Docker is great for containerization, but when you scale applications, you hit limitations:

**Drawbacks of Docker (Loopholes in Using Just Docker)**

**1.Single Host Limitation**

1. Docker runs containers on a **single machine**.
2. What if traffic increases? We need multiple machines.
3. Solution? **Kubernetes (K8s) helps in scaling across multiple nodes.**

**2.Manual Scaling**

1. In Docker, you **manually start** and **stop** containers.
2. No auto-scaling, no self-healing.
3. Kubernetes can **auto-scale and restart failed containers**.

**3.No Load Balancing**

1. Docker itself doesn’t manage traffic distribution.
2. If multiple containers are running, **how will traffic be distributed?**
3. Kubernetes **has built-in load balancing**.

**4.Storage & Networking Challenges**

1. Docker doesn’t handle **persistent storage** properly.
2. Networking between multiple Docker hosts is **not easy**.
3. Kubernetes has **Persistent Volumes (PV), Persistent Volume Claims (PVC), and Service Networking**.

**5.No High Availability (HA)**

1. If a container crashes in Docker, **it stays down unless restarted manually**.
2. In Kubernetes, **Pods auto-restart** and **maintain high availability**.

# Kubernetes vs. Docker-Compose:

| Feature | Docker-Compose | Kubernetes |
|---|---|---|
| Scaling | Manual | Auto-scaled Pods |
| Load Balancing | No | Built-in Services & Ingress |
| Self-healing | No | Yes |
| Multi-node Support | No | Yes |
| Storage | Local | Persistent Volumes |

## Key Reasons to Use Kubernetes:

1️⃣ Scalability 🚀

2️⃣ Self-Healing & High Availability 💊

3️⃣ Load Balancing & Service Discovery 🌎

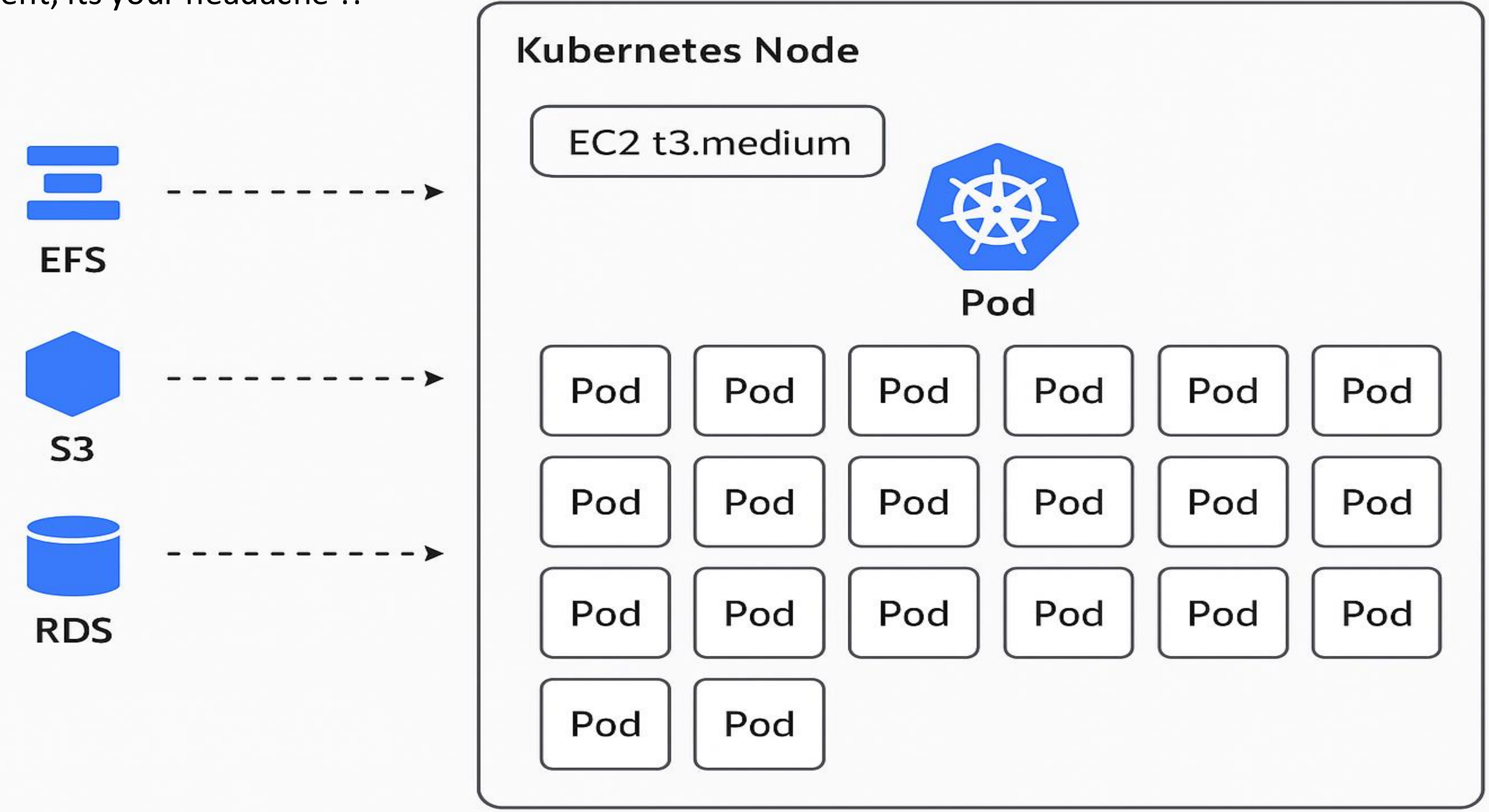
4️⃣ Multi-Cloud & Hybrid Deployments ☁

5️⃣ Networking & Security 🔒 CNI & RBAC

6️⃣ Persistent Storage 📦

✅ Handles scaling automatically
✅ Recovers from failures automatically
✅ Distributes traffic intelligently
✅ Works across multiple clouds
✅ Ensures persistent storage & networking

1 EKS and 1 EC2 we can run 20 Applications, with high availability and Auto scaling, Loadbalencing. Backup u need according to Your requirement, its your headache !!



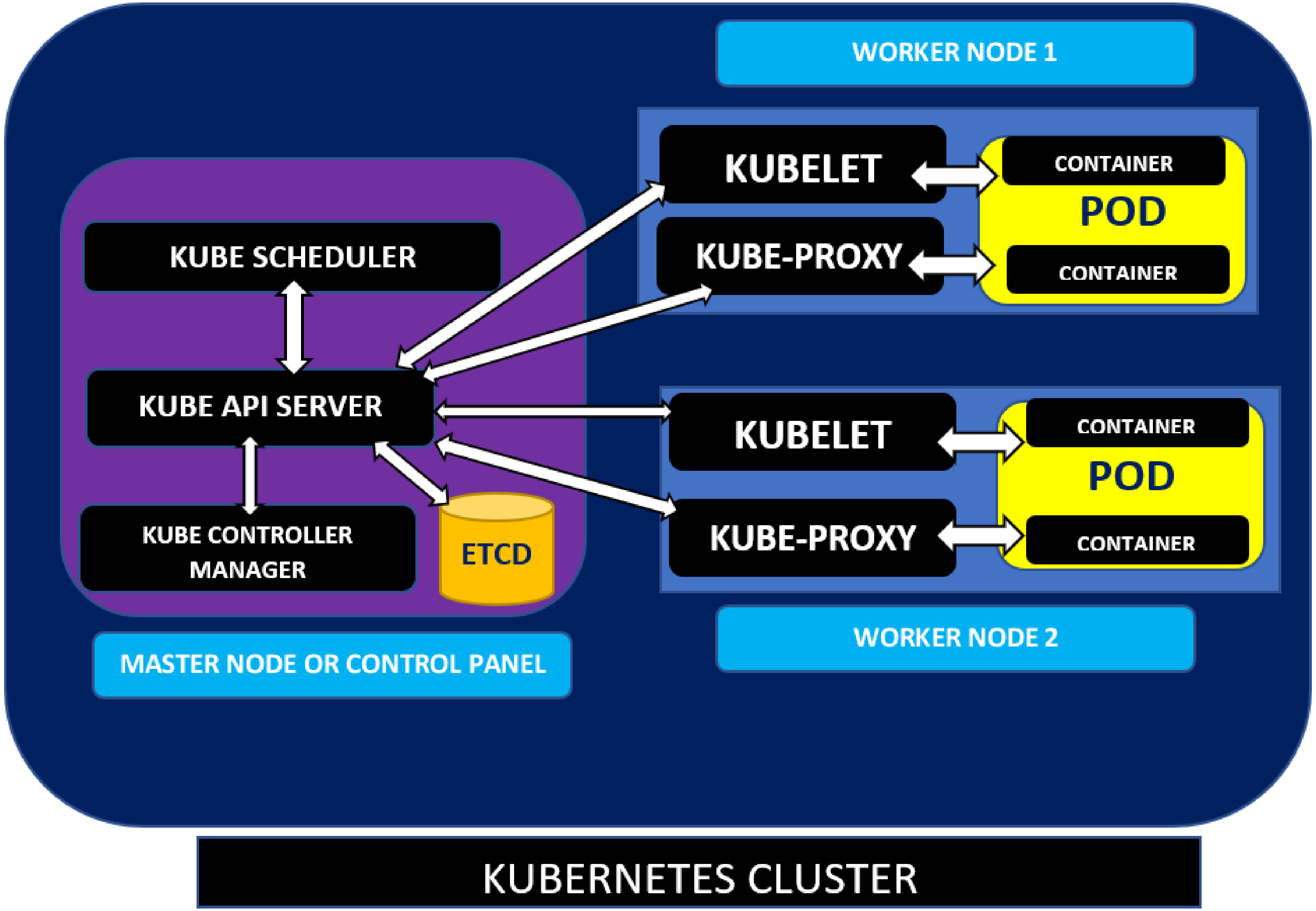

- S3 → Access via AWS SDK
- EFS → Mounted volumes via PVC
- RDS → DB connection strings inside pods

| Deployment Type | Resources Used | Max Applications | Cost per Hour | Monthly Cost (730 hrs) |
|---|---|---|---|---|
| EKS (1 EC2 as Worker Node) | 1x t3.medium (2 vCPU, 4GB RAM) | 10 Applications | $0.1418 (EKS + EC2) | $103.51 |

