Kubernetes

What is Kubernetes?

Kubernetes (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)**.

Why Kubernetes?

Containers (e.g., Docker) solve many problems of traditional application deployment, but managing multiple containers across multiple servers manually is complex. **Kubernetes automates this process** by efficiently managing container lifecycles, networking, scaling, and failover.

Features of Kubernetes

1. **Container Orchestration**
    - Automates deployment, scaling, and management of containerized applications.
2. **Multi-Node Clustering**
    - Runs applications across multiple nodes, avoiding single-host limitations of Docker.
3. **Auto-Healing**
    - Automatically detects failed containers and restarts or replaces them.
4. **Auto-Scaling**
    - Supports **Horizontal Pod Autoscaler (HPA)** and **Vertical Pod Autoscaler (VPA)** to scale containers based on CPU and memory usage.
5. **Load Balancing & Service Discovery**
    - Distributes network traffic evenly to ensure high availability.
    - Built-in service discovery using Kubernetes **Services**.
6. **Rolling Updates & Rollbacks**
    - Allows seamless updates of applications with zero downtime.
    - Enables rollback to previous versions if necessary.
7. **Storage Orchestration**
    - Supports persistent storage (e.g., AWS EBS, Azure Disk, NFS, Ceph, etc.).
    - Manages volumes dynamically.
8. **Declarative Configuration using YAML/JSON**
    - Uses **Kubernetes manifests** for defining application states and desired configurations.

9. **Secret & Configuration Management**
    - Securely manages sensitive data like passwords, API keys, and certificates.
    - Uses **Secrets** and **ConfigMaps**.
10. **Network Policies**
    - Implements fine-grained network access control between pods.
11. **Extensibility & Plugins**
    - Supports **Custom Resource Definitions (CRDs)** to extend Kubernetes functionalities.
    - Integrates with **service meshes** like Istio and Linkerd.
12. **Monitoring & Logging**
    - Integrated with **Prometheus, Grafana, Fluentd, and ELK Stack** for observability.

Challenges of Kubernetes

1. **Steep Learning Curve**
    - Requires knowledge of **Docker, networking, security, YAML configurations**, and **infrastructure concepts**.
2. **Complex Setup & Management**
    - Requires expertise to configure clusters, manage nodes, and optimize workloads.
    - Managing Kubernetes manually can be overwhelming.
3. **Security Risks**
    - Requires strict IAM roles, network policies, and proper secret management.
    - Misconfigured permissions can lead to vulnerabilities.
4. **Resource Overhead**
    - Consumes more compute and memory resources than standalone containers.
    - Needs optimization to prevent inefficiencies.
5. **Networking Complexity**
    - Requires **Container Network Interface (CNI)** plugins like Calico, Flannel, or Cilium.
    - Complex network configurations for multi-cloud or hybrid deployments.
6. **Storage & Data Management**
    - Managing **stateful applications** (e.g., databases) in Kubernetes is challenging.
    - Persistent storage integration varies across cloud providers.
7. **Monitoring & Debugging**
    - Requires third-party tools for in-depth monitoring and logging.
    - Debugging distributed applications can be difficult.

8. **High Initial Costs**
    - Running Kubernetes clusters on **cloud providers (AWS, GCP, Azure)** can be costly.
    - Needs proper cost optimization strategies.
9. **Upgrades & Compatibility Issues**
    - Frequent updates require careful upgrade planning.
    - Some Kubernetes versions may not be compatible with existing plugins.
10. **Networking & Load Balancing in Multi-Cloud Environments**

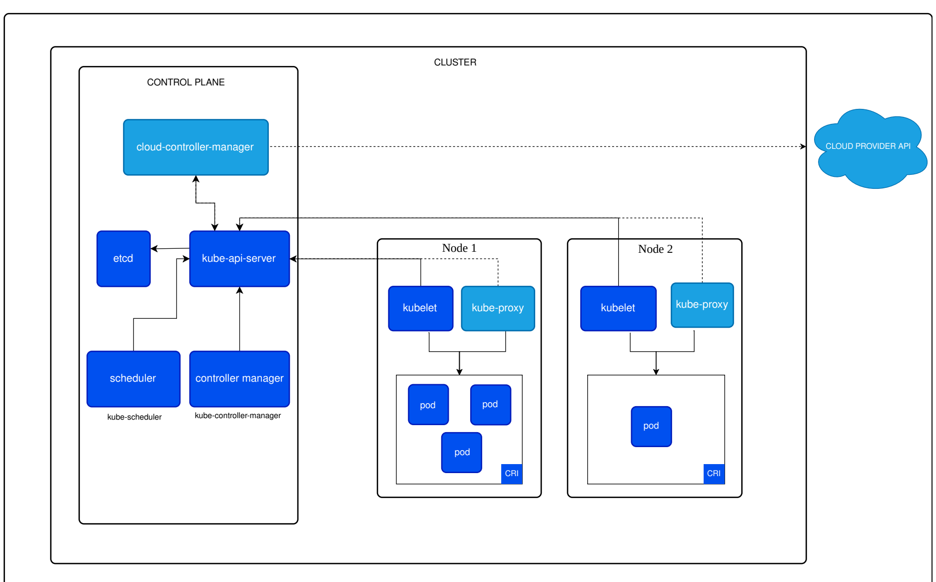
- Managing Kubernetes across different cloud providers adds complexity.
- Load balancing between on-prem and cloud-based Kubernetes clusters is challenging.

Key Advantages of Kubernetes Over Docker

- **Clustered by Design** – Kubernetes works across multiple nodes, unlike Docker's single-host limitation.
- **Auto-Healing** – Automatically restarts failed containers, ensuring reliability.
- **Auto-Scaling** – Adjusts resources dynamically based on workload.
- **Enterprise-Level Support** – Provides advanced features for large-scale deployments.
- **Additional Features:**
    - **Load Balancing** – Distributes traffic efficiently among pods.
    - **Security** – Implements **RBAC (Role-Based Access Control)** and other security measures.
    - **Networking** – Uses **CNI (Container Network Interface)** plugins for connectivity.

