FENIL GAJJAR

# KUBERNETES DAILY TASKS

## KUBERNETES ARCHITECTURE

- COMPREHENSIVE GUIDE
- THEORY + PRACTICAL
- REAL TIME SCENARIO TASKS
- DAILY TASKS SERIES

Unlock the core of Kubernetes! 🔥 Explore its architecture and discover how it powers modern cloud-native applications

FOLLOW FOR MORE


CONTACT US

fenilgajjar.devops@gmail.com

# 🚀 Unveiling Kubernetes Architecture : The Blueprint of Modern Container Orchestration

# 🚀 Welcome Back to Our Kubernetes Journey! 🌍

First and foremost, **a massive thank you** to everyone who supported and engaged with my first post, *Introduction to Kubernetes*! Your feedback, comments, and enthusiasm have been truly motivating. This journey isn't just about learning Kubernetes—it's about **building a strong DevOps mindset and mastering the tools that shape the modern cloud-native world.**

Now, let's move forward to the **next crucial topic—Kubernetes Architecture!** 🏗️

Understanding how Kubernetes is structured is **essential for effectively deploying, managing, and scaling containerized applications**. In this post, we'll break down:

✅ The **key components** of Kubernetes and their roles

✅ How these components work together to ensure **high availability, scalability, and automation**

✅ Why this architecture is designed for **fault tolerance and resilience** in large-scale deployments

Whether you're a beginner or already familiar with Kubernetes, grasping its architecture will **set the foundation for deeper concepts** like networking, scheduling, and workload management.

So, get ready to explore **the powerhouse behind Kubernetes!** 💡 Let's dive in and unlock the secrets of how Kubernetes orchestrates applications at scale. 🔥

# 🔥 The Significance of Kubernetes Architecture in Modern DevOps

Kubernetes is more than just a container orchestration tool—its **architecture is the backbone of scalability, automation, and resilience** in modern cloud environments. Understanding this architecture is crucial because it directly impacts how organizations deploy, manage, and optimize their applications.

## 🚀 Why Does Kubernetes Architecture Matter?

🔹 **Scalability & Performance Optimization**

The distributed nature of Kubernetes allows seamless **horizontal scaling** of applications, ensuring workloads can handle increased demand dynamically.

🔹 **High Availability & Fault Tolerance**

Kubernetes follows a **decentralized control plane and worker node model**, ensuring that failures in one part of the system do not disrupt the entire application.

🔹 **Automation & Efficient Resource Management**

With components like the **Kube-scheduler, Controller Manager, and Kubernetes API Server**, workloads are **automatically scheduled, monitored, and optimized**, reducing manual intervention.

🔹 **Consistency Across Multi-Cloud & Hybrid Environments**

By abstracting infrastructure complexities, Kubernetes architecture enables applications to run **consistently across on-prem, cloud, and hybrid setups**, ensuring portability and flexibility.

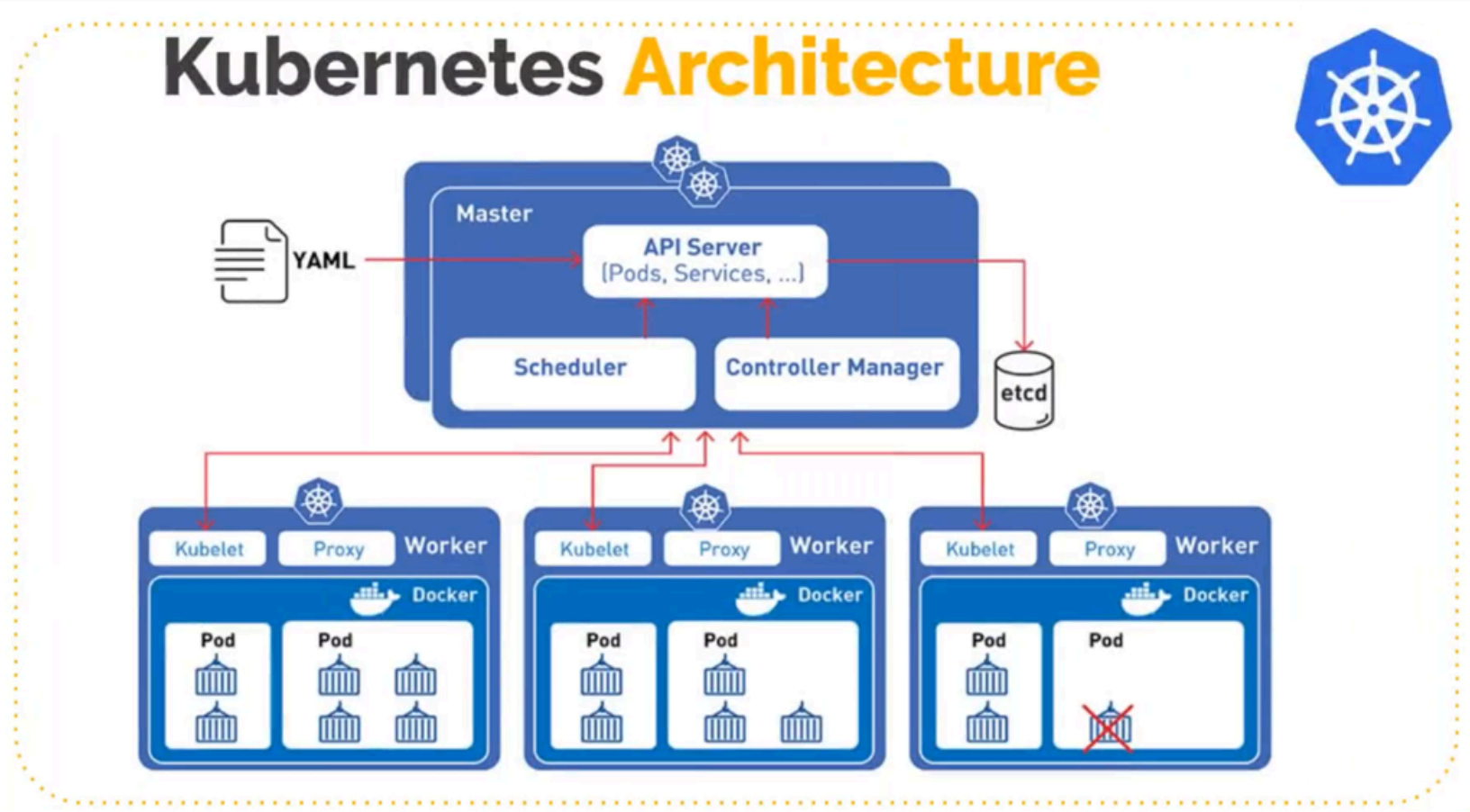
🔹 **Security & Role-Based Access Control (RBAC)**

A well-structured Kubernetes architecture ensures **secure cluster communication, fine-grained access controls, and robust security policies**, safeguarding applications against potential vulnerabilities.

## 🌍 Kubernetes: The Core of Cloud-Native Innovation

A deep understanding of **Kubernetes architecture empowers DevOps teams** to design **scalable, resilient, and automated** deployment strategies, making it an **indispensable tool for cloud-native and enterprise-grade applications.**

# 🏗️ Kubernetes Architecture: The Foundation of Scalable & Resilient Deployments 🚀



# 📌 Kubernetes Architecture Overview

Kubernetes follows a distributed architecture with two main categories of components:

1. **Control Plane (Master Node) →** Responsible for cluster management, scheduling, and monitoring.
2. **Worker Nodes →** Run the actual application workloads inside **Pods (containers)**.

Both these components communicate over a **network layer** using Kubernetes APIs.

# 1️⃣ Control Plane (Master Node)

The **Control Plane** is responsible for making global decisions about the cluster. It maintains the desired state, schedules workloads, and ensures self-healing.

## Components of the Control Plane

### 1. API Server (`kube-apiserver`)

- Acts as the **entry point** for all Kubernetes operations.
- Exposes the **Kubernetes API**, which users, controllers, and worker nodes interact with.
- Accepts REST API requests (`kubectl` commands) and validates them.
- Updates the **cluster state** in **ETCD** and retrieves information when needed.

