Kubernetes (K8S)

Nguyễn Thanh Quân - ntquan@fit.hcmus.edu.vn

Agenda

---

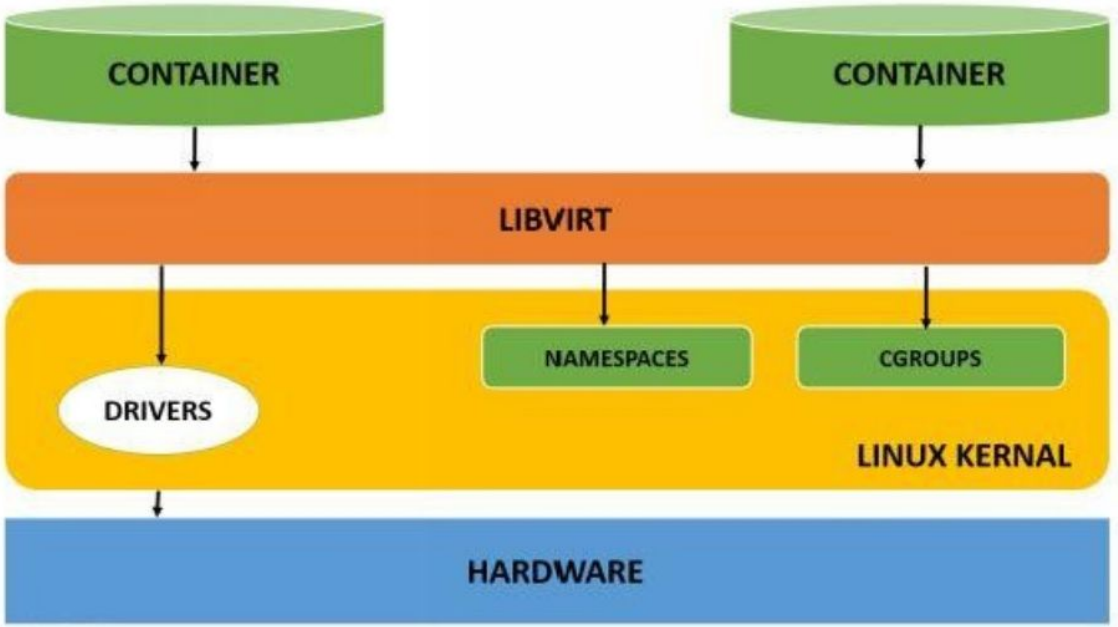
1. Linux Containers, Docker, Podman, Docker Swarm
2. Introduction to Kubernetes
3. Kubernetes Architecture
4. Some Components in Kubernetes
5. Kubernetes Cheatsheet
6. Resource Management in Kubernetes
7. Role-Based Access Control (RBAC)
8. Network Policies
9. Helm

Linux Containers, Docker, Podman, Docker Swarm

Linux Containers

- Namespaces
- Cgroups

  (Control groups)



Linux Containers - Namespace kinds

---

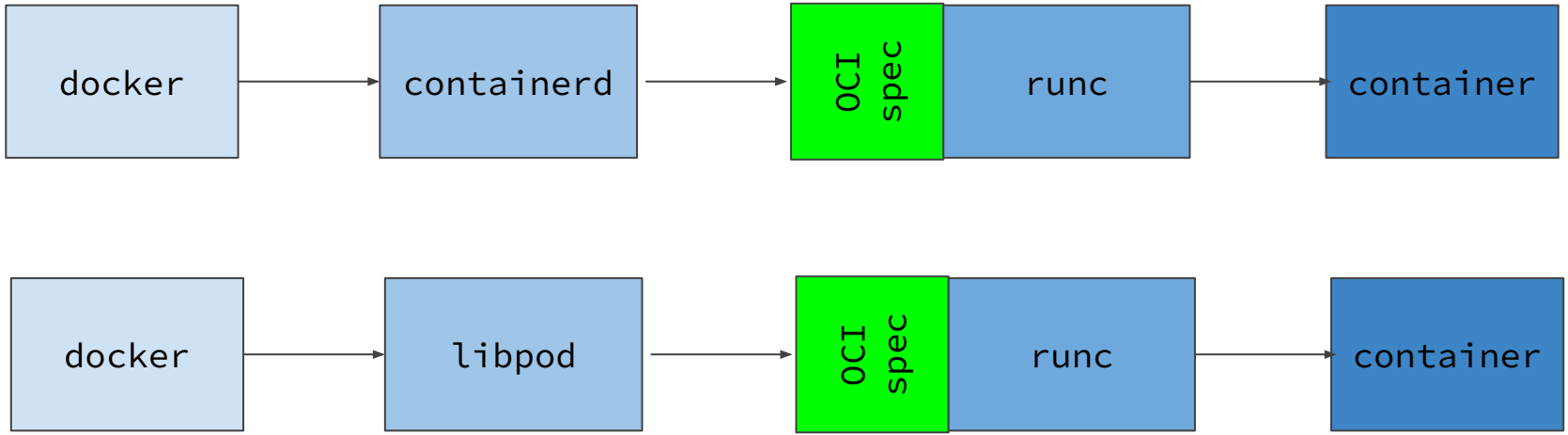
1. Mount (**mnt**): isolates the file system mount points for a set of processes
2. Process ID (**pid**): Every pid namespace forms its own hierarchy and it will be tracked by the kernel
3. Network (**net**): is used for controlling the networks
4. Inter-process Communication (**ipc**): isolates the inter-process communication
5. UTS: provides isolation for hostname and NIS domain name
6. User ID (**user**): isolates the user and group ID namespaces

Linux Containers - Cgroups

Control groups (cgroups): Cgroups are fundamental blocks of making a container. A cgroup allocates and limits resources such as CPU, memory, network I/O that are used by containers. The container engine automatically creates a cgroup filesystem of each type, and sets values for each container when the container is run.

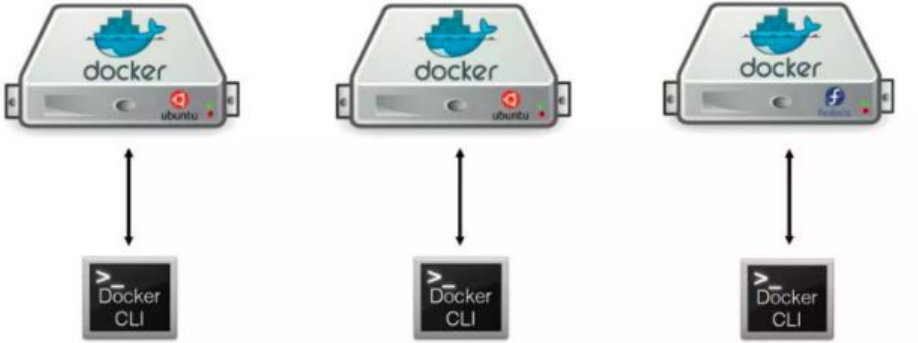
Docker & Podman

Open Container Initiative (OCI)

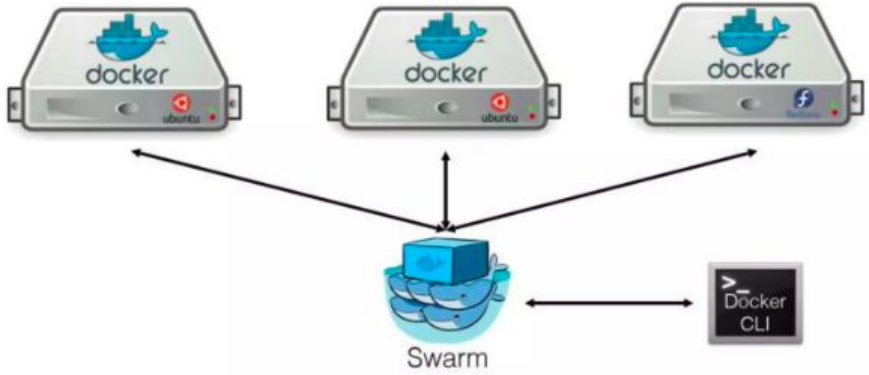


Docker Swarm

Tranditional Mode



Docker Swarm



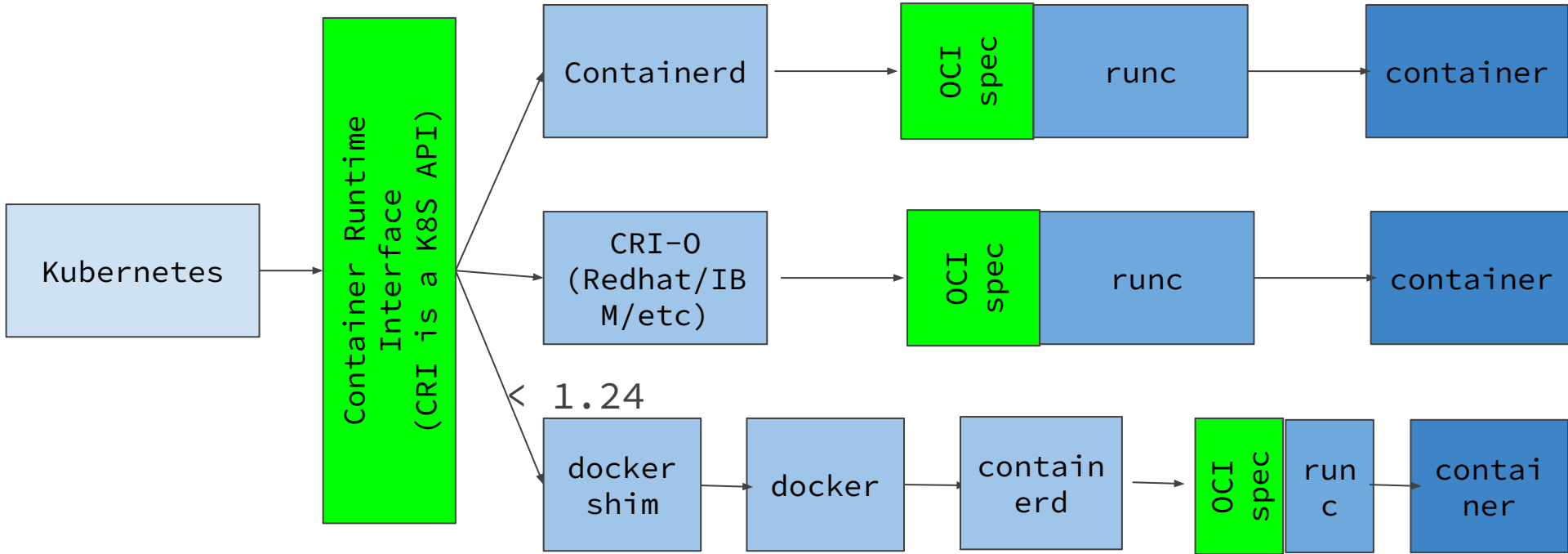
Docker Swarm

- Docker Swarm is straightforward to install, lightweight and easy to use
- Docker Swarm provides automated load balancing within the Docker containers
- Docker Swarm’s automation capabilities are not as robust as those offered by Kubernetes