KUBERNETES ARCHITETURE

Master Node components:

API SERVER
KUBE SCHEDULAR
KUBE CONTROLLER MANAGER




Worker node Components:

Kubelet
Kube-proxy
Container runtime(CRI-O)

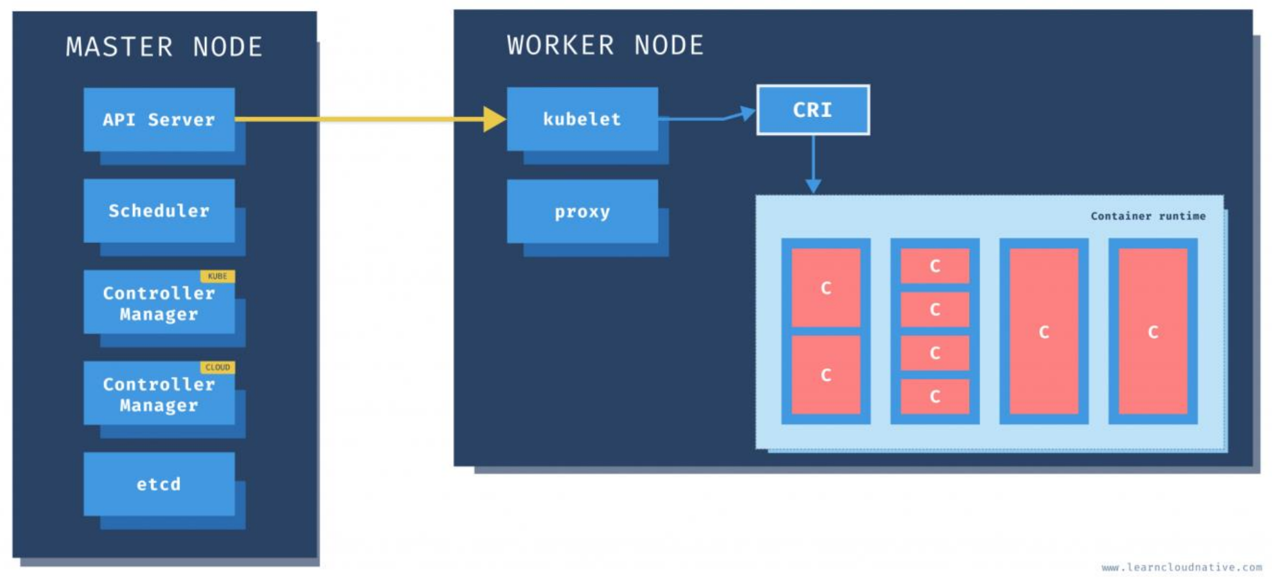
## 💥 How Kubernetes Works (Bird's Eye View):

Kubernetes = **Cluster** of machines.

It has 2 major parts:

**1.Master Node (Control Plane)** — brains 🧠

**2.Worker Nodes** — workers 💪 (where your apps/pods actually run)




## 🧠 Master Node (a.k.a Control Plane Components):

This is where K8s makes all decisions like a boss 😎

| Component | Role |
|---|---|
| **API Server** | 🗣️ The front door! Every command hits this. kubectl → API server |
| **etcd** | 📖 Key-value store → cluster state (like DB of K8s) |
| **Controller Manager** | 👷 Watches and reacts to changes (creates pods if one dies) |
| **Scheduler** | 🧮 Decides *where* to run new pods (which node) |
| **Cloud Controller Manager** | ☁️ Talks to cloud (AWS, GCP) if needed |

## 🔷 What is a Node in Kubernetes?

A **Node** in Kubernetes is a **worker machine (VM or physical server)** where the containers (Pods) run. Each **Kubernetes cluster**

1️⃣ **Control Plane (Master Node)** → Manages the cluster

2️⃣ **Worker Nodes** → Run workloads (containers inside Pods)

## 🔷 Worker Node Components

Each worker node has these core components:

1️⃣ **Kubelet (Agent)**

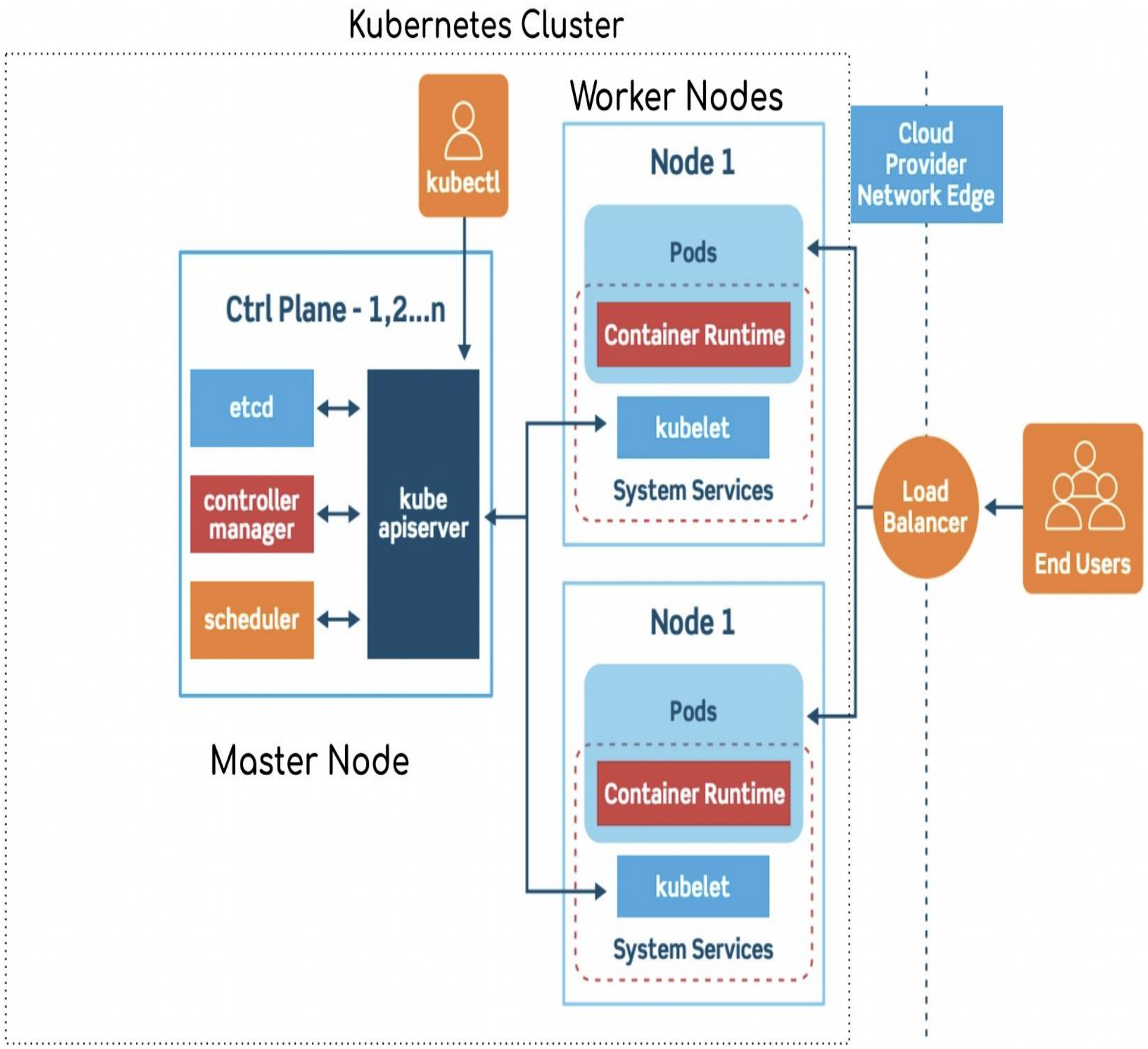
- The **bridge** between the Node and the Control Plane
- **Receives commands** from the API Server
- Ensures the Pod is running as expected
- Communicates with the **Container Runtime**

2️⃣ **Container Runtime (Docker/Containerd/CRI-O)**

- Runs and manages **containers inside the pod**
- The default runtime for Kubernetes is **containerd**

3️⃣ **Kube Proxy (Networking Component)**

- **Handles networking inside the cluster**
- Enables **Pod-to-Pod communication** across nodes
- Implements **Load Balancing for services**



💥 **Real-World Example:**

If your master node dies:

- Existing pods may keep running (until they crash or need rescheduling)
- But you **can't deploy, scale, or recover** anything → app becomes fragile.

If a worker node dies:

- Control plane detects it
- Reschedules pods on other available nodes → **self-healing kicks in**

**High Availability (HA):**

- Without a working master node, new pods **can't be scheduled** or managed.
- Without worker nodes, there's **no place to run** the workloads.
- So, both are **mutually dependent** to keep the app **available and responsive**.

