4 hours

1. Overview of Container Orchestration

Information

- Container orchestration is the process of managing and coordinating multiple containers to work together seamlessly.
- It involves automating the deployment, scaling, and management of containerized applications.
- Container orchestration platforms provide a robust framework for running containers in a distributed environment and ensure that applications are highly available, scalable, and resilient.
- Some of the popular container orchestration platforms include Kubernetes, Docker Swarm, and Apache Mesos.
- These platforms offer various features and capabilities to simplify the management of containerized applications.

Key concepts in container orchestration include:

1. Container Deployment:
    a. Container orchestration platforms facilitate the deployment of containers across a cluster of machines or nodes.
    b. They handle the scheduling and placement of containers based on resource availability and constraints.

2. Scaling:
    a. Container orchestration platforms allow scaling of containers based on application demand.
    b. They can automatically add or remove containers to meet the desired performance and resource requirements.

3. Service Discovery and Load Balancing:
    a. With container orchestration, applications can be accessed through a single endpoint or service name.
    b. The orchestration platform handles the routing of incoming traffic to the appropriate containers, ensuring load balancing and high availability.

4. Health Monitoring and Self-healing:
    a. Orchestration platforms continuously monitor the health of containers and automatically restart or replace any failed or unhealthy containers.
    b. This ensures that applications are resilient and maintain high availability.

5. Configuration and Secret Management:
    a. Container orchestration platforms provide mechanisms for managing configurations and secrets used by containers.
    b. They enable the centralized management and secure storage of sensitive information such as passwords, API keys, and certificates.

6. Rolling Updates and Rollbacks:

a. Orchestration platforms allow for seamless deployment of new container versions by performing rolling updates.
b. This means that containers are updated one by one, minimizing downtime.
c. In case of issues, rollbacks can be easily performed to revert to the previous working version.

7. Persistent Storage:
   a. Container orchestration platforms provide mechanisms to manage and allocate persistent storage for stateful applications.
   b. This ensures that data is preserved even if containers are restarted or replaced.

- Container orchestration brings several benefits to application deployment and management, including improved scalability, fault tolerance, resource utilization, and simplified deployment processes.
- It allows organizations to efficiently manage large-scale containerized deployments and leverage the full potential of container technologies.

2. Different between Docker Swarm and Kubernetes Cluster

- Docker Swarm and Kubernetes are both popular container orchestration platforms, but they have some key differences in terms of their architecture and approach to container management.

Docker Swarm:

- Docker Swarm is a native clustering and orchestration solution for Docker containers.
- It follows a simpler architecture compared to Kubernetes, making it easier to set up and manage for smaller deployments.
- It uses the concept of a swarm, which consists of one or more manager nodes and multiple worker nodes.
- Manager nodes handle orchestration and control plane tasks, while worker nodes run the containerized applications.
- Docker Swarm uses a declarative approach for defining services and tasks, making it easy to deploy and scale applications.
- It has built-in support for Docker Compose, allowing you to define multi-container applications using a single YAML file.

Kubernetes:

- Kubernetes is a highly scalable and flexible container orchestration platform developed by Google.
- It has a more complex architecture that allows for managing large-scale containerized applications.
- Kubernetes architecture is based on a master-worker model.
- The master node contains the control plane components, including the API server, scheduler, and

controller manager. It manages the overall cluster and orchestrates container deployments.

- Worker nodes, also known as minions, host the containerized applications and run the Kubernetes agent called kubelet.
- Kubernetes uses a declarative model with manifest files called YAML or JSON, which define the desired state of the cluster, including pods, services, and deployments.
- It provides advanced features like automatic scaling, rolling updates, and health monitoring.
- Kubernetes has a rich ecosystem of plugins and extensions, offering more flexibility and customization options.
- In summary, Docker Swarm is a simpler and more lightweight solution suitable for smaller deployments, while Kubernetes offers a more robust and scalable platform for managing large-scale containerized applications.
- The choice between the two depends on the specific needs and scale of your container environment.
- Regarding the second part of your question, "Kubernetes Architecture," it would require a more detailed explanation.
- The architecture of Kubernetes is based on several key components such as the API server, etcd, controller manager, scheduler, kubelet, and kube-proxy.
- These components work together to manage and orchestrate containerized applications within a Kubernetes cluster.

