Chatgpt link

Kubernetes notes

Monolithic Architecture vs Microservices Architecture

🧱 Monolithic Architecture (All-in-One)

- In a **monolithic** architecture, the whole application is built as **one single unit**.
- All features like login, database access, UI, etc., are tightly connected and **run together**.

- Example: Like one big container ship—if one part breaks, the whole ship is affected.

Pros:

- Easy to develop at the beginning.
- Simple to deploy (only one deployment).

Cons:

- Hard to update or scale individual parts.
- If one part fails, the whole app might go down.
- Not flexible for large teams.

⚙️ Microservices Architecture (Split into Parts)

- In a microservices architecture, the app is broken into small, independent services.
- Each service handles a specific task (e.g., login, search, billing) and can be developed and deployed separately.

- Example: Like a team of small boats—each boat does a job, and one boat failing doesn't stop the rest.

Pros:

- Easy to scale and update parts independently.
- Better for large teams and complex apps.
- More reliable (one failure doesn't crash the whole app).

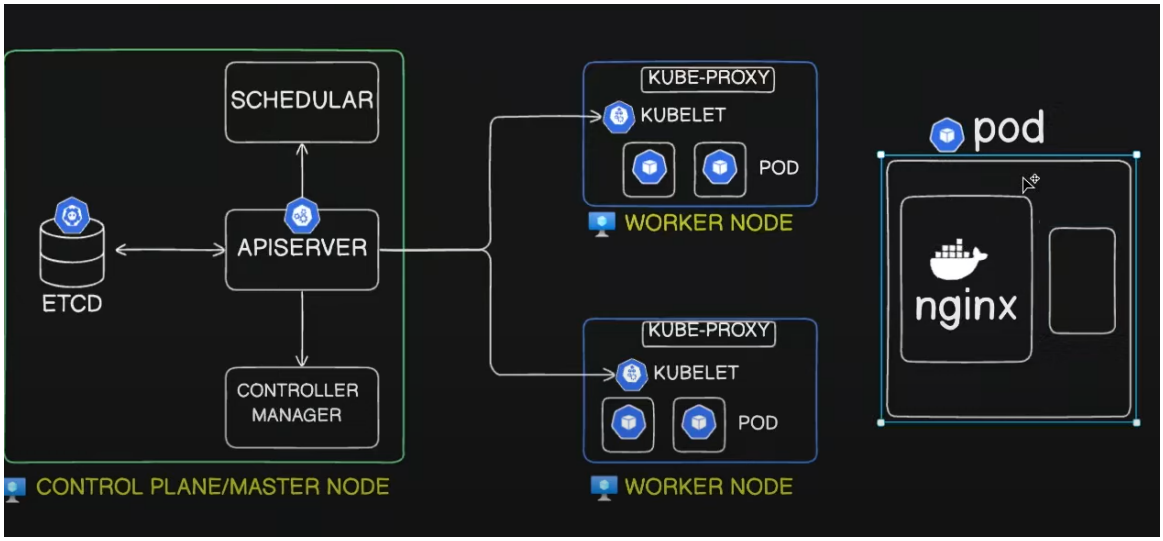
Cons:

- More complex to manage.
- Needs tools like Kubernetes to handle deployment, scaling, and communication.

🧠 Tip:

Kubernetes works best with microservices, because it helps manage multiple services (pods/containers) easily.

🧠 Kubernetes Architecture



🔷 1. Control Plane / Master Node

This is the **brain** of the Kubernetes system. It manages the entire cluster.

Components:

- **API Server**:
  Acts like a **gatekeeper**. All commands from users (like kubectl apply) go through it.
- **Scheduler**:
  Decides **which Worker Node** will run the new Pod.
  📦 Example: "Put this Nginx container on Node 1."
- **Controller Manager**:
  Makes sure the desired state is met (e.g., always keep 3 replicas running).
- **ETCD**:
  A **key-value store** that holds the entire cluster configuration and state.
  🗃️ Think of it like a **Kubernetes database**.

🖥️ 2. Worker Nodes

These nodes **run the actual applications (Pods)**.

Components:

- **Kubelet**:
  Talks to the API server and runs the Pod containers on the node.
- **Kube Proxy**:
  Handles networking and **forwards traffic** to the correct Pod.
- **Pods**:
  Smallest deployable unit in Kubernetes.
  Each Pod can run one or more **containers**.

🍼 **Example:**
A Pod running an **Nginx** container = your web server running inside Kubernetes.

🔁 Workflow (Example)

Here's what happens when you deploy an Nginx app:

1. You apply a YAML file:
   kubectl apply -f nginx-deployment.yaml
2. **API Server** receives the request and stores it in **ETCD**.
3. **Scheduler** picks a **Worker Node** to run the Pod.

4. **Controller Manager** makes sure the correct number of Pods are running.
5. On the chosen Worker Node:
    - **Kubelet** pulls the Nginx image.
    - **Pod** is created and starts the container.
    - **Kube Proxy** helps the Pod communicate with the internet or other Pods.

📝 **Final Notes (for quick memory):**

- Control Plane = Brain 🧠
- Worker Nodes = Muscles 💪
- Pod = Box 📦 containing your app
- ETCD = Memory 📜
- Kubelet = Pod Manager
- Kube Proxy = Traffic Manager

📦 What is a Namespace in Kubernetes?

A **Namespace** in Kubernetes is like a **virtual cluster** within a physical cluster. It helps **organize and separate** your Kubernetes resources.

🔹 Why Use Namespaces?

Imagine you're working on multiple projects or environments (like dev, test, prod). You don't want them to interfere with each other. That's where namespaces help:

- Keep resources **isolated**.
- Avoid name **conflicts** (you can have the same Pod name in different namespaces).
- Apply **resource limits** or **access control** per namespace.

🧠 **Think of it like:**

Real World Example	Kubernetes Analogy
Different folders on PC	Different namespaces in cluster

🔧 Default Namespaces in Kubernetes

Namespace	Purpose
default	Where resources go if you don't specify a namespace.
kube-system	For Kubernetes system components (like DNS, API server).
kube-public	Readable by all users (e.g., for bootstrapping).
kube-node-lease	Used internally for node heartbeat/status.

