[**https://github.com/avizway1/k8s**](https://github.com/avizway1/k8s)

[**https://www.youtube.com/watch?v=uo776QYclkQ&list=PLneBjIzDLEClHPKYqLlVbw43Xt1jT6Mlg&index=10**](https://www.youtube.com/watch?v=uo776QYclkQ&list=PLneBjIzDLEClHPKYqLlVbw43Xt1jT6Mlg&index=10)

**What is a Container?**

A **container** is like a **lunchbox** that holds everything needed for a meal. In software terms, a **container** is a lightweight package that includes an application and everything it needs to run (like code, system libraries, and dependencies). This ensures the application runs the same way, no matter where it's deployed.

**Real-world Example (Lunchbox)**

Imagine you are traveling and need to take food with you.

* You pack a lunchbox with rice, curry, a spoon, and a napkin.
* No matter where you go—home, office, or park—you open the lunchbox and start eating.
* You don’t depend on whether the location has food or utensils.

Similarly, a **container** packages an application along with its necessary files, so it runs consistently across different computers.

**Container Architecture: How It Works (Kitchen Analogy)**

Container architecture includes multiple components that work together to manage and run containers, just like a well-organized kitchen operates efficiently to serve meals.

**1. Container Runtime → Kitchen Stove (Cooking Equipment)**

The **container runtime** is like a **kitchen stove or cooking equipment** that actually **cooks the food** (runs the containers). Without it, no meal (container) can be prepared.

**Example:** **Docker Engine, containerd, CRI-O**

**2. Container Image → Pre-Packaged Meal Kit**

A **container image** is like a **pre-packaged meal kit** that includes all the ingredients and instructions needed to prepare a dish. Once built, you can reuse it multiple times, ensuring consistency.

**Example:** A **biryani meal kit** that anyone can use to cook biryani anywhere.

**3. Container Orchestration → Head Chef (Kitchen Manager)**

When multiple stoves (runtimes) and chefs (containers) are working together, someone must **coordinate operations, manage scaling, and ensure efficiency**. The **orchestration system** is like a **head chef or kitchen manager** who supervises everything.

**Example:** **Kubernetes (K8s), Docker Swarm, Amazon ECS, Amazon EKS, Apache Mesos**

**4. Container Registry → Cookbook/Recipe Shelf**

A **container registry** is like a **cookbook or recipe shelf** that stores multiple meal kits (images) for later use. Whenever you need to prepare a dish, you pick a recipe (pull an image) and start cooking (run a container).

**Example:** **Docker Hub, Amazon ECR, Azure Container Registry**

**Final Analogy Mapping:**

| **Container Concept** | **Easy Analogy** |
| --- | --- |
| **Container Runtime** | **Kitchen Stove (Cooking Equipment)** |
| **Container Image** | **Pre-Packaged Meal Kit** |
| **Container Registry** | **Cookbook/Recipe Shelf** |
| **Container Orchestration** | **Head Chef / Kitchen Manager** |

**Real-world Example (Restaurant)**

Think of a restaurant where meals (applications) are prepared efficiently using a structured process:

* A **cookbook or recipe shelf (container registry)** stores different meal kits (container images) so chefs can quickly pick a recipe and cook.
* A **kitchen stove (container runtime)** is used to cook meals based on a **pre-packaged meal kit (container image)**.
* The **head chef (Kubernetes)** manages multiple stoves and chefs, ensuring meals are prepared at the right time, in the right quantity.

**What Does a Container Orchestrator Do?**

A **container orchestrator** like **Kubernetes** (used in **EKS – Amazon Elastic Kubernetes Service**) is responsible for managing and automating containerized applications. Think of it as a **restaurant manager** who ensures everything runs smoothly.

**Challenges Without Container Orchestrators**

**Manual Scaling & Management** – You must manually start, stop, and scale containers, making it hard to handle high traffic.  
**Lack of Self-Healing** – If a container crashes, you must detect and restart it manually, leading to downtime.  
**Complex Networking & Load Balancing** – Managing container-to-container communication and distributing traffic requires manual setup.

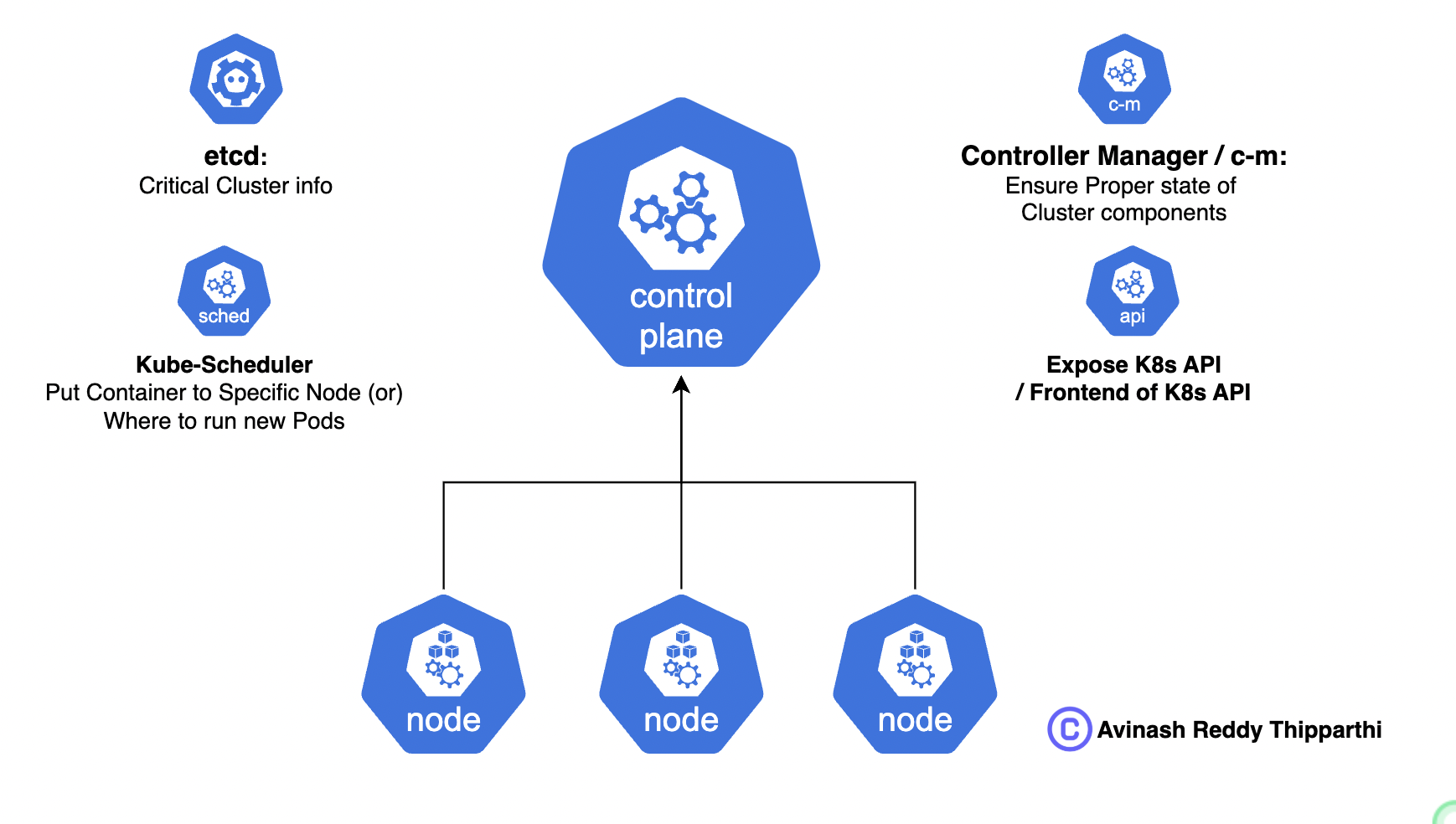
| **Restaurant Scenario** | **K8S Functionality** |
| --- | --- |
| The manager hires more chefs if customers increase. | **Auto-scaling** – EKS automatically adds more containers when traffic increases. |
| The manager replaces a sick chef with a new one. | **Self-healing** – If a container crashes, EKS replaces it automatically. |
| The manager ensures food reaches the right table. | **Networking & Service Discovery** – EKS routes traffic to the correct container. |
| The manager schedules chefs in different sections (Starters, Main Course, Desserts). | **Workload Scheduling** – EKS places containers on the best available servers (nodes). |
| The manager keeps a record of food stock. | **Storage Management** – EKS ensures containers have the storage they need. |
| The manager hires extra staff during peak hours. | **Load Balancing** – EKS distributes traffic to prevent overload. |

