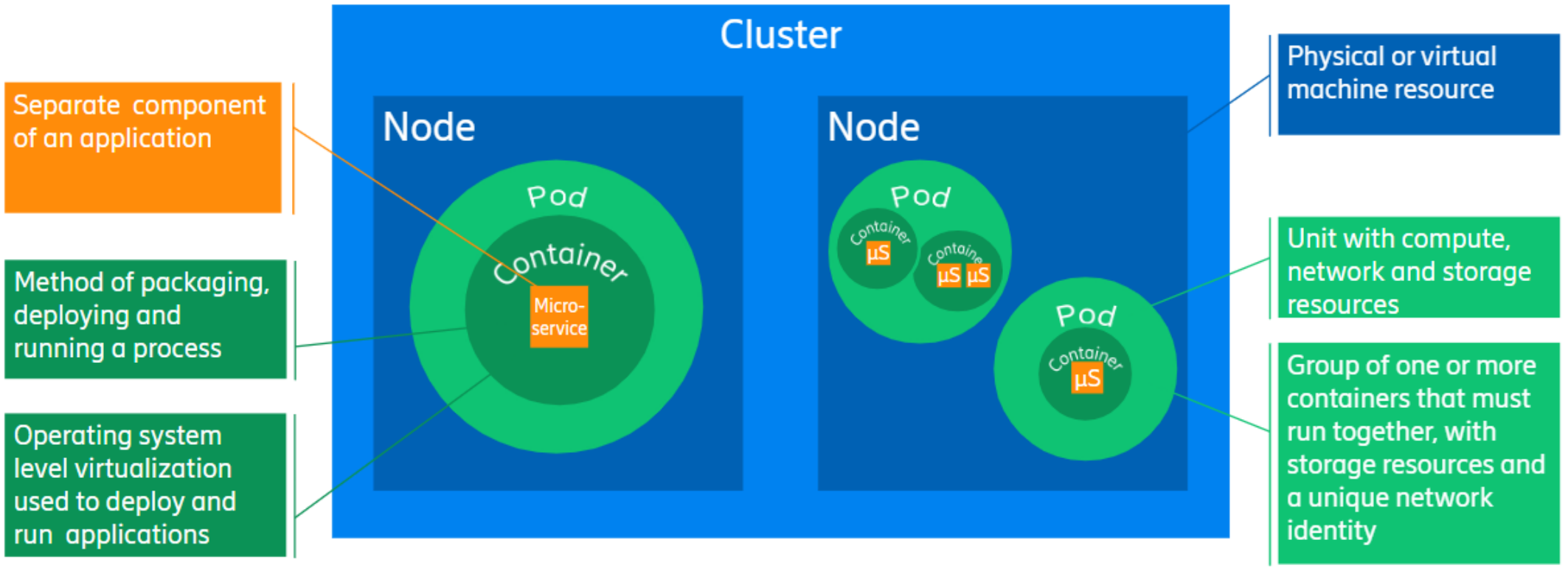
You can deploy multiple copy of pods called replicas

Pods cannot be live migrated to another node like virtual machine do.



DG

Pods, Containers and Microservices



A Pod can be considered as a microservice in the CNF design

DG

Kubernetes Network Model

Kubernetes Network model address the following Communication scenario

Containers within a pod use the networking to communicate

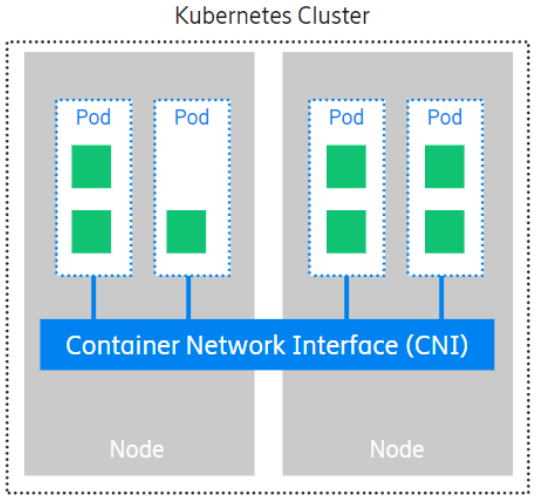
Via loopback address.

Cluster Networking provides communication between different pods.

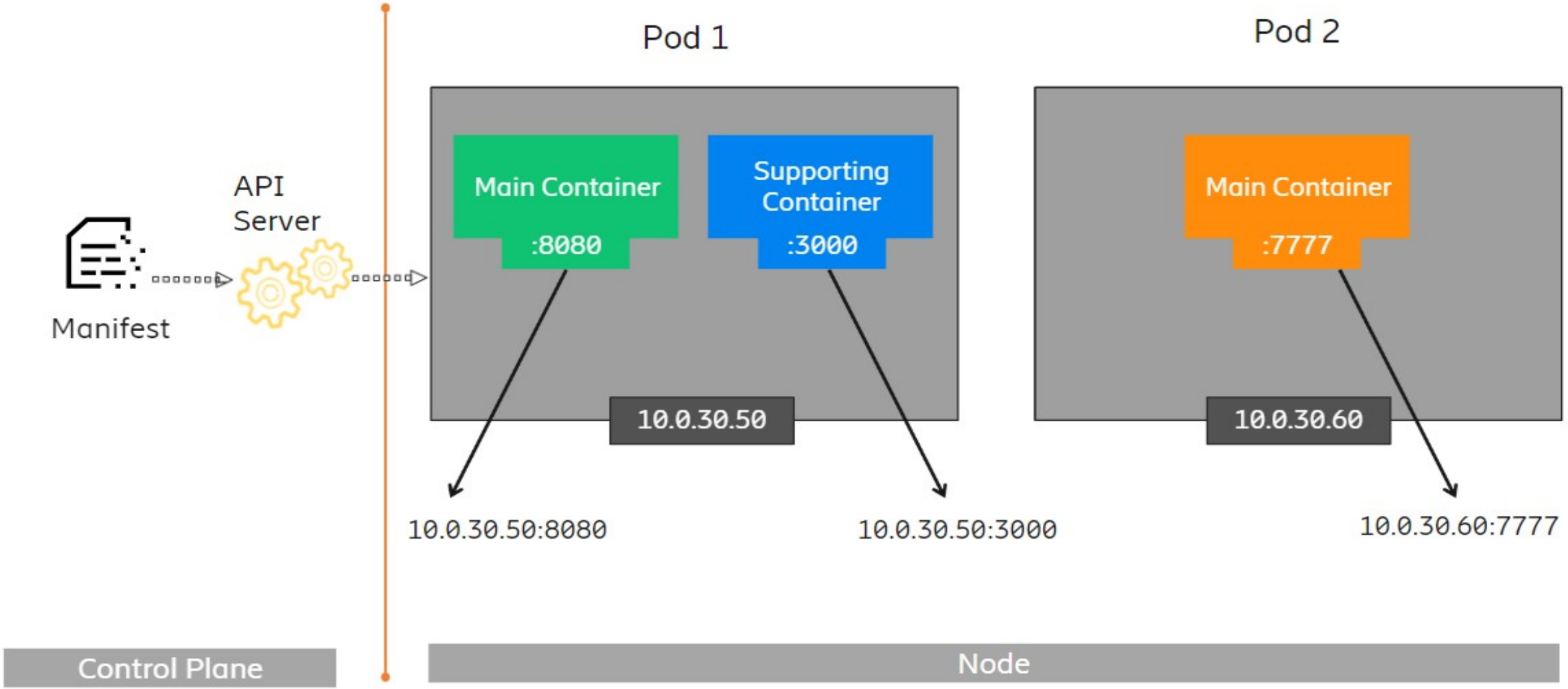
(Via Container Network Interface (CNI) Plugins.

Service API lets you expose an application running on pods to be reachable from

Outside your cluster.



Pod Networking

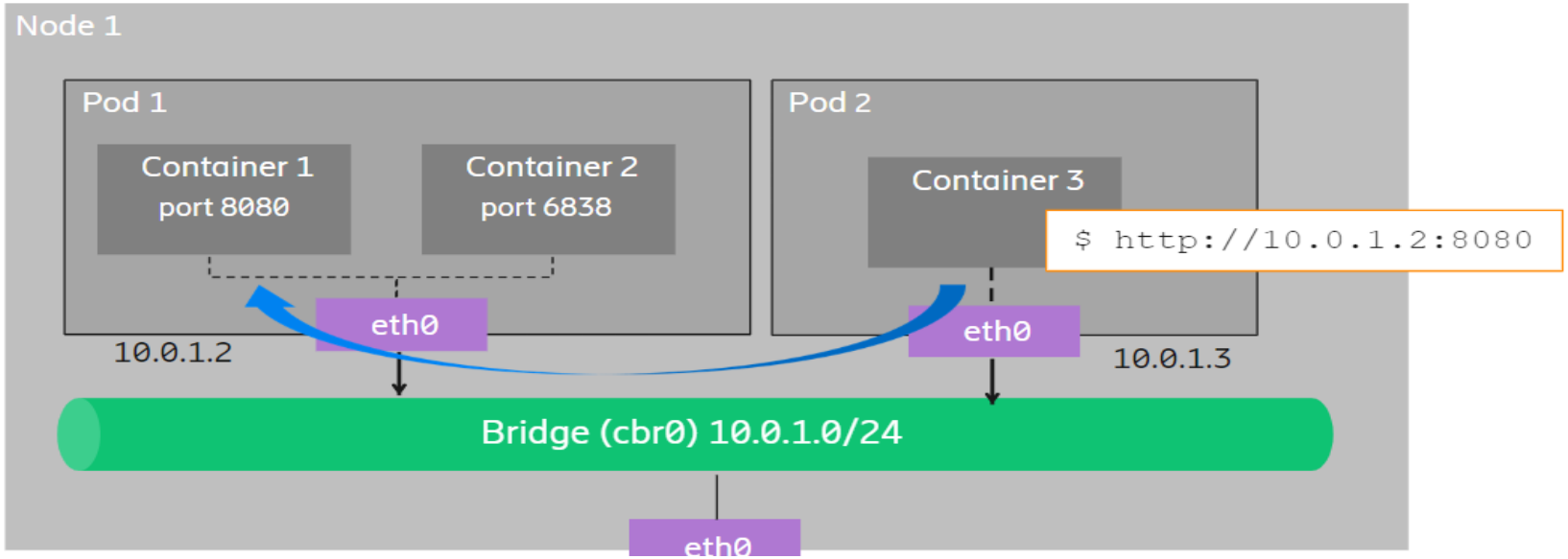


DG

Networking: Containers in differnet Pods

Communication between containers is done using the IP from the Pod and the required port.