## Rule to write a Yaml:

❌ No tabs allowed – use only spaces

✅ Indentation matters – usually 2 spaces per level

🧠 Key: value format
Example: name: mypod

📋 Lists use - (dash)

**Strings don't need quotes (usually):**

```
name: myapp      # ✅ this is fine
name: "myapp"    # ✅ also valid
```

**Boolean values must be lowercase:**

```
enabled: true    # ✅
enabled: FALSE   # ❌ (wrong)
```

**Multiline strings:**

```
note: |
  This is line 1
  This is line 2
```

**Use anchors (&) and aliases (*) for reuse:**

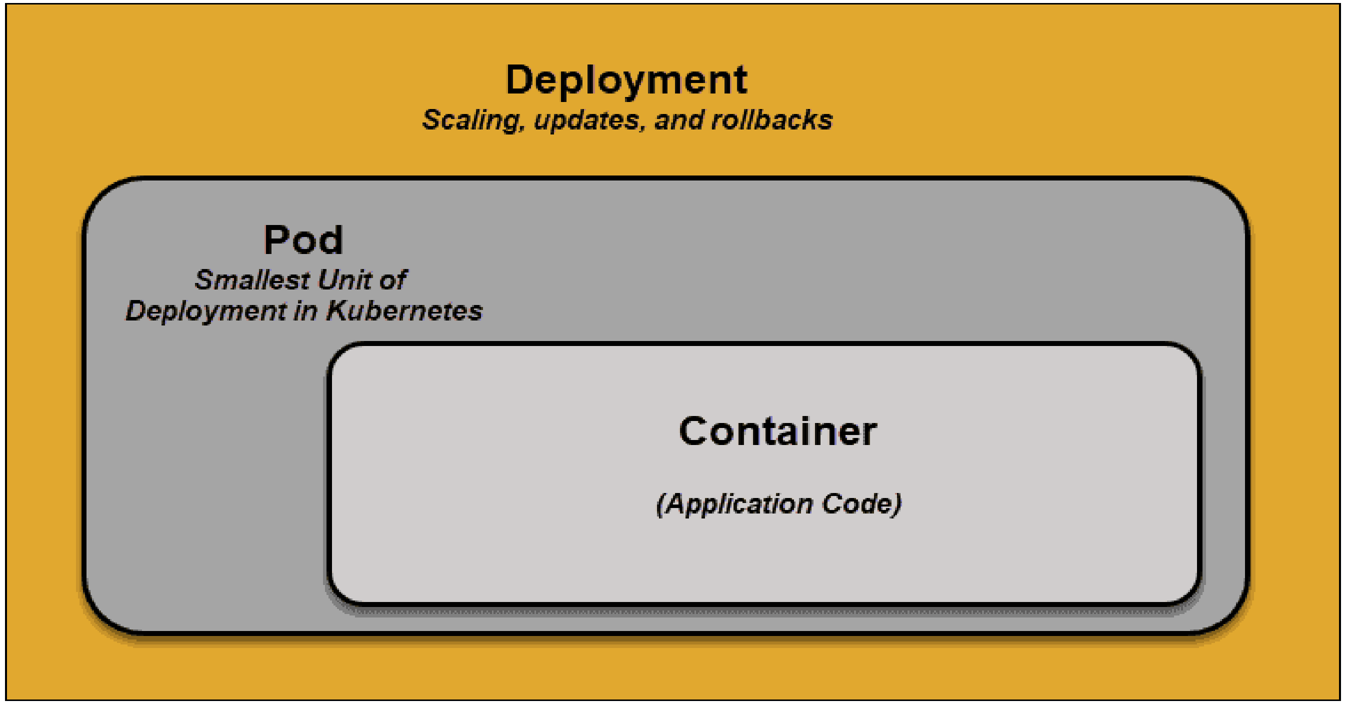
```
default: &app_defaults
  image: nginx
  port: 80

dev:
  <<: *app_defaults
  replicas: 2
```

Pods are the smallest deployable units of computing that you can create and manage in Kubernetes. A Pod (as in a pod of whales is a group of one or more containers, with shared storage and network resources, and a specification for how to run the containers.

**pod.yaml:**

```yaml
apiVersion: v1
kind: Pod
metadata:
  name: my-nginx-pod  # Name of the Pod
  labels:
    app: nginx
spec:
  containers:
  - name: nginx-container
    image: nginx:latest
    ports:
    - containerPort: 80
```




```
>  kubectl apply -f pod.yaml
```

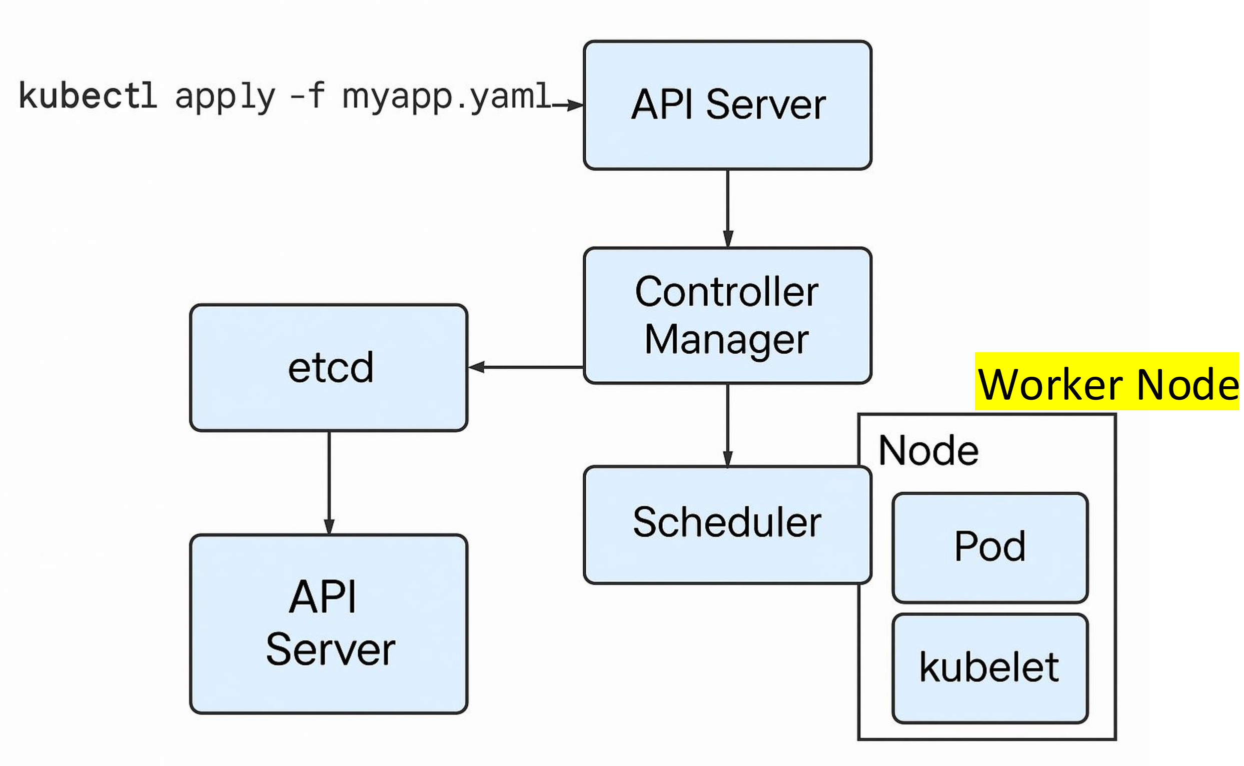
```
> kubectl get pods
```

```
> kubectl describe pod my-nginx-pod
```

```
> kubectl delete pod my-nginx-pod
```

🔷 apiVersion: v1 → Using Kubernetes API version 1
🔷 kind: Pod → Defines that this is a Pod resource
🔷 metadata: → Assigns a name and labels to the pod
🔷 spec: → Defines the container inside the pod
🔷 containers: → List of containers inside the pod
🔷 image: nginx:latest → Uses the official NGINX image from Docker Hub
🔷 ports: → Exposes port 80 inside the container

1. 🔁 How It Flows (Simple Steps):
2. You type kubectl apply -f mypod.yaml
3. kubectl → hits API server
4. API server stores desired state in etcd
5. Controller manager notices no pod is running
6. Scheduler picks a node
7. kubelet (on that node) pulls image, runs container
8. 🎉 Pod is up!



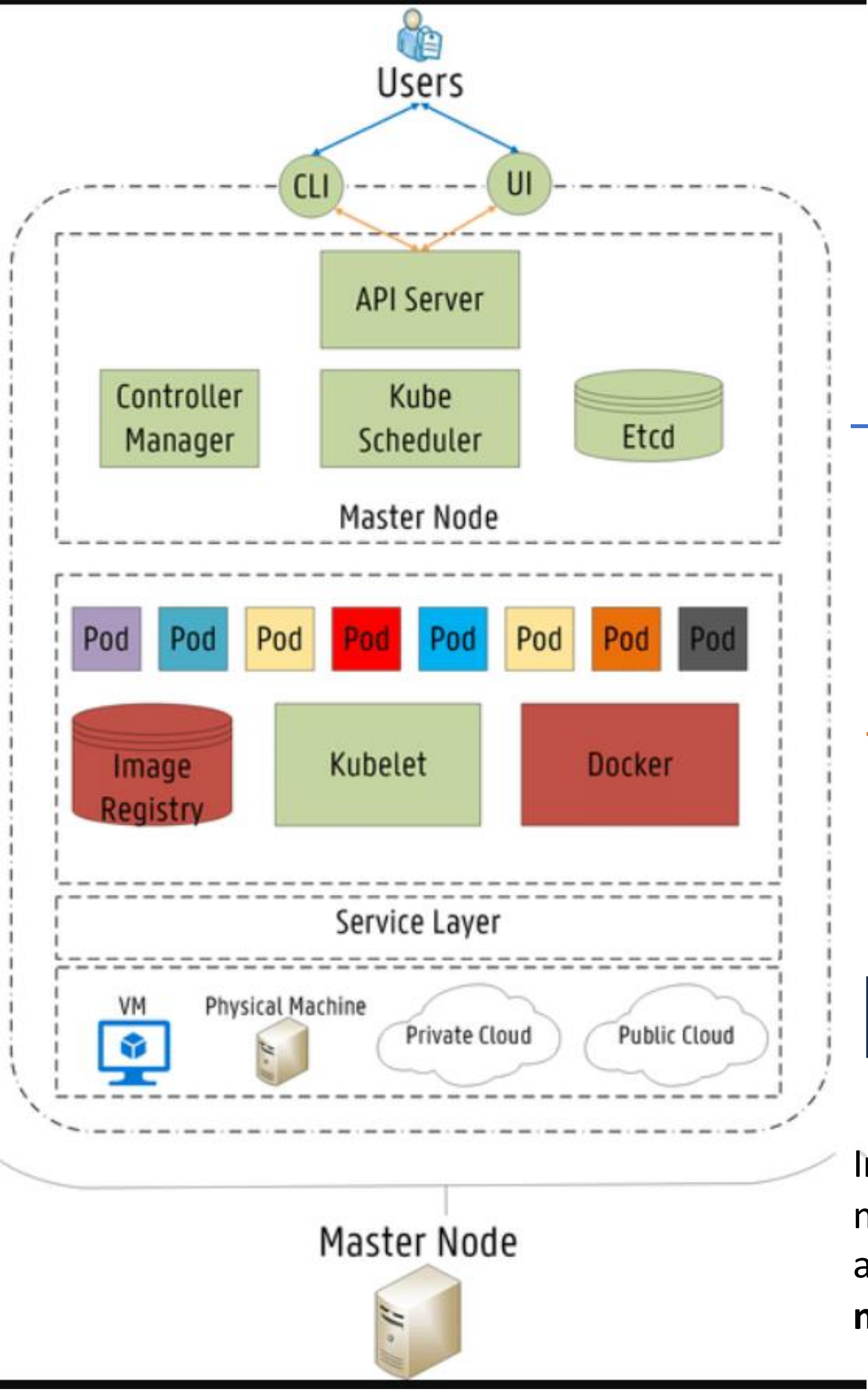
## 📌 Pod Health States in Kubernetes:

A **Pod** in Kubernetes goes through different states based on its lifecycle and health status. Below are the key **Pod health states**:

| State | Description |
|---|---|
| **Pending** | The Pod has been accepted by Kubernetes but **not yet running** (waiting for resources, scheduling, or pulling images). |
| **Running** | The Pod is successfully scheduled and at least one container is in a **Running** state. |
| **Succeeded** | All containers in the Pod have **completed execution successfully** (only applies to jobs/batch workloads). |
| **Failed** | One or more containers in the Pod have **terminated with a non-zero exit code**. |
| **Unknown** | The Pod state is unknown (often due to network or node communication issues). |

| State | Description |
|---|---|
| **Waiting** | The container is waiting to start. It might be pulling an image, waiting for resources, or experiencing an issue. |
| **Running** | The container is running normally without issues. |
| **Terminated** | The container has exited. It could be **successful (Exit Code 0)** or **failed (Exit Code ≠ 0)**. |

- **You may see Waiting with a reason, such as `ImagePullBackOff`, `CrashLoopBackOff`, or `ErrImagePull`.**




Users Employee/ who has access for master node

Master Node receiving User/client

Worker Node Architecture with objects

YOUR ENVIRONMENT SETUP

In Kubernetes, the **Master Node (Control Plane)** and **Worker Nodes** always work **together** to manage and run applications. This coordination ensures **high availability**, **fault tolerance**, and **automated recovery** of workloads. For production-grade clusters, using **multiple master and worker nodes** is essential for achieving true **high availability**.

# Few commands to learn

## 🚀 Kubernetes Commands You Must Know

| # | Command | What It Does |
|---|---|---|
| 1 | kubectl get pods | List all running pods in the current namespace |
| 2 | kubectl get pods -A | Get pods in **all namespaces** |
| 3 | kubectl get nodes | List all nodes (workers + masters) in the cluster |
| 4 | kubectl get svc | Show services (ClusterIP, NodePort, LB, etc.) |
| 5 | kubectl get deployments | List all deployments |
| 6 | kubectl get all | Show all K8s objects: pods, svc, deployments, etc. |
| 7 | kubectl describe pod <pod-name> | Detailed pod info + events + errors |
| 8 | kubectl describe node <node-name> | Detailed info of a specific node |
| 9 | kubectl logs <pod-name> | Show logs of a pod |
| 10 | kubectl logs <pod> -c <container> | Logs for a **specific container** in a pod |
| 11 | kubectl exec -it <pod> -- /bin/bash | SSH into a running pod/container |
| 12 | kubectl apply -f app.yaml | Create/update K8s resources from YAML |
| 13 | kubectl create -f app.yaml | Create K8s resources from YAML (fails if already exists) |
| 14 | kubectl delete -f app.yaml | Delete K8s resources defined in YAML |
| **15** | **kubectl delete pod <pod-name>** | **Delete a single pod** |
| 16 | kubectl scale deployment <deploy-name> --replicas=5 | Scale up/down pods |
| 17 | kubectl expose deployment <deploy-name> --type=NodePort --port=80 | Expose app as a service |
| 18 | kubectl rollout status deployment/<deploy-name> | Check rollout progress |
| 19 | kubectl rollout undo deployment/<deploy-name> | Rollback to previous deployment |
| 20 | kubectl edit deployment <deploy-name> | Open live YAML editor for deployment |
| 21 | kubectl port-forward pod/<pod-name> 8080:80 | Access pod port locally (dev/testing) |
| 22 | kubectl top pod | Show pod-level resource usage (CPU/RAM) |
| 23 | kubectl top node | Show node-level resource usage |
| 24 | kubectl config view | Show your current kubeconfig setup |
| 25 | kubectl config use-context <context> | Switch between clusters (like dev/stage/prod) |
| 26 | kubectl get pvc | List PersistentVolumeClaims |
| 27 | kubectl get pv | List PersistentVolumes |
| 28 | kubectl cp <pod-name>:<path> <local-path> | Copy file from pod to local machine |
| 29 | kubectl version --short | Show client/server version |
| 30 | kubectl explain pod | Show schema/documentation for a resource (great for YAML help!) |

## 1 ReplicationController (RC) [Old]

- Ensures a **specified number of pod replicas** are running at all times.
- If a pod crashes, it creates a new one.
- **Obsolete** → Replaced by ReplicaSet.

## 2 ReplicaSet (RS) [Improved RC]

- Works like ReplicationController but supports **label selectors with set-based selectors** (more flexible).
- Ensures the **desired number of pod replicas** are running.
- If a pod fails, it replaces it with a new one.

## 3 Deployment [Higher-Level Controller]

- **Manages ReplicaSets automatically** and provides additional features like:

✅ **Rolling updates** (zero downtime updates)
✅ **Rollback to previous versions**
✅ **Declarative updates** (modify pod templates easily)
✅ **Self-healing & scaling**

**Replicaset.yaml:**

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

## 🤔 Why Prefer Deployment Over ReplicaSet?

| Feature | ReplicaSet | Deployment |
|---|---|---|
| Ensures Pod Count | ✅ Yes | ✅ Yes |
| Rolling Updates | ❌ No | ✅ Yes |
| Rollbacks | ❌ No | ✅ Yes |
| Auto-Manages ReplicaSet | ❌ No | ✅ Yes |
| Recommended Usage | ❌ No (Only for special cases) | ✅ Yes (Preferred) |

💡 **Use Deployments for managing apps, not ReplicaSets directly.**

```
kubectl apply -f deployment.yaml
kubectl get deployments
```

<u>Rollout:</u> `kubectl rollout undo deployment my-nginx-deployment`

| Feature | Description |
|---|---|
| ✅ **Declarative Control** | You tell K8s what state you want, it makes it happen. |
| 🔄 **Rolling Updates** | Update app without downtime. K8s replaces old pods slowly. |
| ♻️ **Self-Healing** | If a pod crashes, Deployment brings it back automatically. |
| 📈 **Scaling** | Easy to increase/decrease pod count with 1 command. |
| 🧠 **Version Control** | Roll back to a previous version if something breaks. |
| 🔒 **Safe & Controlled** | Can pause/resume/monitor updates — great for production apps. |

**<u>deployment.yaml:</u>**

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

## 🔷 Labels & Selectors (For Grouping and Identifying Resources)

**Labels** are **key-value pairs** attached to Kubernetes objects (Pods, Nodes, etc.).
**Selectors** help filter and select resources based on labels.

**NOTE: When we create multiple Pods using a Deployment, labels are defined inside the Pod template.**
**The Deployment uses selector.matchLabels to find and manage the Pods based on those labels.**

## 🔷 Taints & Tolerations (For Node-Pod Restrictions)

**💡 Taints prevent certain Pods from running on a Node unless the Pod has a matching Toleration.**

**📌 Example: Taint a Node**

```
kubectl taint nodes my-node key=value:NoSchedule
```

```
apiVersion: v1
kind: Pod
metadata:
  name: my-tolerated-pod
spec:
  tolerations:
  - key: "key"
    operator: "Equal"
    value: "value"
    effect: "NoSchedule"
  containers:
  - name: nginx
    image: nginx:latest
```

```
apiVersion: v1
kind: Pod
metadata:
  name: my-nginx-pod
  labels:
    app: nginx
    env: production
spec:
  containers:
  - name: nginx
    image: nginx:latest
```

```
selector:
  matchLabels:
    app: nginx
```

**💡 Simple Flow:**

**You define labels under template.metadata.labels (inside the Deployment)**
**selector.matchLabels in the Deployment spec looks for those labels**
**Kubernetes then manages only the Pods that match those labels**

## 📌 What is Node Affinity?

Node Affinity allows you to **control which nodes a pod can be scheduled on** based on node labels.

It's an **advanced version of nodeSelector** that supports:

✅ **Soft (preferred) rules** → Pod *tries* to be scheduled on preferred nodes.

✅ **Hard (required) rules** → Pod *must* be scheduled on specified nodes.

