1. What is k8s and Benefits of Using Kubernetes?

K8s, or KuberneteS, is an open-source container orchestration platform designed to automate the deployment, scaling, and management of containerized applications

Benifites:

1. Automated deployment and management:

If you are using Kubernetes for deploying the application then no need for manual intervention kubernetes will take care of everything like automating the deployment, scaling, and containerizing the application

1. Scalability

You can scale the application containers depending on the incoming traffic Kubernetes offers Horizontal pod scaling the pods will be scaled automatically depending on the load.

1. High availability

You can achieve high availability for your application with the help of Kubernetes and also it will reduce the latency issues for the end users.

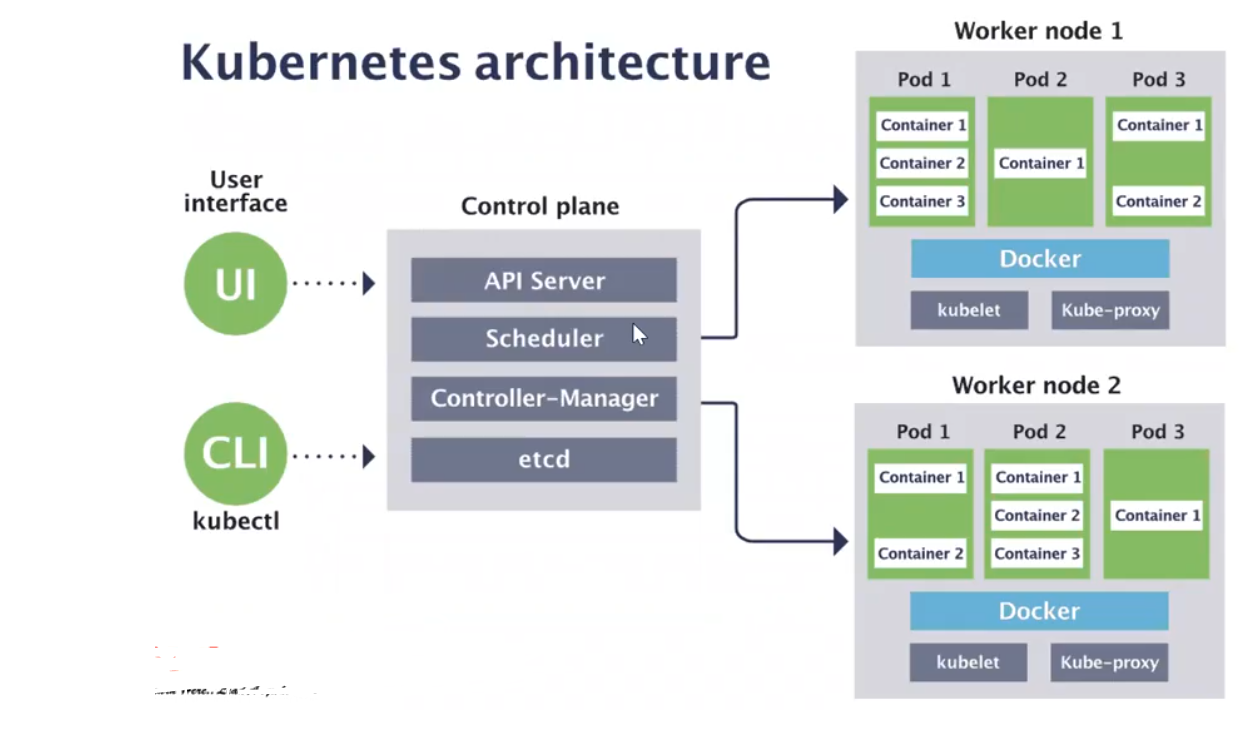
1. Cost-effectiveness

If there is unnecessary use of infrastructure the cost will also increase kubernetes will help you to reduce resource utilization and control the overprovisioning of infrastructure.

1. Improved developer productivity

Developer can concentrate more on the developing part kubernetes will reduce the efforts of deploying the application.

1. **Difference Between K8S and Docker Swarm:**
2. K8s architecture?



kubectl writes to the API Server.

API Server validates the request and persists it to etcd.

etcd notifies back the API Server.

API Server invokes the Scheduler.

Scheduler decides where to run the pod on and return that to the API Server.

API Server persists it to etcd.

etcd notifies back the API Server.

API Server invokes the Kubelet in the corresponding node.

Kubelet talks to the Docker daemon using the API over the Docker socket to create the container.

Kubelet updates the pod status to the API Server.

API Server persists the new state in etcd.

