There are two main parts:

* **Control Plane (Master components)** — manages the cluster.
* **Worker Nodes** — run the actual applications (your containers).
* "**Kubernetes Control Plane manages the cluster, and the Worker Nodes run the actual applications (Pods) based on the Control Plane's instructions.**"

A Pod has:

* **One or more Containers** (usually Docker containers, but not limited to Docker)
* **Storage Volumes** (optional shared storage)
* **A unique IP address**

|  |  |
| --- | --- |
| **ReplicaSet** | Ensure a fixed number of Pod replicas are running. |

|  |  |
| --- | --- |
| **DaemonSet** | Run one Pod on **each Node** (e.g., logging agent). |

|  |  |
| --- | --- |
| **StatefulSet** | **Manage stateful applications** with persistent IDs. |

Pods group **one or more containers together into a single unit** with shared networking and storage. **They are ephemeral** and are usually managed by controllers like Deployments. **Pods help Kubernetes treat containers more smartly** and robustly.

In pod our actual appn run , pod contain one or more contrianers inside them, we deployed our appn isndie pod , pod can communicate with each other.

## What is a ReplicaSet?

* A **ReplicaSet** is a **Kubernetes controller** whose main job is:
* Replica set **ensure that right number of pods are always running**. If any pod dead then RS automatically create a new one.

If a Pod dies → ReplicaSet **automatically creates a new one**.  
If extra Pods appear → ReplicaSet **deletes the extras.**

🔵 **ReplicaSet = Self-healing Pod manager**.

**ReplicaSet** is that "something" that **continuously monitors and maintains** the desired state (desired number of Pods).

Key Features of ReplicaSet

| **Feature** | **Description** |
| --- | --- |
| **Self-Healing** | Replaces failed Pods automatically. |
| **Scalability** | Can scale Pods up or down easily. |

## How is ReplicaSet different from Pod?

| **Feature** | **Pod** | **ReplicaSet** |
| --- | --- | --- |
| Responsibility | Run one instance of your app | Ensure multiple replicas |
| Recovery | No automatic recovery | **Automatic self-healing** |
| Scalability | Manual | **Automatic scaling** (changing replicas) |

* **ReplicaSet** manages Pods.
* **Deployment** manages ReplicaSets.
* **Important:**  
  When you create a **Deployment**, **it automatically creates a ReplicaSet** inside.

# Final 1-line Summary

* "**ReplicaSet ensures that a specified number of Pods are running at any time, by monitoring and maintaining the Pods automatically. It is mostly used by Deployments in Kubernetes today.**"
* RS ensure that right no of pods are always running

| **Feature** | **ReplicaSet** | **Deployment** | **StatefulSet** |
| --- | --- | --- | --- |
| **Purpose** | **Ensure a fixed number of identical Pods** are running | Manage ReplicaSets + handle updates/rollbacks | Manage stateful applications (ordered Pods, persistent storage) |
| **Manages** | Only Pods | **ReplicaSets (and indirectly** Pods) | Stateful Pods (with unique identities) |
| **Pod Identity** | All Pods are identical | All Pods are identical | Each Pod has a **stable, unique identity** (like pod-0, pod-1) |
| **Updates/Rollbacks** | Manual (not easy) | **Easy (rolling updates, rollback to previous version)** | Controlled, ordered updates |

## What is a Deployment?

* A **Deployment** is a **higher-level Kubernetes controller**.
* It manages **ReplicaSets** — which in turn manage **Pods**.

Deployment makes it easy:

* **Rolling updates** without downtime
* **Rollbacks** if something fails
* **Automatic scaling** and management
* **Version history** maintained

## Key Features of Deployment

| **Feature** | **Description** |
| --- | --- |
| **Rolling Updates** | Update Pods gradually without downtime |
| **Rollback** | Go back to a previous working version |
| **Scaling** | Scale replicas easily |
| **Self-Healing** | Replaces unhealthy Pods automatically |

**Deployment manages ReplicaSets, ReplicaSets manage Pods.**"

| **Feature** | **Description** |
| --- | --- |
| **Rolling Update** | **Slowly replace old Pods with new on**es — no downtime.  Replace old pods with new ones |
| **Rollback** | **Revert to a previous ReplicaSet version automatically** if the new one fails. |

# Why not just update existing Pods?

Because:

* **Pods are immutable** in Kubernetes (you can’t modify running Pods for things like images, env variables, etc.).
* For safety and consistency:  
  ➔ "Replace" is better than "mutate."

