### ****Module 1: Core Concepts****

#### ****Topic 1: Overview of Container Orchestration****

#### ****1. What is Container Orchestration?****

Container orchestration is the automated management of containerized applications, including deployment, scaling, networking, and lifecycle management. It ensures that containers are efficiently allocated and run across multiple hosts in a cluster.

#### ****2. Why Do We Need Container Orchestration?****

Before orchestration, managing containers manually was difficult due to:

* The need to run applications consistently across different environments
* Scaling applications dynamically based on demand
* Automating failure recovery and ensuring high availability
* Managing networking between containers and services

Container orchestration tools like **Kubernetes, Docker Swarm, and OpenShift** automate these tasks, making it easier to run microservices-based architectures at scale.

#### ****3. Key Features of Container Orchestration****

* **Automated Deployment & Scaling**: Containers are deployed and scaled up or down automatically based on demand.
* **Load Balancing & Service Discovery**: Routes traffic to the appropriate container instances.
* **Self-Healing**: Automatically restarts failed containers or replaces unhealthy ones.
* **Storage Orchestration**: Manages persistent storage for stateful applications.
* **Security & Configuration Management**: Ensures secure secrets management and environment-specific configurations.

#### ****4. Popular Container Orchestration Tools****

| **Tool** | **Description** | **Pros** | **Cons** |
| --- | --- | --- | --- |
| **Kubernetes** | Open-source container orchestration platform | Highly scalable, large community, flexible | Steeper learning curve, complex setup |
| **Docker Swarm** | Native container orchestration tool from Docker | Easy to set up, lightweight | Less feature-rich than Kubernetes |
| **OpenShift** | Kubernetes-based orchestration platform from Red Hat | Security-focused, enterprise support | Requires Red Hat ecosystem |
| **Amazon ECS/EKS** | AWS-managed container orchestration | Integrated with AWS services | Vendor lock-in |

# ****Introduction to Kubernetes****

## ****1. What is Kubernetes?****

Kubernetes (often abbreviated as **K8s**) is an open-source **container orchestration** platform that automates 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)**.

Kubernetes enables organizations to efficiently run applications in a distributed environment by managing multiple containers across different machines.

## ****2. Why Kubernetes?****

Before Kubernetes, managing containers at scale was difficult. Kubernetes solves common container management challenges, such as:

### ****Challenges Without Kubernetes****

| **Problem** | **How Kubernetes Helps** |
| --- | --- |
| **Manual Scaling**: You need to manually start/stop containers to handle increased/decreased traffic. | **Auto-scaling**: Kubernetes automatically scales based on CPU/memory usage. |
| **Service Discovery Issues**: Containers might move between nodes, making it difficult to access services. | **Service Discovery**: Kubernetes automatically assigns DNS names and load balances traffic. |
| **Container Failures**: If a container crashes, manual intervention is required to restart it. | **Self-Healing**: Kubernetes restarts failed containers automatically. |
| **Networking Complexity**: Managing communication between containers across multiple machines is difficult. | **Built-in Networking**: Kubernetes provides pod-to-pod communication and external exposure options. |

Kubernetes makes deploying and managing containerized applications **scalable, resilient, and automated**.

## ****3. Kubernetes Core Components****

### ****A. Cluster Architecture****

A Kubernetes **cluster** consists of:

**Master Node (Control Plane)**

Manages the entire cluster and makes scheduling decisions.

Components: API Server, Controller Manager, Scheduler, etcd.

**Worker Nodes**

Run the application workloads in containers (Pods).

Components: Kubelet, Kube-Proxy, Container Runtime (e.g., Docker, containerd).

**Pods**

The smallest deployable unit in Kubernetes that runs one or more containers.

### ****B. Kubernetes Objects (Building Blocks)****

| **Object** | **Description** |
| --- | --- |
| **Pod** | The smallest deployable unit that runs a container. |
| **Service** | Exposes a set of pods to the network (ClusterIP, NodePort, LoadBalancer). |
| **Deployment** | Manages replicas and rolling updates of pods. |
| **ReplicaSet** | Ensures a specified number of pod replicas are running. |
| **ConfigMap & Secrets** | Store configuration data and sensitive information securely. |
| **Ingress** | Manages external access to services using HTTP/HTTPS. |

## ****4. Kubernetes Use Cases****

Kubernetes is widely used in:

* **Microservices Architecture**: Deploying and managing multiple microservices efficiently.
* **CI/CD Pipelines**: Automating application deployment using tools like Jenkins, ArgoCD, or Tekton.
* **Hybrid & Multi-Cloud Deployments**: Running workloads across AWS, Azure, GCP, and on-premise.
* **Big Data & AI/ML**: Managing workloads like Apache Spark and TensorFlow.
* **Edge Computing & IoT**: Running workloads on distributed edge locations.

# ****Understanding Kubernetes Architecture****

## ****1. Kubernetes Architecture Overview****

Kubernetes follows a **master-worker architecture**, where the **Control Plane (Master Node)** manages the **Worker Nodes** that run containerized applications.

A Kubernetes cluster consists of:

* **Control Plane (Master Node)** – Manages the cluster.
* **Worker Nodes** – Run applications inside containers (Pods).
* **Networking Layer** – Connects all components.

## ****2. Kubernetes Cluster Components****

### ****A. Control Plane Components (Master Node)****

| **Component** | **Description** |
| --- | --- |
| **API Server (kube-apiserver)** | The **entry point** for all Kubernetes commands (kubectl, UI, automation). |
| **Controller Manager (kube-controller-manager)** | Runs controllers to maintain cluster state (e.g., replication, endpoint, node lifecycle controllers). |
| **Scheduler (kube-scheduler)** | Assigns Pods to the best Worker Node based on resource availability. |
| **etcd** | Stores cluster configuration data (key-value store). |

**Example:**

When a user creates a **Deployment**, the **API Server** stores it in **etcd**, the **Scheduler** assigns Pods to Nodes, and the **Controller Manager** ensures the right number of replicas.

### ****B. Worker Node Components****

| **Component** | **Description** |
| --- | --- |
| **Kubelet** | Runs on each worker node; ensures containers are running in Pods. |
| **Kube-Proxy** | Manages networking and load balancing across services. |
| **Container Runtime (Docker, containerd, CRI-O)** | Runs containers inside Pods. |

**Example:**

When a Pod is scheduled to a Node, **Kubelet** pulls the container image and runs it using the **Container Runtime**.

### ****C. Kubernetes Networking****

Kubernetes uses a flat network model, where all Pods can communicate.  
Key networking concepts:

* **Pod-to-Pod Communication**: Managed by the network plugin (CNI: Calico, Flannel, Cilium).
* **Service Discovery**: Kubernetes **Services** provide stable networking for Pods.
* **Ingress Controller**: Routes external traffic to the correct Service inside the cluster.

## ****3. Kubernetes Objects and How They Work Together****

| **Object** | **Role in Architecture** |
| --- | --- |
| **Pod** | Smallest unit that runs containers. |
| **Deployment** | Manages Pods and ensures replicas are maintained. |
| **Service** | Exposes Pods internally or externally. |
| **ConfigMap/Secret** | Stores configuration data securely. |
| **Ingress** | Controls external HTTP/HTTPS access. |
|  |  |

### Kubernetes Component: ****Kube API Server****

#### What it Does:

The **Kube API Server** is one of the central components of Kubernetes and serves as the primary interface for interacting with the Kubernetes cluster. It is responsible for exposing the Kubernetes API and handling all requests made to the cluster, whether from users, other services, or even the nodes and pods within the cluster itself.

Key functions of the Kube API Server include:

1. **API Handling**: It exposes a REST API that allows users and other components to interact with the cluster. It validates and processes API requests, executes corresponding actions, and returns the results.
2. **Cluster State Management**: The API server maintains the desired state of the cluster by communicating with the etcd data store. It saves and retrieves cluster state information, such as configurations, resource definitions, and metadata.
3. **Authentication & Authorization**: It enforces security policies by authenticating and authorizing incoming requests.
4. **Communication Hub**: The API Server serves as the central point of communication between all components, such as the kube-scheduler, kube-controller-manager, and worker nodes.

#### How it Works:

* The **Kube API Server** listens on port 6443 by default and can handle requests using HTTP/HTTPS protocols. It processes requests for resources like pods, deployments, services, etc.
* When a user or service wants to interact with the cluster (e.g., create a pod, retrieve the status of a deployment), they send a request to the API server using the Kubernetes API endpoints.
* The API server processes the request and interacts with the etcd cluster (a key-value store) to read or write the state of the resources.
* The API server enforces security measures such as validating requests, ensuring the right permissions, and ensuring that only authorized users can perform specific actions.
* It then communicates with the necessary components, like the scheduler, controllers, and nodes, to act on the requests.
* The API server is typically stateless, relying on **etcd** for persistent state storage.

#### Example of How It Works:

Suppose you want to deploy a new application (a pod) in the Kubernetes cluster. You would send a kubectl command like:

kubectl run nginx --image=nginx

1. **User Request**: The kubectl command makes an HTTP request to the **Kube API Server**, specifically a POST request to /api/v1/pods.
2. **Request Processing**: The API Server validates the request, ensuring the user has the necessary permissions, and checks the provided resource definition (i.e., the nginx image).
3. **Interaction with etcd**: The API server writes the new pod resource definition to the **etcd** key-value store to record the desired state.
4. **Controller Interaction**: The **kube-controller-manager** notices the new pod specification in the etcd database and ensures that the pod is scheduled onto an appropriate node.
5. **Response**: The API server responds to kubectl with the status of the request, confirming that the pod creation was successful.

This whole process involves a series of interactions, and the **Kube API Server** serves as the central point that manages and tracks these actions across the cluster.

### Example API Requests:

* **Get Pod Information**:

curl -k https://<API\_SERVER\_IP>:6443/api/v1/pods/<POD\_NAME>

* **Create Pod (Using JSON/YAML)**:

curl -k -X POST https://<API\_SERVER\_IP>:6443/api/v1/namespaces/default/pods \

-H "Content-Type: application/json" \

-d '{"apiVersion": "v1", "kind": "Pod", "metadata": {"name": "nginx"}, "spec": {"containers": [{"name": "nginx", "image": "nginx"}]}}'

In the above requests, the **Kube API Server** processes the interaction, validates the request, updates **etcd**, and interacts with the Kubernetes scheduler and other components to ensure that the state is updated accordingly.

### Real-World Use Case:

In a large enterprise environment, multiple teams may use a shared Kubernetes cluster. Each team needs to deploy applications or update resources but must do so securely and within predefined permissions. The **Kube API Server** ensures that each team's actions are valid by authenticating them and enforcing access control policies (like Role-Based Access Control, or RBAC). For example, a DevOps team can deploy new applications, but only a system administrator might have the permissions to manage cluster-wide resources like nodes or network configurations.

#### Conclusion:

The **Kube API Server** is critical to the functioning of a Kubernetes cluster, handling all cluster management tasks through its REST API. It acts as a central controller, interacting with other components, managing state via **etcd**, and enforcing security and governance policies.

### Kubernetes Component: ****ETCD****

#### What it Does:

**etcd** is a distributed key-value store that is used by Kubernetes to store and manage the state of the entire cluster. It holds critical data, such as the configuration, resource definitions (e.g., pods, services, deployments), and the current state of the cluster. This makes **etcd** a central component for ensuring the consistency and reliability of the cluster.

Key functions of **etcd** include:

1. **Cluster State Management**: It stores all Kubernetes resource definitions, including pods, deployments, nodes, and services, ensuring the cluster's state is preserved.
2. **Configuration Storage**: **etcd** is used to store configuration data, such as the Kubernetes API server’s configuration, secrets, and custom configurations for components.
3. **Consistent State**: It uses the **Raft consensus algorithm** to ensure that all instances of **etcd** in the cluster maintain a consistent view of the data. This guarantees that the state of the cluster is accurate and reliable.
4. **Leader Election**: **etcd** enables leader election for high availability and fault tolerance, ensuring that there is a consistent leader to manage the write operations to the key-value store.

#### How it Works:

* **etcd** stores data in a key-value format, where the key is a unique identifier for the data and the value is the associated data.
* **etcd** uses the **Raft consensus algorithm** to ensure that all nodes in the etcd cluster agree on the state of the data, even if some nodes fail.
* **etcd** is designed to be highly available and reliable. If one of the **etcd** instances (nodes) fails, another instance will take over, ensuring there is no data loss and no interruption in service.
* **Kubernetes** interacts with **etcd** through the **Kube API Server**. When a resource is created, updated, or deleted, the **API Server** writes to **etcd**, ensuring the cluster's desired state is stored.
* **etcd** is often deployed as a cluster (usually with an odd number of nodes) to maintain fault tolerance and availability. It can be scaled horizontally, and the leader election process ensures that only one instance is responsible for handling writes at any given time.

#### Example of How It Works:

Let's say a user wants to create a new deployment for a web application in Kubernetes. When the user runs the command:

kubectl apply -f deployment.yaml

1. **API Server Request**: The kubectl command sends a request to the **Kube API Server** to create the deployment.
2. **Write to etcd**: The **Kube API Server** validates the request and then writes the deployment definition to **etcd**.
3. **etcd Stores the State**: The deployment's configuration is stored in **etcd** as a key-value pair. The key might represent the type of resource (e.g., deployment), and the value contains the configuration details of the deployment (e.g., replicas, container image, labels).
4. **Cluster State**: All components, such as the **kube-scheduler** and **kube-controller-manager**, read the cluster state from **etcd** to ensure the desired state is achieved. For example, the **scheduler** will decide where to run the new pods based on the information stored in **etcd**.
5. **Leader Election**: If the **etcd** leader crashes during the process, another leader will be elected, ensuring that the system remains consistent and operational.

#### Example API Interaction with etcd:

* **Get Cluster Information from etcd**: To retrieve cluster data directly from **etcd** (e.g., the list of all pods in the cluster), you can use the following command (assuming etcdctl is installed and configured):

etcdctl get /registry/pods --prefix

This would retrieve all the pod information stored in **etcd**.

* **Put Data in etcd**: To store a key-value pair directly in **etcd**, you can use:

etcdctl put /myapp/config '{"replicas": 3, "image": "nginx"}'

This stores a configuration for an application in **etcd** under the key /myapp/config.

#### Real-World Use Case:

Consider a Kubernetes cluster in a cloud-based production environment where different services (like databases, web servers, and microservices) need to be highly available. **etcd** ensures that the state of each service is maintained accurately and consistently across all Kubernetes components. If a pod crashes or if the scheduler needs to move a pod to another node, **etcd** holds the authoritative record of the cluster's desired state, allowing the system to recover quickly and restore the correct configuration.

#### Conclusion:

**etcd** is a critical component of Kubernetes, acting as the centralized store for all cluster data and configurations. It ensures consistency and availability of this data, which is essential for the proper operation of the Kubernetes control plane. By using the Raft consensus algorithm, **etcd** ensures that the data stored within it is consistent, even in the face of failures, providing reliability to Kubernetes clusters.

### Kubernetes Component: ****Kube Controller Manager****

#### What it Does:

The **Kube Controller Manager** is a control plane component that runs controllers responsible for maintaining the desired state of the Kubernetes cluster. It is a set of controllers that regulate various aspects of the cluster, ensuring that the cluster's actual state matches the desired state defined by the user or system.

Controllers are loops that watch the cluster's state and take corrective actions if the actual state deviates from the desired state. For example, if you declare that you want a certain number of replicas for a deployment, the **Kube Controller Manager** ensures that the correct number of replicas are running in the cluster, adding or removing pods as necessary.

Key responsibilities of the **Kube Controller Manager** include:

1. **Replication**: Ensures that the correct number of pod replicas are running as specified in a Deployment or ReplicaSet.
2. **Node Management**: Handles the lifecycle of nodes in the cluster, ensuring that nodes are healthy, registered, and properly managed.
3. **Resource Cleanup**: Handles the cleanup of resources like terminating pods or jobs when their lifecycle ends or when specified conditions are met.
4. **Lease Management**: Manages the leasing mechanism for resources like statefulsets or jobs to control their state across nodes.
5. **Cross-resource Validation**: Ensures that resources like PodDisruptionBudgets and HorizontalPodAutoscalers are applied properly, often in coordination with other controllers.

In essence, the **Kube Controller Manager** works to ensure the cluster is in the desired state and will continuously monitor and correct it.

#### How it Works:

The **Kube Controller Manager** runs a set of controllers that each take care of specific tasks. Each controller listens to events in the cluster and checks whether the current state of resources matches the desired state. If there is a discrepancy, the controller takes corrective action.

* **Event-Driven Architecture**: The controllers are event-driven and watch the Kubernetes API Server for changes to resources. When a resource (like a Pod or Deployment) is modified, the controller reacts by updating the system to bring the cluster back to the desired state.
* **Controller Types**: Some of the main controllers within the **Kube Controller Manager** include:
  + **ReplicaSet Controller**: Ensures that the desired number of replicas are running for a specific Deployment.
  + **Deployment Controller**: Manages the creation, scaling, and rollout of applications defined in Deployments.
  + **Node Controller**: Monitors the health of nodes and manages node registration.
  + **Job Controller**: Ensures that jobs (i.e., workloads that should run to completion) run successfully to completion.
  + **StatefulSet Controller**: Manages the lifecycle of stateful applications and ensures that the desired number of replicas are always running.
  + **EndpointSlice Controller**: Manages the slices of the network endpoints for services.

Each controller has a loop that watches the cluster's state and performs actions to reconcile it with the desired state.

#### Example of How It Works:

Suppose you deploy a Deployment with a replicas: 3 specification:

apiVersion: apps/v1

kind: Deployment

metadata:

name: nginx-deployment

spec:

replicas: 3

selector:

matchLabels:

app: nginx

template:

metadata:

labels:

app: nginx

spec:

containers:

- name: nginx

image: nginx

1. **User Request**: When you apply this configuration using kubectl apply -f nginx-deployment.yaml, the **Kube API Server** stores the Deployment object in **etcd**.
2. **Controller Manager Watches**: The **Deployment Controller** in the **Kube Controller Manager** continuously watches the API server for changes related to Deployment resources.
3. **Replica Management**: The **Deployment Controller** detects that the desired state is to have 3 replicas of the nginx pod. It will then check whether there are already 3 replicas running or not.
4. **Action Taken**: If there are fewer than 3 replicas, the **Deployment Controller** will create new Pods to bring the actual state in line with the desired state.
5. **Reconciliation**: The **Controller Manager** continues to monitor the state of the Pods (such as if a pod crashes or is deleted) and ensures that the number of nginx pods stays at 3.

If one of the nginx pods crashes and stops running, the controller manager will take corrective action and launch a new pod to replace it, thus maintaining the desired state of the Deployment.

#### Example of a Controller in Action:

* **Scaling a Deployment**: If you change the number of replicas for your Deployment:

kubectl scale deployment nginx-deployment --replicas=5

* The **Deployment Controller** in the **Kube Controller Manager** will detect that the desired state is now 5 replicas and will ensure that the necessary pods are created to match that count.

#### Real-World Use Case:

In a Kubernetes-based microservices environment, different applications (e.g., a web app, database, etc.) might be running on separate Deployments. Each application might have varying replica requirements, and some applications need to be highly available at all times.

The **Kube Controller Manager** ensures that these applications always run with the right number of replicas. For instance, if the database application is set to have 3 replicas for redundancy, the **Controller Manager** will ensure that the database always runs with 3 pods. If any of those pods fail, the **Controller Manager** will create a new pod to maintain the desired state.

#### Conclusion:

The **Kube Controller Manager** is a crucial component of the Kubernetes control plane. It runs controllers that ensure the cluster is always in the desired state, by monitoring changes and taking corrective actions. From maintaining pod replicas to ensuring the proper functioning of resources like stateful sets and jobs, the **Kube Controller Manager** is responsible for ensuring the reliability, scalability, and health of the Kubernetes cluster.

### Kubernetes Component: ****Kube Scheduler****

#### What it Does:

The **Kube Scheduler** is a core component of the Kubernetes control plane responsible for deciding which node a newly created pod should run on. It is the component that ensures that workloads (pods) are distributed efficiently across the available nodes in the cluster based on various scheduling rules and constraints.

Key functions of the **Kube Scheduler** include:

1. **Node Selection**: It selects an appropriate node for each pod, considering resource requirements, affinity rules, taints, tolerations, and available resources (CPU, memory, etc.).
2. **Resource Optimization**: It attempts to optimize resource utilization across the cluster by placing pods on nodes that have the required resources and conditions.
3. **Scheduling Constraints**: The scheduler can take into account user-defined scheduling constraints such as:
   1. **Pod Affinity/Anti-Affinity**: Ensuring that certain pods are scheduled together or apart.
   2. **Taints and Tolerations**: Ensuring that pods are scheduled on nodes that can tolerate specific taints.
   3. **Resource Requests/Limitations**: Ensuring that nodes have enough resources (e.g., CPU, memory) to run the pod.
   4. **Node Selectors**: Placing pods on nodes that match specific labels.
4. **Scheduling Policies**: It can take into account policies such as the **PodPriority** and **PodDisruptionBudget** to determine which pods should be scheduled first.

#### How it Works:

The **Kube Scheduler** works by monitoring the API server for newly created pods that do not yet have an assigned node. When a pod is created or needs to be rescheduled (for example, after a node failure), the scheduler follows a series of steps to determine which node the pod should be placed on:

1. **Filter Nodes**: It filters out nodes that cannot accommodate the pod based on:
   1. Resource availability (e.g., CPU, memory).
   2. Affinity and anti-affinity rules (pods that need to be placed together or separated).
   3. Taints and tolerations (ensuring that pods only land on nodes with matching tolerations).
   4. Node selectors (if specified in the pod’s configuration).
2. **Score Nodes**: After filtering, the scheduler assigns scores to the remaining nodes based on various factors such as resource utilization, proximity to other pods, and the overall load on the nodes.
3. **Select Node**: The scheduler then picks the node with the highest score.
4. **Bind Pod to Node**: Finally, the scheduler informs the **Kube API Server** to update the pod’s specification with the chosen node. The pod is then scheduled to run on that node.

#### Example of How It Works:

Let’s say you deploy a pod and you specify that it requires 2 CPUs and 4 GB of memory. If there are three nodes in your cluster with the following available resources:

* **Node A**: 4 CPUs, 8 GB of memory
* **Node B**: 2 CPUs, 6 GB of memory
* **Node C**: 8 CPUs, 16 GB of memory

1. **Pod Request**: You create the following pod definition:

apiVersion: v1

kind: Pod

metadata:

name: nginx-pod

spec:

containers:

- name: nginx

image: nginx

resources:

requests:

cpu: "2"

memory: "4Gi"

**Scheduler Evaluation**: The **Kube Scheduler** will check the available nodes for resources that match the pod’s requirements:

* + **Node A** has enough resources (4 CPUs, 8 GB memory).
  + **Node B** has enough resources (2 CPUs, 6 GB memory).
  + **Node C** has more than enough resources but may not be ideal for reasons like being underutilized or having other scheduling constraints (such as taints or node affinity).

**Node Scoring**: The scheduler scores each node based on how well they fit the pod's resource requirements and any constraints defined in the pod's configuration (e.g., affinity rules). If all nodes are eligible, the scheduler might choose the node with the lowest load.

**Binding**: After selecting the best node (e.g., **Node B**), the scheduler will update the pod's specification to bind it to that node.

**Pod Scheduled**: The pod is now scheduled to run on **Node B** with 2 CPUs and 4 GB of memory.

#### Example of Scheduling with Taints and Tolerations:

Suppose you want to ensure that certain pods are scheduled only on nodes with specific characteristics. You can use **taints** and **tolerations** for this.

1. **Taint a Node**: You taint a node with a custom taint:

kubectl taint nodes node-1 key=value:NoSchedule

This means that no pod will be scheduled on **node-1** unless it has a matching **toleration**.

1. **Pod Toleration**: You then create a pod that tolerates this taint:

apiVersion: v1

kind: Pod

metadata:

name: nginx-pod

spec:

containers:

- name: nginx

image: nginx

tolerations:

- key: "key"

operator: "Equal"

value: "value"

effect: "NoSchedule"

1. **Scheduler Action**: The **Kube Scheduler** will then place the pod on **node-1** because it has the matching toleration for the taint applied to that node.

#### Real-World Use Case:

In a production environment, organizations might need to schedule pods in a way that minimizes latency or optimizes resource usage. For example:

* **Affinity/Anti-Affinity**: Pods for a web application might need to be scheduled on nodes close to a database pod (using **pod affinity**), while preventing certain workloads from running on the same node (using **anti-affinity**).
* **Resource-based Scheduling**: A cluster might have specialized nodes (e.g., nodes with GPUs for machine learning workloads), and you can use **node selectors** to ensure that such workloads only run on the nodes with GPUs.
* **Custom Taints**: Nodes designated for batch processing might be tainted to ensure that no other pods (e.g., real-time applications) are scheduled on those nodes.

#### Conclusion:

The **Kube Scheduler** is responsible for making intelligent decisions about where to place pods within the cluster based on available resources, scheduling constraints, and policies. It ensures that the desired state of the system, such as resource distribution and affinity requirements, is met. By evaluating node resources and policies like affinity, tolerations, and resource requests, the scheduler ensures that workloads are optimally placed, helping maintain the efficiency and health of the cluster.

### Kubernetes Component: ****Kubelet****

#### What it Does:

The **Kubelet** is an essential node-level component in Kubernetes. It is an agent that runs on each node in the cluster and ensures that the containers specified in PodSpecs are running and healthy. The **Kubelet** communicates with the **Kube API Server** to manage the state of the node, as well as to report back on the state of the containers running on it.

Key functions of the **Kubelet** include:

1. **Pod Lifecycle Management**: It ensures that containers are started, stopped, and run according to the specifications in the Pod definition.
2. **Health Checking**: The **Kubelet** monitors the health of containers (using **liveness** and **readiness probes**) and takes corrective actions (e.g., restarting the container if it fails).
3. **Resource Reporting**: It reports node-level resource usage (e.g., CPU, memory) to the **Kube API Server** so that the scheduler can make informed decisions about pod placement.
4. **Node Registration**: The **Kubelet** registers the node with the Kubernetes cluster, making it available for scheduling pods.
5. **Container Runtime Interaction**: The **Kubelet** interacts with the container runtime (e.g., Docker, containerd) to manage containers on the node. It requests the container runtime to pull images, create containers, and manage container lifecycle events.

#### How it Works:

The **Kubelet** runs on every node in the Kubernetes cluster and watches the **Kube API Server** for the Pods assigned to its node. The main steps it follows to manage pod lifecycle are:

**Pod Spec Synchronization**:

* 1. The **Kubelet** regularly checks the **Kube API Server** for the list of Pods assigned to its node.
  2. If a new pod is scheduled to run on the node, the **Kubelet** downloads the PodSpec (the specification of the pod) from the API Server.
  3. The **Kubelet** then makes sure the containers defined in the PodSpec are running.