**Control plane:** The control plane node is responsible for the overall control and management of the Kubernetes cluster. It consists of several key components:

1. **API Server:** The API Server is responsible for exposing the Kubernetes API and serving as the entry point for managing the cluster. It plays a pivotal role in the communication and co-ordination of all other components within the cluster.

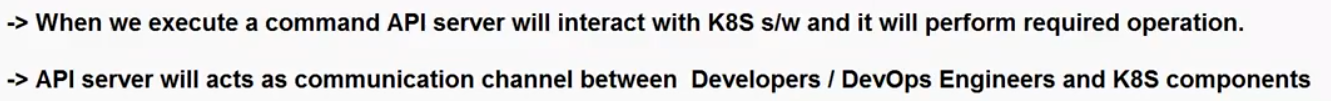
Here are some key aspects and functions of the API Server:

1. **Exposes Kubernetes API:** The API Server provides a RESTful HTTP API that allows users, administrators, and external systems to interact with the Kubernetes cluster.

This API is the primary means of communication for performing actions like creating and updating resources (e.g., pods, services, deployments), querying cluster state, and managing the cluster's configuration.

1. **Authentication and Authorization:** The API Server handles authentication and authorization for incoming API requests. It verifies the identity of users or processes making requests and enforces access control policies to ensure that only authorized actions are allowed.
2. **Validation and Admission Control:** The API Server performs validation checks on incoming requests to ensure they conform to Kubernetes resource schemas and policies. Admission controllers can be configured to enforce additional rules and policies before resources are created or updated in the cluster.
3. **Synchronization with etcd:** The API Server interacts with the etcd key-value store, which serves as the cluster's data store. It stores and retrieves information about the cluster's desired state and actual state. When an API request is made to change the cluster's state.
4. **Proxy and Load Balancing:** In multi-node clusters, multiple API Servers may be deployed for high availability and load distribution. The API Server instances are typically fronted by a load balancer or proxy to distribute incoming requests evenly across them.
5. **Resource Endpoints:** The API Server exposes endpoints for various Kubernetes resources, such as pods, services, config maps, secrets, and more. Users interact with these endpoints to create, update, or query the state of these resources.
6. **Event Handling:** The API Server generates events to communicate changes and updates within the cluster. Users can subscribe to these events to monitor the cluster's state and respond to changes as needed.

**In summary, the API Server in Kubernetes serves as the primary interface for users and automation tools to interact with the cluster. It enforces authentication, authorization, and validation rules, maintains cluster state, and plays a crucial role in maintaining the desired configuration and orchestrating changes within the cluster.**



1. **What is ETCD:**

* In Kubernetes, etcd is a distributed key-value store that serves as the primary source of truth and the backend data store for the entire cluster.
* It is a critical component of the control plane, and it stores the configuration data, state information, and metadata about all resources and objects in the cluster.

Here are key aspects and functions of etcd in the Kubernetes control plane:

1. Consistent and Highly Available Data Store:

* etcd is designed to be highly available and consistent. It uses the Raft consensus algorithm to ensure that data stored across the cluster remains consistent, even in the presence of node failures or network partitions.
* This consistency is crucial for maintaining the integrity of the cluster's configuration and state.

1. Cluster Configuration:

* etcd stores information about the cluster's configuration, including details about nodes (both worker nodes and master nodes), network configuration, API server endpoints, and cluster-wide settings.

1. Resource State:

* All resource and object state information in Kubernetes is stored in etcd. This includes information about pods, services, nodes, deployments, replica sets, and more. Whenever a resource is created, updated, or deleted, the corresponding changes are reflected in etcd.

1. Configuration Data:

* etcd can store configuration data that is critical for the functioning of various Kubernetes components. This includes configuration information for the API server, controller manager, scheduler, and other components.]

1. **Schedular:**

* In Kubernetes (K8s), the Scheduler is a critical component of the control plane responsible for making decisions about where and when to place newly created pods onto available worker nodes in the cluster.
* Its primary role is to ensure that workloads are distributed efficiently across the cluster.

Here's an overview of the Scheduler and its role in Kubernetes:

1. Scheduling Decisions:

* When a user or controller creates a new pod specification, it does not specify which node in the cluster the pod should run on.
* Instead, the pod's scheduling requirements, such as CPU and memory requests, node affinity, and other constraints, are defined in its configuration.
* The Scheduler's job is to determine the best node for each pod based on these requirements.

1. Node Selection:

* The Scheduler evaluates the suitability of each node in the cluster by considering factors like available resources (CPU, memory), node capacity, and the already running pods on each node.
* It then selects the node that best meets the pod's requirements.