**Whenever a Deployment YAML is updated (changing template spec), Kubernetes creates a new ReplicaSet** with new Pods and gradually replaces the old Pods. It never updates existing Pods directly because Pods are immutable."

## What is NodeSelector?

**NodeSelector** is the **simplest way** to **tell Kubernetes to run a Pod on a specific node (or a group of nodes)** based on **labels**.

## How does NodeSelector work?

* You add **labels** to nodes manually (or during node setup).
* In the Pod's YAML file, you use **nodeSelector** to specify which labeled node the Pod should land on.

if the interviewer asks **"What if no node matches NodeSelector?"**

✅ Answer: "**Pod will stay Pending forever, Kubernetes won't reschedule automatically unless you change something manually.**"

## 🌟 What is a DaemonSet?

**DaemonSet** ensures that **a copy of a specific Pod is running on ALL (or specific) Nodes** in the Kubernetes cluster.

✅ One Pod **per Node**.  
✅ Automatically adds Pod to **new Nodes**.  
✅ Automatically removes Pod from **deleted Nodes**

## Why do we need DaemonSets?

Think about stuff like:

* Log collection agents (like Fluentd, Logstash).
* Monitoring agents (like Prometheus Node Exporter, Datadog agent).

## How DaemonSet is different from Deployment?

| **Aspect** | **DaemonSet** | **Deployment** |
| --- | --- | --- |
| Purpose | Run one Pod **per node** | Run specified **number** of Pods (replicas) |
| Scaling | Automatically matches node count | Manual or auto-scaling (HPA) |

## How to restrict DaemonSet to specific nodes?

Use:

* **nodeSelector** (simple matching)
* **nodeAffinity** (advanced matching)
* **Tolerations** (to tolerate tainted nodes)
* **A DaemonSet ensures that a specific Pod runs on every (or selected) node in the cluster.  
  It is used mainly for cluster-wide services like logging agents, monitoring, or networking components.  
  It automatically manages Pods as Nodes join or leave the cluster.  
  DaemonSets maintain exactly one Pod per node, unlike Deployments.**"

**Run a copy of each pod on every node** – only single pod copy

## What is Node Affinity?

**Node Affinity** is a way to **control where Pods are scheduled**, based on **rules** about node labels.

✅ It’s **an advanced and flexible version of NodeSelector**.

**In short:**  
**Node Affinity** = **More powerful + more flexible scheduling**.

## Types of Node Affinity

| **Type** | **Meaning** | **requiredDuringSchedulingIgnoredDuringExecution or preferredDuringSchedulingIgnoredDuringExecution** |
| --- | --- | --- |
| **Required (Hard)** | Pod MUST be scheduled ONLY on matching nodes, or stay Pending. | requiredDuringSchedulingIgnoredDuringExecution |
| **Preferred (Soft)** | Pod SHOULD be scheduled on matching nodes if possible, but can be scheduled elsewhere too. | preferredDuringSchedulingIgnoredDuringExecution |

## What happens if no node matches Node Affinity?

✅ If it’s **required**, Pod stays in Pending state.  
✅ If it’s **preferred**, Pod still gets scheduled **somewhere else** if needed.

**Node Affinity allows Pods to specify rules based on Node labels for scheduling.  
It supports both hard (required) and soft (preferred) rules using flexible match expressions.**

## What are Resource Requests and Limits?

✅ In Kubernetes, **Requests** and **Limits** are used to **control** how much **CPU** and **Memory** a **Pod/Container** can use.

| **Term** | **Meaning** |
| --- | --- |
| **Request** | The minimum amount of resource that Kubernetes guarantees to the container. |
| **Limit** | The maximum amount of resource the container is allowed to use. |

Request = gruanteee it will use till this

Limit = maximum itne use kar skate hai

## What happens if container uses more than Limit?

* **CPU:**  
  If the container tries to use more CPU than its **limit**, Kubernetes will **throttle** it (slow it down), **not kill** it.
* **Memory:**  
  If it tries to use more **Memory** than the **limit**, the container **will be terminated** (OOMKilled — Out of Memory Kill).

✅ So, **memory** is stricter than **CPU**.