**Container Management**:

* 1. The **Kubelet** interacts with the container runtime (such as Docker, containerd, or CRI-O) to create, start, stop, and manage containers as described in the PodSpec.
  2. It monitors the status of the containers, checking whether they are running, healthy, or terminated.

**Health Monitoring**:

* 1. The **Kubelet** checks the health of each container using **liveness** and **readiness probes**:
     1. **Liveness probes**: Ensure that a container is still alive and functioning. If a container fails the liveness check, the **Kubelet** will kill and restart the container.
     2. **Readiness probes**: Ensure that the container is ready to serve traffic. If a container fails the readiness probe, the **Kubelet** will mark the pod as "not ready," preventing it from receiving traffic.

**Reporting Back to API Server**:

* 1. The **Kubelet** constantly reports the status of the node and containers to the **Kube API Server**, which then updates the state of the cluster.
  2. If a pod is not running or if the node is under high load, the **Kubelet** reports this to the API Server, which may trigger rescheduling or other actions.

**Resource Management**:

* 1. The **Kubelet** monitors the resource usage (CPU, memory) of containers and ensures they do not exceed the specified limits. It may evict pods if the node runs out of resources, ensuring the overall health of the node.

**Node Health Checks**:

* 1. The **Kubelet** also checks the overall health of the node itself, reporting whether the node is ready to schedule new pods or if it should be marked as unhealthy.

#### Example of How It Works:

Let’s say a user creates a pod with two containers in the following manifest:

apiVersion: v1

kind: Pod

metadata:

name: nginx-pod

spec:

containers:

- name: nginx-container

image: nginx:latest

ports:

- containerPort: 80

- name: redis-container

image: redis:latest

ports:

- containerPort: 6379

**Pod Assignment**: The **Kube Scheduler** assigns this pod to a node in the cluster. The **Kubelet** running on that node detects the new pod and fetches the PodSpec from the **Kube API Server**.

**Container Launching**: The **Kubelet** then instructs the container runtime (e.g., Docker) to pull the nginx:latest and redis:latest images from the container registry and run them as containers in the pod.

**Health Monitoring**: The **Kubelet** will monitor the containers using configured probes (for example, liveness and readiness probes) to ensure that both containers are healthy. If any container fails the probe, the **Kubelet** will restart the container.

**Reporting**: The **Kubelet** reports the status of the pod and containers (whether they are running, failed, or completed) back to the **Kube API Server**, which keeps track of the overall state of the cluster.

**Node Resource Management**: The **Kubelet** will monitor resource usage (e.g., CPU and memory) and make sure that the containers do not exceed the allocated resources. If the node runs low on resources, the **Kubelet** may evict pods to make room for others.

#### Example of Health Check with Liveness and Readiness Probes:

Here’s an example where the **Kubelet** uses **liveness** and **readiness** probes to manage container health:

apiVersion: v1

kind: Pod

metadata:

name: nginx-pod

spec:

containers:

- name: nginx

image: nginx:latest

ports:

- containerPort: 80

livenessProbe:

httpGet:

path: /healthz

port: 80

initialDelaySeconds: 5

periodSeconds: 10

readinessProbe:

httpGet:

path: /readiness

port: 80

initialDelaySeconds: 5

periodSeconds: 10

* **Liveness Probe**: The **Kubelet** will call /healthz on the nginx container every 10 seconds (after an initial delay of 5 seconds). If the container does not respond with a healthy status, the **Kubelet** will restart the container.
* **Readiness Probe**: The **Kubelet** will check /readiness to determine when the container is ready to serve traffic. Until the readiness probe passes, the pod will not be marked as "ready" to receive traffic.

#### Real-World Use Case:

Consider a scenario in a Kubernetes cluster where a web application has multiple replicas of the nginx container running. Each **Kubelet** on the worker nodes is responsible for managing the lifecycle of the pods on its node, ensuring that:

* The containers are running.
* The containers are healthy.
* The resources are not being over-consumed.

If a pod’s container fails due to a crash or an issue, the **Kubelet** will detect the failure through the liveness probe and restart the container, ensuring minimal disruption to the service. If a node is under resource pressure (e.g., CPU or memory overload), the **Kubelet** may evict lower-priority pods to free up resources, keeping the node functional.

#### Conclusion:

The **Kubelet** is a vital node-level component in Kubernetes that ensures containers are running as expected, reporting their status, and handling container lifecycle events. It is the primary agent responsible for maintaining the state of the containers on a specific node and works in close coordination with the **Kube API Server**, container runtimes, and other components in the cluster to ensure that applications are running correctly and efficiently.

# ****Module 2: Installation, Configuration & Validation****

## ****Topic 1: Designing a Kubernetes Cluster****

### ****1. Introduction to Kubernetes Cluster Design****

A well-designed Kubernetes cluster ensures **high availability, scalability, and security** for workloads. Before installation, we must consider:  
 **Cluster size and topology** (single-node vs. multi-node, high availability)  
 **Infrastructure choices** (on-premises, cloud, bare metal, VMs)  
 **Networking model** (CNI, Pod-to-Pod communication, external access)  
 **Storage requirements** (Persistent Volumes, CSI plugins)  
 **Security and access control** (RBAC, TLS, firewall rules)

### ****2. Kubernetes Cluster Architecture****

A Kubernetes cluster typically consists of:

| **Component** | **Role** |
| --- | --- |
| **Master Node (Control Plane)** | Manages scheduling, API requests, and cluster state. |
| **Worker Nodes** | Run the actual application workloads inside Pods. |
| **Networking Layer** | Ensures communication between Pods, Nodes, and external users. |

For this module, we will design and install a **1-Master, 2-Worker Node Kubernetes cluster using kubeadm**.

### ****3. Cluster Design Considerations****

Before installing Kubernetes, we must plan:

| **Factor** | **Decision** |
| --- | --- |
| **Control Plane** | Single master node (not highly available) |
| **Worker Nodes** | Two worker nodes for running applications |
| **Networking** | Calico (CNI-based network plugin) |
| **Storage** | Local storage for testing, cloud storage for production |
| **Ingress & Load Balancing** | Nginx Ingress Controller |
| **Security** | RBAC enabled, TLS for secure communication |

**Why this setup?**

A **single master node** is easier to manage for learning/testing.

**Two worker nodes** allow scheduling of workloads with failover.

**Calico** provides secure networking and network policies.

**kubeadm** automates cluster setup, making it production-ready.

## ****LAB: Designing Kubernetes Cluster (Planning Phase)****

### ****Step 1: Define Infrastructure Requirements****

| **Component** | **Details** |
| --- | --- |
| **Master Node** | 2 vCPUs, 4GB RAM, 20GB Disk |
| **Worker Nodes** | 2 vCPUs, 4GB RAM, 20GB Disk each |
| **OS** | Ubuntu 22.04 LTS |
| **Network Plugin** | Calico |
| **Container Runtime** | containerd |
| **Kubernetes Version** | v1.28+ |
| **Required Ports** | 6443 (API Server), 10250 (Kubelet), 2379-2380 (etcd) |

### ****Step 2: Network Design****

Kubernetes networking is critical for **Pod-to-Pod** and **Service** communication.  
 **Pod CIDR:** 192.168.0.0/16 (each Pod gets an IP)  
 **Service CIDR:** 10.96.0.0/12 (virtual IPs for Services)  
 **Node CIDR:** Nodes get their own IP range for Pod allocation

### ****Step 3: Preparing for Installation****

To install a **1-Master, 2-Worker cluster using kubeadm**, we need:

3 Ubuntu 22.04 machines (VMs, bare metal, or cloud instances).

**Static IPs** assigned to each machine.

**Firewall Rules** adjusted for Kubernetes ports.

**SSH Access** for remote management.

### ****Step 4: Verify System Requirements****

Run the following on **all nodes**:

# Check CPU and RAM

lscpu && free -m

# Check OS version

cat /etc/os-release

# Check network connectivity

ping -c 3 google.com

### ****Summary****

* **Planned Kubernetes cluster design** (1 master, 2 workers).
* **Chose network model** (Calico for Pod networking).
* **Verified system requirements** (Ubuntu 22.04, networking, firewall rules).

# ****Installation of Kubernetes 1-Master and 2-Node Cluster using kubeadm****

## ****1. Introduction****

We will install a **1-Master, 2-Worker Kubernetes cluster** using **kubeadm**.

**Master Node:** Controls and manages the cluster.

**Worker Nodes:** Run application workloads inside Pods.

We will set up:  
 **kubeadm** – Tool to bootstrap the cluster.  
 **kubelet** – Runs on all nodes, responsible for managing Pods.  
 **kubectl** – Command-line tool to interact with the cluster.  
 **Container runtime** – containerd for running containers.

## ****2. Prerequisites****

| **Component** | **Requirement** |
| --- | --- |
| **OS** | Ubuntu 22.04 LTS |
| **CPU/RAM** | 2 vCPUs, 4GB RAM per node |
| **Disk** | 20GB free space |
| **Network Plugin** | Calico |
| **Container Runtime** | containerd |
| **Kubernetes Version** | v1.28+ |
| **User Privileges** | Root or sudo access |

### ****📌 Required Ports****

| **Port** | **Component** | **Direction** |
| --- | --- | --- |
| 6443 | API Server | Incoming |
| 2379-2380 | etcd | Internal |
| 10250-10255 | Kubelet | Internal |
| 30000-32767 | NodePort Services | External |

## ****3. LAB: Installing Kubernetes Cluster with kubeadm****

### ****Step 1: Prepare All Nodes****

Run these steps on **Master and Worker Nodes**.

#### ****1.1 Update the system****

sudo apt update && sudo apt upgrade -y

#### ****1.2 Disable Swap (Required by Kubernetes)****

sudo swapoff -a

sed -i '/swap/d' /etc/fstab

#### ****1.3 Load Required Kernel Modules****

cat <<EOF | sudo tee /etc/modules-load.d/k8s.conf

overlay

br\_netfilter

EOF

sudo modprobe overlay

sudo modprobe br\_netfilter

#### ****1.4 Configure Network Settings****

cat <<EOF | sudo tee /etc/sysctl.d/k8s.conf

net.bridge.bridge-nf-call-iptables = 1

net.bridge.bridge-nf-call-ip6tables = 1

net.ipv4.ip\_forward = 1

EOF

sudo sysctl --system

### ****Step 2: Install Container Runtime (****containerd****)****

sudo apt install -y containerd

Configure containerd:

sudo mkdir -p /etc/containerd

containerd config default | sudo tee /etc/containerd/config.toml > /dev/null

Enable SystemdCgroup:

sudo sed -i 's/SystemdCgroup = false/SystemdCgroup = true/' /etc/containerd/config.toml

Restart service:

sudo systemctl restart containerd

sudo systemctl enable containerd

### ****Step 3: Install kubeadm, kubelet, and kubectl****

sudo apt update

sudo apt install -y apt-transport-https ca-certificates curl

curl -fsSL https://pkgs.k8s.io/core:/stable:/v1.28/deb/Release.key | sudo gpg --dearmor -o /etc/apt/trusted.gpg.d/kubernetes-apt-keyring.gpg

echo "deb https://pkgs.k8s.io/core:/stable:/v1.28/deb/ /" | sudo tee /etc/apt/sources.list.d/kubernetes.list

sudo apt update

sudo apt install -y kubelet kubeadm kubectl

sudo apt-mark hold kubelet kubeadm kubectl

Verify installation:

kubeadm version

### ****Step 4: Initialize the Master Node****

Run **ONLY on the Master Node**.

sudo kubeadm init --pod-network-cidr=192.168.0.0/16

After successful initialization, you will see a **kubeadm join** command. Copy it.

#### ****4.1 Set Up**** kubectl ****for the Master Node****

mkdir -p $HOME/.kube

sudo cp -i /etc/kubernetes/admin.conf $HOME/.kube/config

sudo chown $(id -u):$(id -g) $HOME/.kube/config

Verify the cluster:

kubectl get nodes

### ****Step 5: Configure Networking (Calico)****

Install Calico on the **Master Node**:

kubectl apply -f https://docs.projectcalico.org/manifests/calico.yaml

Verify that all Pods are running:

kubectl get pods -n kube-system

### ****Step 6: Join Worker Nodes to the Cluster****

Run the **kubeadm join** command from **Step 4** on each Worker Node. Example:

sudo kubeadm join <MASTER\_IP>:6443 --token <TOKEN> --discovery-token-ca-cert-hash sha256:<HASH>

Verify the cluster:

kubectl get nodes

You should see **Master** and **Worker Nodes** in the Ready state.

## ****📌 Summary****

✅ Installed Kubernetes using **kubeadm**.  
✅ Set up **containerd** as the runtime.  
✅ Configured **Master Node** and initialized the cluster.  
✅ Installed **Calico** for networking.  
✅ **Joined Worker Nodes** to the cluster.

# ****Choosing a Network Solution and Configuring It****

## ****1. Introduction to Kubernetes Networking****

Kubernetes networking ensures seamless communication between **Pods, Services, and external clients**. A **CNI (Container Network Interface) plugin** is required to handle networking.

### ****📌 Key Networking Concepts****

✅ **Pod-to-Pod Communication** – Pods should communicate across nodes without NAT.  
✅ **Service Discovery** – Services allow Pods to find and communicate with each other.  
✅ **Network Policies** – Control traffic between Pods for security.

## ****2. Choosing a Kubernetes Network Solution****

### ****Popular CNI Options****

| **CNI Plugin** | **Features** | **Best For** |
| --- | --- | --- |
| **Calico** | Network policies, security, scalability | Security-focused workloads |
| **Flannel** | Simple, lightweight, easy to set up | Basic Kubernetes networking |
| **Cilium** | eBPF-based, high performance | High-security and observability |
| **Weave** | Encrypted networking, fast setup | Small-to-medium clusters |

💡 **For our setup, we will use** Calico, as it supports **network policies** and is widely used in production.

## ****3. LAB: Installing and Configuring Calico****

### ****Step 1: Install Calico on the Master Node****

Run this on the **Master Node**:

kubectl apply -f https://docs.projectcalico.org/manifests/calico.yaml

This installs:

**calico-node** (responsible for networking on each node).

**calico-kube-controllers** (manages policies and synchronization).

Verify that all Calico Pods are running:

kubectl get pods -n kube-system

Expected output:

NAME READY STATUS RESTARTS AGE

calico-kube-controllers-xxxxx 1/1 Running 0 2m

calico-node-xxxxx 1/1 Running 0 2m

### ****Step 2: Verify Network Connectivity****

Ensure all nodes are in the Ready state:

kubectl get nodes

Test Pod-to-Pod communication:  
1️⃣ Deploy a test Pod:

kubectl run test-pod --image=busybox -- sleep 3600

2️⃣ Get the Pod's IP:

kubectl get pod test-pod -o wide

3️⃣ Deploy another Pod and check connectivity:

kubectl run test-pod2 --image=busybox -- sleep 3600

kubectl exec -it test-pod -- ping <TEST-POD2-IP>

If successful, networking is working correctly.

### ****Step 3: Implement a Network Policy (Optional)****

By default, all Pods can communicate. Let's create a policy to **restrict access**.

#### ****Deny All Traffic****

cat <<EOF | kubectl apply -f -

apiVersion: networking.k8s.io/v1

kind: NetworkPolicy

metadata:

name: deny-all

namespace: default

spec:

podSelector: {}

policyTypes:

- Ingress

- Egress

EOF

This blocks **all traffic**. To allow communication, define specific rules.

### ****📌 Summary****

✅ Installed **Calico** for networking.  
✅ Verified **Pod-to-Pod** communication.  
✅ Implemented **Network Policy** for security.

# ****Verifying Kubernetes Installation with kubectl Commands****

## ****1. Introduction****

After installing Kubernetes and setting up networking, we need to **verify** that everything is working as expected. We will use kubectl, the Kubernetes command-line tool, to check:  
✅ Cluster status  
✅ Node and Pod health  
✅ Networking and DNS resolution

## ****2. LAB: Verifying Kubernetes Installation****

### ****Step 1: Check Cluster Status****

Run on the **Master Node**:

kubectl cluster-info

Expected output:

Kubernetes control plane is running at https://<master-ip>:6443

CoreDNS is running at https://<master-ip>:6443/api/v1/namespaces/kube-system/services/kube-dns:dns/proxy

This confirms that the **API Server and CoreDNS** are running.

### ****Step 2: Verify Nodes****

kubectl get nodes -o wide

Expected output:

NAME STATUS ROLES AGE VERSION INTERNAL-IP

master Ready control-plane 20m v1.28.0 192.168.1.10

worker-1 Ready <none> 15m v1.28.0 192.168.1.11

worker-2 Ready <none> 15m v1.28.0 192.168.1.12

✅ All nodes should be in the Ready state.

### ****Step 3: Check System Pods****

kubectl get pods -n kube-system

Expected output:

NAME READY STATUS RESTARTS AGE

coredns-xxxxx 1/1 Running 0 10m

etcd-master 1/1 Running 0 10m

kube-apiserver-master 1/1 Running 0 10m

kube-controller-manager-master 1/1 Running 0 10m

kube-proxy-xxxxx 1/1 Running 0 10m

kube-scheduler-master 1/1 Running 0 10m

calico-node-xxxxx 1/1 Running 0 10m

✅ All critical services should be **Running**.

### ****Step 4: Verify Pod-to-Pod Communication****

Create a test Pod:

kubectl run test-pod --image=busybox -- sleep 3600

Check its IP:

kubectl get pod test-pod -o wide

Deploy another test Pod:

kubectl run test-pod2 --image=busybox -- sleep 3600

Test connectivity:

kubectl exec -it test-pod -- ping <TEST-POD2-IP>

✅ If ping works, networking is configured correctly.

### ****Step 5: Verify Kubernetes DNS****

Kubernetes uses **CoreDNS** for service discovery. Test it by running:

kubectl exec -it test-pod -- nslookup kubernetes.default

Expected output:

Server: 10.96.0.10

Address: 10.96.0.10#53

Name: kubernetes.default.svc.cluster.local

Address: 10.96.0.1

✅ If resolution works, **CoreDNS is functioning properly**.

### ****Step 6: Verify Worker Nodes Can Run Workloads****

Deploy a test application:

kubectl create deployment nginx --image=nginx

Expose the deployment:

kubectl expose deployment nginx --type=NodePort --port=80

Find the assigned port:

kubectl get svc nginx

Test access from a worker node:

curl http://<worker-node-ip>:<NodePort>

✅ If you get an **Nginx welcome page**, worker nodes are scheduling workloads correctly.

## ****📌 Summary****

✅ Verified **cluster status** (kubectl cluster-info)  
✅ Ensured **all nodes are Ready** (kubectl get nodes)  
✅ Checked **system Pods** (kubectl get pods -n kube-system)  
✅ Confirmed **Pod-to-Pod networking** (ping test-pod2)  
✅ Validated **Kubernetes DNS resolution** (nslookup kubernetes.default)  
✅ Ensured **Worker Nodes can run workloads** (kubectl create deployment nginx)

# ****Understanding Pods, Labels & Selectors in Kubernetes****

## ****1. Introduction to Kubernetes Pods****

A **Pod** is the smallest deployable unit in Kubernetes. A Pod can have:  
✅ **A single container** (most common use case).  
✅ **Multiple containers** that share the same storage and network (e.g., a web server + a sidecar logging agent).

### ****📌 Real-World Use Cases****

**Single-container Pods** → Nginx web server deployment.

**Multi-container Pods** → Log processing with a sidecar container.

## ****2. Lab: Deploying Applications as a Pod****

### ****Step 1: Create a Basic Pod Definition****

Create a file nginx-pod.yaml:

apiVersion: v1

kind: Pod

metadata:

name: nginx-pod

labels:

app: nginx

spec:

containers:

- name: nginx

image: nginx

ports:

- containerPort: 80

### ****Step 2: Deploy the Pod****

kubectl apply -f nginx-pod.yaml

Verify deployment:

kubectl get pods -o wide

### ****Step 3: Access the Pod****

kubectl exec -it nginx-pod -- /bin/sh

Inside the Pod:

curl localhost

✅ If you see an HTML response, the **Pod is running correctly**.

## ****3. Understanding Labels & Selectors****

### ****📌 What are Labels?****

Labels are **key-value pairs** attached to Kubernetes objects.  
✅ Used to identify and organize resources.  
✅ Commonly used for **grouping and selection**.

Example:

metadata:

labels:

app: nginx

environment: production

### ****📌 What are Selectors?****

Selectors are used to **filter resources based on labels**.  
✅ matchLabels → Exact key-value match.  
✅ matchExpressions → Advanced filtering (e.g., "app **in** [nginx, apache]").

Example selector:

selector:

matchLabels:

app: nginx

## ****4. Lab: Managing Labels & Selectors****

### ****Step 1: Add Labels to a Pod****

kubectl label pod nginx-pod environment=production

Check labels:

kubectl get pods --show-labels

### ****Step 2: Select Pods Using Labels****

kubectl get pods -l environment=production

✅ This returns only the nginx-pod because it has that label.

## ****📌 Summary****

✅ **Pods** are the smallest deployable unit in Kubernetes.  
✅ **Labels** are key-value pairs used for organization.  
✅ **Selectors** allow filtering based on labels.  
✅ **Lab** covered deploying a Pod, adding labels, and filtering with selectors.

# ****Understanding Replication Controller & Replica Set in Kubernetes****

## ****1. Introduction to Replication Controller & Replica Set****

A **Replication Controller (RC)** ensures that a specified number of Pod replicas are running at any given time. It was the first way Kubernetes managed Pod replication but has now been **superseded by Replica Sets (RS)**.

A **Replica Set (RS)** is the newer version of a Replication Controller and supports **label selectors with set-based expressions**, making it more flexible.

✅ Ensures **high availability** of applications.  
✅ Automatically replaces failed Pods.  
✅ Supports **scaling up/down** as needed.

## ****2. Differences Between Replication Controller & Replica Set****

| **Feature** | **Replication Controller (RC)** | **Replica Set (RS)** |
| --- | --- | --- |
| **Selector Type** | Only supports matchLabels | Supports matchLabels and matchExpressions |
| **Use Case** | Older way to maintain replicas | Newer and more flexible replacement |
| **Scaling** | Supports manual scaling | Works with **Deployments** for automatic scaling |

✅ **Replica Sets are recommended over Replication Controllers** for modern Kubernetes applications.

## ****3. Real-World Use Cases****

📌 **E-commerce Website:**

Ensure **3 replicas** of the web server Pod are always running.

If one Pod fails, automatically create a new one.

📌 **Microservices Architecture:**

Multiple services (e.g., authentication, payments) run with redundancy.

If traffic increases, manually or automatically scale the number of Pods.

## ****4. Lab: Deploying a Replica Set****

### ****Step 1: Create a Replica Set****

Create a file nginx-replicaset.yaml:

apiVersion: apps/v1

kind: ReplicaSet

metadata:

name: nginx-rs

labels:

app: nginx

spec:

replicas: 3

selector:

matchLabels:

app: nginx

template:

metadata:

labels:

app: nginx

spec:

containers:

- name: nginx

image: nginx

ports:

- containerPort: 80

### ****Step 2: Deploy the Replica Set****

kubectl apply -f nginx-replicaset.yaml

Verify the running Pods:

kubectl get pods

Expected output (3 Pods should be running):

NAME READY STATUS RESTARTS AGE

nginx-rs-xxxxx 1/1 Running 0 30s

nginx-rs-yyyyy 1/1 Running 0 30s

nginx-rs-zzzzz 1/1 Running 0 30s

### ****Step 3: Test Auto-Recovery****

Delete one of the Pods:

kubectl delete pod <nginx-pod-name>

Check again:

kubectl get pods

✅ **A new Pod should be created automatically!**

## ****5. Scaling a Replica Set****

### ****Manual Scaling****

Increase replicas to 5:

kubectl scale --replicas=5 rs/nginx-rs

Check the new number of Pods:

kubectl get pods

✅ Now there should be **5 Pods running**.

### ****Updating a Replica Set (Changing Image)****

To update the image, modify nginx-replicaset.yaml:

containers:

- name: nginx

image: nginx:1.21

Apply the changes:

kubectl apply -f nginx-replicaset.yaml

Verify the updated image:

kubectl describe pod <nginx-pod-name>

✅ This shows the new **nginx:1.21** image.

## ****📌 Summary****

✅ **Replication Controllers** ensure a fixed number of Pods are running.  
✅ **Replica Sets** improve on RCs with flexible label selectors.  
✅ **Auto-recovery** ensures high availability.  
✅ **Scaling & Updates** allow flexible resource management.

# ****Understanding Kubernetes Services – ClusterIP, NodePort & LoadBalancer****

## ****1. Introduction to Kubernetes Services****

A **Service** in Kubernetes provides network access to a set of Pods. Since Pods are **ephemeral** (they can be created and destroyed dynamically), a Service ensures that applications can communicate reliably.

✅ **Abstracts** the Pod IP addresses.  
✅ Provides **load balancing** between multiple Pod replicas.  
✅ Supports **internal and external traffic routing**.

## ****2. Types of Kubernetes Services****

| **Service Type** | **Description** | **Use Case** |
| --- | --- | --- |
| **ClusterIP** (default) | Exposes the service **internally** within the cluster. | Internal microservices (e.g., database, backend APIs). |
| **NodePort** | Exposes the service on a **static port** on each node. | Accessing a service from outside the cluster. |
| **LoadBalancer** | Creates an **external load balancer** (cloud provider required). | Exposing applications to the internet. |
| **ExternalName** | Maps the service to an **external domain name**. | Accessing an external service using a DNS alias. |

## ****3. Real-World Use Cases****

📌 **ClusterIP (Default)** – Used for **internal communication**  
🔹 A backend service (e.g., database) that **should not be exposed externally**.  
🔹 Example: A PostgreSQL database used by internal microservices.

📌 **NodePort** – Exposes a service on each **worker node's IP**  
🔹 Good for **small setups** where you want to expose an app but don’t have a load balancer.  
🔹 Example: Running a web app accessible from outside the cluster.

📌 **LoadBalancer** – Best for **cloud deployments**  
🔹 Automatically provisions an **external** cloud load balancer.  
🔹 Example: A production-ready website accessible over the internet.

## ****4. Lab: Creating and Managing Kubernetes Services****

### ****Step 1: Deploy a Simple Web Application****

Create a file nginx-deployment.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

ports:

- containerPort: 80

Apply the deployment:

kubectl apply -f nginx-deployment.yaml

## ****5. ClusterIP Service****

### ****Step 1: Create a ClusterIP Service****

Create nginx-clusterip.yaml:

apiVersion: v1

kind: Service

metadata:

name: nginx-clusterip

spec:

selector:

app: nginx

ports:

- protocol: TCP

port: 80

targetPort: 80

type: ClusterIP

Apply the Service:

kubectl apply -f nginx-clusterip.yaml

Check the Service:

kubectl get svc nginx-clusterip

✅ The service should have a **ClusterIP**, which is **accessible only inside the cluster**.

