Kubernetes

Kubernetes (K8s) is an **orchestration platform** that **automates** the deployment, scaling, and management of containerized applications. It ensures that applications run efficiently, handle traffic properly, and recover from failures automatically.

Basic Components of Kubernetes

1. Node (Worker Machine)

A physical or virtual machine where applications run. Two types:

- **Master Node** → Manages cluster (scheduling, scaling, monitoring).
- **Worker Node** → Runs application workloads.

2. Pod

- The **smallest unit** in Kubernetes.
- A **Pod contains one or more containers** (like Docker containers).
- Kubernetes deploys, scales, and restarts Pods as needed.

3. Deployment

- Defines how Pods should be created and managed.
- Ensures the correct number of Pods are running.
- Supports **rolling updates** (zero downtime deployments).

4. Service

- Exposes Pods to **internal** or **external** traffic.
- **Types**:
  - **ClusterIP** → Internal service (default).
  - **NodePort** → Exposes service on a static port.
  - **LoadBalancer** → Uses a cloud provider's load balancer.

5. Ingress

- Routes external HTTP/HTTPS requests to Services.
- Allows domain-based routing (`myapp.com/api → backend-service`).

6. ConfigMap & Secret

- **ConfigMap** → Stores configuration settings (e.g., API URLs).
- **Secret** → Stores sensitive data (e.g., passwords, API keys).

7. Persistent Volume (PV) & Persistent Volume Claim (PVC)

- Provides **storage** for Pods.
- **PV** → Storage resource.

- **PVC** → A request for storage.

8. Horizontal Pod Autoscaler (HPA)

- **Automatically scales** Pods based on CPU or memory usage.

Example: Food Delivery App (Like Uber Eats)

Imagine you are running a **food delivery app** where:

- Restaurants register to sell food.
- Customers place orders through the app.
- Delivery drivers pick up and deliver food.

Your application has multiple components:

1. **Frontend** (React/Next.js) → Customer UI, Restaurant UI, Delivery UI.
2. **Backend** (Node.js/Express) → Handles orders, payments, users.
3. **Database** (PostgreSQL) → Stores user data, orders, etc.
4. **Worker Service** (RabbitMQ, Kafka) → Handles background tasks like sending notifications.

Now, let's see how **Kubernetes** helps deploy and manage this system efficiently.

1. Kubernetes Concepts Applied to Our App

1.1 Containers (Docker)

Each component of your app (frontend, backend, database, worker) runs inside its own **container**.

- **Why?** Containers ensure that your app runs the same way on any machine.

Example:

- frontend-container → Runs React app.
- backend-container → Runs Node.js/Express API.
- db-container → Runs PostgreSQL.
- worker-container → Runs background tasks.

1.2 Pods

In Kubernetes, the smallest unit is a **Pod**.
A Pod is a wrapper around one or more containers.

Example:

- frontend-pod → Contains frontend-container.
- backend-pod → Contains backend-container.

- db-pod → Contains db-container.
- worker-pod → Contains worker-container.

Each **Pod** runs on a **Node** (a machine in the Kubernetes cluster).

2. Workflow of Deploying Our App in Kubernetes

Step 1: Define Deployments

A **Deployment** in Kubernetes manages Pods.

You write a YAML file for each component:

Example: Backend Deployment (backend-deployment.yaml)

```
// put yoapiVersion: apps/v1
kind: Deployment
metadata:
  name: backend
spec:
  replicas: 3  # Runs 3 backend Pods
  selector:
    matchLabels:
      app: backend
  template:
    metadata:
      labels:
        app: backend
    spec:
      containers:
      - name: backend-container
        image: myrepo/backend:v1
        ports:
        - containerPort: 5000ur code here
```

What happens?

- **Kubernetes creates 3 backend pods running the API.**
- **If a backend pod crashes, Kubernetes restarts it.**

Step 2: Create a Service

A **Service** exposes Pods so other components (or users) can communicate with them.

Example: Backend Service (backend-service.yaml)

```
apiVersion: v1
kind: Service
metadata:
  name: backend-service
spec:
  selector:
    app: backend
  ports:
    - protocol: TCP
      port: 80  # External port
      targetPort: 5000  # Pod port
  type: ClusterIP
```

What happens?