**Requests and Limits in Kubernetes define the minimum guaranteed and maximum allowed CPU/Memory for a container.  
Requests are used during scheduling, while Limits enforce usage control during runtime.**"

Both are used in production

## What is a Service in Kubernetes?

✅ A **Service** is a **stable network abstraction** that exposes **Pods** so that

**Pod IPs are not permanent**, but **Service IPs are stable**.

## Why do we need a Service?

| **Problem** | **How Service Helps** |
| --- | --- |
| Pods have **dynamic IP addresses** (change on restart). | Service gives a **fixed IP** and **DNS name**. |
| Hard to **load balance** between multiple Pods manually. | Service automatically **load balances** across Pods. |
| Need to **expose** app to outside the cluster. | Service types like **NodePort** or **LoadBalancer** expose apps outside. |

We can expose our pod to external world , using nodeprot –open port of pods as static pod

## How does a Service work internally?

✅ A Service **selects Pods** based on **labels** using a **selector**.

✅ It **forwards traffic** to healthy Pods automatically.

## Types of Services (VERY IMPORTANT)

| **Type** | **Purpose** | **Accessibility** |
| --- | --- | --- |
| **ClusterIP** (default) | Expose service **inside** the cluster only. | Internal only. |
| **NodePort** | Expose service on a **port** of each **Node’s IP**. | External (limited). |
| **LoadBalancer** | Expose service through a **cloud provider’s LoadBalancer**. | External, scalable. |

|  |  |
| --- | --- |
| Why do we need a Service? | To provide stable access to changing Pod IPs and load balancing. |
| * **To expose a pod to service** |  |
| What happens if Pod dies? | Service automatically routes to healthy Pods matching the label. |
| Default type of Service? | ClusterIP. |

**Headless Services** exist (by setting ClusterIP: None) when you **don't want load balancing**, but want to discover all Pod IPs directly

## What is NodePort Service?

✅ A **NodePort Service** in Kubernetes **exposes an application outside the cluster** by **opening a port** (called NodePort) **on every Node**.

* The Service listens on a specific **high port (30000–32767)** on the Node.
* When you send a request to **NodeIP:NodePort**, Kubernetes **forwards** it to the **Service**, and then to the **correct Pod**.

## Important Concepts

| **Term** | **Meaning** |
| --- | --- |
| **port** | Service port inside the cluster (example: 80). |
| **targetPort** | Pod's container port (example: 8080). |
| **nodePort** | Port exposed on each Node (range 30000–32767). |

|  |  |
| --- | --- |
| What is NodePort? | It exposes a service outside the cluster by opening a specific port on each Node. |

|  |  |
| --- | --- |
| Default NodePort range? | 30000–32767. |

## What is a LoadBalancer Service?

✅ A **LoadBalancer** Service **automatically provisions an external Load Balancer** (from your cloud provider — AWS, GCP, Azure)  
✅ This Load Balancer **forwards external traffic** directly to your **Kubernetes Service** inside the cluster.

## 2. 🧠 Why use LoadBalancer?

| **Reason** | **Explanation** |
| --- | --- |
| Publicly expose services | Make your app reachable globally (internet facing). |

**LoadBalancer Service provisions a public cloud Load Balancer outside the Kubernetes cluster. It forwards incoming external traffic to the Kubernetes Service, and then load-balances it across backend Pods.**"

| **Service Type** | **One-liner Summary** |
| --- | --- |
| **ClusterIP** | "Default **service for internal communication between apps** inside the cluster." |
| **NodePort** | "**Expose service on each Node’s IP at a static port** — simple external access."   * Expose serive on static node port ip 32667 |
| **LoadBalancer** | "Provision external cloud load balancer to expose services publicly." |
| **Headless** | "**Expose pods directly without load balancing** — useful for StatefulSets and service discovery." |

## What is Ingress?

✅ **Ingress** is a Kubernetes **API object**  
✅ It **manages external access to services** inside a Kubernetes cluster  
✅ It **routes HTTP/HTTPS traffic** to the correct Service **based on rules** (like URL path, hostname).

| **Component** | **Meaning** |
| --- | --- |
| **Ingress** | Defines rules: which URL/host → which Service |
| **Ingress Controller** | Software that reads Ingress objects and processes the traffic (examples: nginx, Traefik) |

|  |  |
| --- | --- |
| **Services & Pods** | Actual backends that receive the traffic |

## Very Important Notes (PRO Level for Interview)

* Ingress **only defines the rules**.  
  **Ingress Controller** actually implements it.
* You **must** install an Ingress Controller manually (unless your cloud does it for you).
* **pathType** (in YAML) can be:

 **NodePort** → Basic way to expose a Service on Node IP.

 **LoadBalancer** → External access through a Cloud LB.

 **Ingress** → Smart routing to multiple services with one IP.

| **Question** | **Quick Tip** |
| --- | --- |
| Does Ingress expose Services directly? | ❌ No, it routes traffic to Services. |
| Is Ingress enough by itself? | ❌ No, needs an Ingress Controller installed. |
| Can Ingress do HTTPS termination? | ✅ Yes, via TLS configuration. |

## What is a Service Account?

✅ A **Service Account** is a Kubernetes object that provides an **identity** for **pods** (or processes inside pods) to **interact with the Kubernetes API Server**.