Introduction to Kubernetes

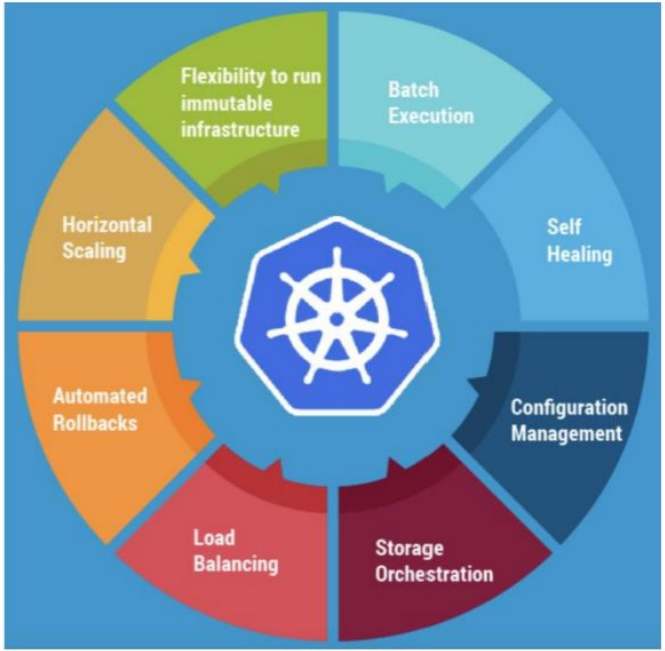
Introduction to Kubernetes

- Kubernetes (k8s) is an open-source system used to automate the deployment, scaling, and management of containerized applications.
- Kubernetes was developed by Google.
- Today, Kubernetes has a large development community and is maintained by the CNCF (Cloud Native Computing Foundation).


Kubernetes
Container Runtime Interface (CRI is a K8S API)
Containerd
OCI spec
runc
container
CRI-O (Redhat/IBM/etc)
OCI spec
runc
container
< 1.24
docker shim
docker
containerd
OCI spec
runc
container

Benefits

- **Scalability and Expandability**: Kubernetes helps automate the distribution and operation of containers in large-scale environments.
- **Self-healing**: Kubernetes can automatically restart or replace failed containers, or move them to other nodes when necessary.
- **Storage and Configuration Management**: Kubernetes allows easy management of storage resources and configurations (ConfigMaps, Secrets).
- **Security and Access Control**: Kubernetes provides robust security methods such as RBAC (Role-Based Access Control), Network Policies, and Service Accounts.



Problems Kubernetes Can Solve

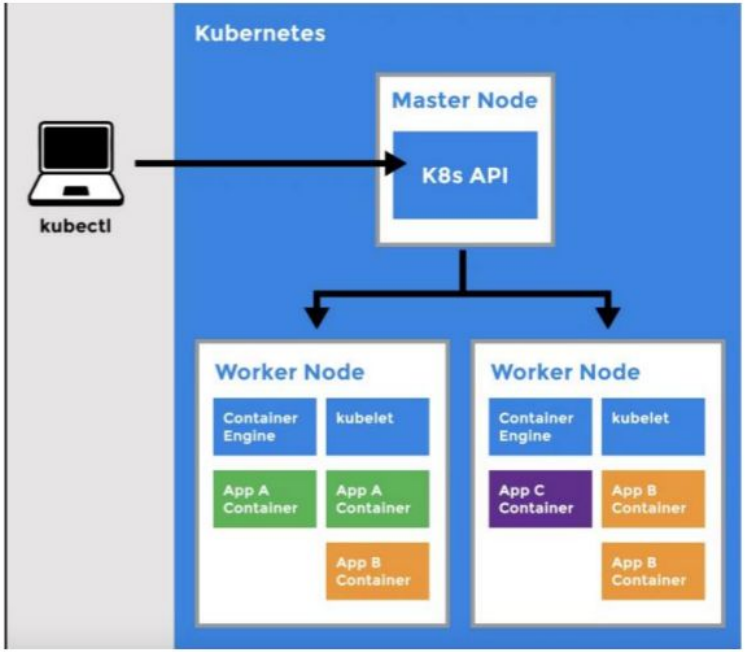
- Managing Large-Scale Containers: Kubernetes helps organize and monitor thousands of containers without facing issues of resource fragmentation.
- Automatic Load Balancing: Kubernetes can automatically distribute workloads across containers, minimizing downtime.
- Recovery and Self-Healing: Automatically detects failed containers or nodes and replaces them automatically.
- Easily Scale Applications:
  - Kubernetes supports auto-scaling, allowing applications to automatically scale up or down based on traffic.
  - Provides Horizontal Pod Autoscaling and Cluster Autoscaling.
- Resource Optimization: Efficiently allocates resources between containers and nodes.
- Storage and Configuration Management:
  - Supports various types of storage, such as Persistent Volumes (PV) and Persistent Volume Claims (PVC).
  - Provides configuration management mechanisms like ConfigMaps and Secrets.

Kubernetes Architecture

Kubernetes Architecture

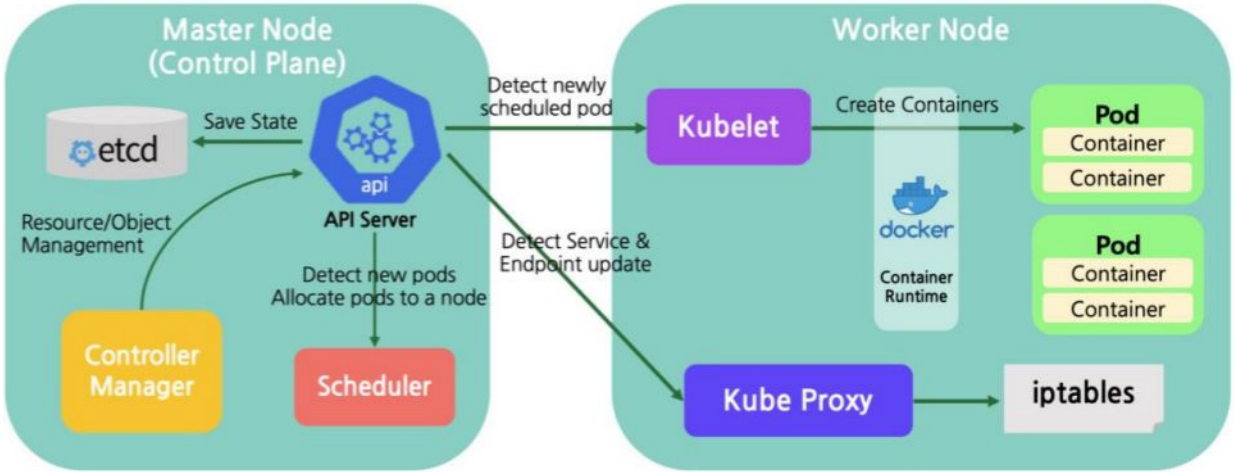
Kubernetes has a client-server architecture, consisting of two main parts:

- Control Plane (Master Node)
- Worker Nodes.



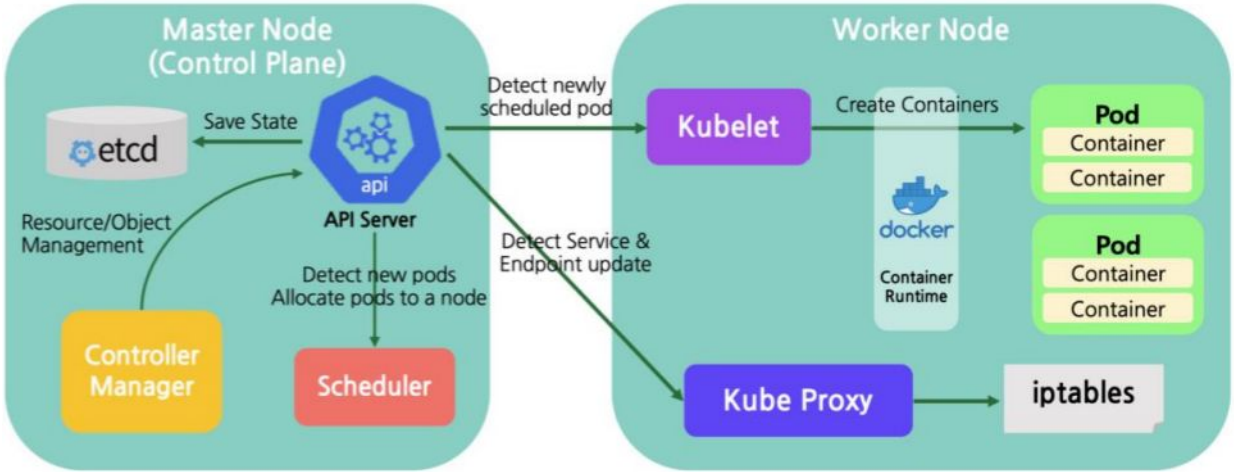
Control Plane

- **API Server:** The central point for all requests; it processes commands and interacts with other components of Kubernetes.
- **Scheduler:** Selects the appropriate node to deploy pods.