**Primary Components of Kubernetes (K8s) and Their Purpose**

Kubernetes has two main parts:

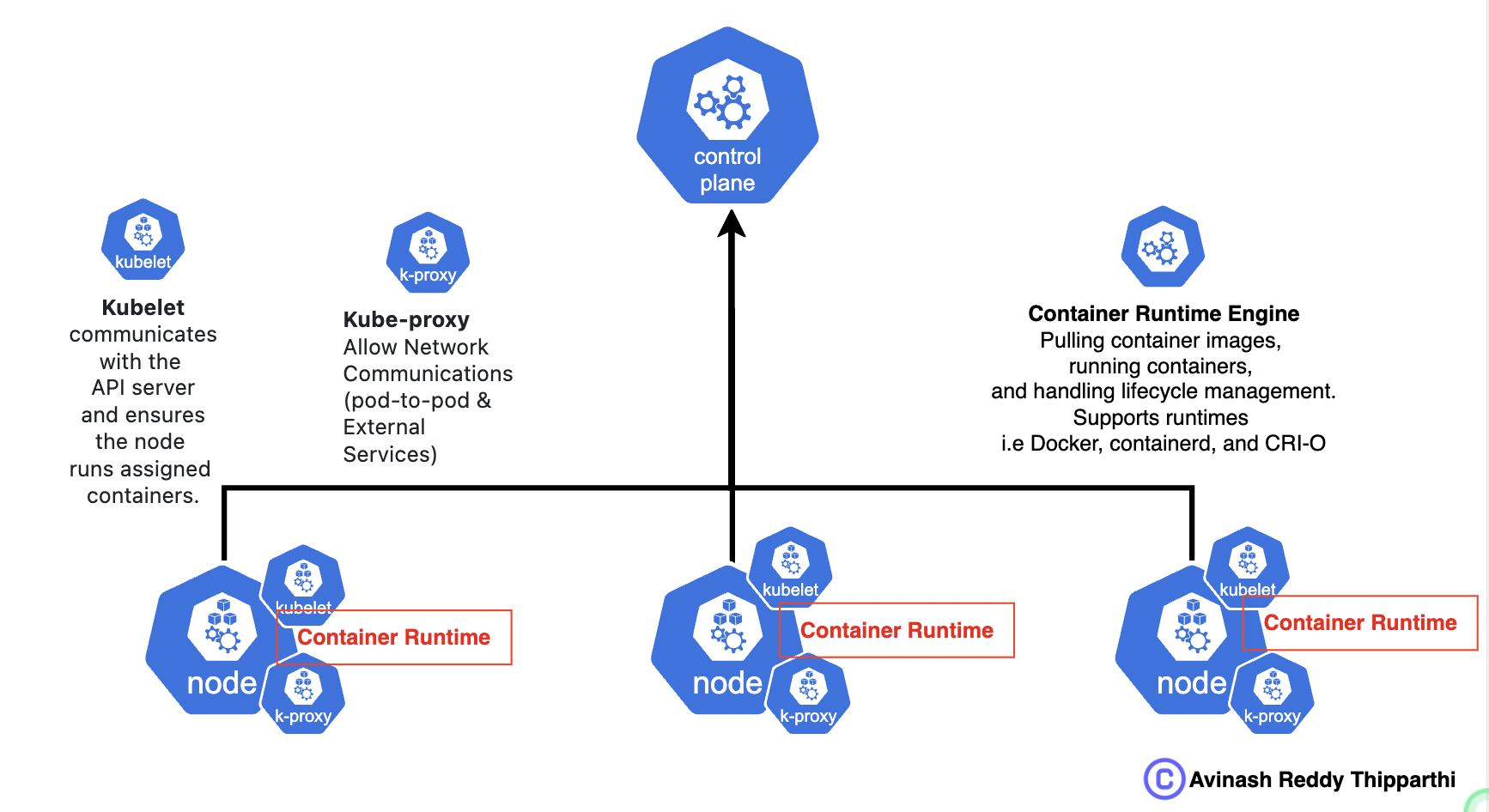
1️⃣ **Control Plane (Manages the Cluster)**  
2️⃣ **Worker Nodes (Runs Applications)**

**1. Control Plane Components (Brain of Kubernetes)**

[](https://private-user-images.githubusercontent.com/57447641/410424529-e92181e3-16ea-41c5-8aef-6ca57d13f010.png?jwt=eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9..BiD1cr1fMQPiDIJkw9wmb0bBaGMLsgMbhQF0SMc4DD8)

| **Component** | **Purpose / Job** |
| --- | --- |
| **API Server (kube-apiserver)** | The **front door** of Kubernetes; handles all API requests from users and tools (kubectl, dashboards, etc.). |
| **Controller Manager (kube-controller-manager)** | Ensures the desired state of the cluster (e.g., maintains replicas, replaces failed pods, manages node health). |
| **Scheduler (kube-scheduler)** | Decides **where** to run new pods based on resource availability. |
| **etcd (Key-Value Store)** | Stores all cluster data (state, configurations, deployments) in a distributed database. |
| **Cloud Controller Manager** | Integrates K8s with cloud providers (AWS, Azure, GCP) for managing networking, storage, and load balancers. |

**2. Worker Node Components (Executes Applications)**

[](https://private-user-images.githubusercontent.com/57447641/410427056-c13a41ad-ccc0-4a84-91f3-89fc6db3d6ec.png?jwt=eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9..cO_SUBoLqHCe_OhEhevPIWTJNyBg_F9fRFsu6N1D5L8)

| **Component** | **Purpose / Job** |
| --- | --- |
| **Kubelet** | Runs on each worker node; communicates with the API server and ensures the node runs assigned containers. |
| **Container Runtime** | Runs containers on the node (e.g., Docker, containerd, CRI-O). |
| **Kube Proxy** | Manages networking and ensures pods can communicate with each other and external services. |
| **Pods** | The smallest unit in Kubernetes; a pod contains one or more containers that share storage and networking. |

**Control Plane** – Manages and schedules workloads.  
**Worker Nodes** – Run the actual applications (containers).

**Detailed Explanation of Kubernetes Control Plane Components**

The **Control Plane** is the **brain** of Kubernetes. It manages the cluster, schedules workloads, and ensures everything is running as expected.

