Container Orchestration

Session 5 - 6 including Kubernetes Services (Pod, Node, Cluster)

Container

A **container** is a lightweight, portable, and self-sufficient software unit that includes everything needed to run an application: code, runtime, system tools, libraries, and dependencies.

- It ensures consistency across environments, meaning an application runs the same regardless of where it is deployed—whether on a developer's laptop, a testing server, or a cloud-based production system.
- Containers are isolated from each other, preventing conflicts between applications running on the same system.
- They are highly **scalable**, meaning they can be deployed in large numbers and can scale up or down based on demand.
- Unlike traditional virtual machines (VMs), containers share the same host OS kernel, making them **lightweight** and faster to start.

What is Container Orchestration?

Container orchestration is the **automated management** of containerized applications across multiple environments.

It ensures that containers are deployed, managed, scaled, networked, and maintained efficiently without manual intervention.

Think of it as an advanced **traffic control system** for containers—it decides:

- **Where** to deploy containers.
- **How** they communicate.
- **When** to scale them up or down.
- **What** to do if a container crashes.
- **How** to update them without downtime.

Key Functions of Container Orchestration

- **Automated Deployment**: Ensures containers are launched and assigned to available computing resources.
- **Scaling**: Increases or decreases the number of running containers based on demand.
- **Load Balancing**: Distributes traffic efficiently across containers to avoid overload.
- **Fault Recovery**: Detects and restarts failed containers automatically.
- **Networking & Service Discovery**: Ensures seamless communication between containers and external systems.

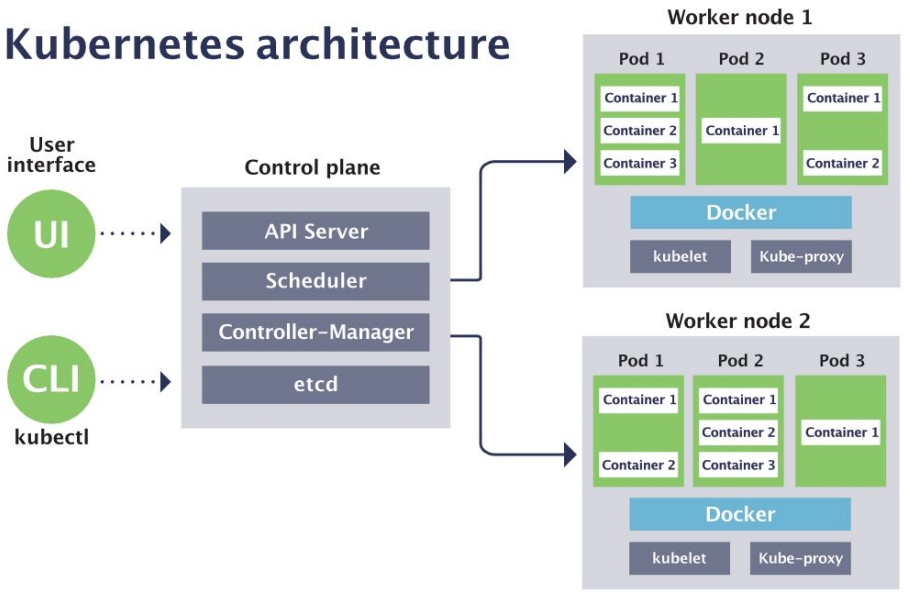
Example of a Container Orchestration Tool:

One of the most widely used tool for container orchestration is **Kubernetes**.

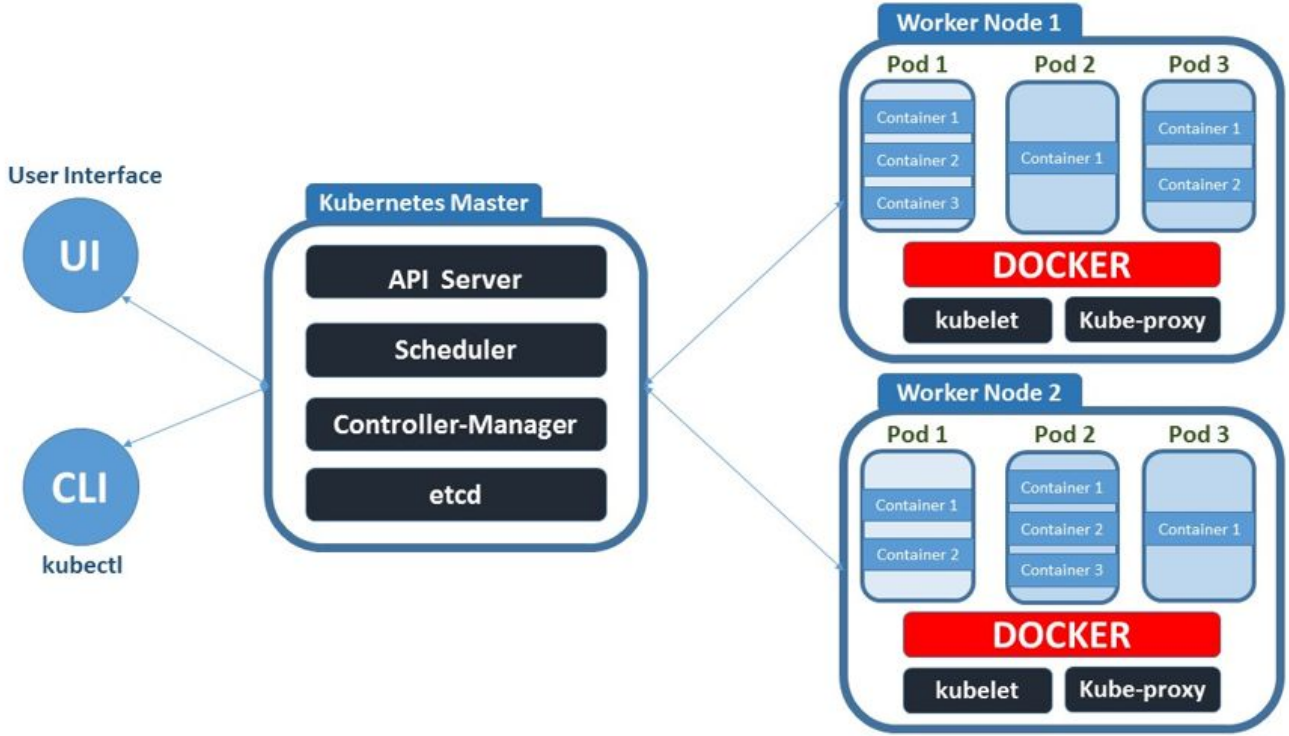
Orchestration Tool: Kubernetes

Originally developed by Google, it is now maintained by the **Cloud Native Computing Foundation (CNCF)**.

- Kubernetes groups containers into **Pods**, which are the smallest deployable units.
- It uses a **Master-Worker architecture**, where the Master node manages containerized applications across multiple worker nodes.
- Key components include:
  - **API Server**: Acts as the front-end for Kubernetes.
  - **Controller Manager**: Ensures the cluster's desired state.
  - **Scheduler**: Assigns workloads to nodes based on available resources.
  - **Kubelet**: Runs on each node to ensure containers are running.



Orchestration Tool: Kubernetes



Source: https://sensu.io/t

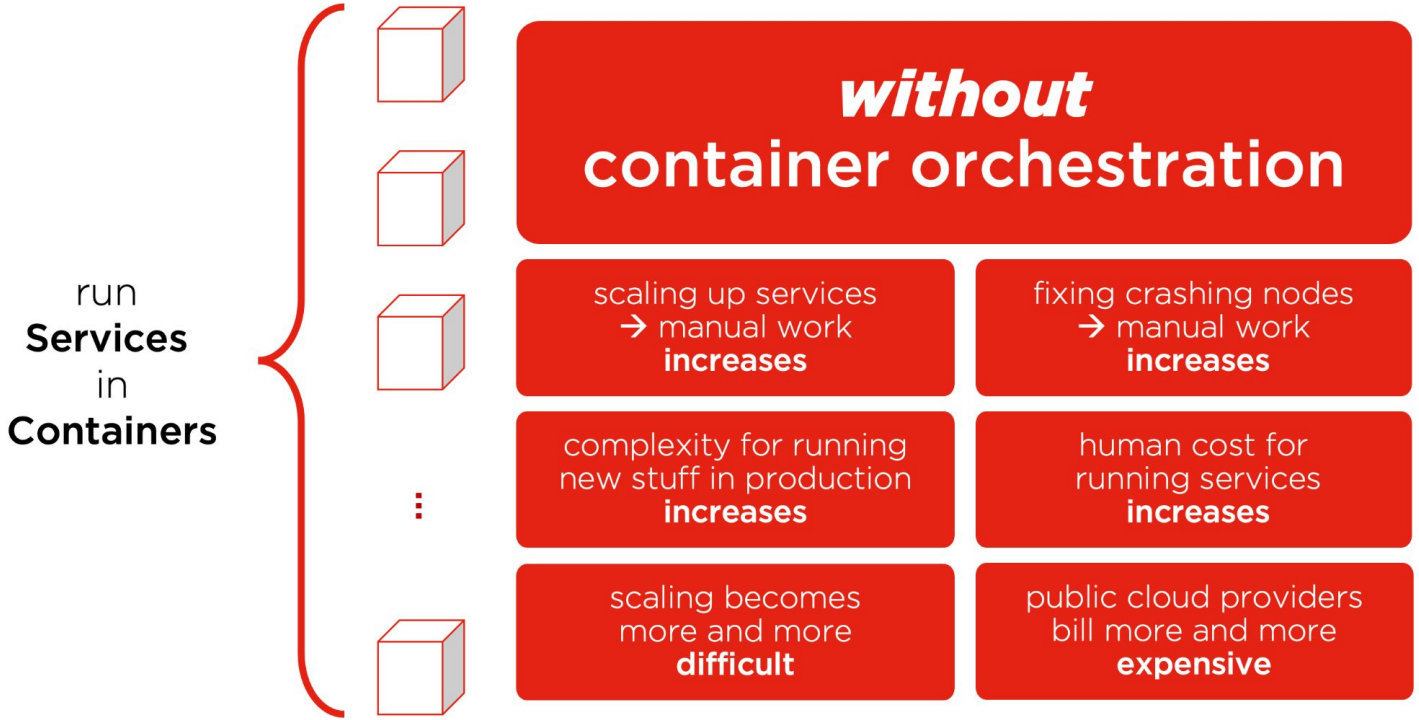
Orchestration Tools

Why Do We Need Container Orchestration?

When dealing with a single container, management is straightforward. However, in modern applications, multiple containers are required to work together efficiently. This introduces **complexity**, which is why container orchestration becomes essential which provides the following:

- **Scalability**: A real-world application consists of multiple microservices, each running in its own container. As demand grows, orchestration helps scale up or down automatically.
- **Inter-Container Communication**: Different containers (e.g., an application server and a database) must interact seamlessly. Orchestration ensures smooth communication.
- **Resilience & Fault Tolerance**: If a container fails, orchestration ensures another instance is launched automatically.
- **Load Balancing**: Efficient distribution of traffic among multiple container instances ensures optimized performance.
- **Automated Deployment & Updates**: Orchestration tools manage rolling updates and rollbacks, preventing downtime during updates.

Why Do We Need Container Orchestration?



Source: https://www.exoscale.com/

Kubernetes

Kubernetes (often abbreviated as **K8s**) is an **open-source** container orchestration platform designed to **automate** the deployment, scaling, and management of containerized applications. It enables organizations to efficiently run applications across **multiple machines**, whether in the **cloud, on-premise, or hybrid environments**.

Kubernetes was originally developed by **Google**, leveraging their **15 years of experience** managing production workloads, and is now maintained by the **Cloud Native Computing Foundation (CNCF)**.

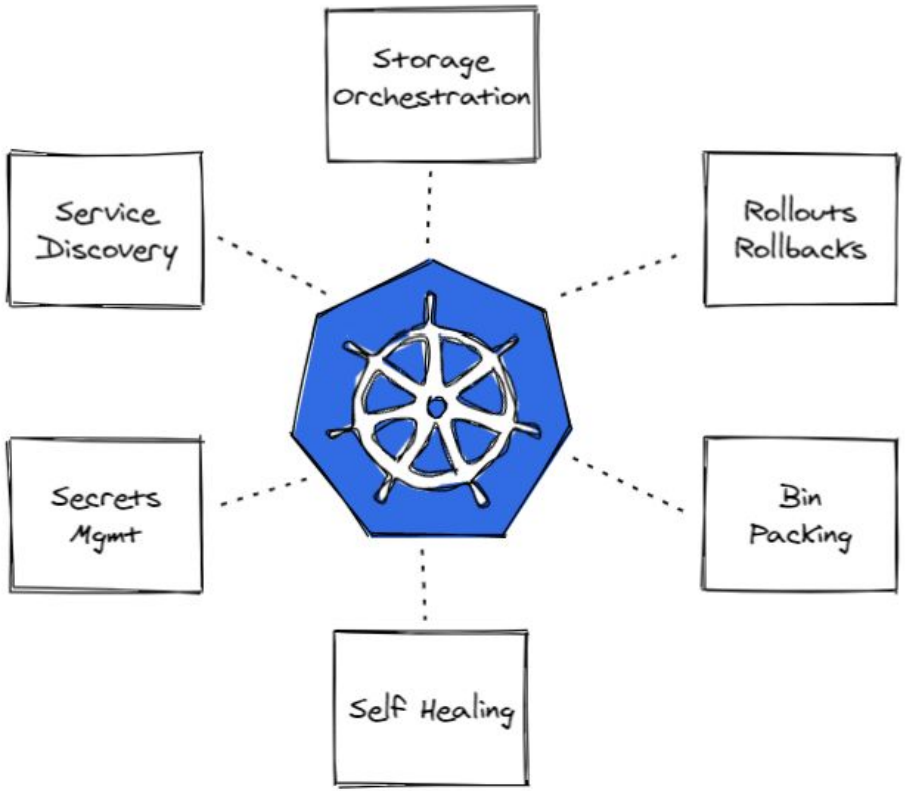
Key Features of Kubernetes

Automated Deployment & Scaling

- Kubernetes dynamically deploys applications by pushing configuration files to running containers.
- It supports **rolling updates** (updating applications without downtime) and **rollbacks** (reverting changes if something goes wrong).
- Ensures that the desired number of application instances (containers) are always running.

Service Discovery & Load Balancing

- Applications often consist of multiple interdependent services (e.g., web servers, databases).
- Kubernetes provides an internal DNS system to **automatically discover services** and allow seamless communication between them.
- It distributes incoming traffic **efficiently** across multiple containers to **prevent overload** on any single instance.