- Handles **authentication and authorization** using Role-Based Access Control (RBAC).

**Example:**

When you run `kubectl apply -f deployment.yaml`, the request goes to the API Server, which processes and updates the cluster state.

### 2. ETCD (Key-Value Store)

- A **highly available, distributed database** that stores the entire Kubernetes cluster state.
- Stores configurations, secrets, and information about all Kubernetes objects (Pods, Deployments, Services, etc.).
- Runs on the Control Plane and must be backed up regularly to prevent data loss.

**Example:**

If a node crashes, Kubernetes retrieves the last known state from **ETCD** and recreates the workloads.

### 3. Scheduler (`kube-scheduler`)

- Determines **which worker node** should run a new Pod.
- Monitors resource availability (CPU, memory) on worker nodes.
- Uses different scheduling policies like **Affinity, Anti-Affinity, Taints, and Tolerations**.
- Ensures workloads are evenly distributed across the cluster.

**Example:**

If a new Pod needs to be created, the Scheduler checks available nodes and assigns the Pod to the **best-suited node** based on CPU, memory, and constraints.

### 4. Controller Manager (`kube-controller-manager`)

- Runs multiple **controller processes** to maintain the desired state of the cluster.
- Controllers watch the cluster state and take corrective actions if necessary.

#### Types of Controllers

- **Node Controller:** Detects node failures and replaces them.
- **Replication Controller:** Ensures the correct number of Pods are running.
- **Service Account & Token Controller:** Manages authentication tokens for service accounts.

**Example:**

If a node goes down, the Node Controller detects the failure and schedules new Pods on a healthy node.

### 5. Cloud Controller Manager (Optional)

- Only required when running Kubernetes in a cloud environment (AWS, Azure, GCP).
- Integrates Kubernetes with **cloud services** like Load Balancers, Volumes, and Network Policies.

# 2️⃣ Worker Node Components

Worker Nodes are responsible for running application workloads inside **Pods**. Each Worker Node consists of multiple components to ensure efficient container execution and networking.

## Components of a Worker Node

### 1. Kubelet

- An agent running on every worker node that **communicates with the Control Plane**.
- Ensures that the containers in Pods are running as expected.
- Receives instructions from the API Server and executes them locally.
- Monitors the health of running Pods and restarts them if they fail.

**Example:**
If a Pod crashes, the Kubelet reports this failure to the API Server, which triggers a new Pod creation.

### 2. Container Runtime

- Responsible for running containers inside Pods.
- Kubernetes supports multiple container runtimes:
    - **Docker**
    - **containerd**
    - **CRI-O**

- The container runtime pulls container images, starts containers, and manages networking.

**Example:**
When a Pod is scheduled, the container runtime fetches the required container image from a container registry (Docker Hub, AWS ECR) and starts it.

**3. Kube-Proxy**

- Manages network communication between different Pods, services, and external users.
- Implements **iptables or IPVS rules** to enable service discovery and load balancing.
- Ensures that traffic is correctly routed between Pods across different worker nodes.

**Example:**
When a user accesses an application, Kube-Proxy routes the request to the correct **Pod** behind a **Service**.

# 3️⃣ Kubernetes Networking

Kubernetes **networking model** ensures seamless communication between Pods, Services, and external users.

**Types of Networking in Kubernetes**

1. **Pod-to-Pod Communication**
    - Every Pod gets a **unique IP address** (from a network overlay like Flannel or Calico).
    - Pods can communicate directly within a namespace.
2. **Service Networking**
    - A **Service** provides a **stable IP address** and **load balancing** for a set of Pods.
    - Used when multiple replicas of a Pod are running behind a common endpoint.
3. **Ingress Networking**
    - Exposes services to external users using a **reverse proxy (NGINX, Traefik, HAProxy)**.
    - Allows **host-based and path-based routing** for multiple services.

**Example:**

- If `frontend` needs to communicate with `backend`, it does so through a **Service**.
- If an external user wants to access `myapp.com`, an **Ingress Controller** routes traffic to the correct service.

# 4️⃣ Kubernetes Objects

Kubernetes manages applications using different **objects**, which define how workloads should run.

**Important Kubernetes Objects**

- **Pod:** Smallest unit in Kubernetes, runs containers.
- **Deployment:** Manages the desired number of Pod replicas and allows rolling updates.
- **Service:** Provides networking and load balancing for Pods.
- **ConfigMap & Secret:** Stores environment variables and sensitive data.
- **PersistentVolume & PersistentVolumeClaim:** Provides persistent storage for stateful applications.

**Example:**

- A **Deployment** ensures that 3 replicas of an application are always running.
- A **Service** exposes the Deployment to users.

# 5️⃣ How Kubernetes Works - Step-by-Step

1. **User Deploys an Application**
    - The user runs `kubectl apply -f deployment.yaml`.
    - The request is sent to the **API Server**, which stores the configuration in **ETCD**.

2. **Scheduler Assigns Workloads**
    - The **Scheduler** selects the best worker node to run the Pod.
3. **Pods are Created and Managed**
    - The **Kubelet** on the worker node pulls the container image and starts the Pod.
4. **Networking is Set Up**
    - **Kube-Proxy** routes network traffic between Pods and Services.
5. **Scaling and Self-Healing**
    - If demand increases, **Auto-scaling** creates more Pods.
    - If a Pod crashes, **Kubelet** restarts it.

# Kubernetes Architecture: A Storytelling Way

## The Story: Kubernetes Island

Imagine a bustling **island called Kubernetes**, where various **villages (applications)** live in harmony. The island is run by a highly organized government and a team of tireless workers who ensure everything functions smoothly. This island is magical, as it can expand or shrink to accommodate more villagers whenever needed.

Let's explore how Kubernetes Island works, its governance, and its citizens:

## 1. The Central Government: Control Plane

The **Control Plane** is the central government of Kubernetes Island. It ensures that every village is running as planned, new villages are built properly, and resources are allocated efficiently.

### Key Members of the Government

1. **API Server (The Town Hall)**:
    - The **API Server** is the town hall where villagers and external visitors (users) make their requests.
    - If you want to build a new village or add more houses, you visit the town hall and submit your plans.
    - Example: A farmer submits a request to grow crops (deploy a pod). The town hall processes and approves the request.

2. **Scheduler (The Builder)**:

    - The **Scheduler** is like a skilled builder who decides where new villages (pods) will be constructed.
    - It checks the available space and resources on the island and chooses the best spot for the new village.
    - Example: If you request to build three houses (pods) for your workers, the builder ensures they're placed in areas with enough food, water, and land.

3. **Controller Manager (The Overseer)**:

    - The **Controller Manager** is the overseer who ensures that every village matches its blueprint.
    - If a house collapses in a village, the overseer ensures it is rebuilt immediately.
    - Example: If you ask for 3 houses in a village and one gets destroyed, the overseer makes sure it is replaced.

4. **etcd (The Record Keeper)**:

    - **etcd** is the magical record keeper who remembers everything about the island—every village, house, and road.
    - If the government forgets something, it checks with the record keeper.
    - Example: If someone asks, "How many villages exist on the island?" etcd provides the exact number.

## 2. The Tireless Workers: Worker Nodes

The **Worker Nodes** are the hardworking citizens of Kubernetes Island. These nodes perform all the actual tasks to keep the villages thriving.

**Key Components of a Worker Node**

1. **Kubelet (The Manager)**:
    - The **Kubelet** is the manager on each worker node. It ensures the workers (containers) do their jobs properly and reports back to the government.
    - Example: If a farmer is supposed to plant crops (run a container), the manager ensures the farmer does the task as expected.
2. **Kube-proxy (The Postmaster)**:
    - The **Kube-proxy** is like the island's postmaster. It manages communication between villages (pods) and ensures messages are delivered.
    - Example: If one village needs tools from another, the postmaster routes the request to the right place.
3. **Container Runtime (The Workers)**:
    - The **Container Runtime** is the workforce that does the actual job. These are the farmers, carpenters, and fishermen (containers) working hard to sustain the village.