All Containers in the same host are attached to the same bridge.



DG

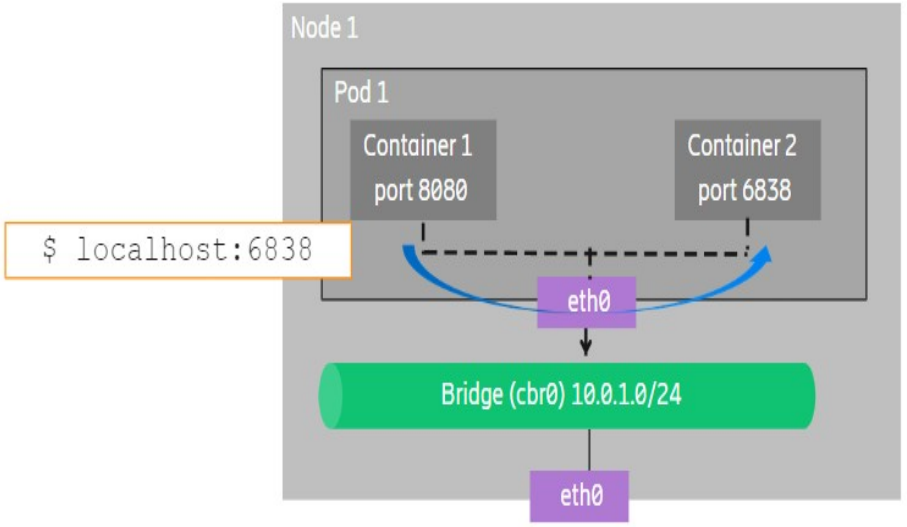
Networking: Containers in the same Pods

All Containers within the same pods expose the same IP address and share a common network stack.

Application Services from each container are exposed through ports.

Ports can not be shared between containers in the same pod.

Containers inside the Pod Communicate with each other through “ local host: port “



Namespace

Namespace:

A Namespace provides a way to divide cluster resources among multiple users or teams. It allows for resource isolation and management.

Kubernetes using an imperative command, you can use the kubectl create namespace command:

Here's an example:

kubectl create namespace <mynamespace>

kubectl create namespace eric-ec-apps

list and view

kubectl get namespace or kubectl get ns

kubectl describe ns eric-ec-apps

kubectl get ns eric-ec-apps –o yaml

To create a namespace using the declarative method with a YAML file

you'll first need to create a YAML file that defines the namespace. Here's an example YAML file named

mynamespace.yaml

```
apiVersion: v1
kind: Namespace
metadata: \
   name: eric-ec-apps
```

kubectl apply -f mynamespace.yaml

Or

Kubectl create namespace mynamespacename

Managing Kubernetes Controllers

Managing Kubernetes controllers involves understanding and working with different types of controllers that automate the management of applications and services in a Kubernetes cluster.

Types of Kubernetes Controllers:

- **ReplicationController (RC)**
- **ReplicaSet (RS)**
- **Deployment**
- **StatefulSet**
- **DaemonSet**

ReplicationController (RC)

A ReplicationController is a component in Kubernetes, a popular open-source platform for automating the deployment, scaling, and operation of application containers.

Purpose

- **Ensure Pod Availability:** The primary function of a ReplicationController is to ensure that a specified number of pod replicas are running at any given time.
- **Self-Healing:** If a pod fails or is deleted, the ReplicationController will automatically create a new one to replace it.

ReplicationController (RC)

Key Features:

1.Replica Management:

1. We can specify the desired number of replicas.
2. The ReplicationController continuously monitors the current state and makes adjustments to maintain the desired state.

2.Label Selector:

1. Uses label selectors to identify which pods it manages.
2. It can manage multiple pods that match a specified label selector.

3.Scaling:

1. Allows for easy scaling of applications by simply changing the replica count.
2. The ReplicationController will create or delete pods as necessary to match the new replica count.

ReplicationController (RC)

Components of a ReplicationController:

- **apiVersion:** Defines the version of the API (e.g., v1).
- **kind**: Specifies the type of Kubernetes object (e.g., ReplicationController).
- **metadata**: Includes metadata such as name, namespace, and labels.
- **spec:** Specifies the desired state, including the replica count and the pod template.

example

```
#replicationcontroller.yaml
apiVersion: v1
kind: ReplicationController
metadata:
  name: my-replication-controller
  labels:
    app: my-app
spec:
   replicas: 3
   selector:
      matchLabels:
           app: my-app
     template:
        metadata:
            labels:
                app: my-app
         spec:
            containers:
                - name: my-container
                  image: my-image
                  ports:
```

ReplicationController (RC)

Use Cases

- Legacy Applications: Some older Kubernetes setups might still use ReplicationControllers.
- Simple Applications: For straightforward use cases where advanced features of Deployments are not needed.

Transition to ReplicaSets:

- **ReplicaSets:** ReplicationControllers have largely been replaced by ReplicaSets in newer versions of Kubernetes. ReplicaSets offer more powerful and flexible label selectors.
- **Deployment:** For most use cases, Deployments are recommended over directly using ReplicaSets or ReplicationControllers. Deployments provide additional features such as rolling updates and rollbacks.

ReplicaSet(RS)

ReplicaSet in Kubernetes ensures that a specified number of pod replicas are running at any given time. If a pod goes down, the ReplicaSet creates another to replace it. This is essential for maintaining application availability and fault tolerance.

Key Concepts

- **Pod:** The smallest deployable unit in Kubernetes, which can contain one or more containers.
- **ReplicaSet:** A Kubernetes object that manages a set of identical pods, ensuring a specified number of replicas are running.

ReplicaSet(RS)

How a ReplicaSet Works

- **Specification:** You define the desired state in a YAML file, including the number of replicas and the pod template.
- **Creation:** When you apply this YAML file, Kubernetes creates the ReplicaSet.
- **Monitoring:** The ReplicaSet constantly monitors the pods to ensure the correct number is running.
- **Scaling:** You can manually scale the ReplicaSet up or down by changing the replica count in the YAML file and reapplying it.
- **Self-Healing:** If a pod fails or is deleted, the ReplicaSet automatically creates a new one to maintain the desired number of replicas.

ReplicaSet(RS)

Example YAML for a ReplicaSet

```
apiVersion: apps/v1
kind: ReplicaSet
metadata:
  name: example-replicaset
spec:
  replicas: 3
  selector:
    matchLabels:
      app: myapp
  template:
    metadata:
      labels:
        app: myapp
    spec:
      containers:
      - name: mycontainer
        image: myimage:latest
        ports:
        - containerPort: 80
```

ReplicaSet(RS)

Key Fields

- **apiVersion:** The version of the Kubernetes API.
- **kind:** The type of resource (ReplicaSet).
- **metadata:** Data to uniquely identify the ReplicaSet, like name and labels.
- **spec**: The desired state of the ReplicaSet, including the number of replicas and the pod template.
- **replicas:** The desired number of pod replicas.
- **selector:** A label query to select the pods managed by this ReplicaSet.
- **template:** The pod template describing how each pod should be created.

ReplicaSet(RS)

Command for Managing a ReplicaSet

- **Create**: kubectl apply -f replicaset.yaml
- **View:** kubectl get replicaset
- **Update:** Modify the YAML file and reapply with kubectl apply -f replicaset.yaml
- **Delete:** kubectl delete replicaset example-replicaset

NB: ReplicaSets are often used indirectly through Deployments, which provide additional management features like rolling updates and rollbacks.

Deployment

A Kubernetes Deployment used how to create or modify instances of the pods that hold a containerized application. Deployments can help to efficiently scale the number of replica pods, enable the rollout of updated code in a controlled manner, or roll back to an earlier deployment version if necessary.

- **Key Concepts**
- **Pod:** The smallest deployable unit in Kubernetes, which can contain one or more containers.
- **ReplicaSet:** A Kubernetes object that manages a set of identical pods, ensuring a specified number of replicas are running.