**Simple Words:**  
Service Account =  
**Username/password for Pods** ➔ to **talk to Kubernetes API** ➔ securely.

## Default Behavior

✅ By default:

* Kubernetes creates a **default** Service Account in every namespace.
* If you don’t specify anything, your Pods use default Service Account.

| **Question** | **Quick Tip** |
| --- | --- |

|  |  |
| --- | --- |
| What is a Service Account? | Identity for pods to talk to Kubernetes API securely |

|  |  |
| --- | --- |
| How is it different from User Accounts? | Service Accounts are for apps, not humans |

|  |  |
| --- | --- |
| What is the default behavior? | Every namespace has a default Service Account |

# What is Helm in Kubernetes?

✅ **Helm** is a **package manager** for Kubernetes.  
(Just like apt for Ubuntu or yum for CentOS.)

It **helps you define, install, and upgrade** complex Kubernetes applications easily.

**Simple Words:**  
Without Helm → You have to write big YAML files manually.  
With Helm → You install apps with **one command** (helm install) and manage configurations smartly.

# Key Concepts in Helm

| **Concept** | **Meaning** |
| --- | --- |
| **Chart** | A Helm package. Contains YAML templates + values. |
| **Release** | A running instance of a chart inside Kubernetes. |
| **Repository** | A place where charts are stored (like DockerHub for images). |

# How Helm Works (Simple)

1. You create or download a **Chart**.
2. You **install it using** helm install.
3. Helm **converts templates into real Kubernetes** YAMLs and applies them.
4. Kubernetes resources (Pods, Services, Ingress, etc.) are created automatically.

# Basic Commands

| **Command** | **Purpose** |
| --- | --- |
| helm repo add | Add a Helm repository (like Bitnami charts) |
| helm search repo | Search for available charts |
| helm install <release-name> <chart-name> | Install a chart |
| helm upgrade <release-name> <chart-name> | Upgrade the release |

**Helm is a package manager for Kubernetes that simplifies deployment, management, and configuration of applications through reusable charts.**

# What is a Namespace in Kubernetes?

✅ A **Namespace** is a way to **divide** a Kubernetes cluster into **multiple virtual clusters** inside the same physical cluster.

* Logical separation
* Isolation between teams, projects, or environments
* Resource grouping and management

Each folder (namespace) can have its own Pods, Services, Deployments, etc.  
They can have same names too inside different namespaces without conflict!

# Important Default Namespaces

| **Namespace** | **Purpose** |
| --- | --- |
| default | If **you don't specify any namespace**, it goes here |
| kube-system | Core Kubernetes system components (kube-dns, kube-proxy) |
| kube-public | **Publicly readable data** (used very rarely) |
| kube-node-lease | Helps nodes send heartbeats **for faster failure detection** |

| **Question** | **Quick Tip Answer** |
| --- | --- |
| What are namespaces in Kubernetes? | Virtual clusters inside physical cluster for isolation. |
| Why do we need namespaces? | Separation, organization, resource control. |
| What happens if you don’t specify a namespace? | Resources go into the default namespace. |
| Can two pods with the same name exist? | Yes, in different namespaces. |

# What is Authorization in Kubernetes?

✅ **Authorization** decides **"whether a user is allowed to perform an action"** after **Authentication**.

* **Authentication** = Who are you?
* **Authorization** = Are you allowed to do this?

# Types of Authorization in Kubernetes

| **Authorization Mode** | **Meaning** | **Example** |
| --- | --- | --- |
| **RBAC** (Role-Based Access Control) | Assign permissions to users based on roles | Admin can create pods; Developer can only read pods |

# Role-Based Access Control (RBAC) in Kubernetes

RBAC controls **who can perform what action on which resource**.