Source: https://www.sokube.io/blog/introduction-to-kubernetes-operators-1

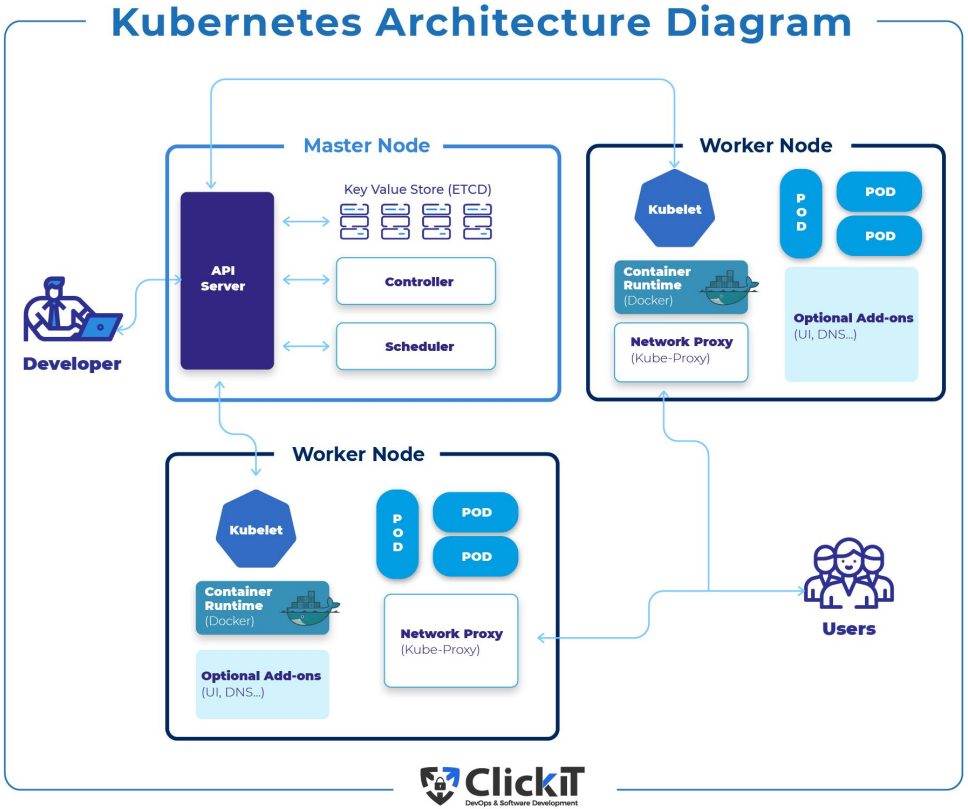
Feature: High Availability in Kubernetes (HA)

High Availability (**HA**) refers to an application's ability to remain operational and accessible **despite failures** in the underlying infrastructure. Kubernetes **ensures high availability** through intelligent automation mechanisms.

How Does Kubernetes Ensure High Availability?

- **Distributed Architecture:**
  - Kubernetes **distributes workloads** across multiple worker nodes. If one node fails, another node automatically takes over the workload.
  - The **Master Node** continuously monitors the state of the cluster and ensures all applications are running as expected.

Kubernetes Distributed Architecture



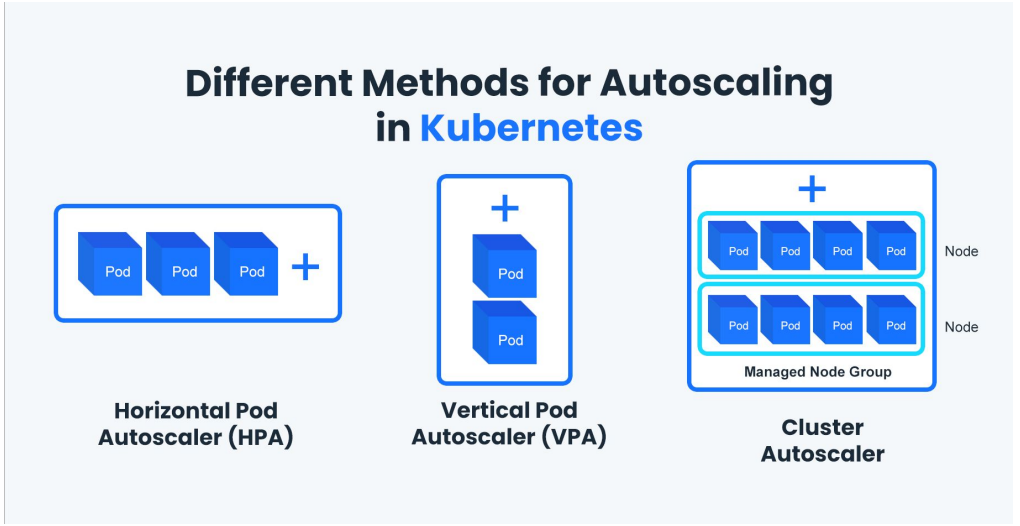
Source: https://www.clickittech.com/devops/kubernetes-architecture-diagram/

Feature: Auto-Scaling

- Kubernetes dynamically adjusts the number of running containers **based on demand**.

1. Horizontal Pod Autoscaler (HPA):

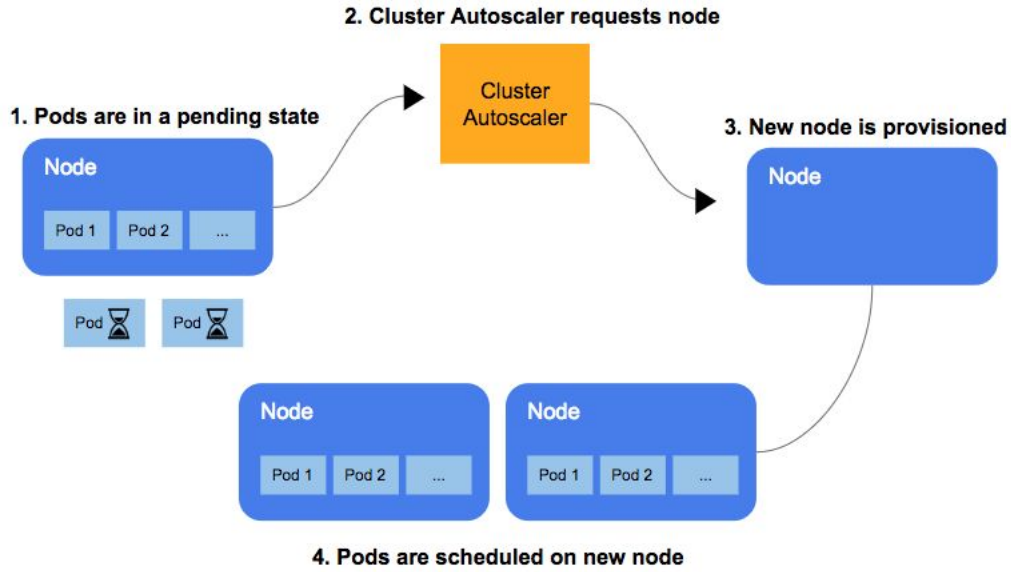
- **Increases or decreases the number of Pods**.
- It monitors metrics like **CPU/memory utilization** of existing Pods.
- If utilization exceeds a threshold, **more Pods are created**.
- If utilization drops, **Pods are terminated**.
- **Scales at the Pod level**, not individual containers inside the Pod.



Source: https://www.nops.io/blog/comprehensive-guide-kubernetes-autoscaling/

2. Cluster Autoscaler:

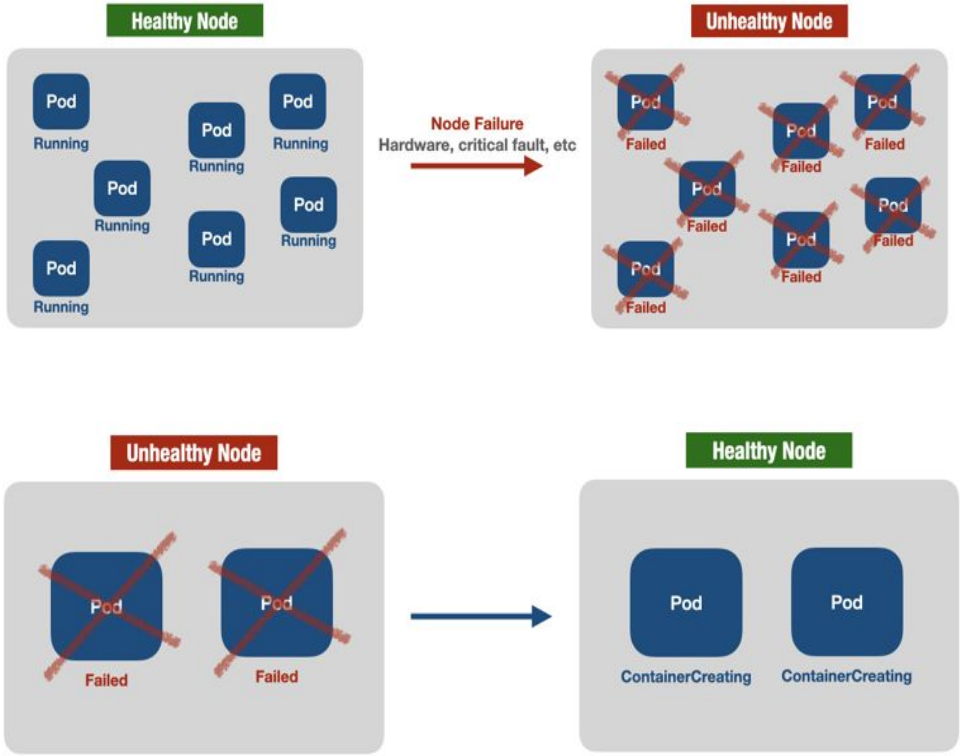
- **Increases or decreases the number of worker nodes**.
- If new Pods cannot be scheduled due to resource limits, **new nodes are added**.
- When nodes are underutilized, **they may be removed**.
- **Scales at the node level**, not directly at the Pod/container level.



Source: https://www.nops.io/blog/comprehensive-guide-kubernetes-autoscaling/

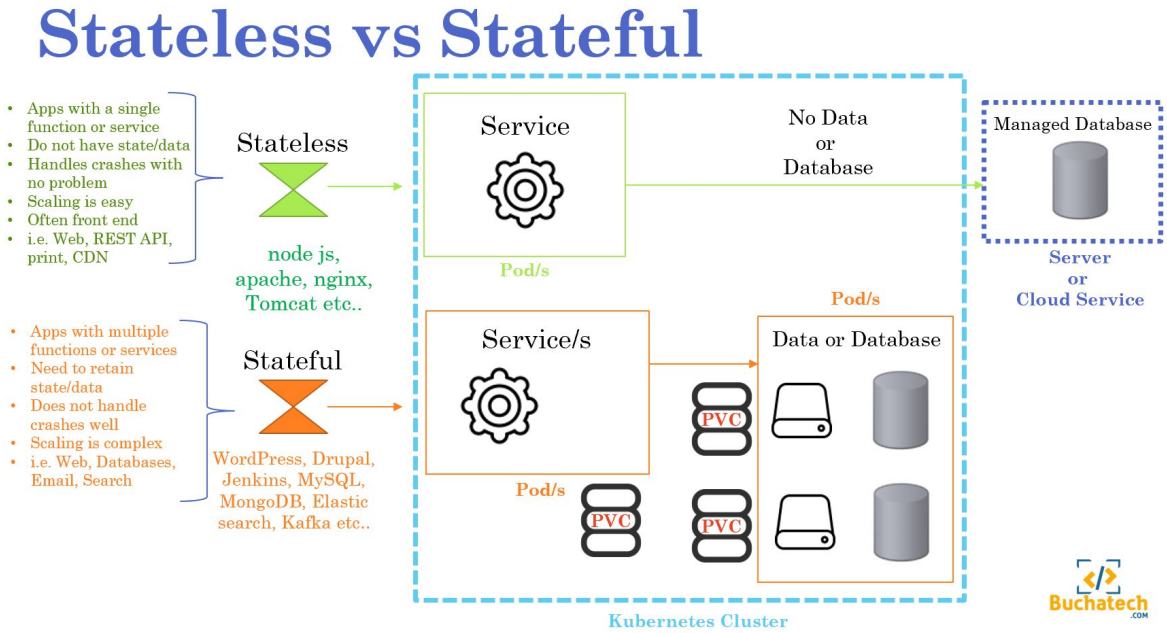
Feature: Self-Healing & Failover Mechanism

- If a **container crashes**, Kubernetes automatically restarts it.
- If a **node fails**, the system reschedules workloads onto healthy nodes.
- If an update goes wrong, Kubernetes **rolls back** to the previous stable state.



Feature: Stateful Application Management

- Kubernetes ensures **stateful applications** (like databases) remain highly available by using **Persistent Volumes (PVs)** and **StatefulSets**.
- In case of node failures, Kubernetes maintains **data integrity** and ensures applications continue running smoothly.



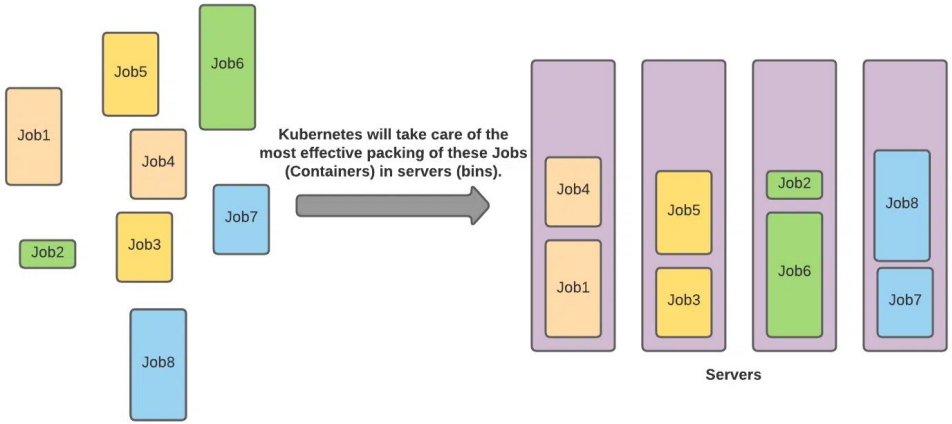
Source: https://www.buchatech.com/2022/08/running-stateful-apps-in-kubernetes/

Feature: Automatic Bin Packing (Intelligent Resource Allocation)

Kubernetes efficiently places containers on available **worker nodes** to maximize **resource utilization** while maintaining application performance.

- Instead of running containers **randomly**, Kubernetes schedules them based on:
  - **CPU and Memory Requests & Limits**: Ensures that each container gets the necessary resources while preventing overconsumption.
  - **Node Capacity & Constraints**: It fits containers onto nodes based on available CPU, memory, and custom constraints like affinity rules.
  - **Pod Priorities**: Ensures critical workloads get placed before less important ones.

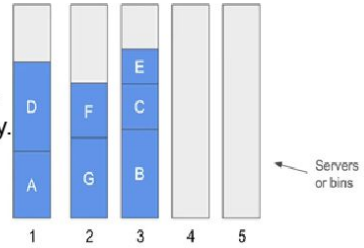
Automatic Bin Packing



Automatic Bin Packaging

We have 5 servers each having 10 GB of RAM and we have jobs to run on these 5 servers. Every Job has different resource

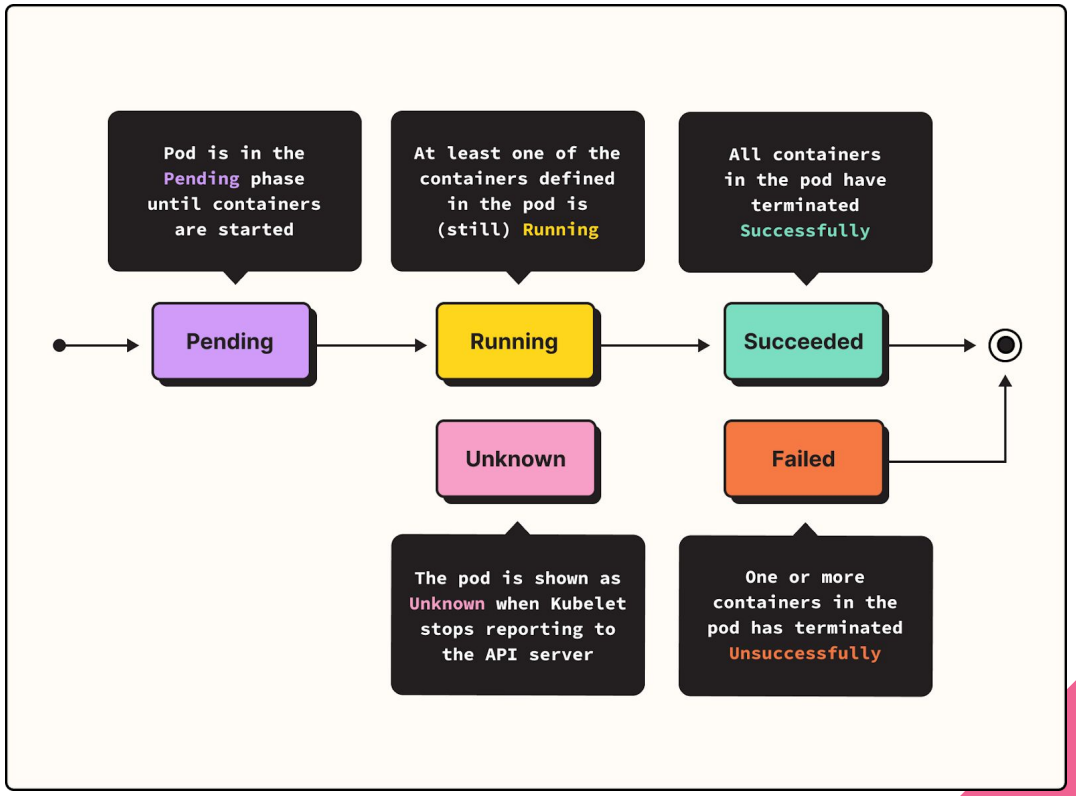
Kubernets will take care of packaginf these jobs in servers in the most optimal/efficient way.