Example YAML for a Deployment

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: nginx-deployment
  labels:
    app: nginx
spec:
  replicas: 3
  selector:
    matchLabels:
      app: nginx
  template:
    metadata:
      labels:
        app: nginx
    spec:
      containers:
      - name: nginx
        image: nginx:1.14.2
        ports:
        - containerPort: 80
```

StatefulSet

•A StatefulSet is a Kubernetes workload API object that is used to manage stateful applications. StatefulSets are designed to handle applications that require persistent storage and stable network identities. This is particularly important for applications like databases and distributed systems, where each instance needs to be uniquely identifiable and maintain its state across restarts.

•**Key features and benefits of using StatefulSets:**

•**Stable, Unique Pod Names:** Each pod in a StatefulSet has a unique, stable name. These names follow a predictable pattern, which helps in maintaining the order and identity of each pod. For example, if the StatefulSet is named web, the pods will be named web-0, web-1, web-2, and so on.

StatefulSet

- **Ordered, Graceful Deployment and Scaling**: Pods in a StatefulSet are created and deleted in a specific order. When scaling up or down, Kubernetes ensures that operations are performed sequentially. This ensures that the application maintains its expected order and dependencies.
- **Persistent Storage**: StatefulSets work closely with PersistentVolume Claims (PVCs). Each pod in a StatefulSet can have its own persistent storage, which remains even if the pod is deleted. This is crucial for applications that need to retain data between restarts.

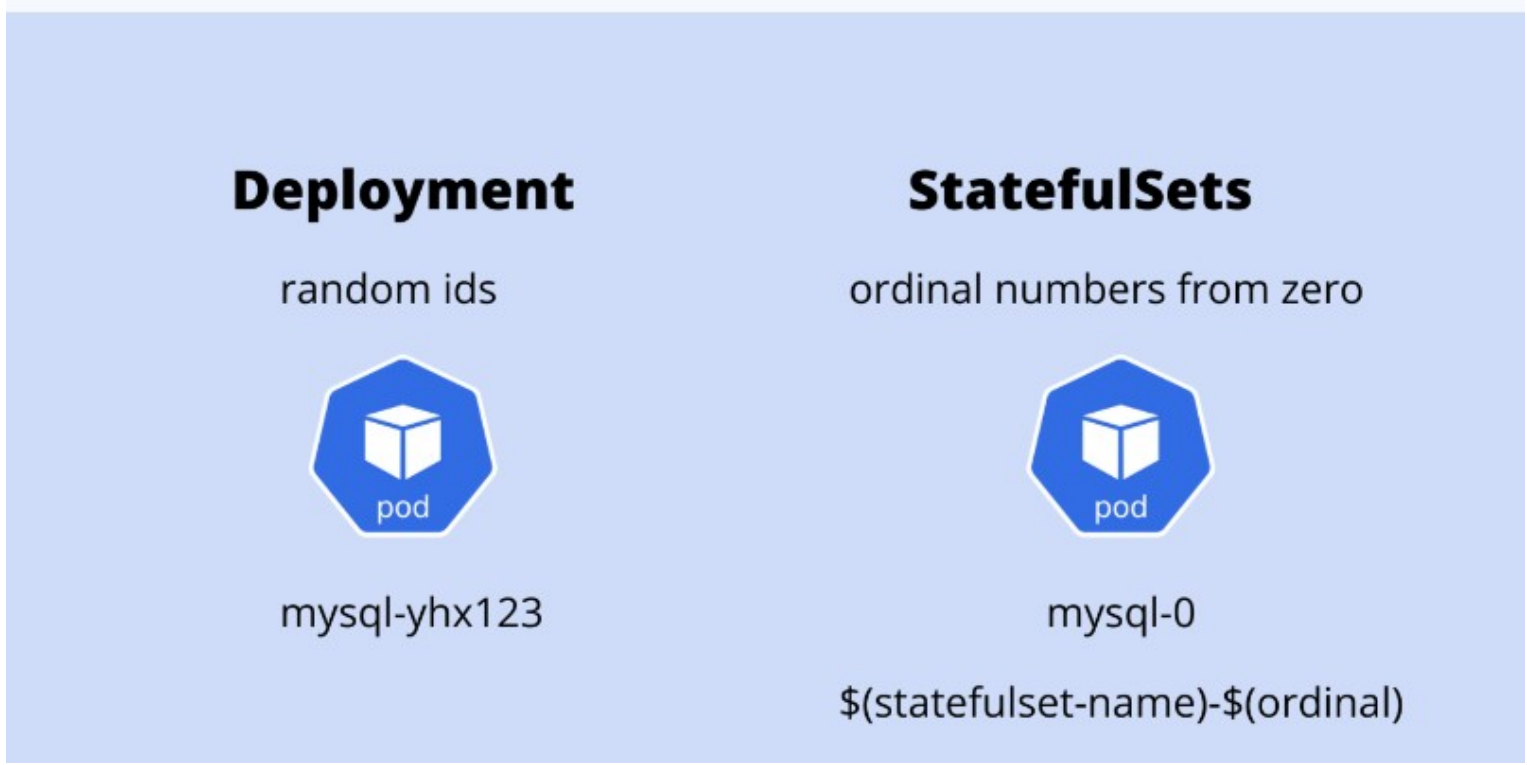
- **Stable Network Identities**: Pods in a StatefulSet get stable network identities. Each pod is assigned a unique DNS endpoint, which remains consistent even if the pod is rescheduled to a different node. This helps in maintaining communication within the application and with external clients.
- **Rolling Updates**: StatefulSets support rolling updates, allowing you to update your application without downtime. Kubernetes will update the pods one by one, ensuring that the application remains available during the update process.

StatefulSet Vs Deployment

	Feature	StatefulSet	Deployment
	Pod identities	Pods are assigned a persistent identifier, derived from the StatefulSet's name and their ordinal creation index.	Pods are assigned random identifiers, derived from the Deployment's name and a unique random string.
	Pod interchangeability	Pods in a StatefulSet are not interchangeable. It's expected that each Pod has a specific role, such as always running as a primary or read-only replica for a database application.	All Pods are identical, so they're interchangeable and can be replaced at any time.
	Rollout ordering	Pods are guaranteed to be created and removed in sequence. When you scale down the StatefulSet, Kubernetes will terminate the most recently created Pod.	No ordering is supported. When you scale down the Deployment, Kubernetes will terminate a random Pod.
	Storage access	Each Pod in the StatefulSet is assigned its own Persistent Volume (PV) and Persistent Volume Claim (PVC).	All Pods share the same PV and PVC.

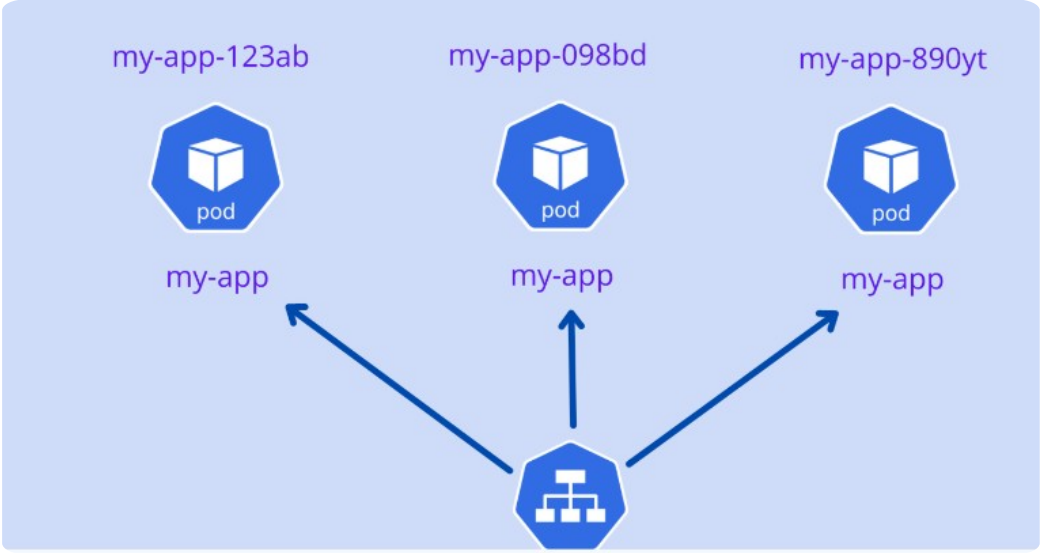
Deployment Vs StatefulSet

The diagram shows how Deployment and StatefulSets assign names to the Pods.



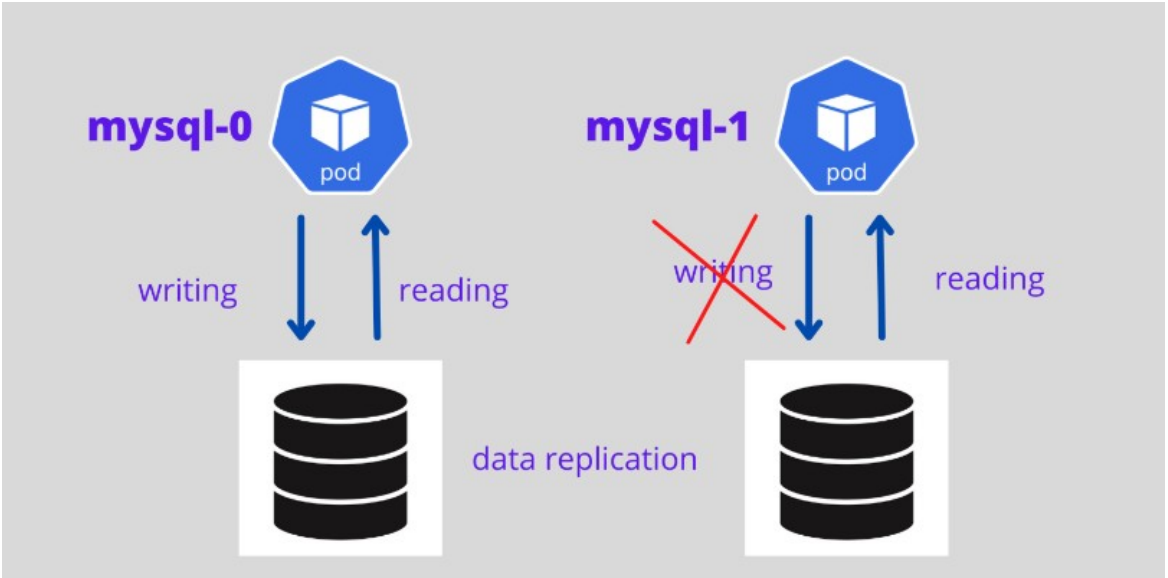
Deployments

All these Pods are associated with one load balancer service. So in a stateless application, changes in the Pod name are easily identified, and the service object easily handles the random IDs of Pods and distributes the load. This type of deployment is very suitable for stateless applications.



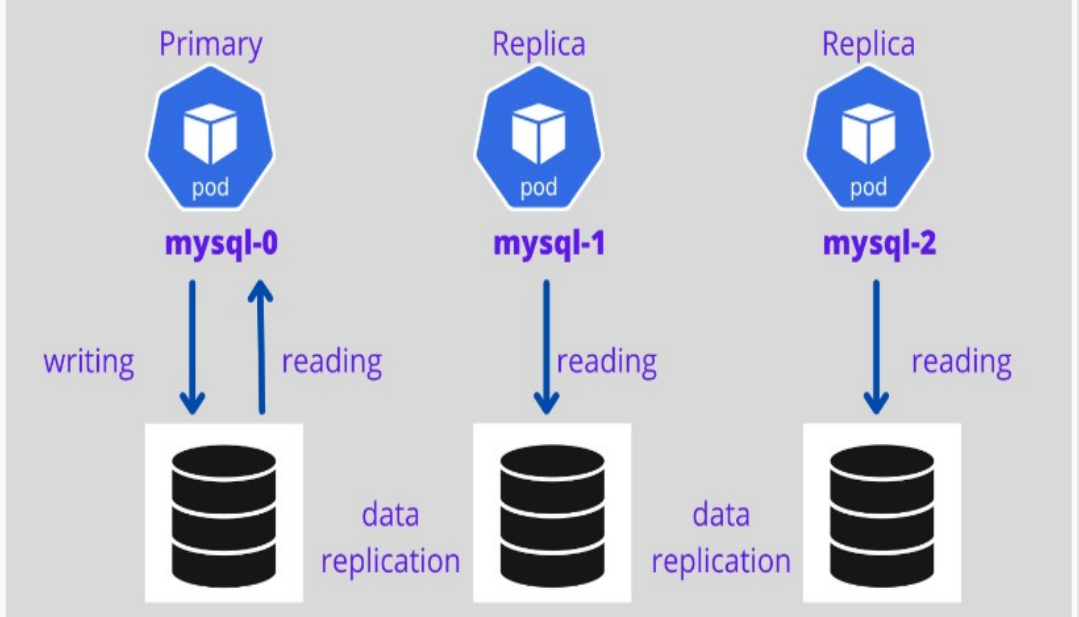
Stateful applications cannot be deployed like deployment in kubernetes. The stateful application needs a sticky identity for each Pod because replica Pods are not identical Pods.