### 🔹 Types of Node Affinity

| Type | Effect |
|---|---|
| `requiredDuringSchedulingIgnoredDuringExecution` | **Hard rule** → Pod must be scheduled on the specified node. |
| `preferredDuringSchedulingIgnoredDuringExecution` | **Soft rule** → Scheduler tries to place the Pod on preferred nodes but will schedule elsewhere if needed. |

### 🔹 How is Node Affinity Different from Taints & Tolerations?

| Feature | Node Affinity | Taints & Tolerations |
|---|---|---|
| **Who sets it?** | Pod definition | Node definition |
| **Purpose** | Control pod placement based on labels | Restrict certain pods from running on nodes |
| **Effect** | Scheduler decides based on preferences | Node blocks pods unless they have tolerations |

**nodeaffinity.yaml:**

```yaml
apiVersion: v1
kind: Pod
metadata:
  name: my-app-pod
spec:
  affinity:
    nodeAffinity:

requiredDuringSchedulingIgnoredDuringExecution:
        nodeSelectorTerms:
        - matchExpressions:
          - key: env
            operator: In
            values:
            - production
  containers:
  - name: my-app
    image: nginx
```

# 🔷 Requests and Limits in Kubernetes

📌 **What are Requests and Limits?**

Requests and Limits **control resource allocation (CPU & Memory) for Pods** in Kubernetes.

- **Requests** → Minimum resources a container gets.
- **Limits** → Maximum resources a container can use.

💡 **Why use them?**

✅ Prevents a single container from consuming all resources.
✅ Ensures fair resource distribution across Pods.
✅ Helps Kubernetes **schedule** Pods efficiently.

🔷 **What Happens If a Pod Exceeds Its Limits?**

✅ **If CPU exceeds the limit** → Pod is **throttled** (slows down but doesn't crash).
✅ **If Memory exceeds the limit** → Pod is **OOMKilled** (Out of Memory).

**requestandlimits,.yaml:**

```
apiVersion: v1
kind: Pod
metadata:
  name: my-app
spec:
  containers:
  - name: nginx
    image: nginx
    resources:
      requests:
        memory: "128Mi"   # Minimum memory
        cpu: "250m"       # Minimum CPU (0.25 cores)
      limits:
        memory: "256Mi"   # Max memory allowed
        cpu: "500m"       # Max CPU (0.5 cores)
```

✅ **Pod gets at least 128Mi memory and 0.25 CPU**
✅ **Cannot exceed 256Mi memory and 0.5 CPU**

## 🔷 Pod Autoscaling in Kubernetes

### 📌 What is HPA?

HPA (**Horizontal Pod Autoscaler**) **automatically scales the number of pods** in a deployment **based on CPU, memory, or custom metrics**.

### 💡 Why use HPA?

✅ Handles traffic spikes **automatically**.
✅ Prevents unnecessary over-provisioning.
✅ Optimizes resource utilization and costs.

### 🔷 How HPA Works

1. Monitors **CPU/Memory usage** of running pods.
2. If usage **exceeds the defined threshold**, HPA **adds more pods**.
3. If usage **drops below the threshold**, HPA **removes extra pods**.
4. Uses **metrics-server** to get resource usage.

```
kubectl top pods
```

```
kubectl get hpa
```

```
kubectl apply -f https://github.com/kubernetes-sigs/metrics-server/releases/latest/download/components.yaml
```

**autoscaling.yaml:**

```yaml
apiVersion: autoscaling/v2
kind: HorizontalPodAutoscaler
metadata:
  name: my-app-hpa
spec:
  scaleTargetRef:
    apiVersion: apps/v1
    kind: Deployment
    name: my-app-deployment
  minReplicas: 2
  maxReplicas: 10
  metrics:
  - type: Resource
    resource:
      name: cpu
      target:
        type: Utilization
        averageUtilization: 50  # Scale if CPU > 50%
```

## 🔹 HPA vs VPA vs Cluster Autoscaler

| Feature | HPA (Horizontal) | VPA (Vertical) | Cluster Autoscaler |
|---|---|---|---|
| **Scaling Type** | More Pods | Bigger Pods | More Nodes |
| **Triggers** | CPU, Memory, Custom Metrics | CPU, Memory | Node resource needs |
| **Use Case** | Handle high traffic | Optimize per-pod resources | Scale entire cluster |

🔷 **How Metrics Server Works in Kubernetes**

📌 **What is Metrics Server?**

The **Metrics Server** is a Kubernetes component that collects **resource usage metrics (CPU & Memory) from nodes and pods** and provides them to **HPA (Horizontal Pod Autoscaler), VPA (Vertical Pod Autoscaler), and kubectl top commands**.

💡 **Why do we need it?**

✅ **Enables HPA & VPA** for auto-scaling.

✅ **Provides real-time CPU & Memory metrics** of Pods and Nodes.

✅ **Lightweight & efficient**, unlike full monitoring tools like Prometheus.

kubectl get pods -n kube-system | grep metrics-server

## 🔹 Metrics Server vs Prometheus

| Feature | Metrics Server | Prometheus |
|---|---|---|
| **Purpose** | Provides CPU & memory usage | Full monitoring & alerting |
| **Use Case** | HPA, `kubectl top` | Detailed metrics & dashboards |
| **Data Storage** | Does not store data | Stores historical metrics |
| **Complexity** | Simple & lightweight | More complex setup |

## Health Probes (Liveness, Readiness, and Startup Probes)

Kubernetes uses **probes** to check if a container is **healthy** and **ready** to serve traffic.

| Probe Type | Purpose |
|---|---|
| Liveness Probe | Checks if the container is alive. If it fails, Kubernetes **restarts** the container. |
| Readiness Probe | Checks if the container is ready to accept traffic. If it fails, traffic is **not sent** to the pod. |
| Startup Probe | Ensures the app **fully starts** before checking liveness. |

✅ **If the liveness probe fails, the pod is restarted.**
✅ **If the readiness probe fails, the pod does not receive traffic.**

- **Probes are used to check the health of containers in a Pod.**
- **There are three types: Liveness, Readiness, and Startup probes.**
- **Liveness Probe checks if the app is still running. If not, the container is restarted.**
- **Readiness Probe checks if the app is ready to serve traffic. If not, traffic is stopped.**
- **Startup Probe is used for apps that take time to start — it delays other probes until it's ready.**
- **Probes can work using httpGet, exec, or tcpSocket.**
- **They help in self-healing, high availability, and zero-downtime deployments.**
- **Without probes, Kubernetes won't know if the app is stuck or unhealthy.**

**myprobe.yaml:**

```
apiVersion: v1
kind: Pod
metadata:
  name: my-app
spec:
  containers:
  - name: my-container
    image: nginx
    livenessProbe:
      httpGet:
        path: /
        port: 80
      initialDelaySeconds: 5
      periodSeconds: 10
    readinessProbe:
      httpGet:
        path: /
        port: 80
      initialDelaySeconds: 3
      periodSeconds: 5
```

## ConfigMaps (Manage Configuration in Kubernetes):

ConfigMaps **store configuration data** (key-value pairs) separately from the application code.

💡 **Why use ConfigMaps?**

✅ Keeps configuration **separate** from application code.

✅ Allows **dynamic updates** without rebuilding the container.

✅ Can be used in **environment variables**, **command arguments**, or **mounted as files**.

### ConfigMap.yaml

```yaml
apiVersion: v1
kind: ConfigMap
metadata:
  name: app-config
data:
  database_url: "mongodb://localhost:27017"
  app_mode: "production"
  log_level: "info"
```

### How to Use the ConfigMap in a Pod:

```yaml
apiVersion: v1
kind: Pod
metadata:
  name: app-pod
spec:
  containers:
  - name: app-container
    image: myapp:latest
    envFrom:
    - configMapRef:
        name: app-config
```

### ConfigMap as a file inside the container

```yaml
apiVersion: v1
kind: Pod
metadata:
  name: app-pod
spec:
  containers:
    - name: app-container
      image: myapp:latest
      volumeMounts:
        - name: config-volume
          mountPath: /etc/config
  volumes:
    - name: config-volume
      configMap:
        name: app-config
```

## Namespaces (Isolate Resources in Kubernetes)

Namespaces **logically separate Kubernetes resources** in the same cluster.

💡 **Why use Namespaces?**

✅ Helps **organize** resources in large clusters.
✅ Enables **RBAC (Role-Based Access Control)** per namespace.
✅ Supports **resource quotas** per team/project.

**Namespace.yaml**

```yaml
apiVersion: v1
kind: Namespace
metadata:
  name: dev-environment
```

**Declarative creating namespace:**

**Kubeclt create ns <ns name>**

**Secrets.yaml**

```yaml
apiVersion: v1
kind: Secret
metadata:
  name: my-secret
type: Opaque
data:
  username: YWRtaW4=    # Base64 "admin"
  password: cGFzc3dvcmQxMjM=  # Base64 encoded "
```

**Encoding base64**

```
echo -n "admin" | base64  # YWRtaW4=
echo -n "password123" | base64  # cGFzc3dvcmQxMjM=
```

### 🔷 Secrets in Kubernetes:

📌 **What is a Secret?**

A **Secret** is a Kubernetes object used to **store sensitive data** such as passwords, API keys, or TLS certificates securely.

💡 **Why use Secrets instead of ConfigMaps?**

✅ **Encoded (Base64) data** for security.
✅ **Prevents storing secrets in container images**.
✅ **Can be mounted as files or environment variables**.

## 📌 What is a Service in Kubernetes?

A **Service** is a Kubernetes resource that **exposes** a set of **Pods** as a **network-accessible endpoint**.
Since Pods are **ephemeral** (they can be restarted, deleted, or rescheduled), a Service ensures **stable communication** between them.

## 💡 Why Use Services?

✅ Provides a **stable IP address & DNS** name.
✅ Load-balances traffic to multiple Pods.
✅ Allows external access to internal applications.
✅ Enables communication between different components in a cluster.

