Kubernetes Assignment 1

1. What is a Kubernetes cluster?

- A Kubernetes cluster is a set of nodes that run containerized applications. Containerizing applications packages an app with its dependences and some necessary services. They are more lightweight and flexible than virtual machines. In this way, Kubernetes clusters allow for applications to be more easily developed, moved and managed.
- Kubernetes clusters allow containers to run across multiple machines and environments: virtual, physical, cloud-based, and on-premises. Kubernetes containers are not restricted to a specific operating system, unlike virtual machines. Instead, they are able to share operating systems and run anywhere.
- Kubernetes clusters are comprised of one master node and a number of worker nodes. These nodes can either be physical computers or virtual machines, depending on the cluster.

2. What are the different parts of the Kubernetes architecture?

Nodes (Minions):

A node is a machine either physical or virtual machine on which Kubernetes is installed. A node is a worker machine and this is where containers inside the pods will be launched by Kubernetes.

Cluster:

If our a node fails, our app will be down. So a cluster is a set of nodes grouped together. Even if one node fails, your application will still be accessible from the other nodes. In addition, having multiple nodes helps in sharing the computational load as well.

A Kubernetes cluster consists of one or more nodes managed by Kubernetes. The nodes are bare-metal servers, on-premises VMs, or VMs on a cloud provider. Every node contains a container runtime (for example a Docker Engine), Kubelet (responsible for starting, stopping, and managing individual containers by requests from the Kubernetes control plane), and kube-proxy (responsible for networking and load balancing).

Master Node:

Who is responsible for managing the cluster? Where is the information about the members of the cluster stored? How are the nodes monitored? When a node fails, how do you move the workload of the failed node to another worker node? — Here comes Master Node

A Kubernetes cluster also contains one or more master nodes that run the Kubernetes control plane. The control plane consists of different processes, such as an API server (provides JSON over HTTP API), scheduler (selects nodes to run containers), controller manager (runs controllers — see below), and etcd (a globally available configuration store).

Pod:

A pod is the smallest deployable unit that can be managed by Kubernetes. A pod is a logical group of one or more containers that share the same IP address and port space. The main purpose of a pod is to support co-located processes, such as an application server and its local cache.

Containers within a pod can find each other via localhost and can also communicate with each other using standard inter-process communications like SystemV, semaphores, or POSIX shared memory. In other words, a pod represents a "logical host". Pods are not durable; they will not survive scheduling failures or node failures. If a node where the pod is running dies, the pod is deleted. It can then be replaced by an identical pod, with even the same name, but with a new unique identifier (UID).

Note: When you install Kubernetes on a System, you are actually installing the following components: an API Server, an ETCD service, a kubelet service, a Container Runtime, Controllers. and Schedulers.

Master Node Components

1. API Server 2. Controller Manager 3. ETCD 4. Scheduler (Not in below pic)

Kube-APIServer:

APIs allow applications to communicate with one another. There is a component on the master that exposes the Kubernetes API. It is the front-end for the Kubernetes control plane. It is designed to scale horizontally — that is, it scales by deploying more instances. The users, management

devices, and command line interfaces all talk to the API server to interact with the Kubernetes cluster.

Kube-Scheduler:

It is a component on the master node that watches newly created pods that have no node assigned and selects a node for them to run on.

Factors taken into account for scheduling decisions include individual and collective resource requirements, hardware/software/policy constraints, affinity and anti-affinity specifications, data locality, inter-workload interference, and deadlines.

Kube-Controller-manager:

This is a component on the master that runs controllers.

Logically, each controller is a separate process, but to reduce complexity, they are all compiled into a single binary and run in a single process.

These controllers include:

Node Controller: Responsible for noticing and responding when nodes go down.

Replication Controller: Responsible for maintaining the correct number of pods for every replication controller object in the system.

Endpoints Controller: Populates the Endpoints object (that is, it joins Services and Pods).

Service Account and Token Controllers: Create default accounts and API access tokens for new namespaces.

Cloud-Controller-Manager: Cloud-controller-manager runs controllers that interact with the underlying cloud providers. The cloud-controller-manager binary is an alpha feature introduced in Kubernetes release 1.6. Cloud-

controller-manager runs cloud-provider-specific controller loops only. You must disable these controller loops in the Kube-controller-manager. You can disable the controller loops by setting the --cloud-provider flag to external when starting the Kube-controller-manager.

Node Controller: For checking the cloud provider to determine if a node has been deleted in the cloud after it stops responding.

Route Controller: For setting up routes in the underlying cloud infrastructure.

Service Controller: For creating, updating, and deleting cloud provider load balancers.

Volume Controller: For creating, attaching, and mounting volumes, and interacting with the cloud provider to orchestrate volumes.