Feature Self-Healing (Built-in Fault Recovery)

Kubernetes ensures high availability by **automatically detecting and recovering from failures** without manual intervention.

- **Container Restarts**: If a container crashes, Kubernetes automatically restarts it.
- **Node Failover**: If a node becomes unresponsive, Kubernetes shifts workloads to healthy nodes.
- **Health Checks**:
  - **Liveness Probes**: Checks if a container is still running; if not, Kubernetes restarts it.
  - **Readiness Probes**: Ensures a container is ready to accept traffic before adding it to a service.
- **Pod Replacement**: If a container within a Pod fails repeatedly, Kubernetes automatically terminates the Pod and creates a new one to maintain application availability.

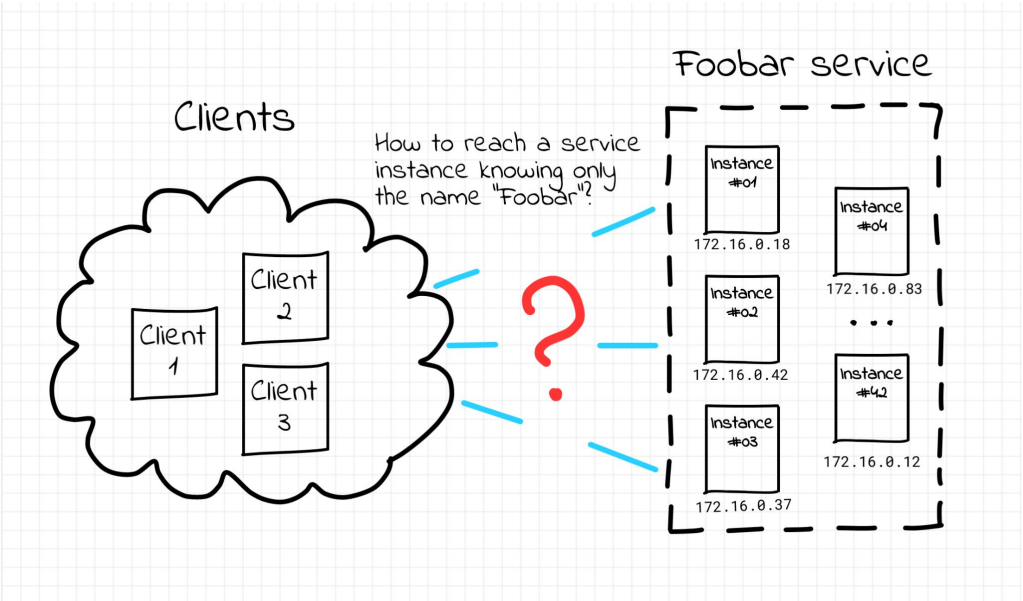


Source: https://www.groundcover.com/blog/kubernetes-benefits

Service Discovery: How Containers Find Each Other

Unlike traditional monolithic applications, containerized microservices need a **dynamic way** to communicate with each other because:

- Containers are **ephemeral** (they can be restarted, rescheduled, or moved).
- Their **IP addresses keep changing** dynamically.
- Manually configuring IP addresses is **not feasible** in large-scale deployments.



Kubernetes Service Discovery

Kubernetes provides two primary methods for **service discovery**:

1. **DNS-Based Discovery (Preferred Method)**
   - Kubernetes assigns a **DNS name** to each service.
   - Containers can refer to services using **human-readable names** instead of hardcoded IP addresses.
   - Example: A web server can communicate with a database service using `database-service.default.svc.cluster.local` instead of an unknown IP.
2. **IP-Based Discovery (Less Common)**
   - Kubernetes can expose services using **static cluster IPs**, but this approach is **less flexible** than DNS.
   - Typically used for **internal communication** within the cluster.

Load Balancing: Distributing Traffic Efficiently

As applications scale, incoming traffic must be **distributed across multiple instances** to:

- Prevent overloading a single container.
- Ensure **high availability** and **fault tolerance**.
- Provide a **smooth user experience**.

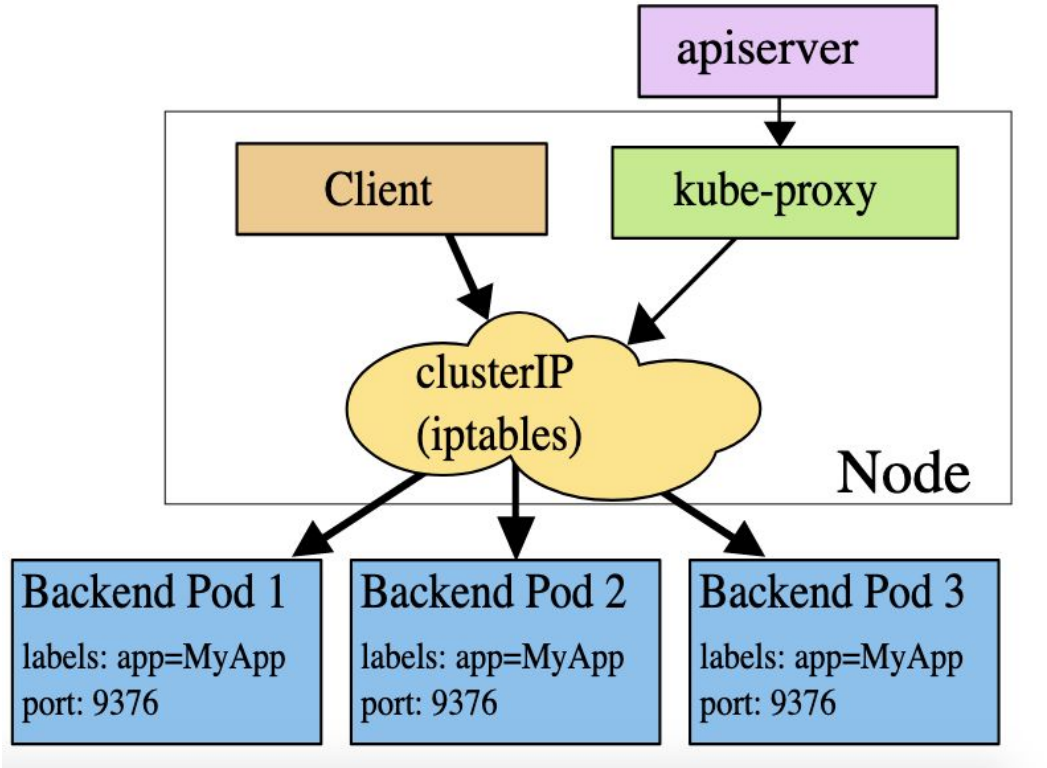
How Kubernetes Handles Load Balancing

1. **Internal Load Balancing (ClusterIP)**
   - By default, Kubernetes assigns a **ClusterIP** to services, allowing them to distribute requests evenly across **all healthy pods** within the cluster.
2. **External Load Balancing**
   - For services that need **external access**, Kubernetes supports:
     - **NodePort**: Exposes the service on a static port of each node.
     - **LoadBalancer**: Integrates with cloud providers (AWS, GCP, Azure) to create external load balancers.
3. **Ingress for Advanced Routing**
   - Kubernetes **Ingress** allows defining custom routing rules, **TLS termination**, and **host-based traffic distribution**.
   - Example: It can route `api.example.com` to backend service A and `shop.example.com` to backend service B.

ClusterIP

Internal Load Balancing (ClusterIP)

- By default, Kubernetes assigns a **ClusterIP** to services, allowing them to distribute requests evenly across **all healthy pods** within the cluster.



NodePort

1. **External Load Balancing**
    - For services that need **external access**, Kubernetes supports:
        - **NodePort**: Exposes the service on a static port of each node.

ClusterIP

1. **Internal Load Balancing (ClusterIP)**
   - By default, Kubernetes assigns a **ClusterIP** to services, allowing them to distribute requests evenly across **all healthy pods** within the cluster.
2. **External Load Balancing**
   - For services that need **external access**, Kubernetes supports:
     - **NodePort**: Exposes the service on a static port of each node.
     - **LoadBalancer**: Integrates with cloud providers (AWS, GCP, Azure) to create external load balancers.
3. **Ingress for Advanced Routing**
   - Kubernetes **Ingress** allows defining custom routing rules, **TLS termination**, and **host-based traffic distribution**.
   - Example: It can route `api.example.com` to backend service A and `shop.example.com` to backend service B.

Relationship between ClusterIP, NodePort, and LoadBalancer

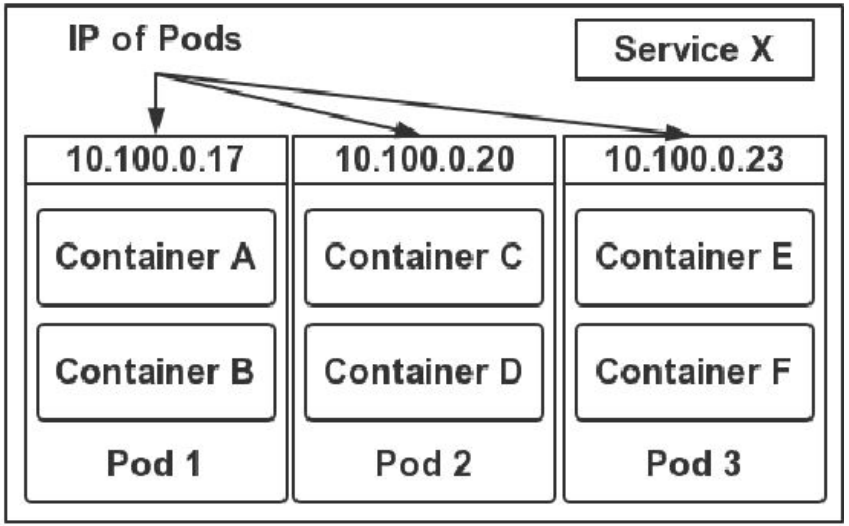
Think of Kubernetes like this:

- **Container**: The actual application running (e.g., an Nginx server or your custom API).
- **Pod**: A wrapper around one or more containers. It's the smallest deployable unit in Kubernetes.
- **Node**: A physical or virtual machine in the cluster where Pods run.
- **Service**: A Kubernetes object that provides a **stable IP and DNS name** to access Pods. It **decouples access from Pod details** (like IPs, which change when Pods restart).
- **So, when you create a Service:**

  You are saying:

  "*Hey Kubernetes, please give me a stable way to reach my app Pods, regardless of how many there are, where they run, or what their current IPs are.*"

  You are **not creating** a Pod, container, or Node — those already exist (or are created via a Deployment, StatefulSet, etc.). The Service just **sits on top** and **routes traffic** to the correct Pods based on label selectors.



Step 1: ClusterIP – The Default, Internal Door

By default, when you create a service in Kubernetes, it's of type `ClusterIP`. This makes your service reachable **only inside the cluster**. It assigns a **virtual IP** that other Pods in the cluster can use to talk to it. This is ideal for backend services like databases or internal microservices that should **not be accessible from outside**.

In a real-world system, nearly all backend components use `ClusterIP`. These services are only exposed to other services or apps inside the Kubernetes environment. For example, a web frontend talks to an authentication service via its ClusterIP address.

Step 2: NodePort – The Gateway to the Outside

Now suppose you want to make that frontend web application accessible from outside the cluster. You use a `NodePort` service. This exposes the service **on a fixed port on every node** (like port `30080`). You can now access the service via `<any-node-IP>:30080`.

However, this approach has limitations:

- You have to remember node IPs and port numbers.
- Traffic isn't load-balanced automatically — it just lands on whichever node the request hits.
- It exposes ports directly on your nodes, which may not be ideal for security.

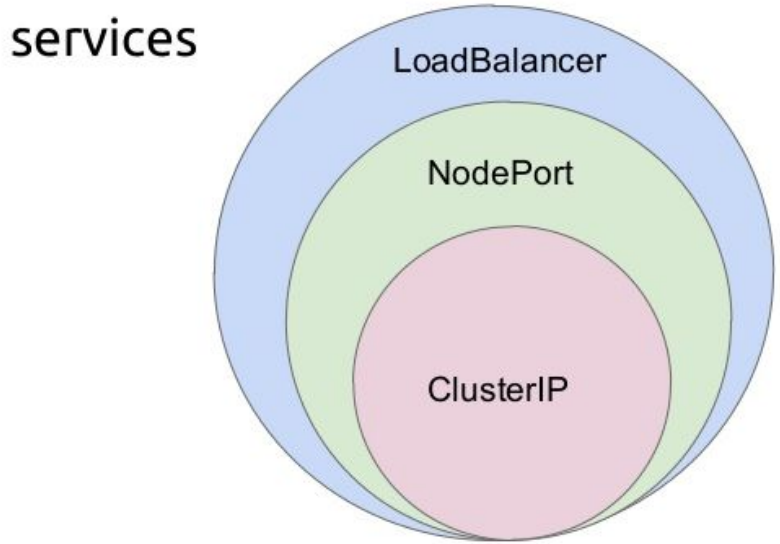
In most production systems, `NodePort` is **used internally** — not as a direct exposure method — but as the **backend for external load balancers or Ingress controllers**.

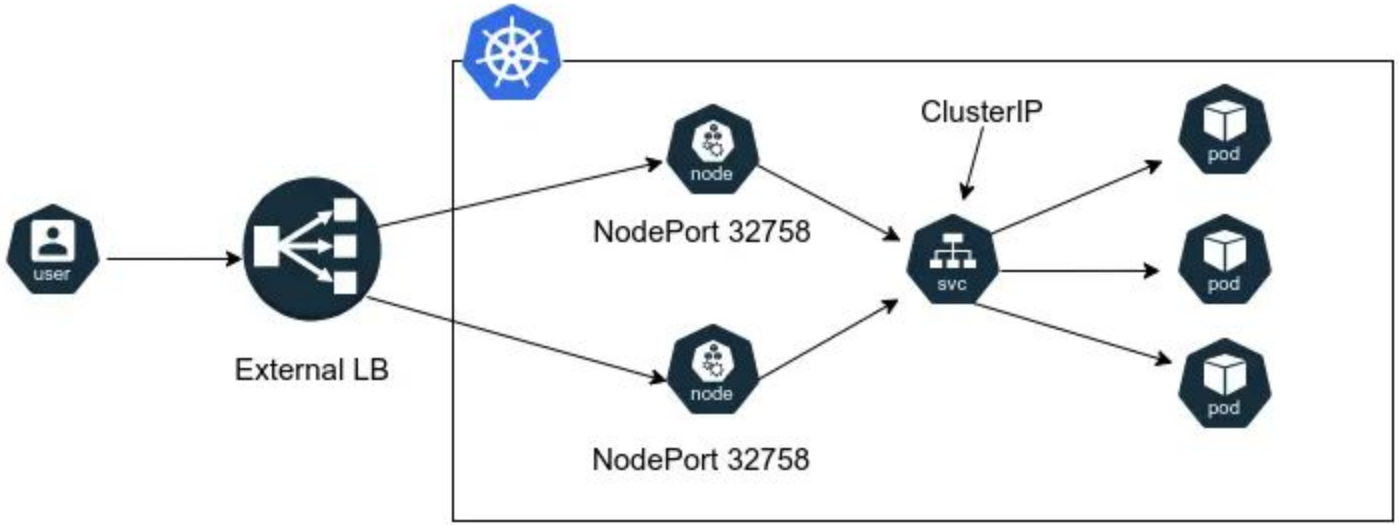
Step 3: LoadBalancer – The Cloud Way

When you're running Kubernetes in a cloud environment (like AWS, GCP, Azure), you can use a `LoadBalancer` service. This tells Kubernetes to **provision a cloud provider's load balancer** (like AWS ELB) and assign it a **public IP address**.