### 🔹 Types of Kubernetes Services

| Service Type | Use Case | Accessible From | Example |
|---|---|---|---|
| **ClusterIP** (default) | Internal communication | Inside the cluster only | Microservices |
| **NodePort** | Expose service on a fixed port | Any node's IP + port | Basic external access |
| **LoadBalancer** | Publicly expose service | Internet via cloud provider | Web applications |
| **ExternalName** | Maps service to an external domain | External (DNS resolution only) | Connecting to external DBs |

**ClusterIP.yaml**

```
apiVersion: v1
kind: Service
metadata:
  name: my-clusterip-service
spec:
  selector:
    app: my-app
  ports:
    - protocol: TCP
      port: 80      # Service port
      targetPort: 8080  # Pod's container
port
  type: ClusterIP
```

LoadBalancer

```
apiVersion: v1
kind: Service
metadata:
  name: loadbalancer-service
spec:
  selector:
    app: myapp
  ports:
    - protocol: TCP
      port: 80
      targetPort: 8080
  type: LoadBalancer
```

NodePort Service

```
apiVersion: v1
kind: Service
metadata:
  name: nodeport-service
spec:
  selector:
    app: myapp
  ports:
    - protocol: TCP
      port: 80
      targetPort: 8080
      nodePort: 30000
  type: NodePort
```

A **ClusterIP** service exposes the application on an internal IP within the Kubernetes cluster.
A **LoadBalancer** service exposes the application externally and can be used with cloud providers to allocate an external IP.
A **NodePort** service exposes the application on a static port on each node's IP, which can be accessed externally.

# 🚀 Understanding Pod Disruption Budgets (PDB) in Kubernetes! 🚀

In a highly available microservices environment, maintaining application stability during voluntary disruptions (like node upgrades, scaling, or maintenance) is critical. This is where Pod Disruption Budgets (PDBs) come into play!

## 🔷 What is a PDB?

A Pod Disruption Budget ensures that a minimum number of pods remain available during voluntary disruptions. It helps maintain service availability when nodes are drained or updated.

## 🔷 Key PDB Parameters:

✅ minAvailable → The minimum number of pods that must be running.
✅ maxUnavailable → The maximum number of pods that can be disrupted at a time

**when Nodes(blue) are cordon:**

| **PDB** | **Deployment:** | **pod status:** |
|---|---|---|
| minAvailable: 3 | replicas: 5 | 3 running 2 pending |
| minAvailable: 1 | replicas: 5 | 1 Running 4 pending |
| maxUnavilable: 2 | replicas: 5 | 3 running 2 pending |

**pdb.yaml:**

```
apiVersion: policy/v1
kind: PodDisruptionBudget
metadata:
  name: pdb-minavailable-count
spec:
  minAvailable: 2        // maxUnavailable: 1
  selector:
    matchLabels:
      app: myapp
```

| Command  NODE/POD Level | Purpose |
|---|---|
| kubectl exec -it <pod-name> -- /bin/sh | 🔍 Get into container shell (fix config or test app manually) |
| kubectl debug node/<node-name> --image=busybox | 🛠 Debug node with ephemeral container |
| kubectl describe events | 🧠 View all recent events for any debugging |
| kubectl get events --sort-by=.metadata.creationTimestamp | 🔎 Events sorted by time (great for finding root cause) |
| kubectl taint nodes <node> key=value:NoSchedule | 🚫 Prevent pods from scheduling on specific node (node maintenance) |

| Command CPU/metrics | Purpose |
|---|---|
| kubectl top pod --containers | 📊 See CPU/memory usage per container in pod |
| kubectl top node --use-protocol-buffers | More accurate node metrics |
| kubectl get --raw "/apis/metrics.k8s.io/v1beta1/nodes" | Raw metrics endpoint — useful for scripting |

| Command | Purpose |
|---|---|
| kubectl get sa | List service accounts in current namespace |
| kubectl describe sa <name> | Show tokens and bindings |
| kubectl get roles,rolebindings,clusterroles,clusterrolebindings | Full RBAC scope |
| kubectl create secret generic <name> --from-literal=password=admin123 | Create a basic secret |
| `kubectl get secrets -o jsonpath="{.items[*].data}" | base64 decode secrets manually` |

## 🌐 What is Ingress in Kubernetes?

An **Ingress** is an API object in Kubernetes that manages **external access to services** in a cluster, typically HTTP/HTTPS routes.
It lets you define **rules for routing traffic** to backend services based on the request **host or path**—think of it like a smart router for your services.

## ⚙️ What is an Ingress Controller?

An **Ingress Controller** is the **actual implementation** that reads the Ingress rules and **configures a reverse proxy** (like NGINX, Traefik, etc.) to enforce those rules.
🔷 Without an **Ingress Controller**, your Ingress rules do nothing.

## 🧠 How it works:

You create an Ingress resource with rules (e.g., route /api to Service A and /web to Service B).
The Ingress Controller sees those rules and configures its proxy (e.g., NGINX) to implement them.
Incoming requests hit the Ingress Controller's LoadBalancer IP.
The controller routes the request based on the Ingress rules.

## Ingress controller installation:

```
kubectl apply -f https://raw.githubusercontent.com/kubernetes/ingress-nginx/controller-v1.9.1/deploy/static/provider/cloud/deploy.yaml
```

| Component | Description |
|---|---|
| **Ingress** | Defines routing rules to expose internal services |
| **Ingress Controller** | Listens for Ingress resources and configures the proxy |
| **NGINX/Traefik** | Popular Ingress Controllers (implement the logic) |
| **LoadBalancer/NodePort** | Expose the Ingress Controller to the internet |

**Note:** Ingress by itself cannot do anything. It relies on the presence of an **Ingress Controller**. Once an Ingress Controller is installed, it watches for Ingress resources and generates all the necessary configurations internally. The Ingress communicates with the Ingress Controller, which then routes traffic based on **host**, **path**, or **network-based** rules defined in the Ingress resource.

### 🔄 Relationship between Ingress and Ingress Controller

- **Ingress** is just a **set of rules** that define how external HTTP/HTTPS traffic should be routed to services inside your Kubernetes cluster.
- **Ingress Controller** is the **actual implementation** (like NGINX, Traefik, or AWS ALB) that **reads those Ingress rules** and configures a reverse proxy to route the traffic accordingly.

### ⚙️ How it works:

1. You define an **Ingress** resource with routing rules.
2. The **Ingress Controller** constantly watches the Kubernetes API for new or updated Ingress rules.
3. It **generates configuration** (e.g., for NGINX) based on those rules.
4. It **handles incoming requests** and routes them to the correct service.

```yaml
# ingress.yaml
apiVersion: networking.k8s.io/v1
kind: Ingress
metadata:
  name: hello-ingress
  annotations:
    nginx.ingress.kubernetes.io/rewrite-target: /
spec:
  rules:
  - host: hello.example.com
    http:
      paths:
      - path: /
        pathType: Prefix
        backend:
          service:
            name: hello-service
            port:
              number: 80
```

## StatefullSet and Stateless vs DeamonSet:

🧱 **StatefulSet:**

👉 Use When:

You need persistent storage, stable network identities, or ordered deployment (e.g., for databases like MongoDB, Cassandra, Kafka).

✅ Features:

Persistent Volumes: Each pod gets its own storage (PVC) that persists across restarts.

Stable DNS names: Pods get predictable names like pod-0, pod-1 etc.

Ordered scaling: Pods are created/terminated in order.

🔄 Example Use Case:

Databases, message brokers, clustered applications.

⚙️ **Stateless (Deployment):**

👉 Use When:

You don’t need to store data locally, and pods can be easily replaced. Most web apps, APIs, or frontend services fall here.

✅ Features:

Pods are interchangeable (no identity).

No guaranteed order of deployment or termination.

Easy scaling and rolling updates.

🔄 Example Use Case:

Web servers, REST APIs, microservices.

## 📦 DaemonSet:

👉 **Use When:**

You want to run **one pod on every (or selected) node** in the cluster.

✅ **Features:**

- Automatically deploys one pod per node.
- Used for **cluster-wide services** like monitoring, logging, networking agents.
- Pods automatically appear on **new nodes** added to the cluster.

🔄 **Example Use Case:**

- **Fluentd/Logstash** for logging.
- **Prometheus Node Exporter** for monitoring.
- **CNI plugins**, or security agents.

🧠 StatefulSet = For apps that need identity + persistent data.
⚡ Deployment = For stateless apps that can be scaled easily.
📡 DaemonSet = For running a pod on every node, usually for system-level tasks.

| Feature | StatefulSet | Deployment (Stateless) | DaemonSet |
|---|---|---|---|
| **Pod Identity** | Sticky, stable (e.g., pod-0) | No identity (all equal) | One pod per node |
| **Storage** | Unique, persistent per pod | Shared or none | Optional, often none |
| **Use Case** | Databases, Kafka | Web apps, APIs | Monitoring, logging agents |
| **Scaling** | Ordered | Random | Based on node count |
| **Pod Per Node** | No | No | Yes |

**StatefulSet YAML (with PVC)**

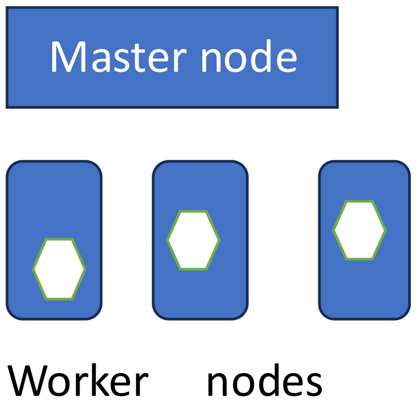
```yaml
apiVersion: apps/v1
kind: StatefulSet
metadata:
  name: my-db
spec:
  serviceName: "my-db-headless"
  replicas: 3
  selector:
    matchLabels:
      app: my-db
  template:
    metadata:
      labels:
        app: my-db
    spec:
      containers:
        - name: db
          image: mongo:5
          ports:
            - containerPort: 27017
          volumeMounts:
            - name: db-storage
              mountPath: /data/db
  volumeClaimTemplates:
    - metadata:
        name: db-storage
      spec:
        accessModes: ["ReadWriteOnce"]
        resources:
          requests:
            storage: 1Gi
```

**Stateless Deployment YAML:**

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

**DaemonSet YAML:**

```yaml
apiVersion: apps/v1
kind: DaemonSet
metadata:
  name: node-monitor
spec:
  selector:
    matchLabels:
      name: node-monitor
  template:
    metadata:
      labels:
        name: node-monitor
    spec:
      containers:
        - name: node-exporter
          image: prom/node-exporter:latest
          ports:
            - containerPort: 9100
```




**Through DaemonSet POD schedule In all Nodes**

**A pod is created on every node automatically.**

**Useful for cluster-wide agents like metrics/log collection.**

## Ways to Schedule Pods on Selected or All Nodes:

1. **DaemonSet**

```
kind: DaemonSet
spec:
  template:
    spec:
      nodeSelector:
        node-role.kubernetes.io/worker: ""
```

📌 **Use Case**: Logging agents, monitoring exporters (e.g., Prometheus Node Exporter).