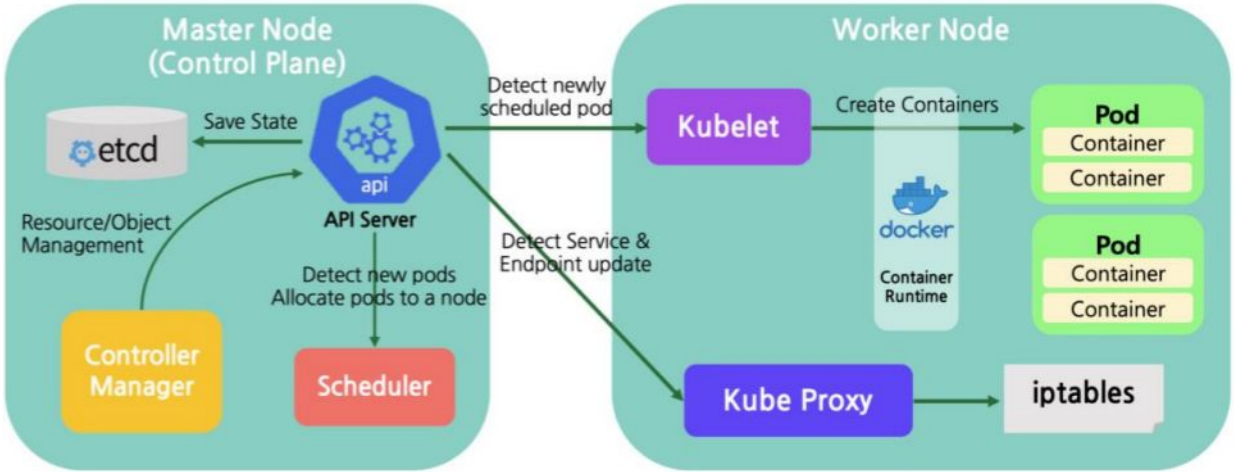
Control Plane

- **Controller Manager:** Monitors the state of resources and ensures that the actual state is always updated.
- **etcd:** A distributed database that stores all configuration data and the state of Kubernetes.



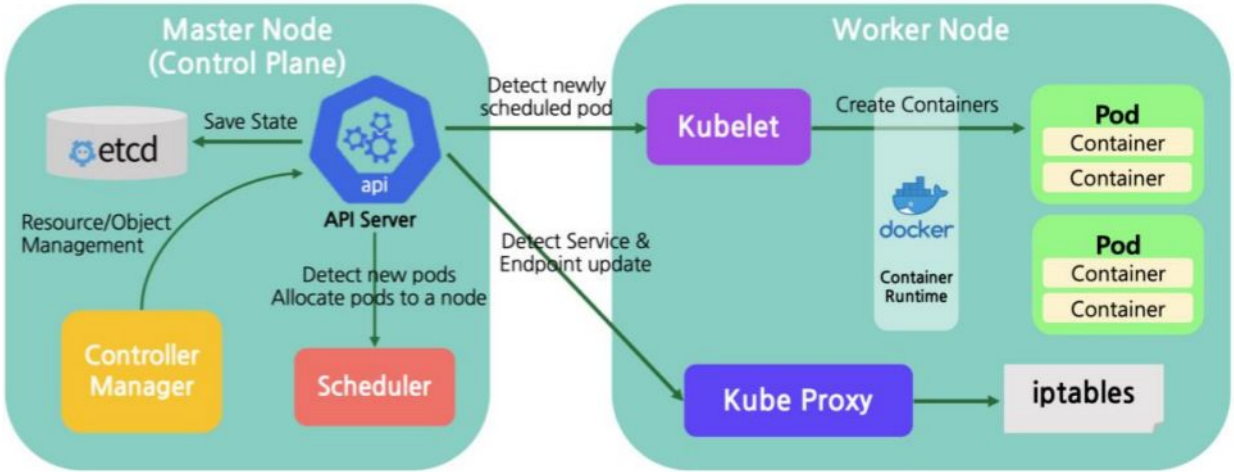
Worker Node

- **Kubelet:** An agent running on each node, ensuring that containers are running and managed properly.
- **Kube Proxy:** Manages networking between Pods and provides load balancing mechanisms.



Worker Node

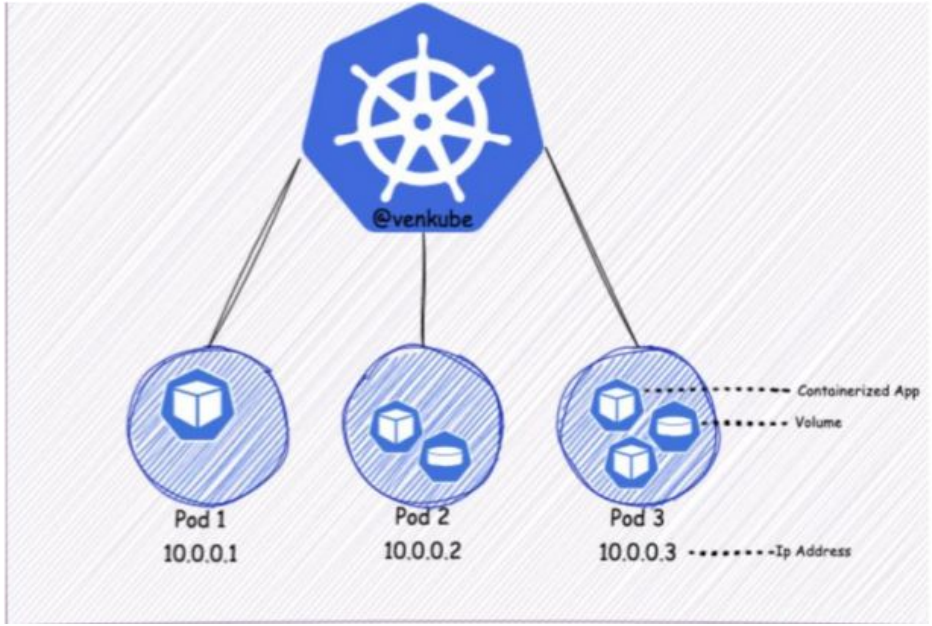
- **Container Runtime**: The software used to run containers (e.g., Docker, containerd).



Some Components in Kubernetes

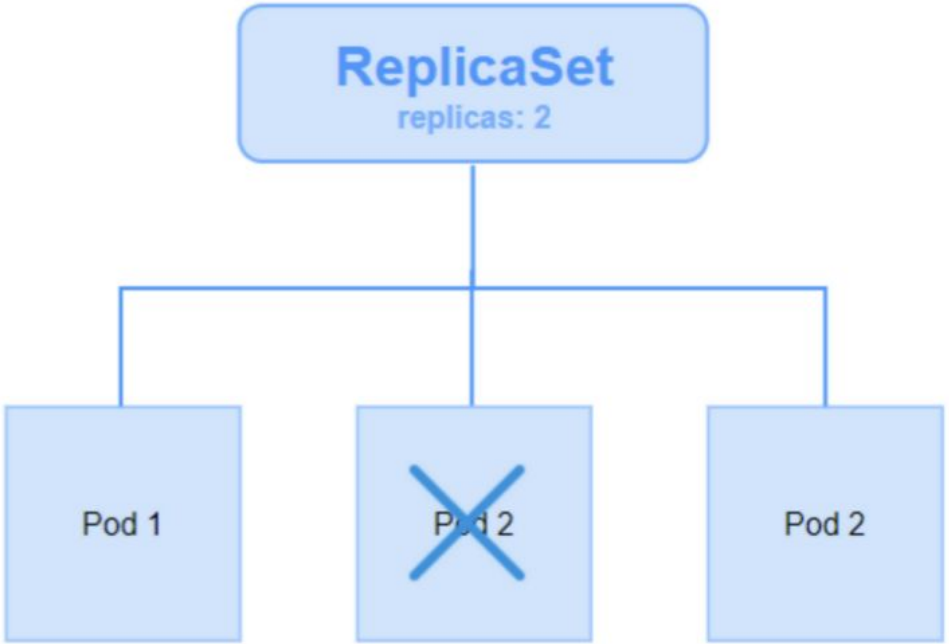
1. Pod

- A **Pod** is the basic deployment unit in Kubernetes.
- A **Pod** can contain one or more containers, and the containers within the same Pod share resources and networking.
- Kubernetes uses objects such as **Deployment** and **ReplicaSet** to manage Pods.



2. ReplicaSet

- ReplicaSet is an object in Kubernetes used to ensure that the number of Pod replicas is always maintained at a specified level at any given time.
- If a Pod fails, ReplicaSet will automatically create a new Pod to replace it.



2. ReplicaSet

- ReplicaSet uses a selector to find and manage the appropriate Pods. Only Pods with labels matching the selector are managed by the ReplicaSet.
- The Pod Template is used to define how new Pods are created when needed.
- ReplicaSet manages Pods, but the Pods themselves are independent entities. When the ReplicaSet is deleted, the Pods are not automatically removed.