Kubernetes Control Plane and Data Plane (Architecture)

- Kubernetes architecture consists of **two main components**:
    1. **Control Plane** – Manages cluster operations.
    2. **Data Plane** – Handles workloads and executes instructions from the control plane.



Control Plane Components:

- **API Server –** Acts as the main gateway for communication within the cluster.
- **etcd –** Stores the cluster's state in a distributed key-value store.
- **Scheduler –** Assigns workloads (Pods) to available nodes.
- **Controller Manager –** Ensures the cluster maintains the desired state.
- **Cloud Controller Manager –** Manages integration with cloud providers.

Data Plane Components:

- **Kubelet –** Ensures pods and containers are running correctly.
- **Kube-Proxy –** Handles networking between pods, including **IP management and load balancing**.
- **Container Runtime –** Executes containerized applications within pods (**Docker, containerd, CRI-O**).

Docker vs. Kubernetes – Component Comparison

Feature	Docker	Kubernetes
Basic Unit	Container	Pod (a group of one or more containers)
Container Management	Docker Engine	Kubelet
Networking	Bridge Networking	Kube-Proxy with CNI plugins
Scaling	Manual	Automatic (Horizontal & Vertical Scaling)
Self-Healing	No	Yes (Auto-restarts failed pods)

Detailed Breakdown of Kubernetes Components

- **Kubelet**
  - Ensures pods are running and healthy.
  - Reports container status to the control plane.
- **Kube-Proxy**
  - Manages networking for pods and handles traffic routing.
  - Provides internal load balancing for services.
- **Container Runtime**
  - Required to run containers inside pods.
  - Kubernetes supports **Docker, containerd, CRI-O, and other runtimes**.

Control Plane Components Explained

- **Kubernetes API Server (`kube-apiserver`)**

What is the API Server?

- The **API Server** is the **core component** of the **Kubernetes Control Plane**.
- It acts as the **gateway** for all interactions with the Kubernetes cluster.
- It is responsible for **processing REST requests**, validating them, and updating the cluster state in **etcd**.

Key Functions of the API Server

1. **Handles All Kubernetes API Requests**
    - Processes `kubectl`, `client SDK`, and `other API` requests.
    - Communicates with internal Kubernetes components (e.g., Scheduler, Controller Manager).
2. **Authentication & Authorization**
    - Ensures that requests are **authenticated** (via certificates, tokens, or third-party identity providers).
    - Implements **RBAC (Role-Based Access Control)** for access management.
3. **Validation & Admission Control**
    - Ensures the request structure is valid before storing in `etcd`.
    - Uses **Admission Controllers** to apply additional security policies.
4. **Communication Hub**
    - Routes information between all Kubernetes components.
    - Exposes APIs for external users (DevOps tools, dashboards, monitoring tools, etc.).
5. **Load Balancing & High Availability**
    - In a multi-master setup, API Servers are typically behind a **load balancer** for high availability.
    - API requests are distributed across multiple instances.

API Server Architecture

The API Server follows a **RESTful architecture**, supporting `CRUD` operations on Kubernetes resources.

Operation	HTTP Method	Example Command
Create	`POST`	`kubectl create -f pod.yaml`
Read	`GET`	`kubectl get pods`
Update	`PUT/PATCH`	`kubectl edit deployment`
Delete	`DELETE`	`kubectl delete pod my-pod`

How the API Server Works

1. **User sends a request** via `kubectl`, an SDK, or an external tool.
2. The API Server **authenticates** and **authorizes** the request.
3. It **validates** the request and checks for policy compliance.
4. If valid, the API Server **persists the updated state** in `etcd`.
5. The API Server **notifies controllers** (like Scheduler, Controller Manager) to act based on the new state.

Key Components of the API Server

1. **Request Handler**
    - Receives API requests and processes them based on the HTTP method.
2. **Admission Controllers**
    - Enforce policies before storing data (e.g., `ResourceQuota`, `PodSecurityPolicy`).
3. **etcd Storage Interface**
    - Stores and retrieves cluster state from `etcd`.
4. **Aggregation Layer**
    - Allows extension of Kubernetes API with custom APIs.
5. **Authentication & Authorization**
    - Uses `TLS`, `Bearer Tokens`, `OAuth`, `RBAC`, and `Service Accounts` for security.

High Availability for API Server

To ensure high availability:

- Deploy **multiple API Server replicas**.
- Use a **Load Balancer** in front of API Servers.
- Use **etcd in HA mode** to maintain consistent cluster state.

Securing the API Server

1. **Enable RBAC**: Restrict access to API resources.
2. **Use TLS Encryption**: Secure API Server communication.
3. **Enable Audit Logging**: Monitor API activities.
4. **Limit API Access**: Use Network Policies or API Rate Limiting.
5. **Disable Anonymous Access**: Prevent unauthorized API calls.

Commands to Interact with API Server

- Check API Server status:

  ```
  kubectl get componentstatuses
  ```

- View API Server logs:

  ```
  kubectl logs -n kube-system kube-apiserver-master-node
  ```

- List available API resources:

  ```
  kubectl api-resources
  ```

- Get API Server version:

  ```
  kubectl version --short
  ```

Conclusion

- The **API Server** is the **brain** of Kubernetes, managing all cluster communication.
- It **authenticates, validates, processes, and routes** all Kubernetes API requests.
- Ensuring **high availability, security, and performance** is crucial for production deployments.

■ Scheduler

- The **Kubernetes Scheduler** is a control plane component responsible for **assigning pods to nodes** based on resource availability, constraints, and scheduling policies. It ensures that workloads are efficiently distributed across a cluster while meeting application requirements.

The Kubernetes Scheduler is responsible for **deciding** which node in the cluster should run a newly created pod. It considers multiple factors such as:

- **Resource availability** (CPU, memory, disk, network).
- **Node taints and tolerations**.
- **Affinity and anti-affinity rules**.
- **Node selectors and constraints**.
- **Custom scheduling policies**.