It consists of **five key components:**

**1️⃣ API Server (kube-apiserver) – The Front Door**

**Purpose:**

* Acts as the **entry point** for all Kubernetes commands (from kubectl, dashboards, or automation tools).
* It **validates** requests and forwards them to other components.
* Exposes the **Kubernetes API**, allowing internal and external users to communicate with the cluster.

**Example:**

* You run:

kubectl get pods

* + The API Server **receives the request**, **retrieves data from etcd**, and **returns the pod list**.

**2️⃣ etcd – The Database**

**Purpose:**

* Stores all **cluster data** (pod status, deployments, configurations).
* A **key-value store** that holds the **desired and current state** of Kubernetes.
* Highly available and distributed to avoid data loss.

**Example:**

* You deploy an application:

kubectl apply -f myapp.yaml

* + The API Server **writes the new deployment details into etcd**.
  + If a pod crashes, etcd still holds the data, allowing Kubernetes to restore it.

**3️⃣ Controller Manager (kube-controller-manager) – The Supervisor**

**Purpose:**

* Watches etcd and ensures the cluster **matches the desired state**.
* Runs different **controllers** that handle specific tasks.

**Important Controllers:**

* **Node Controller** – Monitors node health and replaces failed nodes.
* **Job controller** – Watches for Job objects that represent one-off tasks, then creates Pods to run those tasks to completion.
* **Service Account Controller** – Manages authentication tokens for pods.

**Example:**

* If a **node fails**, the **Node Controller** detects it and **moves pods to another node** automatically.

**4️⃣ Scheduler (kube-scheduler) – The Decision Maker**

**Purpose:**

* Decides **which node** will run a newly created pod.
* Picks a node based on **resource availability, taints, and tolerations**.

**Example:**

* You create a pod:
* apiVersion: v1
* kind: Pod
* metadata:
* name: my-app
* spec:
* containers:
* - name: my-container

image: nginx

* + The **API Server receives it**, and the **Scheduler assigns it to a node with enough resources**.

**5️⃣ Cloud Controller Manager – The Cloud Bridge**

**Purpose:**

* Connects Kubernetes to **cloud providers (AWS, Azure, GCP, etc.)**.
* Manages cloud-specific resources like **load balancers, storage, and networking**.

**Example in AWS EKS:**

* When you create a Kubernetes Service of type **LoadBalancer**, the Cloud Controller Manager:
  + Requests an AWS **Elastic Load Balancer (ELB)**.
  + Configures the ELB to forward traffic to the Kubernetes service.

**How They Work Together (Simple Flow)**

1. **You Deploy a Pod** (kubectl apply -f pod.yaml)
2. **API Server** receives the request and stores it in **etcd**.
3. **Scheduler** picks the best node for the pod.
4. **Controller Manager** ensures the pod runs properly.
5. **The Pod Runs Successfully!**

**Detailed Explanation of Kubernetes Worker Node Components**

A **Worker Node** is where the actual applications (containers) run. The Control Plane **manages** the cluster, but the Worker Nodes **execute** workloads.

Each Worker Node has **four key components** that ensure smooth operation:

**1️⃣ Kubelet – The Node Manager**

**Purpose:**

* **Main agent** running on each Worker Node.
* Talks to the **API Server** and ensures pods run correctly.
* Continuously monitors the health of pods and reports back to the Control Plane.

**Example:**

* The Scheduler assigns a pod to a Worker Node.
* The **Kubelet receives instructions** and starts the container using the container runtime (e.g., Docker, containerd).
* If the pod crashes, Kubelet **restarts it automatically**.

**2️⃣ Container Runtime – Runs Containers**

**Purpose:**

* Responsible for **pulling container images**, **running containers**, and **handling lifecycle management**.
* Kubernetes supports different runtimes like **Docker, containerd, and CRI-O**.

**Example:**

* You define a pod using an Nginx container:
* apiVersion: v1
* kind: Pod
* metadata:
* name: my-nginx
* spec:
* containers:
* - name: nginx-container

image: nginx:latest

* The **Kubelet asks the container runtime** to pull the nginx image and run it.
* The **runtime starts the container** inside the pod.

**3️⃣ Kube Proxy – Manages Networking**

**Purpose:**

* Handles **networking rules** and **pod communication** inside the cluster.
* Ensures **each pod gets a unique IP address**.
* Manages **Load Balancing** between pods in a service.

**Example:**

* A pod (frontend-app) wants to talk to another pod (backend-service).
* Kube Proxy ensures **network routing** happens correctly between them.
* If one backend pod fails, Kube Proxy **automatically routes traffic to healthy pods**.

**4️⃣ Pods – The Smallest Deployable Unit**

**Purpose:**

* A pod is **a wrapper around one or more containers**.
* It shares **storage, networking, and configuration** between containers.
* Each pod has a unique **IP address** inside the cluster.

**Example:**

* You deploy a pod with **two containers (app + logging sidecar)**:
* apiVersion: v1
* kind: Pod
* metadata:
* name: my-app-pod
* spec:
* containers:
* - name: app-container
* image: my-app
* - name: logging-container

image: log-collector

* Both containers share **storage and network**, allowing them to communicate easily.

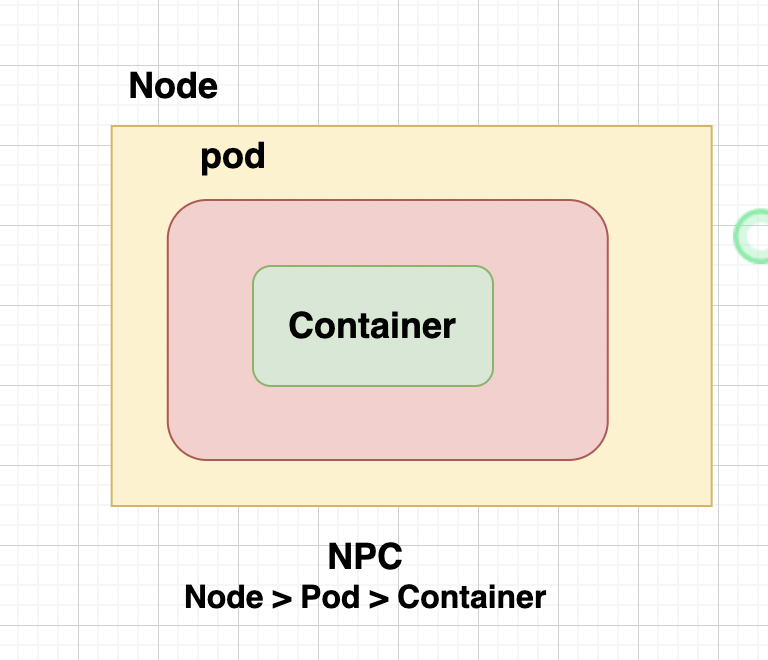
**How They Work Together (Simple Flow)**

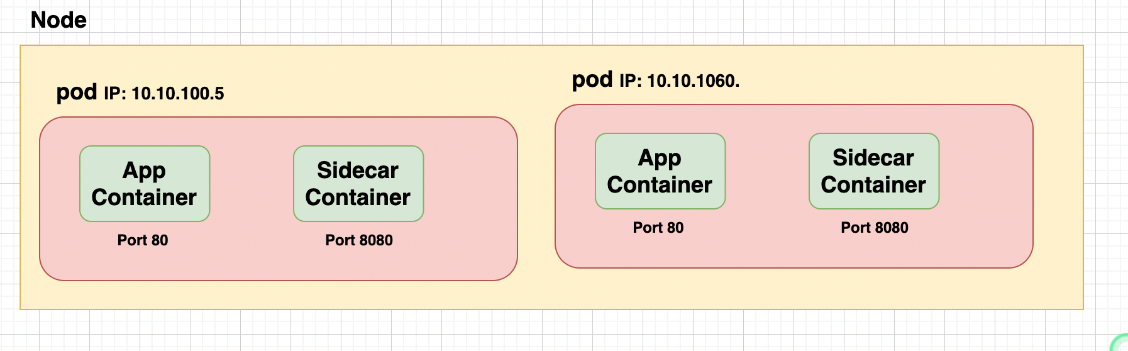
1️⃣ **Scheduler assigns a pod** to a Worker Node.  
2️⃣ **Kubelet receives the instruction** and starts the pod.  
3️⃣ **Container Runtime pulls the image** and runs the container inside the pod.  
4️⃣ **Kube Proxy sets up networking** so the pod can communicate with other pods/services.  
5️⃣ **Pod runs successfully** and serves requests!