2. **nodeSelector** – Run pods on SELECTED nodes

Assign pods to specific nodes using **key-value labels**.

```
spec:
  nodeSelector:
    disktype: ssd
```

📌 **Use Case**: Run a database only on nodes with SSDs.

3. **nodeAffinity** – More advanced node matching

📌 **Use Case**: Better control and flexibility over scheduling compared to nodeSelector.

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

4. **Taints & Tolerations** – Allow or prevent pods from running on nodes

> Taints: Mark a node as restricted.

> Tolerations: Allow pods to "tolerate" the taint and run there.

```
# Taint on node:
kubectl taint nodes node1
key=value:NoSchedule

# Toleration in pod:
tolerations:
- key: "key"
  operator: "Equal"
  value: "value"
  effect: "NoSchedule"
```

📌 **Use Case**: Control workloads (e.g., only system pods on master nodes).

## 🌐 What is CoreDNS?

**CoreDNS** is the **default DNS server** used by Kubernetes (since v1.13+) for **cluster DNS resolution**. It helps pods and services **communicate using names** instead of IP addresses.

| Term | Meaning |
|---|---|
| **DNS** | Domain Name System – converts names to IPs |
| **CoreDNS** | A DNS server written in Go and designed to be extensible |
| **kube-dns** | The older DNS service (replaced by CoreDNS) |

### 🔧 What CoreDNS Does in Kubernetes

Listens for DNS queries from pods.
Resolves service names like my-svc.default.svc.cluster.local to a cluster IP.
Works with kubelet and kube-proxy for name-to-IP mapping.
Uses plugins (like kubernetes, forward, cache, etc.) for flexibility.

### How resolution works:

CoreDNS receives the query: "What is the IP of my-svc.default.svc.cluster.local?"
It then looks up the Kubernetes API to find the service my-svc in the default namespace.
If found, it returns the ClusterIP (e.g., 10.97.53.42) to the pod that requested it.
The pod uses this IP to connect to the service.

**Corefile (CoreDNS Configuration):**

```
.:53 {
   errors
   health
   kubernetes cluster.local in-addr.arpa ip6.arpa {
      pods insecure
      fallthrough in-addr.arpa ip6.arpa
   }
   forward . /etc/resolv.conf
   cache 30
   loop
   reload
   loadbalance
}
```

1 Listens for DNS queries from Pods

Every Pod in Kubernetes has a file called /etc/resolv.conf that tells it where to send DNS queries.
This file usually contains:

**nameserver 10.96.0.10  # IP of the CoreDNS service (cluster IP)**

When a pod wants to communicate with another service (e.g., my-service.default.svc.cluster.local), it performs a DNS query.
That DNS query is sent to CoreDNS, which is running as a service in the kube-system namespace.

2 Resolves service names like my-svc.default.svc.cluster.local to ClusterIP

Example Breakdown:

my-svc.default.svc.cluster.local

- my-svc: the name of the service
- default: the namespace
- svc: indicates it's a service
- cluster.local: the cluster domain (default DNS suffix)

✅ This enables **service discovery** using names instead of hardcoded IPs.

**3 Works with kubelet and kube-proxy for name-to-IP mapping**

| Component | Role |
| --- | --- |
| **kubelet** | Configures the pod's networking and injects the correct DNS settings (like pointing to CoreDNS IP). |
| **kube-proxy** | Creates IP tables or IPVS rules so that the **ClusterIP returned by CoreDNS** routes traffic to one of the backend pods for that service. |

**4 Uses Plugins for Flexibility**

CoreDNS is **modular**, and its behavior is defined in the **Corefile** (usually stored in the CoreDNS ConfigMap).
Common plugins and their roles:

```yaml
apiVersion: v1
kind: ConfigMap
metadata:
  name: coredns
  namespace: kube-system
data:
  Corefile: |
    .:53 {
        errors
        health
        kubernetes cluster.local in-addr.arpa ip6.arpa {
          pods insecure
          fallthrough in-addr.arpa ip6.arpa
        }
        forward . /etc/resolv.conf
        cache 30
        loop
        reload
        loadbalance
    }
```

### 📈 How a Pod Uses CoreDNS

Pod is created with /etc/resolv.conf pointing to CoreDNS.
When a pod tries to connect to my-service.default.svc.cluster.local, it sends a DNS query.
CoreDNS checks its internal rules and returns the correct IP.
The pod uses that IP to connect to the service.

- 🔷 .:53 means listen on port 53 for all domains.
- 🔷 forward . /etc/resolv.conf = forward unknown queries (e.g., google.com) to the host's DNS.

## Plugins for Flexibility:

| Plugin | What it does |
|---|---|
| kubernetes | Enables service and pod name resolution using the Kubernetes API. |
| forward | Forwards unresolved DNS queries to an external DNS (like Google DNS or the node's /etc/resolv.conf). |
| cache | Caches DNS responses to improve speed and reduce API load. |
| loop | Detects and prevents infinite DNS query loops. |
| health | Exposes an HTTP health endpoint for CoreDNS. |
| reload | Auto-reloads CoreDNS if its config changes. |
| loadbalance | Randomizes the list of A records returned (helps in load balancing across pods). |

## 📜 1. What is a Service Account in Kubernetes?

A ServiceAccount (SA) is an identity used by pods to interact with the Kubernetes API.

### 🧠 Every pod runs as a service account (default or custom).

This account is automatically mounted into the pod as a token in /var/run/secrets/kubernetes.io/serviceaccount/.

### 🧷 Why is it needed?

Pods often need to interact with the Kubernetes API (e.g., to read config maps, list pods, watch services).
Instead of using user credentials (bad practice), they use service accounts.

**ServiceAccount.yaml**

```yaml
apiVersion: v1
kind: ServiceAccount
metadata:
  name: custom-sa
  namespace: default
```

```yaml
apiVersion: v1
kind: Pod
metadata:
  name: mypod              #POD Attaching in Service Account
spec:
  serviceAccountName: custom-sa
  containers:
   - name: nginx
     image: nginx
```

```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
 name: app-deploy
spec:
 replicas: 2
 selector:
  matchLabels:
   app: testapp
 template:
  metadata:
   labels:
    app: testapp
  spec:
   serviceAccountName: custom-sa
containers:
   - name: app
    image: nginx
```

### 🔁 How a Deployment Uses a Service Account

When you create a Deployment, each Pod it creates will use a Service Account to interact with the Kubernetes API.

✅ By default, pods use the default service account in the same namespace, unless overridden.

## 🔐 How RBAC Protects Kubernetes Resources Using Service Accounts

RBAC controls **what the service account is allowed to do**.

If your app inside the pod tries to:

- List all pods
- Get secrets
- Create deployments
- Access nodes

🛑 It will **only succeed** if **RBAC allows it**.

## 🔐 What is RBAC (Role-Based Access Control)?

**RBAC** lets you define **what actions** (verbs) a user, group, or service account can perform on **which resources** in Kubernetes.

RBAC works using 4 main components:

| Component | What it does |
|---|---|
| **Role** | Defines permissions within a **namespace** |
| **ClusterRole** | Defines cluster-wide permissions (or reusable across namespaces) |
| **RoleBinding** | Grants a Role to a user/SA **in a namespace** |
| **ClusterRoleBinding** | Grants a ClusterRole to a user/SA across the **entire cluster** |

## 🔷 Create Role:

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

## 🔷 Bind Role to a ServiceAccount:

```
apiVersion: rbac.authorization.k8s.io/v1
kind: RoleBinding
metadata:
  name: read-pods-binding
  namespace: default
subjects:
- kind: ServiceAccount
  name: custom-sa
  namespace: default
roleRef:
  kind: Role
  name: pod-reader
  apiGroup: rbac.authorization.k8s.io
```

