Module 1: Core Concepts

Topic 1: Overview of Container Orchestration

1. What is Container Orchestration?

Container orchestration is the automated management of containerized applications, including deployment, scaling, networking, and lifecycle management. It ensures that containers are efficiently allocated and run across multiple hosts in a cluster.

2. Why Do We Need Container Orchestration?

Before orchestration, managing containers manually was difficult due to:

- The need to run applications consistently across different environments
- Scaling applications dynamically based on demand
- Automating failure recovery and ensuring high availability
- Managing networking between containers and services

Container orchestration tools like **Kubernetes, Docker Swarm, and OpenShift** automate these tasks, making it easier to run microservices-based architectures at scale.

3. Key Features of Container Orchestration

- **Automated Deployment & Scaling**: Containers are deployed and scaled up or down automatically based on demand.
- **Load Balancing & Service Discovery**: Routes traffic to the appropriate container instances.
- **Self-Healing**: Automatically restarts failed containers or replaces unhealthy ones.
- **Storage Orchestration**: Manages persistent storage for stateful applications.
- **Security & Configuration Management**: Ensures secure secrets management and environment-specific configurations.

4. Popular Container Orchestration Tools

Tool	Description	Pros	Cons
Kubernetes	Open-source container orchestration platform	Highly scalable, large community, flexible	Steeper learning curve, complex setup
Docker Swarm	Native container orchestration tool from Docker	Easy to set up, lightweight	Less feature-rich than Kubernetes
OpenShift	Kubernetes-based orchestration platform from Red Hat	Security-focused, enterprise support	Requires Red Hat ecosystem
Amazon ECS/EKS	AWS-managed container orchestration	Integrated with AWS services	Vendor lock-in

Introduction to Kubernetes

1. What is Kubernetes?

Kubernetes (often abbreviated as **K8s**) is an open-source **container orchestration** platform that automates the deployment, scaling, and management of containerized applications. It was originally developed by Google and is now maintained by the **Cloud Native Computing Foundation (CNCF)**.

Kubernetes enables organizations to efficiently run applications in a distributed environment by managing multiple containers across different machines.

---

2. Why Kubernetes?

Before Kubernetes, managing containers at scale was difficult. Kubernetes solves common container management challenges, such as:

Challenges Without Kubernetes

Problem	How Kubernetes Helps
Manual Scaling: You need to manually start/stop containers to handle increased/decreased traffic.	**Auto-scaling**: Kubernetes automatically scales based on CPU/memory usage.
Service Discovery Issues: Containers might move between nodes, making it difficult to access services.	**Service Discovery**: Kubernetes automatically assigns DNS names and load balances traffic.
Container Failures: If a container crashes, manual intervention is required to restart it.	**Self-Healing**: Kubernetes restarts failed containers automatically.
Networking Complexity: Managing communication between containers across multiple machines is difficult.	**Built-in Networking**: Kubernetes provides pod-to-pod communication and external exposure options.

Kubernetes makes deploying and managing containerized applications **scalable, resilient, and automated**.

---

3. Kubernetes Core Components

A. Cluster Architecture

A Kubernetes **cluster** consists of:

Master Node (Control Plane)

Manages the entire cluster and makes scheduling decisions.

Components: API Server, Controller Manager, Scheduler, etcd.

Worker Nodes

Run the application workloads in containers (Pods).

Components: Kubelet, Kube-Proxy, Container Runtime (e.g., Docker, containerd).

Pods

The smallest deployable unit in Kubernetes that runs one or more containers.

B. Kubernetes Objects (Building Blocks)

Object	Description
Pod	The smallest deployable unit that runs a container.
Service	Exposes a set of pods to the network (ClusterIP, NodePort, LoadBalancer).
Deployment	Manages replicas and rolling updates of pods.
ReplicaSet	Ensures a specified number of pod replicas are running.
ConfigMap & Secrets	Store configuration data and sensitive information securely.
Ingress	Manages external access to services using HTTP/HTTPS.

---

4. Kubernetes Use Cases

Kubernetes is widely used in:

- **Microservices Architecture**: Deploying and managing multiple microservices efficiently.
- **CI/CD Pipelines**: Automating application deployment using tools like Jenkins, ArgoCD, or Tekton.
- **Hybrid & Multi-Cloud Deployments**: Running workloads across AWS, Azure, GCP, and on-premise.
- **Big Data & AI/ML**: Managing workloads like Apache Spark and TensorFlow.
- **Edge Computing & IoT**: Running workloads on distributed edge locations.

Understanding Kubernetes Architecture

1. Kubernetes Architecture Overview

Kubernetes follows a **master-worker architecture**, where the **Control Plane (Master Node)** manages the **Worker Nodes** that run containerized applications.

A Kubernetes cluster consists of:

- **Control Plane (Master Node)** – Manages the cluster.
- **Worker Nodes** – Run applications inside containers (Pods).
- **Networking Layer** – Connects all components.