### ****Step 2: Test Internal Access****

Get inside a Pod:

kubectl run test-pod --image=busybox -- sleep 3600

kubectl exec -it test-pod -- /bin/sh

Inside the Pod:

wget -qO- nginx-clusterip

✅ You should get the **Nginx welcome page**.

## ****6. NodePort Service****

### ****Step 1: Create a NodePort Service****

Create nginx-nodeport.yaml:

apiVersion: v1

kind: Service

metadata:

name: nginx-nodeport

spec:

selector:

app: nginx

ports:

- protocol: TCP

port: 80

targetPort: 80

nodePort: 30007

type: NodePort

Apply the Service:

kubectl apply -f nginx-nodeport.yaml

Check the service:

kubectl get svc nginx-nodeport

✅ The service should have a **NodePort (e.g., 30007)** assigned.

### ****Step 2: Access the Service from Outside the Cluster****

Find the worker node IP:

kubectl get nodes -o wide

Test access from your **local machine**:

curl http://<worker-node-ip>:30007

✅ You should see the **Nginx welcome page**.

## ****7. LoadBalancer Service****

### ****Step 1: Create a LoadBalancer Service****

Create nginx-loadbalancer.yaml:

apiVersion: v1

kind: Service

metadata:

name: nginx-loadbalancer

spec:

selector:

app: nginx

ports:

- protocol: TCP

port: 80

targetPort: 80

type: LoadBalancer

Apply the Service:

kubectl apply -f nginx-loadbalancer.yaml

Check the external IP:

kubectl get svc nginx-loadbalancer

✅ If running in **cloud** (GCP, AWS, Azure), this will assign a **public IP**.

### ****Step 2: Access the Service****

curl http://<external-ip>

✅ You should see the **Nginx welcome page**.

## ****📌 Summary****

✅ **ClusterIP** → Internal communication within the cluster.  
✅ **NodePort** → Exposes the service on a static port on each node.  
✅ **LoadBalancer** → Creates an external load balancer in cloud environments.  
✅ **Lab** covered creating and testing all three types of services.

# ****Understanding DaemonSets in Kubernetes****

## ****1. Introduction to DaemonSets****

A **DaemonSet** ensures that a copy of a specific Pod runs on every node in the cluster. Unlike Deployments or ReplicaSets, which scale based on the number of replicas, DaemonSets automatically create Pods on all nodes.

✅ Ensures that critical services run on **all or specific nodes**.  
✅ Automatically deploys Pods to **newly added nodes**.  
✅ Deletes Pods from **removed nodes**.

## ****2. Real-World Use Cases****

📌 **Logging Agents** – Run **Fluentd** or **Logstash** on every node to collect logs.  
📌 **Monitoring Agents** – Deploy **Prometheus Node Exporter** on every node to collect metrics.  
📌 **Security Agents** – Run security tools like **Falco** on all nodes for real-time security monitoring.  
📌 **Storage Daemons** – Use DaemonSets to run **Ceph or GlusterFS** on every node for distributed storage.

## ****3. Lab: Deploying a DaemonSet****

### ****Step 1: Create a DaemonSet for Fluentd****

Create a file fluentd-daemonset.yaml:

apiVersion: apps/v1

kind: DaemonSet

metadata:

name: fluentd-daemonset

labels:

app: fluentd

spec:

selector:

matchLabels:

app: fluentd

template:

metadata:

labels:

app: fluentd

spec:

tolerations:

- key: node-role.kubernetes.io/master

effect: NoSchedule

containers:

- name: fluentd

image: fluent/fluentd:v1.14

ports:

- containerPort: 24224

### ****Step 2: Deploy the DaemonSet****

kubectl apply -f fluentd-daemonset.yaml

Check if the DaemonSet is running:

kubectl get daemonsets

Check the Pods created:

kubectl get pods -o wide

✅ You should see a **Fluentd Pod running on every node**.

## ****4. Verifying DaemonSet Behavior****

### ****Step 1: Add a New Worker Node****

If you add a new node to the cluster, Kubernetes **automatically schedules a Fluentd Pod on it**:

kubectl get nodes

kubectl get pods -o wide | grep fluentd

✅ The new node will automatically run a Fluentd Pod.

### ****Step 2: Remove a Node****

If a node is **deleted or drained**, Kubernetes **removes the Fluentd Pod** from that node:

kubectl drain <node-name> --ignore-daemonsets --delete-emptydir-data

kubectl get pods -o wide | grep fluentd

✅ The DaemonSet **adjusts dynamically** to cluster changes.

## ****5. Limiting DaemonSets to Specific Nodes****

If you want to **run DaemonSets only on certain nodes**, use **node selectors** or **taints & tolerations**.

### ****Example: Running Fluentd Only on Worker Nodes****

Modify fluentd-daemonset.yaml:

spec:

template:

spec:

nodeSelector:

role: worker

Apply changes:

kubectl apply -f fluentd-daemonset.yaml

✅ Now, Fluentd will **only run on worker nodes**.

## ****📌 Summary****

✅ **DaemonSets** ensure that a Pod runs on all (or selected) nodes.  
✅ Used for **logging, monitoring, security, and storage agents**.  
✅ Automatically adapts when nodes are **added or removed**.  
✅ Supports **nodeSelector** and **tolerations** for node-specific scheduling.  
✅ **Lab covered deploying Fluentd using a DaemonSet**.

# ****Lab: Deploying Applications as a Pod in Kubernetes****

## ****1. Introduction****

A **Pod** is the smallest deployable unit in Kubernetes, representing **one or more containers** that share storage and network resources. In this lab, you will learn how to:

✅ Deploy an application as a **Pod**.  
✅ View and manage the Pod using kubectl.  
✅ Access the running application.  
✅ Understand Pod behavior and troubleshooting.

## ****2. Lab Environment Setup****

✅ **Pre-requisites:**

A Kubernetes cluster with **1 Master & 2 Worker Nodes** (kubeadm setup).

kubectl installed and configured.

Ensure your cluster is running:

kubectl get nodes

✅ All nodes should be in **Ready** state.

## ****3. Step-by-Step Guide: Deploying a Pod****

### ****Step 1: Create a Pod YAML File****

Create a new file nginx-pod.yaml:

apiVersion: v1

kind: Pod

metadata:

name: nginx-pod

labels:

app: nginx

spec:

containers:

- name: nginx

image: nginx:latest

ports:

- containerPort: 80

✅ This Pod runs an **Nginx** container on **port 80**.

### ****Step 2: Deploy the Pod****

Apply the YAML configuration:

kubectl apply -f nginx-pod.yaml

Check the Pod status:

kubectl get pods

✅ Expected Output:

NAME READY STATUS RESTARTS AGE

nginx-pod 1/1 Running 0 10s

### ****Step 3: Describe & Inspect the Pod****

View detailed Pod information:

kubectl describe pod nginx-pod

✅ This provides details on **IP address, container image, events, and logs**.

Check logs from the Pod:

kubectl logs nginx-pod

✅ This helps **debug application behavior**.

### ****Step 4: Access the Pod Internally****

Run a temporary test Pod:

kubectl run test-pod --image=busybox --restart=Never -- sleep 3600

Access the Nginx Pod from inside the cluster:

kubectl exec -it test-pod -- wget -qO- nginx-pod

✅ You should see the **Nginx welcome page**.

### ****Step 5: Delete the Pod****

To remove the Pod:

kubectl delete pod nginx-pod

Check if the Pod is deleted:

kubectl get pods

✅ The Pod is **no longer running**.

## ****4. Summary****

✅ **Pods** are the smallest deployable units in Kubernetes.  
✅ **kubectl** helps deploy, inspect, and manage Pods.  
✅ **Pods are temporary** and get replaced if deleted.  
✅ **Accessing a Pod** inside the cluster requires another Pod.  
✅ **Lab covered deploying, inspecting, and deleting a Pod**.

# ****Managing Labels & Selectors in Kubernetes****

## ****1. Introduction to Labels & Selectors****

In Kubernetes, **Labels** are key-value pairs assigned to resources (Pods, Services, Nodes, etc.) to **organize and identify** them. **Selectors** allow you to query resources based on labels.

✅ **Labels** help categorize resources (e.g., app=nginx, env=dev).  
✅ **Selectors** allow filtering and managing resources dynamically.  
✅ Used in **Pods, Deployments, Services, and Node scheduling**.

## ****2. Real-World Use Cases****

📌 **Service Discovery** – Grouping backend Pods with app=backend.  
📌 **Environment Segmentation** – Deploying apps in env=dev vs env=prod.  
📌 **Scaling** – Selecting a subset of Pods for rolling updates.  
📌 **Node Affinity** – Scheduling workloads to specific node groups.

## ****3. Lab: Using Labels & Selectors****

### ****Step 1: Create a Pod with Labels****

Create a file nginx-labeled-pod.yaml:

apiVersion: v1

kind: Pod

metadata:

name: nginx-labeled-pod

labels:

app: nginx

env: production

spec:

containers:

- name: nginx

image: nginx:latest

ports:

- containerPort: 80

✅ This Pod has labels app=nginx and env=production.

Apply the Pod:

kubectl apply -f nginx-labeled-pod.yaml

### ****Step 2: View Labels of a Pod****

List Pods with labels:

kubectl get pods --show-labels

✅ You will see the labels associated with each Pod.

To view a specific Pod's labels:

kubectl get pod nginx-labeled-pod --show-labels

### ****Step 3: Select Pods Using Label Selectors****

List all Pods with a specific label:

kubectl get pods -l app=nginx

List all Pods in **production environment**:

kubectl get pods -l env=production

✅ Selectors allow **dynamic filtering** of resources.

### ****Step 4: Modify Labels of an Existing Pod****

Add a new label:

kubectl label pod nginx-labeled-pod tier=frontend

Check if the label is applied:

kubectl get pods --show-labels

✅ The label tier=frontend is now added.

Remove a label:

kubectl label pod nginx-labeled-pod tier-

✅ The tier label is now removed.

### ****Step 5: Delete the Pod****

kubectl delete pod nginx-labeled-pod

✅ The Pod is removed.

## ****4. Summary****

✅ **Labels** categorize resources using key-value pairs.  
✅ **Selectors** filter and manage Kubernetes objects dynamically.  
✅ **Real-world use cases** include Service Discovery, Scaling, and Node Scheduling.  
✅ **Lab covered adding, modifying, and selecting Pods using labels**.

# ****Understanding Replication Controller & ReplicaSet in Kubernetes****

## ****1. Introduction to Replication Controller & ReplicaSet****

A **Replication Controller (RC)** ensures that a specified number of Pod replicas are running at any given time. It replaces failed Pods automatically.  
A **ReplicaSet (RS)** is an improved version of Replication Controller that supports **label selectors** with **match expressions** for more flexibility.

✅ Ensures **high availability** by maintaining the desired number of replicas.  
✅ **Auto-restarts** failed Pods to ensure reliability.  
✅ Supports **scaling** applications by increasing/decreasing replicas.

## ****2. Key Differences: Replication Controller vs ReplicaSet****

| **Feature** | **Replication Controller (RC)** | **ReplicaSet (RS)** |
| --- | --- | --- |
| Label Selector | Supports simple key-value pairs | Supports advanced match expressions |
| API Version | v1 | apps/v1 |
| Flexibility | Limited | More flexible |
| Recommended Usage | Legacy | Modern Alternative |

✅ **ReplicaSet is the preferred choice** in modern Kubernetes deployments.

## ****3. Real-World Use Cases****

📌 **High Availability** – Ensures critical applications always have a set number of running instances.  
📌 **Scaling** – Easily increase or decrease replicas to handle traffic.  
📌 **Self-healing** – Automatically restarts Pods if they fail.

## ****4. Lab: Deploying a ReplicaSet****

### ****Step 1: Create a ReplicaSet YAML File****

Create a file nginx-replicaset.yaml:

apiVersion: apps/v1

kind: ReplicaSet

metadata:

name: nginx-rs

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

✅ This **ReplicaSet** ensures **3 Nginx Pods** are always running.

### ****Step 2: Deploy the ReplicaSet****

Apply the YAML:

kubectl apply -f nginx-replicaset.yaml

Verify that the Pods are running:

kubectl get pods

✅ You should see 3 running Pods.

Check the ReplicaSet:

kubectl get replicaset

### ****Step 3: Test Auto-Healing Feature****

Delete one Pod manually:

kubectl delete pod <nginx-pod-name>

Check again:

kubectl get pods

✅ A new Pod is **automatically created** to maintain 3 replicas.

### ****Step 4: Scale the ReplicaSet****

Increase replicas to 5:

kubectl scale replicaset nginx-rs --replicas=5

Check the updated ReplicaSet:

kubectl get replicaset

kubectl get pods

✅ Now, **5 Pods** should be running.

### ****Step 5: Delete the ReplicaSet****

kubectl delete replicaset nginx-rs

✅ All Pods under the ReplicaSet will be removed.

## ****5. Summary****

✅ **Replication Controller & ReplicaSet** ensure high availability of applications.  
✅ **ReplicaSet is the modern replacement** for Replication Controller.  
✅ **Auto-healing** ensures Pods are recreated if they fail.  
✅ **Scaling is easy** with the kubectl scale command.  
✅ **Lab covered deploying, scaling, and self-healing behavior of ReplicaSets**.

# ****Understanding Kubernetes Services – ClusterIP, NodePort & LoadBalancer****

## ****1. Introduction to Kubernetes Services****

In Kubernetes, a **Service** provides a stable network endpoint to access a group of Pods. Since Pods are ephemeral (they can be recreated with different IPs), Services ensure **consistent access** to applications.

✅ **Decouples applications from Pod IP changes**  
✅ **Load balances traffic across multiple Pods**  
✅ **Supports different exposure methods (ClusterIP, NodePort, LoadBalancer, Ingress)**

## ****2. Types of Kubernetes Services****

| **Service Type** | **Access Method** | **Use Case** |
| --- | --- | --- |
| **ClusterIP** | Internal cluster-only access | Used for internal microservices |
| **NodePort** | Exposes service on each node's IP at a static port | Used for external access without a LoadBalancer |
| **LoadBalancer** | Provisions a cloud Load Balancer | Used for public-facing applications |

## ****3. Real-World Use Cases****

📌 **ClusterIP** – Internal database services (e.g., **MySQL, Redis**) communicating within the cluster.  
📌 **NodePort** – Exposing internal dashboards (e.g., **Prometheus, Grafana**) externally for monitoring.  
📌 **LoadBalancer** – Deploying **customer-facing web applications**.

## ****4. Lab: Creating and Managing Kubernetes Services****

### ****Step 1: Deploy an Application (Nginx)****

First, create a **Deployment** for the Nginx web server:

apiVersion: apps/v1

kind: Deployment

metadata:

name: nginx-deployment

spec:

replicas: 3

selector:

matchLabels:

app: nginx

template:

metadata:

labels:

app: nginx

spec:

containers:

- name: nginx

image: nginx:latest

ports:

- containerPort: 80

Apply the deployment:

kubectl apply -f nginx-deployment.yaml

Check if Pods are running:

kubectl get pods

✅ You should see 3 running Nginx Pods.

## ****5. Creating Different Service Types****

### ****1️⃣ ClusterIP (Internal Communication Only)****

Create a ClusterIP service:

apiVersion: v1

kind: Service

metadata:

name: nginx-clusterip

spec:

selector:

app: nginx

ports:

- protocol: TCP

port: 80

targetPort: 80

Apply it:

kubectl apply -f nginx-clusterip.yaml

Check the Service:

kubectl get services

✅ This service is only accessible **inside the cluster**.

Test it from a temporary Pod:

kubectl run test-pod --image=busybox --restart=Never -- sleep 3600

kubectl exec -it test-pod -- wget -qO- nginx-clusterip

✅ The Nginx page should load.

### ****2️⃣ NodePort (Exposing on External Node Port)****

Modify the service YAML to expose it externally:

apiVersion: v1

kind: Service

metadata:

name: nginx-nodeport

spec:

type: NodePort

selector:

app: nginx

ports:

- protocol: TCP

port: 80

targetPort: 80

nodePort: 30007

Apply the service:

kubectl apply -f nginx-nodeport.yaml

Get service details:

kubectl get services

✅ The service will be accessible on **any node's IP at port 30007**.

Test from your browser or terminal:

curl http://<Node-IP>:30007

✅ You should see the **Nginx welcome page**.

### ****3️⃣ LoadBalancer (Cloud-Managed Load Balancer)****

Modify the YAML to use a LoadBalancer:

apiVersion: v1

kind: Service

metadata:

name: nginx-loadbalancer

spec:

type: LoadBalancer

selector:

app: nginx

ports:

- protocol: TCP

port: 80

targetPort: 80

Apply it:

kubectl apply -f nginx-loadbalancer.yaml

Check the external IP:

kubectl get services

✅ If you're using a **cloud provider**, it will assign a **public IP**.

Test access:

curl http://<EXTERNAL-IP>

✅ You should see the **Nginx page**.

## ****6. Deleting Services****

To remove the services:

kubectl delete service nginx-clusterip

kubectl delete service nginx-nodeport

kubectl delete service nginx-loadbalancer

✅ All services will be deleted.

## ****7. Summary****

✅ **ClusterIP** – Default internal service for inter-Pod communication.  
✅ **NodePort** – Exposes service on a static external port.  
✅ **LoadBalancer** – Creates a cloud Load Balancer for public access.  
✅ **Lab covered creating, testing, and deleting Services**.

# ****Manual Scheduling of Pods in Kubernetes****

## ****1. Introduction to Manual Scheduling****

In Kubernetes, scheduling refers to placing Pods on Nodes. By default, the **kube-scheduler** automatically selects a Node based on available resources and constraints. **Manual scheduling** allows users to assign a Pod to a specific Node explicitly.

✅ **Used when kube-scheduler is disabled** or for debugging.  
✅ **Provides fine-grained control** over Pod placement.  
✅ **Useful for single-node clusters**, debugging, or learning how scheduling works.

## ****2. Real-World Use Cases****

📌 **Debugging Scheduling Issues** – Manually placing Pods on Nodes to check scheduling constraints.  
📌 **Testing Workloads** – Deploying Pods on specific nodes to analyze performance.  
📌 **Air-gapped Systems** – Running clusters without kube-scheduler.  
📌 **Specialized Hardware** – Assigning workloads to nodes with GPUs or high-memory configurations.

## ****3. Lab: Manual Scheduling of Pods****

### ****Step 1: Get the Node Names****

Check available nodes in your cluster:

kubectl get nodes

Example output:

NAME STATUS ROLES AGE VERSION

master-node Ready master 10d v1.27.0

worker-node1 Ready <none> 10d v1.27.0

worker-node2 Ready <none> 10d v1.27.0

✅ You have 1 master and 2 worker nodes.

### ****Step 2: Create a Pod with Manual Scheduling****

Create a **Pod definition** specifying a **NodeName**:

apiVersion: v1

kind: Pod

metadata:

name: manually-scheduled-pod

spec:

nodeName: worker-node1

containers:

- name: nginx

image: nginx:latest

ports:

- containerPort: 80

✅ The nodeName field **forces** the Pod to be scheduled on worker-node1.

Apply the Pod:

kubectl apply -f manually-scheduled-pod.yaml

Check if the Pod is running on the correct node:

kubectl get pods -o wide

Expected output:

NAME READY STATUS NODE AGE

manually-scheduled-pod 1/1 Running worker-node1 10s

✅ The Pod is **manually scheduled** on worker-node1.

### ****Step 3: What Happens if the Node is Unavailable?****

Drain the node and check if the Pod moves:

kubectl drain worker-node1 --ignore-daemonsets --delete-emptydir-data

Check Pods again:

kubectl get pods -o wide

🚨 The Pod **does NOT reschedule** because nodeName is a hard constraint.

✅ **Solution:** Manually delete the Pod and reschedule on another node.

### ****Step 4: Delete the Pod****

kubectl delete pod manually-scheduled-pod

✅ The manually scheduled Pod is now removed.

## ****4. Summary****

✅ **Manual scheduling bypasses the kube-scheduler** and forces a Pod onto a specific Node.  
✅ **Useful for debugging, testing, and constrained environments.**  
✅ **Pods do NOT move automatically** if a Node is unavailable.  
✅ **Lab covered creating, verifying, and handling failures in manual scheduling.**

# ****Taints and Tolerations in Kubernetes****

## ****1. Introduction to Taints and Tolerations****

Kubernetes **Taints** and **Tolerations** help control how Pods are scheduled on Nodes.

**Taints** are applied to **Nodes** to **repel** unwanted Pods.

**Tolerations** are added to **Pods** to **allow** them to run on tainted Nodes.

✅ **Prevent workloads from running on specific Nodes**  
✅ **Ensure only specific Pods run on certain Nodes**  
✅ **Commonly used for node isolation (e.g., dedicated workloads, GPU nodes, master nodes)**

## ****2. Real-World Use Cases****

📌 **Dedicated Nodes for Specific Workloads** – Tainting Nodes for databases, GPU workloads, or compliance reasons.  
📌 **Prevent Workloads on Master Nodes** – Ensuring only control plane components run on master nodes.  
📌 **Maintenance Mode** – Temporarily preventing Pods from being scheduled on a node.

## ****3. Lab: Using Taints and Tolerations in Kubernetes****

### ****Step 1: Check Available Nodes****

kubectl get nodes

Example output:

NAME STATUS ROLES AGE VERSION

master-node Ready master 10d v1.27.0

worker-node1 Ready <none> 10d v1.27.0

worker-node2 Ready <none> 10d v1.27.0

✅ We have **one master and two worker nodes**.

### ****Step 2: Apply a Taint to a Node****

Taint worker-node1 so only specific Pods can run on it:

kubectl taint nodes worker-node1 key1=value1:NoSchedule

Verify the taint:

kubectl describe node worker-node1 | grep Taint

Expected output:

Taints: key1=value1:NoSchedule

✅ This **prevents all Pods** from being scheduled on worker-node1, unless they have a matching toleration.

### ****Step 3: Deploy a Pod Without a Toleration (Expect Failure)****

Create a simple Pod without a toleration:

apiVersion: v1

kind: Pod

metadata:

name: tainted-pod

spec:

nodeName: worker-node1

containers:

- name: nginx

image: nginx:latest

ports:

- containerPort: 80

Apply the Pod:

kubectl apply -f tainted-pod.yaml

Check the Pod status:

kubectl get pods

🚨 **Expected result:** The Pod will be in Pending state because the Node is tainted.

Check why it is not scheduled:

kubectl describe pod tainted-pod

✅ The error will show No nodes available due to the **NoSchedule taint**.

### ****Step 4: Add a Toleration to Allow the Pod on the Tainted Node****

Modify the Pod YAML to include a **toleration**:

apiVersion: v1

kind: Pod

metadata:

name: tolerated-pod

spec:

tolerations:

- key: "key1"

operator: "Equal"

value: "value1"

effect: "NoSchedule"

containers:

- name: nginx

image: nginx:latest

ports:

- containerPort: 80

✅ The **toleration matches the taint**, allowing the Pod to be scheduled.

Apply the Pod:

kubectl apply -f tolerated-pod.yaml

Check if it is running:

kubectl get pods -o wide

✅ The Pod is now successfully scheduled on worker-node1.

### ****Step 5: Remove the Taint from the Node****

kubectl taint nodes worker-node1 key1=value1:NoSchedule-

Verify that the taint is removed:

kubectl describe node worker-node1 | grep Taint

✅ Now, all Pods can be scheduled on this node.

## ****4. Summary****

✅ **Taints prevent Pods from being scheduled unless tolerated.**  
✅ **Used for node isolation, maintenance, and specialized workloads.**  
✅ **Pods with matching tolerations can override taints.**  
✅ **Lab covered applying taints, testing pod scheduling, and removing taints.**

# ****Overview of Deployment in Kubernetes****

## ****1. Introduction to Deployments****

A **Deployment** in Kubernetes is a higher-level abstraction that manages the lifecycle of **Pods and ReplicaSets**. It provides features like:  
✅ **Rolling updates** to update applications without downtime.  
✅ **Rollback capability** in case of failures.  
✅ **Scaling** up or down based on demand.  
✅ **Self-healing** by automatically restarting failed Pods.

## ****2. Real-World Use Cases****

📌 **Zero-Downtime Updates** – Businesses deploy new versions of applications without service disruption.  
📌 **Disaster Recovery** – If an update fails, Kubernetes can automatically rollback.  
📌 **Load Handling** – Deployments can scale Pods dynamically based on traffic.  
📌 **A/B Testing** – Deploying different application versions to test new features.

## ****3. Key Concepts of Deployments****

### ****a) ReplicaSet****

Ensures that a specified number of **replica Pods** are running at all times.

If a Pod crashes, a new one is created.

### ****b) Rolling Updates****

Gradually replaces old Pods with new ones to ensure zero downtime.

### ****c) Rollbacks****

If a Deployment fails, Kubernetes can **rollback** to the previous stable version.

## ****4. Lab: Deploying an Application Using a Deployment****

### ****Step 1: Create a Deployment YAML****

Create a file nginx-deployment.yaml:

apiVersion: apps/v1

kind: Deployment

metadata:

name: nginx-deployment

spec:

replicas: 3

selector:

matchLabels:

app: nginx

template:

metadata:

labels:

app: nginx

spec:

containers:

- name: nginx

image: nginx:1.19

ports:

- containerPort: 80

✅ This **creates 3 replicas** of an nginx container.

### ****Step 2: Deploy the Application****

Apply the deployment:

kubectl apply -f nginx-deployment.yaml

Check the status:

kubectl get deployments

Expected output:

NAME READY UP-TO-DATE AVAILABLE AGE

nginx-deployment 3/3 3 3 10s

✅ Three replicas of nginx are running.

Check the Pods:

kubectl get pods -o wide

✅ The Pods are distributed across worker nodes.

### ****Step 3: Verify Self-Healing****

Delete a Pod and check if Kubernetes recreates it:

kubectl delete pod <pod-name>

Run:

kubectl get pods

✅ A new Pod should be created automatically.

### ****Step 4: Scale the Deployment****

Increase the number of replicas:

kubectl scale deployment nginx-deployment --replicas=5

Check the new replica count:

kubectl get deployments

✅ The number of replicas increased to 5.

### ****Step 5: Delete the Deployment****

kubectl delete deployment nginx-deployment

✅ The application is removed.

## ****5. Summary****

✅ **Deployments manage the lifecycle of Pods and ReplicaSets.**  
✅ **They enable rolling updates, rollbacks, and scaling.**  
✅ **Self-healing ensures high availability of applications.**  
✅ **Lab covered deploying, verifying, scaling, and deleting a deployment.**

# ****Deployment Strategies in Kubernetes****

## ****1. Introduction to Deployment Strategies****

When deploying a new version of an application, minimizing downtime and avoiding failures is crucial. Kubernetes supports several **deployment strategies** to manage updates efficiently. The two most commonly used strategies are:

✅ **Blue/Green Deployment** – Deploys a new version alongside the existing one and switches traffic when ready.  
✅ **Canary Deployment** – Gradually rolls out a new version to a small subset of users before full deployment.