If a pod **does not specify a node**, the scheduler picks one **automatically**. If the scheduler cannot find a suitable node, the pod remains in the **Pending** state.

■ etcd

etcd is a distributed, consistent, and highly available **key-value store** used as the **backbone of Kubernetes** and other distributed systems. It provides **storage for configuration, cluster state, and leader election** while ensuring data consistency using the **Raft consensus algorithm**.

Key Features of etcd

- **Highly Available**: Supports leader election and replication.
- **Strong Consistency**: Uses the Raft consensus algorithm for consistent state replication.
- **Watch Mechanism**: Allows real-time monitoring of changes.
- **Key-Value Store**: Stores structured data in a hierarchical format.
- **Lightweight & Fast**: Optimized for low-latency reads and writes.
- **Secure**: Supports TLS encryption and authentication.

- Used for **disaster recovery and backup**.

- ## Controller Manager

  The **Kubernetes Controller Manager** is a core control plane component responsible for running **controllers** that regulate the cluster's state. It continuously watches the cluster state and ensures the actual state matches the desired state.

  ### Types of Controllers in Kubernetes

1. Node Controller

- Detects **unhealthy nodes** and removes them.
- Monitors **node heartbeats** and updates node status.
- Automatically reschedules pods from failed nodes.

2. Replication Controller

- Ensures the correct number of **pod replicas** are running.
- Creates or deletes pods to match the desired state.

3. Deployment Controller

- Manages **rolling updates** and **rollbacks** for deployments.
- Ensures **zero downtime updates**.

4. StatefulSet Controller

- Manages **stateful applications** (e.g., databases).
- Ensures each pod has a unique **persistent identity**.

5. DaemonSet Controller

- Ensures that a pod runs on **every node** (or specific nodes).
- Used for **logging, monitoring, or networking agents** (e.g., Fluentd, Prometheus Node Exporter).

6. Job Controller

- Manages **batch jobs** that run and terminate after completion.
- Used for **data processing, backups, and cron jobs**.

7. CronJob Controller

- Schedules **recurring jobs** at specified intervals (like a cron job).

8. Service Controller

- Manages **load balancers and service endpoints**.
- Ensures that services correctly route traffic to pods.

9. Endpoint Controller

- Maps **services to available pods**.
- Updates **endpoints** dynamically when pods start/stop.

10. Persistent Volume Controller

- Handles **PersistentVolumeClaims (PVCs)**.
- Ensures pods are bound to persistent storage.

11. Horizontal Pod Autoscaler (HPA) Controller

- Scales pods **up or down** based on CPU/memory usage.

12. Certificate Signing Controller

- Manages Kubernetes **TLS certificates**.
- Automates the signing of CertificateSigningRequests (CSRs).

Role of the Controller Manager in Kubernetes

The Controller Manager is responsible for **automating cluster management** by running multiple controllers in the background. It:

1. Listens to the **Kubernetes API Server** for changes.
2. Compares the **desired state** (from etcd) with the **current state**.
3. Makes necessary adjustments to maintain the **desired state**.

Example:

- If a pod crashes, the **ReplicaSet Controller** creates a new one.
- If a node becomes unhealthy, the **Node Controller** removes it from the cluster.

■ Cloud Controller Manager

A **Cloud Control Manager (CCM)** is a **centralized system** that manages, automates, and monitors cloud resources across different cloud providers (AWS, Azure, Google Cloud, etc.). It provides a **unified interface** to create, update, delete, and monitor cloud infrastructure, ensuring operational efficiency and security.

Why Use a Cloud Control Manager?

1. Centralized Cloud Management

- Single interface to manage resources across multiple cloud platforms.
- Standardized API and UI for operations.

2. Multi-Cloud & Hybrid Cloud Support

- Works across AWS, Azure, Google Cloud, and on-premise infrastructure.

3. Automation & Orchestration

- Automates provisioning and scaling of cloud resources.
- Helps in DevOps workflows and Infrastructure as Code (IaC).

4. Cost Optimization

- Tracks resource utilization and optimizes costs by automating scaling.

5. Security & Compliance

- Implements security policies and ensures regulatory compliance (e.g., GDPR, HIPAA).

Summary of Kubernetes Architecture

- The **Control Plane** manages the cluster, while the **Data Plane** executes workloads.
- The **API Server** is the primary interface for cluster operations.
- **Kubelet and Kube-Proxy** are critical for pod management and networking.
- Kubernetes enhances **scalability, automation, and reliability** over Docker.
- Understanding **component interactions** is key to mastering Kubernetes.

Additional Key Concepts

- **Pod Lifecycle –** Pods go through different phases (Pending, Running, Succeeded, Failed).
- **Namespaces –** Isolate resources within a Kubernetes cluster.
- **Ingress Controller –** Manages external traffic to services inside the cluster.
- **Persistent Volumes (PV & PVC) –** Handles stateful applications by providing storage.
- **Helm –** A package manager for Kubernetes, simplifies deployment management.
- **Service Mesh (Istio/Linkerd) –** Enhances microservice networking, security, and observability.

Understanding Kubernetes in Production

1. Importance of Understanding Kubernetes in Production

- DevOps engineers must **understand Kubernetes in real-world production environments**.
- Many learners use **local setups like Minikube or K3s**, but these are **not production-ready**.
- Minikube and K3s are useful for **development and learning**, but **real-world deployments** require a more scalable and resilient Kubernetes setup.

---

2. Kubernetes Distributions

- **Kubernetes distributions** are variations of Kubernetes tailored for **enterprise use**.
- Different organizations use different **Kubernetes distributions**, and understanding these is crucial for **job interviews**.
- Popular Kubernetes distributions include:
    - **Amazon EKS (Elastic Kubernetes Service) –** Fully managed by AWS.
    - **Red Hat OpenShift –** Enterprise Kubernetes with built-in DevOps tools.
    - **VMware Tanzu –** Kubernetes with integration for VMware environments.
    - **Google GKE (Google Kubernetes Engine) –** Managed Kubernetes service on Google Cloud.
    - **Azure AKS (Azure Kubernetes Service) –** Microsoft's managed Kubernetes offering.
- Distributions offer **enterprise support, security enhancements, and better integrations**.