---

2. Kubernetes Cluster Components

A. Control Plane Components (Master Node)

Component	Description
API Server (kube-apiserver)	The **entry point** for all Kubernetes commands (kubectl, UI, automation).
Controller Manager (kube-controller-manager)	Runs controllers to maintain cluster state (e.g., replication, endpoint, node lifecycle controllers).
Scheduler (kube-scheduler)	Assigns Pods to the best Worker Node based on resource availability.
etcd	Stores cluster configuration data (key-value store).

Example:

When a user creates a **Deployment**, the **API Server** stores it in **etcd**, the **Scheduler** assigns Pods to Nodes, and the **Controller Manager** ensures the right number of replicas.

B. Worker Node Components

Component	Description
Kubelet	Runs on each worker node; ensures containers are running in Pods.
Kube-Proxy	Manages networking and load balancing across services.
Container Runtime (Docker, containerd, CRI-O)	Runs containers inside Pods.

Example:

When a Pod is scheduled to a Node, **Kubelet** pulls the container image and runs it using the **Container Runtime**.

C. Kubernetes Networking

Kubernetes uses a flat network model, where all Pods can communicate.
Key networking concepts:

- **Pod-to-Pod Communication**: Managed by the network plugin (CNI: Calico, Flannel, Cilium).
- **Service Discovery**: Kubernetes **Services** provide stable networking for Pods.
- **Ingress Controller**: Routes external traffic to the correct Service inside the cluster.

---

3. Kubernetes Objects and How They Work Together

Object	Role in Architecture
Pod	Smallest unit that runs containers.
Deployment	Manages Pods and ensures replicas are maintained.
Service	Exposes Pods internally or externally.
ConfigMap/Secret	Stores configuration data securely.
Ingress	Controls external HTTP/HTTPS access.

Kubernetes Component: Kube API Server

What it Does:

The **Kube API Server** is one of the central components of Kubernetes and serves as the primary interface for interacting with the Kubernetes cluster. It is responsible for exposing the Kubernetes API and handling all requests made to the cluster, whether from users, other services, or even the nodes and pods within the cluster itself.

Key functions of the Kube API Server include:

1. **API Handling**: It exposes a REST API that allows users and other components to interact with the cluster. It validates and processes API requests, executes corresponding actions, and returns the results.
2. **Cluster State Management**: The API server maintains the desired state of the cluster by communicating with the etcd data store. It saves and retrieves cluster state information, such as configurations, resource definitions, and metadata.
3. **Authentication & Authorization**: It enforces security policies by authenticating and authorizing incoming requests.
4. **Communication Hub**: The API Server serves as the central point of communication between all components, such as the kube-scheduler, kube-controller-manager, and worker nodes.

How it Works:

- The **Kube API Server** listens on port 6443 by default and can handle requests using HTTP/HTTPS protocols. It processes requests for resources like pods, deployments, services, etc.
- When a user or service wants to interact with the cluster (e.g., create a pod, retrieve the status of a deployment), they send a request to the API server using the Kubernetes API endpoints.

- The API server processes the request and interacts with the etcd cluster (a key-value store) to read or write the state of the resources.
- The API server enforces security measures such as validating requests, ensuring the right permissions, and ensuring that only authorized users can perform specific actions.
- It then communicates with the necessary components, like the scheduler, controllers, and nodes, to act on the requests.
- The API server is typically stateless, relying on **etcd** for persistent state storage.

Example of How It Works:

Suppose you want to deploy a new application (a pod) in the Kubernetes cluster. You would send a kubectl command like:

```
kubectl run nginx --image=nginx
```

1. **User Request**: The kubectl command makes an HTTP request to the **Kube API Server**, specifically a POST request to /api/v1/pods.
2. **Request Processing**: The API Server validates the request, ensuring the user has the necessary permissions, and checks the provided resource definition (i.e., the nginx image).
3. **Interaction with etcd**: The API server writes the new pod resource definition to the **etcd** key-value store to record the desired state.
4. **Controller Interaction**: The **kube-controller-manager** notices the new pod specification in the etcd database and ensures that the pod is scheduled onto an appropriate node.
5. **Response**: The API server responds to kubectl with the status of the request, confirming that the pod creation was successful.

This whole process involves a series of interactions, and the **Kube API Server** serves as the central point that manages and tracks these actions across the cluster.

Example API Requests:

- **Get Pod Information**:

```
curl -k https://<API_SERVER_IP>:6443/api/v1/pods/<POD_NAME>
```

- **Create Pod (Using JSON/YAML)**:

```
curl -k -X POST https://<API_SERVER_IP>:6443/api/v1/namespaces/default/pods \
-H "Content-Type: application/json" \
-d '{"apiVersion": "v1", "kind": "Pod", "metadata": {"name": "nginx"}, "spec": {"containers":
[{"name": "nginx", "image": "nginx"}]}}'
```

In the above requests, the **Kube API Server** processes the interaction, validates the request, updates **etcd**, and interacts with the Kubernetes scheduler and other components to ensure that the state is updated accordingly.