🛠 Example: Creating a Namespace

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

Then you can deploy a Pod into that namespace:

kubectl create namespace dev-environment

kubectl apply -f nginx-pod.yaml -n dev-environment

🔍 Commands

List all namespaces:

```
kubectl get namespaces
```

Create a namespace:

```
kubectl create namespace <name>
```

Deploy something in a specific namespace:

```
kubectl apply -f myfile.yaml -n <namespace-name>
```

Switch between namespaces (in kubectl context):

```
kubectl config set-context --current --namespace=<name>
```

✅ Summary:

- Namespaces = **Logical separation** of resources in Kubernetes.
- Great for **multi-team**, **multi-project**, or **multi-env** clusters.

- Helps with **organization**, **access control**, and **resource limits**.

Would you like a simple diagram for this too?

Kunurnetes commandlist:-
For logs and describe
Kubectl describe pod/<podename> -n <namesapce>
For login in pods or containers
Kuebctl exec -it <podname> -n <namespace> -- bash

🧱 What is a Pod in Kubernetes?

A **Pod** is the **smallest and simplest unit** in the Kubernetes object model.
It represents **a single instance of a running process** in your cluster.

🔹 Key Points:

- A Pod wraps **one or more containers** (usually one).
- Containers in a Pod:
  - **Share the same network IP**.
  - **Can talk to each other via localhost**.
  - **Share storage (volumes)** if defined.

📦 Simple Analogy:

A **Pod** is like a **box** that holds one or more **containers**, and they all live in the same room (network & storage).

🛠 Example YAML of a Pod running Nginx:

```
apiVersion: v1
kind: Pod
metadata:
  name: nginx-pod
spec:
  containers:
  - name: nginx-container
    image: nginx
    ports:
```

- containerPort: 80

👥 Single vs Multi-container Pods:

- **Single-container Pod** (common case)
  One container runs your app — simple and clean.
- **Multi-container Pod** (advanced case)
  Used when two or more containers need to work closely together, e.g.:
  - One container logs the data
  - Another container serves it

They can **share data via volume** and **communicate over localhost**.

💡 How Pods Fit into the Cluster (using your diagram):

- **Pods run on Worker Nodes**.
- Each Pod runs a container (like nginx in your image).
- The **Kubelet** on each node manages the Pods.
- **Scheduler** assigns Pods to Worker Nodes.
- **API Server** communicates with the cluster to create/manage Pods.

🔍 Useful Commands:

Create a pod:
`kubectl apply -f nginx-pod.yaml`
List pods:
`kubectl get pods`
Describe a pod:
`kubectl describe pod <pod-name>`
Delete a pod:
`kubectl delete pod <pod-name>`

✅ Summary:

Concept	Description
Pod	Smallest unit in Kubernetes, wraps container(s)
Purpose	Run applications, provide network & storage
Lives where	On **Worker Nodes**
Manages	One or more containers

Would you also like the explanation of **ReplicaSet** or **Deployment** next?

Labels and Selectors in Kubernetes:-

🔹 **What are Labels?**

- **Labels** are **key-value pairs** attached to Kubernetes objects (like Pods, Services, etc.).
- They help in **organizing**, **grouping**, and **selecting** objects.
- Labels **do not affect behavior** — they are just for identification.

🧠 Think of labels like **tags** on social media posts.

✅ **Example of a Label:**

```
metadata:
name: myapp-pod
  labels:
    app: myapp
    environment: dev
```

🔍 **What are Selectors?**

- **Selectors** are used to **filter** or **match** Kubernetes objects based on their labels.
- Useful when:
    - A **Service** needs to find Pods.
    - A **Deployment** targets specific Pods.
    - You want to run kubectl commands for specific groups.

🎯 Two Types of Selectors:

1. Equality-Based Selector:

```
selector:
  matchLabels:
    app: myapp
```

Matches Pods with label app=myapp.

2. Set-Based Selector (more flexible):

```
selector:
  matchExpressions:
  - key: environment
    operator: In
    values:
    - dev
    - test
```

Matches Pods where environment is either dev or test.

🛠 Example: Service Using a Selector

```
apiVersion: v1
kind: Service
metadata:
  name: my-service
spec:
  selector:
    app: myapp
  ports:
    - port: 80
      targetPort: 8080
```

This service will automatically send traffic to all Pods with app: myapp.

🔍 Commands

- **View labels on a Pod:**

  kubectl get pods --show-labels

- **Add label to a Pod:**

  kubectl label pod myapp-pod version=v1

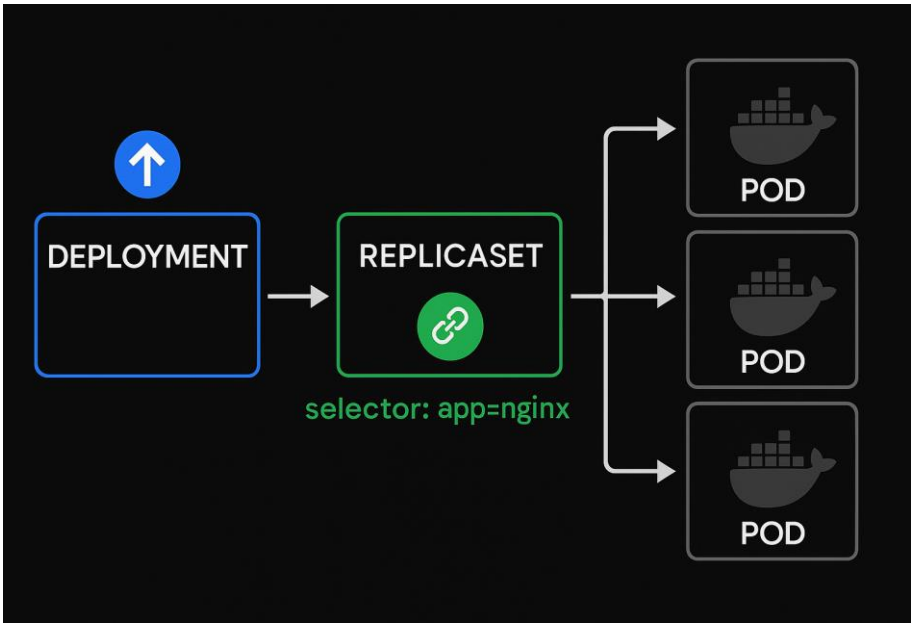
- **Select Pods with a label:**

  kubectl get pods -l app=myapp

✅ Summary Table

Term	Description
Label	Key-value tag for identification (e.g. app=myapp)
Selector	Way to filter or match objects using labels
Used In	Pods, Services, ReplicaSets, Deployments, etc.