Internally, the `LoadBalancer` service **wraps a `NodePort`** behind the scenes. The cloud load balancer forwards traffic to the node's IP on the assigned NodePort, and Kubernetes routes it to the right Pod.

This is the **default choice** for exposing a service to the public internet in the cloud. It's managed, scalable, and reliable.


services
LoadBalancer
NodePort
ClusterIP


user
External LB
node
NodePort 32758
node
NodePort 32758
ClusterIP
svc
pod
pod
pod

Kubernetes Deployment

`myapp-deployment.yaml`

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: myapp
spec:
  replicas: 3
  selector:
    matchLabels:
      app: myapp
  template:
    metadata:
      labels:
        app: myapp
    spec:
      containers:
      - name: myapp-container
        image: myapp:latest
```

`kubectl apply -f myapp-deployment.yaml`

Kubernetes will:

1. **Create a Deployment object** called `myapp`.
2. The Deployment will:
   - Launch 3 Pods.
   - Each Pod will run one container:
     - Name: `myapp-container`
     - Image: `myapp:latest` (pulled from your container registry, like **DockerHub**)
3. Each Pod will be labeled with `app: myapp`.
4. Kubernetes will constantly monitor and **maintain 3 running Pods** at all times:
   - If one crashes, Kubernetes restarts it.
   - If you scale up to 5 replicas later, Kubernetes creates 2 more Pods automatically.

What does a Deployment focus on?

- Creating and managing **Pods**.
- Specifying the **container image** to run.
- Defining how many **replicas** (Pods) you want.
- Managing updates, restarts, and rollbacks.

How to create a Kubernetes ClusterIP Service

`clusteripservice.yml`

```
apiVersion: v1
kind: Service
metadata:
  name: myapp-service
spec:
  selector:
    app: myapp
  ports:
    - protocol: TCP
      port: 8081    # The port exposed by the service
      targetPort: 80    # The port on the container
```

This creates a **Service of type `ClusterIP`** that:

- Is named **`myapp-service`**.
- Selects all Pods labeled **`app: myapp`**.
- Forwards incoming traffic on **port `8081`** (within the cluster) to **port `80`** on the target Pods (where the `myapp` container is listening).

`kubectl apply -f clusteripservice.yml`

Kubernetes creates a virtual IP (**ClusterIP**) internally and maps:

`myapp-service:8081 → container on port 80`

You can see the service details by running:

```
kubectl get svc
```

You'll get an output like:

```
NAME            TYPE        CLUSTER-IP      PORT(S)

myapp-service   ClusterIP   10.96.0.12      8081/TCP
```

Then, from another Pod or instance **inside the same cluster**, you can test the service by curling:

```
curl 10.96.0.12:8081
```

How to create a Kubernetes NodePort Service

`nodeport.yml`

```
apiVersion: v1
kind: Service
metadata:
  name: myapp-service-np
spec:
  type: NodePort
  selector:
    app: myapp
  ports:
    - protocol: TCP
      port: 8081        # Service is accessible inside cluster at this port
      targetPort: 80    # Container inside the Pod listens on port 80
      nodePort: 32002   # Static port exposed on all nodes
```

`kubectl apply -f nodeport.yml`

This creates a Service named `myapp-service-np` and Maps:

- `myapp-service-np:8081` (within the cluster)
- `<Node IP>:32002` (externally) to
- Port `80` on the `myapp` container

You check the service:

`kubectl get svc`

You'll see output like:

NAME	TYPE	CLUSTER-IP	PORT(S)
myapp-service-np	NodePort	10.96.0.15	8081:32002/TCP

How to Access It:

From within the cluster:

`curl 10.96.0.15:8081`

From outside the cluster (external machine or browser):

`curl <Node IP>:32002`

Automated Rollouts and Rollbacks

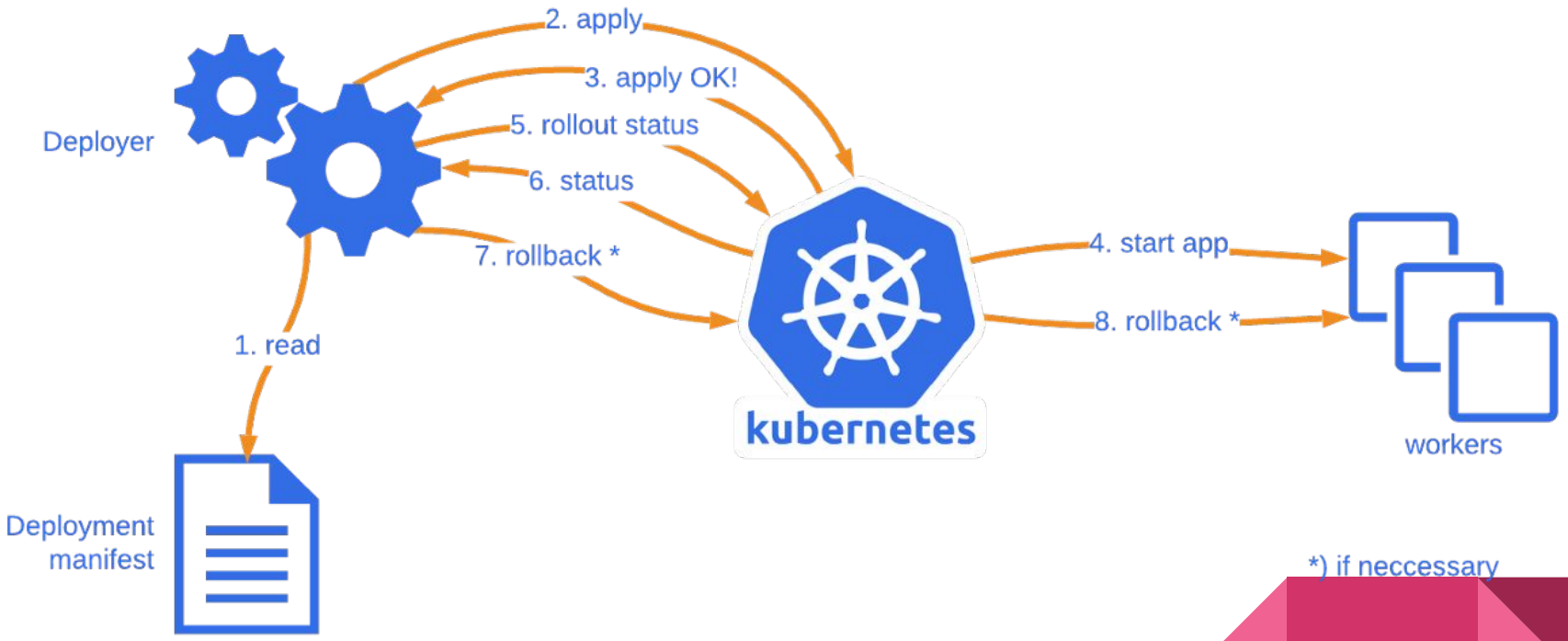
One of Kubernetes' most powerful features is its ability to **automatically update applications** while ensuring stability. If an update goes wrong, Kubernetes can **roll back to a previous stable version** without downtime.

How Rollouts Work

- Kubernetes applies updates **gradually** rather than replacing all containers at once.
- It ensures that a certain number of old instances remain active while **new instances are deployed**.
- This prevents sudden failures and ensures a **smooth transition**.

Rollback Mechanism

- If Kubernetes detects **errors** (e.g., failing health checks), it automatically **stops the rollout**.
- It can **revert to a previous stable version** to prevent service disruption.
- This is useful in case of **misconfigurations, bugs, or compatibility issues**.


2. apply
3. apply OK!
Deployer
5. rollout status
6. status
7. rollback *
4. start app
8. rollback *
1. read
kubernetes
workers
Deployment
manifest
*) if neccessary

Kubernetes supports horizontal scaling, automatically adjusting the number of Pods running your application in response to demand.

Types of Scaling in Kubernetes

Manual Scaling

- Admins can manually adjust the number of Pods using the command:

  ```
  kubectl scale deployment my-app --replicas=5
  ```

The command scales the `my-app` deployment to run exactly **5 replicas (Pods)**, ensuring the specified number of instances are active in the Kubernetes cluster.

This is useful for planned workload increases but requires human intervention.

Horizontal Pod Autoscaler (HPA)

- Kubernetes can **automatically adjust** the number of running Pods based on **CPU or memory usage**.
- It continuously monitors resource usage and **adds/removes Pods as needed**.
- Example command to enable HPA:

```
kubectl autoscale deployment my-app --cpu-percent=70 --min=2 --max=10
```

The command enables **Horizontal Pod Autoscaler (HPA)** for the `my-app` deployment, automatically adjusting the number of Pods between **2 and 10** based on CPU usage, scaling up when CPU exceeds **70% utilization**.

This ensures applications **stay responsive** under fluctuating workloads.

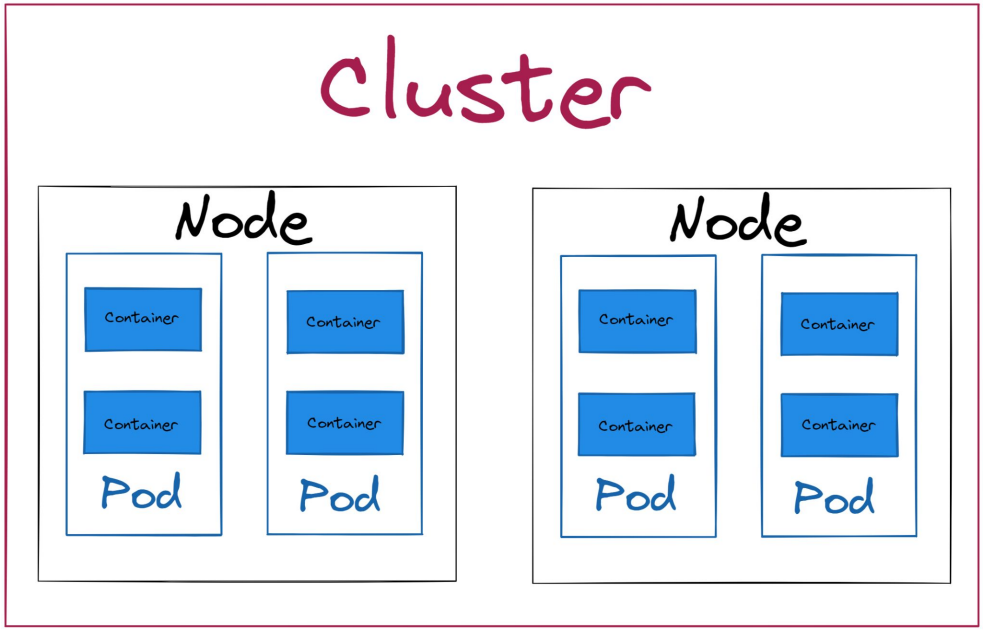
Designed for Extensibility

Kubernetes is built to **integrate easily with third-party tools and custom extensions** without modifying its core functionality.

- It allows developers to **add new features** to clusters without modifying Kubernetes source code.
- Supports **custom controllers, APIs, and plugins** for handling specific tasks.

Kubernetes Pod, Node & Cluster

Kubernetes (K8s) is an open-source container orchestration platform designed to **automate deployment, scaling, and management** of containerized applications. It follows a **distributed architecture**, where multiple machines, known as **nodes**, collaborate as part of a **cluster**. This architecture ensures that applications remain **highly available, fault-tolerant, and scalable**, even under heavy load or in the event of failures.

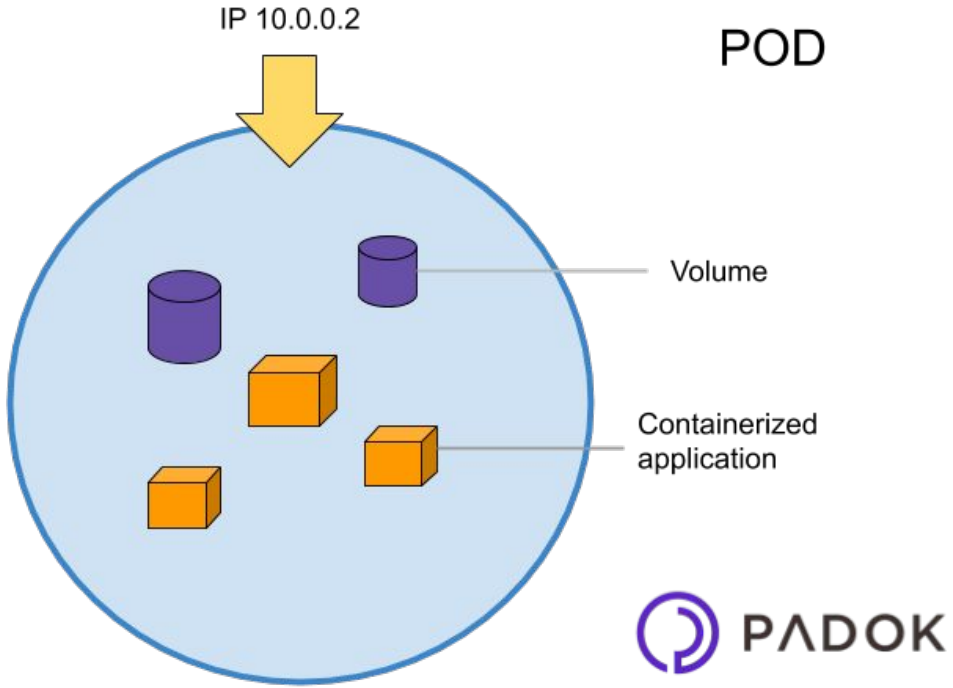


Pods

Containers in Kubernetes are always wrapped inside something called a **Pod**.

Whenever you want to run, stop, move, or scale your application, Kubernetes doesn't interact with the container directly—it works with the Pod.

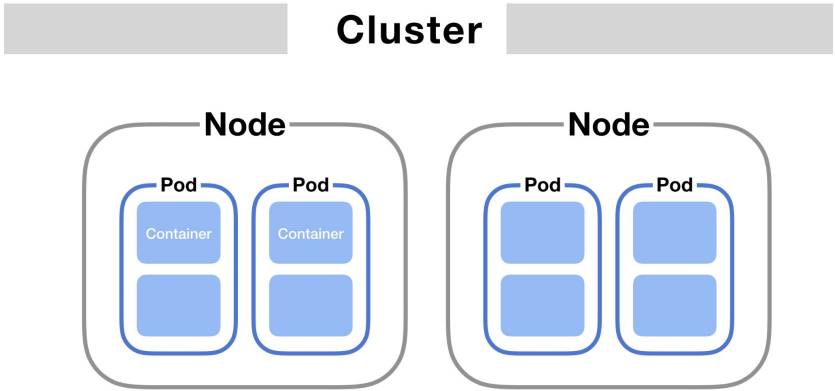
Even if you only need to run a single container, Kubernetes still places it inside a Pod. That's because the Pod is the fundamental unit that Kubernetes understands and manages.

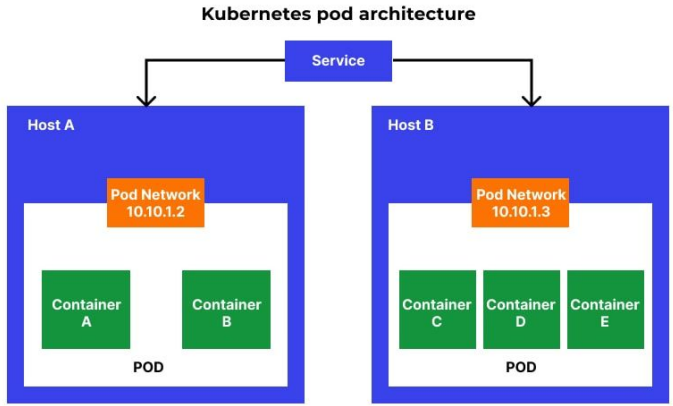


A **Pod** in Kubernetes contains everything needed to run one or more containers together in a coordinated environment. Specifically, a Pod includes:

1. **One or more containers** – These are the actual applications or services running (often just one, but can be more if they need to share resources).

2. **Shared network namespace** – All containers in a Pod **share the same IP address** and port space, which allows them to communicate with each other using `localhost`.
3. **Shared storage volumes** – Pods can define volumes that are mounted into all containers, allowing them to share files or configuration data.





Node

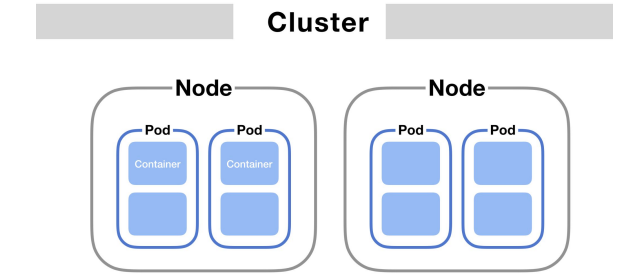
What is a Node in Kubernetes?

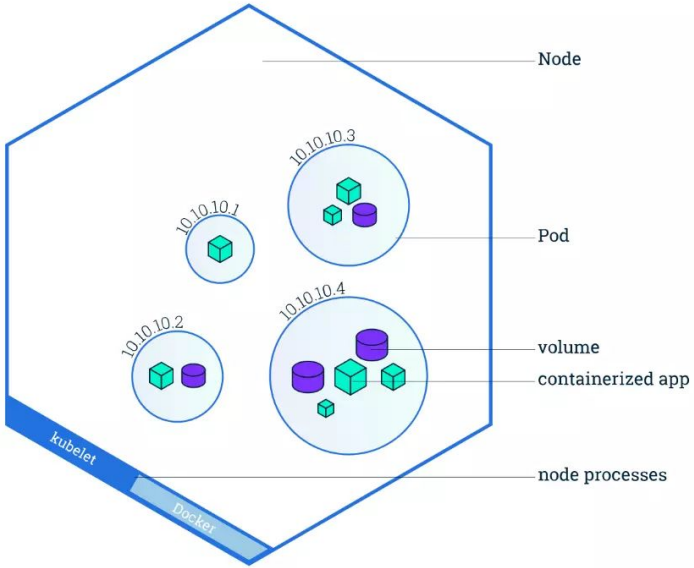
A **node** is a **worker machine** within the Kubernetes cluster. It can be a **physical server or a virtual machine**, and it is where Kubernetes **runs the application workloads** inside containers.

Each node is managed by the **control plane.**

Each node contains the components necessary to run Pods.

Pods are the smallest deployable units in Kubernetes. Kubernetes does not run containers on their own.



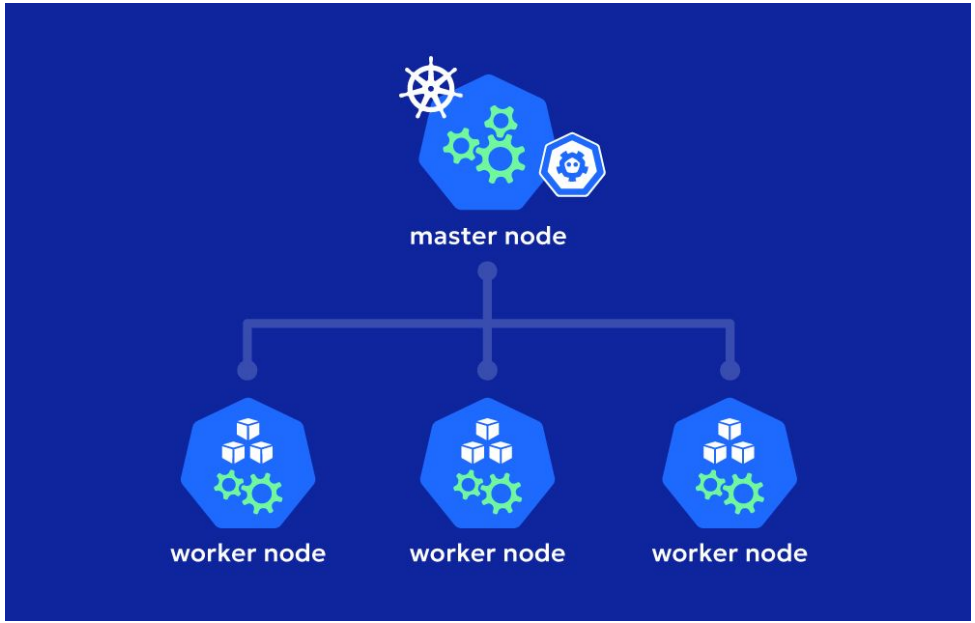


Types of nodes in Kubernetes

Kubernetes nodes are classified into two categories:

Master Node (Control Plane)

- The **brain** of the Kubernetes cluster that manages the entire system.
- Handles **scheduling, scaling, and health monitoring** of workloads.
- Runs essential components:
  - **API Server** – Acts as the entry point for all administrative commands.
  - **Controller Manager** – Ensures the cluster stays in the desired state.
  - **Scheduler** – Assigns workloads (Pods) to worker nodes.
  - **etcd (Key-Value Store)** – Stores cluster configuration data.



API Server – The **central management point** that:

- Receives and validates user/API requests.
- Communicates with other components.
- Stores configuration/state data in **etcd**.

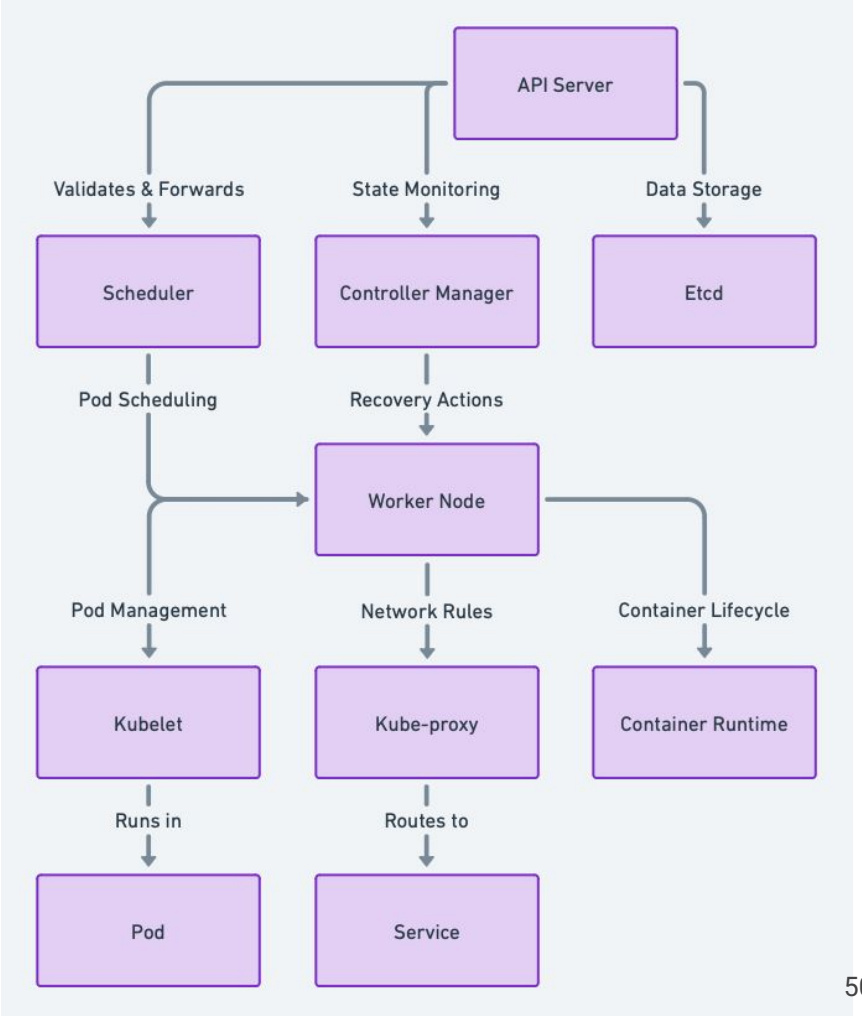
etcd – A **distributed key-value store** used for storing all cluster data (config, state, secrets, etc.).
Scheduler – Decides **which worker node** should run a new Pod based on resource availability.

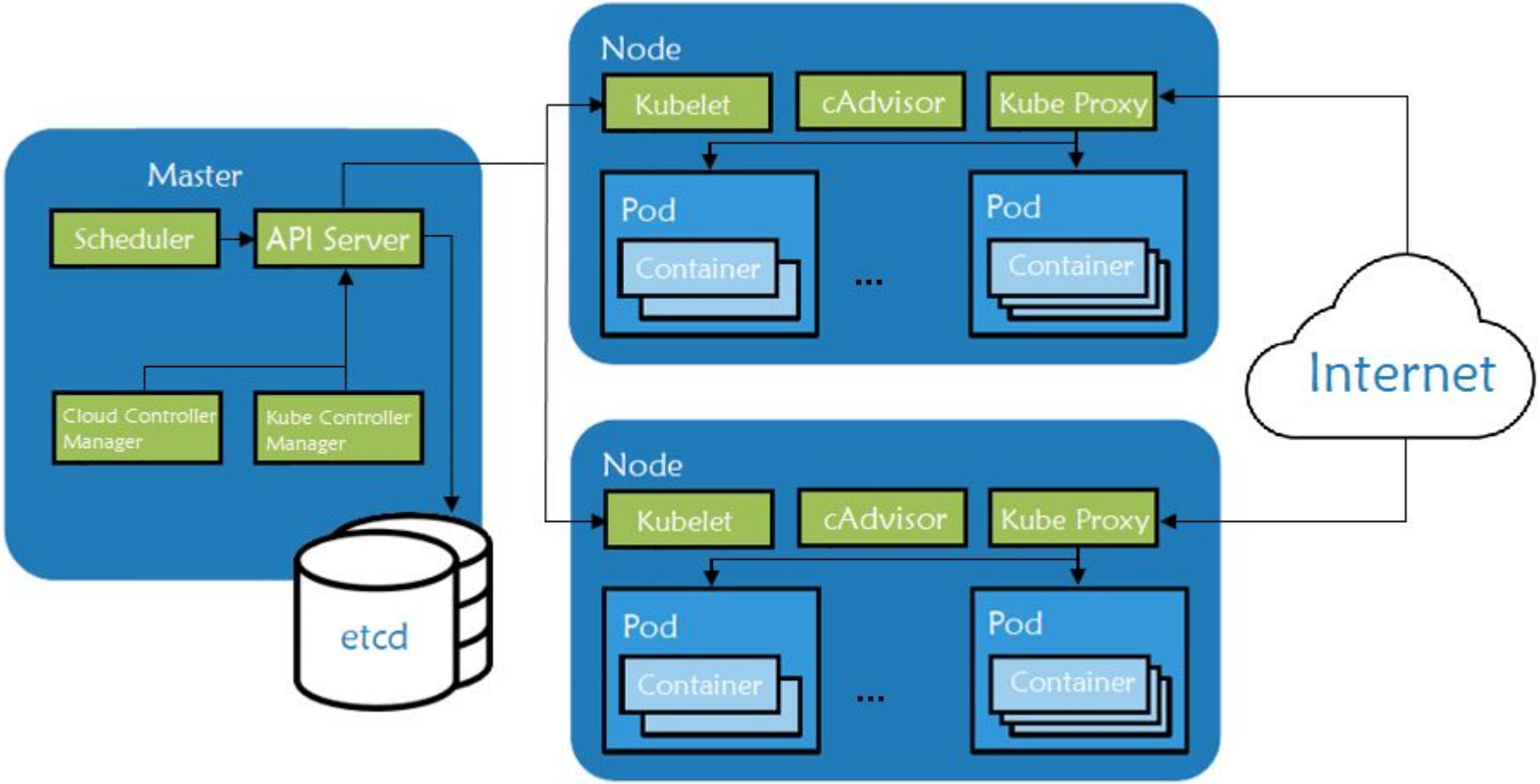
Controller Manager – Continuously **monitors cluster state** and performs **automated recovery actions** (e.g., rescheduling a failed pod).

Worker Node – Executes workloads (Pods). It contains:

- **Kubelet**: Manages pod lifecycle and communicates with the API Server.
- **Kube-proxy**: Manages **network rules**, routing traffic to the correct **Service**.
- **Container Runtime**: Responsible for starting/stopping containers (e.g., Docker, containerd).

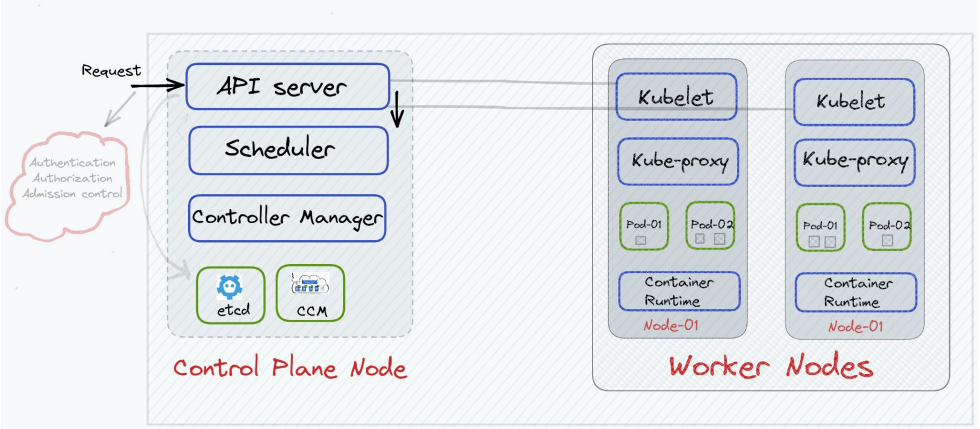
Pod – The **smallest deployable unit** in Kubernetes, managed by **Kubelet**.
Service – Provides a **stable networking endpoint** to access Pods, enabling communication inside or outside the cluster.



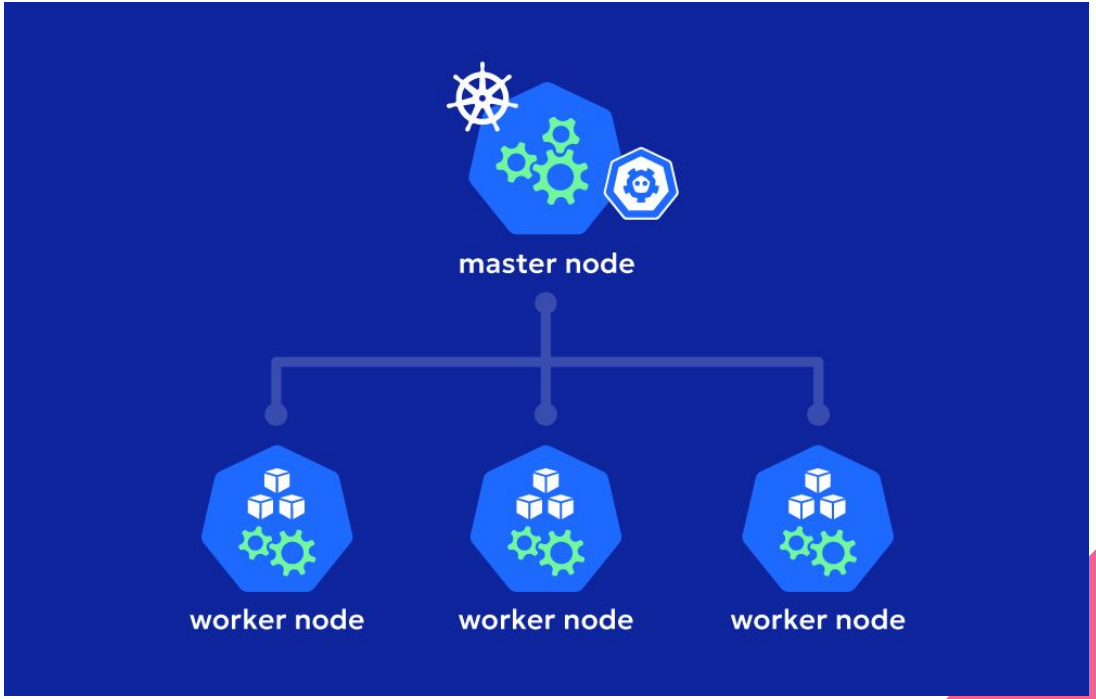

Node
Kubelet
cAdvisor
Kube Proxy
Pod
Container
...
Pod
Container
Master
Scheduler
API Server
Cloud Controller Manager
Kube Controller Manager
etcd
Internet
Node
Kubelet
cAdvisor
Kube Proxy
Pod
Container
...
Pod
Container

Worker Node

- Executes **application workloads** (containers) assigned by the master.
- Runs these key components:
  - **Kubelet** – Ensures containers are running correctly.
  - **Container Runtime** – Runs the actual containers (Docker, containerd, etc.).
  - **Kube Proxy** – Manages networking and load balancing.



Types of nodes in Kubernetes



Kubernetes Cluster

A **Kubernetes cluster** is a collection of **nodes** that work together to **deploy, run, and manage** containerized applications.

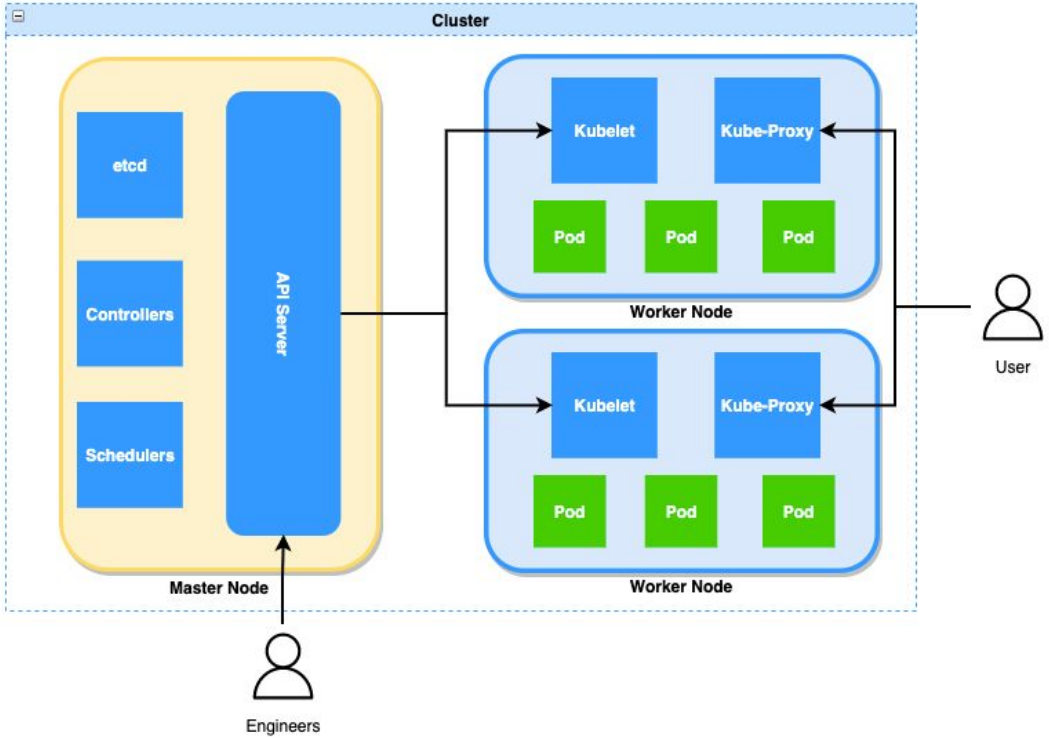
Characteristics of a Cluster

- **Scalability** – A cluster can be dynamically expanded by adding more nodes.
- **Fault Tolerance** – If a node fails, Kubernetes reschedules workloads onto healthy nodes.
- **Networking** – All nodes are interconnected to enable smooth communication.

How Kubernetes Manages the Cluster?

1. The **master node** continuously **monitors and schedules** workloads across **worker nodes**.
2. Applications are deployed as **Pods**, which run on worker nodes.
3. The cluster ensures **automatic failover** if a node crashes.
4. Kubernetes networking allows nodes to **communicate seamlessly** without manual intervention.

Kubernetes Cluster

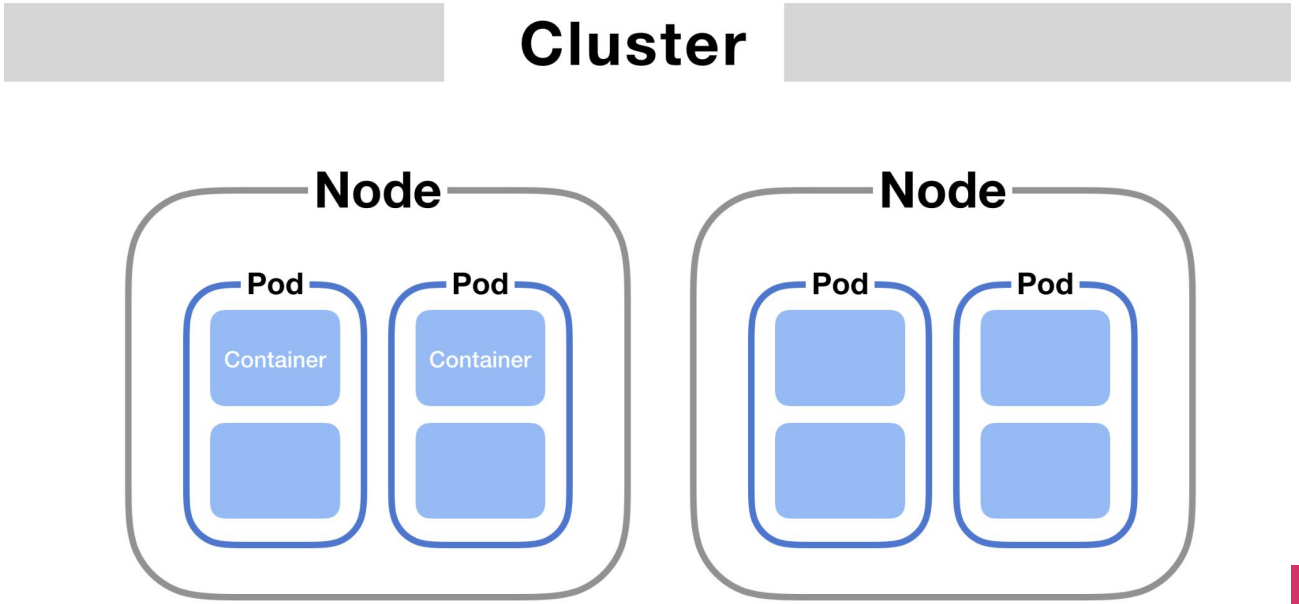


Source: https://www.nccgroup.com/us/research-blog/detection-engineering-for-kubernetes-clusters/

Pod, Node, Cluster Summary

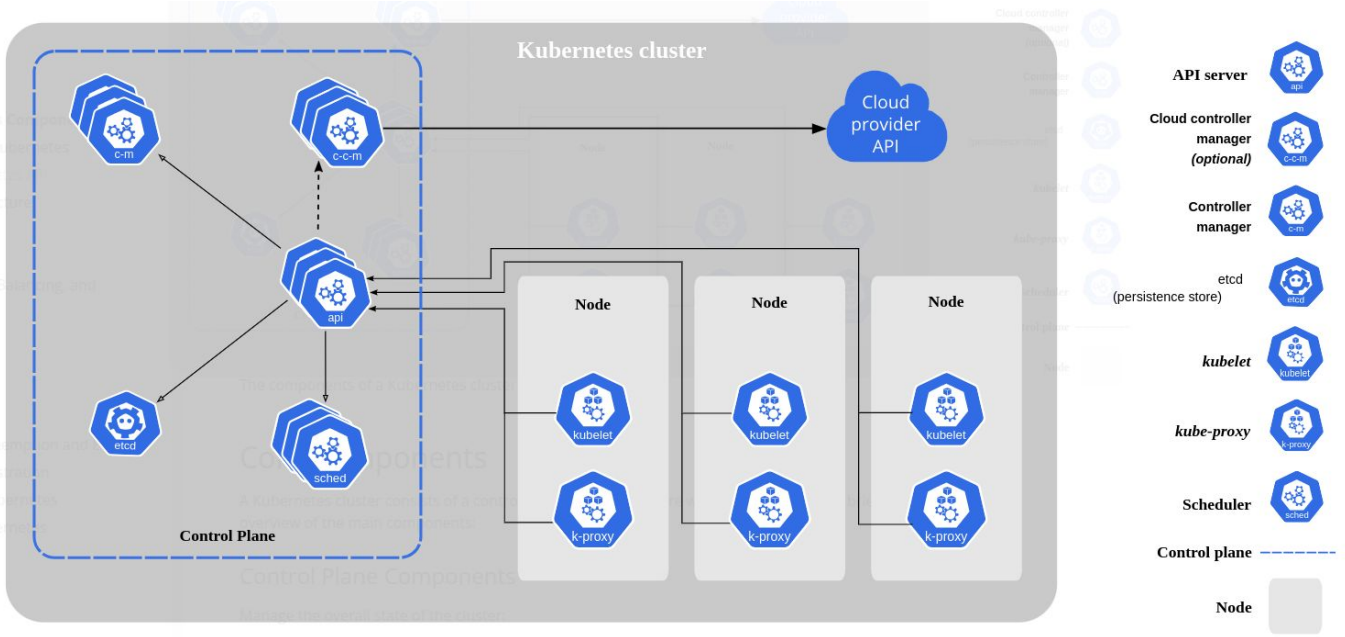
	Pod	**Node**	**Cluster**
Description	The smallest deployable unit in a Kubernetes cluster	A physical or virtual machine	A grouping of multiple nodes in a Kubernetes environment
Role	Isolates containers from underlying servers to boost portability. Provides the resources and instructions for how to run containers optimally.	Provides the compute resources (CPU, volumes, etc) to run containerized apps	Has the control plane to orchestrate containerized apps through nodes and pods