Kubernetes StatefulSets controller offers an ordinal number for each Pod starting from zero, such as mysql-0, mysql-1, mysql-2, and so forth.

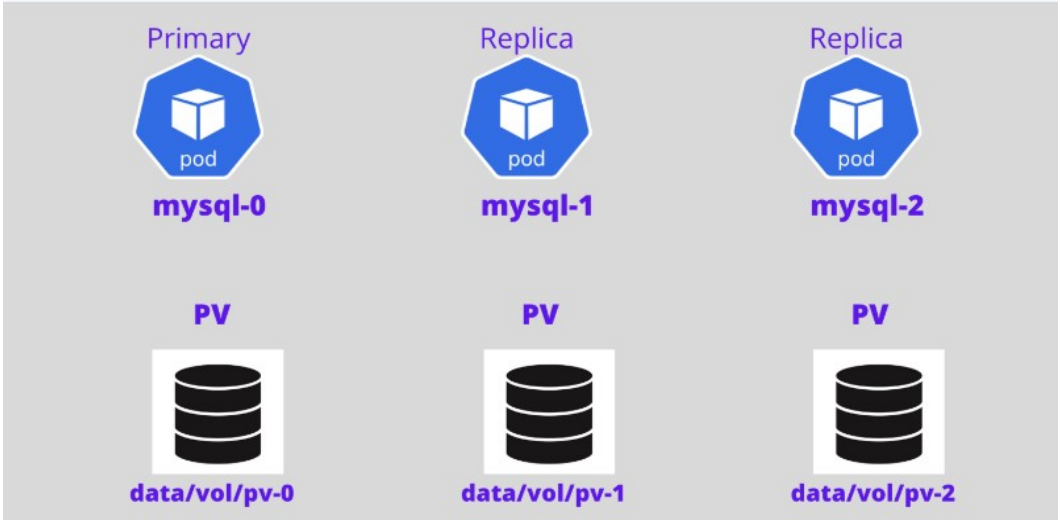


StatefulSets

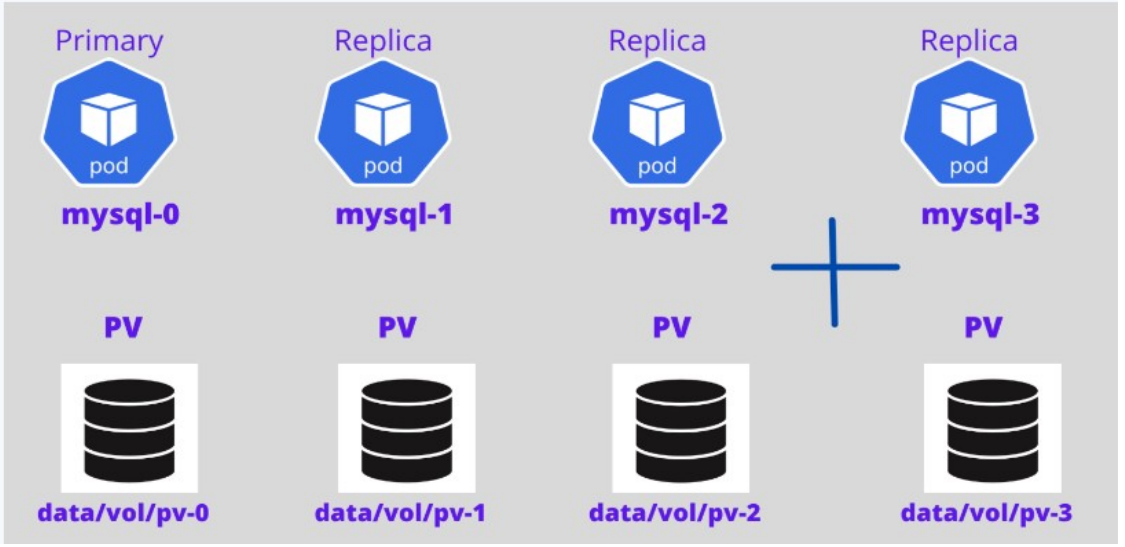
For stateful applications with a StatefulSet controller, it is possible to set the first Pod as primary and other Pods as replicas—the first Pod will handle both read and write requests from the user, and other Pods always sync with the first Pod for data replication. If the Pod dies, a new Pod is created with the same name.



The diagram below shows a MySQL primary and replica architecture with persistent volume and data replication architecture.

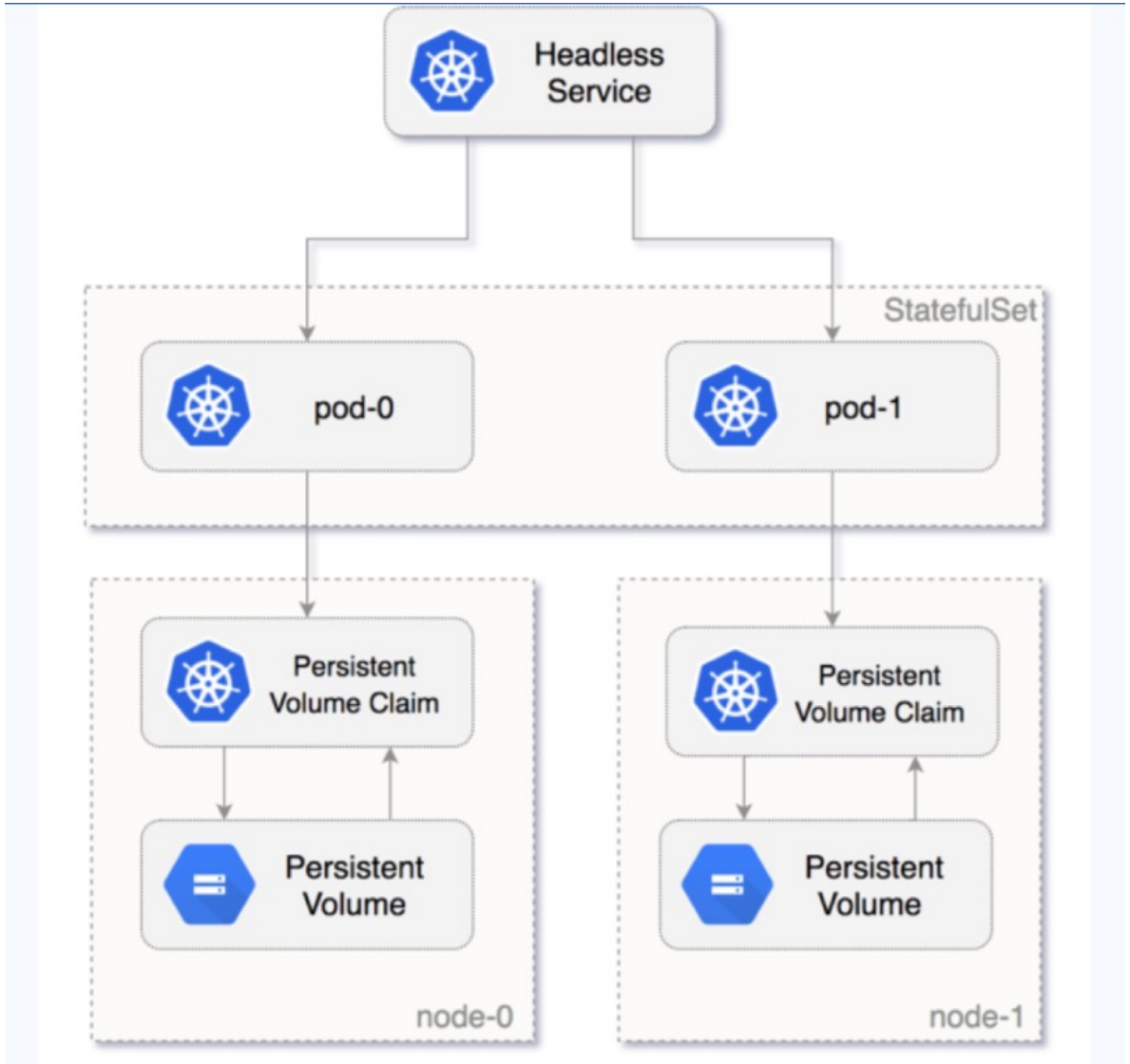


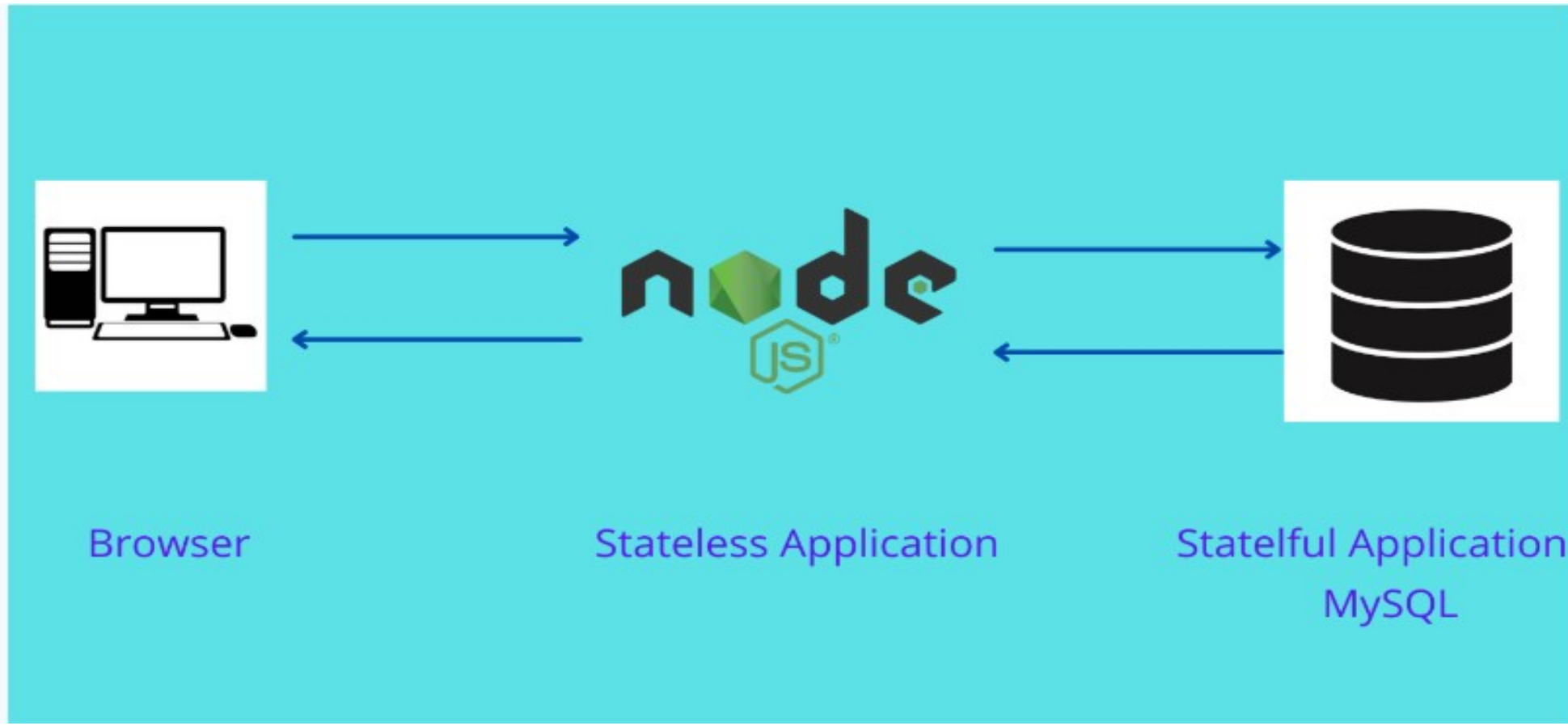
Now, add another Pod to that. The fourth Pod will only be created if the third Pod is up and running, and it will clone the data from the previous Pod.



StatefulSets

The diagram shows how the Pod is numbered from zero and how Kubernetes persistent volume is attached to the Pod in the StatefulSets.




Browser
Stateless Application
Statelful Application
MySQL

StatefulSet

When to Use StatefulSets
There are several reasons to consider using StatefulSets. Here are two examples:

1. Assume you deployed a MySQL database in the Kubernetes cluster and scaled this to three replicas, and a frontend application wants to access the MySQL cluster to read and write data. The read request will be forwarded to three Pods. However, the write request will only be forwarded to the first (primary) Pod, and the data will be synced with the other Pods. You can achieve this by using StatefulSets.
2. Deleting or scaling down a StatefulSet will not delete the volumes associated with the stateful application. This gives you your data safety. If you delete the MySQL Pod or if the MySQL Pod restarts, you can have access to the data in the same volume.

Summary of StatefulSets

In summary, StatefulSets provide the following advantages when compared to Deployment objects:

- Ordered numbers for each Pod
- The first Pod can be a primary, which makes it a good choice when creating a replicated database setup, which handles both reading and writing
- Other Pods act as replicas
- New Pods will only be created if the previous Pod is in running state and will clone the previous Pod's data
- Deletion of Pods occurs in reverse order.

StatefulSet

Use Cases for StatefulSets

StatefulSets are ideal for applications that require one or more of the following:

- Persistent storage for saving data
- Stable and unique network identifiers
- Ordered deployment and scaling
- Ordered, automated rolling updates

Common use cases include:

- Databases (e.g., MySQL, MongoDB)
- Distributed systems (e.g., Apache Cassandra, Elasticsearch)
- Stateful applications (e.g., Zookeeper, Kafka)

Stateful Set

- **Commands :**

- **1. C**reate a StatefulSet from a YAML file:
- *# kubectl apply -f <statefulset.yaml>*.
- 2. To delete the StatefulSet :
- *#kubectl delete s*tatefulset < *s*tatefulset *-name>*
- 3. List StatefulSet :
- *#kubectl get s*tatefulset/sts
- 4. Describe a StatefulSet :
- #kubectl describe daemonset <daemonset-name>
- 5. View pods created by a StatefulSet
- *#kubectl get pods -l app=<label-value>*

StatefulSets YAML file

In this example:

•**apiVersion**: Specifies the Kubernetes API version, such as “apps/v1” for StatefulSets.
•**kind**: Specifies the type of Kubernetes resource, in this case, “StatefulSets.”
•**metadata**: Provides metadata for the StatefulSets , including the name, labels, and annotations.
•**spec**: Defines the desired state of the StatefulSets, including the number of replicas, the pod template, and any other related specifications. It includes:
•**replicas**: Specifies the desired number of identical pod replicas to run.
•**selector**: Specifies the labels that the Replica Set uses to select the pods it should manage.
•**template**: Contains the pod template used for creating new pods, including container specifications, image names, and container ports.

Stateful Example using yaml

StatefulSet.yaml