    - Example: The farmer plants crops, the carpenter builds houses, and the fisherman catches fish—these are the containers running inside pods.

**3. The Island's Roads: Networking**

The villages are connected by an intricate network of roads. These roads allow villagers to visit each other and share resources. Kubernetes uses a **flat networking model**, ensuring all villages can communicate seamlessly.

- **Villages to Villages**: Pods communicate directly with each other using their addresses.
- **Villages to Visitors**: Services act as a gateway, allowing external visitors to interact with villages.

Example: A village hosting a marketplace (web app) sets up a road (service) to allow tourists (users) to visit and buy goods.

**4. The Magic: Self-Healing and Scalability**

Kubernetes Island is magical because it can **heal itself** and **grow or shrink** as needed.

- **Self-Healing**:
    - If a storm destroys a house (pod), the overseer (Controller Manager) rebuilds it.
    - Example: If the database pod crashes, Kubernetes automatically creates a new one.

- **Scalability**:
    - During a festival, the island needs more workers. Kubernetes adds new worker nodes to handle the increased workload.
    - Example: During Black Friday, a web application scales up to handle millions of visitors and scales down afterward to save resources.

**5. The Builders' Blueprint: Declarative Approach**

On Kubernetes Island, everything operates using blueprints (YAML files). Instead of micromanaging, you simply provide a plan, and the government ensures it is followed.

Example:

- You submit a plan stating: "I want 3 houses in Village A."
- Kubernetes takes care of building and maintaining those 3 houses.

## Now, let's dive deeper into each component and explore how they work together to power Kubernetes! 🚀🔍

# Kube-API Server in Kubernetes

## What is Kube-API Server?

The **kube-apiserver** is the **core component** of the Kubernetes control plane. It acts as a **gateway for all communication** in the cluster and is responsible for **handling REST API requests**, **validating them**, and **coordinating the cluster state**.

✅ **It is the only component that directly interacts with the etcd database**, which stores the cluster state.
✅ **All internal and external Kubernetes operations go through the API server**.

## How Kube-API Server Works?

1. **Receives Requests** – Users (kubectl, dashboard, or other tools) and internal components (controller manager, scheduler, etc.) send API requests.
2. **Authentication & Authorization** – Checks if the request is from a valid user and whether they have permission.
3. **Validation & Admission Control** – Ensures the request follows Kubernetes rules before applying changes.
4. **Updates etcd** – If valid, the API server updates etcd with the new cluster state.
5. **Sends Responses** – Returns the updated state to the requester and informs other control plane components.

## Key Responsibilities of Kube-API Server

🔹 **Handles All Kubernetes API Requests** – From kubectl, controllers, web dashboards, and external tools.

🔹 **Authenticates & Authorizes Users** – Ensures security through authentication (tokens, certificates, etc.) and authorization (RBAC, ABAC, etc.).

🔹 **Validates Objects Before Storing** – Ensures YAML manifests and configurations are correct before applying.

🔹 **Acts as a Frontend for etcd** – The only component that reads and writes to the etcd database.

🔹 **Communicates with Control Plane Components** – Passes information to the scheduler, controller manager, and other components.

## How Kube-API Server Interacts with Other Components?

**1. With etcd (Database)**

- **Reads cluster state** (e.g., checking if a Pod exists).
- **Writes updates** when new resources are created or changed.

**2. With Kube-Scheduler**

- The API server **notifies the scheduler** when a new Pod is created but has no assigned node.
- The scheduler **decides the best node** and updates the API server.

**3. With Controller Manager**

- Controllers (e.g., ReplicaSet, Deployment) **watch the API server** for changes.
- If a Pod crashes, the controller requests the API server to create a replacement.

**4. With Kubelet (Node Agent)**

- The API server **sends Pod definitions** to kubelet on worker nodes.
- Kubelet **reports back** on the Pod's health and status.

## Kube-API Server Architecture & Internals

**1. Request Handling Pipeline**

1. **Authentication** – Verifies user identity (certificates, tokens, etc.).
2. **Authorization** – Checks if the user has permission (RBAC, ABAC, etc.).
3. **Admission Controllers** – Validate and mutate requests before storing them.
4. **Object Validation & Schema Checks** – Ensures objects are well-formed.
5. **Persistence in etcd** – Stores the final state of the cluster.

**2. API Server Internals**

- **Runs as a container inside the control plane node**.
- **Written in Go**, designed as a RESTful API server.
- **Handles high traffic efficiently** using caching and watch mechanisms.

## Example: How Kube-API Server Processes a Request?

📌 **Scenario**: A user creates a new Pod using kubectl.

sh

CopyEdit

```
kubectl run mypod --image=nginx
```

🔹 **Step-by-step flow**:

1️⃣ **kubectl sends a REST API request** to kube-apiserver.

2️⃣ **API server authenticates & authorizes** the request.

3️⃣ **Admission controllers validate the request** (e.g., checking resource limits).

4️⃣ **API server writes the Pod definition to etcd**.

5️⃣ **Scheduler detects the new Pod** and assigns it to a node.

6️⃣ **Kubelet on that node pulls the image and starts the Pod**.

## How to Interact with Kube-API Server?

1️⃣ **Check API Server Status**

```
kubectl get componentstatuses
```

2️⃣ **Access API Directly (Without Kubectl)**

```
curl -k https://localhost:6443/version
```

3️⃣ **List Available API Resources**

```
kubectl api-resources
```

4️⃣ **Get Cluster State in JSON Format**

```
kubectl get pods -o json
```

## Security Features of Kube-API Server

- 🔹 **TLS Encryption** – Secures all communication between components.
- 🔹 **RBAC (Role-Based Access Control)** – Defines user permissions.
- 🔹 **Webhook Authentication & Authorization** – Allows external authentication systems.
- 🔹 **Audit Logging** – Tracks all API requests for security monitoring.

## High Availability of Kube-API Server

In production, the API server should be **highly available** by running multiple instances behind a **load balancer**.

- 🔹 **Single-Node Cluster (Not Recommended for Production)**
  - One API server instance running on a control plane node.

- **Multi-Master Cluster (HA Setup)**
  - Multiple API servers run on different control plane nodes.
  - A **load balancer** distributes traffic between them.

# 📌 etcd in Kubernetes

## 🔹 What is etcd?

`etcd` is a **distributed key-value store** used by Kubernetes to store **all cluster data**, including:

✅ Cluster state (nodes, pods, deployments, etc.)

✅ Configuration data (ConfigMaps, Secrets)

✅ Networking details (Services, Endpoints)

✅ Authentication & authorization info (RBAC, users, roles)

It's a **highly available, strongly consistent** database based on the **Raft consensus algorithm**.

## 🔹 Why is etcd Important in Kubernetes?

Kubernetes **completely relies on etcd** to function.

- If etcd goes **down**, **the Kubernetes API server stops working** (since it can't read/write data).
- If etcd **loses data**, **the entire cluster state is lost**.

## 🔹 How etcd Works in Kubernetes?

1️⃣**kube-apiserver** is the only component that interacts with etcd.

2️⃣**Any change in Kubernetes** (e.g., creating a Pod) is sent to the API server.

3️⃣The API server **writes the change to etcd**.

4️⃣ Other components (kube-scheduler, controllers) read changes from etcd and act accordingly.

## 🔹 etcd Data Structure

`etcd` stores data in a simple **key-value format**. Example:

| Key | Value |
| --- | --- |
| `/registry/pods/default/my-app-pod` | Pod details (YAML) |
| `/registry/nodes/ip-10-0-0-1` | Node details |
| `/registry/configmaps/my-config` | ConfigMap data |

To view etcd data, you can use:

```
ETCDCTL_API=3 etcdctl get / --prefix --keys-only
```

## 🔹 High Availability (HA) Setup for etcd

In production, etcd is **deployed in a cluster** for **fault tolerance**.