- **backend-service** allows frontend to talk to backend Pods.

Step 3: Add an Ingress for Public Access

An **Ingress** routes external traffic. **Example: Ingress for Frontend**

```
apiVersion: networking.k8s.io/v1
kind: Ingress
metadata:
  name: food-app-ingress
spec:
  rules:
  - host: foodapp.com
    http:
      paths:
      - path: /
        pathType: Prefix
        backend:
          service:
            name: frontend-service
            port:
              number: 80
      - path: /api
        pathType: Prefix
        backend:
          service:
            name: backend-service
            port:
              number: 80
```

What happens?

- When users visit foodapp.com, requests go to the frontend.
- When users visit foodapp.com/api, requests go to the backend.

Step 4: Auto Scaling (HorizontalPodAutoscaler)

If 1000 users place orders at once, we need more backend servers.

Auto Scaling Backend

```
apiVersion: autoscaling/v2
kind: HorizontalPodAutoscaler
metadata:
  name: backend-hpa
spec:
  scaleTargetRef:
    apiVersion: apps/v1
    kind: Deployment
    name: backend
  minReplicas: 3
  maxReplicas: 10
  metrics:
  - type: Resource
    resource:
      name: cpu
      target:
        type: Utilization
        averageUtilization: 50
```

What happens?

- If CPU usage goes above 50%, Kubernetes adds more backend Pods.
- If traffic reduces, Kubernetes removes extra Pods.

3. Kubernetes Full Workflow (End-to-End)

1 Developer Pushes Code

- Developer updates backend, commits changes.
- CI/CD pipeline builds Docker image (myrepo/backend:v2).
- Pushes the image to Docker Hub/GCR.

2 Kubernetes Pulls New Image

- Deployment is updated (image: myrepo/backend:v2).
- Kubernetes **gradually replaces old backend Pods** with new ones.

3 Traffic Reaches the App

- User visits foodapp.com → Reaches frontend.
- Frontend calls /api/orders → Routed to backend service.

4 Auto Scaling Handles Load

- More users → Kubernetes **adds backend Pods**.
- Less users → Kubernetes **removes extra Pods**.

5 Self-Healing

- If a backend Pod crashes, Kubernetes automatically restarts it.

---

4. Why Use Kubernetes for This?

Scalability – Can handle high traffic automatically.
Self-Healing – If a Pod crashes, Kubernetes restarts it.
Zero Downtime Deployments – Rolling updates replace old versions smoothly.
Load Balancing – Services distribute traffic evenly.
Security & Networking – Controls access using Services & Ingress.

Kubernetes **orchestrates** your app like a **restaurant manager**:

- **Starts & stops servers (Pods) as needed**.
- **Routes customer requests (Ingress & Services)**.
- **Handles scaling when too many orders come in (HPA)**.
- **Fixes issues automatically if a server fails (Self-healing Pods)**.

1. Planning in Kubernetes

Before deploying applications, Kubernetes needs to **plan** resources efficiently.
Planning involves:
Resource Allocation – Ensuring enough CPU, memory, and storage are available.
Workload Distribution – Placing Pods on the right Nodes.
Scaling & Auto-recovery – Keeping apps running under load or failures.

This planning is managed by **etcd, the Scheduler, and the Controller Manager**.

---

2. etcd (Cluster Brain & Data Store)

◆ **What is it?**

- etcd is a **distributed key-value store** that stores **all cluster data**.
- It keeps track of **current and desired state** of Kubernetes objects (Pods, Deployments, Services, etc.).

◆ **How does it work?**

- Every time a change is made (like scaling Pods), it gets stored in **etcd**.
- The **API Server** reads and writes to etcd to keep the cluster state up to date.

◆ **Example:**

- If a Pod crashes, etcd still holds the desired number of replicas.
- The **Controller Manager** detects this and recreates the missing Pod.

◆ **Why is it important?**
Stores entire cluster configuration.
Ensures fault tolerance (runs as a distributed system).
If etcd fails, Kubernetes loses track of cluster state.

---

3. Kubernetes Scheduler (Pod Placement)

◆ **What is it?**
The **Scheduler** decides which Node should run a new Pod.