```
apiVersion: apps/v1
kind: StatefulSet
metadata:
  name: example-statefulset
spec:
  serviceName: "nginx"
  replicas: 2
  selector:
    matchLabels:
      app: nginx
  template:
    metadata:
      labels:
        app: nginx
    spec:
      containers:
```

Stateful Example using yaml

```
---
apiVersion: apps/v1
kind: StatefulSet
metadata:
 name: count
spec:
 serviceName: "nginx"
 replicas: 3
 selector:
  matchLabels:
   app: nginx
 template:
  metadata:
   labels:
    app: nginx
  spec:
   containers:
   - name: nginx
     image: k8s.gcr.io/nginx-slim:0.8
     ports:
     - containerPort: 80
       name: web
     volumeMounts:
     - name: myclaim
       mountPath: /mnt/data
 volumeClaimTemplates:
 - metadata:
    name: myclaim
  spec:
   accessModes: [ "ReadWriteOnce" ]
   resources:
    requests:
```

StatefulSet

- Results using the ymal file

- A StatefulSet named nginx is created.
- It creates 3 replicas of the pod, each with a unique name (nginx-0, nginx-1, nginx-2).
- Each pod runs an Nginx container.
- Each pod has a PersistentVolumeClaim for storage.

StatefulSets provide a powerful way to manage stateful applications in Kubernetes, ensuring data persistence, stable network identities, and ordered deployment.

DaemonSet

•A DaemonSet is a Kubernetes resource that ensures a copy of a specific pod runs on all or some nodes in a Kubernetes cluster. It is typically used for deploying background tasks such as monitoring agents, log collectors, or network services that need to run on every node.

DaemonSet

Key points about DaemonSets:

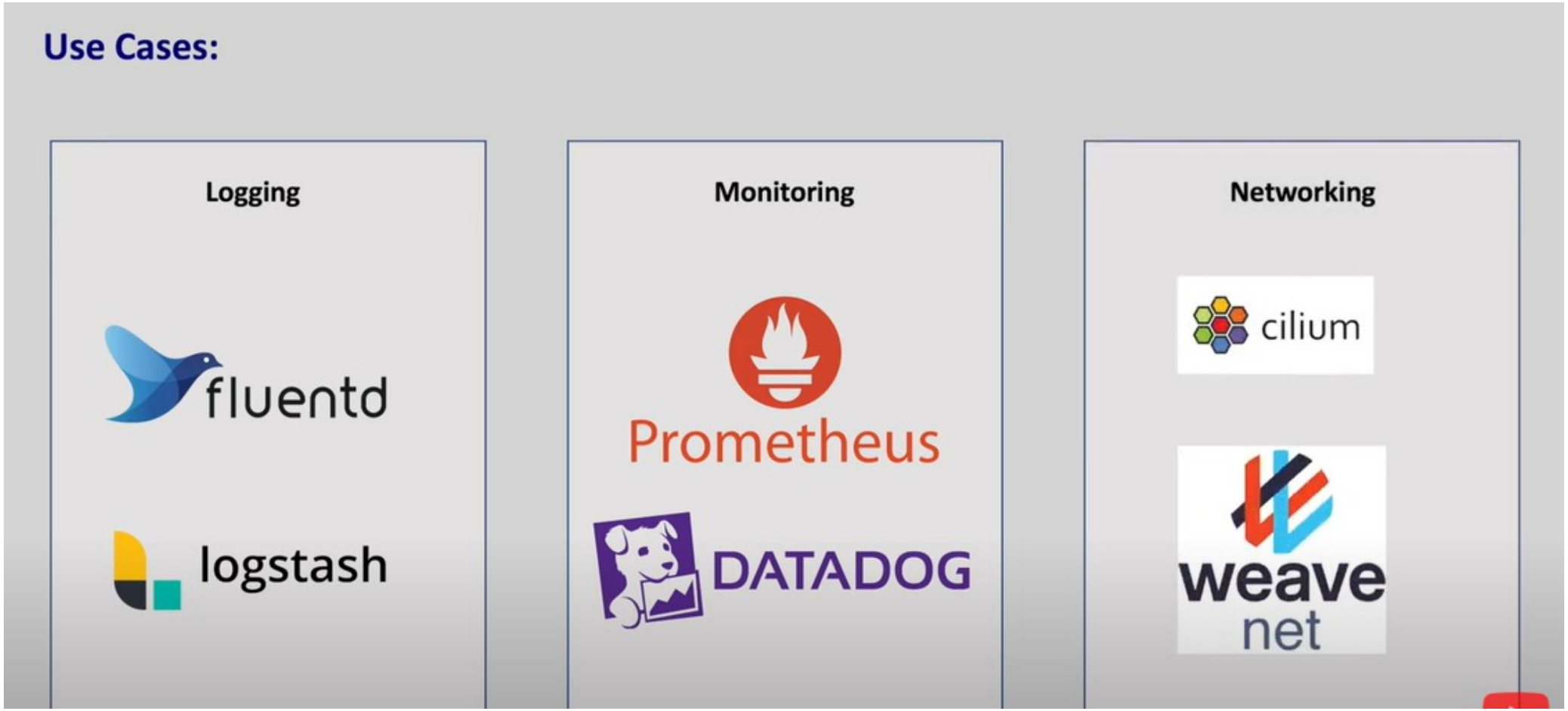
Deployment: When you create a DaemonSet, Kubernetes ensures that the specified pod is scheduled on all eligible nodes. If new nodes are added to the cluster, the DaemonSet automatically schedules the pod on those nodes as well.

Management: You can manage DaemonSets using standard Kubernetes tools like kubectl. This includes updating, scaling, and deleting DaemonSets.

Selective Deployment: You can restrict a DaemonSet to run on specific nodes by using node selectors, node affinity, or taints and tolerations.

Use Case: DaemonSets are ideal for applications that need to run on all nodes or a subset of nodes, like log collection, monitoring, and node management tasks.

DaemonSet


Use Cases:
Logging
fluentd
logstash
Monitoring
Prometheus
DATADOG
Networking
cilium
weave
net

DaemonSet YAML file

In this example:

- **apiVersion: apps/v1** specifies the API version.
- **kind: DaemonSet** indicates that this is a DaemonSet resource.
- **Metadata:** contains the name of the DaemonSet.
- **Spec:** defines the desired state of the DaemonSet, including the pod template.
- **Selector:** specifies the label that the DaemonSet uses to identify the pods it manages.
- **Template:** describes the pods that will be created by the DaemonSet.

DeamonSet

example of a DaemonSet YAML:

```
apiVersion: apps/v1
kind: DaemonSet
metadata:
 name: example-daemonset
spec:
 selector:
  matchLabels:
   app: nginx
 template:
  metadata:
   labels:
    app: nginx
  spec:
   containers:
   - name: example-container
     image: nginx
```

DaemonSet

Functionality of DaemonSets:

1. Pod Scheduling:

- **All Nodes**: By default, DaemonSets ensure that a copy of the specified pod runs on every node in the cluster.
- **Subset of Nodes**: You can use node selectors, node affinity rules, and tolerations to control which nodes the DaemonSet pods are scheduled on.

2. **Automatic Updates**:

- When a new node is added to the cluster, the DaemonSet controller automatically schedules the DaemonSet pod on the new node.
- If a node is removed from the cluster, the DaemonSet controller ensures that the pod on the removed node is cleaned up.

3. **Pod Management**:

- DaemonSets handle the creation, scheduling, and deletion of pods to match the desired state specified in the DaemonSet configuration.
- Rolling updates are supported, allowing you to update the DaemonSet configuration and roll out changes gradually across the cluster.

DaemonSet

Use Cases:

1. **Monitoring Agents**:
   - Deploy agents like Prometheus Node Exporter or Fluentd on all nodes to collect metrics and logs.
2. **Network Services**:
   - Run network services such as CNI (Container Network Interface) plugins that need to be present on all nodes for networking purposes.
3. **Security Agents**:
   - Deploy security monitoring or compliance agents on all nodes to ensure cluster-wide security.
4. **System Upgrades and Maintenance**:
   - Use DaemonSets to roll out system updates or perform maintenance tasks uniformly across all nodes.

DaemonSet

Commands :

1. Create a DaemonSet from a YAML file:

 # kubectl apply -f <daemonset.yaml>.

2. To delete the DaemonSet:

#kubectl delete daemonset <daemonset-name>

3. List DaemonSets:

#kubectl get daemonsets

4. Describe a DaemonSet:

#kubectl describe daemonset <daemonset-name>

5. View pods created by a DaemonSet

#kubectl get pods -l app=<label-value>

Scheduling

Scheduling:

scheduling refers to the process of assigning pods (containers) to nodes in a cluster. The scheduler is a component of the Kubernetes control plane responsible for making these decisions.

Node Selection:

The scheduler takes into account factors like available resources, node affinities/anti-affinities, taints/tolerations, and other user-defined constraints.

Node Affinity/Anti-Affinity: Allows you to constrain a pod to run on nodes with certain labels.

Taints/Tolerations: Taints are used to repel pods, while tolerations are used by pods to indicate their willingness to accept pods with certain taints.

example

- #nginx-pod-with-node-name.yaml

```
apiVersion: v1
kind: Pod
metadata:
  name: my-nginx-pod
  labels:
    app: nginx
spec:
  nodeName: your-node-name
  containers:
  - name: nginx-container
    image: nginx:latest
    ports:
    - containerPort: 80
```

- #nginx-pod-with-node-selector.yaml

```
apiVersion: v1
kind: Pod
metadata:
  name: my-nginx-pod
  labels:
    app: nginx
spec:
  nodeSelector:
    disktype: ssd
  containers:
  - name: nginx-container
    image: nginx:latest
    ports:
    - containerPort: 80
```

Example:

- **nginx-pod-with-node-affinity.yaml**

```
apiVersion: v1
kind: Pod
metadata:
  name: my-nginx-pod
  labels:
    app: nginx
spec:
  affinity:
    nodeAffinity:
      requiredDuringSchedulingIgnoredDuringExecution:
        nodeSelectorTerms:
        - matchExpressions:
          - key: disktype
            operator: In
            values:
            - ssd
  containers:
  - name: nginx-container
    image: nginx:latest
    ports:
    - containerPort: 80
```

- **nginx-pod-with-node-anti-affinity.yaml**

```
apiVersion: v1
kind: Pod
metadata:
  name: my-nginx-pod
  labels:
    app: nginx
spec:
  affinity:
    podAntiAffinity:
      requiredDuringSchedulingIgnoredDuringExecution:
      - labelSelector:
          matchExpressions:
          - key: app
            operator: In
            values:
            - nginx
        topologyKey: "kubernetes.io/hostname"
  containers:
  - name: nginx-container
    image: nginx:latest
    ports:
    - containerPort: 80
```

Example:

- **nginx-pod-with-toleration.yaml**

```
apiVersion: v1
kind: Pod
metadata:
  name: my-nginx-pod
  labels:
    app: nginx
spec:
  containers:
  - name: nginx-container
    image: nginx:latest
    ports:
    - containerPort: 80
  tolerations:
  - key: "key"
    operator: "Equal"
    value: "value"
    effect: "NoSchedule"
```

kubectl taint nodes <node-name> <key>=<value>:<effect>

Example:

kubectl taint nodes node-1 example-key=value:NoSchedule

node-taint.yaml

```
apiVersion: v1
kind: Node
metadata:
  name: node-1
  labels:
    foo: bar
spec:
  taints:
  - key: example-key
    value: value
    effect: NoSchedule
```

KUBECONFIG

kubeconfig is a configuration file used by Kubernetes to organize information about how to connect to a Kubernetes cluster. It typically contains details such as the cluster's address, authentication details, and other settings. The file is used by tools like kubectl (Kubernetes Command Line Interface) to interact with the cluster

1. **Cluster Configuration**: Describes the Kubernetes cluster, including its API server's address.
2. **User Configuration**: Specifies the user's identity and authentication details. This could include a username and password, client certificate, or token.
3. **Context Configuration**: Connects a user to a specific cluster, defining the combination of cluster and user to use. It also includes the namespace within the cluster.
4. The kubeconfig file can be located at ~/.kube/config by default, but you can specify a different file using the KUBECONFIG environment variable.

Example:

```
apiVersion: v1
kind: Config

clusters:
- name: my-cluster
  cluster:
    server: https://cluster-api-server-url
    certificate-authority: /path/to/ca.crt

users:
- name: my-user
  user:
    client-certificate: /path/to/client.crt
    client-key: /path/to/client.key

contexts:
- name: my-context
  context:
    cluster: my-cluster
    user: my-user
    namespace: my-namespace