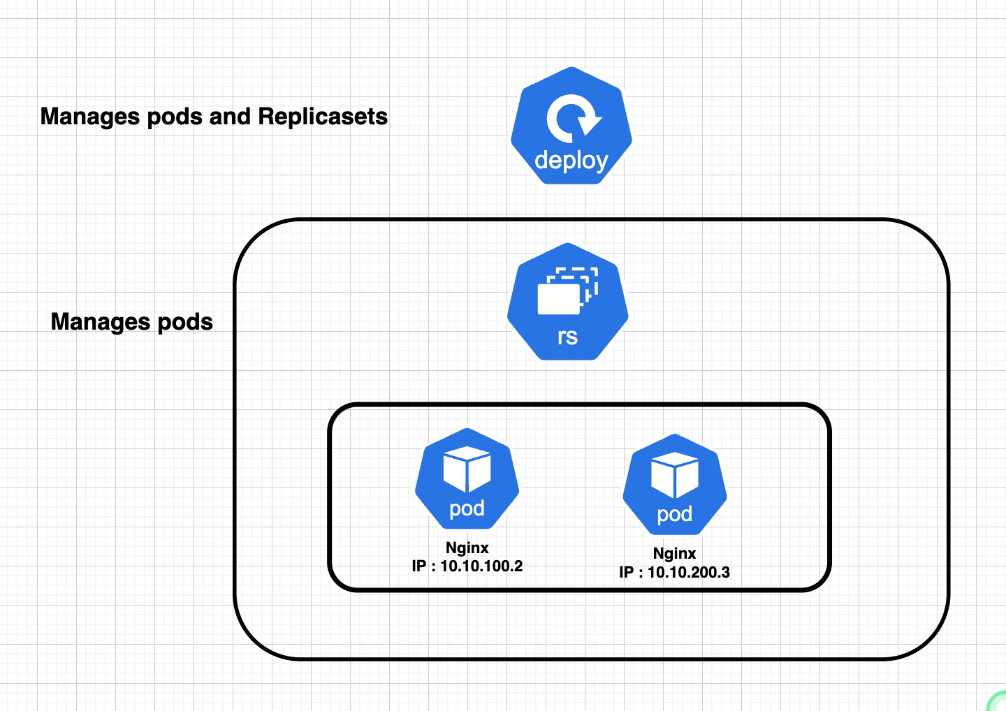
**What is a Container?**

A container is a lightweight, standalone, and executable unit of software that includes everything needed to run an application: **Code Runtime System libraries Dependencies**

**A Pod is the smallest and simplest deployable unit in Kubernetes. It represents one or more containers.**

[](https://private-user-images.githubusercontent.com/57447641/415888757-5bd555d1-6800-4869-a279-7b1d563fb1b1.png?jwt=eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9..G8saKzMCjoy-bn2u7Yhu2ThrBc9Ds18lHGUyvA6JdlM)

[](https://private-user-images.githubusercontent.com/57447641/415888792-a5a17cef-1b28-421f-8f26-0b134327fc18.png?jwt=eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9..pTTaGPw6Z9G5jawCb0wQ7Ksf9g0bjodEN68kTuZWs0o)

[](https://private-user-images.githubusercontent.com/57447641/415888835-5e8509d7-b167-4de8-aa43-e18b8698f7e8.png?jwt=eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9..2Dl_6A9qII5Eul7nt1M6Ec8pWS0tzP9E3KHfEadt_V8)

**What is a ReplicaSet in Kubernetes?**

A **ReplicaSet** is a Kubernetes object that ensures a **specified number of identical pod replicas** are always running. It acts as a safety net for your applications by automatically replacing failed or deleted Pods, guaranteeing high availability and fault tolerance.

**Key Features of ReplicaSet**

| **Feature** | **Description** |
| --- | --- |
| **Pod Availability** | Maintains the exact number of Pod replicas defined in the replicas field. |
| **Self-Healing** | Automatically replaces failed, crashed, or deleted Pods to match the desired state. |
| **Manual Scaling** | You can scale Pods up/down by updating the replicas count (e.g., kubectl scale). |
| **Label Selector** | Uses labels to identify and manage Pods (e.g., app: my-web-app). |
| **No Updates** | **Does NOT support rolling updates** – Use a **Deployment** for updating Pods. |

**How ReplicaSet Works (Example Scenario)**

Imagine you run a **web application** that needs **3 replicas** for **high availability**.

1️⃣ You define a **ReplicaSet** with replicas: 3.  
2️⃣ Kubernetes **creates 3 identical pods**.  
3️⃣ If 1 pod **crashes or is deleted**, ReplicaSet **replaces it immediately**.

**Example: ReplicaSet YAML**

apiVersion: apps/v1

kind: ReplicaSet

metadata:

name: nginx-replicaset

spec:

replicas: 3

selector:

matchLabels:

app: nginx

template:

metadata:

labels:

app: nginx

spec:

containers:

- name: nginx

image: nginx:latest

**What Happens?**

* **Creates 3 pods** with the Nginx container.
* If a pod **crashes**, it **creates a new one automatically**.
* If a pod is **deleted manually**, ReplicaSet **replaces it**.

**Limitations of ReplicaSet:**

* Cannot **update existing pods** (e.g., rolling out a new version).
* Only ensures the **desired number of replicas** but does not support version control.

**Scaling a ReplicaSet**

**Manually Scale Up/Down:**

kubectl scale rs nginx-replicaset --replicas=5

This increases the pod count from 3 → 5.

**Check Running Pods:**

kubectl get pods

**Delete a Pod & See Auto-Recovery:**

kubectl delete pod <pod-name>

ReplicaSet will **immediately create a new pod** to maintain 3 replicas.

**When Should You Use Deployment Instead?**

ReplicaSet **does not support rolling updates** (changing container images). If you want **version updates, rollbacks, or gradual updates**, use a **Deployment**, which internally uses a ReplicaSet.

**What is a Deployment in Kubernetes?**

A **Deployment** in Kubernetes is a higher-level abstraction that manages **ReplicaSets** and **allows rolling updates, rollbacks, and scaling** without downtime.

Unlike a **ReplicaSet**, which only ensures a fixed number of pods, a **Deployment** allows **smooth updates**, so you can upgrade your application without stopping it.

**Example Scenario:**

* You deploy **version 1** of your app.
* Later, you update the image to **version 2** → Deployment ensures a **smooth rollout**.
* If the update fails, you can **rollback to the previous version**.