Each strategy has its use cases depending on the business requirements, risk tolerance, and rollback needs.

## ****2. Blue/Green Deployment****

### ****a) How It Works****

**Blue Deployment (Current Version)**: The running application that users are accessing.

**Green Deployment (New Version)**: A new version deployed alongside Blue but not receiving traffic yet.

**Traffic Switch**: Once Green is tested and confirmed stable, traffic is switched from Blue to Green.

**Rollback**: If issues arise, switching back to Blue is instant.

### ****b) Real-World Use Cases****

📌 **Banking & FinTech** – Ensures zero downtime for mission-critical applications.  
📌 **E-commerce Platforms** – Prevents revenue loss during deployment failures.  
📌 **Enterprise Software Updates** – Allows thorough testing before release.

## ****3. Canary Deployment****

### ****a) How It Works****

A **small percentage** of traffic is routed to the new version (Canary).

If successful, more traffic is gradually shifted from the old version.

If errors occur, the Canary is rolled back without impacting all users.

### ****b) Real-World Use Cases****

📌 **Social Media Platforms** – Deploying features to a subset of users first.  
📌 **Cloud Services** – Testing new API versions with limited users.  
📌 **Mobile Apps Backend** – Rolling out new services to specific regions.

## ****4. Lab: Implementing Deployment Strategies****

### ****🔹 Lab 1: Blue/Green Deployment****

#### ****Step 1: Deploy the "Blue" Version****

Create a deployment for v1 of the application:

apiVersion: apps/v1

kind: Deployment

metadata:

name: app-blue

spec:

replicas: 3

selector:

matchLabels:

app: myapp

version: blue

template:

metadata:

labels:

app: myapp

version: blue

spec:

containers:

- name: myapp

image: myapp:v1

ports:

- containerPort: 80

Apply the deployment:

kubectl apply -f blue-deployment.yaml

Expose it via a service:

kubectl expose deployment app-blue --type=LoadBalancer --port=80 --target-port=80 --name=myapp-service

✅ myapp-service routes traffic to app-blue.

#### ****Step 2: Deploy the "Green" Version (New Release)****

Create a new deployment for v2 of the application:

apiVersion: apps/v1

kind: Deployment

metadata:

name: app-green

spec:

replicas: 3

selector:

matchLabels:

app: myapp

version: green

template:

metadata:

labels:

app: myapp

version: green

spec:

containers:

- name: myapp

image: myapp:v2

ports:

- containerPort: 80

Apply the deployment:

kubectl apply -f green-deployment.yaml

✅ The Green version is now running **but not receiving traffic**.

#### ****Step 3: Switch Traffic to Green****

Update the service to point to app-green:

kubectl patch service myapp-service -p '{"spec":{"selector":{"version":"green"}}}'

Check if traffic is now directed to app-green:

kubectl get pods -o wide

✅ Users now receive traffic from the new Green version.

#### ****Step 4: Rollback if Needed****

If issues occur, revert back to Blue:

kubectl patch service myapp-service -p '{"spec":{"selector":{"version":"blue"}}}'

✅ The rollback is instant, and users are back to the stable version.

### ****🔹 Lab 2: Canary Deployment****

#### ****Step 1: Deploy the Stable Version****

Create a deployment for v1 of the application:

apiVersion: apps/v1

kind: Deployment

metadata:

name: canary-app

spec:

replicas: 3

selector:

matchLabels:

app: myapp

version: stable

template:

metadata:

labels:

app: myapp

version: stable

spec:

containers:

- name: myapp

image: myapp:v1

ports:

- containerPort: 80

Apply the deployment:

kubectl apply -f stable-deployment.yaml

Expose it via a service:

kubectl expose deployment canary-app --type=LoadBalancer --port=80 --target-port=80 --name=myapp-service

✅ myapp-service routes traffic to the stable version.

#### ****Step 2: Deploy the Canary Version****

Create a smaller deployment for v2 with only **1 replica**:

apiVersion: apps/v1

kind: Deployment

metadata:

name: canary-release

spec:

replicas: 1

selector:

matchLabels:

app: myapp

version: canary

template:

metadata:

labels:

app: myapp

version: canary

spec:

containers:

- name: myapp

image: myapp:v2

ports:

- containerPort: 80

Apply the deployment:

kubectl apply -f canary-deployment.yaml

✅ The new version is deployed but only **1 Pod out of 4** gets traffic.

#### ****Step 3: Gradually Increase Canary Traffic****

Scale up the canary deployment gradually:

kubectl scale deployment canary-release --replicas=2

kubectl scale deployment canary-release --replicas=3

Monitor user feedback, and if successful, **switch fully**:

kubectl delete deployment canary-app

kubectl scale deployment canary-release --replicas=3

✅ The canary release **gradually replaces** the old version.

## ****5. Summary****

✅ **Blue/Green Deployment** – Deploys two versions side by side and switches traffic.  
✅ **Canary Deployment** – Rolls out a new version gradually to minimize risk.  
✅ **Labs covered deploying, switching traffic, and rolling back both strategies.**

# ****Understanding Environment Variables in Kubernetes****

## ****1. Introduction to Environment Variables****

Environment variables in Kubernetes allow us to **configure applications dynamically** without modifying their code. They are commonly used to:

✅ Store **configuration settings** (e.g., API URLs, feature flags).  
✅ Manage **sensitive data** (e.g., database credentials, API keys).  
✅ Enable **dynamic behavior** based on different environments (e.g., Dev, Staging, Prod).

## ****2. Using Plain Key Environment Variables****

Kubernetes allows you to define **environment variables directly** inside a Pod specification.

### ****Use Case****

📌 You need to provide an application with **static values** like log levels (DEBUG or INFO).

### ****Lab: Using Plain Key Environment Variables****

#### ****Step 1: Create a Pod with Environment Variables****

Create a file plain-env-pod.yaml:

apiVersion: v1

kind: Pod

metadata:

name: env-demo

spec:

containers:

- name: demo-container

image: busybox

command: ["sh", "-c", "echo 'LOG\_LEVEL is set to $LOG\_LEVEL' && sleep 3600"]

env:

- name: LOG\_LEVEL

value: "DEBUG"

✅ The environment variable LOG\_LEVEL is set to "DEBUG".

#### ****Step 2: Deploy the Pod****

Apply the configuration:

kubectl apply -f plain-env-pod.yaml

Check if the Pod is running:

kubectl get pods

#### ****Step 3: Verify Environment Variable Inside the Pod****

Exec into the Pod:

kubectl exec -it env-demo -- sh

Inside the Pod, run:

echo $LOG\_LEVEL

✅ The output should be:

DEBUG

#### ****Step 4: Cleanup****

Delete the Pod:

kubectl delete pod env-demo

### ****Key Takeaways****

✔ **Plain key environment variables** are simple but not secure for **sensitive data**.  
✔ They are useful for **non-confidential configurations** like log levels and feature flags.

# ****Using ConfigMaps for Environment Variables in Kubernetes****

## ****1. Introduction to ConfigMaps****

A **ConfigMap** in Kubernetes is used to store **non-sensitive configuration data** such as:

✅ Application settings (e.g., feature toggles, log levels).  
✅ Database connection parameters (without credentials).  
✅ External API endpoints or URLs.

Unlike **plain environment variables**, ConfigMaps **decouple configuration from application code**, making it easier to manage settings across multiple environments.

## ****2. Real-World Use Cases****

📌 **Microservices** – Storing service-specific configurations (e.g., PAYMENT\_GATEWAY\_URL).  
📌 **CI/CD Pipelines** – Injecting dynamic configuration values at deployment time.  
📌 **Multi-Environment Applications** – Keeping different settings for Dev, QA, and Production.

## ****3. Lab: Using ConfigMaps as Environment Variables****

### ****Step 1: Create a ConfigMap****

We will create a **ConfigMap** named app-config that holds application settings.

#### ****Method 1: Using a YAML File****

Create a file configmap.yaml:

apiVersion: v1

kind: ConfigMap

metadata:

name: app-config

data:

APP\_ENV: "production"

LOG\_LEVEL: "info"

FEATURE\_X\_ENABLED: "true"

Apply the ConfigMap:

kubectl apply -f configmap.yaml

✅ This creates a ConfigMap with three key-value pairs.

### ****Step 2: Use the ConfigMap in a Pod****

Now, create a Pod that **injects ConfigMap values as environment variables**.

Create a file configmap-pod.yaml:

apiVersion: v1

kind: Pod

metadata:

name: configmap-demo

spec:

containers:

- name: demo-container

image: busybox

command: ["sh", "-c", "echo 'APP\_ENV=$APP\_ENV, LOG\_LEVEL=$LOG\_LEVEL, FEATURE\_X\_ENABLED=$FEATURE\_X\_ENABLED' && sleep 3600"]

envFrom:

- configMapRef:

name: app-config

Apply the Pod configuration:

kubectl apply -f configmap-pod.yaml

✅ This Pod reads values from app-config and sets them as environment variables.

### ****Step 3: Verify Environment Variables in the Pod****

Exec into the Pod:

kubectl exec -it configmap-demo -- sh

Check the environment variables:

echo $APP\_ENV

echo $LOG\_LEVEL

echo $FEATURE\_X\_ENABLED

✅ The output should be:

production

info

true

### ****Step 4: Cleanup****

Delete the Pod and ConfigMap:

kubectl delete pod configmap-demo

kubectl delete configmap app-config

## ****4. Alternative: Mounting ConfigMap as a File****

Instead of using environment variables, ConfigMaps can also be **mounted as files** inside containers. This is useful for applications that read from config files instead of environment variables.

Create a file configmap-mount.yaml:

apiVersion: v1

kind: Pod

metadata:

name: configmap-mount-demo

spec:

containers:

- name: demo-container

image: busybox

command: ["sh", "-c", "cat /config/APP\_ENV && sleep 3600"]

volumeMounts:

- name: config-volume

mountPath: "/config"

readOnly: true

volumes:

- name: config-volume

configMap:

name: app-config

Apply the configuration:

kubectl apply -f configmap-mount.yaml

Check the mounted file:

kubectl exec -it configmap-mount-demo -- sh

ls /config

cat /config/APP\_ENV

✅ The file APP\_ENV is created, and its content is production.

## ****5. Key Takeaways****

✔ **ConfigMaps store non-sensitive configurations** and can be used as **environment variables** or **mounted files**.  
✔ They allow **better separation** between configuration and application logic.  
✔ **Not suitable for sensitive data** like passwords (use Secrets for that).

# ****Using Kubernetes Secrets for Sensitive Data****

## ****1. Introduction to Kubernetes Secrets****

**Kubernetes Secrets** are used to store **sensitive information** such as:

✅ Database credentials (e.g., username, password).  
✅ API keys and tokens.  
✅ TLS certificates and private keys.

Secrets are **encrypted** within the Kubernetes API server and can be accessed only by authorized users and applications. Unlike ConfigMaps, which are plain text, **Secrets are base64 encoded** and can be mounted as environment variables or volumes in a Pod.

## ****2. Real-World Use Cases****

📌 **Database Credentials** – Store credentials to securely connect to databases like MySQL or PostgreSQL.  
📌 **API Keys** – Store keys to third-party services like AWS, Google Cloud, or external APIs.  
📌 **TLS/SSL Certificates** – Secure communication between services by managing certificates.

## ****3. Lab: Using Secrets as Environment Variables****

### ****Step 1: Create a Secret****

Let's create a Secret to store sensitive information like a **database password**.

#### ****Method 1: Create a Secret from Literal Values****

kubectl create secret generic db-secret --from-literal=DB\_PASSWORD=mysecretpassword

This command creates a Secret named db-secret containing a DB\_PASSWORD key with the value mysecretpassword. The value is **base64 encoded** by Kubernetes.

You can verify the Secret with:

kubectl get secret db-secret -o yaml

This will display the encoded value of DB\_PASSWORD.

#### ****Method 2: Create a Secret from a File****

Alternatively, you can create a Secret from a file. For example, a file named db-password.txt with the content mysecretpassword:

kubectl create secret generic db-secret --from-file=DB\_PASSWORD=db-password.txt

### ****Step 2: Use the Secret in a Pod****

Now, create a Pod that uses this Secret as an environment variable.

Create a file secret-pod.yaml:

apiVersion: v1

kind: Pod

metadata:

name: secret-demo

spec:

containers:

- name: demo-container

image: busybox

command: ["sh", "-c", "echo 'Database password is $DB\_PASSWORD' && sleep 3600"]

envFrom:

- secretRef:

name: db-secret

Apply the Pod configuration:

kubectl apply -f secret-pod.yaml

This configuration will inject the DB\_PASSWORD value from the Secret into the container as an environment variable.

### ****Step 3: Verify Environment Variables in the Pod****

Exec into the Pod:

kubectl exec -it secret-demo -- sh

Check the environment variable:

echo $DB\_PASSWORD

✅ The output should be:

mysecretpassword

### ****Step 4: Cleanup****

Delete the Pod and Secret:

kubectl delete pod secret-demo

kubectl delete secret db-secret

## ****4. Mounting Secrets as Files****

Instead of injecting Secrets as environment variables, you can **mount them as files** inside the container. This is ideal for applications that require secrets in **file format** (e.g., private keys or certificates).

Create a file secret-mount.yaml:

apiVersion: v1

kind: Pod

metadata:

name: secret-mount-demo

spec:

containers:

- name: demo-container

image: busybox

command: ["sh", "-c", "cat /etc/secrets/DB\_PASSWORD && sleep 3600"]

volumeMounts:

- name: secret-volume

mountPath: "/etc/secrets"

readOnly: true

volumes:

- name: secret-volume

secret:

secretName: db-secret

Apply the configuration:

kubectl apply -f secret-mount.yaml

Check the mounted file:

kubectl exec -it secret-mount-demo -- sh

ls /etc/secrets

cat /etc/secrets/DB\_PASSWORD

✅ The content of the file should be:

mysecretpassword

## ****5. Key Takeaways****

✔ **Secrets are for sensitive information** and can be injected as environment variables or mounted as files.  
✔ Kubernetes **encrypts Secrets** at rest and base64 encodes them.  
✔ They provide a **secure** way to manage credentials, tokens, and keys in Kubernetes applications.

### ****Next Steps:****

Once you're ready, let me know, and I can guide you to the next topic!

Certainly! Let's dive deeper into each topic one by one, with more details and explanations, and we will include labs wherever applicable.

## ****1. Understanding Volume Management in Kubernetes****

### ****What are Volumes in Kubernetes?****

In Kubernetes, a **Volume** is a directory that is accessible to all containers in a Pod. It is used to store data that should persist beyond the lifecycle of individual containers, or when containers are restarted. By default, containers in a Pod have **ephemeral storage** that is lost when the container is terminated. Volumes allow us to store data that should be retained across restarts and even across Pods.

### ****Types of Volumes in Kubernetes:****

**Ephemeral Volumes:**

These volumes only exist for the duration of a Pod's life and are deleted once the Pod is terminated.

Examples: emptyDir, configMap, secret.

**Persistent Volumes (PVs):**

These are volumes that exist independently of Pods and can persist data beyond Pod restarts or terminations.

They are tied to a **Persistent Volume Claim (PVC)**, which is the user request for storage resources.

**Dynamic Provisioning:**

Kubernetes can automatically provision storage for a PVC using **Storage Classes**. This allows administrators to configure the way volumes are provisioned and mounted.

**Volume Plugins:**

Kubernetes supports multiple types of volume plugins such as:

**HostPath**: Uses a file or directory on the node as the volume.

**NFS**: Network File System for shared storage.

**AWS EBS/Google Cloud Persistent Disk**: Cloud block storage.

**CephFS**, **GlusterFS**, etc. for distributed storage.

### ****Real-World Use Case:****

Imagine you are deploying a **Database** like MySQL on Kubernetes. Databases require persistent storage because they store data, and it’s critical to maintain that data even when the Pod restarts. So, we would create a Persistent Volume (PV) and attach it to the Pod using a Persistent Volume Claim (PVC).

## ****2. Types of Volumes Provisioning****

### ****Static Provisioning****

In static provisioning, the administrator manually creates the **Persistent Volume (PV)**. Once the PV is created, it can be claimed by a Pod via a **Persistent Volume Claim (PVC)**.

The admin specifies the details such as the storage size, access mode, and volume type (e.g., NFS, HostPath).

**Steps for Static Provisioning:**

Define a Persistent Volume (PV).

Define a Persistent Volume Claim (PVC) to match the storage requested by the Pod.

The PVC is bound to a matching PV.

### ****Dynamic Provisioning****

In dynamic provisioning, when a **PVC** is created, Kubernetes will automatically provision a **PV** that matches the storage request defined in the PVC. This requires a **Storage Class** that defines the parameters of the storage.

The dynamic provisioning process is especially useful in environments where the volume type needs to be decided dynamically (for example, in cloud environments).

**Steps for Dynamic Provisioning:**

Define a **Storage Class**.

Define a **PVC** that uses the Storage Class.

Kubernetes will create and bind a PV based on the requested storage class.

### ****Real-World Use Case for Dynamic Provisioning:****

In a cloud environment, you might use dynamic provisioning to automatically provision volumes like **AWS EBS** or **Google Persistent Disk** when a PVC is created. This eliminates the need for manual intervention to create a volume.

## ****3. Persistent Volumes (PV)****

### ****What is a Persistent Volume (PV)?****

A **Persistent Volume (PV)** is a piece of storage in the Kubernetes cluster that is provisioned by an administrator. PVs are not tied to the lifecycle of a Pod. They are usually backed by physical storage in a data center or a cloud provider’s storage.

#### ****Key Features of Persistent Volumes:****

**Capacity**: The amount of storage allocated (e.g., 1Gi for 1 GiB of storage).

**Access Modes**: Defines how the volume can be accessed:

**ReadWriteOnce (RWO)**: The volume can be mounted as read-write by a single node.

**ReadOnlyMany (ROX)**: The volume can be mounted as read-only by many nodes.

**ReadWriteMany (RWX)**: The volume can be mounted as read-write by many nodes.

**ReclaimPolicy**: Determines what happens to the volume after the PVC is deleted:

**Retain**: Keeps the PV after the PVC is deleted.

**Recycle**: Deletes the content of the volume but retains the PV itself.

**Delete**: Deletes the volume and the content when the PVC is deleted.

#### ****Example:****

A sample PersistentVolume definition for an NFS volume could look like this:

apiVersion: v1

kind: PersistentVolume

metadata:

name: my-nfs-pv

spec:

capacity:

storage: 5Gi

accessModes:

- ReadWriteMany

persistentVolumeReclaimPolicy: Retain

nfs:

path: /mnt/data

server: nfs-server.example.com

This PV uses an NFS server for its storage and is configured for **ReadWriteMany** access.

## ****Lab: Creating a Persistent Volume (PV)****

### ****Step 1: Define a Persistent Volume (PV)****

Let's create a PersistentVolume backed by a HostPath volume.

Create a file pv-hostpath.yaml:

apiVersion: v1

kind: PersistentVolume

metadata:

name: hostpath-pv

spec:

capacity:

storage: 1Gi

accessModes:

- ReadWriteOnce

hostPath:

path: /mnt/data

This PV uses the HostPath type to store data in /mnt/data on the host machine.

Apply the PV:

kubectl apply -f pv-hostpath.yaml

### ****Step 2: Check the Persistent Volume****

To verify if the PV was created:

kubectl get pv

## ****4. Persistent Volume Claims (PVC)****

### ****What is a Persistent Volume Claim (PVC)?****

A **Persistent Volume Claim (PVC)** is a request made by a user or application for storage resources within Kubernetes. It is essentially a "claim" on a **Persistent Volume (PV)**, specifying the **storage size**, **access modes**, and sometimes the **storage class**.

Kubernetes uses PVCs to abstract away the details of the underlying storage. The PVC connects Pods to PVs without the need to know where the data is stored.

### ****Key Features of PVCs:****

**Access Modes**: The access mode defined in the PVC (e.g., ReadWriteOnce, ReadOnlyMany) must be compatible with the PV’s access modes.

**Storage Size**: The PVC specifies the requested storage size, and Kubernetes ensures that the PV bound to the PVC meets or exceeds this size.

**Storage Class**: The PVC may specify a **Storage Class** which determines how the volume should be provisioned (e.g., dynamically or statically).

Once a PVC is created, Kubernetes will either:

**Bind the PVC to an available PV**: The PV’s capacity and access modes must satisfy the PVC's request.

**Dynamically provision a PV**: If no matching PV exists, and dynamic provisioning is enabled via a **StorageClass**, Kubernetes will create a PV that satisfies the PVC's request.

### ****Real-World Use Case for PVC:****

Suppose you need to deploy a **WordPress application** and store its data persistently. Rather than hardcoding the storage details, you can create a PVC requesting a specific amount of storage, and the system will automatically bind it to a suitable PV (either static or dynamically provisioned).

### ****Lab: Creating a Persistent Volume Claim (PVC)****

Now, let’s create a **Persistent Volume Claim** that will bind to the **Persistent Volume** we created earlier.

### ****Step 1: Create a Persistent Volume Claim (PVC)****

Create a file pvc-hostpath.yaml:

apiVersion: v1

kind: PersistentVolumeClaim

metadata:

name: hostpath-pvc

spec:

accessModes:

- ReadWriteOnce

resources:

requests:

storage: 1Gi

This PVC requests 1Gi of storage with **ReadWriteOnce** access mode, which matches the earlier defined PV's configuration.

### ****Step 2: Apply the PVC****

Run the following command to apply the PVC:

kubectl apply -f pvc-hostpath.yaml

### ****Step 3: Verify the PVC****

To check if the PVC has been created successfully and if it’s bound to a PV, run:

kubectl get pvc

You should see the PVC with a STATUS of Bound if it successfully bound to the PV.

### ****Real-World Use Case:****

Let’s imagine an application like **PostgreSQL** that needs persistent storage for its database files. A PVC would be created to request storage from the cloud provider (e.g., AWS EBS) and this PVC would be bound to the relevant PV to provide reliable storage for the database.

## ****5. Understanding Storage Class in Kubernetes****

### ****What is a Storage Class?****

A **Storage Class** in Kubernetes provides a way to describe the "types" of storage that a cluster can offer. It is used to define **provisioner** information, **reclaim policies**, and **volume binding modes** for dynamic provisioning of persistent storage. The Storage Class abstracts the complexity of manually provisioning storage for Pods, allowing Kubernetes to automatically provision and manage volumes based on specific requirements.

Each Storage Class can define:

**Provisioner**: The volume plugin that is used for provisioning (e.g., AWS EBS, Google Persistent Disk, NFS, etc.).

**ReclaimPolicy**: What happens to the volume when it is no longer needed (similar to PV's ReclaimPolicy).

Delete - The volume is deleted when the PVC is deleted.

Retain - The volume is retained for future use.

Recycle - The volume content is scrubbed and reused.

**Parameters**: Specific configurations for the storage provisioner (e.g., for cloud providers like AWS, the type of the EBS volume).

**VolumeBindingMode**: This specifies when the volume should be provisioned and bound to the PVC:

**Immediate**: The PVC is bound to a PV as soon as it’s created.

**WaitForFirstConsumer**: The volume is not provisioned until a Pod that uses the PVC is scheduled.

### ****Real-World Use Case for Storage Class:****

In a cloud-native application, such as a **high-availability database** (e.g., PostgreSQL or MongoDB), a Kubernetes Storage Class can be configured to dynamically provision a high-performance **SSD-backed storage volume** when a PVC is created. This ensures that every instance of the database has fast and reliable storage without manual intervention.

### ****Lab: Understanding and Using Storage Class****

Let’s now walk through creating a **Storage Class** that enables dynamic provisioning of volumes.

### ****Step 1: Create a Storage Class****

Create a file storage-class.yaml that defines a Storage Class for dynamic provisioning using **AWS EBS** as the backend provisioner:

apiVersion: storage.k8s.io/v1

kind: StorageClass

metadata:

name: ebs-sc

provisioner: kubernetes.io/aws-ebs

parameters:

type: gp2 # EBS volume type, for general-purpose SSD

fsType: ext4

reclaimPolicy: Retain

volumeBindingMode: WaitForFirstConsumer

provisioner: Specifies the volume provisioner to use. In this case, kubernetes.io/aws-ebs will provision EBS volumes on AWS.

parameters: Includes any configuration for the provisioner, such as the type of EBS volume (gp2 for general-purpose SSD) and the fsType (file system type).

reclaimPolicy: We set this to Retain, which means that when a PVC is deleted, the volume is not automatically deleted but will be retained for future use.

volumeBindingMode: Set to WaitForFirstConsumer, meaning the volume will not be created until the PVC is actually used by a Pod.

Apply the Storage Class:

kubectl apply -f storage-class.yaml

### ****Step 2: Create a PVC Using the Storage Class****

Create a file pvc-with-storage-class.yaml to request storage from the newly defined Storage Class:

apiVersion: v1

kind: PersistentVolumeClaim

metadata:

name: ebs-pvc

spec:

accessModes:

- ReadWriteOnce

resources:

requests:

storage: 5Gi

storageClassName: ebs-sc # Reference the Storage Class

This PVC requests 5Gi of storage and references the ebs-sc Storage Class for dynamic provisioning.

Apply the PVC:

kubectl apply -f pvc-with-storage-class.yaml

### ****Step 3: Verify the PVC and PV****

After the PVC is created, Kubernetes will dynamically provision the volume. You can check the status of the PVC and the bound PV:

kubectl get pvc

kubectl get pv

You should see the PVC bound to a dynamically provisioned PV.

### ****Real-World Use Case for Storage Class:****

Imagine you're running a **large-scale video processing application** where each Pod requires a large volume of high-speed storage. You can configure a **Storage Class** with SSD provisioning to ensure that each new PVC request is fulfilled with a high-performance disk, ensuring fast read/write access to the video files.

## ****1. Understanding Kubernetes Authentication****

### ****What is Kubernetes Authentication?****

Kubernetes uses **Authentication** to verify the identity of users, services, and applications that interact with the Kubernetes API server. Authentication is the first step in ensuring that only authorized users can interact with the Kubernetes cluster.

Kubernetes provides several mechanisms for authenticating users and services:

### ****Authentication Methods:****

**Certificates**: Client certificates issued by the API server or a Certificate Authority (CA).

Kubernetes supports client certificates where each user/service has a public/private key pair.

**Bearer Tokens**: Used by applications that interact with the Kubernetes API (e.g., when the API server needs to interact with the Kubelet or when you use kubectl to interact with the cluster).

Tokens are typically generated by an identity provider or service account.

**OpenID Connect (OIDC)**: Kubernetes supports integration with identity providers that use OIDC, such as Google, Azure Active Directory, or LDAP. This provides centralized authentication.

**Service Accounts**: Service accounts are special accounts used by pods to authenticate with the Kubernetes API. A service account’s token is automatically mounted into a Pod, allowing the container to interact with the Kubernetes API.