3. Kubernetes Architecture

- Kubernetes is an open-source container orchestration platform that automates the deployment, scaling, and management of containerized applications.
- It follows a client-server architecture with various components working together to provide a scalable and resilient platform for running applications.
- Let's explore the architecture of Kubernetes:

Master Node:

1. Control Plane: The control plane manages and controls the entire Kubernetes cluster.
2. API Server: The API server is the central management point for all cluster operations. It exposes the Kubernetes API, which allows clients to interact with the cluster.
3. etcd: etcd is a distributed key-value store that stores the configuration and state of the cluster.
4. Scheduler: The scheduler assigns pods to nodes based on resource availability and other constraints.
5. Controller Manager: The controller manager oversees various controllers that handle cluster-wide functions such as replication, scaling, and managing node health.

Worker Nodes:

1. Node: A node, also known as a worker node, is a physical or virtual machine where containers are deployed and run.
2. Kubelet: The kubelet is an agent that runs on each node and communicates with the master node. It manages the containers and ensures they are running and healthy.
3. Container Runtime: The container runtime, such as Docker, is responsible for pulling container images and running containers on the node.
4. Kube Proxy: The kube-proxy is responsible for network proxying and load balancing between services in the cluster.

Networking:

1. Cluster Network: The cluster network provides communication between the nodes and pods within the cluster.
2. Service Network: The service network enables communication between services in the cluster.
3. Ingress: Ingress allows external traffic to access services within the cluster.

Volumes and Storage:

1. Persistent Volumes: Persistent volumes provide persistent storage for stateful applications running in the cluster.

2. Storage Classes: Storage classes define different storage configurations and allow dynamic provisioning of persistent volumes.

Add-ons:

1. DNS: The DNS add-on provides DNS-based service discovery within the cluster.
2. Dashboard: The Kubernetes dashboard is a web-based graphical interface for managing and monitoring the cluster.
3. Metrics Server: The metrics server collects resource usage data from nodes and pods for monitoring purposes.

- The architecture of Kubernetes is designed to be highly available, scalable, and resilient.
- It provides mechanisms for automatic scaling, load balancing, and self-healing to ensure the smooth operation of containerized applications.
- Note that the actual architecture may vary depending on the Kubernetes deployment model and configuration choices.

4. Installation of Kubernetes – Minikube and EKS

Installation steps for both Minikube and Amazon Elastic Kubernetes Service (EKS):

Minikube Installation:

- Minikube is a tool that allows you to run a single-node Kubernetes cluster on your local machine for development and testing purposes.
- Make sure you have a hypervisor installed on your machine (VirtualBox, VMware, or Hyper-V).
- Download and install Minikube by following the instructions specific to your operating system from the Minikube GitHub repository.
- Once installed, open a terminal or command prompt and start Minikube by running the command minikube start.
- Minikube will create a single-node Kubernetes cluster on your local machine.

Amazon EKS Installation:

- Amazon Elastic Kubernetes Service (EKS) is a managed Kubernetes service provided by AWS.
- Sign in to the AWS Management Console and navigate to the EKS service.
- Create an EKS cluster by following the provided steps. This will involve configuring cluster details, networking, and authentication.
- Once the cluster is created, you can use the AWS CLI or other tools like kubectl to interact with the cluster.
- Install and configure the AWS CLI on your local machine by following the AWS CLI documentation.
- Set up the kubeconfig file to connect to your EKS cluster by running the command aws eks update-kubeconfig --name <cluster-name> --region <region>.

- After completing the installation steps, you should have Minikube running a local Kubernetes cluster, and you should have an EKS cluster set up in your AWS account.
- It's worth noting that Minikube is primarily used for local development and testing, while EKS is a production-ready managed Kubernetes service in the AWS cloud.
- The choice between Minikube and EKS depends on your specific use case and requirements.

5. Kubernetes Nodes

- In Kubernetes, a node is a worker machine that runs containerized applications.
- It is part of the Kubernetes cluster and is responsible for running the actual workloads.

Let's explore some key points about Kubernetes nodes:

1. Definition: A node, also known as a worker node or minion, is a physical or virtual machine that is registered with the Kubernetes cluster. It is where containers are scheduled and executed.

2. Roles and Components: Each node in a Kubernetes cluster plays different roles and includes several components:

3. Kubelet: It is an agent that runs on each node and communicates with the Kubernetes control plane. It is responsible for managing the containers on the node.