**Why Use a Deployment?**

**Ensures a fixed number of replicas (via ReplicaSet)**  
**Supports rolling updates (zero downtime updates)**  
**Allows rollbacks (revert to a previous version if needed)**  
**Handles scaling up/down automatically**

**How Deployment Works (Example Scenario)**

Imagine you run an **Nginx web server** with **3 replicas** for **high availability**.

1️⃣ You create a **Deployment** named nginx-deployment with replicas: 3.  
2️⃣ Kubernetes creates a **ReplicaSet**, which in turn starts **3 pods** running Nginx.  
3️⃣ If you update the **Nginx image version**, the Deployment performs a **rolling update**.  
4️⃣ If the update **fails**, you can **rollback** to the previous version.

**Example: Deployment YAML (nginx-deployment)**

apiVersion: apps/v1

kind: Deployment

metadata:

name: nginx-deployment

spec:

replicas: 3 # Maintain 3 running pods

selector:

matchLabels:

app: nginx

template:

metadata:

labels:

app: nginx

spec:

containers:

- name: nginx

image: nginx:latest

ports:

- containerPort: 80

**Rolling Update Deployment (Zero Downtime) is most commonly used model**

* **How it works**: Updates **one pod at a time** instead of stopping everything at once.
* **Use case**: When **zero downtime** is required, like web applications.
* **Drawback**: Slow if the update is large.

**Example**

strategy:

type: RollingUpdate

rollingUpdate:

maxUnavailable: 1 # Max 1 pod can be unavailable during update

maxSurge: 1 # 1 extra pod can be created temporarily

**ReplicaSet vs Deployment**

| **Feature** | **ReplicaSet** | **Deployment** |
| --- | --- | --- |
| **Ensures a fixed number of pods** | Yes | Yes (via ReplicaSet) |
| **Automatically replaces failed pods** | Yes | Yes |
| **Supports Rolling Updates** | ❌ No | Yes |
| **Supports Rollbacks** | ❌ No | Yes |
| **Manages multiple ReplicaSets (for updates)** | ❌ No | Yes |
| **Recommended for production** | ❌ No | Yes |

**What is Service and why it is important.?**

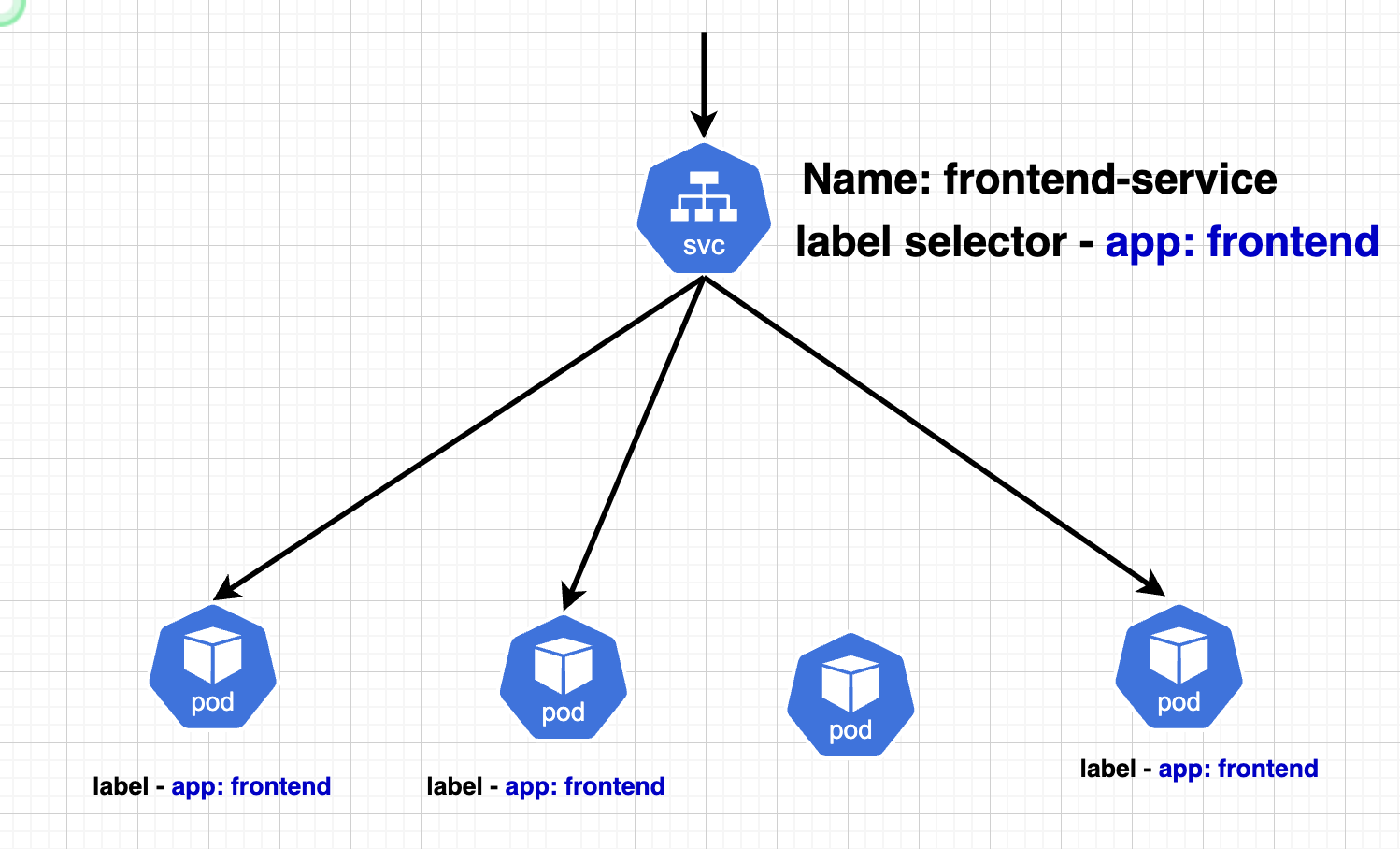
In Kubernetes, a **Service** is an abstraction that defines a logical set of Pods and a policy to access them. It provides a stable endpoint (IP address and DNS name) for communication with the Pods, even as the Pods themselves may come and go due to scaling, updates, or failures.

**Why Services are Needed:**

Pods in Kubernetes are ephemeral, meaning they can be created, destroyed, or replaced dynamically. Each Pod gets its own IP address, but these IPs are not stable. If you rely on Pod IPs directly, your application will break when Pods are replaced or scaled. Services solve this problem by providing a stable network identity for your application.