1. Prioritization:

* The Scheduler assigns each node a score, and nodes with higher scores are considered better fits for the pod.
* The scoring system allows administrators to implement custom policies and prioritize certain nodes over others, based on application requirements.

1. **Controller –Manager:**

* In Kubernetes (K8s), the Controller Manager responsible for managing various controllers that regulate the state of the cluster.
* Each controller focuses on ensuring that a specific desired state is maintained for resources and objects in the cluster. Here's an explanation of the Controller Manager and its role:

1. Controller:

* In Kubernetes, a controller is a control loop that continuously watches the desired state of a resource and takes corrective actions to make the current state match the desired state.
* Controllers are responsible for managing the lifecycle of various resources, such as pods, replica sets, deployments, services, and more.

1. Controller Manager:

* The Controller Manager is a Kubernetes control plane component that houses and manages multiple controllers.
* Each controller is responsible for one or more resource types, and they ensure that the desired configuration for those resources is maintained.

The key controllers managed by the Controller Manager include:

1. Replication Controller: Ensures that the desired number of pod replicas are running at all times. It automatically scales the number of replicas up or down to match the desired state.
2. Deployment Controller: Manages Deployments, which allow declarative updates to applications. It ensures that the specified number of replica sets are created and maintained as the application is updated or rolled back.
3. StatefulSet Controller: Manages StatefulSets, which are used for state full applications. It ensures ordered pod creation, scaling, and deletion while maintaining stable network identities and persistent storage.

stateful applications: Managing stateful applications can be more challenging than stateless ones due to the complexity of maintaining data integrity, ensuring high availability, and managing data backups and recovery. However, they are essential for many use cases where data persistence and consistency are critical

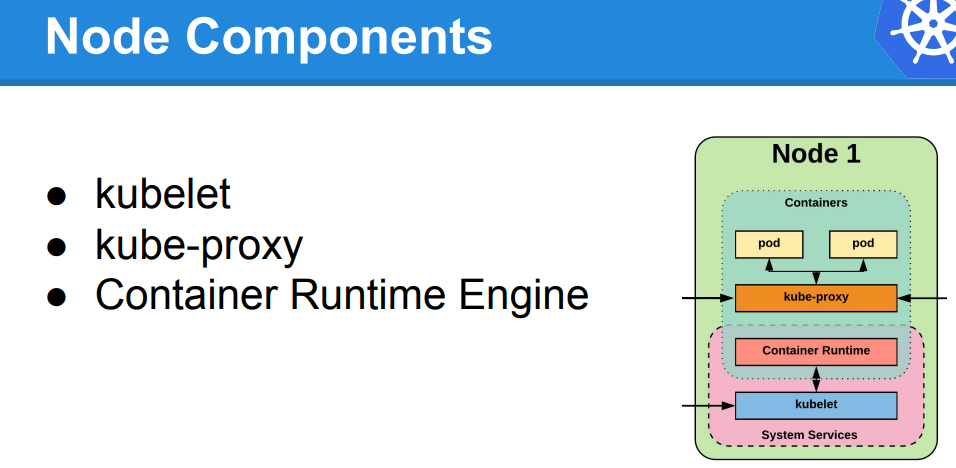
1. Job and CronJob Controllers: Manages Job and CronJob resources, which are used for batch processing and scheduled tasks, respectively.
2. Service Controller: Manages Kubernetes Services, which provide network access to a set of pods. It ensures that services are properly configured and that endpoints are updated as pods are created or destroyed.

**Worker Node**:

1. **Kubelet:**

* A node is a physical or virtual machine in the Kubernetes cluster where containers are deployed. Each node runs a container runtime (such as Docker) and a kubelet, which is an agent responsible for managing containers on that node.
* kubelet is a critical component of the cluster responsible for managing and maintaining individual nodes (also known as worker nodes or minions).
* It plays a key role in ensuring that containers are running as expected on each node.

Here's an overview of the kubelet and its responsibilities:



* Kubernetes creates a Node object internally (the representation). Kubernetes checks that a kubelet has registered to the API server that matches the metadata.name field of the Node. If the node is healthy (i.e. all necessary services are running), then it is eligible to run a Pod. Otherwise, that node is ignored for any cluster activity until it becomes healthy.
* Kubernetes keeps the object for the invalid Node and continues checking to see whether it becomes healthy.

1. Container Management:

* The kubelet is responsible for managing containers on its assigned node.
* It ensures that containers specified in Kubernetes Pod manifests are started, stopped, and maintained in the desired state.
* It communicates with the container runtime (e.g., Docker, containerd, or CRI-O) on the node to start and stop containers.