Kubernetes Deployment



🔷 What is a Deployment?

A **Deployment** is a **Kubernetes object** used to manage and update your application.
It automatically handles:

- Creating Pods
- Updating Pods without downtime
- Restarting crashed Pods
- Scaling Pods (up or down)

🧠 **Think of a Deployment as a manager** for your Pods.

📦 Why Use a Deployment?

Without a Deployment, you'd have to manage Pods manually (hard and error-prone).

With a Deployment, Kubernetes will:

✅ Ensure the correct number of Pods

✅ Replace unhealthy Pods

✅ Perform rolling updates

🛠 Example: Nginx Deployment (YAML)

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

🔍 **What Happens Here?**

- **replicas: 3** → Kubernetes creates 3 Pods.
- **selector** → Selects Pods with app=nginx label.
- **template** → Defines the Pod (container image, port, etc.).
- **Deployment** watches and ensures all 3 Pods are running.

🔁 **Deployment Workflow (Step-by-Step)**

You run:

kubectl apply -f nginx-deployment.yaml

API Server receives the request.

Deployment controller checks:

“Are 3 Pods running?”

“Do they match the template?”

If not, it **creates, updates, or deletes Pods** until desired state is achieved.

If one Pod crashes → Deployment replaces it.

🔧 Useful Commands

Command	Description
kubectl get deployments	Show deployments
kubectl describe deployment <name>	Detailed info
kubectl scale deployment nginx-deployment --replicas=5	Scale Pods
kubectl rollout status deployment nginx-deployment	Check rollout status
kubectl delete deployment <name>	Delete deployment

🧠 Summary Table

Term	Meaning
Deployment	Manages replica Pods
Replica	Number of Pod copies
Template	Pod structure
Selector	Chooses which Pods to manage
Use Case	Updates, scaling, recovery

📦 Rolling Update in Kubernetes:-

A **Rolling Update** is the default strategy used by Kubernetes Deployments to update Pods **without downtime**.

🔁 What is Rolling Update?

It is a method to update your app **gradually**, by replacing old Pods with new ones in a **step-by-step** way.

🧠 How It Works:

1. You create or update a **Deployment** (e.g., change image version).
2. Kubernetes creates **new Pods** (with new version).
3. It slowly deletes the **old Pods**, one at a time.
4. This continues until all Pods are updated.

⚙️ **Example:**

Suppose you have 3 Pods running Nginx v1:

```
spec:
  replicas: 3
  template:
    spec:
      containers:
      - name: nginx
        image: nginx:1.14
```

Now you update it to nginx:1.16. Kubernetes will:

- Start 1 new pod with nginx:1.16
- Delete 1 old pod with nginx:1.14
- Repeat until all are updated

🕹️ **Rolling Update Settings (optional):**

You can control how fast updates happen:

```
strategy:
  type: RollingUpdate
  rollingUpdate:
    maxSurge: 1       # Extra pod allowed during update
    maxUnavailable: 1  # Max pods that can be down during update
```

✅ Benefits:

- No downtime
- Safe and automatic
- Easy to rollback if needed

🔁 ReplicaSet in Kubernetes

📘 **What is a ReplicaSet?**

A **ReplicaSet** ensures that a **specific number of identical Pods** are always running in your cluster.

🎯 Goal: **Keep N copies of a Pod running at all times.**

If a Pod crashes, the ReplicaSet will create a new one automatically.
If extra Pods are running, it deletes them.

🔁 **How It Works:**

- You define how many Pods (replicas) you want.
- You also define what kind of Pods (using **labels**).
- ReplicaSet ensures those Pods always run.

🧠 **Example:**

```
apiVersion: apps/v1
kind: ReplicaSet
metadata:
  name: my-replicaset
spec:
  replicas: 3
  selector:
    matchLabels:
      app: demo
  template:
    metadata:
      labels:
        app: demo
    spec:
      containers:
      - name: demo-container
        image: nginx
```

🔍 **What This Does:**

- Runs **3 Pods** with Nginx image.
- Selects Pods that have app=demo label.
- Keeps watching and maintaining the number of Pods.

🧩 **Connection to Deployments:**

- **Deployment** creates and manages **ReplicaSets**
- **You normally don't create ReplicaSets manually**

- When you update a Deployment → Kubernetes creates a **new ReplicaSet** and removes the old one

---

🧠 **Summary Table**

Term	Description
ReplicaSet	Maintains desired number of Pods
Replica	Each running copy of the Pod
Selector	Matches which Pods to manage
Template	Defines Pod structure
Used by	Mostly managed by Deployments

⚙️ DaemonSet in Kubernetes

📘 What is a DaemonSet?

A **DaemonSet** ensures that a **specific Pod runs on every (or selected) node** in the Kubernetes cluster.

🧠 Think of it as:
"Run **one Pod per Node** – automatically."

📦 Common Use Cases:

DaemonSets are used for **background system tasks** that must run on **every node**, such as:

- **Log collectors** (e.g., Fluentd, Logstash)
- **Monitoring agents** (e.g., Prometheus Node Exporter)
- **Security agents** (e.g., antivirus, intrusion detection)
- **Storage daemons** (e.g., Glusterd)

🛠️ Example YAML:

```
apiVersion: apps/v1
kind: DaemonSet
metadata:
  name: log-agent
spec:
  selector:
    matchLabels:
      name: log-agent
```

```
template:
  metadata:
    labels:
      name: log-agent
  spec:
    containers:
    - name: log-agent-container
      image: fluentd
```

🔁 **What Happens?**

- As soon as a new node joins the cluster, the DaemonSet **automatically creates a Pod** on that node.
- If a node leaves, the Pod is removed.

🔍 **Difference from Deployment:**

Feature	Deployment	DaemonSet
Purpose	Run app replicas	Run 1 pod per node
Pod scheduling	Based on replica count	One per node (auto-scheduled)
Use Cases	Web apps, APIs	Logs, monitoring, storage agents

✅ **Summary Table**

Concept	Description
DaemonSet	Ensures a Pod runs on **every Node**
Use Case	System agents, monitoring, logging
Auto-pods	Yes, new node = new Pod
Managed by	Kubernetes automatically

🧱 Job in Kubernetes

📘 **What is a Job?**

A **Job** is used in Kubernetes to **run a task once** and **ensure it completes successfully**.

🧠 Think of it as:
“Run this script **one time** (or few times) until it's done.”