---

3. Differences Between Kubernetes and Managed Services

- **Self-managed Kubernetes (Kubeadm, KOPS, etc.)**:
    - Users **install, configure, and manage** the cluster.
    - Requires **troubleshooting, upgrades, monitoring, and security management**.
    - More control but **higher operational burden**.
- **Managed Kubernetes Services (EKS, AKS, GKE, OpenShift, etc.)**:
    - Cloud provider **manages the control plane**, networking, security, and upgrades.
    - Easier deployment and maintenance.
    - Suitable for enterprises looking for **scalability and support**.
- Some organizations **prefer self-managed Kubernetes** for specific needs like **on-premises deployment, cost control, or regulatory compliance**.

---

4. Managing Kubernetes Clusters in Production

- **Managing multiple clusters** is a key responsibility of **DevOps engineers**.
- Tools for managing Kubernetes lifecycle:
    - **KOPS (Kubernetes Operations) –** Automates cluster provisioning, upgrades, and scaling.
    - **Kubeadm –** Basic tool for setting up Kubernetes clusters (mostly used for small setups or testing).
    - **Rancher –** Enterprise-grade Kubernetes cluster management platform.
    - **Cluster API –** Provides declarative Kubernetes cluster management.
    - **Terraform –** Automates Kubernetes infrastructure deployment.
- **KOPS vs. Kubeadm**:
    - **KOPS** is widely used in production due to **better automation and scaling support**.

    - **Kubeadm** requires **manual configuration**, making it less ideal for managing large-scale clusters.

---

5. Additional Considerations for Production Kubernetes

- **High Availability (HA)**:
    - Run **multi-node clusters** to ensure uptime.
    - Use **load balancers** for distributing traffic.
    - Ensure **etcd** (Kubernetes database) is replicated for fault tolerance.
- **Security Best Practices**:
    - Enable **RBAC (Role-Based Access Control)**.
    - Use **network policies** to restrict pod communication.
    - Regularly **update and patch clusters**.
- **Observability & Monitoring**:
    - Use **Prometheus & Grafana** for monitoring.
    - Implement **logging tools like ELK (Elasticsearch, Logstash, Kibana)**.
    - Set up **alerting mechanisms** for failures.
- **Scaling & Performance Optimization**:
    - Use **Horizontal Pod Autoscaler (HPA)** to auto-scale workloads.
    - Implement **Cluster Autoscaler** to adjust node count dynamically.
    - Optimize resource requests and limits for efficient usage.

Conclusion

- **Understanding Kubernetes in production is essential** for DevOps professionals.
- **Kubernetes distributions** (EKS, OpenShift, Tanzu, etc.) are commonly used in enterprises.
- **Self-managed vs. managed Kubernetes**: trade-off between control and ease of management.
- **KOPS, Kubeadm, and other tools** help in managing clusters at scale.
- **Security, monitoring, and scaling** are critical for production deployments.

Kubernetes vs. Docker

- **Kubernetes Advantages:**
    - Manages clusters of containers efficiently.
    - Supports **auto-scaling** (adjusts resources based on load).
    - Ensures **auto-healing** (restarts failed containers automatically).
    - Facilitates **enterprise-level deployment** with robust networking, storage, and service discovery.
    - Implements **rolling updates** and **rollback strategies** seamlessly.

- Kubernetes acts as an orchestrator, managing multiple containers across different nodes, unlike Docker, which primarily handles individual containers on a single host.

Understanding Kubernetes Pods

- **Pods**: The smallest deployable unit in Kubernetes, grouping one or more containers.
- Unlike Docker, which runs standalone containers, Kubernetes uses **YAML configuration files** to define pod specifications.
- Pods enable better resource management and networking capabilities compared to standalone containers in Docker.

Pod Specifications and YAML Configuration

- YAML files define key parameters such as API version, pod name, container image, and resource limits.
- Structure of a simple Pod YAML file:

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

- Mastering YAML is essential for efficient Kubernetes management and automation.

Networking and Storage in Kubernetes Pods

- Containers within the same pod share **network namespaces**, allowing inter-container communication via `localhost`.
- **Storage volumes** are shared across containers in a pod, useful for applications requiring persistent data sharing.
- Use cases for multi-container pods:
  - **Sidecar containers** (e.g., logging, monitoring agents).
  - **Init containers** (set up preconditions before the main container runs).

Introduction to kubectl

- `kubectl` is the command-line tool used to interact with Kubernetes clusters.
- Essential `kubectl` commands:
  - `kubectl get nodes` – Lists all nodes in the cluster.
  - `kubectl get pods` – Retrieves all running pods.

    - kubectl describe pod <pod-name> – Provides detailed information about a specific pod.
    - kubectl logs <pod-name> – Displays logs for troubleshooting.
    - kubectl delete pod <pod-name> – Deletes a pod.

Setting Up Minikube

- Minikube enables local Kubernetes cluster creation for testing and learning.
- Steps to install Minikube:
    - Choose the appropriate virtualization platform (VirtualBox, Docker, Hyper-V, etc.).
    - Install Minikube and start a cluster using:

```
minikube start
```

    - Verify cluster status:

```
kubectl cluster-info
```

    - Deploy applications and experiment with Kubernetes locally.

Deploying a Pod in Kubernetes

- Example: Running an **Nginx** container using YAML:

```
kubectl apply -f pod.yaml
```

- Check deployment status:

```
kubectl get pods
```

- Benefits of YAML-based deployment:
    - Declarative approach to infrastructure management.
    - Enables version control and repeatability.

Debugging and Managing Kubernetes Pods

- Common debugging techniques:
    - kubectl logs <pod-name> – Check container logs.
    - kubectl describe pod <pod-name> – View detailed information about pod state.
    - kubectl exec -it <pod-name> -- /bin/sh – Access container shell.