What it hosts	Application containers, supporting volumes, and similar IP addresses for logically similar containers	Pods with application containers inside them, kubelet	Nodes containing the pods that host the application containers, control plane, kube-proxy, etc

Pod, Node, Cluster



Source: https://matthewpalmer.net/kubernetes-app-developer/articles/kubernetes-networking-guide-beginners.html

Kubernetes Components



Source: https://kubernetes.io/docs/concepts/overview/components/

Kubernetes Networking: Pods and Communication

Kubernetes (**K8s**) follows a unique networking model that ensures **seamless communication** between containers and services running in a cluster. The fundamental unit of networking in Kubernetes is a **Pod**.

A **Pod** is the **smallest deployable unit** in Kubernetes that encapsulates one or more **containerized applications** running in the same environment.

Kubernetes Networking: Pods and Communication

Characteristics of a Pod

- Each **Pod** is assigned a **unique cluster-wide IP address**, allowing it to communicate with other Pods in the cluster.
- It contains one or more **containers** that share the same **network namespace**, meaning:
  - All containers inside the same Pod can communicate via **localhost**.
  - They **share storage volumes** and network interfaces.
- **Pods are ephemeral**, meaning they can be restarted or rescheduled to different nodes if needed.

Importance of Pods

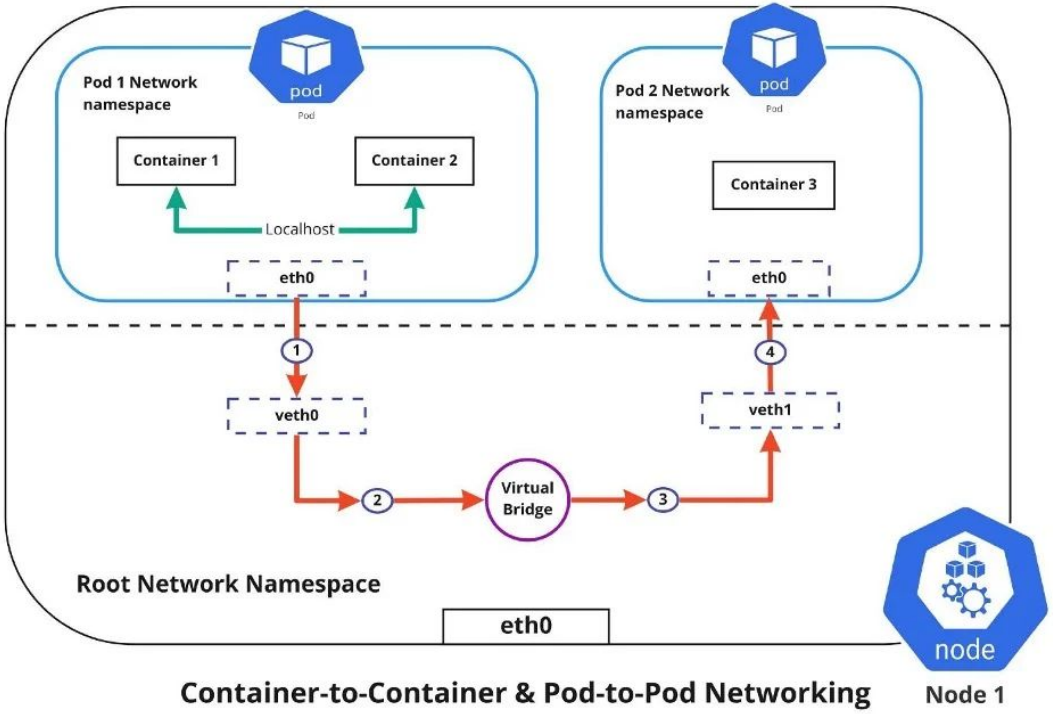
- Pods ensure that **related containers run together** and can communicate efficiently.
- Pods provide **resource sharing** (network, storage) among tightly coupled applications.
- Pods simplify **scaling and orchestration** of microservices.

How Kubernetes Handles Pod Networking

Kubernetes enforces a **flat networking model**, where all Pods in a cluster can communicate **without NAT (Network Address Translation)**. This means:

- Every Pod has a **unique IP**, ensuring **direct communication**.
- Pods can communicate **across nodes** without additional routing configurations.
- Kubernetes provides **network policies** to control **which Pods can talk to each other**.

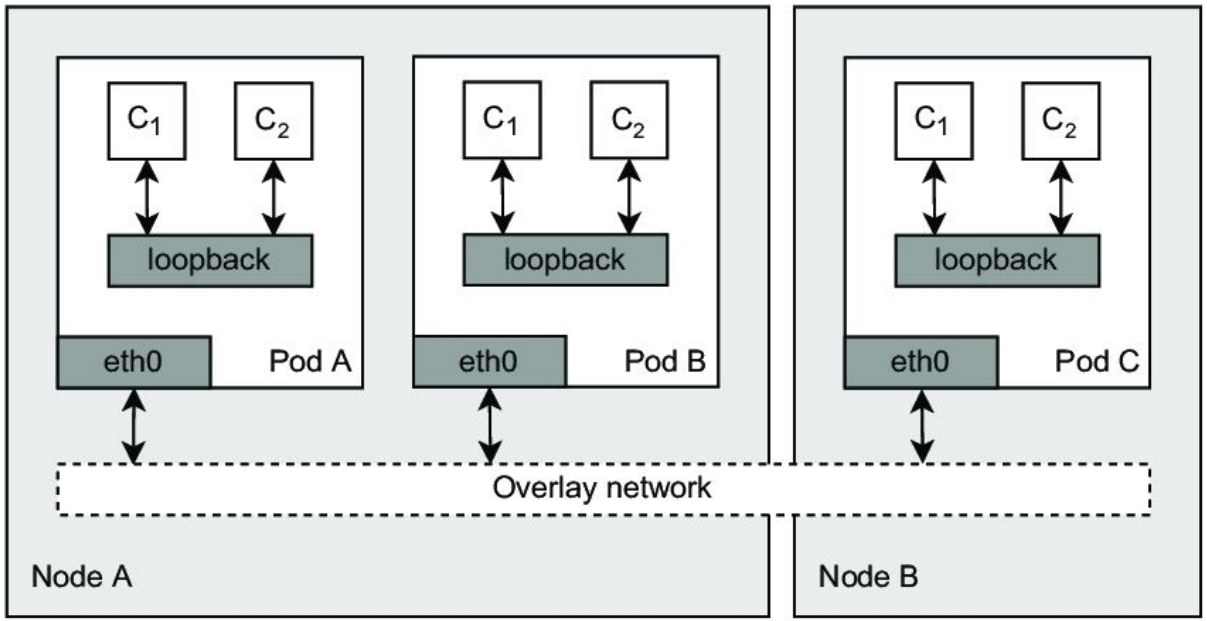
How Kubernetes Handles Pod Networking



Source: https://opensource.com/article/22/6/kubernetes-networking-fundamentals

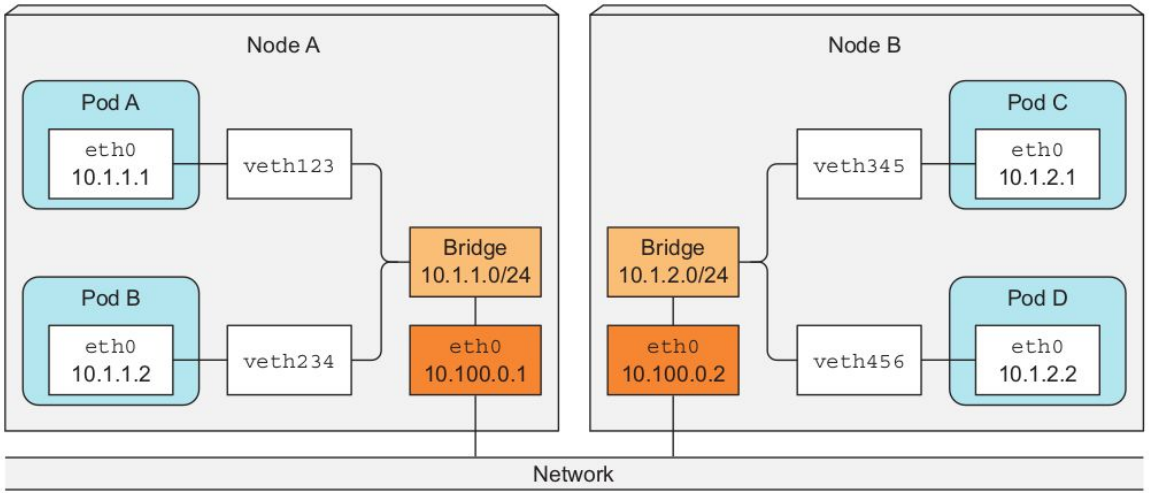
How Pods Communicate?

- **Intra-Pod Communication**
  - Containers within the same Pod use **localhost** to communicate (loopback).
  - They share the same **network namespace**, making inter-container communication fast and efficient.



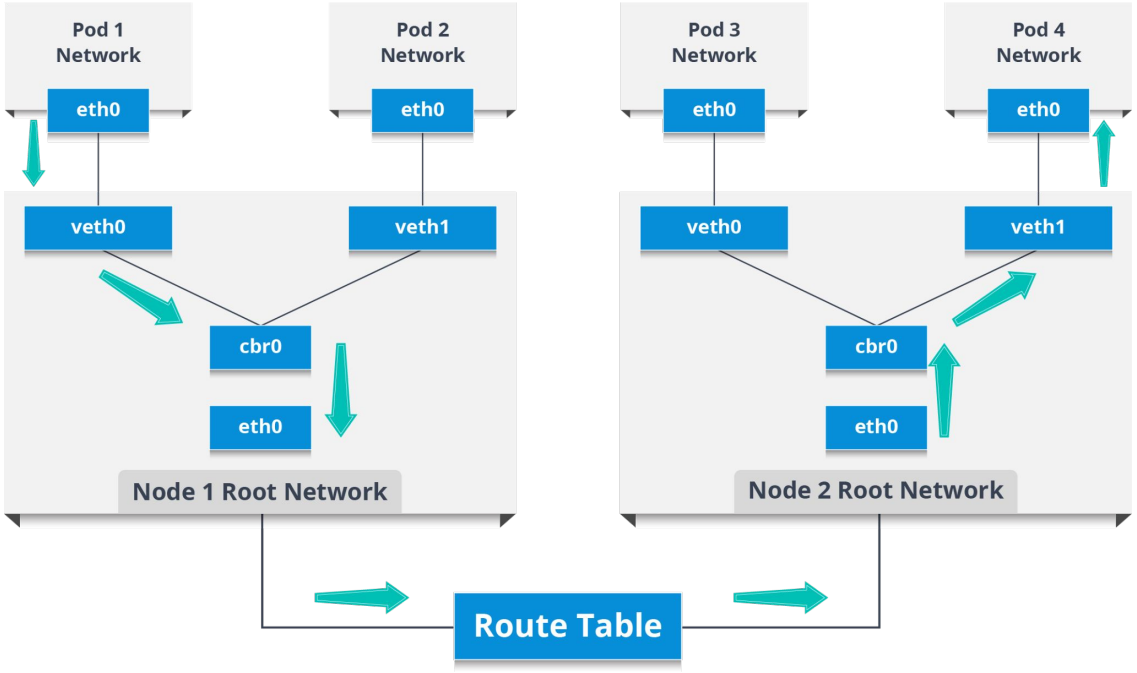
Source: https://www.researchgate.net/figure/A-sketch-of-the-networking-in-Kubernetes-intra-pod-communications-exploit-the-loopback_fig3_371922884

- **Inter-Pod Communication (Pod-to-Pod)**
  - Pods in different nodes communicate using **their assigned IPs**.
  - The **Pod network** ensures seamless connectivity across all nodes in the cluster.
  - Kubernetes **resolves DNS names** to Pod IPs dynamically.



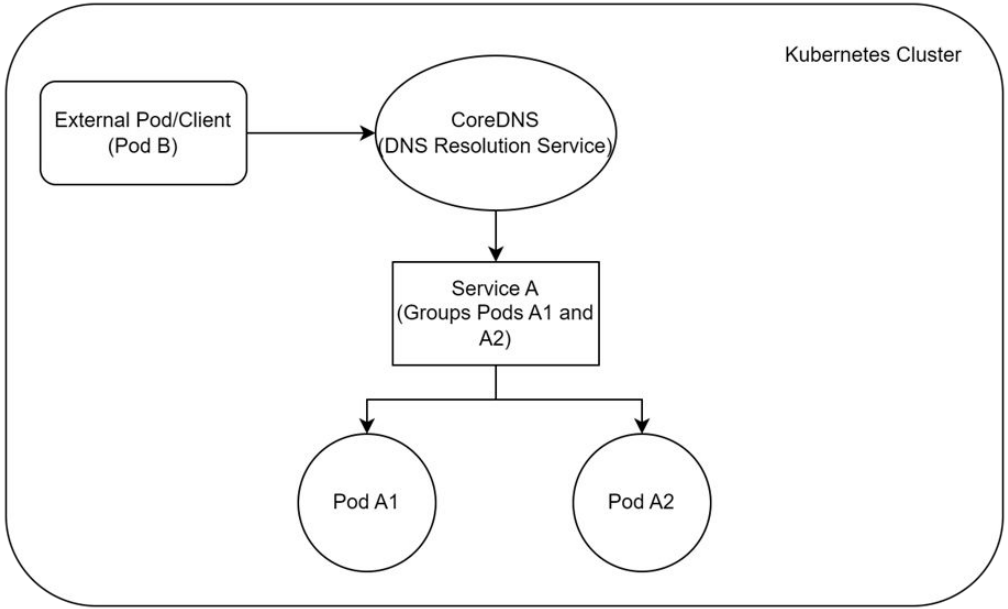
Source: https://stackoverflow.com/questions/58859875/kubernetes-pod-communication-across-nodes-how-does-it-work

- **Inter-Pod Communication (Pod-to-Pod)**
  - Pods in different nodes communicate using **their assigned IPs**.
  - The **Pod network** ensures seamless connectivity across all nodes in the cluster.
  - Kubernetes **resolves DNS names** to Pod IPs dynamically.



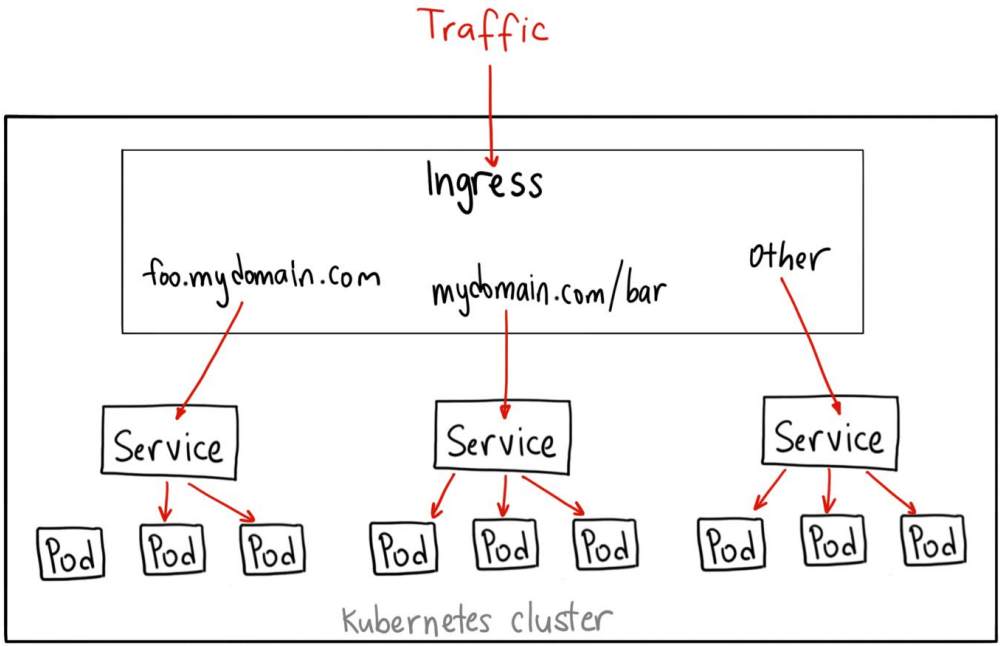
Source: https://dzone.com/articles/kubernetes-networking

- **Pod-to-Service Communication**
  - Services provide a **stable endpoint** for Pods, allowing clients to reach applications even if individual Pods restart or change IPs.
  - Kubernetes **Service Discovery** assigns a **DNS name** to each service for easy access.



Source: https://faun.pub/resolving-service-discovery-problems-in-kubernetes-60773a12ce77

- **Pod-to-Service Communication**
  - Services provide a **stable endpoint** for Pods, allowing clients to reach applications even if individual Pods restart or change IPs.
  - Kubernetes **Service Discovery** assigns a **DNS name** to each service for easy access.



Source: https://rtfm.co.ua/en/kubernetes-clusterip-vs-nodeport-vs-loadbalancer-services-and-ingress-an-overview-with-examples/

Understanding Kubernetes Services

Kubernetes provides a **dynamic networking model**, where applications run in **Pods** that can be restarted, rescheduled, or replaced. Since Pod IPs are **ephemeral** (temporary), direct communication between them is **unreliable**. This is where **Kubernetes Services** come in.

What is a Kubernetes Service?

A **Service** in Kubernetes is a logical abstraction that **exposes and manages network access** to a set of **Pods**. It provides a **stable endpoint (IP or DNS)** that remains constant even if the underlying Pods change.

Key Functions of a Service

- **Ensures stable communication** between Pods despite dynamic changes.
- **Load balances** traffic across multiple Pod instances.
- **Facilitates external access** to applications running inside the cluster.

Types of Kubernetes Services

Kubernetes supports different types of services, depending on how Pods need to be accessed:

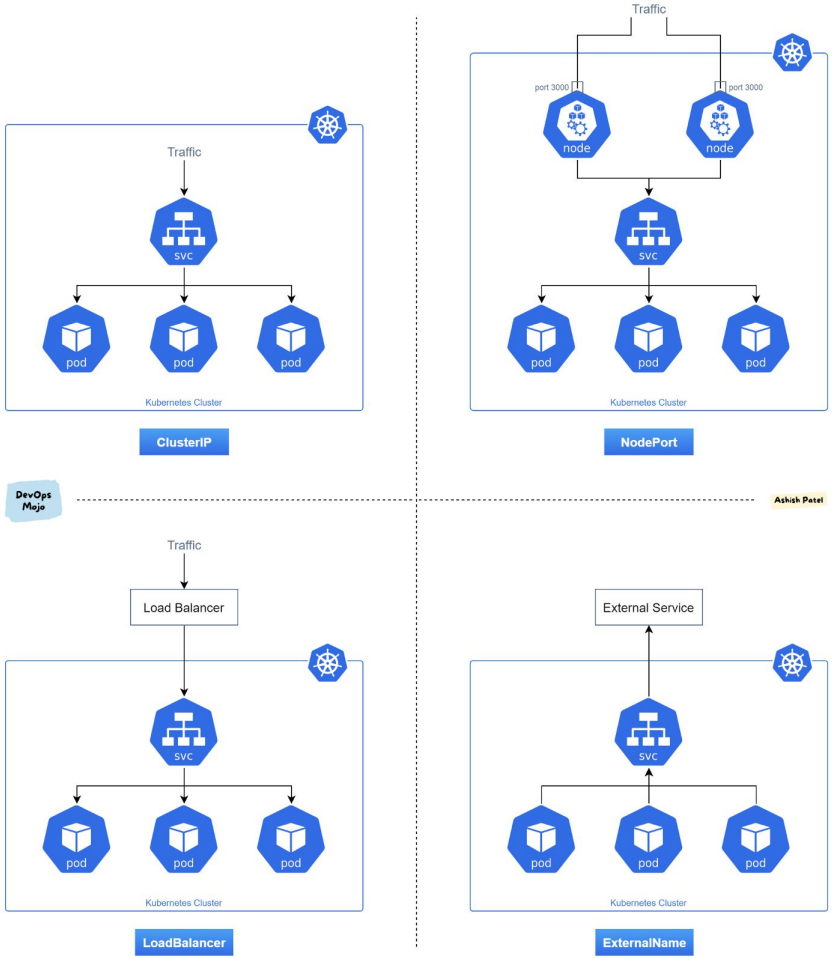
1. **ClusterIP (Default)**
   - Exposes the service internally within the cluster.
   - Best for **Pod-to-Pod communication** inside Kubernetes.
2. **NodePort**
   - Exposes the service externally on a fixed port of each node.
   - Useful for **testing and basic external access**.
3. **Load Balancer**
   - Integrates with cloud providers (AWS, GCP, Azure) to expose the service externally via a **managed load balancer**.
   - Best for **production deployments** requiring high availability.
4. **ExternalName**
   - Maps the service to an **external DNS name** instead of a Pod IP.
   - Used for **redirecting traffic to services outside Kubernetes**.

ClusterIP: The default service type, exposing the service only within the cluster for internal communication.

NodePort: Exposes the service on a static port on each node, allowing external access via `<NodeIP>:<Port>`.

LoadBalancer: Integrates with cloud provider load balancers to expose the service externally with automatic traffic distribution.

ExternalName: Maps the service to an external DNS name, redirecting requests to an external service outside the cluster.



Kubernetes Service API and YAML Definition

A **Service is defined using YAML configuration**, specifying how it should route traffic.

apiVersion: Defines which Kubernetes API version to use.

kind: Specifies that this resource is a **Service**.

metadata: Provides a name for the Service (`my-service`).

spec:

- **selector:** Matches Pods labeled `MyApp`, ensuring traffic is routed correctly.
- **ports:**
  - **protocol:** Defines TCP as the communication protocol.
  - **port (80):** The Service's exposed port.
  - **targetPort (9376):** The actual port inside the Pod where the application runs.

Basic YAML Structure of a Service