✅ **Use Cases for Jobs:**

- Data backups
- Sending emails
- Image or video processing
- Database migrations
- Cron-style tasks (when not using CronJobs)

🛠️ **Example YAML:**

```
apiVersion: batch/v1
kind: Job
metadata:
  name: example-job
spec:
  template:
    spec:
      containers:
      - name: hello
        image: busybox
        command: ["echo", "Hello from Job!"]
      restartPolicy: Never
```

⚙️ **How it Works:**

1. Kubernetes creates a Pod using the Job spec.
2. The Pod **runs the task** (like printing, processing, migrating).
3. When the Pod finishes successfully, the Job is marked **complete**.
4. If the Pod fails, Kubernetes **retries it** (based on settings).

🔁 **Controlling Repeats:**

```
spec:
  completions: 3      # Run job 3 times successfully
  parallelism: 2      # Max 2 Pods can run at the same time
```

🧠 **Key Points:**

Concept	Description
One-time task	Runs a task till success
Retry on fail	Kubernetes can retry failed Pods
restartPolicy	Usually set to Never or OnFailure
Short-living	Meant for short, temporary background tasks

📍 **Commands**

kubectl create -f job.yaml
kubectl get jobs
kubectl describe job <job-name>
kubectl delete job <job-name>

🕒 CronJob in Kubernetes

📘 **What is a CronJob?**

A **CronJob** runs **Jobs on a schedule**, just like Linux cron.

🧠 Think of it as:
“Run this task automatically every day, hour, or week.”

✅ **Use Cases for CronJobs:**

- Daily backups
- Sending weekly emails
- Monthly report generation
- Scheduled cleanup scripts
- Periodic database updates

🛠️ **Example YAML:**

```
apiVersion: batch/v1
kind: CronJob
metadata:
  name: my-cronjob
spec:
  schedule: "*/5 * * * *"  # Run every 5 minutes
  jobTemplate:
    spec:
```

```
  template:
    spec:
      containers:
      - name: hello
        image: busybox
        command: ["echo", "Hello from CronJob!"]
      restartPolicy: OnFailure
```

🕐 **Cron Format (in "schedule"):**

Field	Meaning	Example
Minute	0–59	5
Hour	0–23	3
Day	1–31	15
Month	1–12	6
Weekday	0–6 (Sun–Sat)	1

Example:

- "0 0 * * *" → every day at midnight
- "*/10 * * * *" → every 10 minutes

🧠 **Key Points:**

Concept	Description
Scheduled jobs	Automatically runs Jobs at defined times
Based on cron	Uses standard Linux cron syntax
Runs Jobs	Internally creates Jobs as per the schedule
restartPolicy	Usually set to OnFailure

📍 **Useful Commands:**

```
kubectl create -f cronjob.yaml
kubectl get cronjobs
kubectl describe cronjob <name>
kubectl delete cronjob <name>
```

📦 Kubernetes PV & PVC

📘 **What is PV (Persistent Volume)?**

Persistent Volume (PV) is a piece of storage in the cluster **provisioned by an admin** or dynamically.

🧠 Think of PV as:

"📁 A real storage unit (like a hard drive) that lives inside your cluster."

📘 What is PVC (Persistent Volume Claim)?

Persistent Volume Claim (PVC) is a **request** for storage by a user or application.

🧠 Think of PVC as:

"📄 A request form asking: 'Give me 1GB storage with Read/Write access'."

🧠 Real-World Example:

Imagine you're in a library:

- 📦 PV = Bookshelf
- 📄 PVC = Request slip asking for a book
- 📚 Pod = Reader who uses the book

📊 Difference between PV and PVC:

Feature	PV (Persistent Volume)	PVC (Persistent Volume Claim)
Who creates it?	Cluster Admin / Dynamically	Developer / User
What it represents?	Actual storage (like disk)	Request for storage
Lifespan	Independent of Pods	Bound to Pod usage

🛠️ Example YAML for PV:

```
apiVersion: v1
kind: PersistentVolume
metadata:
  name: my-pv
spec:
  capacity:
    storage: 1Gi
  accessModes:
    - ReadWriteOnce
  hostPath:
    path: "/data/myapp"
```

🛠️ Example YAML for PVC:

```
apiVersion: v1
```

```
kind: PersistentVolumeClaim
metadata:
  name: my-pvc
spec:
  accessModes:
    - ReadWriteOnce
  resources:
    requests:
      storage: 1Gi
```

📦 **Pod Using PVC:**

```
volumes:
  - name: storage
    persistentVolumeClaim:
      claimName: my-pvc
```

💡 **Key Terms:**

Term	Meaning
ReadWriteOnce	One node can read/write
ReadOnlyMany	Many nodes can only read
ReadWriteMany	Many nodes can read/write
hostPath	Uses path from node (for demo/testing)
storageClass	Automates dynamic volume provisioning

🌐 Kubernetes Service Types: NodePort vs ClusterIP

In Kubernetes, **Services** expose your Pods so they can be accessed internally or externally.
Let's explore **ClusterIP** and **NodePort** — the two most used service types.

🚩 **1. ClusterIP (Default)**

📘 **What is ClusterIP?**

- It **exposes the service **inside the cluster only.
- You **cannot access it from outside** the Kubernetes cluster.
- Other **Pods** can use the service to communicate.

🔒 Internal communication only.

🔧 **Use Case:**

- Internal microservice communication

- Backend services (e.g., database, internal APIs)

🛠️ **Example YAML:**

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

🌎 **2. NodePort**

📘 **What is NodePort?**

- It **exposes the service on a static port** (range: 30000–32767) on each Node's IP.
- You can access the service via:
- http://<NodeIP>:<NodePort>

🌐 Used when you want to access your app **outside the cluster** without a LoadBalancer.

🔧 **Use Case:**

- Quick testing on local/minikube
- Accessing service from outside (manually)

🛠️ **Example YAML:**

```
apiVersion: v1
kind: Service
metadata:
  name: frontend-service
spec:
  type: NodePort
  selector:
    app: frontend
  ports:
    - port: 80
      targetPort: 8080
      nodePort: 30036
```

📊 **Comparison Table**

Feature	ClusterIP	NodePort
Default?	✅ Yes	❌ No
Access Scope	Internal only	Internal + External
Access Format	ClusterIP:Port	NodeIP:NodePort
Use Case	Pod-to-Pod communication	Access from browser / outside
External Access	❌ Not possible	✅ Yes
Port Range	Any	30000–32767

🔁 **Optional Add-on: LoadBalancer (Extra)**

- Automatically provisions an **external IP**
- Works in **cloud providers**
- Best for production/public exposure

🧠 **Pro Tips:**

- **ClusterIP** is used most commonly in **internal microservice architecture**.
- **NodePort** is great for **development, testing, or small setups**.
- **LoadBalancer** is best for **production in cloud**.