- A **minimum of 3 nodes** is recommended (always use an **odd number** to avoid split-brain issues).
- etcd uses the **Raft algorithm** to maintain consistency.

**Example etcd Cluster Setup**

- **3 etcd nodes running on control plane nodes**
- etcd config file (`/etc/etcd/etcd.conf.yaml`):

```
name: etcd-1

initial-cluster:
etcd-1=https://10.0.0.1:2380,etcd-2=https://10.0.0.2:2380,etcd-3=https://10.0.0.3:2380

initial-cluster-state: new

listen-peer-urls: https://10.0.0.1:2380

listen-client-urls: https://10.0.0.1:2379
```

- If one etcd node **fails**, others can still **maintain cluster state**.

## 🔹 Best Practices for etcd in Kubernetes

✅ **Always take etcd backups**

```
ETCDCTL_API=3 etcdctl snapshot save /backup/etcd-backup.db
```

✅ **Run etcd on separate dedicated nodes** (if possible)
✅ **Use TLS encryption** for etcd communication
✅ **Monitor etcd health using Prometheus & Grafana**

## 🔹 How to Backup & Restore etcd in Kubernetes 🚀

Taking regular etcd **backups** is crucial for disaster recovery. Here's how you can **backup and restore etcd** step by step.

## ✅ Step 1: Set Environment Variables

Set up environment variables for **etcdctl** (Run this on a control plane node):

```
export ETCDCTL_API=3

export ETCDCTL_ENDPOINTS="https://127.0.0.1:2379"

export ETCDCTL_CACERT="/etc/kubernetes/pki/etcd/ca.crt"

export ETCDCTL_CERT="/etc/kubernetes/pki/etcd/server.crt"

export ETCDCTL_KEY="/etc/kubernetes/pki/etcd/server.key"
```

## ✅ Step 2: Take a Backup of etcd

Run this command to save a snapshot of etcd:

```
etcdctl snapshot save /backup/etcd-backup.db
```

Verify the backup file:

```
ls -lah /backup/etcd-backup.db
```

Check snapshot status:

```
etcdctl snapshot status /backup/etcd-backup.db
```

## ✅ Step 3: Restore etcd from Backup

If your etcd data is lost or corrupted, you can restore it from a backup.

**1️⃣ Stop the etcd Service**

On all control plane nodes, stop etcd before restoring:

```
sudo systemctl stop etcd
```

### 2️⃣ Restore the etcd Snapshot

Run this command to restore from backup:

```
etcdctl snapshot restore /backup/etcd-backup.db --data-dir
/var/lib/etcd-new
```

### 3️⃣ Replace Old Data with Restored Data

Move the old etcd data out and replace it with the new data:

```
sudo mv /var/lib/etcd /var/lib/etcd-old

sudo mv /var/lib/etcd-new /var/lib/etcd
```

### 4️⃣ Start etcd Service Again

Restart etcd to load the restored data:

```
sudo systemctl start etcd
```

## ✅ Step 4: Verify the etcd Cluster

Check if etcd is running properly:

```
etcdctl endpoint health
```

Check etcd members:

```
etcdctl member list
```

## ✅ Automating etcd Backups with a Cron Job

You can schedule automatic etcd backups with a cron job:

```
sudo crontab -e
```

Add this line to **run a backup every night at 2 AM**:

```
0 2 * * * etcdctl snapshot save /backup/etcd-$(date
+\%Y\%m\%d).db
```

## 🔹 Kube-Scheduler in Kubernetes 🚀

### ✅ What is Kube-Scheduler?

Kube-Scheduler is a **control plane component** responsible for **assigning pods to worker nodes** in a Kubernetes cluster. It **does not create pods**, but **decides where they should run** based on available resources, constraints, and scheduling policies.

## 🔹 How Kube-Scheduler Works

1️⃣ **Pod Creation** ➜ A pod without a node assignment (`.spec.nodeName` is empty) is detected.
2️⃣ **Filtering Nodes** ➜ It filters out nodes that **don't meet pod requirements**.
3️⃣ **Scoring Nodes** ➜ It ranks the remaining nodes based on a **scoring system**.
4️⃣ **Assigning a Node** ➜ The highest-ranked node is selected.
5️⃣ **Binding the Pod** ➜ Kube-Scheduler **informs the API server**, which updates the pod's `nodeName`.

## 🔹 Scheduling Process in Detail

### 1️⃣ Filtering (Predicates) 🛑

The scheduler first eliminates nodes that **don't meet the pod's requirements**:
✅ **Taints & Tolerations** – Does the node allow the pod to run?
✅ **Node Selectors & Affinity Rules** – Does the pod prefer a specific node?

✅ **Resource Availability** – Does the node have enough CPU & memory?

✅ **Pod Anti-Affinity** – Does the pod avoid certain nodes?

✅ **Node Conditions** – Is the node in a healthy state?

### 2️⃣Scoring (Prioritization) ⭐

After filtering, the scheduler scores nodes based on different criteria:

✅ **Least Requested Resources** – Prefers nodes with the most free resources.

✅ **Node Affinity** – Prefers nodes matching a pod's affinity rules.

✅ **Spread Constraints** – Distributes pods evenly across zones/nodes.

✅ **Custom Plugins** – Users can add custom scheduling logic.

### 3️⃣Binding the Pod to a Node 🔗

After selecting the best node, Kube-Scheduler **updates the API server**, setting the pod's `.spec.nodeName`. The kubelet on that node then pulls the pod's container images and starts the pod.

## 🔹 Example: How Kube-Scheduler Works in a Real Scenario

Let's say you create a pod:

```
apiVersion: v1

kind: Pod

metadata:
```

```
  name: my-app
spec:
  containers:
    - name: my-container
      image: nginx
  nodeSelector:
    environment: production  # Scheduler will look for nodes with this label
```

- The **scheduler checks available nodes** with the `environment=production` label.
- If multiple nodes match, it scores them **based on available CPU, memory, and affinity rules**.
- The **best node is chosen**, and the pod is assigned to it.

## 🔹 Advanced Kube-Scheduler Features

### ✅ Node Affinity (Preferred Nodes)

Example: Schedule a pod on nodes with `disktype=ssd`:

```
affinity:
  nodeAffinity:
```

```
    requiredDuringSchedulingIgnoredDuringExecution:
      nodeSelectorTerms:
      - matchExpressions:
        - key: disktype
          operator: In
          values:
          - ssd
```

### ✅ Taints & Tolerations (Controlling Pod Placement)

Taint a node:

```
kubectl taint nodes node1 key=value:NoSchedule
```

Pod toleration:

```
tolerations:
  - key: "key"
    operator: "Equal"
    value: "value"
    effect: "NoSchedule"
```

✅ **Custom Schedulers**

You can run your own scheduler instead of Kube-Scheduler:

```
kubectl get pods --all-namespaces -o wide | grep scheduler
```

## 🔹 High Availability for Kube-Scheduler

- **Run multiple schedulers** (in an HA control plane).
- **Use leader election** to avoid conflicts.
- **Monitor scheduler health** with Prometheus & logs.

# Kube-Controller Manager in Kubernetes

## 🔹 What is Kube-Controller Manager?

The **kube-controller-manager** is a core component of Kubernetes that **runs multiple controllers in a single process**. These controllers **watch the cluster state** and ensure that the desired state (from manifests) matches the actual state.

📌 **Example:** If a Pod crashes, the **ReplicaSet controller** automatically creates a new Pod.

## 🔹 Key Responsibilities of Kube-Controller Manager

The kube-controller-manager **runs multiple built-in controllers**, each responsible for different cluster resources.

### 1️⃣Node Controller

- Monitors the health of worker nodes
- If a node becomes unresponsive, it marks the node as **NotReady**
- If a node remains unresponsive for a long time, it **removes Pods from that node**

📌 **Example:**
If a worker node stops responding for **5 minutes**, the Node Controller removes the Pods running on that node.

### 2️⃣ Replication Controller

- Ensures that the desired number of Pod replicas are always running
- Works with **ReplicaSet**, **Deployment**, **DaemonSet**, etc.