Introduction to Kubernetes Deployment

- Kubernetes Deployment is a higher-level abstraction that manages **pod creation, scaling, and updates**.
- Provides **auto-healing, auto-scaling, and rolling updates** for applications.
- Helps in ensuring **zero downtime deployments** in production.

Understanding Containers, Pods, and Deployments

- **Containers**: The basic unit of application packaging and execution (e.g., Docker containers).
- **Pods**: The smallest deployable unit in Kubernetes that **encapsulates one or more containers** sharing networking and storage.
- **Deployments**: Manages the lifecycle of pods and provides **replication, auto-scaling, and rolling updates**.

Comparison of Pods and Deployments

Feature	Pods	Deployments
Definition	Smallest deployable unit with one or more containers	Manages multiple replicas of pods
Scaling	Manual	Automated (replica sets)
Self-Healing	No	Yes
Zero Downtime Update	No	Yes
Use Case	Debugging, testing	Production deployments

Why Use Deployments Instead of Pods?

- **Auto-Healing**: If a pod crashes, Kubernetes replaces it automatically.
- **Scaling**: Supports **horizontal scaling** to handle high traffic.
- **Zero Downtime**: Uses **Rolling Updates** to avoid service disruption.
- **ReplicaSet Management**: Ensures the desired number of pods are always running.

Kubernetes Deployment Workflow

1. **Define a Deployment YAML file**
   - Specifies the desired number of replicas, container image, and update strategy.
2. **Create a Deployment**
   - `kubectl apply -f deployment.yaml`

3. **Kubernetes creates a ReplicaSet**
    - Ensures the required number of pod replicas are always running.
4. **Deployment ensures desired state**
    - If a pod crashes, the ReplicaSet replaces it.
5. **Scaling and Updating**
    - Adjust replicas dynamically (`kubectl scale deployment`).
    - Deploy new versions without downtime.

Kubernetes Deployment YAML Example

A **basic deployment YAML file**:

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


Key Elements in YAML

- `replicas: 3` → Runs 3 instances of the pod.
- `selector.matchLabels` → Ensures the Deployment controls specific pods.
- `template.spec.containers` → Defines container details (image, ports).

Kubernetes Commands for Deployments

Command	Description
`kubectl create -f deployment.yaml`	Creates a deployment
`kubectl get deployments`	Lists all deployments
`kubectl describe deployment <deployment-name>`	Shows details of a deployment

Command	Description
kubectl delete deployment <deployment-name>	Deletes a deployment
kubectl scale deployment <deployment-name> --replicas=5	Scales deployment to 5 replicas
kubectl rollout status deployment <deployment-name>	Checks deployment update status
kubectl rollout history deployment <deployment-name>	Shows update history

---

Auto-Healing and Zero-Downtime Deployments

- **Auto-Healing:**
  - If a pod fails, ReplicaSet automatically **replaces it**.
  - Ensures application availability without manual intervention.
- **Zero-Downtime Updates:**
  - **Rolling Updates** update pods **one at a time** to avoid downtime.
  - **Recreate Strategy** replaces all pods at once (causes downtime).

Rolling Update Command:

```
kubectl set image deployment/nginx-deployment nginx=nginx:1.19
```

Checking Update Status:

```
kubectl rollout status deployment/nginx-deployment
```

Rolling Back to a Previous Version:

```
kubectl rollout undo deployment/nginx-deployment
```

Scaling Kubernetes Deployments

- Deployments allow **scaling pods horizontally** to handle increased traffic.
- Can be done manually or automatically (**Horizontal Pod Autoscaler - HPA**).

Manually Scaling a Deployment:

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

Setting Up Auto-Scaling (HPA):

```
kubectl autoscale deployment nginx-deployment --min=2 --max=10 --cpu-percent=80
```

Best Practices for Deployments

1. **Use Deployments instead of standalone pods** for production.
2. **Define resource limits** to avoid overloading nodes.
3. **Enable auto-scaling** for high-traffic applications.
4. **Monitor health with liveness & readiness probes** in YAML:

```
livenessProbe:
  httpGet:
    path: /health
    port: 80
  initialDelaySeconds: 3
  periodSeconds: 5
```

5. **Use ConfigMaps & Secrets** for configuration management.
6. **Leverage Rolling Updates** to deploy new versions without downtime.

Conclusion

- Deployments **automate scaling, updating, and healing** of applications.
- ReplicaSets ensure **desired pod counts** are maintained.
- Use `kubectl` commands to **create, scale, update, and debug deployments**.
- **Zero-downtime updates** and **auto-healing** make Kubernetes ideal for production.

ReplicaSet in Kubernetes

Introduction to ReplicaSet

A **ReplicaSet (RS)** is a Kubernetes resource that ensures a specified number of **identical pod replicas** are running at any given time. It continuously monitors pods and ensures that the desired number of replicas are maintained, automatically replacing failed or deleted pods.

Why Use ReplicaSets?

- **Auto-healing**: If a pod fails or is accidentally deleted, ReplicaSet recreates it.
- **Load balancing**: Ensures multiple instances of an application are running for better performance.
- **Scalability**: Makes it easy to increase or decrease the number of pods dynamically.
- **Stateless applications**: Works well for applications that do not require persistent storage.

How Does a ReplicaSet Work?

1. The **ReplicaSet controller** continuously checks the number of running pods.
2. If the number of pods is **less than the desired state**, it automatically creates new pods.
3. If the number of pods is **greater than the desired state**, it deletes the excess pods.
4. It uses **labels and selectors** to manage pods efficiently.

ReplicaSet YAML Example

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


Explanation of the YAML File

- `replicas: 3` → Ensures three pods are running.
- `selector.matchLabels.app: nginx` → Ensures the ReplicaSet manages pods with the label `app=nginx`.
- `template.spec.containers` → Defines the container to run inside the pod.