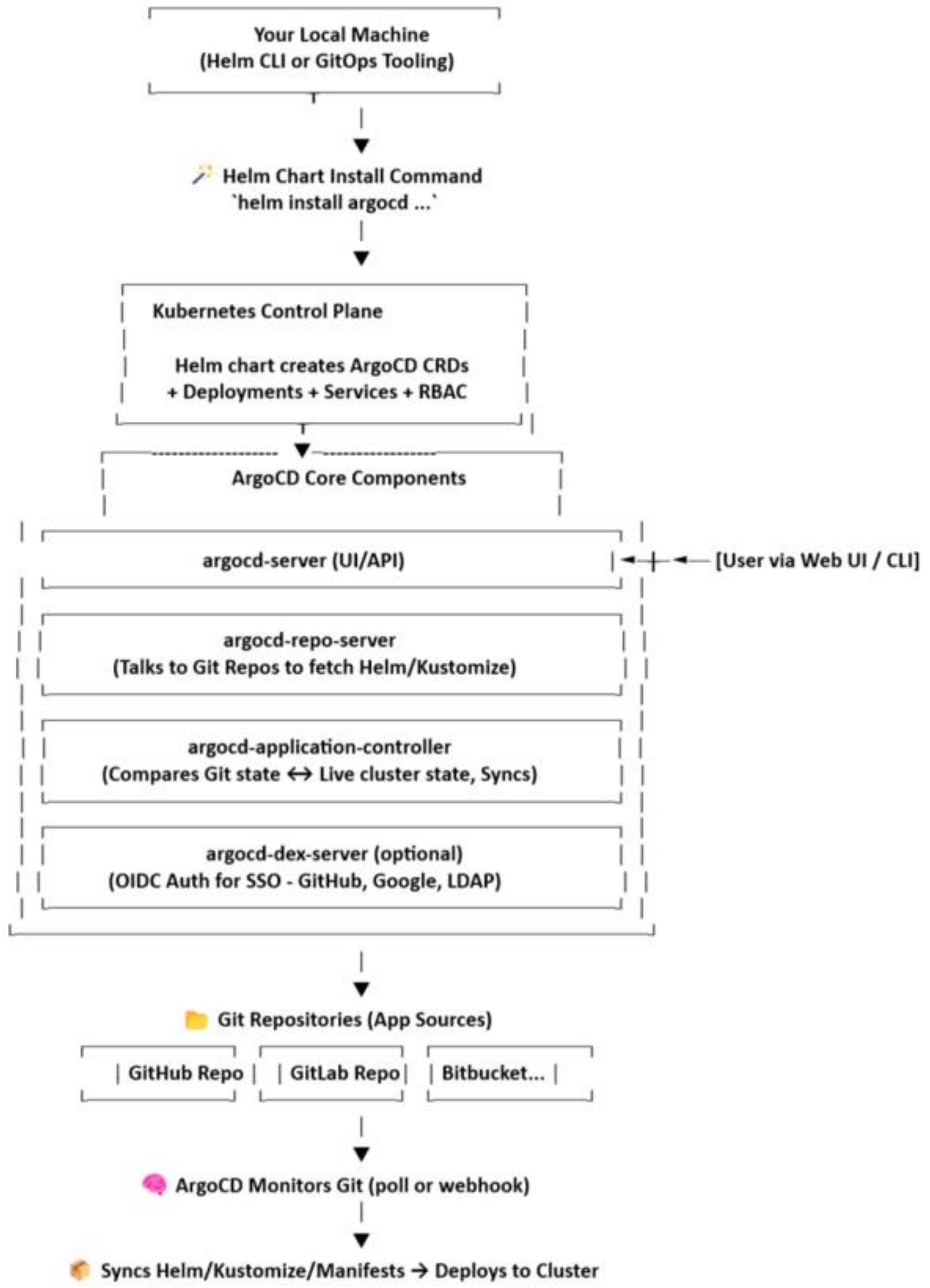
🔐 **Now custom-sa can only read pods in the default namespace.**

| Task | Use |
|---|---|
| Give pod permission to list secrets | Create SA + Role + RoleBinding |
| Let Jenkins/ArgoCD deploy to namespaces | Use ClusterRole + ClusterRoleBinding |
| Protect critical namespaces (e.g., kube-system) | Use RBAC to restrict access |

## Service aaccount Deployment RBAC

```
+------------------------------+
|     Deployment              |
|  Uses SA: cm-reader-sa |
+---------------+------------+
               |
               v
      +------+------+
      |    Pod       |
      |  SA Token    |
      +------+------+
               |
               v
+-----------------------+
|  Kubernetes API Server |
+-----------+-----------+
               |
      [Authorization Check]
               |
+---------------+-----------------+
|                          |
✅ Allowed               ❌ Denied
(Role/RoleBinding Exists)    (No matching Role/Binding)
```

## Argocd Deployment



## 📌 What is Helm?

Helm is a **package manager for Kubernetes** that simplifies deployment and management of applications using **Helm charts**. It allows you to:

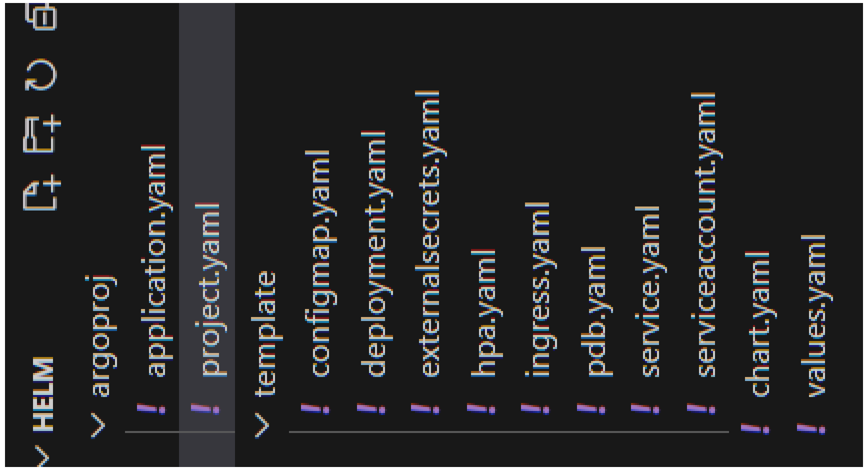
✅ Deploy applications with a **single command**
✅ Use **templates** to manage configurations
✅ **Version control** your deployments
✅ **Easily upgrade/rollback** applications

📌 Key Concepts of Helm

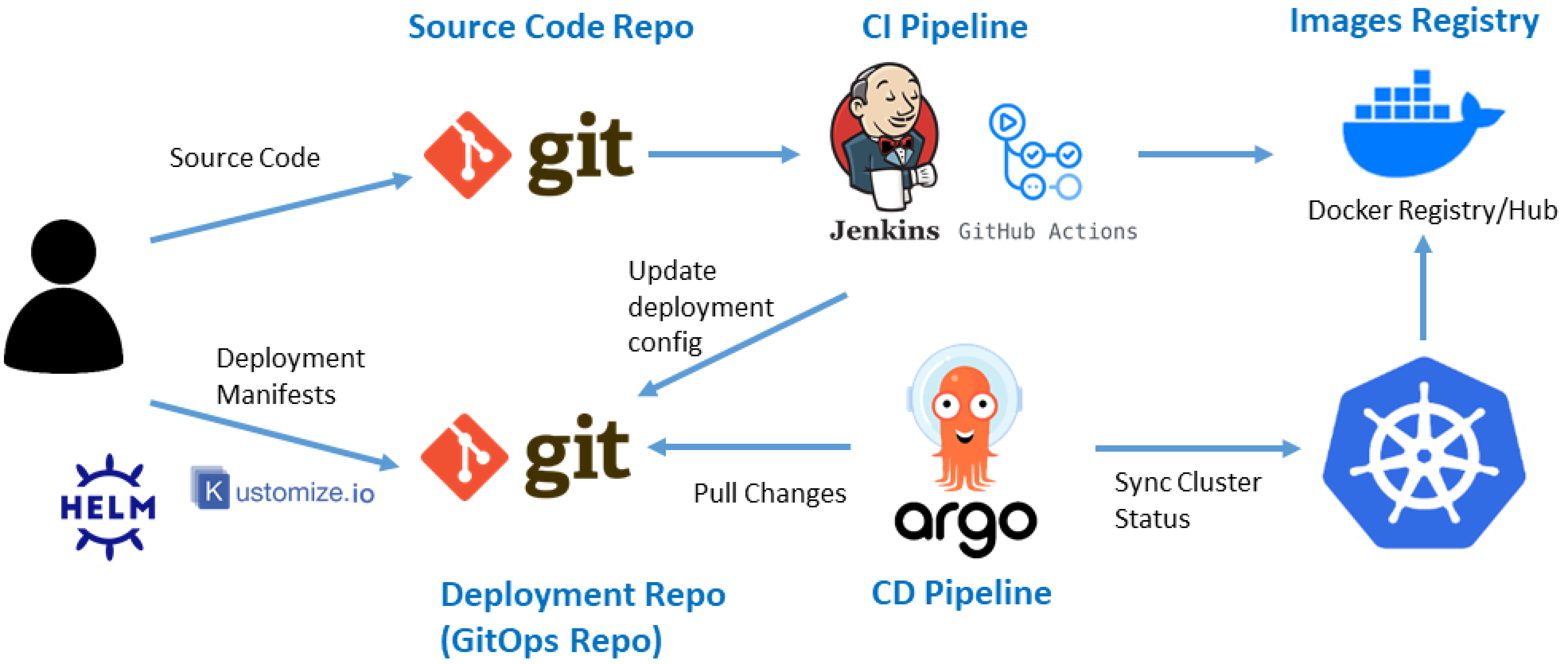
Helm Charts – Pre-packaged application definitions (like a blueprint).
Values.yaml – Configuration file for customizing deployments.
Templates/ – Kubernetes manifests with dynamic placeholders ({{ }}) for customization.
Releases – A deployed instance of a Helm chart.
Repositories – Where Helm charts are stored and shared.

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: {{ .Release.Name }}-app
spec:
  replicas: {{ .Values.replicaCount }}
  selector:
    matchLabels:
      app: {{ .Release.Name }}-app
  template:
    metadata:
      labels:
        app: {{ .Release.Name }}-app
    spec:
      containers:
        - name: {{ .Chart.Name }}
          image: "{{ .Values.image.repository }}:{{ .Values.image.tag }
          ports:
            - containerPort: 8080
```

**<u>My manifest:</u>**



## Application Deployment Process:




Thank You.....!

B SURYA PRAKASH REDDY