📌 **Example:**

If a ReplicaSet defines **3 replicas**, but one Pod crashes, the controller **automatically creates a new Pod**.

### 3️⃣ Endpoint Controller

- Manages the **Endpoints** object, which tracks the IPs of Pods behind a Service
- Ensures Services correctly route traffic to healthy Pods

📌 **Example:**

If a Pod crashes, the Endpoint Controller **removes its IP from the Service** and updates the list of healthy endpoints.

### 4️⃣ Service Account & Token Controller

- Creates **default ServiceAccounts** for every new Namespace
- Generates API tokens for Pods to communicate with the Kubernetes API

📌 **Example:**

Every new Namespace automatically gets a `default` ServiceAccount.

### 5️⃣ Persistent Volume Controller

- Manages Persistent Volumes (PVs) and Persistent Volume Claims (PVCs)
- Binds PVs to PVCs and ensures storage is allocated correctly

📌 **Example:**

If a user requests 10GB storage (PVC), the controller binds it to an available PV.

### 6️⃣ Job Controller

- Manages **Job** and **CronJob** resources
- Ensures that Jobs run to completion

📌 **Example:**

If a **batch job** is supposed to run once, the Job Controller ensures it completes successfully.

### 🔹 How Kube-Controller Manager Works?

1. Watches cluster objects (**Nodes, Pods, Services, etc.**)
2. Detects **differences** between desired and actual state
3. Takes **corrective action** using built-in controllers

📌 **Example Flow:**

- A Deployment specifies **5 Pods**
- One Pod crashes ➜ The Replication Controller **detects it**
- A new Pod is **automatically created** to maintain 5 Pods

## 🔹 High Availability (HA) for Kube-Controller Manager

Since it is a critical component, HA is **recommended**.

✅ **How to enable HA?**

- Run **multiple kube-controller-manager instances** across control plane nodes
- Use **leader election** (only one active instance at a time)
- If the active one fails, another takes over

📌 **Check active leader:**

```
kubectl get endpoints kube-controller-manager -n kube-system
```

## 🔹 Deploying Kube-Controller Manager Manually (Self-Managed Cluster)

If you're running a **self-managed Kubernetes cluster**, you can manually set up kube-controller-manager.

1️⃣ **Install kube-controller-manager**

Download and install Kubernetes binaries:

```
wget
https://dl.k8s.io/release/v1.28.0/bin/linux/amd64/kube-contr
oller-manager

chmod +x kube-controller-manager

mv kube-controller-manager /usr/local/bin/
```

### 2️⃣ Create a systemd Service for Kube-Controller Manager

Create a service file `/etc/systemd/system/kube-controller-manager.service`:

```
[Unit]

Description=Kubernetes Controller Manager

After=network.target


[Service]

ExecStart=/usr/local/bin/kube-controller-manager \

  --bind-address=0.0.0.0 \

  --cluster-signing-cert-file=/etc/kubernetes/pki/ca.crt \

  --cluster-signing-key-file=/etc/kubernetes/pki/ca.key \

  --kubeconfig=/etc/kubernetes/controller-manager.kubeconfig
\
```

```
  --leader-elect=true
Restart=always

[Install]
WantedBy=multi-user.target
```

Start the service:

```
systemctl daemon-reload
systemctl enable kube-controller-manager
systemctl start kube-controller-manager
```

📌 **Note:** In a managed Kubernetes cluster (e.g., EKS, AKS, GKE, kops), this setup is handled **automatically**.

In Kubernetes, **each controller** (like Node Controller, ReplicaSet Controller, etc.) is **logically a separate function** that manages different resources.

However, **instead of running each controller as a separate process**, Kubernetes **combines all controllers into a single program** called the **kube-controller-manager**. This helps reduce complexity and resource usage.

📌 **Simple Analogy:**
Think of each controller as a different worker in a company. Instead of having

separate offices for each worker, they all work in **one office (kube-controller-manager)** to make communication and management easier.

📌 **Why is this done?**

- **Less overhead** (running multiple processes would be inefficient)
- **Easier to manage** (all controllers run under one service)
- **Uses leader election** (only one instance actively makes decisions)

## 📌 List of Controllers in Kube-Controller Manager

The **kube-controller-manager** runs multiple controllers, each responsible for managing different resources in the Kubernetes cluster.

### 1️⃣ Node Controller

- Monitors the health of worker nodes
- Marks unresponsive nodes as **NotReady**
- Deletes Pods from a failed node after a timeout

📌 **Example:**

If a node stops responding for **5 minutes**, the controller removes its Pods.

### 2️⃣ Replication Controller

- Ensures the desired number of Pod replicas are always running
- Works with **ReplicaSet, Deployment, DaemonSet**

📌 **Example:**

If a ReplicaSet defines **3 Pods**, and one crashes, it creates a new Pod.

### 3️⃣ Endpoints Controller

- Updates the **Endpoints** object to keep track of healthy Pod IPs
- Ensures Services route traffic to the correct Pods

📌 **Example:**

If a Pod behind a Service crashes, the controller updates the Endpoint list.

### 4️⃣ Service Account & Token Controller

- Creates a **default ServiceAccount** for each Namespace
- Generates API tokens for Pods to authenticate with the API server

📌 **Example:**

When a new Namespace is created, a `default` ServiceAccount is automatically added.

### 5️⃣ Persistent Volume (PV) Controller

- Manages **Persistent Volume (PV) and Persistent Volume Claim (PVC)** bindings
- Handles dynamic storage provisioning

📌 **Example:**

If a user requests **10GB storage**, the controller binds a PV to the PVC.

### 6️⃣ Job Controller

- Ensures that **Jobs** and **CronJobs** run to completion
- Restarts failed Jobs if necessary

📌 **Example:**

If a batch job is scheduled to **run once**, the controller ensures it completes.

### 7️⃣ Garbage Collector Controller

- Cleans up unused or orphaned Kubernetes objects
- Detects and removes **dangling Pods, Deployments, or Services**

📌 **Example:**

If a Deployment is deleted, its ReplicaSet and Pods are also removed.

### 8️⃣ Namespace Controller

- Deletes **all resources inside a Namespace** when the Namespace is deleted

📌 **Example:**

If `dev-namespace` is deleted, all Pods, Services, and Secrets inside it are also removed.

### 9️⃣ Horizontal Pod Autoscaler (HPA) Controller

- Adjusts the number of Pods based on CPU/memory usage
- Works with **metrics-server**

📌 **Example:**

If CPU usage exceeds **80%**, it scales Pods **from 2 to 4**.

### 🔟 Certificate Signing Controller

- Manages **certificate requests** for TLS security
- Signs **Kubernetes API server certificates**

📌 **Example:**

If a new worker node joins the cluster, this controller helps in securing communication.

### 1️⃣1️⃣ Cloud Controller Manager (for Cloud Providers)

- Manages Kubernetes integration with cloud providers
- Handles **Load Balancers, Volume Attachments, and Node Health**

📌 **Example:**

If using AWS, this controller automatically provisions an **ELB (Elastic Load Balancer)**.

# Cloud Controller Manager (CCM) in Kubernetes

## 📌 What is Cloud Controller Manager (CCM)?

The **Cloud Controller Manager (CCM)** is a Kubernetes component that allows the cluster to **interact with the underlying cloud provider** (AWS, GCP, Azure, etc.). It is responsible for managing cloud-specific operations like:

✅ Creating **Load Balancers**
✅ Attaching **Persistent Volumes**
✅ Managing **Nodes (VMs/Instances)**
✅ Handling **Networking (Routes & IPs)**

## 1️⃣ Why is CCM Needed?

Earlier, cloud provider-specific code was embedded in the **kube-controller-manager**, which made Kubernetes heavily dependent on cloud platforms.