1. Pod Lifecycle Management:

* Pods are the smallest deployable units in Kubernetes, and each node can run multiple pods. The kubelet is responsible for ensuring the lifecycle of pods on its node.
* It monitors the pod's status, reports it to the Kubernetes control plane, and takes actions to recover pods if they fail.

1. Resource Monitoring and Reporting:

* The kubelet collects resource usage and performance data from containers running on its node, such as CPU and memory usage.
* This information is exposed via the Kubernetes API and can be used for monitoring, scaling, and resource allocation decisions.

1. Node Status Updates:

* The kubelet regularly reports the status of its node (e.g., node conditions, available resources, and capacity) to the Kubernetes control plane, allowing the scheduler to make informed decisions about where to place pods.

1. Image Management:

* It ensures that the required container images are pulled from container registries and available for use when starting pods that reference those images.

1. Health Checks:

* The kubelet performs periodic health checks on containers to detect and respond to any failures. If a container becomes unhealthy, the kubelet can restart or recreate the pod as needed.

1. Pod Eviction:

* In cases of resource constraints or node failures, the kubelet can evict (remove) pods from the node to free up resources or to ensure that critical pods are rescheduled onto healthy nodes.

1. Pod Security Policies:

* The kubelet enforces certain security policies at the node level, such as ensuring that containers run with specified security contexts or within specific namespaces.
* How can use kubectl to view a Node's status and other details:

#kubectl describe node <insert-node-name-here>

1. **Kube-proxy:**

* In Kubernetes (K8s), kube-proxy is a one of the component that runs on every worker node in the cluster. Its primary responsibility is to maintain network connectivity and provide basic load balancing for services within the cluster.
* kube-proxy ensures that network traffic is properly routed to the appropriate pods or endpoints.
* Service Abstraction:

Kube-proxy is responsible for implementing this service abstraction and ensuring that traffic to services is correctly load-balanced to the appropriate pods.

* Network Routing:
* Load Balancing:

When a service is exposed with type "LoadBalancer" or "NodePort," kube-proxy manages the load balancing of incoming traffic to the service.

* Kube-proxy plays a crucial role in ensuring that services within a Kubernetes cluster are accessible and that network traffic is correctly directed to the appropriate pods.

**POD:**

* In Kubernetes (K8s), a Pod is the smallest deployable unit and the fundamental building block for running containers.
* A Pod can contain one or more tightly coupled containerized applications that share the same network namespace, IP address, and storage volumes.
* Pods are used to deploy, manage, and scale applications in a K8s cluster.

Here are some key points to understand about Pods:

1. Atomic Unit: A Pod represents the smallest deployable entity in Kubernetes. It is often used to encapsulate a single container, but it can also host multiple containers that are tightly coupled and need to share resources.
2. Shared Network Namespace: All containers within the same Pod share the same network namespace, meaning they can communicate with each other using localhost. This makes it easy for containers in the same Pod to work together.
3. Shared Storage Volumes: Containers in a Pod can share storage volumes, allowing them to read and write data to the same filesystem. This facilitates data sharing and communication between containers.
4. Single IP Address: A Pod is assigned a single IP address within the cluster. This IP address is reachable from other Pods in the same cluster but is not directly reachable from outside the cluster without additional network configurations.
5. Pod Lifecycle: Pods have their own lifecycle. They can be created, started, stopped, and deleted. When a Pod is deleted, all the containers within it are also terminated.
6. Immutable: Pods are considered immutable. When changes are required, such as updating container images or configuration, a new Pod is created with the desired changes, and the old one is terminated.

The immutable objects are objects whose value can not be changed after initialization. We can not change anything once the object is created.

**Kubernetes Cluster Set-up:**

Setting up a Kubernetes cluster can be done in several ways, depending on your requirements and infrastructure.

1.Local Development Cluster: You can set up a local development cluster on your machine for testing and development purposes. Tools like Minikube, KinD (Kubernetes in Docker), and K3d (K3s in Docker) allow you to run a single-node or multi-node Kubernetes cluster on your local system.

2. Hosted Kubernetes Services: Many cloud providers offer managed Kubernetes services that make it easy to set up and manage a Kubernetes cluster. Examples include Google Kubernetes Engine (GKE), Amazon Elastic Kubernetes Service (EKS), Microsoft Azure Kubernetes Service (AKS), and IBM Kubernetes Service.

3.Kubeadm: Kubeadm is a tool for bootstrapping a minimal, production-ready Kubernetes cluster. It's often used for setting up clusters on bare-metal servers, virtual machines, or cloud instances. Kubeadm automates many of the manual steps required to initialize a cluster.