Managing ReplicaSets Using kubectl

Command	Description
`kubectl create -f replicaset.yaml`	Creates a ReplicaSet
`kubectl get rs`	Lists all ReplicaSets
`kubectl describe rs <replicaset-name>`	Provides details of a ReplicaSet
`kubectl delete rs <replicaset-name>`	Deletes a ReplicaSet
`kubectl scale rs <replicaset-name> --replicas=5`	Increases/decreases the number of replicas
`kubectl get pods --selector=app=nginx`	Lists all pods managed by the ReplicaSet

Scaling a ReplicaSet

- ReplicaSets support **manual scaling** to increase or decrease the number of pods.

Example: Scale a ReplicaSet to 5 replicas

```
kubectl scale rs nginx-replicaset --replicas=5
```

Conclusion

- **ReplicaSets ensure high availability and auto-healing** of applications.
- **They manage pod replicas** but lack advanced update and rollback features.
- **Deployments are preferred in production**, as they provide **auto-scaling and zero-downtime updates**.

Kubernetes Services

A **Kubernetes Service** is an abstraction that provides a **stable IP and DNS name** to a group of **pods**, enabling seamless communication even if pod IPs change. It ensures **load balancing, service discovery, and external access**

Understanding the Need for Services

- In Kubernetes, pods are **ephemeral**, meaning they can be **created, deleted, or rescheduled** dynamically.
- If an application is deployed without a service, it becomes difficult to **track** and **access pods** when their IPs change.

- Example:
  - A pod is created → Assigned an IP (e.g., `10.1.2.3`).
  - The pod crashes → A new pod is created with a **different IP** (e.g., `10.1.2.4`).
  - If a user/application was communicating with the old IP, it **fails** because the IP changed.
- **Services solve this problem** by providing a **stable** way to access pods.

Challenges Without Services

- If no service is used:
  - Users need to manually **track** pod IPs, which keep changing.
  - A **pod restart** results in a **new IP**, making the application inaccessible unless the new IP is updated everywhere.
  - Load distribution is **manual** (users might hit the wrong pod).

Example Scenario Without Services

Time	Action	Pod IP	User Access
10:00 AM	Pod created	`10.1.2.3`	User accesses it successfully
10:05 AM	Pod crashes	`10.1.2.3` is lost	User request fails
10:06 AM	New Pod created	`10.1.2.4`	User needs to manually update the IP

➡️ **Kubernetes services eliminate these issues by providing a fixed access point to the pods.**

Service Discovery Mechanism

- Kubernetes uses **labels and selectors** to **track** pods dynamically.
- When a service is created, it selects pods based on **labels** instead of fixed IPs.
- If a pod crashes and a new one is created, the service automatically **routes traffic** to the new pod without manual intervention.

Types of Kubernetes Services

1. ClusterIP (Default Service)

- **Scope:** Internal only (within the cluster).
- **Use Case:** Microservices **talking to each other**.
- **Access:** Cannot be accessed from outside the cluster.

2. NodePort

- **Scope:** Exposes the service on a **static port** on each Kubernetes node.
- **Use Case:** For **external access** (within a company or test environments).
- **Access:** `http://<Node-IP>:<NodePort>`

3. LoadBalancer

- **Scope:** Uses **cloud provider's load balancer** to provide external access.
- **Use Case: Production** applications (e.g., web apps, APIs).
- **Access:** `http://<External-IP>:80`

4. ExternalName

- **Scope:** Maps a service to an **external domain name** (DNS).
- **Use Case:** When Kubernetes needs to **communicate with an external database** or external service.
- **Access:** Resolves the domain but does not create a new Kubernetes resource.

5. Headless Service

- **Scope:** Allocate directly to NodeIP rather than clusterIP
- **Use Case:** Used in **stateful applications** (like databases) where direct pod access is needed.
- **Access:** Uses DNS to resolve pod IPs directly.

Introduction to Kubernetes Ingress

Ingress in Kubernetes is a way to manage incoming traffic to your applications inside a cluster. It acts like a smart traffic controller that directs requests (mainly HTTP and HTTPS) to the right service based on rules.

Instead of exposing each service separately, Ingress allows multiple services to share a single external IP, supports HTTPS, and can distribute traffic efficiently.

Understanding the Need for Ingress

- Before Kubernetes version 1.1, users relied solely on services for application exposure and load balancing.
- Kubernetes services provided only basic load balancing (round-robin), which was insufficient for advanced use cases.
- Enterprise-level load balancing capabilities were lacking, leading to performance and scalability concerns.
- The cost of using LoadBalancer-type services was high, as cloud providers charged per static IP, making it expensive to expose multiple applications.

Limitations of Kubernetes Services

- Traditional load balancers offered advanced features such as:
    - Sticky sessions
    - Path-based routing
    - Security capabilities (e.g., SSL termination, authentication, etc.)
- Kubernetes services (ClusterIP, NodePort, LoadBalancer) lacked these features.
- Users migrating to Kubernetes found its service-level round-robin load balancing inadequate.
- Cloud providers charged per static IP for LoadBalancer services, increasing costs for organizations managing multiple applications.

Introduction to Ingress Resource

- Ingress solves these problems by allowing users to define routing rules within Kubernetes.
- It enables path-based and host-based routing to direct traffic without requiring multiple load balancer IPs.
- Users must create Ingress resources, which are managed by an Ingress controller.
- Benefits of Ingress:
    - Reduces the number of load balancer IPs needed
    - Provides advanced routing capabilities
    - Enables SSL/TLS termination
    - Supports authentication mechanisms

Ingress Controller Functionality

- Ingress controllers (e.g., nginx, HAProxy, Traefik) manage Ingress resources and implement routing logic.
- Organizations can choose an Ingress controller based on their specific needs.
- Ingress controllers continuously monitor Ingress resources and update configurations accordingly.
- They act as reverse proxies, efficiently directing external traffic to internal services.

Practical Implementation of Ingress

    - Verify the application is running.
    - Create an Ingress resource YAML file defining routing rules.