4. Container Runtime: It is the software responsible for running containers, such as Docker, containerd, or CRI-O.

5. Kube-proxy: It is responsible for network routing and load balancing between different pods.

6. Node Status: Nodes have a status that indicates whether they are ready to accept workloads. The status is updated by the kubelet and can be observed by the Kubernetes control plane.

7. Node Capacity: Each node in a Kubernetes cluster has a certain amount of resources available, such as CPU, memory, and storage. These resources are collectively referred to as the node's capacity. The Kubernetes scheduler uses this information to determine where to place workloads within the cluster.

8. Node Labels and Selectors: Nodes can be labeled with custom key-value pairs to add metadata or categorize nodes based on their characteristics. These labels can be used as selectors in pod specifications to specify which nodes the pods should be scheduled on.

9. Node Management: Nodes can be added or removed from the cluster dynamically. Kubernetes provides

mechanisms for scaling the number of nodes based on workload demands. Nodes can also be cordoned or drained to gracefully stop scheduling new pods on them or evict the existing pods, respectively.

- Understanding Kubernetes nodes is essential for effectively managing and scaling your cluster.
- By leveraging the resources and capabilities of nodes, Kubernetes ensures that your containerized applications are deployed and executed efficiently across the cluster.

6. Kubernetes Pods

- In Kubernetes, a pod is the smallest and simplest unit of deployment. It represents a single instance of a running process or a set of co-located processes.

Let's explore some key points about Kubernetes pods:

1. Definition: A pod is a logical group of one or more containers that are tightly coupled and share the same resources, such as network namespace, IP address, and storage volumes. Pods are the basic building blocks of applications in Kubernetes.

2. Single-Container and Multi-Container Pods: A pod can contain a single container or multiple containers that work together and share the same resources. Multi-container pods are often used for closely

related processes that need to run together and communicate with each other.

3. Pod Lifecycle: Pods have a lifecycle that includes several phases:

4. Pending: The pod is being created, and the necessary resources are being allocated.

5. Running: The pod has been scheduled to a node, and the container(s) are running.

6. Succeeded: All containers in the pod have completed successfully and terminated.

7. Failed: At least one container in the pod has terminated with an error.

8. Unknown: The state of the pod could not be obtained.

9. Pod Networking: Each pod in a Kubernetes cluster has a unique IP address and is able to communicate with other pods in the same cluster. Pods can also expose network services to other pods or external clients using services.

10. Pod Scheduling: Pods are scheduled to run on specific nodes in the cluster based on resource requirements, node availability, and any scheduling constraints defined. The Kubernetes scheduler ensures that pods are evenly distributed across nodes.

11. Pod Affinity and Anti-Affinity: Pod affinity and anti-affinity are mechanisms used to influence pod scheduling. Pod affinity ensures that pods are scheduled on nodes with specific characteristics or labels, while pod anti-affinity prevents pods from being scheduled on nodes with specific characteristics or labels.

12. Pod Autoscaling: Kubernetes provides features for scaling pods based on CPU or custom metrics. Horizontal Pod Autoscaling (HPA) automatically scales the number of pod replicas based on resource utilization, ensuring optimal performance and resource allocation.

- Pods are designed to be ephemeral and can be created, terminated, and replaced dynamically.
- They provide a convenient abstraction for managing containers and enable easy scaling, load balancing, and deployment of containerized applications in Kubernetes.

7. Kubernetes Deployments

- In Kubernetes, a Deployment is a higher-level resource that manages and controls the lifecycle of multiple pods.
- Deployments are used to ensure that a specific number of replicas of an application are running and

to manage the rolling updates and rollbacks of the application.

Here are some key points about Kubernetes Deployments:

1. Replica Sets: Deployments internally use Replica Sets, which are responsible for creating and managing the actual pods. The Replica Set ensures that the desired number of replicas (pods) are running at all times, and it automatically replaces any failed or terminated pods.

2. Desired State: Deployments define the desired state of the application, including the number of replicas, the container image version, and other configuration details. Kubernetes continuously monitors the current state of the application and takes actions to maintain the desired state.

3. Rolling Updates: Deployments support rolling updates, which allow for updating the application while minimizing downtime. During a rolling update, the Deployment gradually replaces old pods with new ones, ensuring that the application remains available and the update is seamless.