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

2. ReplicaSet

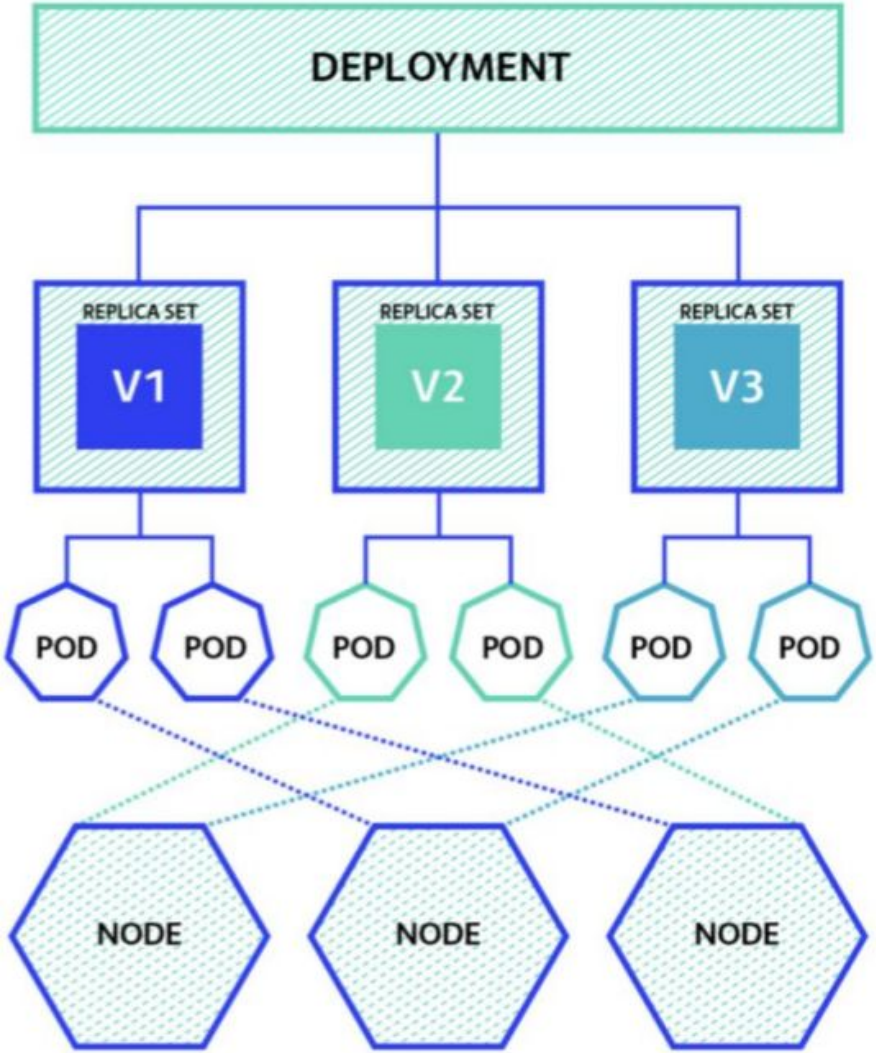
A ReplicaSet is defined in a YAML file with the following main components:

- **metadata**: General information (name, labels, etc.).
- **spec**:
  - **replicas**: The number of Pod replicas.
  - **selector**: The conditions used to manage the Pods.
  - **template**: The configuration for creating new Pods.

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

3. Deployment

Deployment is an API in Kubernetes used to manage the deployment, updating, and scaling of containerized applications.



3. Deployment

- **Managing ReplicaSet:**
  - Deployment creates and manages ReplicaSets, which in turn create and manage Pods.
  - If the Deployment is deleted, its associated ReplicaSet and Pods will also be deleted.
- **Application Updates:** Supports Rolling Updates and Rollback to safely update applications.
- **Scaling Applications:** Easily scale applications by changing the number of Pods.
- **Self-Healing:** Deployment automatically ensures that the desired number of Pods is always maintained.

3. Deployment

A Deployment is defined in a YAML file with the following main components:

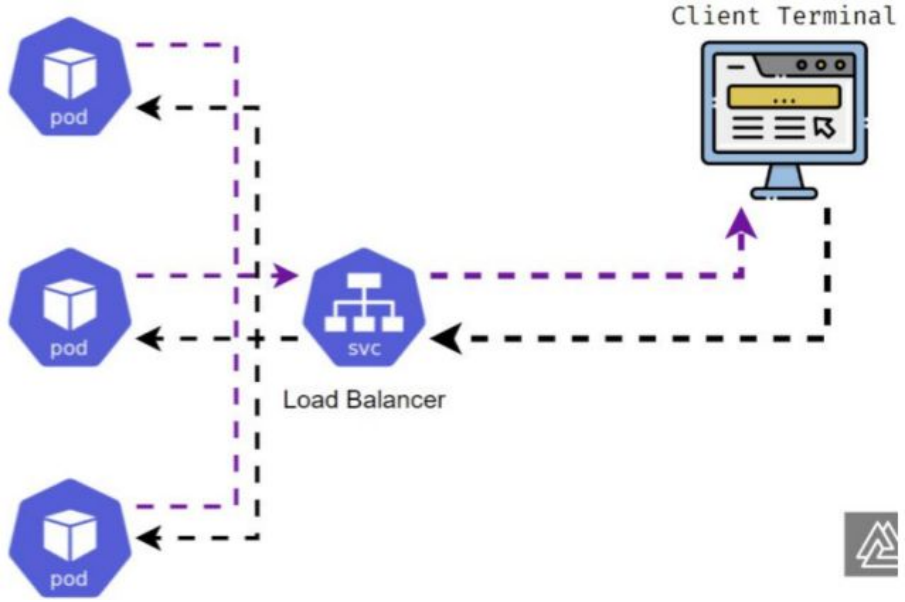
- **metadata**: General information (name, labels, etc.).
- **spec**:
  - **replicas**: The number of Pod replicas.
  - **selector**: The conditions used to manage the Pods.
  - **template**: Defines the configuration of the Pod.

```yaml
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:1.21
        ports:
        - containerPort: 80
```

4. Service

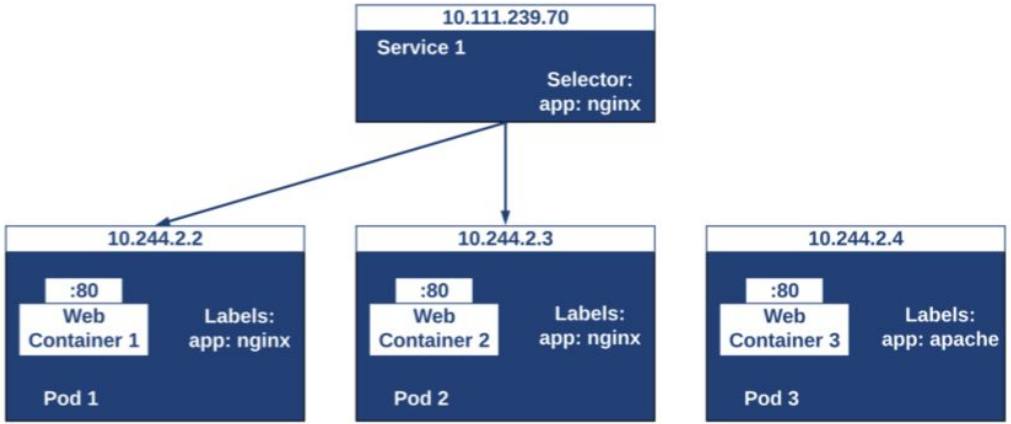
Service is an object in Kubernetes that provides a way to connect and communicate between Pods, or between Pods and the outside world.

A Service acts as a "router", abstracting the communication and ensuring that traffic is forwarded to the correct Pods.



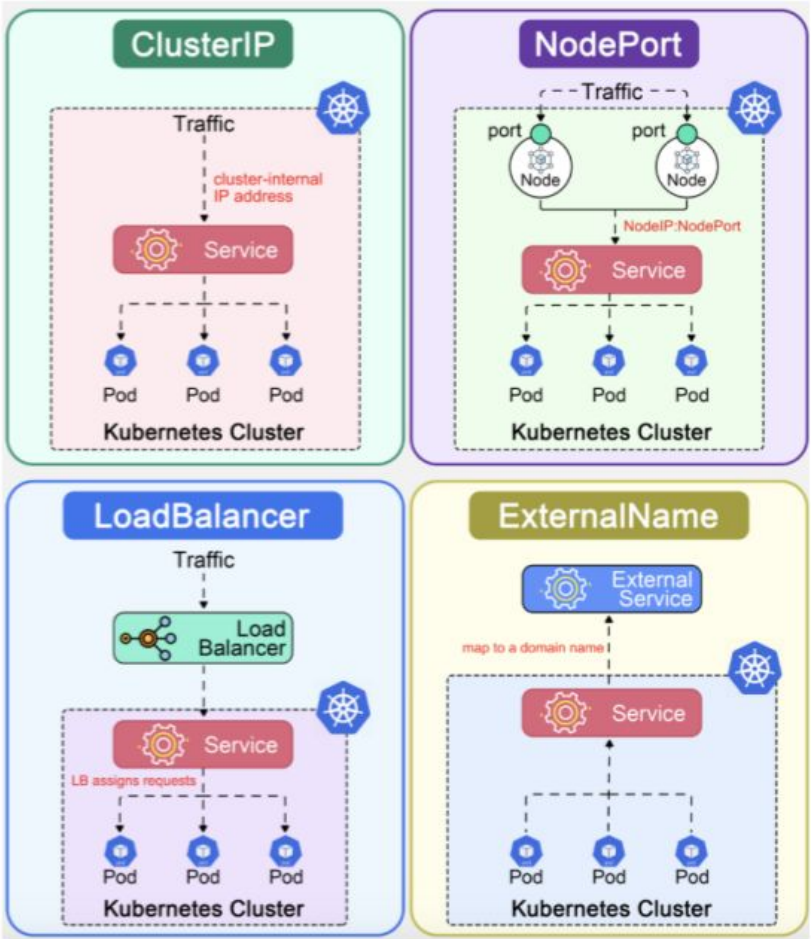
4. Service

- **Selector:**
  - Used to select the Pods that the Service will forward traffic to.
  - The Service uses the labels of the Pods to find matching Pods.
- **Endpoints:**
  - A list of the IP addresses of the Pods managed by the Service.
  - Endpoints are automatically updated when Pods are deleted or newly added.



4. Service

- Service Types:
  - **ClusterIP** (default): Accessible only from within the cluster.
  - **NodePort**: Opens a port on each Node to allow access from outside the cluster.
  - **LoadBalancer**: Integrates with cloud providers to create an external Load Balancer.
  - **ExternalName**: Connects to external services outside of Kubernetes using DNS.