To **separate cloud-specific logic**, Kubernetes introduced CCM, which allows:

✔️ **Decoupling Kubernetes from cloud providers**
✔️ **Easier Kubernetes updates** without worrying about cloud changes
✔️ **Running Kubernetes on-premise** without unnecessary cloud logic

## 2️⃣ Key Responsibilities of Cloud Controller Manager

🛠️ **CCM runs multiple controllers to manage cloud interactions:**

1️⃣ **Node Controller**

📌 **Manages cloud-based Nodes (VMs/Instances)**

- Checks if a Node (VM) is still active in the cloud
- If a Node is deleted in the cloud, the controller removes it from Kubernetes

✅ **Example:**

If an EC2 instance is manually terminated in AWS, CCM automatically removes it from the cluster.

2️⃣ **Route Controller**

📌 **Manages routing between Kubernetes Nodes**

- Works in cloud environments that don't support automatic networking (e.g., GCE, AWS, Azure)
- Ensures correct IP routing between Pods and Nodes

✅ **Example:**

On AWS, it configures **VPC route tables** so Pods on different Nodes can communicate.

3️⃣ **Service Controller**

📌 **Manages Load Balancers for Services**

- Provisions **cloud-based Load Balancers** for `Service type: LoadBalancer`
- Updates Load Balancer with correct backend Pod IPs

✅ **Example:**

If you expose a Service in AWS with `type: LoadBalancer`, CCM provisions an **AWS Elastic Load Balancer (ELB)** and links it to the Service.

4️⃣ **Volume Controller**

📌 **Handles Persistent Volume (PV) provisioning**

- Works with cloud-based storage (EBS, GCE Persistent Disk, Azure Disk)
- Attaches & detaches volumes to Kubernetes Nodes

✅ **Example:**

If a user requests **50GB storage** via a PersistentVolumeClaim (PVC), CCM provisions an **AWS EBS Volume** and attaches it to the Pod.

## 3️⃣ How Cloud Controller Manager Works?

1️⃣ When a **new Node (VM)** is created, CCM communicates with the cloud provider to check if the Node exists and fetches its details.

2️⃣ If a Service with `type: LoadBalancer` is created, CCM provisions a **cloud-based Load Balancer** and assigns it a public IP.

3️⃣ If a Persistent Volume is requested, CCM provisions and attaches a **cloud storage volume** to the correct Node.

4️⃣ If a Node is deleted in the cloud, CCM removes it from Kubernetes.

# Where Are Kubernetes Control Plane Components Stored & Run?

All **control plane components** in Kubernetes—**kube-apiserver, etcd, kube-scheduler, kube-controller-manager, and cloud-controller-manager (if applicable)**—are typically **run as containers inside the control plane node**.

### How Are Control Plane Components Deployed?

#### 1️⃣ In Self-Managed Clusters (Kubeadm, KOPS, Bare Metal)

- These components run as **static pods** managed by **kubelet**.

The YAML manifests for these components are stored in:

```
/etc/kubernetes/manifests/
```

- Kubelet monitors these files and ensures the components are always running.

#### 2️⃣ In Managed Kubernetes Clusters (EKS, AKS, GKE, etc.)

- Control plane components are **fully managed by the cloud provider**.
- You don’t have direct access to them.
- They are deployed across multiple nodes for **high availability (HA)**.

## Control Plane Components & Their Storage/Execution Details

### 1️⃣ Kube-API Server

📌 **Function:** Handles all Kubernetes API requests.

- **Runs as a container inside the control plane node.**
- **Stores API data in etcd (database).**
- Manifests are in:

```
/etc/kubernetes/manifests/kube-apiserver.yaml
```

- Logs can be checked via:

```
journalctl -u kube-apiserver
```

### 2️⃣ etcd (Key-Value Store for Kubernetes Data)

📌 **Function:** Stores all cluster data (Pods, Deployments, ConfigMaps, etc.).

- **Runs as a container in control plane.**
- Data is stored in:

```
/var/lib/etcd/
```

- Uses **Raft consensus algorithm** for HA.
- Backup command:

```
etcdctl snapshot save /backup/etcd-snapshot.db
```

- Logs can be checked via:

```
journalctl -u etcd
```

### 3️⃣ Kube-Scheduler

📌 **Function:** Assigns Pods to Nodes based on resources & policies.

- **Runs as a container in control plane.**
- Manifests are in:

```
/etc/kubernetes/manifests/kube-scheduler.yaml
```

- Logs can be checked via:

```
journalctl -u kube-scheduler
```

### 4️⃣ Kube-Controller Manager

📌 **Function:** Runs multiple controllers (Node, Deployment, ReplicaSet, etc.).

- 🔹 **Runs as a container in control plane.**
- 🔹 Manifests are in:

```
/etc/kubernetes/manifests/kube-controller-manager.yaml
```

- 🔹 Logs can be checked via:

```
journalctl -u kube-controller-manager
```

### 5️⃣ Cloud Controller Manager (Optional, for Cloud Environments)

📌 **Function:** Manages cloud integrations (Load Balancers, Nodes, Storage).

- 🔹 **Runs as a container in control plane.**
- 🔹 Manifests (if self-managed) are in:

```
/etc/kubernetes/manifests/cloud-controller-manager.yaml
```

- 🔹 **Managed Kubernetes (EKS, GKE, AKS) already includes it.**

## Hidden but Important Facts We Should Know

1️⃣ **All these components run as containers inside the control plane node.**

You can verify using:

```
crictl ps
```

- This will show running control plane components as containerized workloads.

2️⃣ **They use host networking.**

- Unlike normal Pods, these containers run with `hostNetwork: true` so they can directly communicate with the system.

3️⃣ **Static Pods are managed by kubelet.**

- If the kube-apiserver crashes, kubelet automatically restarts it using the YAML from `/etc/kubernetes/manifests/`.

4️⃣ **No kube-proxy on control plane nodes.**

- kube-proxy (which manages network traffic between Nodes) usually runs only on worker nodes.

5️⃣ **Control plane components must be highly available.**

- In HA setups, control plane nodes are spread across multiple machines for fault tolerance.

## all control plane components run as static pods.

These include:

✅ **kube-apiserver**

✅ **etcd**

✅ **kube-scheduler**

✅ **kube-controller-manager**

✅ **cloud-controller-manager** (if applicable)

### Who Restarts Them if They Fail?

🛠️ **Kubelet is responsible for restarting static pods.**

- Static pods are **not managed by the Kubernetes API** (i.e., no Deployment or ReplicaSet controls them).
- Instead, **kubelet watches the static pod manifests stored in `/etc/kubernetes/manifests/`.**
- If a static pod **crashes or is deleted**, kubelet will automatically **restart it** using the manifest file.

### How Does Kubelet Manage Static Pods?

1️⃣ Kubelet runs on the control plane node.

2️⃣ It **monitors the YAML files** inside:

`/etc/kubernetes/manifests/`

3️⃣ If any static pod (like kube-apiserver) crashes or is removed, **kubelet detects it and starts it again**.

4️⃣ Kubelet uses the **container runtime (like containerd or Docker)** to restart the failed container.

**How to Check If Control Plane Pods Are Running?**

Run this command on the control plane node:

```
kubectl get pods -n kube-system
```

You'll see all control plane components running as pods in the `kube-system` namespace.

**How to Verify Static Pod Configurations?**

Check the static pod manifests stored in:

```
ls /etc/kubernetes/manifests/
```

You'll see files like:

```
kube-apiserver.yaml
```

```
etcd.yaml
```

```
kube-scheduler.yaml

kube-controller-manager.yaml
```

These files define the static pods.

**If a Control Plane Pod Keeps Crashing, What Should You Do?**

1️⃣ Check logs:

```
kubectl logs -n kube-system <pod-name>
```

2️⃣ Check kubelet logs for errors:

```
journalctl -u kubelet -f
```

3️⃣ Manually restart the kubelet service:

```
systemctl restart kubelet
```

4️⃣ If a static pod YAML file is missing or corrupted, restore it from a backup.

# Worker Node Components in Kubernetes

A **worker node** is responsible for running application workloads in a Kubernetes cluster. It contains the necessary components to execute pods and communicate with the control plane.