```
apiVersion: v1
kind: Service
metadata:
    name: my-service
spec:
    selector:
        app.kubernetes.io/name: MyApp
        ports:
            protocol: TCP
            port: 80
            targetPort: 9376
```

Summary of Kubernetes Services

Service Discovery:

- Kubernetes provides **built-in DNS resolution**, allowing Pods to find services using a name like `my-service.default.svc.cluster.local`.
- This eliminates the need for hardcoded IPs.

Traffic Routing & Load Balancing:

- Services distribute incoming requests across **multiple Pods** to balance the workload.
- If a Pod crashes, Kubernetes automatically redirects traffic to healthy Pods.

Types of Kubernetes Services:

- **ClusterIP (Default):** Accessible only within the cluster, used for internal communication.
- **NodePort:** Exposes the service on a static port on every node for external access.
- **LoadBalancer:** Integrates with cloud providers to expose the service via a managed load balancer.
- **ExternalName:** Maps the service to an external DNS name.

Kubectl Commands: Managing Kubernetes

Kubernetes provides a **command-line tool** called `kubectl` to interact with the cluster, manage resources, and monitor the system.

Checking Node Status with `kubectl get nodes`

- The **`kubectl get nodes`** command lists all the nodes in a cluster along with their **status, roles, age, and version**.
- It is crucial for **monitoring cluster health** and identifying whether a node is functioning correctly.

Example Output

NAME	STATUS	ROLES	AGE	VERSION
docker-desktop	Ready	control-plane	19h	v1.30.2

NAME: The node's name (e.g., `docker-desktop`).

STATUS: Indicates whether the node is operational.

ROLES: Defines if the node is a **control-plane (master) or worker node**.

AGE: Shows how long the node has been running.

VERSION: Displays the Kubernetes version running on the node.

Creating a Deployment

The following command creates a Deployment named **`hello-node`** and runs a container using the image **`registry.k8s.io/e2e-test-images/agnhost:2.39`**:

kubectl create deployment hello-node --image=registry.k8s.io/e2e-test-images/agnhost:2.39 --
/agnhost netexec --http-port=8080

Breakdown

- **kubectl create deployment hello-node** → Creates a Deployment named `hello-node`.
- **--image=registry.k8s.io/e2e-test-images/agnhost:2.39** → Specifies the container image to use.
- **/agnhost netexec --http-port=8080** → Runs the container with the `netexec` command, exposing port `8080`.

Checking Pod Status

The `kubectl get pods` command lists all the running Pods in the cluster, showing their status and availability.

This is useful for verifying if applications are running correctly.

Example Command & Output