current-context: my-context
```

Storage

Volume:

A volume is a directory that may be backed by storage. It allows data to persist across the lifetime of a pod.

Storage class:

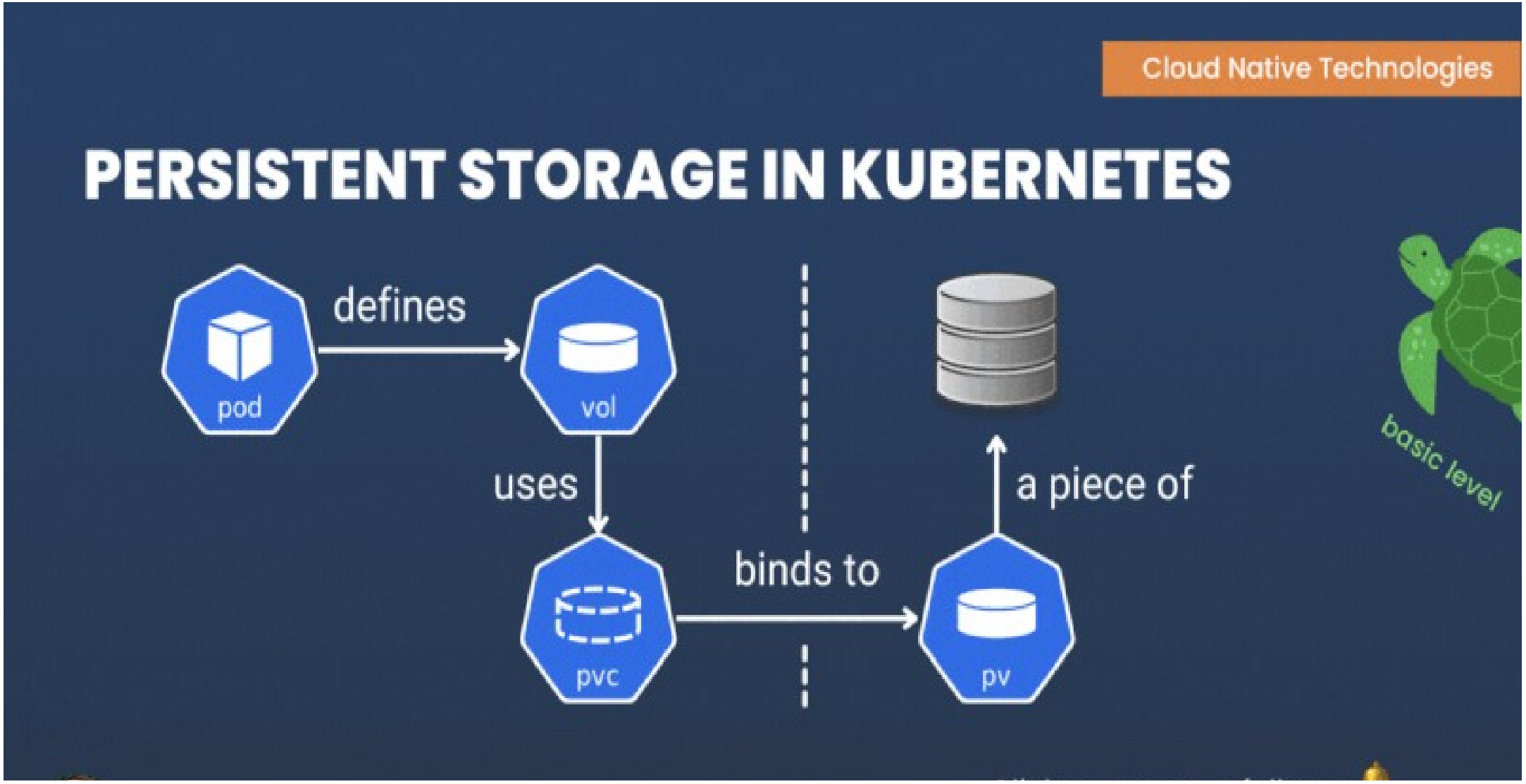
StorageClass is an abstraction layer that defines the characteristics and provisioning mechanisms of the underlying storage for Persistent Volumes (PVs). It allows you to dynamically provision storage resources without having to manually create PVs.

Persistent Volume:

Persistent Volume (PV) is a piece of storage in the cluster that has been manually provisioned or dynamically provisioned using a StorageClass. PVs are used to store data in a way that allows it to persist across pod restarts and rescheduling

Persistent Volume (PV) and Persistent Volume Claim (PVC):**

PV and PVC are abstractions for managing storage in a cluster. A PV is a piece of storage, while a PVC is a request for storage.


Cloud Native Technologies
PERSISTENT STORAGE IN KUBERNETES
pod
defines
vol
uses
pvc
binds to
a piece of
pv
basic level

VM

Example:

example-SC.yaml

```
apiVersion: storage.k8s.io/v1
kind: StorageClass
metadata:
  name: fast
provisioner: kubernetes.io/aws-ebs
parameters:
  type: gp2
reclaimPolicy: Retain
```

- **example-pv.yaml**

```
apiVersion: v1
kind: PersistentVolume
metadata:
  name: example-pv
spec:
  capacity:
    storage: 1Gi
  accessModes:
    - ReadWriteOnce
  hostPath:
    path: /data/example
```

- **example-pvc.yaml**

```
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
  name: example-pvc
spec:
  accessModes:
    - ReadWriteOnce
  resources:
    requests:
      storage: 1Gi
```

- **example-pod.yaml**

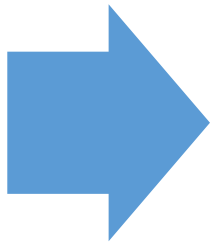
```
apiVersion: v1
kind: Pod
metadata:
  name: example-pod
spec:
  containers:
  - name: example-container
    image: nginx:latest
    volumeMounts:
    - name: storage
      mountPath: /usr/share/nginx/html
  volumes:
  - name: storage
    persistentVolumeClaim:
      claimName: example-pvc
```

Security

- Kubernetes security involves implementing measures to protect containerized applications and the Kubernetes infrastructure itself. Kubernetes is a powerful container orchestration platform, but ensuring the security of your clusters is crucial to prevent unauthorized access, data breaches, and other security threats.
- Main Aspects of Kubernetes security:
- **Cluster Configuration**
- **Network Security**

Cluster Configuration

Secure the cluster by properly configuring access controls and authentication mechanisms. Use strong authentication methods, such as certificates or tokens, and implement role-based access control (RBAC) to restrict user permissions



Role-Based Access Control (RBAC) is a security paradigm used in Kubernetes to control access to resources based on the roles and responsibilities of individual users within a cluster. With RBAC, you can define what actions (such as creating, updating, or deleting resources) users or system components are allowed to perform.

RBAC concepts in Kubernetes

Roles and ClusterRoles:

Role: A set of rules that define permissions within a specific namespace. For example, you can create a role that allows users to list and read pods within a particular namespace.

ClusterRole: Similar to a role, but applies globally across all namespaces in the cluster. ClusterRoles are useful when you need to grant permissions that span multiple namespaces.

RoleBindings and ClusterRoleBindings:

RoleBinding: Binds a role to a user, group, or service account within a specific namespace. It associates a set of permissions defined in a role with a specific identity.

ClusterRoleBinding: Similar to a RoleBinding but applies globally across all namespaces. It binds a ClusterRole to a user, group, or service account at the cluster level.

Subjects:

Subjects are the entities to which roles or cluster roles are bound. They can be users, groups, or service accounts.

User: An individual human user.

Group: A collection of users.

Service Account: An identity used by pods to access the Kubernetes API.

Verbs:

Verbs define the actions that are allowed on resources. Common verbs include "get," "list," "watch," "create," "update," and "delete."

Example:

- # Define a Role named "pod-reader" that allows "get," "list," and "watch" actions on pods.

```
apiVersion: rbac.authorization.k8s.io/v1
kind: Role
metadata:
  namespace: default
  name: pod-reader
rules:
- apiGroups: [""]
  resources: ["pods"]
  verbs: ["get", "list", "watch"]
```

- # Create a RoleBinding to bind the "pod-reader" Role to a specific User.apiVersion: rbac.authorization.k8s.io/v1

```
apiVersion: rbac.authorization.k8s.io/v1
kind: RoleBinding
metadata:
  name: read-pods
  namespace: default
subjects:
- kind: User
  name: "john" # Replace with the actual username
  apiGroup: rbac.authorization.k8s.io
roleRef:
  kind: Role
  name: pod-reader
  apiGroup: rbac.authorization.k8s.io
```

Example:

- # Create a Role named "pod-reader"
- kubectl create role pod-reader --verb=get,list,watch --resource=pods --namespace=default

- # Create a RoleBinding to bind the Role to a specific User
- kubectl create rolebinding read-pods --role=pod-reader --user=john --namespace=default

VM

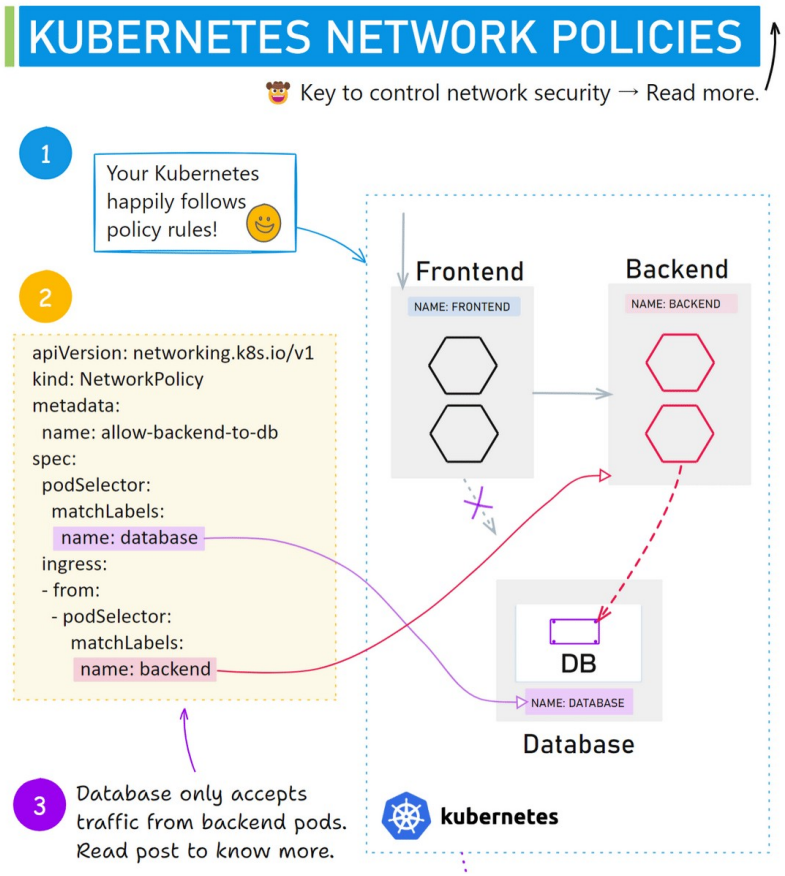
Network Security

Isolate and secure communication between cluster components and nodes. Use network policies to control traffic between pods and define rules for ingress and egress traffic.

Employ tools like Network Policies and implement firewalls to restrict access to the Kubernetes API server.

Network Policies

- **Network Policies**: These are resources that allow you to control the traffic flow between pods. They provide a way to specify how pods are allowed to communicate with each other and other network endpoints
- This helps to isolate pods from each other and to prevent unauthorized and un-necessary access.
- In Kubernetes, Network plugins are important. They are responsible for enforcing the network policy rules
- Some common network plugins that happily support network policies include:
  Calico
  Cilium
  Weave Net



Example