- Deploy an Ingress controller in the Kubernetes cluster.
- Apply the Ingress resource and verify traffic routing.
  without an active Ingress controller, the Ingress resource will not function.

Additional Notes:

- Ingress is essential for managing external access efficiently in Kubernetes clusters.
- Choosing the right Ingress controller depends on performance, security, and feature requirements.
- Proper DNS configuration is necessary for smooth domain-based traffic routing.

Understanding Kubernetes RBAC

What is RBAC?

- **RBAC (Role-Based Access Control)** is a security mechanism used in Kubernetes to manage user and service account permissions.
- It helps control **who** can perform **what actions** on **which resources** within a cluster.
- Essential for organizations to prevent unauthorized access and ensure secure operations.

Key Concepts of RBAC

1. **User Management**
    - In a local Kubernetes cluster, users typically have administrative access by default.
    - In enterprise environments, different access levels must be defined for **development, QA, and operations teams**.
    - RBAC allows permissions to be assigned based on user roles, preventing unauthorized actions.
2. **Service Account Management**
    - Service accounts are used by applications and pods running inside Kubernetes.
    - They define what resources a pod can access.
    - Misconfigured service accounts can expose sensitive resources to malicious pods.
    - RBAC ensures applications have only the necessary permissions.

Components of Kubernetes RBAC

Kubernetes RBAC consists of three main components:

1. Roles and ClusterRoles

- **Roles**: Define permissions within a specific **namespace**.
- **ClusterRoles**: Define permissions **cluster-wide**, across all namespaces.

- Roles specify **verbs (actions)** like `get`, `list`, `create`, `delete` and associate them with Kubernetes resources.

Example of a Role

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

- This Role allows users to **get, list, and create** Pods and Services in the `dev` namespace.

2. RoleBinding and ClusterRoleBinding

- **RoleBinding**: Grants a **Role** to a **user or service account** within a specific namespace.
- **ClusterRoleBinding**: Grants a **ClusterRole** across all namespaces.

Example of a RoleBinding

```
apiVersion: rbac.authorization.k8s.io/v1
kind: RoleBinding
metadata:
  name: dev-rolebinding
  namespace: dev
subjects:
- kind: User
  name: john-doe
  apiGroup: rbac.authorization.k8s.io
roleRef:
  kind: Role
  name: dev-role
  apiGroup: rbac.authorization.k8s.io
```

- This RoleBinding assigns **dev-role** to the user **john-doe** in the `dev` namespace.

3. Service Accounts

- Service accounts are used by pods to interact with the Kubernetes API.
- RBAC ensures that service accounts only have permissions they require.

Example of a Service Account

```
apiVersion: v1
kind: ServiceAccount
metadata:
  name: app-service-account
  namespace: dev
```

- This creates a service account named **app-service-account** in the `dev` namespace.

User Management and Identity Providers

- Kubernetes does not handle user authentication directly.
- It relies on external **Identity Providers (IDPs)** like:
    - **AWS IAM** (for AWS-managed Kubernetes clusters)
    - **LDAP** (Lightweight Directory Access Protocol)
    - **OAuth** (for third-party authentication)
- Configuring IDPs correctly ensures that only authorized users can access Kubernetes resources.

Exploring the OpenShift Environment

- OpenShift provides a **user-friendly web interface** for managing Kubernetes workloads.
- Users are assigned a **dedicated namespace** to explore features safely.
- The OpenShift UI allows users to:
    - Monitor workloads
    - Manage deployments
    - Scale applications
- OpenShift simplifies Kubernetes RBAC by offering built-in tools for role and policy management.

Key Takeaways

1. **RBAC is crucial** for securing Kubernetes clusters by controlling access to resources.
2. **Roles and ClusterRoles** define **what actions** can be performed.
3. **RoleBindings and ClusterRoleBindings** link roles to **users or service accounts**.
4. **Service Accounts** provide controlled access for applications running inside Kubernetes.
5. **External Identity Providers** manage user authentication in Kubernetes.
6. **OpenShift offers a free sandbox environment** for practicing RBAC concepts.

By understanding and implementing RBAC correctly, Kubernetes administrators can enforce strong security policies and prevent unauthorized access to the cluster.

Here’s a detailed version of your study notes on **Custom Resources in Kubernetes** with well-structured explanations:

Custom Resources in Kubernetes

Overview of Kubernetes Resources

Kubernetes provides a set of **built-in resources** that allow users to manage workloads efficiently within a cluster. These include:

- **Deployments –** Used for managing replicas of an application.
- **Services –** Provide network connectivity to applications.
- **Pods –** The smallest deployable unit in Kubernetes.
- **ConfigMaps & Secrets –** Store configuration data and sensitive information, respectively.

Extending Kubernetes with Custom Resources

- Sometimes, **built-in resources** may not be enough to meet specific requirements.
- Kubernetes allows users to extend its API by defining **custom resources**, which introduce **new resource types** that can be managed like native resources.
- Examples of tools using **custom resources**:
  - **Kube-hunter –** A security tool for scanning Kubernetes clusters.
  - **Argo CD –** A GitOps continuous deployment tool that integrates with Kubernetes.

The Need for Custom Resource Definitions (CRDs)

What is a CRD?

- A **Custom Resource Definition (CRD)** is a **YAML configuration** that defines a new **API resource type** in Kubernetes.
- A **CRD acts as a blueprint** for creating custom resources.
- Once a CRD is created, users can manage new resource types **just like built-in Kubernetes objects**.
- **Custom Resource Definitions (CRDs)** are used to define **new resource types** in Kubernetes, expanding its default API.
- Examples of tools relying on **CRDs**:
  - **Istio –** Uses CRDs like `VirtualService` to manage **traffic routing** in a service mesh.
  - **Cert-Manager –** Uses CRDs to manage **TLS certificates** within Kubernetes.
- **Deployment Responsibility**:
  - DevOps engineers deploy **CRDs** to extend Kubernetes.
  - Users can then create **instances** of the newly defined custom resources.