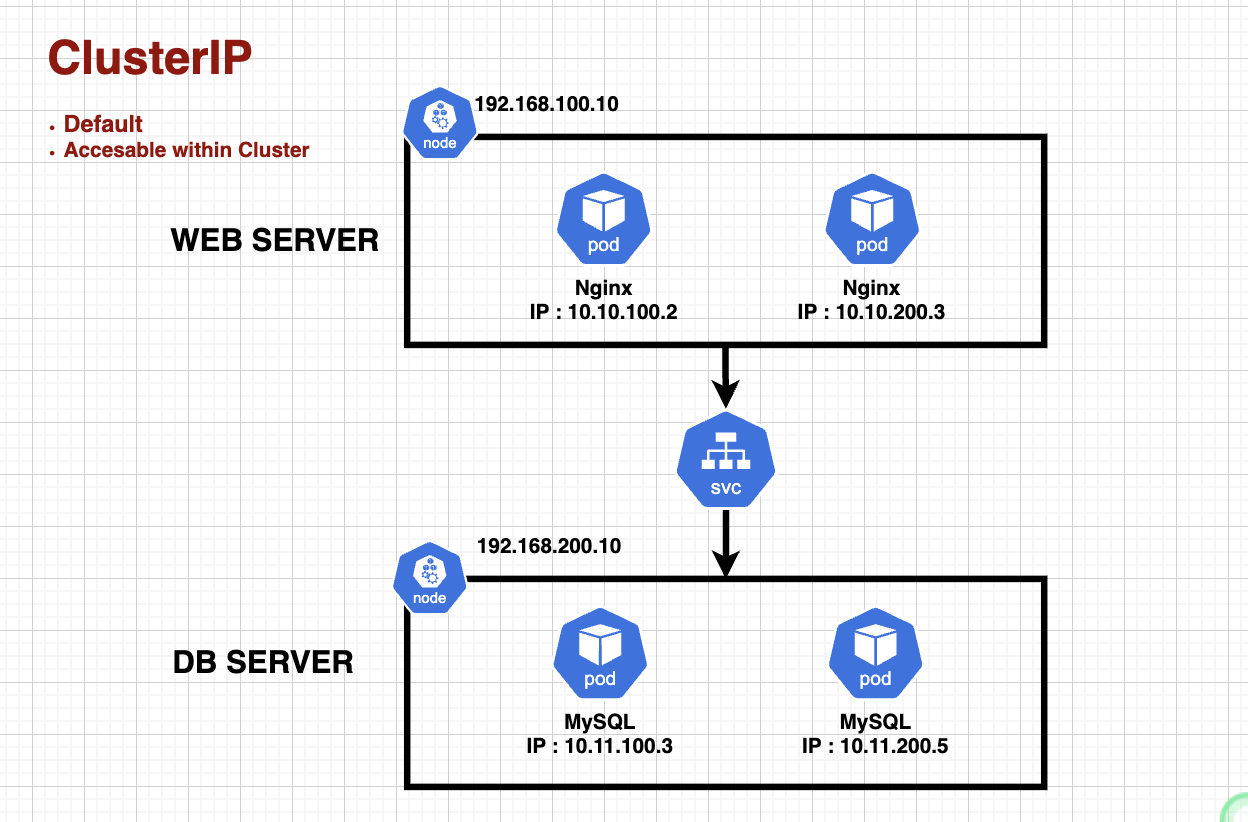
**How Services Work:**

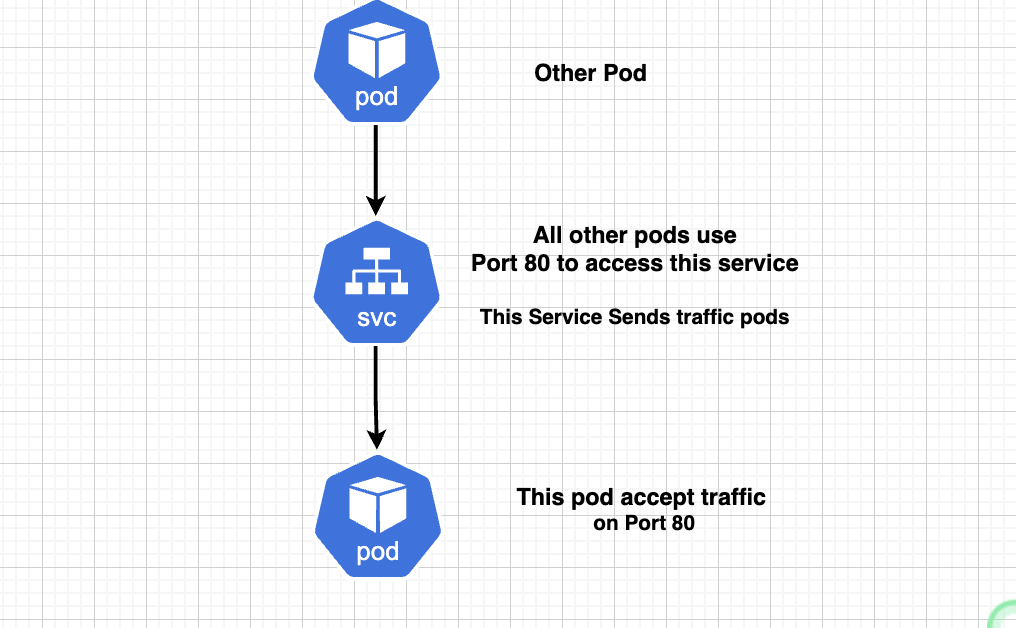
1. **Selector**: A Service uses a selector to identify the Pods it should route traffic to. For example, if the selector is app: my-app, the Service will route traffic to all Pods with the label app: my-app.
2. **Endpoints**: Kubernetes automatically creates an Endpoints object that lists the IPs and ports of the Pods matching the selector.
3. **kube-proxy**: The kube-proxy component running on each node ensures that the Service IP is reachable and routes traffic to the correct Pods.

[](https://private-user-images.githubusercontent.com/57447641/416966442-c1168d7a-a541-45f3-a04d-b094b4f80223.png?jwt=eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9..dqYOBm5yhuwq_FtJNzefYDE4ctyEq2Q2qQ_fKRYQil8)

**Types of Services in Kubernetes:**

1. **ClusterIP** (Default)
   * **What it does**: Exposes the Service on a cluster-internal IP. This makes the Service only reachable from within the cluster.
   * **Use case**: Communication between different components of your application (e.g., frontend to backend).
   * **Advantages**:
     + Secure, as it is not accessible from outside the cluster.
     + Provides a stable IP and DNS name for internal communication.

[](https://private-user-images.githubusercontent.com/57447641/416967367-19d3e498-84a1-4c88-bb52-dbf1e5b9e9a8.png?jwt=eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9..rPUjRpZ2D0qE9WK5rYCrJP1KSxdVBLopirVGnC218cQ)

[](https://private-user-images.githubusercontent.com/57447641/416968607-eaeb2d74-f4f4-4332-b7d1-5064860dbd21.png?jwt=eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9..GxFFqKqAzQZUvNTZqLgcRxl87siR56EYelv4lCGYQlQ)

Example YAML:

apiVersion: v1

kind: Service

metadata:

name: my-service

spec:

type: ClusterIP

selector:

app: my-app

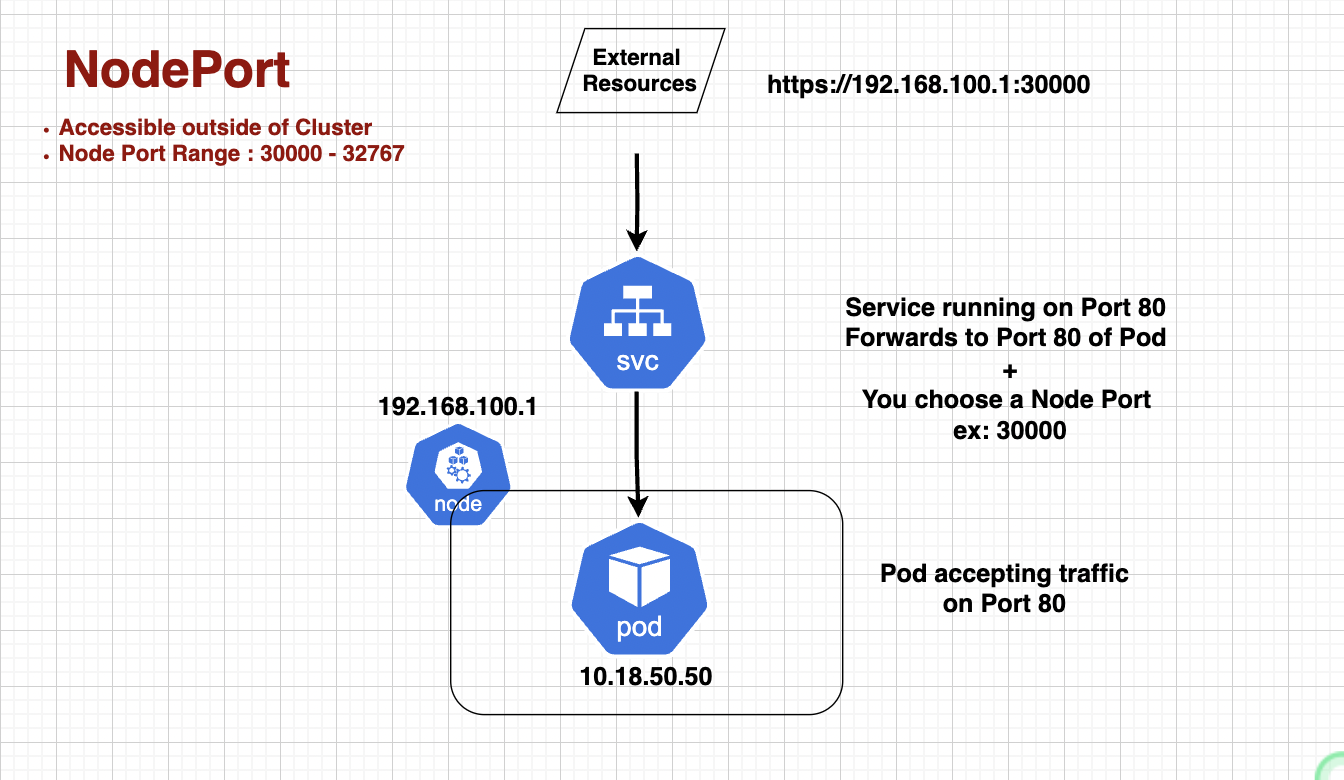
ports:

- protocol: TCP

port: 80

targetPort: 80

1. **NodePort**
   * **What it does**: Exposes the Service on a static port on each Node's IP. The Service is accessible from outside the cluster using <NodeIP>:<NodePort>.
   * **Use case**: When you need to expose your application to external traffic, but don’t want to use a LoadBalancer.
   * **Advantages**:
     + Simple way to expose your application externally.
     + Works without a cloud provider.

[](https://private-user-images.githubusercontent.com/57447641/416973413-3d473a43-caee-4a8d-97bd-f1fa7dd1cbe1.png?jwt=eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9..DJVz0YgmZArnW3GiBL5w0ZLXCycBZOW9_6jL5yB6TJI)

Example YAML:

apiVersion: v1

kind: Service

metadata:

name: my-service

spec:

type: NodePort

selector:

app: my-app

ports:

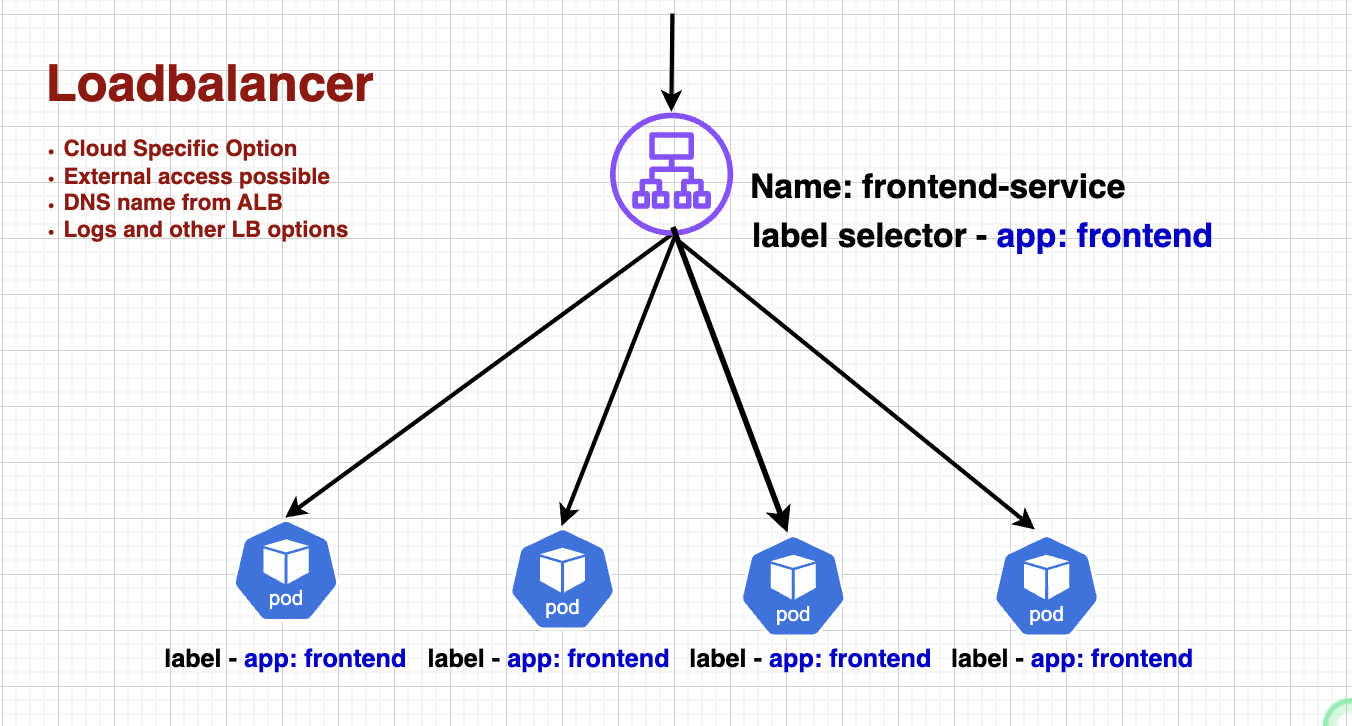
- protocol: TCP

nodePort: 30000

port: 80

targetPort: 80

1. **LoadBalancer**
   * **What it does**: Exposes the Service externally using a cloud provider's load balancer. Automatically assigns an external IP address.
   * **Use case**: When running Kubernetes in a cloud environment (e.g., AWS, GCP, Azure) and you want to expose your application to the internet.
   * **Advantages**:
     + Automatically provisions a load balancer.
     + Handles traffic distribution across multiple nodes.