Real-World Use Case:

In a large enterprise environment, multiple teams may use a shared Kubernetes cluster. Each team needs to deploy applications or update resources but must do so securely and within predefined permissions. The **Kube API Server** ensures that each team's actions are valid by authenticating them and enforcing access control policies (like Role-Based Access Control, or RBAC). For example, a DevOps team can deploy new applications, but only a system administrator might have the permissions to manage cluster-wide resources like nodes or network configurations.

Conclusion:

The **Kube API Server** is critical to the functioning of a Kubernetes cluster, handling all cluster management tasks through its REST API. It acts as a central controller, interacting with other components, managing state via **etcd**, and enforcing security and governance policies.

Kubernetes Component: ETCD

What it Does:

etcd is a distributed key-value store that is used by Kubernetes to store and manage the state of the entire cluster. It holds critical data, such as the configuration, resource definitions (e.g., pods, services, deployments), and the current state of the cluster. This makes **etcd** a central component for ensuring the consistency and reliability of the cluster.

Key functions of **etcd** include:

1. **Cluster State Management**: It stores all Kubernetes resource definitions, including pods, deployments, nodes, and services, ensuring the cluster's state is preserved.
2. **Configuration Storage**: **etcd** is used to store configuration data, such as the Kubernetes API server’s configuration, secrets, and custom configurations for components.
3. **Consistent State**: It uses the **Raft consensus algorithm** to ensure that all instances of **etcd** in the cluster maintain a consistent view of the data. This guarantees that the state of the cluster is accurate and reliable.
4. **Leader Election**: **etcd** enables leader election for high availability and fault tolerance, ensuring that there is a consistent leader to manage the write operations to the key-value store.

How it Works:

- **etcd** stores data in a key-value format, where the key is a unique identifier for the data and the value is the associated data.
- **etcd** uses the **Raft consensus algorithm** to ensure that all nodes in the etcd cluster agree on the state of the data, even if some nodes fail.
- **etcd** is designed to be highly available and reliable. If one of the **etcd** instances (nodes) fails, another instance will take over, ensuring there is no data loss and no interruption in service.
- **Kubernetes** interacts with **etcd** through the **Kube API Server**. When a resource is created, updated, or deleted, the **API Server** writes to **etcd**, ensuring the cluster's desired state is stored.
- **etcd** is often deployed as a cluster (usually with an odd number of nodes) to maintain fault tolerance and availability. It can be scaled horizontally, and the leader election process ensures that only one instance is responsible for handling writes at any given time.

Example of How It Works:

Let's say a user wants to create a new deployment for a web application in Kubernetes. When the user runs the command:

```
kubectl apply -f deployment.yaml
```

1. **API Server Request**: The kubectl command sends a request to the **Kube API Server** to create the deployment.
2. **Write to etcd**: The **Kube API Server** validates the request and then writes the deployment definition to **etcd**.
3. **etcd Stores the State**: The deployment's configuration is stored in **etcd** as a key-value pair. The key might represent the type of resource (e.g., deployment), and the value contains the configuration details of the deployment (e.g., replicas, container image, labels).
4. **Cluster State**: All components, such as the **kube-scheduler** and **kube-controller-manager**, read the cluster state from **etcd** to ensure the desired state is achieved. For example, the **scheduler** will decide where to run the new pods based on the information stored in **etcd**.
5. **Leader Election**: If the **etcd** leader crashes during the process, another leader will be elected, ensuring that the system remains consistent and operational.

Example API Interaction with etcd:

- **Get Cluster Information from etcd**: To retrieve cluster data directly from **etcd** (e.g., the list of all pods in the cluster), you can use the following command (assuming etcdctl is installed and configured):

```
etcdctl get /registry/pods --prefix
```

This would retrieve all the pod information stored in **etcd**.

- **Put Data in etcd**: To store a key-value pair directly in **etcd**, you can use:

```
etcdctl put /myapp/config '{"replicas": 3, "image": "nginx"}'
```

This stores a configuration for an application in **etcd** under the key `/myapp/config`.

Real-World Use Case:

Consider a Kubernetes cluster in a cloud-based production environment where different services (like databases, web servers, and microservices) need to be highly available. **etcd** ensures that the state of each service is maintained accurately and consistently across all Kubernetes components. If a pod crashes or if the scheduler needs to move a pod to another node, **etcd** holds the authoritative record of the cluster's desired state, allowing the system to recover quickly and restore the correct configuration.

Conclusion:

etcd is a critical component of Kubernetes, acting as the centralized store for all cluster data and configurations. It ensures consistency and availability of this data, which is essential for the proper operation of the Kubernetes control plane. By using the Raft consensus algorithm, **etcd** ensures that the data stored within it is consistent, even in the face of failures, providing reliability to Kubernetes clusters.