4. Service

Structure of a Service

A Service is defined in a YAML file with the following main components:

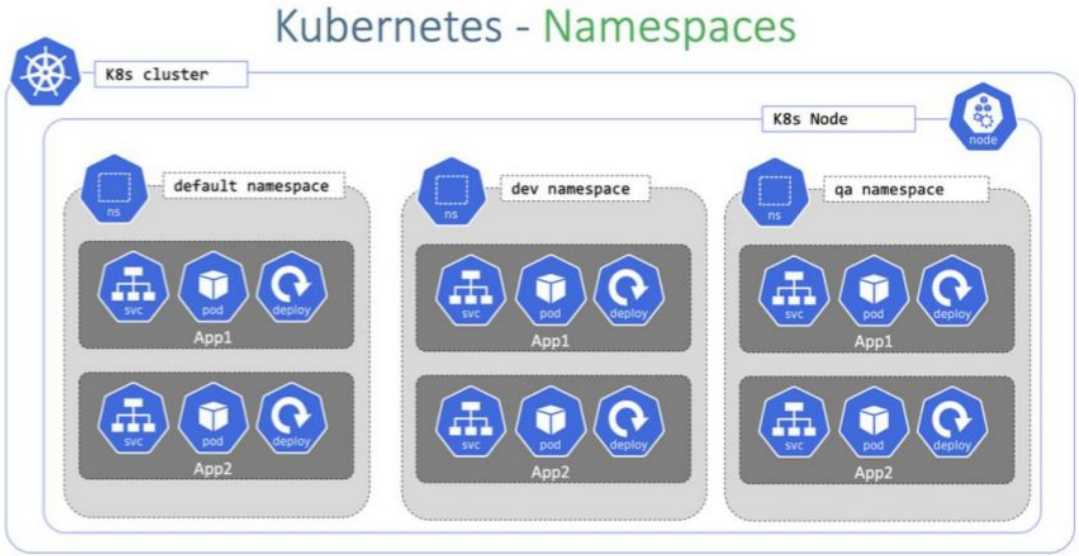
- **metadata**: General information (name, labels, etc.).
- **spec**:
  - **selector**: Labels used to find the matching Pods.
  - **ports**: The communication ports of the Service.
  - **type**: The type of Service (ClusterIP, NodePort, LoadBalancer, ExternalName).

```yaml
apiVersion: v1
kind: Service
metadata:
  name: nginx-service
spec:
  selector:
    app: nginx
  ports:
  - protocol: TCP
    port: 80
    targetPort: 80
  type: ClusterIP
```

5. Namespace

Namespace helps divide resources and objects in Kubernetes into independent workspaces.

Namespaces separate resources (Pods, Services, Deployments, etc.) within a cluster to avoid name conflicts or to manage resources by group.



5. Namespace

- Practical Applications
  - Separate development, testing, and production environments (dev, qa).
  - Manage resources by application group or user group.
- Some resources are not tied to a Namespace, such as:
  - Nodes
  - PersistentVolumes
  - ClusterRoles
- Default Namespaces in Kubernetes
  - **default**:
    - This is the default Namespace when no specific Namespace is specified.
    - Suitable for applications that do not require resource partitioning.
  - **kube-system**:
    - Contains the resources and Pods essential for the operation of Kubernetes (e.g., kube-dns, kube-proxy, coredns).

6. Persistent Volume (PV) và Persistent Volume Claim (PVC)

Persistent Volume (PV) is a storage resource in Kubernetes that is managed by the cluster.

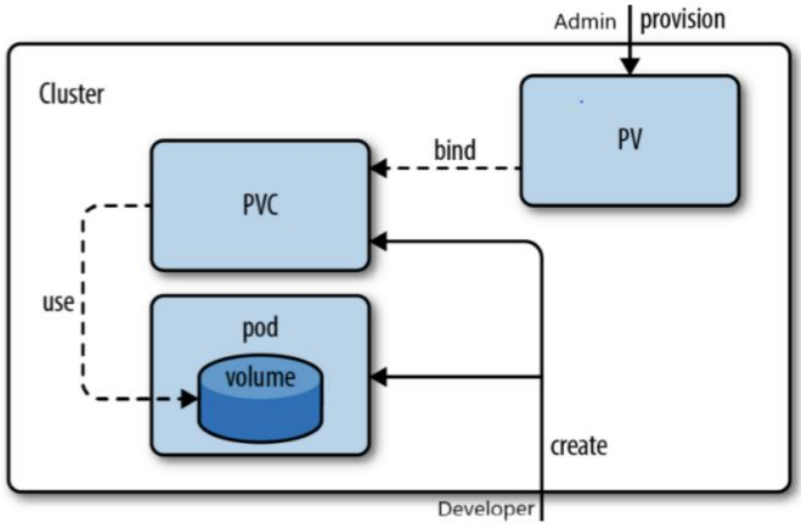
- Provides long-term data storage for Pods, even if a Pod is deleted or replaced.
- By default, Pods have ephemeral storage, meaning data is lost when the Pod is deleted.
- PV allows data storage to be independent of the Pod's lifecycle.

Persistent Volume Claim (PVC) is a user request to consume storage resources (Persistent Volume).

- PVC specifies requirements such as capacity, access modes, and storage class.

The Relationship Between PV and PVC

- **PV** (Persistent Volume): Provided by the cluster administrator.
- **PVC** (Persistent Volume Claim): Requested by the user. Kubernetes automatically binds a PVC to a suitable PV based on the PVC's requirements.



6. Persistent Volume (PV) và Persistent Volume Claim (PVC)

Components of Persistent Volume (PV)

- Storage Classes:
  - Define the type of storage (SSD, HDD, cloud storage, etc.).
  - Kubernetes uses StorageClass for automatically creating PVs (Dynamic Provisioning).
- Access Modes:
  - ReadWriteOnce (RWO): Only one Pod can read/write at a time.
  - ReadOnlyMany (ROX): Multiple Pods can read, but none can write.
  - ReadWriteMany (RWX): Multiple Pods can read and write simultaneously.
- Reclaim Policy:
  - Retain: The PV is not deleted after the PVC is deleted.
  - Delete: The PV is automatically deleted when the PVC is deleted.

6. Persistent Volume (PV) và Persistent Volume Claim (PVC)

Example

PV

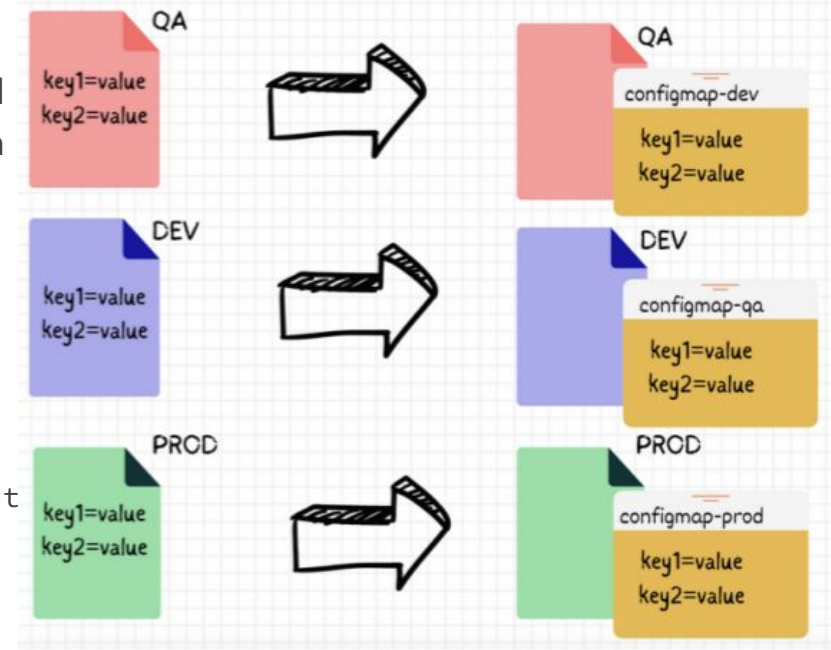
```yaml
apiVersion: v1
kind: PersistentVolume
metadata:
  name: pv-example
spec:
  capacity:
    storage: 1Gi
  accessModes:
    - ReadWriteOnce
  persistentVolumeReclaimPolicy: Retain
  hostPath:
    path: "/mnt/data"
```

PVC

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

7. ConfigMap

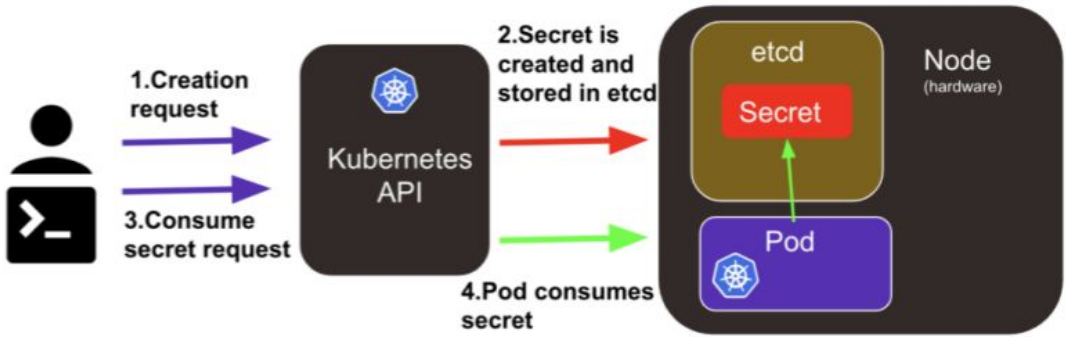
- ConfigMap is an object in Kubernetes used to store non-sensitive configuration data in key-value pairs.
- The configuration stored in a ConfigMap can be used by Pods or Containers within Kubernetes.
- ConfigMap helps separate configuration from the application source code.
    - It allows for easy configuration changes without needing to update the container image.
    - It provides centralized configuration management.



8. Secrets

Secret is an object in Kubernetes used to store and manage sensitive information (such as passwords, API tokens, or SSL certificates).

Secret allows you to avoid storing sensitive data directly in YAML files or source code.



8. Secrets

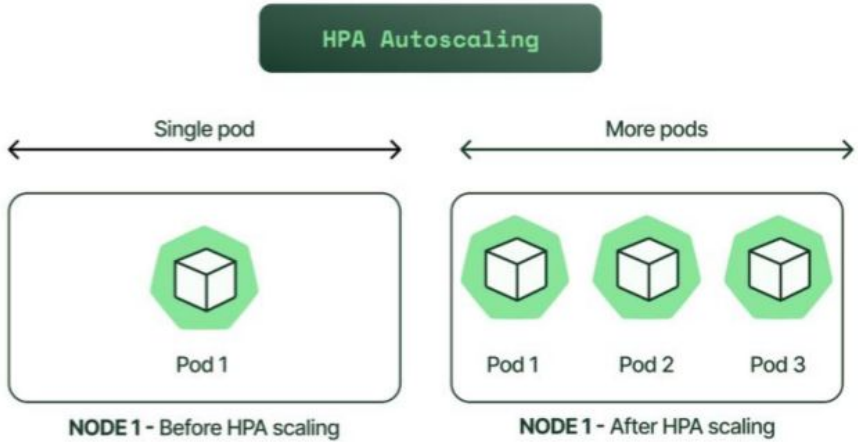
Types of Secret

- **Opaque**: The default type, used to store arbitrary key-value pairs.
- **kubernetes.io/dockerconfigjson**: Stores Docker authentication information for pulling images from a private registry.
- **TLS**: Stores a TLS key pair (certificate and private key).
- **Service Account Token**: A Secret automatically created by Kubernetes to authenticate with the API server.