**Basic Authentication**: Deprecated, but it allows username/password authentication. It is not recommended to use this method.

### ****Real-World Use Case:****

Imagine a scenario where you have a Kubernetes cluster with different user groups. Developers need to deploy and manage applications, while the Operations team needs to manage the infrastructure. Kubernetes Authentication ensures that each team only has access to the resources they need.

**For example**:

A developer should have access to kubectl commands to manage deployments but should not have access to manage **nodes** or **etcd**.

A service account for an application running inside Kubernetes can authenticate with the Kubernetes API to create and manage resources without giving it full administrative access.

### ****Lab: Creating and Managing Users in Kubernetes****

Now, let’s walk through creating users and setting up **Authentication** in Kubernetes.

In this lab, we will use **Service Accounts** and **Bearer Tokens** to authenticate Pods and users.

### ****Step 1: Create a Service Account****

To allow an application running inside Kubernetes to authenticate with the API server, we’ll create a **Service Account**.

Create a file service-account.yaml:

apiVersion: v1

kind: ServiceAccount

metadata:

name: my-service-account

This YAML creates a service account named my-service-account.

Apply the Service Account:

kubectl apply -f service-account.yaml

### ****Step 2: Get the Bearer Token for the Service Account****

Once the Service Account is created, Kubernetes automatically generates a secret containing a **Bearer Token** for the service account. We can retrieve the token using the following command:

kubectl get secret $(kubectl get serviceaccount my-service-account -o jsonpath='{.secrets[0].name}') -o jsonpath='{.data.token}' | base64 --decode

This command extracts the Bearer Token associated with the my-service-account.

### ****Step 3: Test Authentication Using the Token****

You can use this token to authenticate with the Kubernetes API using curl. For example, to check the cluster info:

curl -k -H "Authorization: Bearer <token>" https://<k8s-api-server-url>/api

Replace <token> with the value obtained from the previous step and <k8s-api-server-url> with the URL of your Kubernetes API server.

## ****2. Understanding Role, ClusterRole, RoleBinding & ClusterRoleBinding****

### ****What are Roles and RoleBindings in Kubernetes?****

In Kubernetes, **Roles** and **RoleBindings** are used to manage **RBAC** (Role-Based Access Control) to grant users or service accounts specific permissions on resources within a cluster. This allows you to control access based on the user's identity and their role.

### ****Role (rbac.authorization.k8s.io/v1)****

A **Role** defines a set of permissions within a specific namespace. It grants access to resources (such as Pods, Deployments, ConfigMaps) within that namespace.

**Rules**: A Role consists of a list of rules that specify what actions can be performed on which resources. For example, it might allow reading and writing to Pods in a specific namespace.

**Scope**: A Role is scoped to a single namespace.

Example Role that allows users to list and get Pods within a namespace:

apiVersion: rbac.authorization.k8s.io/v1

kind: Role

metadata:

namespace: default

name: pod-reader

rules:

- apiGroups: [""]

resources: ["pods"]

verbs: ["get", "list"]

### ****RoleBinding (rbac.authorization.k8s.io/v1)****

A **RoleBinding** grants the permissions defined in a **Role** to a user or service account within a specific namespace.

**Subjects**: The RoleBinding defines which users, groups, or service accounts have the permissions.

**Role Reference**: The RoleBinding references the Role that defines the permissions to be granted.

Example RoleBinding that grants the pod-reader role to a service account default/my-service-account:

apiVersion: rbac.authorization.k8s.io/v1

kind: RoleBinding

metadata:

name: read-pods-binding

namespace: default

subjects:

- kind: ServiceAccount

name: my-service-account

namespace: default

roleRef:

kind: Role

name: pod-reader

apiGroup: rbac.authorization.k8s.io

### ****ClusterRole (rbac.authorization.k8s.io/v1)****

A **ClusterRole** is similar to a **Role**, but it is cluster-wide and can be applied across all namespaces. ClusterRoles are often used to grant permissions to resources that are not namespaced, like Nodes or PersistentVolumes, or to provide permissions that apply across the entire cluster.

Example ClusterRole that grants access to Nodes:

apiVersion: rbac.authorization.k8s.io/v1

kind: ClusterRole

metadata:

# Cluster-wide permissions

name: node-reader

rules:

- apiGroups: [""]

resources: ["nodes"]

verbs: ["get", "list"]

### ****ClusterRoleBinding (rbac.authorization.k8s.io/v1)****

A **ClusterRoleBinding** is used to bind a **ClusterRole** to a user, service account, or group across all namespaces in the cluster.

Example ClusterRoleBinding that grants the node-reader ClusterRole to the default/my-service-account service account:

apiVersion: rbac.authorization.k8s.io/v1

kind: ClusterRoleBinding

metadata:

name: read-nodes-binding

subjects:

- kind: ServiceAccount

name: my-service-account

namespace: default

roleRef:

kind: ClusterRole

name: node-reader

apiGroup: rbac.authorization.k8s.io

### ****Real-World Use Case:****

In an organization with multiple teams, the **Operations team** might need to monitor the state of the **Kubernetes nodes**. You could create a **ClusterRole** like node-reader that grants read-only access to the nodes. Then, use a **ClusterRoleBinding** to bind this role to an Operations team service account.

Similarly, a **developer team** might need a **Role** to manage only their application resources in a specific namespace (e.g., Pods, Deployments, etc.). A **RoleBinding** would be used to grant that Role to the development team’s service account.

### ****Lab: Creating and Managing Roles and RoleBindings****

Now, let’s create some **Roles** and **RoleBindings** in Kubernetes.

### ****Step 1: Create a Role for Pod Access****

Create a file role-pod-reader.yaml to define a role that allows read-only access to Pods:

apiVersion: rbac.authorization.k8s.io/v1

kind: Role

metadata:

namespace: default

name: pod-reader

rules:

- apiGroups: [""]

resources: ["pods"]

verbs: ["get", "list"]

Apply the Role:

kubectl apply -f role-pod-reader.yaml

### ****Step 2: Create a RoleBinding for a Service Account****

Create a file rolebinding-pod-reader.yaml to bind the pod-reader Role to a service account my-service-account:

apiVersion: rbac.authorization.k8s.io/v1

kind: RoleBinding

metadata:

name: read-pods-binding

namespace: default

subjects:

- kind: ServiceAccount

name: my-service-account

namespace: default

roleRef:

kind: Role

name: pod-reader

apiGroup: rbac.authorization.k8s.io

Apply the RoleBinding:

kubectl apply -f rolebinding-pod-reader.yaml

### ****Step 3: Verify the Role and RoleBinding****

To verify the Role and RoleBinding:

kubectl get role,rolebinding -n default

### ****Real-World Use Case:****

In a production environment, only specific users should be allowed to **scale** or **delete Pods**. For this, you can create a **Role** that grants those specific privileges, and then bind it to a set of users or service accounts who require those permissions.

## ****3. Understanding Security Context****

### ****What is a Security Context?****

A **Security Context** in Kubernetes defines privilege and access control settings for a Pod or a container. It helps control the behavior and security of a container by specifying security-related attributes such as user, group, and capabilities. A security context can be set at both the **Pod level** and the **Container level**.

### ****Key Security Context Options:****

**RunAsUser**: Defines the user ID (UID) to run the container as. It allows you to specify a non-root user for better security.

Example: runAsUser: 1000 makes the container run as user 1000 instead of root (UID 0).

**RunAsGroup**: Defines the group ID (GID) the container should run as.

Example: runAsGroup: 3000 makes the container run under group 3000.

**FSGroup**: Sets a group ID for volume ownership. This is useful when containers need to have the same GID to access shared volumes.

**Privileged**: Specifies whether the container runs in privileged mode. In privileged mode, containers have elevated access to the host system. This is typically used for containers that need to interact with the host’s kernel (e.g., networking containers, containers requiring access to hardware).

Example: privileged: true allows the container to have full control over the host system.

**Capabilities**: Kubernetes allows you to add or remove Linux capabilities from containers. By default, containers are given a limited set of capabilities. Additional capabilities can be added using the add field, or unwanted capabilities can be dropped using the drop field.

Example: capabilities: { add: ["NET\_ADMIN"], drop: ["NET\_RAW"] } adds the NET\_ADMIN capability but removes NET\_RAW.

**AllowPrivilegeEscalation**: When set to false, this setting prevents the container from gaining more privileges than its parent process.

Example: allowPrivilegeEscalation: false stops a container from escalating privileges (e.g., root privileges).

**SELinuxOptions**: Kubernetes allows the specification of SELinux options to control access based on security labels.

Example: seLinuxOptions: { level: "s0:c123,c456" }.

**SeccompProfile**: Seccomp (Secure Computing Mode) is a Linux kernel feature that restricts the system calls a container can make. Kubernetes allows you to specify a Seccomp profile to limit the system calls a container can use.

Example: seccompProfile: { type: RuntimeDefault } applies the default seccomp profile to the container.

**ReadOnlyRootFilesystem**: If set to true, the root filesystem of the container will be mounted as read-only, preventing any modifications to files within the container.

Example: readOnlyRootFilesystem: true is used to ensure that no files can be modified within the container.

### ****Real-World Use Case:****

**Non-root Users**: In a security-sensitive environment, such as running an application that handles sensitive data (e.g., customer data), containers should not run as root. A Security Context can ensure the application runs with a non-root user to minimize the risk of privilege escalation attacks.

**Privileged Containers**: Certain containers, such as those used for **networking** or **device management**, may need to run with elevated privileges. These containers can be explicitly granted privileged access through the Security Context.

**Immutable Containers**: For highly secure applications, such as databases that should not be tampered with, the root filesystem of the container can be mounted as read-only. This would prevent unauthorized access or modification to sensitive files within the container.

### ****Lab: Adding Security Context to a Pod to Enable Ping****

In this lab, we will configure a **Security Context** for a Kubernetes Pod and add the necessary configuration to allow it to run ping commands, which typically require elevated privileges.

### ****Step 1: Create a Pod with Security Context****

Create a file pod-with-security-context.yaml that defines a Pod with a Security Context to allow the ping command. The NET\_ADMIN capability is required for the ping operation.

apiVersion: v1

kind: Pod

metadata:

name: pod-with-ping

spec:

containers:

- name: ping-container

image: busybox

command: ["sleep", "3600"]

securityContext:

capabilities:

add: ["NET\_ADMIN"]

runAsUser: 1000

runAsGroup: 3000

allowPrivilegeEscalation: false

### ****Explanation****:

securityContext: Defines the security attributes for the container.

capabilities.add: Adds the NET\_ADMIN capability to allow the container to execute network administration commands like ping.

runAsUser and runAsGroup: Set the container to run as a non-root user (UID 1000) and group (GID 3000).

allowPrivilegeEscalation: false: Prevents the container from escalating privileges.

### ****Step 2: Apply the Pod Configuration****

Apply the configuration to the Kubernetes cluster:

kubectl apply -f pod-with-security-context.yaml

### ****Step 3: Verify Pod Creation and Check the Container's Capabilities****

To check if the Pod has been created successfully and to view the capabilities of the container:

kubectl get pods

kubectl describe pod pod-with-ping

You can also use kubectl exec to enter the container and verify the ability to run ping:

kubectl exec -it pod-with-ping -- ping 8.8.8.8

This should allow the container to run the ping command because it has been granted the NET\_ADMIN capability through the security context.

### ****Step 4: Modify the Pod's Security Context to Restrict Capabilities (Optional)****

If you want to restrict the container's capabilities (e.g., removing unnecessary privileges), modify the Security Context like so:

capabilities:

drop: ["NET\_RAW"]

Apply the changes:

kubectl apply -f pod-with-security-context.yaml

This drops the NET\_RAW capability, further restricting the container's ability to perform raw network operations.

### ****Real-World Use Case:****

In an environment where security is a top priority (e.g., a **financial institution**), containers running applications should not have excessive privileges. By using the Security Context to limit privileges (e.g., not allowing root access, restricting capabilities), you can reduce the attack surface of your Kubernetes applications.

## ****4. Labs: Managing Service Accounts and Roles****

### ****Lab 1: Creating and Managing Service Accounts****

A **ServiceAccount** in Kubernetes is an account for processes in a Pod to interact with the Kubernetes API. Service accounts are useful when your application needs to authenticate to the Kubernetes API, such as when it needs to access secrets or configuration maps.

### ****Step 1: Create a Service Account****

First, let's create a ServiceAccount named my-service-account in the default namespace.

Create a file service-account.yaml with the following content:

apiVersion: v1

kind: ServiceAccount

metadata:

name: my-service-account

namespace: default

Apply the configuration:

kubectl apply -f service-account.yaml

### ****Step 2: Verify Service Account Creation****

Check the status of the service account to ensure it was created successfully:

kubectl get serviceaccount my-service-account -n default

### ****Step 3: Attach the Service Account to a Pod****

We can now use this service account in a Pod by referencing it in the Pod specification. Create a file pod-with-service-account.yaml:

apiVersion: v1

kind: Pod

metadata:

name: pod-with-service-account

spec:

serviceAccountName: my-service-account

containers:

- name: nginx

image: nginx

ports:

- containerPort: 80

This will attach the my-service-account service account to the Pod. Now apply the configuration:

kubectl apply -f pod-with-service-account.yaml

### ****Step 4: Verify Service Account in Pod****

You can verify that the Pod is using the specified service account by checking the serviceAccountName:

kubectl describe pod pod-with-service-account

Under the "Service Account" section, it should show my-service-account.

### ****Lab 2: Managing Roles and RoleBindings****

Now, let's manage **Roles** and **RoleBindings** to control access for this service account.

### ****Step 1: Create a Role****

Let's create a **Role** that allows get and list operations on Pods in the default namespace.

Create a file role-pod-reader.yaml:

apiVersion: rbac.authorization.k8s.io/v1

kind: Role

metadata:

namespace: default

name: pod-reader

rules:

- apiGroups: [""]

resources: ["pods"]

verbs: ["get", "list"]

Apply the role:

kubectl apply -f role-pod-reader.yaml

### ****Step 2: Create a RoleBinding****

Create a **RoleBinding** to bind the pod-reader role to the my-service-account service account.

Create a file rolebinding-pod-reader.yaml:

apiVersion: rbac.authorization.k8s.io/v1

kind: RoleBinding

metadata:

name: read-pods-binding

namespace: default

subjects:

- kind: ServiceAccount

name: my-service-account

namespace: default

roleRef:

kind: Role

name: pod-reader

apiGroup: rbac.authorization.k8s.io

Apply the role binding:

kubectl apply -f rolebinding-pod-reader.yaml

### ****Step 3: Verify RoleBinding****

To check if the RoleBinding was successfully created, run:

kubectl get rolebinding -n default

You should see the read-pods-binding listed.

### ****Step 4: Verify Access with Service Account****

To test if the service account has the appropriate access, we can try accessing the Pods with the service account.

Run the following command to impersonate the my-service-account and list the Pods:

kubectl auth can-i list pods --as=system:serviceaccount:default:my-service-account

If the permissions are set correctly, it should return yes. If it returns no, there may be an issue with the Role or RoleBinding configuration.

### ****Real-World Use Case:****

In a production environment, Kubernetes clusters often have various applications that need different levels of access to the cluster's resources. For instance, **service accounts** are often used to ensure **microservices** have the least privilege to interact with the Kubernetes API. This approach minimizes the attack surface in the event of a compromised service account.

A **backend service** that handles sensitive data might only need to access specific **secrets** and **config maps**, so it is assigned a service account with only the necessary roles (e.g., read-only access to secrets).

A **monitoring agent** running in the cluster might need a service account with permissions to access metrics from the entire cluster but without modifying resources. In this case, a **ClusterRole** bound to a service account would be used to provide the necessary permissions.

## ****5. Labs: Managing ClusterRole and ClusterRoleBinding****

### ****ClusterRole and ClusterRoleBinding Overview****

**ClusterRole**: Similar to a **Role**, but the **ClusterRole** is used at the **cluster level** and can apply to all namespaces. It provides permissions to perform operations across the whole cluster, not just within a single namespace.

**ClusterRoleBinding**: It binds a **ClusterRole** to users, groups, or service accounts within a specific namespace or across all namespaces.

### ****Lab 1: Creating and Managing ClusterRoles****

In this lab, we will create a **ClusterRole** that provides get, list, and delete permissions for Pods across the entire cluster, and bind it to a service account using a **ClusterRoleBinding**.

### ****Step 1: Create a ClusterRole****

Let's create a **ClusterRole** named cluster-pod-admin with permissions to manage Pods across the entire cluster.

Create a file clusterrole-pod-admin.yaml:

apiVersion: rbac.authorization.k8s.io/v1

kind: ClusterRole

metadata:

# This is the name of the ClusterRole

name: cluster-pod-admin

rules:

- apiGroups: [""]

resources: ["pods"]

verbs: ["get", "list", "delete"]

This ClusterRole allows users to get, list, and delete Pods cluster-wide.

Apply the ClusterRole:

kubectl apply -f clusterrole-pod-admin.yaml

### ****Step 2: Create a ClusterRoleBinding****

Now, we will create a **ClusterRoleBinding** to bind the cluster-pod-admin ClusterRole to the my-service-account service account in the default namespace.

Create a file clusterrolebinding-pod-admin.yaml:

apiVersion: rbac.authorization.k8s.io/v1

kind: ClusterRoleBinding

metadata:

name: pod-admin-binding

subjects:

- kind: ServiceAccount

name: my-service-account

namespace: default

roleRef:

kind: ClusterRole

name: cluster-pod-admin

apiGroup: rbac.authorization.k8s.io

Apply the ClusterRoleBinding:

kubectl apply -f clusterrolebinding-pod-admin.yaml

### ****Step 3: Verify ClusterRoleBinding****

You can verify that the **ClusterRoleBinding** was created successfully by running:

kubectl get clusterrolebinding pod-admin-binding

### ****Step 4: Verify Access with ClusterRole****

Just like with **Roles**, we can test if the service account has the required access by impersonating the my-service-account service account and running a command. This time, we'll check if it can delete Pods across the cluster.

Run the following command:

kubectl auth can-i delete pods --as=system:serviceaccount:default:my-service-account

If everything is set up correctly, the output should be yes, meaning the service account now has delete permissions on Pods across the entire cluster.

### ****Lab 2: Testing Access with ClusterRole****

We can test access by creating a Pod and then deleting it using the service account’s credentials.

**Create a Pod in the default namespace**:

kubectl run my-test-pod --image=nginx --restart=Never

**Impersonate the service account to delete the Pod**:

kubectl delete pod my-test-pod --as=system:serviceaccount:default:my-service-account

The service account should now be able to delete the Pod since it has been granted the required permissions via the ClusterRole and ClusterRoleBinding.

### ****Real-World Use Case:****

**Admin-Level Access**: ClusterRoles are often used for admin-level permissions. For example, a monitoring tool might need cluster-wide access to read metrics from all nodes and Pods in a Kubernetes cluster. By assigning it a **ClusterRole** with read-only access to all resources, the monitoring tool can access data without the risk of modifying any resources.

**Security Compliance**: Some organizations may have strict security and compliance requirements. For example, only specific teams or users should have the ability to delete or modify Pods across the cluster. By creating a ClusterRole and binding it to the appropriate users or service accounts, you can ensure that only authorized entities can perform high-risk operations, such as deleting critical Pods.

**Access Control for CI/CD Tools**: CI/CD tools often need to interact with the Kubernetes API to deploy applications. A service account bound to a ClusterRole can give the CI/CD tool the necessary permissions to interact with the cluster but restrict its permissions from performing any other destructive actions outside the CI/CD pipeline.

## ****5. Labs: Managing ClusterRole and ClusterRoleBinding****

### ****ClusterRole and ClusterRoleBinding Overview****

**ClusterRole**: Similar to a **Role**, but the **ClusterRole** is used at the **cluster level** and can apply to all namespaces. It provides permissions to perform operations across the whole cluster, not just within a single namespace.

**ClusterRoleBinding**: It binds a **ClusterRole** to users, groups, or service accounts within a specific namespace or across all namespaces.

### ****Lab 1: Creating and Managing ClusterRoles****

In this lab, we will create a **ClusterRole** that provides get, list, and delete permissions for Pods across the entire cluster, and bind it to a service account using a **ClusterRoleBinding**.

### ****Step 1: Create a ClusterRole****

Let's create a **ClusterRole** named cluster-pod-admin with permissions to manage Pods across the entire cluster.

Create a file clusterrole-pod-admin.yaml:

apiVersion: rbac.authorization.k8s.io/v1

kind: ClusterRole

metadata:

# This is the name of the ClusterRole

name: cluster-pod-admin

rules:

- apiGroups: [""]

resources: ["pods"]

verbs: ["get", "list", "delete"]

This ClusterRole allows users to get, list, and delete Pods cluster-wide.

Apply the ClusterRole:

kubectl apply -f clusterrole-pod-admin.yaml

### ****Step 2: Create a ClusterRoleBinding****

Now, we will create a **ClusterRoleBinding** to bind the cluster-pod-admin ClusterRole to the my-service-account service account in the default namespace.

Create a file clusterrolebinding-pod-admin.yaml:

apiVersion: rbac.authorization.k8s.io/v1

kind: ClusterRoleBinding

metadata:

name: pod-admin-binding

subjects:

- kind: ServiceAccount

name: my-service-account

namespace: default

roleRef:

kind: ClusterRole

name: cluster-pod-admin

apiGroup: rbac.authorization.k8s.io

Apply the ClusterRoleBinding:

kubectl apply -f clusterrolebinding-pod-admin.yaml

### ****Step 3: Verify ClusterRoleBinding****

You can verify that the **ClusterRoleBinding** was created successfully by running:

kubectl get clusterrolebinding pod-admin-binding

### ****Step 4: Verify Access with ClusterRole****

Just like with **Roles**, we can test if the service account has the required access by impersonating the my-service-account service account and running a command. This time, we'll check if it can delete Pods across the cluster.

Run the following command:

kubectl auth can-i delete pods --as=system:serviceaccount:default:my-service-account

If everything is set up correctly, the output should be yes, meaning the service account now has delete permissions on Pods across the entire cluster.

### ****Lab 2: Testing Access with ClusterRole****

We can test access by creating a Pod and then deleting it using the service account’s credentials.

**Create a Pod in the default namespace**:

kubectl run my-test-pod --image=nginx --restart=Never

**Impersonate the service account to delete the Pod**:

kubectl delete pod my-test-pod --as=system:serviceaccount:default:my-service-account

The service account should now be able to delete the Pod since it has been granted the required permissions via the ClusterRole and ClusterRoleBinding.

### ****Real-World Use Case:****

**Admin-Level Access**: ClusterRoles are often used for admin-level permissions. For example, a monitoring tool might need cluster-wide access to read metrics from all nodes and Pods in a Kubernetes cluster. By assigning it a **ClusterRole** with read-only access to all resources, the monitoring tool can access data without the risk of modifying any resources.

**Security Compliance**: Some organizations may have strict security and compliance requirements. For example, only specific teams or users should have the ability to delete or modify Pods across the cluster. By creating a ClusterRole and binding it to the appropriate users or service accounts, you can ensure that only authorized entities can perform high-risk operations, such as deleting critical Pods.

**Access Control for CI/CD Tools**: CI/CD tools often need to interact with the Kubernetes API to deploy applications. A service account bound to a ClusterRole can give the CI/CD tool the necessary permissions to interact with the cluster but restrict its permissions from performing any other destructive actions outside the CI/CD pipeline.

## ****6. Labs: Adding Security Context to Pods****

### ****Security Context Overview****

A **SecurityContext** defines privilege and access control settings for a Pod or container. It allows you to define settings such as user and group IDs, whether a container should run as a privileged container, or whether it should allow certain Linux capabilities (e.g., the ability to use ping or raw sockets). Security contexts are essential for ensuring that Pods are running with the least privilege required for operation.

### ****Lab 1: Adding Security Context to a Pod****

In this lab, we will create a Pod with a **SecurityContext** to enable the ping command inside the container. By default, many containers don't allow the ping command due to restrictions in Linux capabilities. We will modify the **SecurityContext** to enable this functionality.

### ****Step 1: Create a Pod with Security Context****

First, let's create a Pod that uses the nginx image and allows the ping command. Create a file pod-with-security-context.yaml:

apiVersion: v1

kind: Pod

metadata:

name: pod-with-security-context

spec:

containers:

- name: nginx

image: nginx

securityContext:

capabilities:

add: ["NET\_PING"]

command: ["/bin/sh", "-c", "sleep 3600"]

In the above YAML, we added the NET\_PING capability to the container using the **SecurityContext**. This capability will allow the container to run the ping command.

### ****Step 2: Apply the Pod Configuration****

Apply the YAML configuration to create the Pod:

kubectl apply -f pod-with-security-context.yaml

### ****Step 3: Verify Pod Creation****

Check if the Pod was created successfully:

kubectl get pod pod-with-security-context

The Pod should be in the Running state.

### ****Step 4: Test the**** ping ****Command Inside the Pod****

Now, let's test if the ping command works inside the container. First, access the container using kubectl exec:

kubectl exec -it pod-with-security-context -- /bin/sh

Once inside the container, run the ping command:

ping 8.8.8.8

If the security context is applied correctly, the ping command should work, showing packets being sent to Google's DNS server.

### ****Step 5: Remove the Security Context and Verify****

Now, to demonstrate the impact of the SecurityContext, let’s modify the Pod to remove the NET\_PING capability and try running the ping command again.

First, edit the Pod configuration:

kubectl edit pod pod-with-security-context

Remove the securityContext section and save the file. The Pod will be automatically restarted.

Next, try running the ping command again inside the Pod:

kubectl exec -it pod-with-security-context -- /bin/sh

ping 8.8.8.8

You should receive a permission error, as the ping command is no longer allowed in the container.

### ****Lab 2: Using Security Context to Run Containers with Specific User ID and Group ID****

Another common use of **SecurityContext** is specifying the user and group IDs (UID and GID) under which the container should run. This is important when the container needs to access files or directories with specific permissions.

### ****Step 1: Create a Pod with UID and GID****

In this example, we will create a Pod that runs the container with a specific UID and GID. Create a file pod-with-uid-gid.yaml:

apiVersion: v1

kind: Pod

metadata:

name: pod-with-uid-gid

spec:

containers:

- name: nginx

image: nginx

securityContext:

runAsUser: 1000

runAsGroup: 2000

command: ["/bin/sh", "-c", "sleep 3600"]

Here, we specify that the container should run as UID 1000 and GID 2000.

### ****Step 2: Apply the Pod Configuration****

Create the Pod by applying the configuration:

kubectl apply -f pod-with-uid-gid.yaml

### ****Step 3: Verify Pod Creation****

Check that the Pod is running:

kubectl get pod pod-with-uid-gid

### ****Step 4: Verify the User and Group ID Inside the Pod****

Now, let's verify that the container is running with the specified UID and GID. Access the container:

kubectl exec -it pod-with-uid-gid -- /bin/sh

Inside the container, check the UID and GID:

id

The output should show that the container is running as UID 1000 and GID 2000.