Ingress and Annotations

🌐 **Ingress in Kubernetes**

📘 **What is Ingress?**

Ingress is a Kubernetes **API object** that **manages external access** to services in a cluster, typically **HTTP/HTTPS** traffic.

Rather than exposing services via NodePort or LoadBalancer, **Ingress provides smart routing** to multiple services under **one IP/domain**.

🧭 **Key Features of Ingress:**

- Routes HTTP(S) traffic to **internal services**.

- Supports **path-based routing** (e.g., /app1, /app2).
- Supports **host-based routing** (e.g., app.example.com).
- Can use **TLS/SSL** for secure connections.
- Requires an **Ingress Controller** (like NGINX, Traefik).

⚙️ **Ingress Example YAML:**

```
apiVersion: networking.k8s.io/v1
kind: Ingress
metadata:
  name: my-ingress
  annotations:
    nginx.ingress.kubernetes.io/rewrite-target: /
spec:
  rules:
    - host: myapp.example.com
      http:
        paths:
          - path: /app1
            pathType: Prefix
            backend:
              service:
                name: app1-service
                port:
                  number: 80
          - path: /app2
            pathType: Prefix
            backend:
              service:
                name: app2-service
                port:
                  number: 80
```

🔁 **Workflow:**

1. User sends request to myapp.example.com/app1
2. Ingress forwards to app1-service
3. Pod handles the request and responds

📦 **Ingress Controller:**

Ingress won't work without an **Ingress Controller**.

Examples:

- NGINX Ingress Controller

- Traefik
- AWS ALB Ingress Controller

You must deploy it in the cluster separately.

🏷️ Annotations in Kubernetes

📘 What are Annotations?

Annotations are **key-value pairs** used to attach **non-identifying metadata** to Kubernetes objects.

Unlike labels (used for selection), **annotations are for extra info/config** — like hints for tools, logs, or controllers.

🧠 Use Cases for Annotations:

- Configure Ingress behavior (e.g., rewrite, rate limiting)
- Link external tools (e.g., Prometheus scraping)
- Store custom info (e.g., build timestamp, version)

📜 Example:

```
metadata:
 name: my-pod
 annotations:
  created-by: "cloud-team"
  nginx.ingress.kubernetes.io/rewrite-target: /
```

📌 Common Ingress Annotations (NGINX)

Annotation Key	Description
nginx.ingress.kubernetes.io/rewrite-target	Rewrites URL paths
nginx.ingress.kubernetes.io/ssl-redirect	Forces HTTPS
nginx.ingress.kubernetes.io/whitelist-source-range	Restrict IP ranges
nginx.ingress.kubernetes.io/configuration-snippet	Inject custom config into NGINX
nginx.ingress.kubernetes.io/enable-cors	Enables Cross-Origin Resource Sharing

---

📊 Difference: Labels vs Annotations

Feature	Labels	Annotations
Purpose	Used for **selection/filtering**	Used for **metadata/config**
Max size	Small (KBs)	Can be larger
Examples	app=frontend, env=prod	author=teamA, docs-url=https://

🗃️ StatefulSet in Kubernetes

📘 What is a StatefulSet?

StatefulSet is a Kubernetes **controller** used to manage **stateful applications**, i.e., apps that **need persistent storage and stable network identity**.

While **Deployment** is used for stateless apps (e.g., Nginx), **StatefulSet** is used when:

- Each pod needs a **unique identity** (like pod-0, pod-1, etc.)
- Pods need to be started/stopped **in order**
- Data must **persist across restarts**

🧠 Use Cases:

- Databases (MySQL, MongoDB, Cassandra)
- Distributed systems (Kafka, Zookeeper)
- Any app requiring **stable storage + stable network**

📄 Example 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: mongo
        image: mongo:4.4
        volumeMounts:
          - name: mongo-data
            mountPath: /data/db
volumeClaimTemplates:
  - metadata:
      name: mongo-data
    spec:
      accessModes: ["ReadWriteOnce"]
      resources:
        requests:
          storage: 5Gi
```

🔁 **How It Works:**

- **Pod names** will be: my-db-0, my-db-1, my-db-2
- Each pod gets its **own PersistentVolumeClaim** (PVC)
- If a pod is deleted, it's **recreated with the same name + same volume**
- Pods are created **in order** (0, then 1, then 2)
- Pods are **terminated in reverse order**

🧩 **Key Features of StatefulSet:**

Feature	StatefulSet	Deployment
Stable Pod Names	✅ pod-0, pod-1, etc.	❌ Random names
Ordered Startup/Shutdown	✅ Yes	❌ No order
Persistent Volumes (PVCs)	✅ Unique per pod	❌ Usually shared or ephemeral
Use Case	Databases, Stateful apps	Web servers, APIs

🔗 **Headless Service:**

You must create a **headless service** (ClusterIP: None) for the StatefulSet to allow stable network identities like my-db-0.my-db-headless.default.svc.cluster.local.

```
apiVersion: v1
kind: Service
metadata:
  name: my-db-headless
spec:
  clusterIP: None
  selector:
    app: my-db
```

```
ports:
  - port: 27017
```

🔧 ConfigMap & 🔐 Secret in Kubernetes

Kubernetes provides **ConfigMap** and **Secret** to manage configuration data **separately from application code**.

🔧 ConfigMap (Non-sensitive data)

Stores configuration data like environment variables, command-line args, config files, etc.

📌 **Use Case**: You want to pass non-sensitive configuration to your pods (like URLs, mode, etc.).

✅ **Example:**

```
apiVersion: v1
kind: ConfigMap
metadata:
  name: my-config
data:
  APP_MODE: "production"
  APP_PORT: "8080"
```

📦 **Using in a Pod:**

```
spec:
  containers:
    - name: my-app
      image: myapp:latest
      envFrom:
        - configMapRef:
            name: my-config
```

🔐 Secret (Sensitive data)

Stores sensitive info like passwords, tokens, SSH keys, etc.

📌 **Use Case**: You want to pass secret info **securely** to your pods.

✅ **Example:**