[](https://private-user-images.githubusercontent.com/57447641/416973666-b7a5f3fa-4c49-458e-a1b7-303fdd4d6698.png?jwt=eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9..3V4joaiJC0E5dqpgqnDmEdj7q7Ah7_2ZcbF9auseBrI)

Example YAML:

apiVersion: v1

kind: Service

metadata:

name: my-service

spec:

type: LoadBalancer

selector:

app: my-app

ports:

- protocol: TCP

port: 80

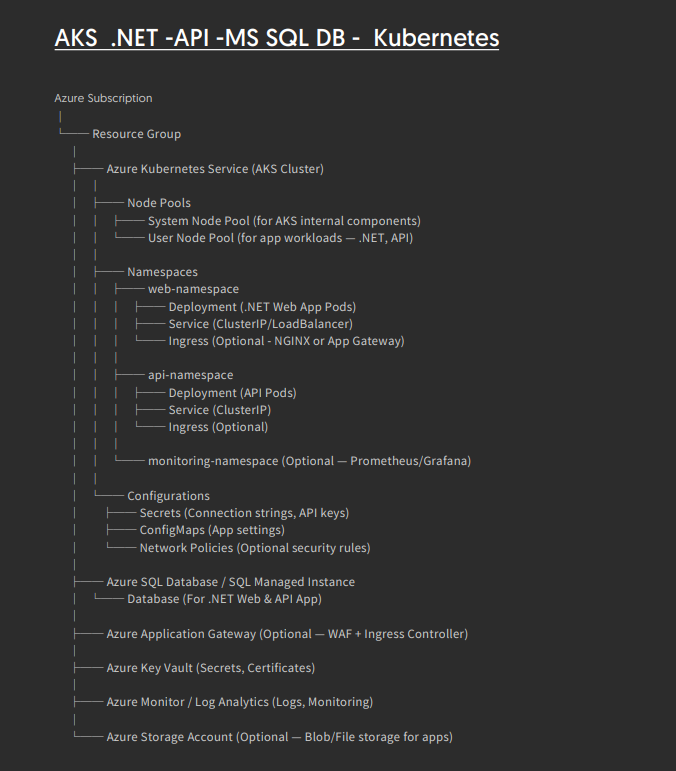
targetPort: 9376

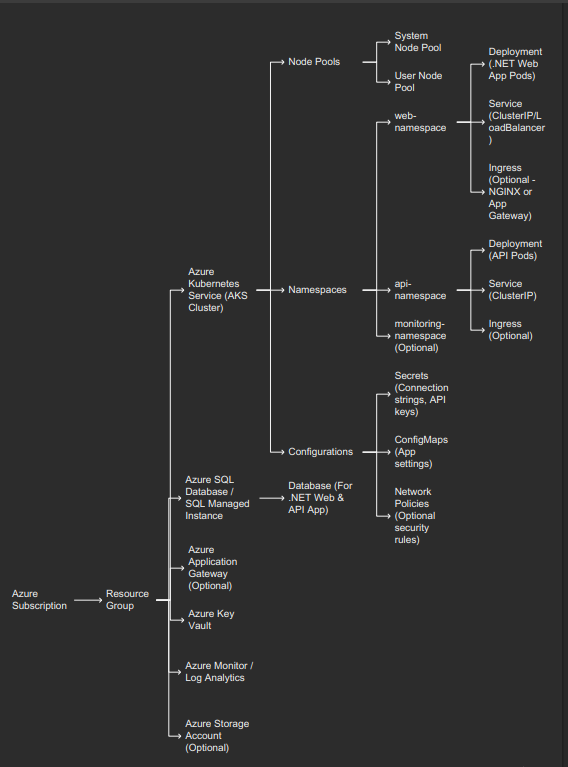
**Key Advantages of Services:**

1. **Stable Network Identity**: Services provide a stable IP and DNS name, even as Pods come and go.
2. **Load Balancing**: Distributes traffic evenly across Pods.
3. **Decoupling**: Applications don’t need to know about individual Pods; they just communicate with the Service.
4. **Scalability**: Easily scale your application up or down without affecting clients.
5. **Flexibility**: Different Service types allow you to expose your application in various ways (internally, externally, or to specific endpoints).

**Example Use Cases:**

* **ClusterIP**: Frontend app talking to a backend API.
* **NodePort**: Exposing a web app for testing or development.
* **LoadBalancer**: Exposing a production app to the internet.
* **ExternalName**: Connecting to an external database or third-party service.





# ---------------------- NAMESPACES ----------------------

apiVersion: v1

kind: Namespace

metadata:

name: web-namespace

---

apiVersion: v1

kind: Namespace

metadata:

name: api-namespace

# ---------------------- SECRETS ----------------------

apiVersion: v1

kind: Secret

metadata:

name: app-secrets

namespace: web-namespace

type: Opaque

stringData:

sql-connection: "Server=sqlserver.database.windows.net;Database=appdb;User Id=appuser;Password=YourStrongPassword!;"

---

apiVersion: v1

kind: Secret

metadata:

name: app-secrets

namespace: api-namespace

type: Opaque

stringData:

sql-connection: "Server=sqlserver.database.windows.net;Database=appdb;User Id=appuser;Password=YourStrongPassword!;"

# ---------------------- CONFIGMAPS ----------------------

apiVersion: v1

kind: ConfigMap

metadata:

name: app-config

namespace: web-namespace

data:

LOG\_LEVEL: "Information"

FEATURE\_FLAG: "true"

---

apiVersion: v1

kind: ConfigMap

metadata:

name: app-config

namespace: api-namespace

data:

LOG\_LEVEL: "Information"

FEATURE\_FLAG: "true"

# ---------------------- WEB DEPLOYMENT ----------------------

apiVersion: apps/v1

kind: Deployment

metadata:

name: webapp-deployment

namespace: web-namespace

spec:

replicas: 2

selector:

matchLabels:

app: webapp

template:

metadata:

labels:

app: webapp

spec:

containers:

- name: webapp-container

image: yourregistry.azurecr.io/webapp:latest

ports:

- containerPort: 80

env:

- name: ASPNETCORE\_ENVIRONMENT

value: "Production"

- name: ConnectionStrings\_\_DefaultConnection

valueFrom:

secretKeyRef:

name: app-secrets

key: sql-connection

# ---------------------- WEB SERVICE ----------------------

---

apiVersion: v1

kind: Service

metadata:

name: webapp-service

namespace: web-namespace

spec:

type: ClusterIP

selector:

app: webapp

ports:

- port: 80

targetPort: 80

# ---------------------- WEB INGRESS ----------------------

---

apiVersion: networking.k8s.io/v1

kind: Ingress

metadata:

name: webapp-ingress

namespace: web-namespace

annotations:

kubernetes.io/ingress.class: nginx

spec:

rules:

- host: webapp.example.com

http:

paths:

- path: /

pathType: Prefix

backend:

service:

name: webapp-service

port:

number: 80

# ---------------------- API DEPLOYMENT ----------------------

---

apiVersion: apps/v1

kind: Deployment

metadata:

name: apiapp-deployment

namespace: api-namespace

spec:

replicas: 2

selector:

matchLabels:

app: apiapp

template:

metadata:

labels:

app: apiapp

spec:

containers:

- name: apiapp-container

image: yourregistry.azurecr.io/apiapp:latest

ports:

- containerPort: 80

env:

- name: ASPNETCORE\_ENVIRONMENT

value: "Production"

- name: ConnectionStrings\_\_DefaultConnection

valueFrom:

secretKeyRef:

name: app-secrets

key: sql-connection

# ---------------------- API SERVICE ----------------------

---

apiVersion: v1

kind: Service

metadata:

name: apiapp-service

namespace: api-namespace

spec:

type: ClusterIP

selector:

app: apiapp

ports:

- port: 80

targetPort: 80

# ---------------------- API INGRESS ----------------------

---

apiVersion: networking.k8s.io/v1

kind: Ingress

metadata:

name: apiapp-ingress

namespace: api-namespace

annotations:

kubernetes.io/ingress.class: nginx

spec:

rules:

- host: apiapp.example.com

http:

paths:

- path: /

pathType: Prefix

backend:

service:

name: apiapp-service

port:

number: 80