### ****Real-World Use Cases for SecurityContext:****

**Least Privilege**: Running containers with the least amount of privilege is a fundamental security practice. By using **SecurityContext**, you can ensure that containers only have the necessary privileges to perform their tasks. For instance, disabling certain Linux capabilities (such as NET\_RAW or SYS\_ADMIN) can prevent a compromised container from performing dangerous actions.

**User Access Control**: In multi-tenant environments, you may want to ensure that each application in the cluster runs with a specific UID and GID to ensure it can only access its own resources and not those of other applications.

**Running Privileged Containers**: There are cases when containers may need to run with privileged access (e.g., for managing Kubernetes nodes or accessing certain hardware devices). **SecurityContext** allows you to grant a container the necessary privileges while ensuring it doesn’t have more access than necessary.

## ****7. Labs: Adding Security Context to Pods to Enable Ping****

In this lab, we will specifically focus on adding **SecurityContext** to a Pod to enable the ping command inside a container. This will illustrate how to modify container capabilities using security settings to meet the requirements of certain tasks.

### ****Step 1: Create a Pod with Security Context to Allow Ping****

First, let's create a Pod that allows the ping command to function inside the container by adding the appropriate capability (NET\_PING) in the **SecurityContext**.

Create a file pod-with-ping-security-context.yaml:

apiVersion: v1

kind: Pod

metadata:

name: pod-with-ping-security-context

spec:

containers:

- name: nginx

image: nginx

securityContext:

capabilities:

add: ["NET\_PING"]

command: ["/bin/sh", "-c", "sleep 3600"]

In this YAML, we are adding the NET\_PING capability to the nginx container. This capability enables the container to execute the ping command, which is usually restricted by default in Kubernetes containers.

### ****Step 2: Apply the Pod Configuration****

Run the following command to create the Pod:

kubectl apply -f pod-with-ping-security-context.yaml

### ****Step 3: Verify Pod Creation****

Check the status of the Pod:

kubectl get pod pod-with-ping-security-context

The output should show that the Pod is running:

NAME READY STATUS RESTARTS AGE

pod-with-ping-security-context 1/1 Running 0 1m

### ****Step 4: Access the Container and Run Ping Command****

Now, we will test the ping command inside the container. To do this, we need to execute a command inside the running container.

kubectl exec -it pod-with-ping-security-context -- /bin/sh

Once inside the container, run the ping command to check if it works:

ping 8.8.8.8

If the SecurityContext was configured correctly and the NET\_PING capability was added, the ping command should work, and you should see output like:

PING 8.8.8.8 (8.8.8.8): 56 data bytes

64 bytes from 8.8.8.8: icmp\_seq=0 ttl=54 time=14.6 ms

64 bytes from 8.8.8.8: icmp\_seq=1 ttl=54 time=14.6 ms

### ****Step 5: Remove the Security Context and Verify the Effect****

To observe the effect of not having the correct capabilities, let's modify the Pod definition to remove the NET\_PING capability.

Edit the Pod configuration:

kubectl edit pod pod-with-ping-security-context

Remove the securityContext section, then save and exit the editor. This will force Kubernetes to restart the Pod with the updated configuration.

### ****Step 6: Verify Ping Command After Modification****

Once the Pod has been restarted, try executing the ping command again:

kubectl exec -it pod-with-ping-security-context -- /bin/sh

ping 8.8.8.8

You should now see an error similar to:

ping: socket: Operation not permitted

This error occurs because the container no longer has the NET\_PING capability, which is required for running the ping command.

### ****Real-World Use Case for SecurityContext (Ping)****

**Network Troubleshooting**: Containers in a Kubernetes environment often need to test network connectivity. The ability to execute the ping command inside a container can help with debugging network-related issues between Pods, services, or external networks.

**System Administration**: Containers running in Kubernetes may need specific privileges to interact with the network stack. For instance, running network-related operations such as ping, traceroute, or creating raw sockets can be essential for containerized applications that perform networking tasks (e.g., DNS testing, network monitoring, or VPN services).

**Security Considerations**: The ping capability (NET\_PING) is added to ensure that the application inside the container has the necessary privileges without granting full privileged mode. It's a more secure approach as it only enables the specific capabilities required for the task rather than granting the container unrestricted access.

Let me know when you're ready to move to the next topic!

## ****8. Labs: Managing Roles and RoleBindings****

In this lab, we will focus on creating and managing **Roles**, **ClusterRoles**, **RoleBindings**, and **ClusterRoleBindings** in Kubernetes. These resources help manage permissions and access control at different levels in the cluster, such as within specific namespaces or at the cluster-wide level.

### ****Step 1: Creating a Role****

A **Role** defines a set of permissions within a specific namespace. It allows you to specify which actions (such as creating, deleting, or modifying resources) are allowed on which resources.

Create a file role-example.yaml to define a **Role** in the default namespace that grants permission to read Pods.

apiVersion: rbac.authorization.k8s.io/v1

kind: Role

metadata:

# The Role will be in the "default" namespace

namespace: default

name: pod-reader

rules:

- apiGroups: [""]

# The resource we are allowing access to

resources: ["pods"]

verbs: ["get", "list"]

This Role allows users to **get** and **list** Pods in the default namespace.

### ****Step 2: Applying the Role****

Apply the Role configuration to create it in the cluster:

kubectl apply -f role-example.yaml

### ****Step 3: Creating a RoleBinding****

A **RoleBinding** grants the permissions defined in a Role to a specific user or group of users within a namespace. Create a file rolebinding-example.yaml to bind the pod-reader Role to a user called reader.

apiVersion: rbac.authorization.k8s.io/v1

kind: RoleBinding

metadata:

name: read-pods-binding

namespace: default

subjects:

- kind: User

name: reader

apiGroup: rbac.authorization.k8s.io

roleRef:

kind: Role

name: pod-reader

apiGroup: rbac.authorization.k8s.io

In this file:

**subjects**: Defines the users (or service accounts) to which the role will be granted.

**roleRef**: Specifies the Role that the binding applies to.

### ****Step 4: Applying the RoleBinding****

Create the RoleBinding by applying the configuration:

kubectl apply -f rolebinding-example.yaml

### ****Step 5: Verifying the Role and RoleBinding****

You can verify that the Role and RoleBinding were created successfully by running:

kubectl get role -n default

kubectl get rolebinding -n default

### ****Step 6: Testing Role and RoleBinding Permissions****

Now, let's test if the user reader has the expected permissions. First, simulate the behavior by assuming the reader user (this assumes you've set up a Kubernetes RBAC system with a user reader). You can also use kubectl with impersonation:

kubectl auth can-i get pods --as=reader -n default

This command checks if the reader user can **get** Pods in the default namespace. If everything is configured correctly, you should see:

yes

### ****Step 7: Cleaning Up Resources****

To clean up the resources after testing, run:

kubectl delete -f role-example.yaml

kubectl delete -f rolebinding-example.yaml

### ****Real-World Use Cases for Role and RoleBinding****

**Namespace-Based Access Control**: In a multi-tenant Kubernetes environment, you may want to restrict users to specific namespaces. By creating Roles and RoleBindings, you can limit users’ actions to certain namespaces, ensuring they cannot access or modify resources in other namespaces.

**Least Privilege Access**: Rather than giving users broad cluster-wide permissions, you can create granular RoleBindings that only allow them to perform certain actions on specific resources. For example, a developer may only need read access to Pods and not be able to delete or modify them.

**Service Accounts**: You may also create RoleBindings for **service accounts**, which are often used by applications running in Pods to interact with the Kubernetes API. This approach allows for programmatic access to Kubernetes resources while maintaining strict control over what actions the application can perform.

## ****1. Understanding OS Upgrade****

### ****Overview****

Upgrading the operating system (OS) on nodes is essential for maintaining system security, stability, and performance. OS upgrades are often required to support new Kubernetes versions, software dependencies, and security patches.

In Kubernetes, OS upgrades are critical because the Kubelet, kubeadm, and other Kubernetes components are tightly coupled with the underlying OS. However, OS upgrades should be performed carefully to avoid breaking the Kubernetes cluster or its workloads.

### ****Steps Involved in OS Upgrade:****

**Preparation:**

**Backup Configuration**: Before upgrading the OS, ensure that your Kubernetes configurations, like kubeadm configurations, kubelet configurations, and any other configuration files, are backed up.

**Plan the Upgrade**: Decide whether to upgrade all nodes at once or perform rolling upgrades. A rolling upgrade ensures that at least part of the cluster is functional at all times, minimizing downtime.

**Upgrade Procedure:**

**Upgrade the OS**: Upgrade the OS using the respective package manager (e.g., apt on Ubuntu or yum on CentOS). For example:

sudo apt-get update

sudo apt-get upgrade -y

**Ensure Kubernetes Components are Compatible**: After upgrading the OS, check that the Kubernetes components (like kubeadm, kubelet, and kubectl) are still compatible with the OS version.

**Rebooting Nodes**:

Reboot the node if necessary to apply OS updates (especially kernel upgrades).

Ensure the node comes back up correctly and the Kubelet is running.

**Verify Kubernetes Cluster Health**: After upgrading the OS and rebooting the node, run the following commands to ensure the cluster is healthy:

kubectl get nodes

kubectl get pods --all-namespaces

**Upgrade Other Nodes**: If you're performing a rolling upgrade, proceed to upgrade other nodes, one at a time. Always ensure the cluster has at least one healthy node to handle traffic.

### ****Important Considerations****:

**Node Taints**: During the upgrade process, you may want to use taints to prevent new workloads from being scheduled on the upgrading node.

**Pod Disruption Budgets (PDBs)**: Use PDBs to ensure that the number of unavailable replicas of critical services does not exceed the acceptable threshold.

### ****Lab: OS Upgrade in Kubernetes Cluster****

#### ****Objective:****

Perform an OS upgrade on a Kubernetes node (worker or master node) and verify that the upgrade doesn't break the cluster functionality.

#### ****Prerequisites:****

A running Kubernetes cluster with at least 1 master node and 1 worker node.

SSH access to the nodes in the cluster.

#### ****Step-by-Step Process:****

**Check Kubernetes Version**: On the master node, check the Kubernetes version to ensure it's working before the upgrade.

kubectl version

**Backup Kubernetes Configuration Files**: Backup important configuration files (e.g., Kubelet, Kubeadm):

sudo cp /etc/kubernetes/kubelet.conf /etc/kubernetes/kubelet.conf.bak

sudo cp /etc/kubernetes/kubeadm.conf /etc/kubernetes/kubeadm.conf.bak

**Upgrade the OS**:

For Ubuntu, run the following commands:

sudo apt-get update

sudo apt-get upgrade -y

sudo apt-get dist-upgrade -y

For CentOS, use:

sudo yum update -y

**Reboot the Node**: After the upgrade is completed, reboot the node to apply changes:

sudo reboot

**Verify Kubernetes Cluster Health**: Once the node is back online, verify that the node and cluster are still healthy:

kubectl get nodes

kubectl get pods --all-namespaces

**Upgrade Other Nodes (if needed)**: Repeat the same process for the other worker nodes, performing rolling upgrades to minimize downtime.

### ****Key Points to Verify Post-Upgrade:****

**Node Status**: Ensure the node comes back in a Ready state.

**Pod Scheduling**: Ensure new pods can be scheduled to the upgraded node.

**Pod Health**: Verify that existing pods are still running and healthy.

## ****2. Static Pod****

### ****Overview****

Static Pods are managed directly by the kubelet on each node rather than by the Kubernetes control plane (master node). They are primarily used for critical components that need to run on specific nodes, such as the Kubernetes control plane components (like the API server, controller manager, scheduler, etc.).

Unlike regular Pods, which are managed by the Kubernetes API server and scheduled by the control plane, static Pods are scheduled by the kubelet based on the configuration files located on each node.

### ****Key Characteristics of Static Pods:****

**Node-Specific**: Static Pods are defined in configuration files placed on specific nodes. These files are typically located in /etc/kubernetes/manifests/ on the node.

**No Controller Management**: Static Pods do not have a controller like ReplicaSet, Deployment, or StatefulSet managing them. They are managed directly by the kubelet.

**Self-Healing**: While Static Pods are not managed by controllers, if a Static Pod is deleted or crashes, the kubelet will attempt to restart it based on the manifest file.

**No API Server Interaction**: Unlike regular Pods, Static Pods do not appear in the Kubernetes API server by default. However, they can be listed with the kubectl get pods --all-namespaces command if they are scheduled by the kubelet.

### ****Use Cases for Static Pods****:

**Critical Components**: Static Pods are often used for components like the kube-apiserver, kube-controller-manager, kube-scheduler, and etcd on control plane nodes.

**System Daemon Services**: For system-level services that need to run on specific nodes, such as monitoring agents or logging agents.

**Edge Computing**: In environments where you need to ensure specific services are always running on certain nodes without the overhead of controllers.

### ****How Static Pods Work:****

**Manifest File**: Static Pods are defined in a manifest file (YAML format) on the node where they should run. The kubelet will automatically read these manifest files and create the pods.

**Pod Scheduling**: The kubelet is responsible for scheduling Static Pods on nodes. Unlike other Kubernetes components, the control plane does not schedule Static Pods.

**Pod Monitoring**: The kubelet continuously monitors these Pods and attempts to restart them if they fail or get deleted.

### ****Static Pod Limitations****:

Static Pods are not managed by controllers, so you don’t get features like scaling, rolling updates, or declarative state management.

They are only visible on the node where they are defined unless explicitly configured to be shown.

### ****Lab: Deploying a Static Pod****

#### ****Objective:****

Deploy a Static Pod on a Kubernetes node and verify that it runs and can be accessed.

#### ****Prerequisites:****

A running Kubernetes cluster with at least one node (worker or master).

Access to the node via SSH.

#### ****Step-by-Step Process:****

**Create a Manifest File for the Static Pod:**

On the node where you want to deploy the Static Pod (e.g., a worker node), create a YAML file that defines the Static Pod.

Create a new file in /etc/kubernetes/manifests/ with the following content (as an example, we’ll use a simple Nginx container):

apiVersion: v1

kind: Pod

metadata:

name: nginx-static-pod

spec:

containers:

- name: nginx

image: nginx:latest

ports:

- containerPort: 80

**Verify the Static Pod is Running:**

After creating the manifest file, the kubelet on the node will automatically detect it and create the Static Pod. To verify this:

kubectl get pods --all-namespaces

You should see the Static Pod listed under the default namespace (even though it's not managed by the control plane):

NAME READY STATUS RESTARTS AGE

nginx-static-pod 1/1 Running 0 5m

**Access the Static Pod:**

The Static Pod is running the Nginx container, and you can access it on the node's IP address.

Find the node's IP address:

hostname -I

Now, try to access the Nginx service from a web browser or using curl:

curl http://<node-ip>:80

**Modify or Delete the Static Pod:**

To modify the Static Pod (e.g., change the container image), you can edit the manifest file directly on the node, and the kubelet will update the Pod accordingly.

To delete the Static Pod, simply delete the manifest file:

sudo rm /etc/kubernetes/manifests/nginx-static-pod.yaml

**Verify Pod Deletion and Restart:**

After deleting the manifest file, the kubelet will automatically detect the change and restart the Static Pod. You should see the pod come back up after a short delay.

### ****Key Points to Verify:****

**Pod Status**: Ensure that the Static Pod is running and healthy.

**Pod Accessibility**: Verify that the container inside the Static Pod is accessible via the node’s IP.

**Pod Restart**: After modifying the manifest file, check if the pod restarts successfully.

## ****3. Cron Job****

### ****Overview****

A **CronJob** in Kubernetes is a controller that allows you to run scheduled jobs in a manner similar to the traditional cron job in Linux. It helps you automate periodic tasks such as backups, cleanup operations, sending reports, or any other recurring jobs.

CronJobs are useful for workloads that need to be executed on a regular schedule, such as:

Backup jobs (e.g., database backup)

Cleanup tasks (e.g., cleaning old logs)

Batch processing (e.g., processing files or data)

A CronJob runs based on a cron expression, and it can schedule the execution of Pods at a specific time or interval. The cron expression follows the format:

<minute> <hour> <day-of-month> <month> <day-of-week>

Example:

"0 12 \* \* \*" - This means the job will run at 12:00 PM (noon) every day.

### ****Key Features of CronJobs****:

**Cron Expression**: The schedule is defined using a cron expression, which determines when the job should run.

**Concurrency Policy**: Defines how multiple instances of the same CronJob should behave if a previous job is still running. The possible values are:

Allow: Allows multiple instances to run concurrently.

Forbid: Ensures that only one instance is running at a time.

Replace: Replaces the previous job if it’s still running when the new one starts.

**History Limits**: CronJobs keep a history of job executions, which can be limited to a set number of past jobs (successful or failed) to avoid using excessive resources.

### ****Use Cases for CronJobs****:

**Automated Backups**: CronJobs are often used for scheduling database backups or periodic configuration snapshots.

**Batch Jobs**: Running batch processing tasks such as data transformations or periodic analysis.

**System Maintenance**: Running cleanup tasks such as log file removal or deleting old data files.

### ****Lab: Deploying a CronJob****

#### ****Objective:****

Deploy a CronJob in Kubernetes that runs a periodic task (e.g., a simple echo command) and verify its execution.

#### ****Prerequisites:****

A running Kubernetes cluster with at least one node.

kubectl access to the cluster.

#### ****Step-by-Step Process:****

**Create a CronJob YAML Manifest:** Create a YAML file to define the CronJob. For example, let’s create a CronJob that runs an echo command every minute.

Create a file named cronjob-example.yaml:

apiVersion: batch/v1

kind: CronJob

metadata:

name: cronjob-example

spec:

schedule: "\*/1 \* \* \* \*" # Cron expression to run every minute

jobTemplate:

spec:

template:

spec:

containers:

- name: echo

image: busybox

command:

- /bin/sh

- -c

- "echo 'Hello, Kubernetes CronJob!'"

restartPolicy: OnFailure

**Deploy the CronJob:** Apply the manifest to create the CronJob:

kubectl apply -f cronjob-example.yaml

**Check the CronJob Status:** To see the CronJob’s status and verify if it is scheduled, use:

kubectl get cronjobs

You should see the CronJob listed in the output with details such as LAST SCHEDULE and SCHEDULED.

**Check for the Created Jobs:** CronJobs will create jobs according to their schedule. To view the jobs created by the CronJob, use:

kubectl get jobs

**Check the Logs of the CronJob’s Pod:** Each job runs as a Pod. To view the logs of the pod created by the CronJob, first get the pod name:

kubectl get pods --selector=job-name=cronjob-example-<job-id>

Replace <job-id> with the actual job ID found in the previous command. Then, view the logs:

kubectl logs <pod-name>

You should see the following output:

Hello, Kubernetes CronJob!

**Delete the CronJob:** If you want to stop the CronJob from running, you can delete it:

kubectl delete -f cronjob-example.yaml

### ****Key Points to Verify:****

**CronJob Execution**: Ensure that the CronJob is running on the schedule and creating jobs as expected.

**Logs**: Verify that the logs from the CronJob’s Pods contain the expected output.

**CronJob Deletion**: Ensure that deleting the CronJob stops further job executions.

## ****4. ETCD Backup****

### ****Overview****

ETCD is a distributed key-value store that Kubernetes uses to store all of its cluster state, including configuration data, secrets, and the state of various objects (pods, services, etc.). Since ETCD is a critical part of Kubernetes, regular backups are essential to prevent data loss in case of failures or disasters. ETCD backups can help restore the cluster to a previous state in the event of data corruption or accidental deletion of resources.

Kubernetes provides various methods for backing up ETCD, which include:

**ETCD Snapshots**: A snapshot of the ETCD database can be taken to back up the current state.

**ETCD Cluster Backup**: In the case of a multi-node ETCD setup, backups can be taken for the entire ETCD cluster.

### ****Why Back Up ETCD?****

**Disaster Recovery**: If your cluster’s control plane fails or if there’s data corruption, a backup of ETCD allows you to restore the cluster state.

**Cluster Migration**: When migrating Kubernetes clusters or upgrading to a new version, having ETCD backups ensures you can revert changes if necessary.

**Data Safety**: ETCD contains sensitive data like secrets, configurations, and resource definitions, making it essential to protect it.

### ****How to Back Up ETCD?****

Kubernetes provides the etcdctl command-line tool to interact with ETCD, and it can be used to take manual snapshots of the ETCD data.

### ****ETCD Backup Process:****

**Snapshotting ETCD**: ETCD backups are typically taken using snapshots. These snapshots are consistent and provide a point-in-time backup of the ETCD database.

**Backup Location**: The snapshot is stored in a specified directory or object storage (e.g., Amazon S3, Google Cloud Storage).

**Restore from Snapshot**: In case of failure, the ETCD data can be restored using the snapshot.

### ****Steps for Backing Up ETCD:****

**Backup using etcdctl**: To take a snapshot of the ETCD cluster, use the etcdctl snapshot save command.

### ****Lab: ETCD Backup****

#### ****Objective:****

Perform an ETCD backup on your Kubernetes cluster to ensure data safety.

#### ****Prerequisites:****

A running Kubernetes cluster with at least one master node.

etcdctl CLI tool installed on the master node.

#### ****Step-by-Step Process:****

**Set Up Environment Variables for ETCD:** The ETCD API and other credentials need to be available for the backup process. On the master node, set up the environment variables to authenticate with the ETCD API.

Example:

export ETCDCTL\_API=3

export ETCD\_ENDPOINTS="https://<etcd-endpoint>:2379"

export ETCD\_CERT\_FILE="/etc/kubernetes/pki/etcd/server.crt"

export ETCD\_KEY\_FILE="/etc/kubernetes/pki/etcd/server.key"

export ETCD\_CA\_FILE="/etc/kubernetes/pki/etcd/ca.crt"

Replace <etcd-endpoint> with the actual IP address or hostname of your ETCD server.

**Take an ETCD Snapshot:** Use the etcdctl command to take a snapshot of your ETCD data:

etcdctl snapshot save /tmp/etcd-backup.db

This command will create a snapshot and save it as /tmp/etcd-backup.db. You can specify any directory for the backup.

**Verify the Snapshot:** Check the snapshot directory to ensure the snapshot file is created:

ls /tmp/etcd-backup.db

You should see the snapshot file created with the timestamp in the filename.

**Backup to Remote Storage (Optional):** If you want to back up the snapshot to remote storage (e.g., AWS S3), you can use the aws s3 cp command:

aws s3 cp /tmp/etcd-backup.db s3://your-backup-bucket/etcd-backup.db

**Restore from ETCD Snapshot (If Needed):** In case of failure, you can restore the cluster from the snapshot by running:

etcdctl snapshot restore /tmp/etcd-backup.db --data-dir /var/lib/etcd

After restoring, restart the ETCD service:

systemctl restart etcd

**Verify ETCD Restoration:** After restoring the snapshot, check the ETCD cluster's health to ensure it is functioning correctly:

etcdctl endpoint health

You should see a healthy status if the ETCD restoration was successful.

**Delete Backup (Optional):** If the backup is no longer needed, remove the snapshot file:

rm /tmp/etcd-backup.db

### ****Key Points to Verify:****

**Snapshot Creation**: Ensure the snapshot is created successfully.

**Snapshot Integrity**: Verify the snapshot file is available and can be uploaded to remote storage if required.

**Cluster Health**: After restoration, check the health of the ETCD cluster to confirm that it is functioning properly.

## ****1. Understand How to Monitor Application and Cluster Components****

### ****Overview****

Monitoring is critical for ensuring the health and performance of both Kubernetes clusters and the applications running on them. Kubernetes provides a robust ecosystem for monitoring that includes:

**Monitoring Cluster Components**: This involves keeping track of the health, availability, and resource usage of core Kubernetes components such as the API server, controllers, scheduler, and etcd.

**Monitoring Applications**: Applications running on Kubernetes are also critical to monitor, as they may have issues like high response time, resource consumption, or crashes.

Effective monitoring involves setting up infrastructure to collect metrics, logs, and events from the cluster and applications, followed by analyzing them to make informed decisions on scaling, troubleshooting, and optimizing your Kubernetes environment.

### ****Why Monitor Kubernetes Cluster and Applications?****

**Performance**: Monitoring provides insights into resource usage (CPU, memory, etc.) to help identify bottlenecks or inefficient resource allocation.

**Troubleshooting**: With monitoring, you can quickly identify issues like failed pods, crashes, or issues in communication between cluster components.

**Capacity Planning**: It helps in understanding how resources are being utilized and forecasting future resource needs.

**Security and Compliance**: Monitoring logs and metrics helps in identifying security incidents or unauthorized access attempts in the cluster or application.

### ****Key Monitoring Areas:****

**Kubernetes Cluster Monitoring**:

**API Server**: Track the availability and performance of the Kubernetes API server.

**Scheduler**: Monitor scheduler latency and performance.

**Controllers and ETCD**: Monitor the health of controllers and etcd.

**Node Health**: Track the state and resource usage of nodes.

**Kubelet**: Monitor the status of the Kubelet and its interaction with the nodes.

**Application Monitoring**:

**Pod Health**: Monitoring the status of pods, including whether they are running, crashing, or not starting.

**Container Logs**: Collect logs from containers to identify application errors, warnings, and critical issues.

**Application Metrics**: Track application-specific metrics (e.g., response times, error rates).

### ****Kubernetes Monitoring Tools****:

There are multiple tools available for monitoring a Kubernetes cluster and applications, including:

**Prometheus**: A powerful open-source tool that is used to collect and store metrics from various sources, including Kubernetes.

**Grafana**: A visualization tool that integrates well with Prometheus, providing dashboards for real-time monitoring.

#### ****Common Monitoring Metrics for Kubernetes****:

**Node metrics**: CPU, memory, disk usage, and network statistics.

**Pod metrics**: Container resource usage (CPU and memory).

**API Server latency**: Latency of requests to the Kubernetes API.

**Scheduler performance**: Time taken to schedule pods.

**Pod status**: Whether pods are running, terminated, or pending.

### ****Lab: Understanding How to Read Application & Cluster Component Logs****

#### ****Objective:****

Learn how to read logs from Kubernetes components and applications to monitor their behavior, identify issues, and troubleshoot.

#### ****Prerequisites:****

A running Kubernetes cluster (Master and 2 Worker nodes).

kubectl configured for accessing your cluster.

#### ****Step-by-Step Process:****

**Accessing Cluster Component Logs:**

Kubernetes stores logs for different cluster components. These logs can be accessed using kubectl commands.

**Kubelet Logs**: To view the logs from the Kubelet (the agent responsible for managing pods on each node):

kubectl logs -n kube-system <node-name> -l k8s-app=kubelet

Replace <node-name> with the name of the node you're interested in.

**API Server Logs**: To view the logs of the API Server, which is the central control point for all interactions in Kubernetes:

kubectl logs -n kube-system <pod-name> -l component=kube-apiserver

Replace <pod-name> with the name of the pod running the API server.