### Main Components of a Worker Node

### 1️⃣ Kubelet (Node Agent)

- **What it does?**
  - Communicates with the API server to receive pod definitions.
  - Ensures that the containers inside the pod are running as expected.
  - Restarts failed containers if needed.
  - Manages **static pods** on the node.
- **Key Responsibility:** Ensures that the node runs assigned workloads correctly.

### 2️⃣ Kube-Proxy (Networking Component)

- **What it does?**
  - Maintains network rules on the node for pod-to-pod and pod-to-service communication.
  - Uses **iptables or IPVS** to enable load balancing for services.
- **Key Responsibility:** Manages network communication inside the cluster.

### 3️⃣ Container Runtime (Docker, containerd, CRI-O, etc.)

- **What it does?**
    - Pulls container images from registries.
    - Creates, starts, stops, and manages containers inside pods.
- **Key Responsibility:** Runs application containers inside pods.

### 4️⃣ Pods (Application Workloads)

- **What they do?**
    - Each worker node runs **one or more pods**, which contain **one or more containers**.
    - These are the actual applications deployed in the cluster.
- **Key Responsibility:** Runs the application workloads inside containers.

# Kubelet in Kubernetes

## What is Kubelet?

Kubelet is an **agent running on each worker node** in a Kubernetes cluster. It is responsible for **managing the lifecycle of pods and containers** assigned to that node.

## Key Responsibilities of Kubelet

### 1️⃣ Registers the Node with the Control Plane

- When a new worker node joins the cluster, **Kubelet registers the node** with the API server.
- The API server then recognizes the node as **ready to run workloads**.

### 2️⃣ Communicates with the API Server

- Kubelet constantly **watches for pod assignments** from the API server.
- It **receives pod specifications (YAML)** and ensures that they run on the node.

### 3️⃣ Manages Pods and Containers

- **Ensures pods are running** as per the pod definition.
- If a pod crashes, Kubelet **automatically restarts** it.
- If a pod is deleted from the API server, Kubelet **removes it from the node**.

### 4️⃣ Interacts with the Container Runtime

- Kubelet **does not run containers directly**—it uses a **container runtime** (like Docker, containerd, or CRI-O).
- It sends requests to the runtime to:
  ✅ Pull images
  ✅ Create and start containers
  ✅ Stop or delete containers when no longer needed

5️⃣ **Monitors Node and Pod Health**

- Kubelet **collects resource usage** (CPU, memory, disk, network) and reports it to the API server.
- It performs **liveness and readiness probes** to check if a container is healthy.
- If a container fails multiple times, Kubelet can **restart the pod** or notify the API server.

6️⃣ **Handles Static Pods (If Configured)**

- Kubelet **monitors** `/etc/kubernetes/manifests/` for **static pod YAMLs**.
- If it finds a new static pod definition, it **creates and runs the pod**.
- These pods are **not managed by the API server** but only by Kubelet.

## How Kubelet Works?

1️⃣ Kubelet **registers the node** with the Kubernetes API server.
2️⃣ It **watches the API server** for pod assignments.
3️⃣ When a pod is assigned to the node, Kubelet **instructs the container runtime** to run the pod.
4️⃣ It **continuously monitors the pod's health** and **restarts** failed pods if necessary.
5️⃣ Kubelet **reports the node's status** (CPU, memory, disk, network usage) to the

API server.

6️⃣ If the node fails, the API server stops scheduling new pods to that node.

## Where is Kubelet Installed?

- **Runs on every worker node** and control plane node (for static pods).
- Installed as a **binary** or as a **systemd service** (`/usr/bin/kubelet`).
- Can be configured via **`/var/lib/kubelet/config.yaml`**.

## Example Kubelet Command (Starting Kubelet)

On a worker node, Kubelet can be started with:

```
kubelet --kubeconfig=/etc/kubernetes/kubelet.conf \
        --config=/var/lib/kubelet/config.yaml \
        --container-runtime=remote \

--container-runtime-endpoint=unix:///run/containerd/containerd.sock
```

**Breakdown:**

✅ `--kubeconfig=/etc/kubernetes/kubelet.conf` → API server connection details

✅ `--config=/var/lib/kubelet/config.yaml` ➜ Kubelet config file

✅ `--container-runtime=remote` ➜ Uses an external container runtime

✅ `--container-runtime-endpoint=unix:///run/containerd/containerd.sock` ➜ Connects to containerd

## How to Check Kubelet Logs?

To troubleshoot Kubelet issues, check logs using:

```
journalctl -u kubelet -f
```

To restart Kubelet:

```
systemctl restart kubelet
```

### Why Does Kubelet Run on Control Plane Nodes?

Even though the control plane is mainly responsible for managing the cluster, it **also needs to run certain workloads**. This is why Kubelet is present on control plane nodes.

### 1️⃣ Runs Static Pods (like control plane components)

- Control plane components (**kube-apiserver, etcd, kube-scheduler, kube-controller-manager**) are **static pods**.
- Kubelet on the control plane node **monitors and ensures these static pods are always running**.

Static pod manifests are stored in:

```
/etc/kubernetes/manifests/
```

### 2️⃣ Allows Control Plane Nodes to Run Regular Pods (Optional)

- If a cluster is small, **control plane nodes can also be used as worker nodes**.
- In such cases, Kubelet **allows scheduling of normal workloads** (pods) on control plane nodes.

To **allow workloads** on a control plane node, you need to remove the taint:

```
kubectl taint nodes <control-plane-node>
node-role.kubernetes.io/control-plane-
```

### 3️⃣ Health Monitoring and Communication

- Kubelet on the control plane nodes **reports node health** to the API server.
- If the control plane node has issues, Kubelet **informs the API server**, which can take necessary actions.

# Kube-Proxy in Kubernetes

## What is Kube-Proxy?

Kube-Proxy is a **networking component** that runs on every node in a Kubernetes cluster. It is responsible for **routing network traffic** between services and pods, ensuring seamless communication inside and outside the cluster.

## Key Responsibilities of Kube-Proxy

### 1️⃣ Manages Service Networking

- Kubernetes **services provide a stable IP** for accessing pods.
- Kube-Proxy **routes traffic to the correct pod behind the service**.

### 2️⃣ Implements Load Balancing

- When multiple pods back a service, Kube-Proxy **distributes incoming traffic** among them.
- Ensures **high availability** and **efficient load distribution**.

### 3️⃣ Handles Cluster IPs

- When a service is created, Kubernetes assigns a **ClusterIP**.
- Kube-Proxy ensures that any request to this IP **gets forwarded to the right pod**.

### 4️⃣ Supports Different Network Modes

Kube-Proxy operates in different **modes** for handling traffic:

✅ **iptables mode (Default)** – Uses Linux iptables to forward traffic efficiently.

✅ **IPVS mode (Better Performance)** – Uses Linux IP Virtual Server (IPVS) for load balancing.

✅ **Userspace mode (Deprecated)** – Uses a userspace process to forward traffic.

## How Kube-Proxy Works?

1️⃣ A service is created in Kubernetes.

2️⃣ Kube-Proxy **sets up networking rules** for that service using `iptables` or `IPVS`.

3️⃣ When a request comes to the service, Kube-Proxy **forwards it to an available pod**.

4️⃣ If a pod fails, Kube-Proxy **reroutes traffic** to healthy pods.

## Example: How Kube-Proxy Routes Traffic

**You create a service for a pod:**

```
apiVersion: v1

kind: Service

metadata:

  name: my-service

spec:

  selector:
```

```
    app: my-app
  ports:
    - protocol: TCP
      port: 80
      targetPort: 8080
```

1. **Kube-Proxy sets up networking rules**
    - Requests to `my-service:80` are forwarded to an available pod's port `8080`.

## Where is Kube-Proxy Installed?

- **Runs on every node (both worker and control plane nodes)**.
- Installed as a **DaemonSet** in `kube-system` namespace.