4. Rollbacks: In case of issues or errors with a new version of the application, Deployments support rollbacks to a previous known good state. Kubernetes can automatically roll back the Deployment to the

previous version, ensuring that the application is quickly restored to a stable state.

5. Scaling: Deployments can be scaled up or down by adjusting the number of replicas. Scaling can be done manually or automatically based on defined metrics using Horizontal Pod Autoscaling (HPA).

6. Service Discovery and Load Balancing: Deployments are typically associated with a Service, which provides a stable network endpoint for accessing the application. Services enable service discovery and load balancing across the pods of the Deployment.

7. Declarative Configuration: Deployments are defined using declarative configuration files, typically written in YAML or JSON. This configuration specifies the desired state of the Deployment, including the container image, resource requirements, environment variables, and other settings.

- Deployments are a powerful mechanism for managing the lifecycle of applications in Kubernetes.
- They provide a declarative and scalable approach to deploying and managing containerized applications, enabling easy updates, rollbacks, and scaling operations.

8. Rolling updates and rollbacks

- Rolling updates and rollbacks are essential features of Kubernetes Deployments that enable seamless application updates and the ability to revert to a previous stable state if needed.

Here's an explanation of rolling updates and rollbacks in Kubernetes:

1. Rolling Updates:

- Rolling updates allow you to update your application without incurring downtime or disruptions to your users.
- During a rolling update, Kubernetes gradually replaces old pods with new ones, ensuring that the application remains available and responsive throughout the update process.
- By default, Kubernetes deploys a new version of the application alongside the existing version and gradually shifts traffic to the new version.
- This gradual transition allows for smooth updates and gives Kubernetes the ability to monitor the health of the new pods and automatically roll back if any issues are detected.
- Rolling updates can be performed manually by specifying the new version of the application in the Deployment configuration or automatically triggered based on defined criteria.

2. Rollbacks:

- Rollbacks come into play when an issue or error is detected in a new version of the application.
- Kubernetes provides an automatic rollback mechanism to revert the Deployment to a previous known good state.
- If an issue is detected during a rolling update, Kubernetes can automatically halt the update process and roll back to the previous version.
- During a rollback, Kubernetes replaces the new pods with the previous version's pods, ensuring a seamless transition back to the stable state.
- Rollbacks can also be initiated manually, allowing you to manually revert to a previous version of the application using the Kubernetes command-line interface (kubectl) or through the Kubernetes API.
- Rolling updates and rollbacks are important features of Kubernetes Deployments that promote reliability and availability during application updates. They help minimize downtime and provide a safety net in case issues arise during the update process.

9. Scaling up and down of the application kubernetes

- Scaling up and down of applications in Kubernetes is a crucial aspect of managing the workload and ensuring optimal resource utilization.

Here's an explanation of scaling up and down in Kubernetes:

1. Scaling Up:

- Scaling up refers to increasing the number of instances (pods) of an application to handle higher traffic or increased demand.
- In Kubernetes, scaling up is achieved by adjusting the replica count of the Deployment or ReplicaSet that manages the application.
- You can manually scale up by updating the replica count in the Deployment or by using the kubectl scale command.
- Kubernetes will create new pod instances based on the desired replica count, ensuring that the application can handle increased traffic or workload.
- Scaling up distributes the load across multiple instances, improving the application's performance and capacity.

2. Scaling Down:

- Scaling down refers to reducing the number of instances (pods) of an application when the demand or traffic decreases.
- Similarly to scaling up, scaling down is done by adjusting the replica count of the Deployment or ReplicaSet.
- You can manually scale down by updating the replica count or using the kubectl scale command.
- Kubernetes will gradually terminate excess pod instances until the desired replica count is reached.

- Scaling down helps conserve resources and optimizes resource utilization, especially during periods of lower demand.

3. Autoscaling:

- Kubernetes also supports autoscaling, which allows you to automatically scale the number of pod instances based on defined metrics or criteria.
- Horizontal Pod Autoscaler (HPA) is a Kubernetes feature that automatically adjusts the replica count based on CPU utilization or custom metrics.
- With HPA, you can set minimum and maximum replica counts and define thresholds for scaling up or down based on resource usage.
- Autoscaling ensures that your application dynamically scales based on actual demand, providing efficient resource allocation.
- Scaling up and down in Kubernetes is essential for maintaining application performance, availability, and cost optimization.