```
apiVersion: v1
kind: Secret
metadata:
  name: my-secret
type: Opaque
data:
  DB_PASSWORD: bXlwYXNzMTIz  # Base64 encoded "mypass123"
```

You can encode values using:
echo -n 'mypass123' | base64

📦 **Using in a Pod:**

```
spec:
 containers:
  - name: my-app
    image: myapp:latest
    env:
     - name: DB_PASSWORD
       valueFrom:
         secretKeyRef:
           name: my-secret
           key: DB_PASSWORD
```

🧠 **Key Differences**

Feature	ConfigMap	Secret
Stores	Non-sensitive data	Sensitive data
Format	Plaintext	Base64 encoded
Access Control	No encryption	Can be encrypted
Mounted as	Env var, file	Env var, file

💡 **Tip:**

- Use **ConfigMap** for things like APP_ENV=dev, LOG_LEVEL=debug
- Use **Secret** for things like API_KEY, DB_PASSWORD, etc.

⚙️ Resources and Limits in Kubernetes

Kubernetes lets you control **how much CPU and memory (RAM)** a container can **use** using requests and limits.

📌 **Why Use Resources and Limits?**

- ✅ Avoid overloading nodes
- ✅ Ensure fair sharing between pods
- ✅ Prevent one pod from consuming all the resources
- ✅ Schedule pods based on available resources

🧩 **Key Terms:**

Term	Meaning
requests	Minimum resources the pod **needs** to run (used for scheduling)
limits	Maximum resources the pod **can use** (enforced at runtime)

✅ **Example: Set CPU & Memory**

```
apiVersion: v1
kind: Pod
metadata:
  name: my-pod
spec:
  containers:
    - name: my-container
      image: myimage:latest
      resources:
        requests:
          memory: "64Mi"
          cpu: "250m"
        limits:
          memory: "128Mi"
          cpu: "500m"
```

📝 **Explanation:**

Requests:

- 64Mi memory
- 250m (0.25 CPU core)

Limits:

- 128Mi memory
- 500m (0.5 CPU core)

💡 **Good Practices:**

1. Always define both requests and limits
2. Helps avoid Out Of Memory (OOM) errors
3. Improves pod scheduling accuracy
4. Prevents resource starvation on nodes

🧠 **Real-Life Example:**

- You have a Node with 1 CPU and 2GB RAM.
- You run 4 pods, each requesting 250m CPU → they'll all fit.
- If one pod tries to use more than 500m CPU (limit), it will be throttled.

Here's a simple explanation of **probes** in Kubernetes — perfect for adding to your notes with examples.

🩺 Probes in Kubernetes (Liveness & Readiness)

Probes are health checks Kubernetes uses to monitor the status of containers.
There are 3 types:

Type	Purpose
Liveness Probe	Checks if the app is **still running**
Readiness Probe	Checks if the app is **ready to receive traffic**
Startup Probe	Checks if the app has **started properly**

✅ Liveness Probe (Is it alive?)

If it fails, Kubernetes **restarts the container**.

```
livenessProbe:
  httpGet:
    path: /health
    port: 8080
  initialDelaySeconds: 10
  periodSeconds: 5
```

✅ Readiness Probe (Is it ready?)

If it fails, Kubernetes **removes the pod from the service** until it's ready again.

```
readinessProbe:
  httpGet:
    path: /ready
    port: 8080
  initialDelaySeconds: 5
  periodSeconds: 3
```

✅ Startup Probe (Has it started?)

Used when apps take a long time to start.

Disables liveness & readiness probes until this passes.

```
startupProbe:
  httpGet:
    path: /start
    port: 8080
  failureThreshold: 30
  periodSeconds: 10
```

🧠 Real Example in a Pod

```
apiVersion: v1
kind: Pod
metadata:
  name: demo
spec:
  containers:
  - name: app
    image: myapp:latest
    ports:
    - containerPort: 8080
    livenessProbe:
      httpGet:
        path: /health
        port: 8080
      initialDelaySeconds: 10
    readinessProbe:
      httpGet:
        path: /ready
        port: 8080
      initialDelaySeconds: 5
```

💡 Summary

Probe Type	When It's Used	Action Taken
Liveness Probe	App is unresponsive	Restart the container
Readiness Probe	App is not ready to serve traffic	Remove from Service
Startup Probe	App needs more time to start	Wait before applying other probes

---

😷 Taints and Tolerations in Kubernetes

Taints and Tolerations work together to control **which pods can run on which nodes**.

- **Taint**: Added to a node to say **"Don't schedule pods here unless they can tolerate it."**
- **Toleration**: Added to a pod to say **"I'm okay with this taint — I can run here."**

📌 Why Use Taints & Tolerations?

- Keep **critical workloads** on specific nodes
- Prevent certain pods from running on **GPU/high-memory nodes**
- Control **pod placement** more precisely

🧩 Example: Taint a Node

kubectl taint nodes node1 key=value:NoSchedule

This means:

🚫 Don't allow pods on node1 unless they have a toleration for key=value.

✅ Add Toleration in Pod YAML

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

This pod **can now run** on a node with the matching taint.

💡 Effects of Taints:

Effect	Meaning
NoSchedule	Never schedule unless toleration is added
PreferNoSchedule	Try to avoid, but not strict
NoExecute	If already running, it will be evicted unless tolerated

---

🧠 Real-Life Analogy:

- A taint is like a **"Staff Only"** sign on a room.
- A toleration is like a **staff badge** that lets you enter anyway.

🛠️ Full Pod Example with Toleration

```
apiVersion: v1
kind: Pod
metadata:
  name: toleration-demo
spec:
  containers:
  - name: app
    image: nginx
  tolerations:
  - key: "env"
    operator: "Equal"
    value: "prod"
    effect: "NoSchedule"
```

This pod can run on nodes tainted with:

kubectl taint nodes <node-name> env=prod:NoSchedule

🔁 HPA (Horizontal Pod Autoscaler)

🧠 What It Does:

HPA **increases or decreases the number of pods** in a deployment based on CPU, memory usage, or custom metrics.

📦 Use Case:

When app traffic increases, HPA automatically adds more pods. When traffic drops, it removes them to save resources.

✅ Example:

kubectl autoscale deployment my-app --cpu-percent=50 --min=2 --max=10

This means:

- If CPU > 50%, Kubernetes will add pods (up to 10).
- If CPU < 50%, it can scale down (minimum 2 pods).

📌 YAML Example:

```
apiVersion: autoscaling/v1
kind: HorizontalPodAutoscaler
metadata:
  name: my-app-hpa
spec:
  scaleTargetRef:
    apiVersion: apps/v1
    kind: Deployment
    name: my-app
  minReplicas: 2
  maxReplicas: 10
  targetCPUUtilizationPercentage: 50
```

📏 VPA (Vertical Pod Autoscaler)

🧠 What It Does:

VPA **adjusts CPU and memory requests/limits** of a pod **automatically** based on usage.

📦 Use Case:

If a pod needs more memory or CPU over time, VPA recommends or sets new resource values — good for backend services that grow with load.

✅ **Modes of VPA:**

Mode	Behavior
Off	Only gives recommendations
Auto	Automatically updates pod resource requests
Initial	Sets values only when pod is first created

📌 **YAML Example:**

```
apiVersion: autoscaling.k8s.io/v1
kind: VerticalPodAutoscaler
metadata:
  name: my-app-vpa
spec:
  targetRef:
    apiVersion: "apps/v1"
    kind: Deployment
    name: my-app
  updatePolicy:
    updateMode: "Auto"
```

🚀 **Key Differences:**

Feature	HPA	VPA
Scales by	Pod count	Pod CPU/Memory resources
Use Case	Web apps with varying traffic	Services with varying resource needs
Restarts Pods	No (just adds/removes)	Yes (to apply new resources)

🧠 **Real-Life Analogy:**

- **HPA**: Like hiring more staff when work increases.
- **VPA**: Like giving your current staff better tools or bigger desks.

📍 Node Affinity in Kubernetes

🧠 What Is It?

Node Affinity is a way to tell Kubernetes **where to place a pod**, based on the **labels of the nodes**.

It ensures that **pods are scheduled on specific types of nodes** (e.g., only nodes in a certain region, or with a GPU, or high memory).

✅ Why Use It?

You may want to run:

- Database pods on high-memory nodes 🧠
- AI/ML pods on GPU nodes ⚡
- Backend apps in specific zones or environments 🌍

🧩 How It Works:

Node affinity uses **node labels** and applies rules using:

1. requiredDuringSchedulingIgnoredDuringExecution

- Must match, otherwise pod won't be scheduled.
- **Hard requirement**

2. preferredDuringSchedulingIgnoredDuringExecution

- Try to match, but not mandatory.
- **Soft preference**

💉 Example

1. Label a node:

kubectl label nodes node-1 disktype=ssd

2. Pod YAML with Node Affinity:

```
apiVersion: v1
kind: Pod
metadata:
  name: ssd-pod
spec:
```

```
 affinity:
   nodeAffinity:
     requiredDuringSchedulingIgnoredDuringExecution:
       nodeSelectorTerms:
       - matchExpressions:
         - key: disktype
           operator: In
           values:
           - ssd
 containers:
 - name: app
   image: nginx
```

This pod will **only run on nodes with disktype=ssd**.

📌 **Key Terms**

Term	Meaning
key	The node label key (e.g. disktype)
operator	In, NotIn, Exists, etc.
values	List of allowed values (e.g. ssd)

🧐 **Difference from nodeSelector**

Feature	nodeSelector	nodeAffinity
Flexibility	Basic key-value only	Advanced expressions
Type	Hard rule	Hard or soft rules
Use case	Simple needs	Complex scheduling logic

🧠 **Real-Life Analogy:**

Think of node affinity like saying:

"I want to live in a house **with air conditioning (label)**, and preferably **in the city center (preferred rule)**."

🔐 RBAC (Role-Based Access Control)

RBAC is used in Kubernetes to **control who can do what** inside the cluster.

It defines **permissions** for users, service accounts, or groups, such as:

- Who can create pods? ✅
- Who can delete services? ❌

- Who can access secrets? 🔒

🧩 Main Components of RBAC

Component	What it does
Role	Defines what actions are allowed (namespace-wide)
ClusterRole	Like Role, but cluster-wide
RoleBinding	Connects Role to a user or service account
ClusterRoleBinding	Connects ClusterRole to a user/service account
ServiceAccount	Special account used by **pods**
User	Represents a **real person** using the cluster

🧠 Example Use Case:

"Allow a pod to read Secrets only in the dev namespace."

1️⃣ Create a Role

```
apiVersion: rbac.authorization.k8s.io/v1
kind: Role
metadata:
  namespace: dev
  name: read-secrets
rules:
- apiGroups: [""]
  resources: ["secrets"]
  verbs: ["get", "list"]
```

2️⃣ Create a RoleBinding

```
apiVersion: rbac.authorization.k8s.io/v1
kind: RoleBinding
metadata:
  name: bind-read-secrets
  namespace: dev
subjects:
- kind: ServiceAccount
  name: dev-pod-sa
  namespace: dev
roleRef:
  kind: Role
  name: read-secrets
  apiGroup: rbac.authorization.k8s.io
```

3️⃣ Create a ServiceAccount

```
apiVersion: v1
kind: ServiceAccount
metadata:
  name: dev-pod-sa
  namespace: dev
```

Use this service account in your pod spec:

```
spec:
  serviceAccountName: dev-pod-sa
```

👤 User vs ServiceAccount

Feature	User	ServiceAccount
Represents	A human (via kubectl)	A pod or workload
Created	Outside cluster (e.g., via certs)	Inside cluster (YAML)
Authentication	Certificates or IAM	Automatically attached to pod

---

🔄 ClusterRole & ClusterRoleBinding

Use when:

- You need access across **all namespaces**.
- Example: Grant user power to list all pods across cluster.

🚨 RBAC Verbs

Verb	Meaning
get	View a single resource
list	View multiple resources
create	Make new resources
update	Modify existing resources
delete	Remove resources

🔐 Real-Life Analogy

Think of Kubernetes RBAC like a company office:

- Role: Job description (e.g., can access files)
- RoleBinding: Assign job to a person
- User: Employee

- ServiceAccount: Office robot
- ClusterRole: Company-wide permission (e.g., Admin)
- ClusterRoleBinding: Assign company-wide role

Here's a **simple explanation of Helm with its life cycle** — perfect for your Kubernetes notes:

⛵ What is Helm?

Helm is the **package manager for Kubernetes** — like apt for Ubuntu or npm for Node.js.

It helps you:

- Deploy complex apps easily
- Manage configuration
- Upgrade or uninstall apps cleanly

Helm packages are called **Charts**.

📦 Helm Chart = Package

A **Helm Chart** contains:

- Kubernetes manifests (YAML files)
- Templates (with variables)
- A values.yaml file for configuration

🔁 Helm Lifecycle

1. Create a Helm Chart

helm create my-chart

Creates boilerplate folder with templates.

---

2. Customize the Chart

Edit:

- values.yaml → set default configs
- templates/ → add your deployment, service, etc.

---

3. Install the Chart

helm install my-release ./my-chart

- my-release is the name of your deployment
- Helm renders the templates and deploys the resources

---

4. Upgrade/Update the Release

helm upgrade my-release ./my-chart

Changes in values.yaml or templates are applied.

---

5. Rollback to Previous Version

helm rollback my-release 1

Restores the release to a previous version if something goes wrong.

---

6. List All Releases

helm list

---

7. Uninstall (Delete) the Release

helm uninstall my-release

Cleans up all Kubernetes resources created by the chart.

🌱 **Helm Chart Directory Structure**

