What are the core components of a Kubernetes cluster (e.g., master, node, etcd, kube-apiserver)? Briefly explain their roles

A **Kubernetes cluster** consists of several core components, categorized into **Control Plane** and **Worker Nodes**:

1. Control Plane (Master Components)

Manages the cluster and ensures the desired state is maintained.

- **etcd** → A distributed key-value store that holds the entire cluster state.
- **kube-apiserver** → The main entry point for all Kubernetes operations, exposing the Kubernetes API.
- **kube-controller-manager** → Manages controllers (e.g., node, deployment, replication controllers) to ensure desired state.
- **kube-scheduler** → Assigns new Pods to appropriate nodes based on resource availability and constraints.

2. Worker Nodes (Execution Environment)

Run applications inside containers.

- **kubelet** → Agent running on each node, responsible for managing Pods and communicating with the control plane.
- **kube-proxy** → Manages networking and load balancing for services.
- **Container Runtime** → Runs containers (e.g., Docker, containerd, or CRI-O).

Each **worker node** is managed by the control plane and runs Pods that contain application workloads.

What is a Pod in Kubernetes?

A **Pod** is the **smallest deployable unit** in Kubernetes, representing one or more **tightly coupled containers** that share **networking and storage**. Pods ensure that related containers work together as a single unit.

Pod vs. Docker Container:

Feature	Kubernetes Pod	Docker Container
Definition	A group of one or more containers running together	A single, isolated process inside a container runtime
Networking	All containers in a Pod share the same **IP address** and **network namespace**	Each container has its own isolated network
Storage	Containers in a Pod can share **volumes** for data persistence	Storage is isolated unless manually shared via volumes
Scaling	Kubernetes scales Pods, not individual containers	Containers are managed manually or via Docker Compose
Communication	Inter-container communication in a Pod is **localhost**	Containers must communicate over a network

Explain the purpose of a Kubernetes deployment. How do deployments ensure high availability of applications?

Purpose of a Kubernetes Deployment

A **Deployment** in Kubernetes automates the management of application updates, scaling, and rollback. It ensures that a specified number of **identical** Pods are running at all times.

How Deployments Ensure High Availability

1. **Replica Management** → Deployments maintain multiple **replicas** of a Pod, ensuring the application is always available.
2. **Self-Healing** → If a Pod fails, the Deployment automatically creates a new one to maintain the desired state.
3. **Rolling Updates** → Deployments **gradually replace** old Pods with new ones, preventing downtime.
4. **Rollback Support** → If an update fails, Kubernetes can **revert** to the previous stable version.
5. **Load Balancing** → Works with **Services** to distribute traffic among running Pods for optimal performance.

What are the different types of services in Kubernetes (e.g., ClusterIP, NodePort, LoadBalancer)? When would you use each type?

Types of Kubernetes Services & When to Use Them

Kubernetes **Services** expose **Pods** to the network, enabling communication inside or outside the cluster.

Service Type	Use Case	Description
ClusterIP *(Default)*	Internal communication	Exposes the service **only within the cluster**. Used for microservices that communicate internally.
NodePort	External access on a static port	Exposes the service on **each node's IP** at a fixed **port (30000–32767)**. Useful for debugging or direct access without a load balancer.
LoadBalancer	External access with a cloud provider	Automatically provisions a **cloud load balancer** (AWS, GCP, Azure) to expose the service to the internet.
ExternalName	Redirect to an external service	Maps a Kubernetes Service to an **external DNS name** instead of a cluster IP. Used to integrate external APIs.

When to Use Each Type?

- **ClusterIP** → Best for internal microservices (e.g., databases, backend APIs).
- **NodePort** → Suitable for quick external access or on-premise clusters without a cloud provider.
- **LoadBalancer** → Ideal for exposing applications to the internet in **cloud environments**.
- **ExternalName** → Used when forwarding requests to an **external service** (e.g., a SaaS API).

How does Kubernetes handle scaling? Explain the concept of Horizontal Pod Autoscaler and how it responds to workload changes?

Kubernetes Scaling Mechanisms

Kubernetes handles scaling in two ways:

1. **Manual Scaling** → Adjusting the number of Pod replicas manually (kubectl scale).
2. **Autoscaling** → Dynamically adjusting resources based on workload demands.

1. Horizontal Pod Autoscaler (HPA)

HPA **automatically scales the number of Pods** based on CPU, memory, or custom metrics.

- **How it works:**
  - Monitors **metrics** (e.g., CPU utilization).
  - If usage exceeds a threshold, HPA **adds more Pods**.
  - If usage drops, HPA **removes extra Pods** to save resources.

2. Vertical Pod Autoscaler (VPA)

Instead of scaling the number of Pods, **VPA adjusts CPU & memory** requests/limits for Pods.

Describe how you would test the load balancing functionality

```
kubectl exec <pod-name> -- /bin/sh -c 'echo "Hello from Pod: $(hostname)" >> /usr/share/nginx/html/index.html'
```

```
for i in {1..10}; do curl http://<ClusterIP>:8080; done
```

or

```
for i in {1..10}; do curl http://<NodeIP>:30080; done
```

If load balancing is working, you should see responses from different pods:

Hello from Pod: nginx-k8s-task-deployment-5b5fcbf9b7-752s7

Hello from Pod: nginx-k8s-task-deployment-5b5fcbf9b7-f2jkr

Hello from Pod: nginx-k8s-task-deployment-5b5fcbf9b7-krt9f

Scaling with Autoscaling

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

Create the Horizontal Pod Autoscaler