Kubernetes Component: Kube Controller Manager

What it Does:

The **Kube Controller Manager** is a control plane component that runs controllers responsible for maintaining the desired state of the Kubernetes cluster. It is a set of controllers that regulate various aspects of the cluster, ensuring that the cluster's actual state matches the desired state defined by the user or system.

Controllers are loops that watch the cluster's state and take corrective actions if the actual state deviates from the desired state. For example, if you declare that you want a certain number of replicas for a deployment, the **Kube Controller Manager** ensures that the correct number of replicas are running in the cluster, adding or removing pods as necessary.

Key responsibilities of the **Kube Controller Manager** include:

1. **Replication**: Ensures that the correct number of pod replicas are running as specified in a Deployment or ReplicaSet.
2. **Node Management**: Handles the lifecycle of nodes in the cluster, ensuring that nodes are healthy, registered, and properly managed.
3. **Resource Cleanup**: Handles the cleanup of resources like terminating pods or jobs when their lifecycle ends or when specified conditions are met.
4. **Lease Management**: Manages the leasing mechanism for resources like statefulsets or jobs to control their state across nodes.
5. **Cross-resource Validation**: Ensures that resources like PodDisruptionBudgets and HorizontalPodAutoscalers are applied properly, often in coordination with other controllers.

In essence, the **Kube Controller Manager** works to ensure the cluster is in the desired state and will continuously monitor and correct it.

How it Works:

The **Kube Controller Manager** runs a set of controllers that each take care of specific tasks. Each controller listens to events in the cluster and checks whether the current state of resources matches the desired state. If there is a discrepancy, the controller takes corrective action.

- **Event-Driven Architecture**: The controllers are event-driven and watch the Kubernetes API Server for changes to resources. When a resource (like a Pod or Deployment) is modified, the controller reacts by updating the system to bring the cluster back to the desired state.
- **Controller Types**: Some of the main controllers within the **Kube Controller Manager** include:
    - **ReplicaSet Controller**: Ensures that the desired number of replicas are running for a specific Deployment.
    - **Deployment Controller**: Manages the creation, scaling, and rollout of applications defined in Deployments.
    - **Node Controller**: Monitors the health of nodes and manages node registration.
    - **Job Controller**: Ensures that jobs (i.e., workloads that should run to completion) run successfully to completion.
    - **StatefulSet Controller**: Manages the lifecycle of stateful applications and ensures that the desired number of replicas are always running.
    - **EndpointSlice Controller**: Manages the slices of the network endpoints for services.

Each controller has a loop that watches the cluster's state and performs actions to reconcile it with the desired state.

Example of How It Works:

Suppose you deploy a Deployment with a replicas: 3 specification:

```
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
```

1. **User Request**: When you apply this configuration using kubectl apply -f nginx-deployment.yaml, the **Kube API Server** stores the Deployment object in **etcd**.
2. **Controller Manager Watches**: The **Deployment Controller** in the **Kube Controller Manager** continuously watches the API server for changes related to Deployment resources.
3. **Replica Management**: The **Deployment Controller** detects that the desired state is to have 3 replicas of the nginx pod. It will then check whether there are already 3 replicas running or not.
4. **Action Taken**: If there are fewer than 3 replicas, the **Deployment Controller** will create new Pods to bring the actual state in line with the desired state.
5. **Reconciliation**: The **Controller Manager** continues to monitor the state of the Pods (such as if a pod crashes or is deleted) and ensures that the number of nginx pods stays at 3.

If one of the nginx pods crashes and stops running, the controller manager will take corrective action and launch a new pod to replace it, thus maintaining the desired state of the Deployment.

Example of a Controller in Action:

- **Scaling a Deployment**: If you change the number of replicas for your Deployment:

```
kubectl scale deployment nginx-deployment --replicas=5
```

- The **Deployment Controller** in the **Kube Controller Manager** will detect that the desired state is now 5 replicas and will ensure that the necessary pods are created to match that count.

Real-World Use Case:

In a Kubernetes-based microservices environment, different applications (e.g., a web app, database, etc.) might be running on separate Deployments. Each application might have varying replica requirements, and some applications need to be highly available at all times.

The **Kube Controller Manager** ensures that these applications always run with the right number of replicas. For instance, if the database application is set to have 3 replicas for redundancy, the **Controller Manager** will ensure that the database always runs with 3 pods. If any of those pods fail, the **Controller Manager** will create a new pod to maintain the desired state.

Conclusion:

The **Kube Controller Manager** is a crucial component of the Kubernetes control plane. It runs controllers that ensure the cluster is always in the desired state, by monitoring changes and taking corrective actions. From maintaining pod replicas to ensuring the proper functioning of resources like stateful sets and jobs, the **Kube Controller Manager** is responsible for ensuring the reliability, scalability, and health of the Kubernetes cluster.

Kubernetes Component: Kube Scheduler

What it Does:

The **Kube Scheduler** is a core component of the Kubernetes control plane responsible for deciding which node a newly created pod should run on. It is the component that ensures that workloads (pods) are distributed efficiently across the available nodes in the cluster based on various scheduling rules and constraints.