9. Horizontal Pod Autoscaling (HPA)

Horizontal Pod Autoscaler (HPA) is a feature in Kubernetes used to automatically adjust the number of Pods in a Deployment, ReplicaSet, or StatefulSet based on resource usage (CPU, memory) or custom metrics.

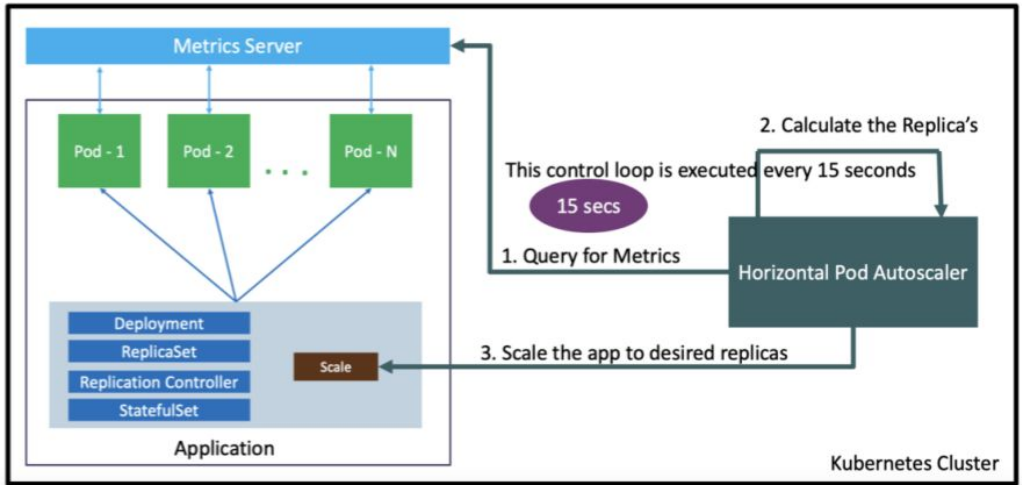
HPA helps applications scale automatically according to demand, ensuring performance and optimizing resource utilization.



9. Horizontal Pod Autoscaling (HPA)

How HPA Works

- Data from Metrics Server: HPA relies on the Metrics Server to collect information about resource usage (CPU, memory, custom metrics).
- Operational Process:
  - Collect metrics data from the Metrics Server.
  - Compare actual values with the configured thresholds.
  - Adjust the number of Pods to maintain the metric value close to the target threshold.



9. Horizontal Pod Autoscaling (HPA)

YAML of HPA

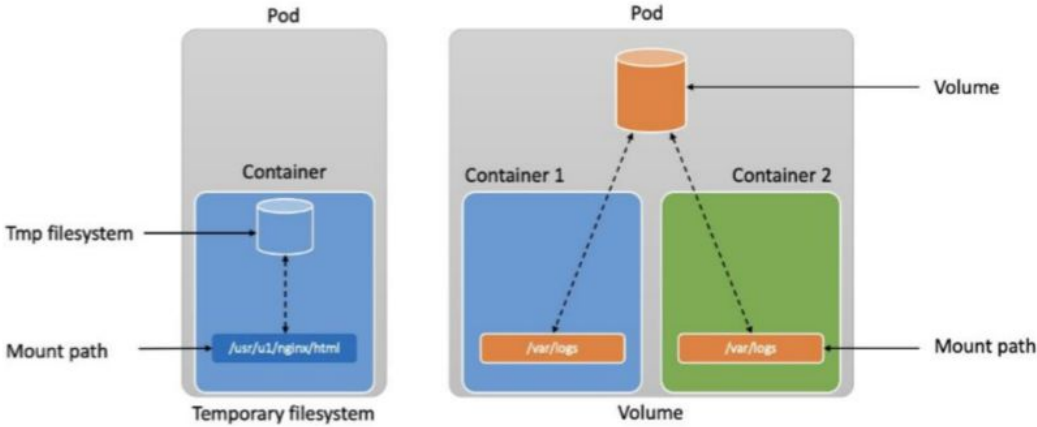
- **scaleTargetRef**: The object that HPA scales (Deployment, ReplicaSet, StatefulSet).
- **metrics**: The metrics used to adjust the number of Pods (CPU, memory, or custom metrics).
- **minReplicas / maxReplicas**: The minimum and maximum limits for the number of Pods.

```yaml
apiVersion: autoscaling/v2
kind: HorizontalPodAutoscaler
metadata:
  name: nginx-hpa
spec:
  scaleTargetRef:
    apiVersion: apps/v1
    kind: Deployment
    name: nginx-deployment
  minReplicas: 2
  maxReplicas: 10
  metrics:
  - type: Resource
    resource:
      name: cpu
      target:
        type: Utilization
        averageUtilization: 50
```

10. Volume

A Volume in Kubernetes is a storage mechanism that allows containers within a Pod to:

- Share data between containers.
- Store data temporarily or persistently.
- Each Volume is attached to a Pod and can be shared by all containers within that Pod.
- Volumes are defined in the Pod specification and are mounted into the containers through a specified path.



10. Volume

Types of Volumes in Kubernetes

- **emptyDir**
  - A temporary Volume that is created when a Pod starts and is deleted when the Pod stops.
  - Commonly used to share data between containers within the same Pod.

```yaml
apiVersion: v1
kind: Pod
metadata:
  name: emptydir-example
spec:
  containers:
  - name: nginx
    image: nginx
    volumeMounts:
    - mountPath: "/data"
      name: temp-storage
  volumes:
  - name: temp-storage
    emptyDir: {}
```

10. Volume

Types of Volumes in Kubernetes

- **PersistentVolume** (PV):
  - Represents a piece of physical storage (NFS, AWS EBS, GCE PD, etc.) within the cluster.
- **PersistentVolumeClaim** (PVC):
  - A user's request to use a PersistentVolume (PV).

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

```yaml
apiVersion: v1
kind: Pod
metadata:
  name: pvc-example
spec:
  containers:
  - name: nginx
    image: nginx
    volumeMounts:
    - mountPath: "/data"
      name: storage
  volumes:
  - name: storage
    persistentVolumeClaim:
      claimName: pvc-example
```

```yaml
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
  name: pvc-example
spec:
  accessModes:
    - ReadWriteOnce
  resources:
    requests:
      storage: 500Mi
```

10. Volume

Types of Volumes in Kubernetes

- **ConfigMap**: Use a ConfigMap as a Volume to provide configuration files to containers.

```yaml
apiVersion: v1
kind: Pod
metadata:
  name: configmap-example
spec:
  containers:
  - name: nginx
    image: nginx
    volumeMounts:
    - mountPath: "/etc/config"
      name: config-volume
  volumes:
  - name: config-volume
    configMap:
      name: my-configmap
```

10. Volume

Types of Volumes in Kubernetes

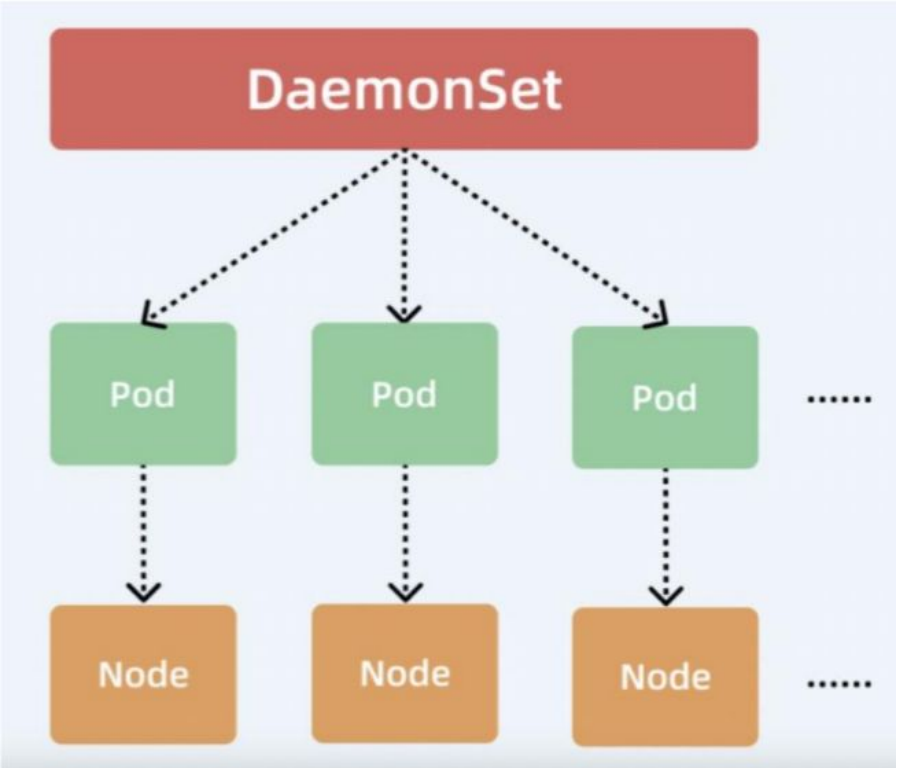
- **Secret**: Use a Secret as a Volume to provide sensitive data (such as passwords, tokens) to containers.

```yaml
apiVersion: v1
kind: Pod
metadata:
  name: secret-example
spec:
  containers:
  - name: nginx
    image: nginx
    volumeMounts:
    - mountPath: "/etc/secret"
      name: secret-volume
  volumes:
  - name: secret-volume
    secret:
      secretName: my-secret
```

11. Daemonset

DaemonSet is an object in Kubernetes that ensures a Pod runs on every Node in the cluster.