```
kubectl get pods
```

NAME	READY	STATUS	RESTARTS	AGE
hello-node-55fdcd95bf-22mh2	1/1	Running	0	18h

Understanding Output:

- **NAME:** Unique name of the Pod (`hello-node-55fdcd95bf-22mh2`).
- **READY:** Shows the number of running containers in the Pod (`1/1` means one container is running successfully).
- **STATUS:** Current state of the Pod (`Running` means it is active).
- **RESTARTS:** Number of times the Pod has restarted (`0` means it has not restarted).
- **AGE:** How long the Pod has been running (`18h` means 18 hours).

Viewing Kubernetes Events

`kubectl get events` provides a chronological list of cluster events, helping debug issues and understand how workloads are scheduled.

It shows details like **Pod creation, image pulling, container start, and scaling actions**.

```
kubectl get events
```

LAST SEEN	TYPE	REASON	OBJECT	MESSAGE
34s	Normal	Scheduled	pod/hello-node-55fdcd95bf-22mh2	Successfully assigned to node
33s	Normal	Pulling	pod/hello-node-55fdcd95bf-22mh2	Pulling image "registry.k8s.io/e2e-test-images/agnhost:2.39"
30s	Normal	Pulled	pod/hello-node-55fdcd95bf-22mh2	Successfully pulled image
28s	Normal	Created	pod/hello-node-55fdcd95bf-22mh2	Created container agnhost
27s	Normal	Started	pod/hello-node-55fdcd95bf-22mh2	Started container agnhost

Understanding Output

LAST SEEN: Time since the event occurred.

TYPE: `Normal` (expected behavior) or `Warning` (potential issue).

REASON: What triggered the event (`Scheduled`, `Pulling`, `Created`, `Started`).

OBJECT: The resource affected (Pod `hello-node-55fdcd95bf-22mh2`).

MESSAGE: Describes what happened (e.g., Pod successfully assigned, image pulled).

When managing applications in Kubernetes, it is essential to **monitor logs** and **inspect services** to ensure smooth operation and troubleshooting. `kubectl` provides commands like `kubectl logs` and `kubectl get services` to analyze application behavior and networking.

```
kubectl logs <pod-name>
```

Example output

```
kubectl logs hello-node-55fdcd95bf-22mh2
```

```
I1112 14:20:48.871286 1 log.go:195] Started HTTP server on port 8080

I1112 14:20:48.872371 1 log.go:195] Started UDP server on port 8081
```

Viewing Services

The `kubectl get services` command lists all services running in the cluster.

```
kubectl get svc
```

NAME	TYPE	CLUSTER-IP	EXTERNAL-IP	PORT(S)	AGE
kubernetes	ClusterIP	10.96.0.1	<none>	443/TCP	6d23h
nginx-service-np	NodePort	10.108.167.246	<none>	8081:30107/TCP	83m

EXTERNAL-IP: If the service is externally accessible (e.g., via a LoadBalancer).

PORT(S): Ports used by the service (`8081:30107/TCP` means traffic on port `8081` inside the cluster is mapped to `30107` externally).

ClusterIP (Internal Networking in Kubernetes)

In Kubernetes, **ClusterIP** is the **default service type** that allows internal communication between Pods within the cluster. It **exposes an internal IP** but does not allow external access.

- A **ClusterIP** service provides **internal communication** between different components (Pods) inside the cluster.
- It **automatically assigns an internal IP address** that is accessible only within the cluster.
- This service is mainly used for **intra-cluster communication**, such as backend services interacting with databases.

Characteristics of ClusterIP

- **Default Service Type** – When no service type is specified, Kubernetes assigns `ClusterIP`.
- **Internal-Only Communication** – The service is **not accessible from outside the cluster**.
- **Pods Access via Service Name** – Other Pods communicate with the service using **DNS resolution (`service-name.namespace.svc.cluster.local`)** instead of direct Pod IPs.
- **Load Balancing** – Even if multiple Pods back the service, Kubernetes distributes traffic between them.

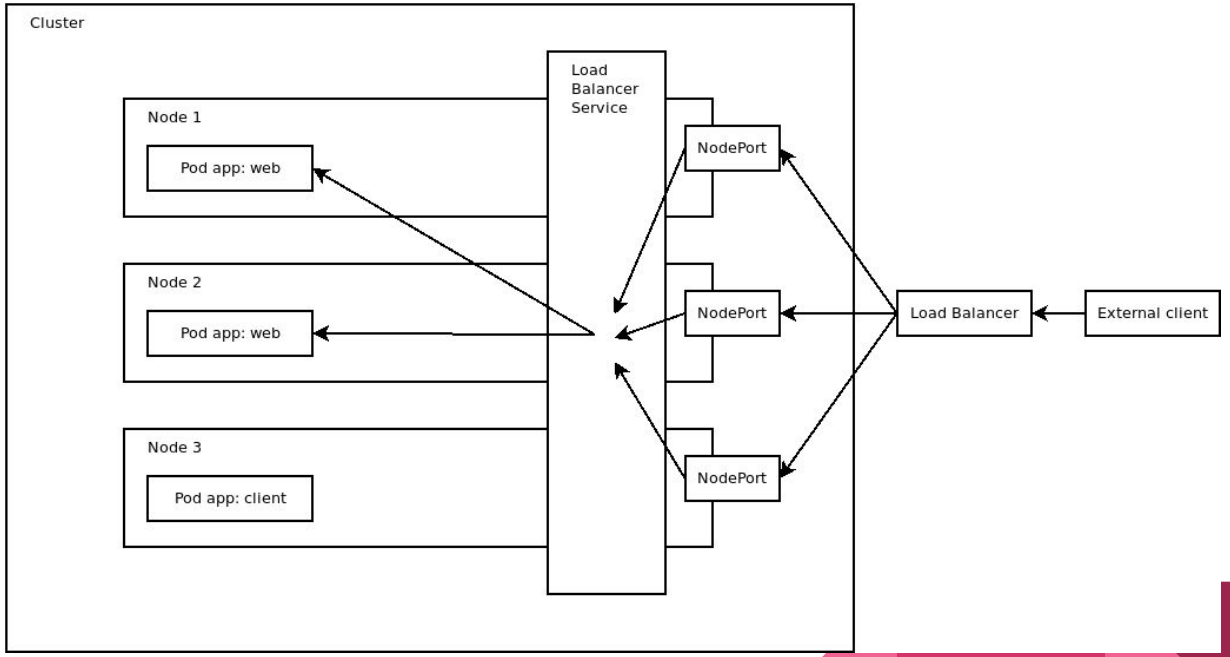
Example YAML for a ClusterIP Service

```
apiVersion: v1
kind: Service
metadata:
  name: my-clusterip-service
spec:
  selector:
    app: my-app
  ports:
    - protocol: TCP
      port: 80      # Service port
      targetPort: 9376  # Container's port
```

Kubernetes Service Type: NodePort

A **NodePort** service in Kubernetes allows external access to applications running inside the cluster by assigning a static port (30,000–32,767) on every node in the cluster. It forwards incoming traffic to the appropriate Pods.

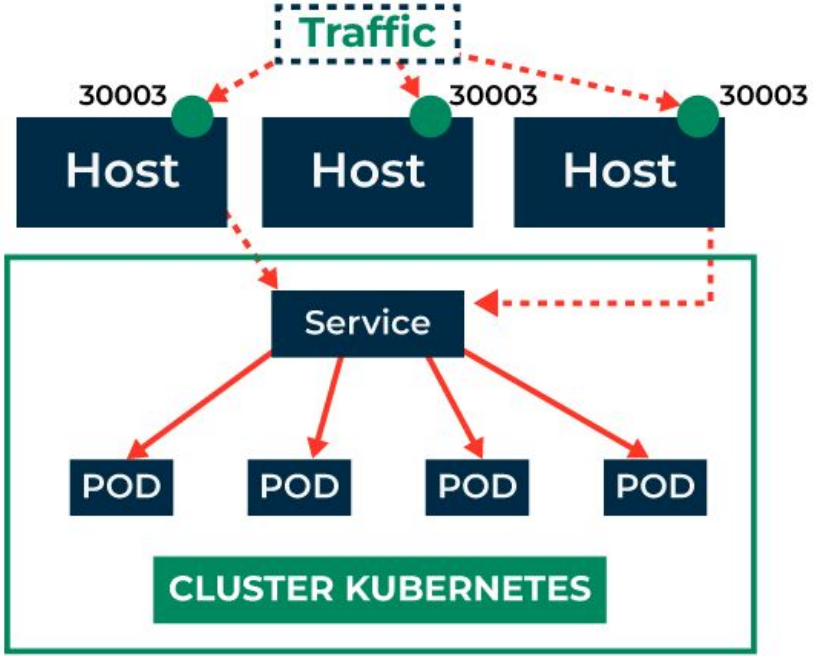
- It **exposes a service externally** by binding it to a static port on each cluster node.
- **Traffic enters the cluster** through this port and gets forwarded to the appropriate **service and Pods**.
- This allows **external clients** to reach Kubernetes applications **without a load balancer**.



Source: https://octopus.com/blog/difference-clusterip-nodeport-loadbalancer-kubernetes

Characteristics of NodePort

- **Allows External Access** – Services can be accessed from outside the cluster via `NodeIP:NodePort`.
- **Each Service Gets One Port** – The port is chosen from the **range 30,000–32,767**.
- **Traffic is Distributed** – Requests entering through the NodePort are **load-balanced across all Pods**.
- **Works Without a Load Balancer** – Unlike LoadBalancer services, NodePort **does not require cloud provider integration**.



Source: https://www.geeksforgeeks.org/kubernetes-nodeport-service/

Example YAML for a NodePort Service

Internal Pods listen on port `9376`.

The service **routes traffic from `port 80` to the Pods**.

The **service is externally accessible** via **`<NodeIP>:31000`**.

```
apiVersion: v1
kind: Service
metadata:
  name: my-nodeport-service
spec:
  type: NodePort
  selector:
    app: my-app
  ports:
    - protocol: TCP
      port: 80       # Internal service port
      targetPort: 9376  # Port on the Pod
      nodePort: 31000   # Exposed static port on each Node
```

More about NodePort

How NodePort Works

1. **User sends a request** to `<NodeIP>:<NodePort>`, e.g., `192.168.1.100:31000`.
2. Kubernetes routes the traffic to the **NodePort service**.
3. The **service forwards the request** to one of the matching **Pods**.
4. The **Pod processes the request** and returns the response.

When to Use NodePort?

- When an application **must be accessed externally** without a LoadBalancer.
- For **on-premise Kubernetes clusters** where cloud-based load balancers are unavailable.
- When developing/testing services that **need external access** without extra networking configurations.

Limitations of NodePort

- **Limited Port Range** – You can only use **30,000–32,767**, which might conflict with existing services.
- **Not Ideal for Large-Scale Apps** – Works best for small applications; **not optimized for production**.
- **Requires Node IP Knowledge** – Clients must know the node's **IP address** to access services.

LoadBalancer Service (External Traffic Management)

A **LoadBalancer** service in Kubernetes provides a way to expose applications externally by automatically provisioning a **public IP address** and distributing traffic among the backend **Pods**.

- It is the most **convenient way to expose a Kubernetes service externally** in cloud environments.
- It **creates a public-facing IP address** and routes external traffic to the appropriate **Pods**.
- Load balancing is handled **automatically**, ensuring even traffic distribution across running instances.

Characteristics of LoadBalancer

- **External Accessibility** – The service is assigned a **public IP address**, making it accessible to external clients.
  Built-in Load Balancing – Spreads incoming traffic evenly across available **Pods**.
  Cloud-Provider Dependent – Works with **cloud platforms** like AWS, GCP, and Azure, which provide **native load balancers**.
  Ideal for Web Applications – Best suited for **HTTP(S) services, APIs, and internet-facing applications**.

Example YAML for a LoadBalancer Service

- The service **receives traffic on port `80`** and routes it to **Pods running on port `9376`**.
- A **public IP is assigned automatically** by the cloud provider.
- The service ensures **even traffic distribution** across all **healthy Pods**.

```
apiVersion: v1
kind: Service
metadata:
  name: my-loadbalancer-service
spec:
  type: LoadBalancer
  selector:
    app: my-app
  ports:
   - protocol: TCP
     port: 80       # External port exposed to users
     targetPort: 9376  # Port on the Pod handling the request
```