- By adjusting the replica count or utilizing autoscaling capabilities, you can efficiently manage your application's workload based on demand fluctuations.

10. Services in Kubernetes

- In Kubernetes, Services provide a way to expose and access applications running inside pods.

- A Service acts as an abstraction layer that enables communication between different parts of an application or between different applications.

Here's an explanation of Services in Kubernetes:

1. Service Types:

a. ClusterIP: The default type, which assigns a stable internal IP address to the Service within the cluster. This type allows communication between different pods within the cluster.

b. NodePort: Exposes the Service on a static port on each cluster node. This type allows external access to the Service using the node's IP address and the assigned port.

c. LoadBalancer: Creates an external load balancer that automatically routes traffic to the Service. This type is typically used in cloud environments that support external load balancers.

d. ExternalName: Maps the Service to a DNS name outside the cluster. This type is useful when you want to provide access to an external service without creating an actual Service within the cluster.

2. Service Discovery:

a. Services have a DNS name that other pods can use to access them. Each Service gets its own DNS entry in the cluster's internal DNS server, allowing other pods to discover and communicate with the Service using its DNS name.

b. Service discovery enables decoupling between applications and allows for dynamic scaling and replacement of pods without impacting the connectivity between them.

3. Load Balancing:

a. Services distribute incoming traffic across all the pods associated with the Service, providing load balancing functionality.

b. Load balancing ensures that the traffic is evenly distributed, improving the availability and performance of the application.

4. Headless Services:

a. Headless Services disable the default load balancing and DNS resolution behavior of Services. Instead, they allow direct communication with individual pods within the Service using their IP addresses.

b. Headless Services are useful when you need to access specific pods or when implementing advanced networking features like stateful applications.

- Services play a vital role in enabling connectivity, load balancing, and service discovery within a Kubernetes cluster.
- They provide a stable network endpoint for accessing and communicating with applications running inside pods.
- By utilizing different types of Services, you can expose your applications to both internal and external traffic as needed.

11. Practical Includes:

i. Installation and configuration of Kubernetes Minikube
ii. Creation of Pods and Deployments using ad-hoc Commands
iii. Creation of Pods and Deployments using YAML files
iv. Scaling up and Scaling Down of the application
v. Rolling out Deployments and Rolling Back
vi. Creation of Services

1. Installation and Configuration of Kubernetes Minikube:

- Install Minikube on your local machine following the official documentation for your operating system.
- Start Minikube by running the minikube start command.

- Verify that Minikube is running by executing minikube status.

2. Creation of Pods and Deployments using ad-hoc Commands:

- Create a Pod using an ad-hoc command: kubectl run my-pod --image=<image_name>
- Verify that the Pod is running: kubectl get pods
- Create a Deployment using an ad-hoc command: kubectl create deployment my-deployment --image=<image_name>
- Verify that the Deployment is running: kubectl get deployments

3. Creation of Pods and Deployments using YAML files:

- Create a Pod YAML file, e.g., pod.yaml, with the desired specifications.
- Apply the Pod configuration: kubectl apply -f pod.yaml
- Create a Deployment YAML file, e.g., deployment.yaml, with the desired specifications.
- Apply the Deployment configuration: kubectl apply -f deployment.yaml

4. Scaling up and Scaling Down of the application:

- Scale a Deployment by increasing the number of replicas: kubectl scale deployment my-deployment --replicas=<new_replica_count>

- Verify that the Deployment has scaled: kubectl get deployments

5. Rolling out Deployments and Rolling Back:

- Update a Deployment with a new version of the application: kubectl set image deployment/my-deployment <container_name>=<new_image>

- Verify that the Deployment is rolling out: kubectl rollout status deployment/my-deployment

- Rollback the Deployment to a previous revision: kubectl rollout undo deployment/my-deployment

- Verify that the Deployment has rolled back: kubectl rollout status deployment/my-deployment

6. Creation of Services:

- Create a Service YAML file, e.g., service.yaml, with the desired specifications.

- Apply the Service configuration: kubectl apply -f service.yaml
- Verify that the Service is created: kubectl get services
- Remember to replace <image_name>, <new_replica_count>, <new_image>, and <container_name> with appropriate values in the commands and YAML files.