```
my-chart/
├── Chart.yaml        # Chart metadata
├── values.yaml       # Default config values
├── templates/        # K8s YAML templates
|   ├── deployment.yaml
|   ├── service.yaml
|   └── ...
```

🧠 **Real-Life Analogy**

Think of Helm like **Docker Compose for Kubernetes**:

- Chart.yaml = App info

- values.yaml = Settings/config
- templates/ = YAML blueprints
- install/upgrade/uninstall = Run, change, stop app

✅ **Helm Benefits**

Feature	Description
Reusability	Use charts across environments
Config Management	Use values.yaml for custom configs
Version Control	Upgrade and rollback easily
Sharing	Use or publish charts in public repos

🧱 Init Container

🧠 **What Is It?**

An **Init Container** runs **before the main container** in a Pod starts.

Used for:

- Setting up things (like downloading code)
- Waiting for another service
- Running setup scripts

It **must complete successfully** before the main container runs.

✅ **Example Use Case:**

- Download code or files from a Git repo before app starts.
- Check if a DB is reachable before launching the app.

🧪 **Sample YAML:**

```
spec:
 initContainers:
 - name: init-myservice
   image: busybox
   command: ['sh', '-c', 'echo Waiting for DB; sleep 10']
 containers:
 - name: myapp-container
   image: nginx
```

📦 Sidecar Container

🧠 What Is It?

A **Sidecar Container** runs **alongside the main container** in the same Pod.

Used for:

- Logging
- Monitoring
- Proxying
- Data synchronization

It works **together with the main container** and can share volumes and network.

✅ Example Use Case:

- A log collector sidecar that sends app logs to an external system.
- A reverse proxy (like Envoy or Nginx) beside the app container.
- A file sync tool like rsync.

💉 Sample YAML:

```
spec:
 containers:
 - name: main-app
   image: my-app-image
 - name: sidecar-logger
   image: busybox
   command: ['sh', '-c', 'tail -f /var/log/app.log']
   volumeMounts:
   - name: log-volume
     mountPath: /var/log
 volumes:
 - name: log-volume
   emptyDir: {}
```

🔁 Comparison Table

Feature	Init Container	Sidecar Container
When It Runs	Before main container	Alongside main container
Runs In Parallel?	❌ No (runs before)	✅ Yes (runs at the same time)
Use Case	Setup/preparation	Support/helper tasks (logging, proxying)
Shared Volume/Net?	✅ Yes	✅ Yes

🧠 Real-Life Analogy

- **Init Container**: Like a housekeeper who **sets the room up before you arrive**.
- **Sidecar Container**: Like a **personal assistant** who works beside you during the day..

🕸️ What is Istio?

Istio is a **service mesh** — a way to **manage, secure, and observe communication** between microservices in Kubernetes (or other platforms).

It **sits between services** and controls how they talk to each other **without changing your app code**.

🚀 Why Use Istio?

Problem	How Istio Helps
Hard to manage service-to-service traffic	✅ Smart routing and traffic control
No visibility into app communication	✅ Telemetry, logs, and tracing built-in
Security between services is tricky	✅ Automatic TLS encryption
Need traffic splitting (canary, A/B)	✅ Fine-grained traffic routing

---

🧩 Istio Components

1. Envoy Proxy (Data Plane)

- Injected into each pod as a **sidecar**
- Handles all incoming/outgoing traffic
- Adds observability, security, routing

2. Istiod (Control Plane)

- Controls and configures all Envoy proxies
- Manages **policies**, **routing rules**, **certificates**

3. Citadel (part of Istiod)

- Issues and rotates **TLS certificates** for secure mTLS

4. Pilot (inside Istiod)

- Sends routing rules to the sidecar proxies

🌐 Istio Mesh Architecture

[Service A] <---> [Envoy A] [Envoy B] <--> [Service B]

| ^

| |

[Istiod] ------- Controls Both Proxies

-

- All traffic **goes through sidecars**
- Istiod tells Envoy **how to route, secure, and monitor** traffic

---

⚙️ **Istio Workflow**

✅ **Example: Canary Deployment with Istio**

```
apiVersion: networking.istio.io/v1beta1
kind: VirtualService
metadata:
  name: my-app
spec:
  hosts:
  - my-app.default.svc.cluster.local
  http:
  - route:
    - destination:
        host: my-app
        subset: v1
      weight: 80
    - destination:
        host: my-app
        subset: v2
      weight: 20
```

80% traffic goes to v1, 20% to v2
No app code changes needed!

🔐 **mTLS (Mutual TLS)**

Istio can automatically:

- **Encrypt traffic** between services
- **Authenticate** each service (who are you?)
- **Authorize** requests (are you allowed?)

📊 **Observability with Istio**

Integrates with:

- **Grafana** (metrics)
- **Prometheus** (monitoring)
- **Jaeger / Zipkin** (tracing)
- **Kiali** (mesh visualization)

🧠 **Real-Life Analogy**

Think of Istio like **air traffic control** for services in Kubernetes:

- **Envoy** = radio and sensors on each airplane ✈️
- **Istiod** = the central air traffic control 🛫
- It ensures planes (services) **fly safely, securely, and efficiently**

✅ **Summary**

Feature	**Istio Provides**
Traffic Routing	Canary, A/B, weighted splitting
Security	mTLS, authN/authZ
Observability	Metrics, logs, tracing
Reliability	Retry, timeout, circuit breaker

Let me know if you want a **hands-on example** of installing Istio and deploying a demo app!