| **Component** | **Meaning** |
| --- | --- |
| **Role** | Permissions (**within a namespace**) |
| **ClusterRole** | Permissions (**across entire cluster**) |
| **RoleBinding** | Assign a **Role to a user/group within a namespace** |
| **ClusterRoleBinding** | Assign a ClusterRole to user/group for the whole cluster |

| **Question** | **Tip Answer** |
| --- | --- |
| What is Authorization in Kubernetes? | It checks if a user is allowed to perform an action after authentication. |
| Name different authorization modes. | RBAC, ABAC, Webhook, Node Authorization. |
| What is RBAC? | Role-Based Access Control based on user roles and permissions. |
| What is a Role vs ClusterRole? | Role = namespace-specific; ClusterRole = cluster-wide. |
| What is RoleBinding vs ClusterRoleBinding? | RoleBinding = binds Role to user in a namespace; ClusterRoleBinding = binds ClusterRole cluster-wide. |

# What is a **ClusterRole**?

| **Feature** | **Description** |
| --- | --- |
| Scope | Cluster-wide (NOT limited to a namespace) |
| Defines | What **actions** can be performed on **resources** (pods, nodes, namespaces, etc.) |

# What is a **ClusterRoleBinding**?

| **Feature** | **Description** |
| --- | --- |
| Scope | Cluster-wide |
| Purpose | **Binds a ClusterRole** to a **user**, **group**, or **service account** |
| Effect | Grants the permissions of the ClusterRole **across the entire cluster** |

| **Question** | **Answer** |
| --- | --- |
| What is a ClusterRole? | **A Role that grants permissions across the entire Kubernetes cluster.** |
| What is a ClusterRoleBinding? | A binding that attaches a ClusterRole to a **user/group/service account cluster-wide.** |
| Can a ClusterRole be used in a namespace? | Yes, using a RoleBinding. |
| When would you use ClusterRole vs Role? | Use ClusterRole when access is needed across all namespaces or for cluster-level resources. |

**ClusterRole defines cluster-wide permissions, while ClusterRoleBinding connects users to those permissions across the whole Kubernetes cluster.**"

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

| **Feature** | **Description** |
| --- | --- |
| Type | A piece of **storage** in the cluster |
| Provisioned By | Admin OR Dynamically |
| Lifespan | **Independent of Pod lifecycle** (survives pod restart/crash) |

**Simple line:** PV is the actual storage resource in the cluster.

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

| **Feature** | **Description** |
| --- | --- |
| Type | A **request** for storage |
| Created By | Developer/User |
| Purpose | Request specific storage requirements (size, access mode) |

**Simple line:** PVC is a request that asks Kubernetes for some storage.

## What is a **ConfigMap**?

| **Feature** | **Description** |
| --- | --- |
| Purpose | To **store non-sensitive configuration data** (key-value pairs) |
| Type | Kubernetes object |
| Scope | Application Configuration (NOT secrets) |

## Why do we need ConfigMaps?

* Imagine you have an app where:
  + Database URL
  + Log level
  + Feature toggles  
    are **configurable** values.

Instead of **hardcoding** them into your app, store them in a **ConfigMap**.  
This way, you can **change configuration without rebuilding the container**.

**Secrets** are for sensitive data (encrypted), **ConfigMaps** are for normal config.

| **Question** | **Quick Answer** |
| --- | --- |
| What is ConfigMap? | Object to store config data |
| Difference between ConfigMap and Secret? | ConfigMap = non-sensitive, Secret = sensitive and encrypted |
| How to inject ConfigMap to pod? | Environment variable / Volume |
| Does pod automatically update when ConfigMap updates? | No, pod restart is needed |

"**ConfigMaps** store non-sensitive configuration data separately from application code and are injected into pods via environment variables or mounted volumes, enabling flexible and scalable application management."

## What is a Secret?

| **Feature** | **Description** |
| --- | --- |
| Purpose | To **store sensitive data securely** like passwords, API keys, SSH keys, tokens |
| Type | Kubernetes object |
| Scope | Sensitive configuration |

✅ **One line:**  
Secrets store ***sensitive information*** that you don't want to expose inside pod specifications or container images.

## Why do we need Secrets?

* Storing **passwords, tokens, or keys directly in pod specs** is insecure.
* Secrets ensure sensitive data is:
  + **Encoded** (at least base64)
  + **Better protected** (can be encrypted at rest in etcd)
  + **Access controlled** (using RBAC)

# Secrets vs ConfigMaps

| **Feature** | **ConfigMap** | **Secret** |
| --- | --- | --- |
| Use Case | Non-sensitive data | Sensitive data |
| Encoding | Plain text | Base64-encoded |

**Secrets** securely manage sensitive information like passwords, API keys, and tokens in Kubernetes, allowing injection into pods while ensuring protection through base64 encoding, optional encryption, and RBAC-controlled access."