◆ **How does it work?**

1. When you create a **Pod**, it has **no assigned Node** initially.
2. The Scheduler checks:
    - **Node resources (CPU, memory, storage)**
    - **Node affinity (rules to prefer certain Nodes)**
    - **Taints & tolerations (rules to avoid certain Nodes)**
3. It **assigns the Pod to the best Node** and updates etcd.

🔹 **Example:**

- If a Node has **high CPU usage**, the Scheduler **avoids placing new Pods** there.
- If a Node is dedicated for **database workloads**, the Scheduler follows that rule.

🔹 **Why is it important?**
Ensures efficient resource use.
Avoids overloading Nodes.
Supports affinity rules for custom placement.

---

4. Controller Manager (Ensures Desired State)

🔹 **What is it?**
The **Controller Manager** runs multiple **controllers** to keep the cluster in the desired state.

🔹 **How does it work?**

- It watches etcd for changes.
- If the **actual state** differs from the **desired state**, it makes corrections.

🔹 **Main Controllers:**

- **Node Controller** → Detects and handles failed Nodes.
- **Replication Controller** → Ensures correct Pod count.
- **Endpoint Controller** → Manages Pod-to-Service mapping.
- **Job Controller** → Handles batch jobs and cron jobs.

🔹 **Example:**

- You define **3 backend Pods** in a Deployment.
- One Pod crashes.
- The Controller Manager detects this and **creates a new Pod** automatically.

🔹 **Why is it important?**
Keeps the cluster running as expected.
Handles failures and reschedules workloads.
Automates scaling and Pod recovery.

How They Work Together 🚀

1 **etcd** stores cluster data (desired + current state).
2 **Scheduler** finds the best Node for new Pods.
3 **Controller Manager** ensures the cluster stays in the correct state.

What is a Namespace?

A **Namespace** is a way to **organize and isolate resources** in a Kubernetes cluster. It helps when you have multiple teams, projects, or environments (dev, staging, prod) running on the same cluster.

Key Features

Isolation – Resources in one namespace don't affect others.
Multi-Tenancy – Different teams can work in different namespaces.
Resource Quotas – You can limit CPU, memory, and storage per namespace.

What is a ConfigMap?

A **ConfigMap** allows you to **store configuration data** in key-value pairs and pass it to applications without modifying container images.

Why use ConfigMap?

Keeps environment-specific data separate from the app.
Easier updates without redeploying containers.
Can be used as environment variables, command-line arguments, or config files.

What is a Service?

A **Service** in Kubernetes exposes a set of **Pods** to the network.
Since Pods are ephemeral (they can be created/destroyed), a **Service provides a stable IP and DNS name** to access the application.

Types of Services

Service Type	Description
ClusterIP (default)	Internal service, accessible only within the cluster.
NodePort	Exposes service on each node's IP at a static port (e.g., nodeIP:30000).
LoadBalancer	Uses cloud provider's external load balancer (AWS, GCP, Azure).
ExternalName	Maps service to an external DNS (e.g., my-db.example.com).

How Services Work?

Pods can be created/destroyed, but the Service always provides a fixed endpoint.
Traffic is load-balanced across multiple Pods.
Pods find Services using DNS (backend-service.default.svc.cluster.local).

Basic kubectl Commands

```
kubectl cluster-info                       # Show cluster details
kubectl get nodes                          # List all worker nodes
kubectl describe node <node-name>          # Get details of a specific node
kubectl get pods                           # List all Pods
kubectl get pods -n <namespace>            # List Pods in a specific namespace
kubectl describe pod <pod-name>            # Detailed info about a Pod
kubectl logs <pod-name>                    # View logs of a Pod
kubectl exec -it <pod-name> -- bash        # Get inside a running Pod
kubectl get deployments                    # List all Deployments
kubectl describe deployment <name>         # Detailed Deployment info
kubectl get services                       # List all Services
kubectl describe service <name>            # Detailed Service info
kubectl apply -f <file.yaml>               # Create or update resources from a YAML file
kubectl delete -f <file.yaml>              # Delete resources defined in a YAML file
kubectl delete pod <pod-name>              # Delete a Pod (recreated if part of a Deployment)
kubectl delete deployment <name>           # Delete a Deployment
```