You can check its running status using:

```
kubectl get pods -n kube-system | grep kube-proxy
```

## How to Restart Kube-Proxy?

If Kube-Proxy crashes, restart it with:

```
systemctl restart kube-proxy
```

or

```
kubectl delete pod -n kube-system -l k8s-app=kube-proxy
```

## Is Kube-Proxy Running as a DaemonSet?

Yes, **Kube-Proxy runs as a DaemonSet** in Kubernetes by default.

- It is deployed in the **`kube-system` namespace**.
- Since it is a DaemonSet, it runs **one instance per node** to handle networking for that node.

You can verify its deployment with:

```
kubectl get ds -n kube-system | grep kube-proxy
```

or

```
kubectl get pods -n kube-system -l k8s-app=kube-proxy
```

**Why is Kube-Proxy Referred to as "Optional" in Kubernetes Docs?**

Kube-Proxy is **not strictly required** because its functionality can be replaced by **CNI plugins** that handle service networking differently. Here's why:

1️⃣ **Some CNI Plugins Handle Networking Without Kube-Proxy**

- Modern CNI plugins like **Cilium, Calico, and Antrea** have built-in mechanisms for handling service traffic.

- These CNIs use **eBPF (Extended Berkeley Packet Filter)**, which is **faster than iptables/IPVS** and eliminates the need for Kube-Proxy.

#### 2️⃣ eBPF-Based Networking is Becoming More Common

- eBPF is an advanced technology that allows direct kernel-level packet filtering and forwarding, making **Kube-Proxy unnecessary** in some setups.
- For example, **Cilium with eBPF** completely **removes the need for Kube-Proxy** while offering better performance.

#### 3️⃣ Some Kubernetes Deployments Don't Use Services for Traffic Routing

- If a Kubernetes cluster only uses **Ingress controllers, service meshes (Istio, Linkerd), or external load balancers**, Kube-Proxy **may not be needed**.

### Should You Disable Kube-Proxy?

- If using a traditional **iptables/IPVS-based setup**, **Kube-Proxy is required**.
- If using **eBPF-based CNIs (like Cilium)**, **Kube-Proxy can be disabled**.
- In **most standard Kubernetes clusters**, Kube-Proxy is **still used** for service networking.

#### To Disable Kube-Proxy (if using eBPF CNIs like Cilium)

```
kubectl -n kube-system delete ds kube-proxy
```

> **Note:** Only disable Kube-Proxy if your CNI plugin explicitly supports it!

Yes, **Kube-Proxy runs on control plane nodes** as well.

**Why Does Kube-Proxy Run on Control Plane Nodes?**

- If **control plane nodes also act as worker nodes** (i.e., they schedule workloads), then Kube-Proxy is required to manage networking for pods running on them.
- Even if the control plane nodes **don't schedule workloads**, they may still need Kube-Proxy to ensure proper communication between Kubernetes services (like `kube-apiserver`, `etcd`, etc.).

**How to Check if Kube-Proxy is Running on Control Plane Nodes?**

You can verify this with:

```
kubectl get pods -n kube-system -o wide | grep kube-proxy
```

This will list all the nodes running Kube-Proxy, including control plane nodes.

**Can We Disable Kube-Proxy on Control Plane Nodes?**

- If control plane nodes **don't run workloads**, **you can disable Kube-Proxy on them** to reduce resource usage.

To do this, **taint control plane nodes** so no pods (other than system components) run there:

```
kubectl taint nodes <control-plane-node-name>
node-role.kubernetes.io/control-plane=:NoSchedule
```

- However, it's generally safe to **keep Kube-Proxy running** on control plane nodes to avoid any unexpected network issues.

## Kubelet runs only as a daemon service on every node in the Kubernetes cluster. It is not deployed as a Kubernetes pod like other components.

### Why Does Kubelet Run as a Daemon Service?

1. **Directly Managed by the OS**
    - Kubelet runs as a **systemd service** (or an init service) on Linux-based nodes.
    - It starts when the node boots and ensures that workloads (pods) are running.
2. **Critical for Node Functionality**
    - Since Kubelet is responsible for **communicating with the API server**, it must always be running for the node to function in the cluster.
    - If Kubelet were a Kubernetes pod, it would **depend on itself** to run, creating a failure loop.
3. **Not a Kubernetes Pod**
    - Unlike other control plane components (which run as **static pods** on control plane nodes), Kubelet **does not run inside Kubernetes itself**.

- Instead, it is **installed manually** or via tools like `kubeadm`, `kops`, or managed services.

**How to Check if Kubelet is Running?**

You can check the status of Kubelet with:

```
systemctl status kubelet
```

or

```
ps aux | grep kubelet
```

If Kubelet is not running, you can start it using:

```
systemctl start kubelet
```

**Can Kubelet Be Run as a Pod?**

❌ No, Kubelet **cannot be a pod** because it is responsible for running and managing all other pods. Running Kubelet as a pod would create a **circular dependency issue**.

However, Kubelet does **register itself as a node** in the Kubernetes cluster, but it remains an external daemon process.

Kubernetes is a **powerful container orchestration system** that automates **deployment, scaling, and management** of containerized applications. It follows a **master-worker architecture**, where the **Control Plane** 🏢 manages the cluster, and **Worker Nodes** 🖥️ handle workloads.

🔹 **Request Handling & State Management**

When a user submits a request (like deploying an application), it first goes to the **Kube-API Server** 🔗, which acts as the central hub of the cluster. The request is **validated** ✅ and then stored in **etcd** 🗄️, a distributed key-value store that keeps the entire cluster state.

🔹 **Scheduling & Workload Execution**

The **Kube-Scheduler** 🎯 selects the best worker node based on available resources, then the **Kubelet** 🤖 on that node pulls the required container images 📦 and ensures the pod is running. Meanwhile, **Kube-Proxy** 🌐 manages networking, enabling seamless communication between pods and services.

🔹 **Cluster Monitoring & Self-Healing**

Controllers within the **Kube-Controller-Manager** ⚙️ continuously monitor the cluster and enforce the desired state. If a pod crashes 💥 or a node fails ❌, Kubernetes automatically reschedules workloads to a healthy node, ensuring high availability and fault tolerance.

🔹 **High Availability & Reliability**

For **high availability**, multiple control plane nodes 🏢🔁 ensure redundancy through **leader election mechanisms** and a **distributed etcd cluster** 🔄. Core

components like the API server, scheduler, and controllers run as **static pods**, allowing the **Kubelet** to restart them if they fail.

With its **self-healing capabilities**, **automated scaling**, and **declarative management**, Kubernetes provides a **robust, scalable, and resilient** platform for running modern cloud-native applications. 🚀🔥

## 🔚 Wrapping Up – The Kubernetes Journey So Far

As we conclude this deep dive into **Kubernetes architecture**, we have explored how its core components seamlessly work together to orchestrate, manage, and scale containerized applications. From understanding the **control plane** and its critical role in cluster management to the **worker nodes** executing workloads efficiently, we have uncovered the inner workings of Kubernetes in detail.

This knowledge is essential for anyone looking to build, manage, or optimize Kubernetes clusters, whether for **self-managed environments** or **cloud-managed services**. Understanding the **API server, scheduler, controllers, etcd, kubelet, and networking components** provides a solid foundation to confidently navigate the Kubernetes ecosystem.

### 📌 What's Next?

Kubernetes is vast, and learning is an ongoing process. **Mastery comes from consistent practice, hands-on experience, and staying updated with evolving best practices**. That's why I'll be sharing **daily Kubernetes tasks**, practical challenges, and insights to help you deepen your understanding and sharpen your skills.

- 🔹 **Follow me** for structured, real-world Kubernetes learning.
- 🔹 Expect **daily tasks, best practices, troubleshooting tips, and in-depth explanations**.
- 🔹 Engage, ask questions, and let's grow together as a DevOps community.

I truly appreciate your time and support throughout this journey. **Your feedback, questions, and discussions have been invaluable.** Thank you for being part of this learning experience!

Let's continue exploring **Kubernetes, DevOps, and cloud technologies** together. 🚀

Stay tuned for more! **#KeepLearning #Kubernetes #DevOps #CloudComputing**