**Reading Application Logs:** To read logs from your application, you first need to find the pod running the application:

kubectl get pods --all-namespaces

Once you know the pod name, use the following command to get the logs:

kubectl logs <pod-name>

**Viewing Logs from Previous Pods:** If your pod has crashed and restarted, Kubernetes stores logs from the previous instance of the pod:

kubectl logs <pod-name> --previous

**Filter Logs by Time or Keyword:** You can filter logs using grep to find errors or warnings:

kubectl logs <pod-name> | grep "error"

**View Logs for a Specific Namespace:** If your application is running in a specific namespace, you can specify that:

kubectl logs -n <namespace> <pod-name>

**Understanding Log Severity Levels:** Logs typically contain different severity levels (e.g., INFO, ERROR, WARN, DEBUG). Understanding these levels helps prioritize issues:

**INFO**: General information about the pod’s operations.

**ERROR**: Indicates a failure or an issue that requires attention.

**WARN**: Potential issues or warnings that don’t necessarily indicate failure.

**Exploring Node Logs:** You can also view the logs for the underlying node system. For example, on Ubuntu, you might access the journalctl logs directly:

journalctl -u kubelet

**Verify Cluster and Application Status:** After reading logs, ensure that your application or the cluster components are functioning as expected:

kubectl get pods

kubectl get nodes

## ****2. Deploying Prometheus & Grafana to Monitor Kubernetes Cluster****

### ****Overview****

Prometheus and Grafana are widely used in the Kubernetes ecosystem for monitoring and visualization. Prometheus is used to collect, store, and query metrics, while Grafana is used for creating dashboards and visualizing these metrics.

In this section, we will deploy Prometheus and Grafana in a Kubernetes cluster, configure them to monitor various Kubernetes metrics, and visualize those metrics in Grafana dashboards.

### ****Why Use Prometheus and Grafana?****

**Prometheus**:

Designed for monitoring and alerting.

Collects and stores time-series data, making it ideal for capturing Kubernetes metrics.

Includes a query language (PromQL) for querying the metrics.

**Grafana**:

Provides rich, interactive dashboards that allow for real-time visualizations.

Integrates with Prometheus to provide insights from the collected metrics.

### ****Setting Up Prometheus and Grafana for Monitoring****

**Prometheus Setup**: Prometheus collects metrics from various Kubernetes components, including the kubelet, API server, and controller manager. It scrapes metrics from exposed HTTP endpoints at regular intervals.

**Grafana Setup**: After Prometheus is set up, Grafana is used to visualize the data collected by Prometheus in a user-friendly dashboard.

### ****Monitoring Key Metrics in Kubernetes****:

**Node Metrics**: CPU usage, memory usage, disk usage, network stats.

**Pod Metrics**: CPU and memory usage by container in a pod.

**Kubernetes Components**: API server, scheduler, kubelet, etcd health.

**Custom Application Metrics**: Metrics specific to your applications.

### ****Lab: Deploying Prometheus & Grafana to Monitor Kubernetes Cluster****

#### ****Objective****:

Set up Prometheus and Grafana in your Kubernetes cluster to monitor key metrics such as node health, pod usage, and Kubernetes component performance.

#### ****Prerequisites****:

A running Kubernetes cluster (1 Master and 2 Worker nodes).

kubectl configured to interact with the cluster.

Helm installed for easy deployment.

#### ****Step-by-Step Process:****

**Install Helm** (If not already installed): Helm is a package manager for Kubernetes that simplifies deploying applications like Prometheus and Grafana.

First, install Helm:

curl https://get.helm.sh/helm-v3.7.0-linux-amd64.tar.gz -o helm-v3.7.0-linux-amd64.tar.gz

tar -zxvf helm-v3.7.0-linux-amd64.tar.gz

sudo mv linux-amd64/helm /usr/local/bin/helm

**Add Prometheus and Grafana Helm Charts**: Add the Prometheus and Grafana repositories to Helm:

helm repo add prometheus-community https://prometheus-community.github.io/helm-charts

helm repo add grafana https://grafana.github.io/helm-charts

helm repo update

**Deploy Prometheus**: Deploy Prometheus using the Helm chart:

helm install prometheus prometheus-community/kube-prometheus-stack --namespace monitoring --create-namespace

This command installs Prometheus along with the necessary components (Prometheus operator, Grafana, etc.) into the monitoring namespace.

**Deploy Grafana**: If Grafana is not included in the above deployment, you can install it separately:

helm install grafana grafana/grafana --namespace monitoring

**Verify Prometheus and Grafana Deployment**: Check the status of the Prometheus and Grafana pods:

kubectl get pods -n monitoring

Ensure that all pods (prometheus, grafana) are running.

**Access Grafana**: To access Grafana, create a port-forward from your local machine to the Grafana service:

kubectl port-forward svc/grafana 3000:80 -n monitoring

Open a browser and go to http://localhost:3000. The default login credentials are:

Username: admin

Password: admin (or check the Helm release notes for the generated password).

**Configure Prometheus as a Data Source in Grafana**: Once logged into Grafana:

Click on the gear icon (Configuration) in the left menu.

Select **Data Sources** and then **Prometheus**.

In the **HTTP** section, set the URL to the Prometheus service: http://prometheus-server.monitoring.svc.cluster.local.

Click **Save & Test** to verify the connection.

**Import Prebuilt Dashboards**: Grafana has prebuilt dashboards for Kubernetes monitoring. To import a dashboard:

In Grafana, click the **+** icon on the left panel.

Select **Import**.

Enter the dashboard ID (e.g., **315** for the Kubernetes cluster monitoring dashboard).

Click **Load** and then **Import** to add the dashboard.

**Monitor Kubernetes Metrics**: Now, you should be able to view various metrics in Grafana, such as:

Node CPU and memory usage.

Pod resource usage.

Kubelet health and API server latency.

**Create Custom Alerts (Optional)**: You can set up alerts in Prometheus to notify you if certain thresholds are exceeded (e.g., CPU usage > 80% or pod status is not running):

Go to **Alerting** in the Prometheus web UI.

Create a new alert rule based on the desired metrics (e.g., avg(container\_cpu\_usage\_seconds\_total) by (pod)).

### ****Key Points to Verify:****

**Prometheus Deployment**: Ensure that Prometheus is collecting data by verifying the pod and service statuses.

**Grafana Access**: Verify that you can access the Grafana UI and that the Prometheus data source is correctly configured.

**Dashboards**: Ensure that the Kubernetes monitoring dashboards are displaying the metrics as expected.

**Alerts**: Check the alerting configuration if set up, to receive notifications on critical metrics.

## ****1. Understand Basics of Kubernetes Networking****

### ****Overview:****

Networking is a fundamental concept in Kubernetes as it enables communication between various components within the cluster, such as Pods, Services, and external resources. Understanding Kubernetes networking is crucial for deploying, troubleshooting, and scaling applications.

In Kubernetes, networking needs to address several key challenges, including:

Communication between containers (Pods) across nodes.

Exposing services to external clients.

Managing internal DNS resolution.

Ensuring high availability and load balancing of services.

### ****Why Kubernetes Networking is Important?****

**Pod-to-Pod Communication**: Kubernetes provides a consistent networking model for communication between Pods, regardless of which node they are scheduled on.

**Service Discovery and Load Balancing**: Kubernetes provides built-in DNS resolution and load balancing for Services.

**External Access**: Kubernetes supports various ways to expose internal services to the outside world.

**Network Policies**: Kubernetes allows you to define network policies to restrict or control traffic between Pods.

### ****Key Kubernetes Networking Components****:

**Pod Networking**: Pods in Kubernetes are the smallest deployable units and are assigned unique IP addresses. Each Pod can communicate with other Pods directly using their IP addresses.

**Services**: Services are abstractions that provide access to a set of Pods. Kubernetes provides different types of services, including ClusterIP, NodePort, and LoadBalancer.

**Network Policies**: Network Policies in Kubernetes define how Pods communicate with each other and with external resources. These policies are implemented using Kubernetes' built-in tools or third-party solutions.

**DNS Resolution**: Kubernetes provides a DNS service (CoreDNS) for Service discovery. Services can be reached using their names, which Kubernetes resolves into IP addresses.

**Ingress**: Ingress is a Kubernetes resource that manages external HTTP(S) access to services within the cluster, typically through an Ingress Controller.

### ****Lab: Understanding Basic Kubernetes Networking****

#### ****Objective****:

Understand how Pods communicate with each other and with services, and explore how networking is managed in Kubernetes.

#### ****Prerequisites****:

A running Kubernetes cluster (1 Master and 2 Worker nodes).

kubectl configured to interact with the cluster.

#### ****Step-by-Step Process****:

**Deploying Two Pods for Networking Test**: First, let's deploy two Pods in the same namespace to check Pod-to-Pod communication.

Create a pod-1.yaml file with the following content:

apiVersion: v1

kind: Pod

metadata:

name: pod-1

spec:

containers:

- name: nginx

image: nginx

Create another pod-2.yaml file with the same content:

apiVersion: v1

kind: Pod

metadata:

name: pod-2

spec:

containers:

- name: nginx

image: nginx

Deploy both Pods:

kubectl apply -f pod-1.yaml

kubectl apply -f pod-2.yaml

**Testing Pod-to-Pod Communication**: Once both Pods are running, access the shell of one Pod:

kubectl exec -it pod-1 -- /bin/bash

Inside the Pod, use curl to test communication with the other Pod:

curl pod-2

You should receive a response from pod-2, confirming that Pods can communicate with each other within the same network.

**Creating a Service to Expose Pods**: Now, let's expose the Pods using a Kubernetes Service. Create a service.yaml file to expose the Pods:

apiVersion: v1

kind: Service

metadata:

name: nginx-service

spec:

selector:

app: nginx

ports:

- protocol: TCP

port: 80

targetPort: 80

type: ClusterIP

Apply the service configuration:

kubectl apply -f service.yaml

**Testing Access to the Service**: Now that the service is created, you can test accessing the service by using the service name (nginx-service) from one of the Pods:

kubectl exec -it pod-1 -- curl nginx-service

You should see the response from one of the Nginx Pods, confirming that the service is properly routing traffic to the Pods.

**Verify DNS Resolution**: Kubernetes provides automatic DNS resolution for services. To verify this, run the following command from within one of the Pods:

kubectl exec -it pod-1 -- nslookup nginx-service

This should return the IP address of the nginx-service.

### ****Key Takeaways****:

Kubernetes automatically provides networking capabilities for Pods, allowing them to communicate with each other without needing to know about each other's physical location (i.e., node).

Services in Kubernetes provide stable network access to dynamic Pods.

DNS resolution is built-in, making it easy to refer to services by name within the cluster.

This concludes the topic on **Basics of Kubernetes Networking**. Let me know when you're ready to proceed to the next topic by saying "next."

## ****2. Understand CNI Overview****

### ****Overview:****

The Container Network Interface (CNI) is a specification and a set of libraries for configuring network interfaces in Linux containers. It is a key component in Kubernetes networking, providing the plugins that Kubernetes uses to manage networking between Pods.

In Kubernetes, CNI is used to implement the networking requirements for Pods, including how they get assigned IP addresses and how they communicate with each other, the host machine, and external networks.

### ****Why CNI is Important in Kubernetes?****

**Customizability**: CNI allows Kubernetes to integrate with various networking solutions (plugins) based on different use cases. This means users can choose from a wide array of network policies, IP management strategies, and routing rules.

**Extensibility**: The CNI specification is flexible and extensible, allowing Kubernetes to adopt and integrate newer networking models as they emerge.

**Consistency**: By abstracting network configuration, CNI provides a consistent approach to managing networking across different container runtimes and orchestration platforms.

### ****How Kubernetes Uses CNI?****

Kubernetes uses CNI plugins to set up networking for Pods during the pod lifecycle:

**Pod IP assignment**: When a Pod is created, the CNI plugin assigns it an IP address.

**Network Policy enforcement**: CNI plugins can enforce network policies, allowing or blocking traffic between Pods, based on predefined rules.

**Connectivity to external networks**: CNI handles the integration of Pods with external networks, facilitating outbound traffic.

### ****Common CNI Plugins****:

**Calico**: A popular CNI plugin that provides network security and routing features, often used in large-scale Kubernetes deployments.

**Flannel**: A simple and easy-to-use CNI plugin that provides overlay networking, ideal for small to medium-sized clusters.

**Weave**: A CNI plugin that offers networking with encryption for Pods, with options for network isolation.

**Cilium**: A CNI plugin designed for large, dynamic Kubernetes environments with advanced networking features like API-aware load balancing.

### ****Lab: Exploring CNI in Kubernetes****

#### ****Objective****:

Understand how Kubernetes uses CNI to manage networking in the cluster and explore how different CNI plugins can be used.

#### ****Prerequisites****:

A running Kubernetes cluster (1 Master, 2 Worker nodes).

kubectl configured to interact with the cluster.

The calico CNI plugin (or any other CNI plugin) installed.

#### ****Step-by-Step Process****:

**Checking the Default CNI Plugin**: First, let’s check which CNI plugin is installed in your Kubernetes cluster. By default, Kubernetes uses a CNI plugin, which is often configured during the installation of the cluster.

To check the CNI plugin:

kubectl get pods --all-namespaces -l k8s-app=kube-dns

This should show the Pods related to CoreDNS (if your cluster is running a DNS service). The underlying CNI plugin for the network can be verified through the Node's kubelet configuration or using the following:

kubectl describe pod <pod\_name> -n kube-system

**Installing a New CNI Plugin (Calico Example)**: If Calico is not already installed, you can install it using the following steps. Calico is an advanced CNI plugin that supports network policies and high-scale deployments.

Download and apply the Calico manifest:

curl https://docs.projectcalico.org/manifests/calico.yaml -O

kubectl apply -f calico.yaml

**Verifying Calico Installation**: After installation, verify that the Calico Pods are running:

kubectl get pods -n kube-system -l k8s-app=calico-node

**Testing Pod Connectivity**: Once Calico is installed, let's test the networking between Pods by creating two Pods and testing communication between them.

Create pod-1.yaml:

apiVersion: v1

kind: Pod

metadata:

name: pod-1

spec:

containers:

- name: nginx

image: nginx

Create pod-2.yaml:

apiVersion: v1

kind: Pod

metadata:

name: pod-2

spec:

containers:

- name: nginx

image: nginx

Deploy the Pods:

kubectl apply -f pod-1.yaml

kubectl apply -f pod-2.yaml

Once both Pods are running, execute the following to test connectivity:

kubectl exec -it pod-1 -- /bin/bash

curl pod-2

You should get a response from pod-2, confirming that the Pods can communicate through Calico.

**Network Policy Example**: Kubernetes allows you to define network policies to control the traffic flow between Pods. Create a simple network policy to deny all incoming traffic to pod-2 except from pod-1.

Create a network-policy.yaml file:

apiVersion: networking.k8s.io/v1

kind: NetworkPolicy

metadata:

name: deny-all-except-pod-1

spec:

podSelector:

matchLabels:

app: nginx

ingress:

- from:

- podSelector:

matchLabels:

app: nginx

Apply the network policy:

kubectl apply -f network-policy.yaml

Now, test connectivity from pod-1 and pod-2:

kubectl exec -it pod-1 -- /bin/bash

curl pod-2 # Should work

kubectl exec -it pod-2 -- /bin/bash

curl pod-1 # Should fail due to network policy

### ****Key Takeaways****:

**CNI** is the specification that allows Kubernetes to configure and manage networking for Pods. CNI plugins enable Kubernetes to have a flexible and customizable networking model.

Different CNI plugins offer various features such as network policies, encryption, and performance optimizations, which can be tailored to the needs of the deployment.

**Calico**, **Flannel**, and **Weave** are some of the commonly used CNI plugins in Kubernetes.

## ****3. Understand Pod Networking Concepts****

### ****Overview:****

Pod networking in Kubernetes is a critical component that enables Pods to communicate with each other, with external systems, and with services running inside the cluster. Kubernetes abstracts networking and assigns each Pod its own unique IP address, which simplifies network management.

In this section, we’ll cover:

How Kubernetes assigns IPs to Pods.

How Pods communicate with each other.

The importance of network policies in controlling traffic flow between Pods.

How network plugins (like CNI) fit into Pod networking.

### ****Key Components of Pod Networking:****

**Pod IP Address**: Each Pod in Kubernetes receives a unique IP address, which is essential for communication between Pods. This IP address is typically provided by the CNI plugin used in the cluster. Kubernetes does not have any internal DNS that maps Pods to their IP addresses, so service discovery mechanisms (like CoreDNS) help resolve Pod names to their respective IP addresses.

**Pod-to-Pod Communication**: Pods within the same cluster can communicate with each other using their assigned IPs. By default, Kubernetes allows all Pods to communicate with each other within the same cluster, unless otherwise restricted by network policies.

**Example Use Case**: Microservices often need to communicate with each other. Each service runs in its own Pod and can reach other Pods in the same namespace or across namespaces by resolving their names or IP addresses.

**Services for Internal Communication**: Kubernetes introduces the concept of Services to abstract the network access to a set of Pods. A Service provides a stable endpoint (usually a DNS name) to access Pods, even if the Pods themselves have dynamically assigned IP addresses.

**Example Use Case**: A "frontend" service needs to communicate with a "backend" service. The backend Pods might be constantly scaling up or down, so a Service ensures the frontend can consistently connect to them, even if individual Pods change IP addresses.

**Pod-to-External Communication**: Pods may need to communicate with external systems outside the Kubernetes cluster, such as databases, third-party APIs, or user requests from the internet. This is typically done through Services, which route external traffic to the appropriate Pods.

**Example Use Case**: A web application running in a Pod might need to fetch data from a third-party API. The Pod can initiate outbound traffic using its assigned IP address.

### ****Pod Networking Model in Kubernetes****:

**Flat Networking**: Kubernetes assumes a "flat" network model where every Pod in the cluster can communicate with every other Pod directly using IP addresses. This makes network setup simpler, but it also requires tight control over security and traffic flow.

**Network Policies**: Network Policies are used to restrict Pod-to-Pod communication. They define how Pods can communicate with each other, or with services external to the cluster. By default, Pods can communicate freely, but with Network Policies, we can control traffic using selectors and rules.

### ****Lab: Understanding Pod Networking Concepts****

#### ****Objective****:

In this lab, we'll explore how Pod networking works in Kubernetes and test Pod-to-Pod communication, service discovery, and Network Policies.

#### ****Prerequisites****:

A running Kubernetes cluster with CNI plugin installed.

kubectl configured for the cluster.

#### ****Step-by-Step Process****:

**Deploy Two Pods for Communication**: Let’s create two simple Pods that will communicate with each other.

Create pod-1.yaml:

apiVersion: v1

kind: Pod

metadata:

name: pod-1

spec:

containers:

- name: nginx

image: nginx

Create pod-2.yaml:

apiVersion: v1

kind: Pod

metadata:

name: pod-2

spec:

containers:

- name: nginx

image: nginx

Apply the Pods:

kubectl apply -f pod-1.yaml

kubectl apply -f pod-2.yaml

**Verify Pod IPs**: Check the IPs assigned to each Pod:

kubectl get pods -o wide

You should see the Pods' names and their respective IP addresses.

**Test Pod-to-Pod Communication**: Let’s test whether pod-1 can communicate with pod-2 using curl.

Execute curl from pod-1 to pod-2:

kubectl exec -it pod-1 -- curl pod-2

This should return the default NGINX page from pod-2.

**Create a Service for Pod Communication**: To abstract the communication between Pods, let’s create a Service.

Create service.yaml:

apiVersion: v1

kind: Service

metadata:

name: nginx-service

spec:

selector:

app: nginx

ports:

- protocol: TCP

port: 80

targetPort: 80

Apply the Service:

kubectl apply -f service.yaml

**Verify Service Access**: Once the service is created, you can access the Pods using the service name:

kubectl exec -it pod-1 -- curl nginx-service

This should again return the NGINX page, confirming that pod-1 can access pod-2 through the Service.

**Deploy Network Policy to Control Communication**: Let’s now restrict traffic from pod-1 to pod-2 using a Network Policy.

Create network-policy.yaml:

apiVersion: networking.k8s.io/v1

kind: NetworkPolicy

metadata:

name: restrict-pod-communication

spec:

podSelector:

matchLabels:

app: nginx

ingress:

- from:

- podSelector:

matchLabels:

app: nginx

Apply the Network Policy:

kubectl apply -f network-policy.yaml

**Test Network Policy**: Test the communication after applying the policy.

From pod-1, try to access pod-2:

kubectl exec -it pod-1 -- curl pod-2 # Should fail

You should see that the communication is blocked due to the applied Network Policy.

### ****Key Takeaways****:

Kubernetes provides each Pod with a unique IP address, enabling easy and efficient communication between Pods.

Network Policies allow you to control and restrict traffic between Pods, helping secure communications and enforce policies within the cluster.

Services provide a stable DNS name to access Pods, ensuring reliable communication in dynamic environments.

## ****4. CoreDNS Overview of Kubernetes****

### ****Overview:****

CoreDNS is the default DNS server in Kubernetes that provides DNS resolution for services and Pods inside the Kubernetes cluster. It plays a critical role in service discovery and ensuring that applications running in Pods can find and communicate with each other through DNS.

In Kubernetes, all services and Pods are automatically registered in the DNS system, allowing other Pods to discover them via DNS names. For example, a Pod in Kubernetes can access a service via a DNS query instead of needing to know the service’s IP address.

### ****CoreDNS in Kubernetes****:

CoreDNS is deployed as a set of Pods within the kube-system namespace in the cluster. It runs as a set of containers that listen on the DNS port (53) and respond to DNS queries.

By default, CoreDNS has a set of configuration files that dictate how it resolves DNS queries. Kubernetes creates DNS records for each Service and Pod and manages these records through CoreDNS.

### ****Key Concepts of CoreDNS****:

**DNS for Services**:

Kubernetes automatically creates DNS records for each service, making it easy for Pods to access services by their name.

A service named nginx-service in the default namespace will automatically get a DNS name nginx-service.default.svc.cluster.local.

**DNS for Pods**:

Pods can also be resolved by DNS in Kubernetes. Pod names are given a specific DNS format to resolve between Pods.

The DNS format for Pods is: pod-name.pod-namespace.pod.cluster.local.

**DNS Resolution Flow**:

When a Pod tries to access a service by its DNS name (e.g., nginx-service), CoreDNS first checks the internal DNS records.

If the DNS record exists, it resolves the query and provides the correct IP address for the service.

If no record is found in CoreDNS, it can forward the request to external DNS servers if configured.

**Customizing CoreDNS**:

CoreDNS can be configured with different plugins to extend its functionality. For example, you might use a plugin to resolve custom DNS records or to forward DNS queries to external servers.

**Cluster DNS**:

CoreDNS in a Kubernetes cluster typically handles DNS queries for services within the cluster and can also resolve external domains if configured correctly.

Services like kubernetes.default.svc.cluster.local are automatically created and serve the internal cluster’s DNS resolution needs.

### ****Key Use Cases for CoreDNS****:

**Service Discovery**: Kubernetes workloads often need to discover services in a dynamic environment. CoreDNS allows Pods to find services by their DNS names without knowing their IPs.

**Example Use Case**: A microservice-based application might need to connect to a database service. The database service can be referred to by its DNS name, and CoreDNS will resolve the IP for the Pod running the database.

**Pod Communication**: Kubernetes allows Pods to communicate with each other by resolving their names through CoreDNS. This is crucial in multi-Pod, multi-service applications.

**Example Use Case**: A web server Pod needs to make an API request to a backend service. Using DNS names, the web server can easily find the backend service without having to know its IP address.

**External DNS Resolution**: CoreDNS can also be configured to resolve external DNS names, providing Pods with the ability to access resources outside the cluster (e.g., public APIs, databases).

**Example Use Case**: A Pod might need to connect to a third-party API. CoreDNS can be configured to forward the DNS request to an external DNS provider, ensuring the Pod can access the service.

### ****Lab: Exploring CoreDNS in Kubernetes****

#### ****Objective****:

In this lab, we'll explore CoreDNS functionality by checking DNS resolution for services and Pods within a Kubernetes cluster. We will also test the customization of CoreDNS.

#### ****Prerequisites****:

A Kubernetes cluster set up using kubeadm (with 2 worker nodes and 1 master).

kubectl configured to interact with the cluster.

#### ****Step-by-Step Process****:

**Verify CoreDNS Deployment**: First, let's verify that CoreDNS is running in the kube-system namespace.

Check the CoreDNS Pods:

kubectl get pods -n kube-system | grep coredns

You should see two CoreDNS Pods running. This indicates that CoreDNS is properly deployed in your cluster.

**Test DNS Resolution for Services**: Now let’s test DNS resolution for services inside the cluster. We'll use the busybox image to perform DNS lookups.

Create a test Pod with busybox:

kubectl run -i --tty --rm busybox --image=busybox --restart=Never --command -- /bin/sh

Inside the busybox Pod, perform a DNS lookup for the kubernetes service:

nslookup kubernetes.default.svc.cluster.local

The nslookup command should resolve the IP address of the kubernetes service. If successful, it confirms that CoreDNS is handling internal DNS resolution.

**Test DNS Resolution for Pods**: Now, let’s resolve the name of a Pod from within another Pod.

Create two Pods:

kubectl run pod-1 --image=nginx --restart=Never

kubectl run pod-2 --image=nginx --restart=Never

Once the Pods are running, find the IPs of the Pods:

kubectl get pods -o wide

Inside pod-1, perform a DNS lookup for pod-2:

kubectl exec -it pod-1 -- nslookup pod-2

You should see the IP address of pod-2, confirming that CoreDNS resolves Pod names to IPs within the cluster.

**Customize CoreDNS Configuration**: CoreDNS can be configured to forward DNS requests to external servers or customize DNS handling. Let’s explore this by modifying the CoreDNS ConfigMap.

View the CoreDNS ConfigMap:

kubectl -n kube-system get configmap coredns -o yaml

You can edit the ConfigMap using:

kubectl -n kube-system edit configmap coredns

You can modify this file to add additional DNS forwarders or enable custom DNS records for your cluster.

**Apply the Configuration Changes**: After editing the CoreDNS ConfigMap, apply the changes:

kubectl -n kube-system rollout restart deployment coredns

This will restart the CoreDNS Pods, applying the new configuration.

### ****Key Takeaways****:

CoreDNS is a critical component for DNS resolution within Kubernetes, enabling service discovery and communication between Pods.

It resolves service names and Pod names into IP addresses, allowing dynamic environments like Kubernetes to function smoothly.

CoreDNS can be customized to forward DNS queries to external servers or to add custom DNS records.

## ****5. Understanding Ingress in Kubernetes****

### ****Overview****:

Ingress in Kubernetes is a powerful resource that manages external access to services within the cluster, typically HTTP and HTTPS traffic. It allows you to expose your services to the outside world without needing to expose individual services with external IP addresses. With Ingress, you can define routes based on hostnames and paths, set up SSL termination, and even load-balance traffic across multiple services.