Key functions of the **Kube Scheduler** include:

1. **Node Selection**: It selects an appropriate node for each pod, considering resource requirements, affinity rules, taints, tolerations, and available resources (CPU, memory, etc.).
2. **Resource Optimization**: It attempts to optimize resource utilization across the cluster by placing pods on nodes that have the required resources and conditions.
3. **Scheduling Constraints**: The scheduler can take into account user-defined scheduling constraints such as:
    1. **Pod Affinity/Anti-Affinity**: Ensuring that certain pods are scheduled together or apart.
    2. **Taints and Tolerations**: Ensuring that pods are scheduled on nodes that can tolerate specific taints.
    3. **Resource Requests/Limitations**: Ensuring that nodes have enough resources (e.g., CPU, memory) to run the pod.
    4. **Node Selectors**: Placing pods on nodes that match specific labels.
4. **Scheduling Policies**: It can take into account policies such as the **PodPriority** and **PodDisruptionBudget** to determine which pods should be scheduled first.

How it Works:

The **Kube Scheduler** works by monitoring the API server for newly created pods that do not yet have an assigned node. When a pod is created or needs to be rescheduled (for example, after a node failure), the scheduler follows a series of steps to determine which node the pod should be placed on:

1. **Filter Nodes**: It filters out nodes that cannot accommodate the pod based on:

   1. Resource availability (e.g., CPU, memory).
   2. Affinity and anti-affinity rules (pods that need to be placed together or separated).
   3. Taints and tolerations (ensuring that pods only land on nodes with matching tolerations).
   4. Node selectors (if specified in the pod's configuration).

2. **Score Nodes**: After filtering, the scheduler assigns scores to the remaining nodes based on various factors such as resource utilization, proximity to other pods, and the overall load on the nodes.
3. **Select Node**: The scheduler then picks the node with the highest score.
4. **Bind Pod to Node**: Finally, the scheduler informs the **Kube API Server** to update the pod's specification with the chosen node. The pod is then scheduled to run on that node.

Example of How It Works:

Let's say you deploy a pod and you specify that it requires 2 CPUs and 4 GB of memory. If there are three nodes in your cluster with the following available resources:

- **Node A**: 4 CPUs, 8 GB of memory
- **Node B**: 2 CPUs, 6 GB of memory
- **Node C**: 8 CPUs, 16 GB of memory

1. **Pod Request**: You create the following pod definition:

```
apiVersion: v1
kind: Pod
metadata:
  name: nginx-pod
spec:
  containers:
  - name: nginx
    image: nginx
  resources:
    requests:
      cpu: "2"
      memory: "4Gi"
```

Scheduler Evaluation: The **Kube Scheduler** will check the available nodes for resources that match the pod's requirements:

- **Node A** has enough resources (4 CPUs, 8 GB memory).
- **Node B** has enough resources (2 CPUs, 6 GB memory).
- **Node C** has more than enough resources but may not be ideal for reasons like being underutilized or having other scheduling constraints (such as taints or node affinity).

Node Scoring: The scheduler scores each node based on how well they fit the pod's resource requirements and any constraints defined in the pod's configuration (e.g., affinity rules). If all nodes are eligible, the scheduler might choose the node with the lowest load.

Binding: After selecting the best node (e.g., **Node B**), the scheduler will update the pod's specification to bind it to that node.

Pod Scheduled: The pod is now scheduled to run on **Node B** with 2 CPUs and 4 GB of memory.

Example of Scheduling with Taints and Tolerations:

Suppose you want to ensure that certain pods are scheduled only on nodes with specific characteristics. You can use **taints** and **tolerations** for this.

1. **Taint a Node**: You taint a node with a custom taint:

```
kubectl taint nodes node-1 key=value:NoSchedule
```

This means that no pod will be scheduled on **node-1** unless it has a matching **toleration**.

1. **Pod Toleration**: You then create a pod that tolerates this taint:

```
apiVersion: v1
kind: Pod
metadata:
  name: nginx-pod
spec:
  containers:
  - name: nginx
    image: nginx
  tolerations:
  - key: "key"
    operator: "Equal"
    value: "value"
    effect: "NoSchedule"
```

1. **Scheduler Action**: The **Kube Scheduler** will then place the pod on **node-1** because it has the matching toleration for the taint applied to that node.

Real-World Use Case:

In a production environment, organizations might need to schedule pods in a way that minimizes latency or optimizes resource usage. For example:

- **Affinity/Anti-Affinity**: Pods for a web application might need to be scheduled on nodes close to a database pod (using **pod affinity**), while preventing certain workloads from running on the same node (using **anti-affinity**).
- **Resource-based Scheduling**: A cluster might have specialized nodes (e.g., nodes with GPUs for machine learning workloads), and you can use **node selectors** to ensure that such workloads only run on the nodes with GPUs.
- **Custom Taints**: Nodes designated for batch processing might be tainted to ensure that no other pods (e.g., real-time applications) are scheduled on those nodes.