- When a new Node is added to the cluster, the DaemonSet automatically adds a Pod to that Node.
- If a Node is removed from the cluster, the DaemonSet will delete the Pod from that Node.



11. Daemonset

Common use cases for DaemonSet:

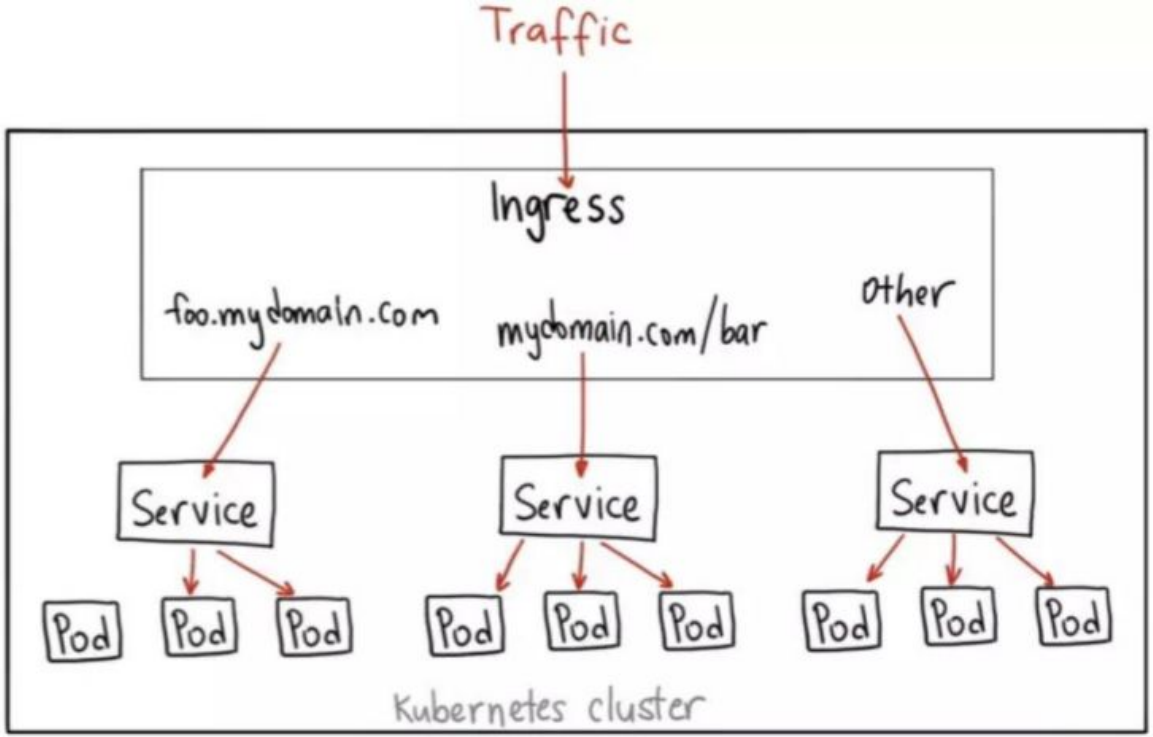
- **Logging**: Deploy log collectors like Fluentd on each Node.
- **Monitoring**: Run monitoring tools such as Prometheus Node Exporter on every Node.
- **Networking**: Install network plugins like Calico, Flannel, Weave Net, or Cilium.
- **Security**: Run security monitoring agents on all Nodes.

11. Ingress

Ingress is an object in Kubernetes that allows you to manage HTTP and HTTPS traffic to applications running inside the cluster.

Ingress provides capabilities such as:

- Routing traffic based on hostname and URL path.
- Establishing HTTPS connections using TLS.
- Centralized management of external traffic into the cluster.



11. Ingress

Components of Ingress

- Ingress Resource: defines the HTTP/HTTPS routing rules:
  - Hostname routing: Routes traffic based on hostname (e.g., example.com).
  - Path-based routing: Routes traffic based on URL paths (e.g., /api, /frontend).
- Ingress Controller
  - The Ingress Resource only works when there is an active Ingress Controller.
  - The Ingress Controller processes Ingress Resources and routes traffic according to the defined rules.
  - Popular Ingress Controllers
    - NGINX Ingress Controller (the most commonly used)
    - Traefik
    - HAProxy

11. Ingress

Path-based Routing

```yaml
spec:
  rules:
  - host: example.com
    http:
      paths:
      - path: /api
        pathType: Prefix
        backend:
          service:
            name: api-service
            port:
              number: 8080
      - path: /web
        pathType: Prefix
        backend:
          service:
            name: web-service
            port:
              number: 80
```

Host-based Routing

```yaml
spec:
  rules:
  - host: api.example.com
    http:
      paths:
      - path: /
        pathType: Prefix
        backend:
          service:
            name: api-service
            port:
              number: 8080
  - host: web.example.com
    http:
      paths:
      - path: /
        pathType: Prefix
        backend:
          service:
            name: web-service
            port:
              number: 80
```

TLS

```yaml
spec:
  tls:
  - hosts:
    - example.com
    secretName: tls-secret
  rules:
  - host: example.com
    http:
      paths:
      - path: /
        pathType: Prefix
        backend:
          service:
            name: my-service
            port:
              number: 80
```

Kubernetes Cheatsheet

Link

Kubernetes Cheatsheet

Creating a Resource

kubectl create namespace [namespace-name]	Create a new namespace
kubectl create –f [filename]	Create a resource from a JSON or YAML file

Applying & Updating a Resource

kubectl apply -f [service-name].yaml	Create a new service with the definition contained in [service-name].yaml

Listing Resources

kubectl get namespaces	Generate a plain-text list of all namespaces
kubectl get pods	Generate a plain-text list of all pods
kubectl get pods -o wide	Generate a detailed plain-text list of all pods
kubectl get pods --field-selector=spec.nodeName=[server-name]	Generate a list of all pods running on a particular node server
kubectl get replicationcontroller [replication-controller-name]	List a specific replication controller in plain text
kubectl get replicationcontroller, services	Generate a plain-text list of all replication controllers and services
kubectl get deamonset	Generate a plain-text list of all daemon sets

Kubernetes Cheatsheet

Printing Container Logs

kubectl logs [pod-name]	Print logs from a pod
kubectl logs -f [pod-name]	Stream logs from a pod

Deleting Resources

kubectl delete -f pod.yaml	Remove a pod using the name and type listed in pod.yaml:
kubectl delete pods,services -l [label-key]=[label-value]	Remove all the pods and services with a specific label:
kubectl delete pods --all	Remove all pods. The command will include uninitialized pods as well

Executing a Command

kubectl exec [pod-name] -- [command]	Receive output from a command run on the first container in a pod:
kubectl exec [pod-name] -c [container-name] -- [command]	Receive output from a command run on a specific container in a pod
kubectl exec -ti [pod-name] -- /bin/bash	Run /bin/bash from a specific pod. The output received comes from the first container

Resource Management in Kubernetes

Main Types of Resources

Kubernetes allows managing CPU, memory, and other resources of the cluster to ensure that applications use resources efficiently.

Main Types of Resources

- CPU:
  - Measured in millicores (1000m = 1 CPU).
  - Example: 500m means using 50% of one CPU core.
- Memory:
  - Measured in bytes (MiB, GiB).
  - Example: 512Mi means 512 MiB of memory.

Resource Requests and Limits

- Requests:
  - The minimum amount of resources that a container requires.
  - Kubernetes uses Requests to schedule Pods on Nodes with enough available resources.
- Limits:
  - The maximum amount of resources a container is allowed to use.
  - A container cannot exceed the specified Limit.

Configuration in Pod

Configuring Requests and Limits in a YAML file

- Request: Kubernetes will ensure that the Node has at least 64Mi of memory and 250m of CPU available to run this Pod.
- Limit: The container will be restricted to a maximum of 128Mi of memory and 500m of CPU.

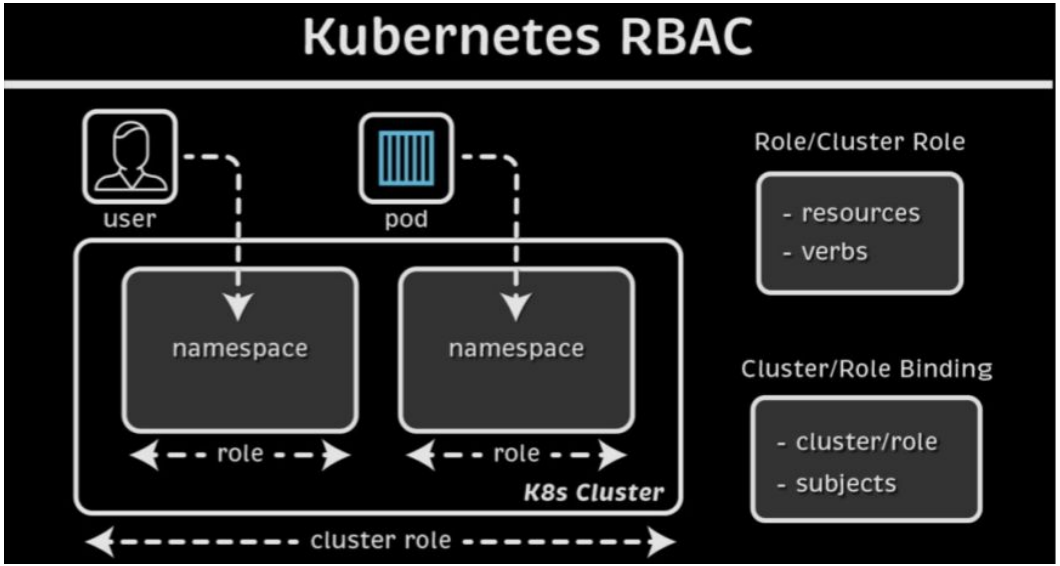
```yaml
apiVersion: v1
kind: Pod
metadata:
  name: resource-demo
spec:
  containers:
  - name: demo-container
    image: nginx
    resources:
      requests:
        memory: "64Mi"
        cpu: "250m"
      limits:
        memory: "128Mi"
        cpu: "500m"
```

Role-Based Access Control (RBAC)

Role-Based Access Control

RBAC (Role-Based Access Control) is an access control mechanism in Kubernetes based on roles and the actions that are allowed to be performed.