### ****Ingress Controller****:

An **Ingress Controller** is responsible for implementing the rules defined in an Ingress resource. It is essentially a load balancer that routes traffic based on the Ingress rules to the appropriate service. Some popular Ingress Controllers include NGINX, Traefik, and HAProxy.

### ****Key Components of Ingress****:

**Ingress Resource**: The Ingress resource defines how HTTP/HTTPS traffic is routed to different services within the cluster. It specifies the rules for routing traffic, which can include:

**Host**: The domain name (e.g., example.com).

**Path**: The URL path (e.g., /api).

**Service**: The backend service that will handle the traffic.

Example of an Ingress resource:

apiVersion: networking.k8s.io/v1

kind: Ingress

metadata:

name: my-ingress

spec:

rules:

- host: my-app.example.com

http:

paths:

- path: /path1

pathType: Prefix

backend:

service:

name: my-service

port:

number: 80

**Ingress Controller**: The Ingress Controller is a load balancer that listens to changes in the Ingress resource and routes traffic accordingly. To use Ingress, you must have an Ingress Controller installed in your cluster.

Common Ingress Controllers:

**NGINX Ingress Controller**: The most commonly used, suitable for both production and development environments.

**Traefik Ingress Controller**: A dynamic, modern Ingress controller that integrates easily with Kubernetes.

**HAProxy Ingress Controller**: Known for high performance, suitable for large-scale Kubernetes clusters.

**Ingress Annotations**: Ingress resources can have annotations to specify additional configuration options, such as SSL termination, rewrite rules, and backend configuration.

Example of SSL termination via an annotation:

apiVersion: networking.k8s.io/v1

kind: Ingress

metadata:

name: my-ingress

annotations:

nginx.ingress.kubernetes.io/ssl-redirect: "true"

### ****Key Features of Ingress****:

**Path-based Routing**: You can route traffic to different services based on URL paths, enabling fine-grained control over how incoming traffic is distributed.

**Example Use Case**: If you have two services, frontend and api, you can route / to frontend and /api to api using Ingress rules.

**Host-based Routing**: You can route traffic to different services based on the hostname in the request. This is useful for managing multiple websites or applications within a single Kubernetes cluster.

**Example Use Case**: If you have app1.example.com and app2.example.com, you can route traffic to different backend services based on the hostname.

**SL Termination**: Ingress can be configured to handle SSL termination, offloading the SSL decryption from your services and simplifying certificate management.

**Example Use Case**: For secure HTTPS traffic, you can terminate SSL at the Ingress Controller and pass the unencrypted traffic to your services.

**Load Balancing**: Ingress provides built-in load balancing, distributing traffic across multiple instances of a service. You can use this for high availability and scalability.

**Example Use Case**: Multiple instances of the web-service can be load balanced by an Ingress Controller, ensuring the application remains highly available.

### ****Ingress vs. Services****:

A **Service** exposes a set of Pods in a cluster, but it doesn’t offer any way to configure HTTP/HTTPS routing or load balancing across different services.

An **Ingress** allows you to route external traffic to different services based on rules and provides advanced features like SSL termination, path-based routing, and host-based routing.

### ****Use Cases****:

**Multi-service Application**: Suppose you have multiple services like a frontend, backend, and API, all running within the same Kubernetes cluster. Instead of creating multiple LoadBalancers for each service, you can define a single Ingress to manage traffic routing to all services based on hostnames and paths.

**Single Point of Entry**: An Ingress provides a single point of entry for all external traffic. This simplifies the management of external access and makes it easier to scale, monitor, and configure routing policies.

**Secure Access**: Ingress allows you to manage secure (HTTPS) traffic with SSL certificates. You can configure SSL termination at the Ingress level and offload the decryption process, ensuring better security practices.

### ****Lab: Configuring and Managing Ingress****

#### ****Objective****:

In this lab, we will deploy an Ingress Controller, configure Ingress resources, and test path and host-based routing within a Kubernetes cluster.

#### ****Prerequisites****:

A Kubernetes cluster (with 2 worker nodes and 1 master) deployed using kubeadm.

kubectl configured to interact with the cluster.

A domain name (if you want to test host-based routing, optional).

#### ****Step-by-Step Process****:

**Install Ingress Controller (NGINX)**: First, we need to install the NGINX Ingress Controller.

Apply the NGINX Ingress Controller manifests:

kubectl apply -f https://raw.githubusercontent.com/kubernetes/ingress-nginx/main/deploy/static/provider/cloud/deploy.yaml

Verify the Ingress Controller is running:

kubectl get pods -n ingress-nginx

**Create Sample Services**: Now, let’s create two services (frontend and api) to test our Ingress rules.

Create a frontend service:

apiVersion: v1

kind: Service

metadata:

name: frontend

spec:

selector:

app: frontend

ports:

- protocol: TCP

port: 80

targetPort: 8080

---

apiVersion: apps/v1

kind: Deployment

metadata:

name: frontend

spec:

replicas: 1

selector:

matchLabels:

app: frontend

template:

metadata:

labels:

app: frontend

spec:

containers:

- name: frontend

image: nginx

ports:

- containerPort: 8080

Create an api service:

apiVersion: v1

kind: Service

metadata:

name: api

spec:

selector:

app: api

ports:

- protocol: TCP

port: 80

targetPort: 8080

---

apiVersion: apps/v1

kind: Deployment

metadata:

name: api

spec:

replicas: 1

selector:

matchLabels:

app: api

template:

metadata:

labels:

app: api

spec:

containers:

- name: api

image: nginx

ports:

- containerPort: 8080

**Create Ingress Resource**: Now, let’s create an Ingress resource to route traffic to these services based on path and hostname.

Example Ingress configuration:

apiVersion: networking.k8s.io/v1

kind: Ingress

metadata:

name: my-ingress

spec:

rules:

- host: my-app.example.com

http:

paths:

- path: /frontend

pathType: Prefix

backend:

service:

name: frontend

port:

number: 80

- path: /api

pathType: Prefix

backend:

service:

name: api

port:

number: 80

Apply the Ingress:

kubectl apply -f my-ingress.yaml

**Test the Ingress**: If you're using a custom domain (e.g., my-app.example.com), ensure the DNS is pointing to the Ingress Controller's external IP. Alternatively, use a temporary solution like port forwarding.

Find the external IP of the Ingress Controller:

kubectl get svc -n ingress-nginx

Test the routing by accessing the services via your browser or curl:

For the frontend service:

curl http://<EXTERNAL-IP>/frontend

For the API service:

curl http://<EXTERNAL-IP>/api

### ****Key Takeaways****:

Ingress allows you to manage external access to your services and applications in a Kubernetes cluster.

You can route traffic based on hostname and path, manage SSL termination, and configure advanced routing rules using annotations.

Ingress Controllers handle traffic routing based on the rules specified in Ingress resources.

## ****6. Understanding Namespace & Use-Cases****

### ****Overview****:

A **Namespace** in Kubernetes provides a mechanism for isolating resources within the cluster. It allows you to group and organize resources in a way that makes it easier to manage them. Namespaces are particularly useful in larger clusters, where different teams or environments need to share the same cluster but want to isolate their resources from each other.

Namespaces help in:

Organizing resources (e.g., Pods, Services, Deployments) by environment or team.

Managing resource quotas for different environments.

Controlling access to resources using RBAC (Role-Based Access Control).

### ****Why Use Namespaces****?

**Multi-Tenant Clusters**: In a shared Kubernetes cluster, different teams or departments can use different namespaces to avoid resource conflicts and allow proper access control.

**Environment Segregation**: You can create separate namespaces for different environments like dev, staging, and production.

**Resource Isolation**: Namespaces can limit resource usage (CPU, memory) through resource quotas and ensure fair resource allocation across different teams.

### ****Default Namespace****:

Kubernetes provides a **default namespace**, which is where objects are created when no specific namespace is specified. Other system-level namespaces include:

**kube-system**: Contains system-related resources, such as the Kubernetes control plane and services.

**kube-public**: Used for resources that are accessible to all users, including unauthenticated users.

**kube-node-lease**: Used for node lease information.

### ****Creating and Managing Namespaces****:

**Creating a Namespace**: You can create a new namespace using the following command:

kubectl create namespace <namespace-name>

Example:

kubectl create namespace dev

**Viewing Namespaces**: To list all namespaces in the cluster:

kubectl get namespaces

**Using a Namespace**: Once a namespace is created, you can refer to it while creating other resources by specifying the namespace field in the resource YAML file or by using the -n flag in kubectl commands.

Example:

kubectl get pods -n dev

**Delete a Namespace**: To delete a namespace and all its resources:

kubectl delete namespace <namespace-name>

### ****Use Cases for Namespaces****:

**Multi-Environment Setup**: In a single Kubernetes cluster, you might want to separate your environments (e.g., dev, staging, production) into different namespaces. This prevents cross-environment interference and allows better management.

Example: You could create a dev namespace for developers to deploy their apps and a prod namespace for production apps. Resources in the prod namespace could have stricter resource limits and access controls.

**Resource Management**: Namespaces allow you to apply **resource quotas** to prevent one namespace from consuming all the resources in the cluster. Resource quotas can limit CPU, memory, or the number of objects (e.g., Pods, Services).

Example: For the dev namespace, you might allocate 4 CPU cores and 16 GB of memory, while for the prod namespace, you allocate 8 CPU cores and 32 GB of memory.

**Role-Based Access Control (RBAC)**: You can use namespaces in combination with RBAC to ensure that users or service accounts have access only to the resources in their specific namespace.

Example: In a multi-team environment, developers in the dev namespace can have access to deploy applications, but users in the prod namespace can have read-only access to the services in that namespace.

**Team Isolation**: In an organization with multiple teams, you can create a separate namespace for each team. This ensures that resources are isolated between teams and prevents accidental resource manipulation across teams.

Example: The marketing team can have its own namespace for their applications, which are completely isolated from the applications in the engineering namespace.

**Handling External Services**: Sometimes you may want to use Kubernetes namespaces for isolating external services (such as databases or message queues) for different environments or teams. This is useful in multi-tenancy situations.

### ****Lab: Creating and Managing Namespaces****

#### ****Objective****:

In this lab, we will create multiple namespaces and deploy resources in these namespaces to understand how they work and how we can isolate resources in a Kubernetes cluster.

#### ****Prerequisites****:

A Kubernetes cluster running with kubeadm.

kubectl configured to access your cluster.

#### ****Step-by-Step Process****:

**Create Namespaces**: Let's start by creating three namespaces:

dev

staging

prod

Create the namespaces using the following commands:

kubectl create namespace dev

kubectl create namespace staging

kubectl create namespace prod

**Verify the Creation of Namespaces**: To ensure that the namespaces were created successfully, list all namespaces:

kubectl get namespaces

**Deploy Pods in Different Namespaces**: Now, let's deploy a simple NGINX pod in each of these namespaces.

Create a YAML file for the dev namespace pod:

apiVersion: v1

kind: Pod

metadata:

name: nginx-dev

namespace: dev

spec:

containers:

- name: nginx

image: nginx

Apply the YAML file:

kubectl apply -f nginx-dev.yaml

Similarly, create and apply the YAML files for staging and prod namespaces, adjusting the namespace field accordingly.

**Verify Pods in Each Namespace**: To check the pods in the dev namespace:

kubectl get pods -n dev

Similarly, check the pods in staging and prod namespaces:

kubectl get pods -n staging

kubectl get pods -n prod

**Access the Pods**: You can use port forwarding to access these pods locally:

kubectl port-forward -n dev pod/nginx-dev 8080:80

Then open a browser and go to http://localhost:8080 to access the NGINX web page running in the dev namespace.

**Cleanup**: Once the lab is complete, delete the namespaces:

kubectl delete namespace dev

kubectl delete namespace staging

kubectl delete namespace prod

### ****Key Takeaways****:

**Namespaces** are used for organizing and isolating resources within a Kubernetes cluster.

They help in managing resources for different teams or environments, improving security and resource management.

You can use namespaces in combination with **resource quotas** and **RBAC** for better access control and resource allocation.

## ****7. Metal Load Balancer****

### ****Overview****:

A **Metal Load Balancer** (also known as **MetalLB**) is a load balancing solution designed for Kubernetes clusters that do not have access to cloud load balancing services (e.g., in on-premise environments or private clouds). MetalLB provides Layer 2 (L2) and Layer 3 (L3) load balancing capabilities for Kubernetes services, particularly **LoadBalancer-type services**.

In cloud environments like AWS, GCP, or Azure, Kubernetes can automatically provision an external load balancer when you create a service of type LoadBalancer. However, in environments like on-premise data centers or when running Kubernetes on bare metal, there is no cloud provider to automatically provision an external load balancer. This is where MetalLB comes into play.

### ****Why Use MetalLB****?

**Bare Metal Clusters**: Kubernetes clusters running on bare metal (without a cloud provider) need an external load balancer to distribute traffic to different Pods. MetalLB provides this functionality.

**Supports Layer 2 and Layer 3 Load Balancing**: MetalLB supports both L2 (ARP-based) and L3 (BGP-based) modes, allowing it to work in a variety of network environments.

**Automatic IP Allocation**: MetalLB can automatically allocate IPs for LoadBalancer services, providing highly available and fault-tolerant load balancing.

### ****Components of MetalLB****:

**Controller**: This component runs as a Kubernetes Deployment and manages the configuration of load balancers in the cluster. It works by ensuring that MetalLB’s configuration aligns with the Kubernetes service objects.

**Speaker**: The speaker runs on each Kubernetes node and is responsible for advertising IPs for LoadBalancer services to the network. It does this via ARP (in L2 mode) or BGP (in L3 mode).

### ****MetalLB Modes****:

**Layer 2 (L2)**: In this mode, MetalLB uses ARP to advertise IPs to the local network. This mode is useful when you don't have access to BGP and want a simpler setup.

**Layer 3 (L3)**: In this mode, MetalLB uses BGP to advertise IPs to the wider network. It provides more advanced routing and is useful for integrating with other networking devices that support BGP.

### ****Use Cases for MetalLB****:

**On-Premise Kubernetes Clusters**: For Kubernetes clusters deployed on bare metal servers, MetalLB can provide external load balancing for services exposed to the outside world.

**Private Clouds**: If you're running Kubernetes in a private cloud without a cloud-native load balancer, MetalLB can fill the gap.

**Lab and Testing Environments**: For developers running Kubernetes on their local machines or within a virtualized environment, MetalLB provides a lightweight, easy-to-setup load balancing solution.

### ****Setting Up MetalLB in Kubernetes****:

**Install MetalLB**: To install MetalLB, we’ll use the kubectl command to apply the manifests from MetalLB’s official GitHub repository:

kubectl apply -f https://raw.githubusercontent.com/metallb/metallb/v0.12.1/manifests/metallb.yaml

This will install the necessary components for MetalLB, including the controller and speaker.

**Configure a Layer 2 Address Pool**: After installation, you need to configure MetalLB with a pool of IP addresses that it can assign to LoadBalancer services. This can be done by creating a ConfigMap in the metallb-system namespace.

First, create a configuration file (metallb-config.yaml) like the following:

apiVersion: v1

kind: ConfigMap

metadata:

name: config

namespace: metallb-system

data:

config: |

address-pools:

- name: default

protocol: layer2

addresses:

- 192.168.1.240-192.168.1.250

Then apply the configuration:

kubectl apply -f metallb-config.yaml

This will configure MetalLB to assign IP addresses from the range 192.168.1.240-192.168.1.250 to services of type LoadBalancer.

**Verify MetalLB Installation**: To check that MetalLB has been successfully installed, run:

kubectl get pods -n metallb-system

You should see MetalLB’s controller and speaker pods running.

**Create a LoadBalancer Service**: Now, let’s create a Kubernetes service of type LoadBalancer. This service will use an IP from the MetalLB address pool.

Here’s an example of a LoadBalancer service YAML file for an NGINX deployment:

apiVersion: v1

kind: Service

metadata:

name: nginx-lb

spec:

selector:

app: nginx

ports:

- protocol: TCP

port: 80

targetPort: 80

type: LoadBalancer

Apply the service definition:

kubectl apply -f nginx-lb.yaml

**Access the LoadBalancer IP**: After creating the service, MetalLB will assign an IP from the configured pool to the nginx-lb service. You can check the assigned IP by running:

kubectl get svc nginx-lb

You should see an external IP (e.g., 192.168.1.240) assigned to the service. You can now access the service via that IP.

**Testing the LoadBalancer**: You can test if the LoadBalancer service is working by accessing it via the assigned IP:

curl http://192.168.1.240

This should return the NGINX welcome page if everything is set up correctly.

### ****Lab: Deploying Metal Load Balancer****

#### ****Objective****:

In this lab, we will install MetalLB on a Kubernetes cluster and configure it to provide external load balancing for services exposed in the cluster.

#### ****Step-by-Step Process****:

**Install MetalLB**: Run the following command to install MetalLB:

kubectl apply -f https://raw.githubusercontent.com/metallb/metallb/v0.12.1/manifests/metallb.yaml

**Configure Layer 2 Address Pool**: Create a configuration file (metallb-config.yaml) with the address pool configuration, then apply it:

kubectl apply -f metallb-config.yaml

**Verify MetalLB Components**: Check the running pods in the metallb-system namespace:

kubectl get pods -n metallb-system

**Create a LoadBalancer Service**: Create and apply the nginx-lb.yaml file to expose the NGINX service as a LoadBalancer:

kubectl apply -f nginx-lb.yaml

**Verify LoadBalancer IP**: Check the external IP assigned to the service:

kubectl get svc nginx-lb

**Access the Service**: Use curl or your browser to access the service via the external IP.

**Cleanup**: Delete the MetalLB resources and the NGINX service once the lab is complete:

kubectl delete -f nginx-lb.yaml

kubectl delete -f metallb-config.yaml

### ****Key Takeaways****:

**MetalLB** provides load balancing for Kubernetes services in on-premise and bare-metal environments.

It supports both Layer 2 and Layer 3 modes of load balancing.

MetalLB can be easily installed and configured using Kubernetes manifests.

It is an essential tool for Kubernetes clusters that do not have access to cloud-provider load balancing services.

## ****Module 12: Troubleshooting****

### ****1. Ways to Troubleshoot ETCD Failure****

#### ****Overview****:

ETCD is a critical component of the Kubernetes control plane. It is a distributed key-value store used to store all cluster data, such as configurations, secrets, and metadata. If ETCD fails, it can lead to loss of cluster state, rendering the Kubernetes cluster unusable.

#### ****Symptoms of ETCD Failure****:

Inability to access cluster API.

Kube-apiserver may crash or experience latency.

Kubernetes components (e.g., kubectl, kubelet) may not function correctly.

Nodes or Pods may become unresponsive or unreachable.

#### ****Troubleshooting Steps****:

**Check ETCD Status**: Use the kubectl command to check if ETCD is running properly:

kubectl get pods -n kube-system -l component=etcd

If ETCD is down, you won't see any pods running for ETCD.

**Check ETCD Logs**: Review the ETCD logs to identify any error messages:

kubectl logs -n kube-system <etcd-pod-name>

Look for errors such as failed to start or network connectivity issues.

**Check the ETCD Cluster Health**: Run the following command to check the health of the ETCD cluster:

etcdctl cluster-health

This command checks the status of all ETCD nodes and provides information on whether they are healthy.

**Check Disk Space**: ETCD relies on disk space to persist data. If the disk is full, ETCD may fail:

df -h

Ensure the disk has enough free space for ETCD to write data.

**Check for Network Issues**: ETCD nodes need network connectivity between them. Ensure that the network is stable and that there are no firewalls blocking communication.

**Restore from Backup**: If ETCD is corrupted, restoring from a backup may be the only solution. Use the following command to restore from a snapshot:

etcdctl snapshot restore /path/to/snapshot.db

Ensure you are restoring to the same version of ETCD

#### ****Real-World Use Case****:

Imagine you are running a Kubernetes cluster for a production application. The ETCD store has failed, and the cluster is no longer responding. By following the steps above, you first determine that ETCD is down and identify that the issue is due to a full disk. After cleaning up disk space and restarting ETCD, the cluster is restored, and your services are back online.

#### ****Lab: Troubleshooting ETCD Failure****

**Simulate ETCD Failure**: To simulate ETCD failure, you can stop the ETCD pod:

kubectl delete pod -n kube-system <etcd-pod-name>

**Check Cluster Status**: Run the following command to verify the status of the cluster:

kubectl get nodes

**Check Logs**: Inspect the ETCD logs for any errors:

kubectl logs -n kube-system <etcd-pod-name>

**Check ETCD Health**: Use etcdctl to verify the health of the ETCD cluster:

etcdctl cluster-health

**Fix the Failure**: If the issue is disk space, clear the space and restart the ETCD pod. If the issue is network-related, ensure communication between the nodes is restored.

### ****2. Ways to Troubleshoot Kubelet Failure****

#### ****Overview****:

The **kubelet** is the primary node agent responsible for maintaining the desired state of pods running on a node. A kubelet failure may lead to pods being stuck in a Pending state, not starting, or being continuously restarted.

#### ****Symptoms of Kubelet Failure****:

Pods failing to start on a node.

Nodes in a NotReady state.

Kubelet logs showing errors or failures to communicate with the API server.

#### ****Troubleshooting Steps****:

**Check Node Status**: Verify that the node is in the Ready state:

kubectl get nodes

If the node is in NotReady or Unknown state, there may be issues with the kubelet.

**Check Kubelet Logs**: Examine the logs of the kubelet to find errors:

journalctl -u kubelet -f

Look for error messages such as failed to communicate with API server or unable to mount volume.

**Check Kubelet Configuration**: Ensure the kubelet is properly configured. Check the kubelet configuration file (usually /etc/kubernetes/kubelet.conf).

**Check Resource Utilization**: High CPU or memory utilization on the node can cause the kubelet to fail. Check resource usage:

top

If resource usage is high, consider increasing the node capacity or optimizing workloads.

**Verify Kubelet Connection to API Server**: Ensure that the kubelet can communicate with the Kubernetes API server:

curl https://<api-server-ip>:6443/healthz

**Check Docker or Container Runtime**: If you are using Docker as the container runtime, check Docker logs to ensure that there are no runtime issues:

journalctl -u docker -f

#### ****Real-World Use Case****:

In a Kubernetes cluster running in production, the kubelet on one of the nodes stops working, causing all the pods on that node to be evicted. The issue is identified as a high CPU usage due to a runaway process. After freeing up resources, the kubelet starts functioning again, and the pods are rescheduled onto the node.

#### ****Lab: Troubleshooting Kubelet Failure****

**Simulate Kubelet Failure**: Stop the kubelet service:

systemctl stop kubelet

**Check Node Status**: Check if the node is in the NotReady state:

kubectl get nodes

**Check Kubelet Logs**: View the kubelet logs to identify errors:

journalctl -u kubelet -f

**Start Kubelet**: Start the kubelet service again:

systemctl start kubelet

**Check Node Status Again**: Verify that the node is now in Ready state:

kubectl get nodes

### ****3. Ways to Troubleshoot Container Runtime Failure****

#### ****Overview****:

The **container runtime** is responsible for running and managing containers in a Kubernetes cluster. If the container runtime fails, Kubernetes cannot run pods on the node, and the kubelet will report issues with starting containers.

#### ****Symptoms of Container Runtime Failure****:

Pods in ContainerCreating state indefinitely.

Kubelet logs showing errors related to container creation or runtime.

Node's container runtime process is unresponsive.

#### ****Troubleshooting Steps****:

**Check Container Runtime Status**: Check the status of the container runtime (e.g., Docker or containerd):

systemctl status docker

# or

systemctl status containerd

**Check Container Runtime Logs**: Review the logs of the container runtime to find errors:

journalctl -u docker -f

# or

journalctl -u containerd -f

**Check Pod Events**: Look for specific pod events that indicate issues with the container runtime

kubectl describe pod <pod-name>

**Verify Container Runtime Version**: Ensure that the container runtime version is compatible with your Kubernetes version.

**Check Network and Disk Space**: Network issues or disk space problems may affect the container runtime. Check for connectivity issues and ensure there is enough disk space for container images.

#### ****Real-World Use Case****:

You notice that pods are stuck in the ContainerCreating state. The logs indicate a failure with the container runtime. Upon inspection, you find that the Docker service is down due to an underlying disk space issue. Once the disk is cleared, the Docker service is restarted, and the pods are successfully created.

#### ****Lab: Troubleshooting Container Runtime Failure****

**Simulate Container Runtime Failure**: Stop the container runtime (e.g., Docker):

systemctl stop docker

**Check Pod Status**: Check if the pods are stuck in ContainerCreating:

kubectl get pods

**Check Container Runtime Logs**: Inspect the container runtime logs:

journalctl -u docker -f

**Start the Container Runtime**: Start the container runtime again:

systemctl start docker

**Verify Pod Status**: Check if the pods are now running:

kubectl get pods

### ****4. Ways to Troubleshoot Scheduler Failure****

#### ****Overview****:

The **Kubernetes scheduler** is responsible for scheduling pods to run on available nodes based on resource requirements, taints, and tolerations. If the scheduler fails, no new pods will be scheduled to run.

#### ****Symptoms of Scheduler Failure****:

Pods remain in Pending state and are not scheduled.

The scheduler pod is not running or showing errors in the logs.

The node may be underutilized but no new pods are assigned to it.

#### ****Troubleshooting Steps****:

**Check Scheduler Status**: Verify the status of the scheduler pod:

kubectl get pods -n kube-system -l component=kube-scheduler

**Check Scheduler Logs**: Review the scheduler logs for error messages:

kubectl logs -n kube-system <scheduler-pod-name>

**Check Resource Availability**: Ensure that there are enough resources (CPU, memory) on the nodes for scheduling new pods:

kubectl describe node <node-name>

**Check Taints and Tolerations**: Ensure there are no taints preventing pod scheduling on nodes:

kubectl describe node <node-name> | grep Taints

**Check Scheduling Policies**: Verify if there are any custom scheduling policies affecting pod scheduling.

#### ****Real-World Use Case****:

In a Kubernetes cluster with limited resources, the scheduler is not assigning new pods to nodes due to resource constraints. By inspecting the scheduler logs and checking the resource utilization on nodes, you find that the scheduler is not allocating resources efficiently. After optimizing resource allocation, the scheduler begins scheduling pods again.

#### ****Lab: Troubleshooting Scheduler Failure****

**Simulate Scheduler Failure**: Stop the scheduler pod:

kubectl delete pod -n kube-system <scheduler-pod-name>

**Check Scheduler Status**: Verify if the scheduler pod is down:

kubectl get pods -n kube-system -l component=kube-scheduler

**Check Scheduler Logs**: View the scheduler logs for errors:

kubectl logs -n kube-system <scheduler-pod-name>

**Verify Resource Availability**: Check the available resources on the nodes:

kubectl describe node <node-name>

**Start Scheduler**: If needed, restart the scheduler pod:

kubectl delete pod -n kube-system <scheduler-pod-name>