Conclusion:

The **Kube Scheduler** is responsible for making intelligent decisions about where to place pods within the cluster based on available resources, scheduling constraints, and policies. It ensures that the desired state of the system, such as resource distribution and affinity requirements, is met. By evaluating node resources and policies like affinity, tolerations, and resource requests, the scheduler ensures that workloads are optimally placed, helping maintain the efficiency and health of the cluster.

Kubernetes Component: Kubelet

What it Does:

The **Kubelet** is an essential node-level component in Kubernetes. It is an agent that runs on each node in the cluster and ensures that the containers specified in PodSpecs are running and healthy. The **Kubelet** communicates with the **Kube API Server** to manage the state of the node, as well as to report back on the state of the containers running on it.

Key functions of the **Kubelet** include:

1. **Pod Lifecycle Management**: It ensures that containers are started, stopped, and run according to the specifications in the Pod definition.
2. **Health Checking**: The **Kubelet** monitors the health of containers (using **liveness** and **readiness probes**) and takes corrective actions (e.g., restarting the container if it fails).
3. **Resource Reporting**: It reports node-level resource usage (e.g., CPU, memory) to the **Kube API Server** so that the scheduler can make informed decisions about pod placement.
4. **Node Registration**: The **Kubelet** registers the node with the Kubernetes cluster, making it available for scheduling pods.
5. **Container Runtime Interaction**: The **Kubelet** interacts with the container runtime (e.g., Docker, containerd) to manage containers on the node. It requests the container runtime to pull images, create containers, and manage container lifecycle events.

How it Works:

The **Kubelet** runs on every node in the Kubernetes cluster and watches the **Kube API Server** for the Pods assigned to its node. The main steps it follows to manage pod lifecycle are:

Pod Spec Synchronization:

1. The **Kubelet** regularly checks the **Kube API Server** for the list of Pods assigned to its node.
2. If a new pod is scheduled to run on the node, the **Kubelet** downloads the PodSpec (the specification of the pod) from the API Server.
3. The **Kubelet** then makes sure the containers defined in the PodSpec are running.

Container Management:

1. The **Kubelet** interacts with the container runtime (such as Docker, containerd, or CRI-O) to create, start, stop, and manage containers as described in the PodSpec.
2. It monitors the status of the containers, checking whether they are running, healthy, or terminated.

Health Monitoring:

1. The **Kubelet** checks the health of each container using **liveness** and **readiness probes**:

1. **Liveness probes**: Ensure that a container is still alive and functioning. If a container fails the liveness check, the **Kubelet** will kill and restart the container.
2. **Readiness probes**: Ensure that the container is ready to serve traffic. If a container fails the readiness probe, the **Kubelet** will mark the pod as "not ready," preventing it from receiving traffic.

Reporting Back to API Server:

1. The **Kubelet** constantly reports the status of the node and containers to the **Kube API Server**, which then updates the state of the cluster.
2. If a pod is not running or if the node is under high load, the **Kubelet** reports this to the API Server, which may trigger rescheduling or other actions.

Resource Management:

1. The **Kubelet** monitors the resource usage (CPU, memory) of containers and ensures they do not exceed the specified limits. It may evict pods if the node runs out of resources, ensuring the overall health of the node.

Node Health Checks:

1. The **Kubelet** also checks the overall health of the node itself, reporting whether the node is ready to schedule new pods or if it should be marked as unhealthy.

Example of How It Works:

Let’s say a user creates a pod with two containers in the following manifest:

```
apiVersion: v1
kind: Pod
metadata:
  name: nginx-pod
spec:
  containers:
  - name: nginx-container
    image: nginx:latest
    ports:
      - containerPort: 80
  - name: redis-container
    image: redis:latest
    ports:
      - containerPort: 6379
```

Pod Assignment: The **Kube Scheduler** assigns this pod to a node in the cluster. The **Kubelet** running on that node detects the new pod and fetches the PodSpec from the **Kube API Server**.

Container Launching: The **Kubelet** then instructs the container runtime (e.g., Docker) to pull the `nginx:latest` and `redis:latest` images from the container registry and run them as containers in the pod.

Health Monitoring: The **Kubelet** will monitor the containers using configured probes (for example, liveness and readiness probes) to ensure that both containers are healthy. If any container fails the probe, the **Kubelet** will restart the container.

Reporting: The **Kubelet** reports the status of the pod and containers (whether they are running, failed, or completed) back to the **Kube API Server**, which keeps track of the overall state of the cluster.

Node Resource Management: The **Kubelet** will monitor resource usage (e.g., CPU and memory) and make sure that the containers do not exceed the allocated resources. If the node runs low on resources, the **Kubelet** may evict pods to make room for others.