RBAC allows you to have fine-grained control over who can do what on which resources within the cluster.



Main Components of RBAC:

- **Role:**
  - Defines a set of permissions (e.g., "get", "list", "create") for specific resources within a Namespace.
  - Only applies within the scope of a single Namespace.
- **ClusterRole:**
  - Similar to Role, but applies across the entire cluster.
  - Used to manage resources that are not Namespace-scoped (e.g., Nodes, PersistentVolumes).
- **RoleBinding:** Binds a Role to a user or group of users within a specific Namespace.
- **ClusterRoleBinding:** Binds a ClusterRole to a user or group of users across the entire cluster.

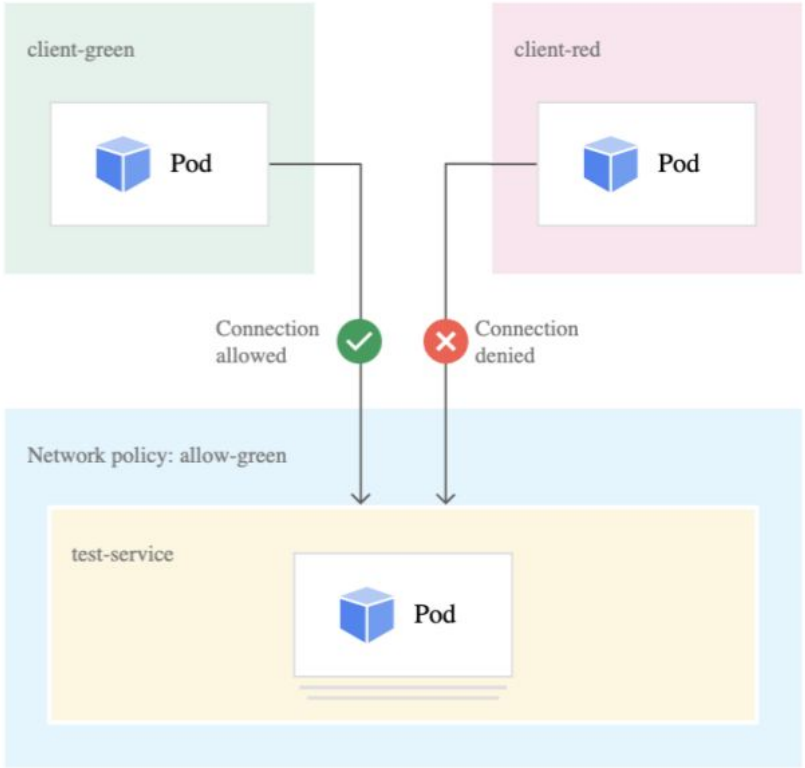
Network Policies

Network Policies

Network Policy is a resource in Kubernetes used to control network traffic to and from Pods within a Namespace.

It allows you to permit or block network traffic based on conditions such as:

- Source and destination.
- Port.
- Protocol.
- Labels of the Pod.



Network Policies

Default Behavior: If no Network Policy is applied, all network traffic between Pods in the cluster is allowed by default.

CNI Plugins that Support Network Policies: Kubernetes requires a CNI (Container Network Interface) plugin that supports Network Policies, such as:

- Calico
- Cilium
- Flannel
- Weave Net

Helm

Helm

Helm is a package management tool in Kubernetes that allows you to define, install, and update Kubernetes applications using charts.

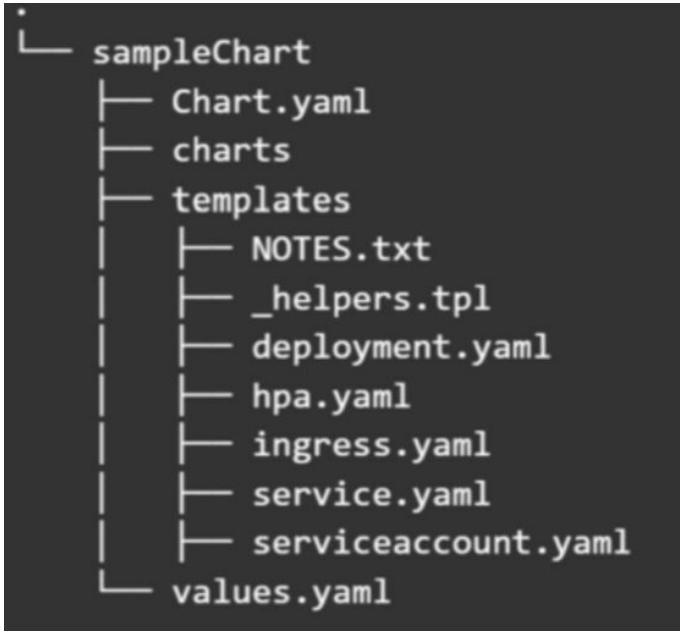
Helm helps package Kubernetes applications into Helm Charts, making it easier to reuse, share, and deploy applications.

Helm

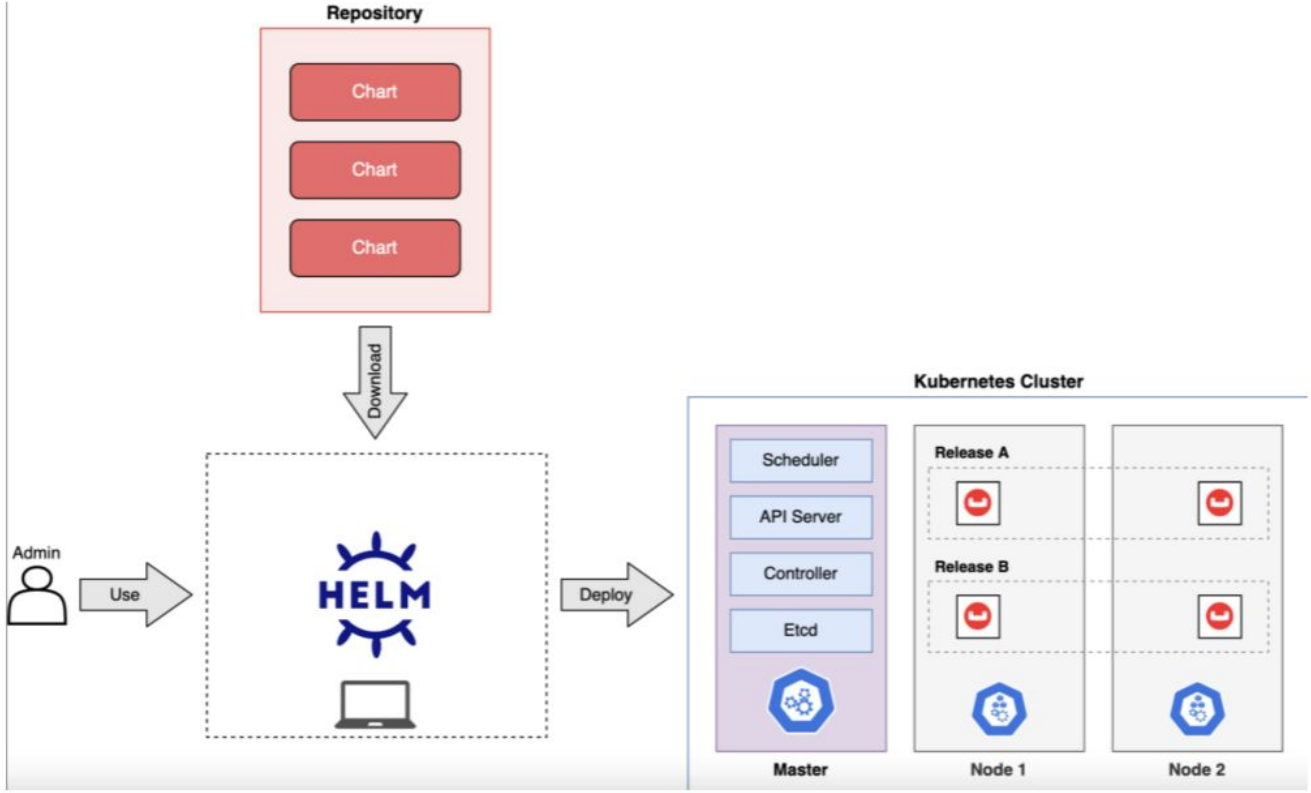
A Helm Chart is a collection of Kubernetes resources packaged in a structured directory format, including YAML files and additional configuration files.

Each Helm Chart includes:

- **Chart.yaml**: Metadata of the chart (name, version, description).
- **values.yaml**: Default configuration values for the chart.
- **templates/**: YAML templates for Kubernetes resources (Pod, Deployment, Service, etc.).
- **charts/**: Sub-charts (if any).
- **README.md**: Description of the chart and usage instructions.



Helm


Repository
Chart
Chart
Chart
Download
Admin
Use
HELM
Deploy
Kubernetes Cluster
Scheduler
API Server
Controller
Etcd
Release A
Release B
Master
Node 1
Node 2

Benefits of Helm:

- Reuse application configurations.
- Easily manage and share Kubernetes applications.
- Automate application deployment, updates, and rollbacks.
- Easily integrate with CI/CD pipelines.

Helm

Installing a Helm Chart

helm install [release-name] [helm-repo-name] -f [value-file] --namespace [tên_namespace]

Example: Install Nginx from Bitnami

helm install my-nginx bitnami/nginx -f values.yaml -namespace dev

Check status:

helm list

Helm

Update Helm Config

helm upgrade [release-name] [helm-repo-name] -f [value-file] --namespace [tên_namespace]

Example: Update the my-nginx Helm release.

helm upgrade my-nginx bitnami/nginx -f values.yaml --namespace dev

Delete Helm Release

helm uninstall [release-name] --namespace [tên_namespace]

Helm

Check the status of a Helm chart:

helm status [release-name]

Example: ***helm status my-nginx***

View the history of a release:

helm history [release-name]

Example: ***helm history my-nginx***

Helm cheetsheat

Link

Thank you