```
# create_ns.yaml
apiVersion: v1
kind: Namespace
metadata:
  name: mynamespace
```

- #pod_web.yaml

```
apiVersion: v1
kind: Pod
metadata:
  name: web-pod-1
  namespace: mynamespace
  labels:
    app: web
spec:
  containers:
  - name: web-container
    image: nginx:latest
```

- #pod_other.yaml

```
apiVersion: v1
kind: Pod
metadata:
  name: other-pod-1
  namespace: mynamespace
  labels:
    app: other
spec:
  containers:
  - name: other-container
    image: busybox:latest
```

Example

- # network_policy.yaml

```
apiVersion: networking.k8s.io/v1
kind: NetworkPolicy
metadata:
  name: allow-web-traffic
  namespace: mynamespace
spec:
  podSelector:
    matchLabels:
      app: web
  ingress:
  - from:
    - podSelector:
        matchLabels:
          app: web
```

```
# Accessing web pod from other pod (allowed)
kubectl exec -it other-pod-1 --namespace=mynamespace -- sh
wget -qO- web-pod-1.mynamespace
# Accessing other pod from web pod (denied)
kubectl exec -it web-pod-1 --namespace=mynamespace -- sh
wget -qO- other-pod-1.mynamespace
```

Deployment

- A *Deployment* provides declarative updates for Pods ReplicaSets.
- You describe a *desired state* in a Deployment, and the Deployment Controller changes the actual state to the desired state at a controlled rate. You can define Deployments to create new ReplicaSets, or to remove existing Deployments and adopt all their resources with new Deployments.
- **The following are typical use cases for Deployments:**
- Create a Deployment to rollout a ReplicaSet. The ReplicaSet creates Pods in the background. Check the status of the rollout to see if it succeeds or not.
- Declare the new state of the Pods by updating the PodTemplateSpec of the Deployment. A new ReplicaSet is created and the Deployment manages moving the Pods from the old ReplicaSet to the new one at a controlled rate. Each new ReplicaSet updates the revision of the Deployment.
- Rollback to an earlier Deployment revision if the current state of the Deployment is not stable. Each rollback updates the revision of the Deployment.
- Scale up the Deployment to facilitate more load.
- Pause the Deployment to apply multiple fixes to its PodTemplateSpec and then resume it to start a new rollout.
- Use the status of the Deployment as an indicator that a rollout has stuck.
- Clean up older ReplicaSets that you don't need anymore

Deployment

- Deployment – on an imperative approach

- Create deployment:
- kubectl create deployment --image=nginx nginx
- Generate Deployment YAML file (-o yaml). Don't create it(--dry-run):
- kubectl create deployment --image=nginx nginx --dry-run -o yaml
- Generate Deployment with 4 Replicas:
- kubectl create deployment nginx --image=nginx --replicas=4
- Scalling deployment:
- kubectl scale deployment nginx --replicas=4
- Update deployment:
- kubectl set image deployment/nginx-deployment nginx=nginx:1.16.1
- Roll back deployment:
- kubectl set image deployment/nginx-deployment nginx=nginx:1.15.1

Deployment

- Deployment – on a declarative approach

```yaml
# nginx-deploy.yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: nginx-deploy
  labels:
    app: nginx-app
spec:
  replicas: 3
  template:
    metadata:
      labels:
        app: nginx-app
    spec:
      containers:
      - name: nginx-container
        image: nginx:1.7.9
        ports:
        - containerPort: 80
  selector:
    matchLabels:
      app: nginx-app
```

Service

- In Kubernetes, a Service is a method for exposing a network application that is running as one or more Pods in your cluster. In other word, An abstract way to expose an application running on a set of Pods as a network service
- A key aim of Services in Kubernetes is that you don't need to modify your existing application to use an unfamiliar service discovery mechanism. You can run code in Pods, whether this is a code designed for a cloud-native world, or an older app you've containerized. You use a Service to make that set of Pods available on the network so that clients can interact with it.
- If you use a Deployment to run your app, that Deployment can create and destroy Pods dynamically.
- Each Pod gets its own IP address (Kubernetes expects network plugins to ensure this). For a given Deployment in your cluster, the set of Pods running in one moment in time could be different from the set of Pods running that application a moment later.
- **Cluster IP** - ClusterIP is the default kubernetes service. This service is created inside a cluster and can only be accessed by other pods in that cluster. So basically, we use this type of service when we want to expose a service to other pods within the same cluster.
- **Nodeport** - NodePort opens a specific port on your node/VM and when that port gets traffic, that traffic is forwarded directly to the service. There are a few limitations and hence its not advised to use NodePort

    - only one service per port

    - You can only use ports 30000-32767

- **Load Balancer** - This is the standard way to expose service to the internet. All the traffic on the port is forwarded to the service. It's designed to assign an external IP to act as a load balancer for the service. There's no filtering, no routing. LoadBalancer uses cloud service. Few limitations with LoadBalancer:

SP - every service exposed will it's own ip address

Service

- **ClusterIP Service Definition:**
  - kubectl create -f nginx-svc-ci.yml
  - **Run below command and You can see the internal-service in the list with a static IP address.**
  - kubectl get svc
  - **To see all the End Points (IP addresses of the Pods which are associated with service)**
  - kubectl describe svc internal-service
  - Remove one of the pod and monitor the Endpoints
  - To access the application you need to use the IP address of service with port number.
  - **Delete the CI service**
  - kubectl delete svc internal-service

```yaml
# Service- ClusterIP
# nginx-svc-ci.yaml
apiVersion: v1
kind: Service
metadata:
  name: internal-service
  labels:
    app: nginx-app
spec:
  selector:
    app: nginx-app
  type: ClusterIP
  ports:
  - port: 80
    targetPort: 80
```

Service

- **Nodeport Service Definition:**
- **Create NodePort service**
- kubectl create -f nginx-svc-np.yml
- kubectl get svc
- kubectl describe svc external-service
- Remove one of the pod and monitor the Endpoints
- To access the application you need to use the IP address of service with port number.
- You can access the application on browser by providing the IP address of you any node with port number 31869 ...IP:31869
- kubectl delete svc external-service

```
# Service- NodePort
# nginx-svc-np.yaml
apiVersion: v1
kind: Service
metadata:
  name: external-service
  labels:
    app: nginx-app
spec:
  selector:
    app: nginx-app
  type: NodePort
  ports:
  - nodePort: 31869
    port: 80
    targetPort: 80
```

Service

- **LoadBalancer Service Definition:**
  - **Create LoadBalancer service**
  - kubectl create -f nginx-svc-lb.yml
  - kubectl get svc
  - The External IP is to be provided by the load balancer (more suitable in cloud based environment) if there is not load balancer then the external ip is in the pending state.
  - kubectl describe svc external-service
  - Remove one of the pod and monitor the Endpoints
  - To access the application you need to use the IP address of service with port number.
  - You can access the application on browser by providing the IP address of you any node with port number 31869 ...IP:31869
  - kubectl delete svc external-service

```yaml
# Service- LoadBalancer
# nginx-svc-lb.yaml
apiVersion: v1
kind: Service
metadata:
  name: external-service
  labels:
    app: nginx-app
spec:
  selector:
    app: nginx-app
  type: LoadBalancer
  ports:
  - nodePort: 31869
    port: 80
    targetPort: 80
```

Service

- **LoadBalancer Service Definition:**
- **Create LoadBalancer service**
- kubectl create -f nginx-svc-lb.yml
- kubectl get svc
- The External IP is to be provided by the load balancer (more suitable in cloud based environment) if there is not load balancer then the external ip is in the pending state.
- kubectl describe svc external-service
- Remove one of the pod and monitor the Endpoints
- To access the application you need to use the IP address of service with port number.
- You can access the application on browser by providing the IP address of you any node with port number 31869 ...IP:31869
- kubectl delete svc external-service

```yaml
# Service- LoadBalancer
# nginx-svc-lb.yaml
apiVersion: v1
kind: Service
metadata:
  name: external-service
  labels:
    app: nginx-app
spec:
  selector:
    app: nginx-app
  type: LoadBalancer
  ports:
  - nodePort: 31869
    port: 80
    targetPort: 80
```

Init Containers

- init containers: specialized containers that run before app containers in a Pod. Init containers can contain utilities or setup scripts not present in an app image.

```yaml
apiVersion: v1
kind: Pod
metadata:
  name: myapp-pod
  labels:
    app: myapp
spec:
  containers:
  - name: myapp-container
    image: busybox:1.28
    command: ['sh', '-c', 'echo The app is running! && sleep 3600']
  initContainers:
  - name: init-myservice
    image: busybox:1.28
    args:
         - /bin/sh
         - -c
         - sleep 20;
```

Monitoring, Logging, Debugging, Troubleshooting

- **Kubernetes Pod logs**
- Kubectl logs <<Pod Name>>
- **To find the Kubernetes cluster events**
- Kubectl events
- **Docker logs**
- docker logs <<container name>>
- **To find the events occurred in docker daemon**
- docker events -- since <<date>>
- **Monitoring**
- Use Metric server implementation to monitor how much memory and CPU utilization for a particular pod or worker node.
- There are other 3rd party tools are also available for monitoring (Prometheus and Garfana)
- **Debugging and Troubleshooting**
- 1. If the Pod is not getting created successfully then run **kubectl describe pod <<podname>>** command to check the events which are occured to created the pod and also check the properties of the pod in case you are getting unexpected value of any attribute
- 2. If you want to remove a pod under Replication controller or Deployment or a Service then run **kubectl get pods -o wide** command and identify the pod which is required to be deleted and delete with the help of **kubectl delete pod <<pod name>>** command
- 3. If a Pod is in running state but not working as per expected result, In that case check the log entries of the Pod
- 4. If a Pod is in the Pending state for a long time, In that case, check the worker node statuses by using **kubectl get nodes** command and check all nodes are in Ready State or not
- 5. If a Pod is not getting created on a particular node. In that case, check the node is tainted or not.

QA