Example of Health Check with Liveness and Readiness Probes:

Here's an example where the **Kubelet** uses **liveness** and **readiness** probes to manage container health:

```
apiVersion: v1
kind: Pod
metadata:
  name: nginx-pod
spec:
  containers:
  - name: nginx
    image: nginx:latest
    ports:
      - containerPort: 80
    livenessProbe:
      httpGet:
        path: /healthz
        port: 80
      initialDelaySeconds: 5
      periodSeconds: 10
    readinessProbe:
      httpGet:
        path: /readiness
        port: 80
      initialDelaySeconds: 5
      periodSeconds: 10
```

- **Liveness Probe**: The **Kubelet** will call /healthz on the nginx container every 10 seconds (after an initial delay of 5 seconds). If the container does not respond with a healthy status, the **Kubelet** will restart the container.
- **Readiness Probe**: The **Kubelet** will check /readiness to determine when the container is ready to serve traffic. Until the readiness probe passes, the pod will not be marked as "ready" to receive traffic.

Real-World Use Case:

Consider a scenario in a Kubernetes cluster where a web application has multiple replicas of the nginx container running. Each **Kubelet** on the worker nodes is responsible for managing the lifecycle of the pods on its node, ensuring that:

- The containers are running.
- The containers are healthy.

- The resources are not being over-consumed.

If a pod's container fails due to a crash or an issue, the **Kubelet** will detect the failure through the liveness probe and restart the container, ensuring minimal disruption to the service. If a node is under resource pressure (e.g., CPU or memory overload), the **Kubelet** may evict lower-priority pods to free up resources, keeping the node functional.

Conclusion:

The **Kubelet** is a vital node-level component in Kubernetes that ensures containers are running as expected, reporting their status, and handling container lifecycle events. It is the primary agent responsible for maintaining the state of the containers on a specific node and works in close coordination with the **Kube API Server**, container runtimes, and other components in the cluster to ensure that applications are running correctly and efficiently.

Module 2: Installation, Configuration & Validation

Topic 1: Designing a Kubernetes Cluster

1. Introduction to Kubernetes Cluster Design

A well-designed Kubernetes cluster ensures **high availability, scalability, and security** for workloads. Before installation, we must consider:
Cluster size and topology (single-node vs. multi-node, high availability)
Infrastructure choices (on-premises, cloud, bare metal, VMs)
Networking model (CNI, Pod-to-Pod communication, external access)
Storage requirements (Persistent Volumes, CSI plugins)
Security and access control (RBAC, TLS, firewall rules)

---

2. Kubernetes Cluster Architecture

A Kubernetes cluster typically consists of:

Component	Role
Master Node (Control Plane)	Manages scheduling, API requests, and cluster state.
Worker Nodes	Run the actual application workloads inside Pods.
Networking Layer	Ensures communication between Pods, Nodes, and external users.

For this module, we will design and install a **1-Master, 2-Worker Node Kubernetes cluster using kubeadm**.

---

3. Cluster Design Considerations

Before installing Kubernetes, we must plan:

Factor	Decision
Control Plane	Single master node (not highly available)
Worker Nodes	Two worker nodes for running applications
Networking	Calico (CNI-based network plugin)
Storage	Local storage for testing, cloud storage for production
Ingress & Load Balancing	Nginx Ingress Controller
Security	RBAC enabled, TLS for secure communication

Why this setup?

A **single master node** is easier to manage for learning/testing.

Two worker nodes allow scheduling of workloads with failover.

Calico provides secure networking and network policies.

kubeadm automates cluster setup, making it production-ready.

LAB: Designing Kubernetes Cluster (Planning Phase)

Step 1: Define Infrastructure Requirements

Component	Details
Master Node	2 vCPUs, 4GB RAM, 20GB Disk
Worker Nodes	2 vCPUs, 4GB RAM, 20GB Disk each
OS	Ubuntu 22.04 LTS
Network Plugin	Calico
Container Runtime	containerd
Kubernetes Version	v1.28+
Required Ports	6443 (API Server), 10250 (Kubelet), 2379-2380 (etcd)

Step 2: Network Design

Kubernetes networking is critical for **Pod-to-Pod** and **Service** communication.
Pod CIDR: 192.168.0.0/16 (each Pod gets an IP)

Service CIDR: 10.96.0.0/12 (virtual IPs for Services)
Node CIDR: Nodes get their own IP range for Pod allocation

Step 3: Preparing for Installation

To install a **1-Master, 2-Worker cluster using kubeadm**, we need:

3 Ubuntu 22.04 machines (VMs, bare metal, or cloud instances).

Static IPs assigned to each machine.

Firewall Rules adjusted for Kubernetes ports.

SSH Access for remote management.

Step 4: Verify System Requirements

Run the following on **all nodes**:

```
# Check CPU and RAM
lscpu && free -m

# Check OS version
cat /etc/os-release

# Check network connectivity
ping -c 3 google.com
```

Summary

- **Planned Kubernetes cluster design** (1 master, 2 workers).
- **Chose network model** (Calico for Pod networking).
- **Verified system requirements** (Ubuntu 22.04, networking, firewall rules).