ETCD: It stores the configuration information which can be used by each of the nodes in the cluster. It is a high availability key-value store that can be distributed among multiple nodes. It is accessible only by Kubernetes API server as it may have some sensitive information. It is a distributed key-value store which is accessible to all.

ETCD is a distributed reliable key-value store used by Kubernetes to store all data used to manage the cluster. Think of it this way, when you have multiple nodes and multiple masters in your cluster, etcd stores all that information on all the nodes in the cluster in a distributed manner. ETCD is responsible for implementing locks within the cluster to ensure there are no conflicts between the Masters.

Scheduler: This is one of the key components of Kubernetes master. It is a service in master responsible for distributing the workload or containers across multiple nodes. It is responsible for tracking the utilization of the working load on cluster nodes and then placing the workload on which resources are available and accept the workload. In other words, this is the mechanism responsible for allocating pods to available nodes. The scheduler is responsible for workload utilization and allocating pod to a new node. It looks for newly created containers and assigns them to Nodes.

Kubernetes Node Components

Node components run on every node, maintaining running pods and providing the Kubernetes runtime environment.

Docker:

The first requirement of each node is Docker which helps in running the encapsulated application containers in a relatively isolated but lightweight operating environment.

Container Runtime:

The container runtime is the underlying software that is used to run containers. In our case, it happens to be Docker. The container runtime is the software that is responsible for running containers. Kubernetes supports several runtimes: Docker, containerd, cri-o, rktlet, and any implementation of the Kubernetes CRI (Container Runtime Interface).

Kubelet:

Kubelet is the agent that runs on each node in the cluster. The agent is responsible for making sure that the containers are running on the nodes as expected.

It's a small service in each node responsible for relaying information to and from control plane service. It interacts with etcd store to read configuration details and write values. This communicates with the master component to receive commands and work. The kubelet process then assumes responsibility for maintaining the state of work and the node server. It manages network rules, port forwarding, etc.

The kubelet takes a set of PodSpecs that are provided through various mechanisms and ensures that the containers described in those PodSpecs are running and healthy. The kubelet doesn't manage containers which were not created by Kubernetes.

Kubernetes Proxy Service:

This is a proxy service which runs on each node and helps in making services available to the external host. It helps in forwarding the request to correct containers and is capable of performing primitive load balancing. It makes sure that the networking environment is predictable and accessible and at the same time it is isolated as well. It manages pods on node, volumes, secrets, creating new containers' health checkup, etc.

Hit the clap button if you like :)

3. What exactly do you mean by "container orchestration"?

Container orchestration is the automation of much of the operational effort required to run containerized workloads and services. This includes a wide range of things software teams need to manage a container's lifecycle, including provisioning, deployment, scaling (up and down), networking, load balancing and more.

Container orchestration is key to working with containers, and it allows organizations to unlock their full benefits. It also offers its own benefits for a containerized environment, including:

Simplified operations: This is the most important benefit of container orchestration and the main reason for its adoption. Containers introduce a large amount of complexity that can quickly get out of control without container orchestration to manage it.

Resilience: Container orchestration tools can automatically restart or scale a container or cluster, boosting resilience.

Added security: Container orchestration's automated approach helps keep containerized applications secure by reducing or eliminating the chance of human error.

4. What are the various features of Kubernetes?

Here are the essential Kubernetes features:

- Automated Scheduling
- Self-Healing Capabilities
- Automated rollouts & rollback
- Horizontal Scaling & Load Balancing

- Offers environment consistency for development, testing, and production
- Infrastructure is loosely coupled to each component can act as a separate unit
- Provides a higher density of resource utilization
- Offers enterprise-ready features
- Application-centric management
- Auto-scalable infrastructure
- You can create predictable infrastructure

5. Explain the relationship between Kubernetes and Docker?

Even though Docker and Kubernetes are separate technologies, they actually complement each other and work great together. In fact, they have a symbiotic relationship.

Docker is at the core of containerization technology—it allows you to create and deploy application containers. If your application is still simple, Docker has the essential infrastructure for managing its lifecycle.

As your containerized application becomes bigger and more complex, possibly needing multiple clusters and more sophisticated management, Kubernetes becomes a handy tool. It offers a useful orchestration platform for your Docker containers. Kubernetes does not create containers; it actually requires a container tool to run, of which Docker is the most popular option.

So Docker vs. Kubernetes actually points to their ability to work together to realize the promise of the containerization technology—code once and run anywhere, regardless of the scale.

Using Kubernetes with Docker results in the following benefits:

- It enhances the robustness of your infrastructure. Your applications are more highly available.
- It improves the scalability of your applications. You can easily spin up your applications to handle more load on demand, potentially lowering resource wastages and enhancing user experience.
- Because apps are broken down into smaller constituents, they are easier to maintain.