Understanding Custom Resource Definitions (CRDs)

A **Custom Resource Definition (CRD)** acts as a **blueprint** for a new Kubernetes resource.

- A **CRD** is defined in a **YAML file** and specifies:
  - **The new API type**
  - **The structure of the custom resource**
  - **Validation rules** to ensure the correct format
- When a user creates a **custom resource**, Kubernetes checks if it matches the **CRD specifications**.
- Example:
  - If Istio defines a `VirtualService` CRD, users can create a `VirtualService` resource to control **service-to-service communication**.

The Role of Custom Controllers

- A **Custom Controller** manages **custom resources**, ensuring that they remain in the **desired state**.
- It follows the **Kubernetes Controller pattern**, similar to how **built-in controllers** (like the ReplicaSet controller) operate.

- **Functions of a Custom Controller**:
  - Watches for **changes** in custom resources.
  - Performs **actions** based on the changes (e.g., deploying a new component).
  - Ensures the system maintains the **desired state**.
- **Deployment Methods**:
  - **Helm charts**
  - **Kubernetes manifests** (`YAML` files)

Steps to Implement Custom Resources

1. **Define the CRD**

   - Create a **CRD YAML file** to specify the new API resource type.
   - Apply the CRD to the cluster using:

     ```
     kubectl apply -f custom-resource-definition.yaml
     ```

2. **Create Custom Resource Instances**

   - Once the CRD is in place, users can create instances of the new resource.
   - Example:

     ```
     apiVersion: myorg.example.com/v1
     kind: MyCustomResource
     metadata:
       name: example-resource
     spec:
       key: value
     ```

   - Apply it using:

     ```
     kubectl apply -f custom-resource.yaml
     ```

3. **Deploy the Custom Controller**

   - The controller **watches** for changes in the custom resource and takes necessary actions.
   - It can be deployed using Kubernetes manifests or Helm.

Writing Custom Controllers

- **Custom controllers** are responsible for managing the lifecycle of **custom resources**.
- **Programming Languages**:
  - **Go** (Recommended, as Kubernetes is built in Go and has strong library support).
  - **Python, Java**, or other languages can also be used.
- **Key Components** of a Controller:

    - **Watcher** – Monitors the state of custom resources.
    - **Reconciler** – Ensures the actual state matches the desired state.
    - **Kubernetes API Interaction** – Uses **client-go** library in Go.
- **Example of a Controller in Go**:

```go
package main

import (
    "fmt"
    "time"
)

func main() {
    for {
        fmt.Println("Watching for changes in custom resources...")
        time.Sleep(10 * time.Second) // Simulate watching logic
    }
}
```

- **Deployment Methods**:
    - Using **YAML manifests**
    - Using **Helm charts**
    - Running as a **Kubernetes deployment**

Resources for Learning and Implementation

- **Official Kubernetes Documentation** – Covers CRDs and controllers in detail.
- **Istio Documentation** – Practical examples of CRDs used for traffic management.
- **GitHub Repositories**:
    - [Kubernetes Sample Controller] – Example implementation of a custom controller.
    - [Operator SDK] – Helps in building Kubernetes operators (advanced controllers).

Conclusion and Next Steps

- **Summary**:
    - **Custom Resources** extend Kubernetes with new resource types.
    - **CRDs** define the structure of custom resources.
    - **Custom Controllers** manage these resources and ensure the desired state.

Kubernetes: Config Maps and Secrets.

Understanding Config Maps

- **Definition**: Stores **non-sensitive configuration data** required by applications.
- **Use Case**:
    - Example: A backend app retrieving data from a database.
    - Instead of hardcoding details (e.g., **DB port, username, password**), store them in a Config Map.
    - This allows **easy updates** without modifying application code.
- **How Config Maps Work**:
    - Kubernetes retrieves Config Map data as **environment variables** or **files within the container**.
    - Keeps configuration **separate** from application code for better **maintainability**.

3️⃣ Live Demo: Creating a Config Map

1. **Create a YAML file (`cm.yaml`)** defining the Config Map:

```
apiVersion: v1
kind: ConfigMap
metadata:
  name: my-config
data:
  db_port: "5432"
  db_username: "admin"
```

2. **Apply the Config Map in Kubernetes**:

```
kubectl apply -f cm.yaml
```

3. **Retrieve and describe the Config Map**:

```
kubectl get configmap my-config
kubectl describe configmap my-config
```

4. **Use Config Map in a Deployment**:
    - Reference it as **environment variables** in a Kubernetes deployment.

Introduction to Secrets

- **Definition**: Similar to Config Maps but designed for **storing sensitive data** (e.g., passwords, API keys).
- **Security Features**:
  - Stored **encrypted in etcd** to prevent unauthorized access.
  - **Access control via RBAC** ensures only authorized components can use them.
- **Why Use Secrets?**
  - **Prevents exposure** of sensitive information.
  - Ensures **security best practices** in Kubernetes applications.

5️⃣ Differences: Config Maps vs. Secrets

Feature	Config Maps 🛠️	Secrets 🔐
Purpose	Stores **non-sensitive** configuration data.	Stores **sensitive** information (passwords, API keys).
Security	Data is **not encrypted**.	Data is **encrypted at rest** in etcd.
Access Control	Accessible to all by default.	**RBAC policies** restrict access.
Storage	Stored as plain text.	Stored as **base64-encoded**, but not encrypted by default.

6️⃣ Live Demo: Creating a Secret

1. **Create a Secret from CLI**:

```
kubectl create secret generic my-secret --from-literal=db_password=mysecurepassword
```

2. **Verify Secret**:

```
kubectl get secret my-secret
kubectl describe secret my-secret
```

3. **Base64 Encoding**:
   - Secrets are stored in **base64 encoding** (not encryption).
   - To decode:

```
echo "bXlzZWN1cmVwYXNzd29yZA==" | base64 --decode
```

4. **Use Secret in a Pod**:
    - Mount as **environment variables** or **files inside a container**.