Installation of Kubernetes 1-Master and 2-Node Cluster using kubeadm

1. Introduction

We will install a **1-Master, 2-Worker Kubernetes cluster** using **kubeadm**.

Master Node: Controls and manages the cluster.

Worker Nodes: Run application workloads inside Pods.

We will set up:
kubeadm – Tool to bootstrap the cluster.
kubelet – Runs on all nodes, responsible for managing Pods.
kubectl – Command-line tool to interact with the cluster.
Container runtime – containerd for running containers.

---

2. Prerequisites

Component	Requirement
OS	Ubuntu 22.04 LTS
CPU/RAM	2 vCPUs, 4GB RAM per node
Disk	20GB free space
Network Plugin	Calico
Container Runtime	containerd
Kubernetes Version	v1.29
User Privileges	Root or sudo access

Required Ports

Port	Component	Direction
6443	API Server	Incoming
2379-2380	etcd	Internal
10250-10255	Kubelet	Internal
30000-32767	NodePort Services	External

3. LAB: Installing Kubernetes Cluster with kubeadm

Step 1: Prepare All Nodes

Run these steps on **Master and Worker Nodes**.

1.1 Update the system

```
sudo apt update && sudo apt upgrade -y
```

1.2 Disable Swap (Required by Kubernetes)

```
sudo swapoff -a
sed -i '/swap/d' /etc/fstab
```

1.3 Load Required Kernel Modules

```
cat <<EOF | sudo tee /etc/modules-load.d/k8s.conf
overlay
```

```
br_netfilter
EOF
sudo modprobe overlay
sudo modprobe br_netfilter
```

1.4 Configure Network Settings

```
cat <<EOF | sudo tee /etc/sysctl.d/k8s.conf
net.bridge.bridge-nf-call-iptables = 1
net.bridge.bridge-nf-call-ip6tables = 1
net.ipv4.ip_forward = 1
EOF
  sudo sysctl --system
```

Step 2: Install Container Runtime (containerd)

```
sudo apt install -y containerd
```

Configure containerd:

```
sudo mkdir -p /etc/containerd
containerd config default | sed 's/SystemdCgroup = false/SystemdCgroup = true/' | sed 's/sandbox_image = "registry.k8s.io\/pause:3.8"/sandbox_image = "registry.k8s.io\/pause:3.9"/' | sudo tee /etc/containerd/config.toml
```

Restart service:

```
sudo systemctl restart containerd
sudo systemctl enable containerd
```

Step 3: Install kubeadm, kubelet, and kubectl

```
sudo apt update
sudo apt-get install -y apt-transport-https ca-certificates curl gpg
sudo mkdir -p -m 755 /etc/apt/keyrings
curl -fsSL https://pkgs.k8s.io/core:/stable:/v1.29/deb/Release.key | sudo gpg --dearmor -o /etc/apt/keyrings/kubernetes-apt-keyring.gpg
echo 'deb [signed-by=/etc/apt/keyrings/kubernetes-apt-keyring.gpg] https://pkgs.k8s.io/core:/stable:/v1.29/deb/ /' | sudo tee /etc/apt/sources.list.d/kubernetes.list

sudo apt update
sudo apt install -y kubelet kubeadm kubectl
sudo apt-mark hold kubelet kubeadm kubectl
sudo systemctl enable --now kubelet
```

Verify installation:

```
kubeadm version
```

Step 4: Initialize the Master Node

Run **ONLY on the Master Node**.

```
kubeadm init --pod-network-cidr=192.168.0.0/16 >> cluster_initialized.txt

cat cluster_initialized.txt
```

After successful initialization, you will see a **kubeadm join** command. Copy it.

4.1 Set Up kubectl for the Master Node

```
mkdir -p $HOME/.kube
sudo cp -i /etc/kubernetes/admin.conf $HOME/.kube/config
sudo chown $(id -u):$(id -g) $HOME/.kube/config
```

Verify the cluster:

```
kubectl get nodes
```

Step 5: Configure Networking (Calico)

Install Calico on the **Master Node**:

```
kubectl apply -f https://docs.projectcalico.org/manifests/calico.yaml
```

Verify that all Pods are running:

```
kubectl get pods -n kube-system
```

Step 6: Join Worker Nodes to the Cluster

Run the **kubeadm join** command from **Step 4** on each Worker Node. Example:

```
sudo kubeadm join <MASTER_IP>:6443 --token <TOKEN> --discovery-token-ca-cert-hash
sha256:<HASH>
```

Verify the cluster:

```
kubectl get nodes
```

You should see **Master** and **Worker Nodes** in the Ready state.

Summary

✅ Installed Kubernetes using **kubeadm**.
✅ Set up **containerd** as the runtime.
✅ Configured **Master Node** and initialized the cluster.

✅ Installed **Calico** for networking.
✅ **Joined Worker Nodes** to the cluster.