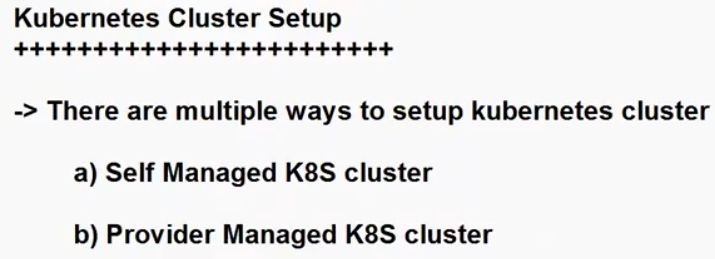
4.Kops (Kubernetes Operations): Kops is a tool for creating, upgrading, and managing Kubernetes clusters on AWS (Amazon Web Services). It simplifies the process of deploying Kubernetes on AWS infrastructure.

5.RKE (Rancher Kubernetes Engine): RKE is a tool for deploying, managing, and scaling Kubernetes clusters. It's commonly used in conjunction with Rancher, a container management platform, and can be used on various infrastructure providers.

6.Custom Installation: For advanced users, it's possible to manually set up a Kubernetes cluster by installing and configuring all the necessary components, including the control plane components (API server, etcd, etc.) and worker nodes.

7.Managed Kubernetes Distributions: Some companies offer their distributions of Kubernetes, which are preconfigured and optimized for specific use cases. Examples include OpenShift by Red Hat and Tanzu Kubernetes Grid by VMware.

8.Multi-Cluster Management Tools: There are tools like Crossplane and ArgoCD that help manage and deploy multiple Kubernetes clusters across different environments, enabling cluster lifecycle management and GitOps workflows.



1. **Self Managed K8S Cluster:**

* A self-managed Kubernetes (K8s) cluster, also known as a DIY (Do-It-Yourself) Kubernetes cluster, refers to a Kubernetes cluster that you set up, configure, and manage entirely on your own, typically without the assistance of a managed Kubernetes service provided by a cloud provider or a Kubernetes distribution.

Key characteristics and considerations of a self-managed Kubernetes cluster include:

Manual Setup:

* You are responsible for provisioning the infrastructure (physical servers, virtual machines, or cloud instances), installing the necessary software components (container runtime, Kubernetes control plane, and worker node components), and configuring networking and security settings.

1. Complete Control:

* With a self-managed cluster, you have full control over every aspect of the Kubernetes environment. This level of control allows you to customize the cluster to your specific requirements and integrate it with your existing infrastructure and tools.

1. Complexity:

* Setting up and managing a self-managed Kubernetes cluster can be complex and time-consuming, especially for production environments. It requires expertise in Kubernetes, infrastructure management, networking, and security.

1. Flexibility:

* Self-managed clusters offer flexibility in terms of the choice of container runtimes, networking solutions, and storage options. You can tailor the cluster to meet your organization's unique needs.

1. Maintenance:

* You are responsible for routine maintenance tasks such as upgrading Kubernetes versions, applying security patches, monitoring cluster health, and ensuring high availability. These tasks can be resource-intensive.

1. Scalability:

* You can scale a self-managed Kubernetes cluster by adding or removing nodes as needed to handle
* increased workloads. This scalability is under your direct control.

1. Backup and Disaster Recovery:

* Implementing backup and disaster recovery solutions is crucial to ensure data and application availability in case of failures or data loss.

1. Resource Management:

* Managing resources like CPU, memory, and storage for applications and pods running in the cluster is your responsibility. You may need to implement resource quotas and limits to prevent resource contention.

1. Monitoring and Logging:

* You'll need to set up monitoring and logging tools to gain insights into the cluster's performance and troubleshoot issues effectively.

1. Security:

* Ensuring the security of your self-managed cluster is paramount. You must configure network policies, implement access controls, and regularly apply security best practices.

1. **kubeadm:**

* **Multi-Node Clusters:** kubeadm is used to initialize and join nodes to a Kubernetes cluster. It's suitable for creating multi-node clusters with separate control plane and worker nodes.
* **Production Ready:** It's often used in production environments where you need a reliable, scalable Kubernetes cluster.
* **Customization:** kubeadm provides more flexibility for customizing your Kubernetes cluster configuration. You can choose the container runtime, networking solution, and other components.
* **Configuration Files:** You can define the cluster's configuration in YAML files, making it easier to version and manage your cluster configuration.

In summary, Minikube is a tool designed for local development and testing, whereas kubeadm is used for setting up more complex and scalable Kubernetes clusters, whether for development, testing, or production. The choice between the two depends on your use case.

Popular examples of provider-managed Kubernetes services include:

Amazon Elastic Kubernetes Service (EKS) on AWS.

Google Kubernetes Engine (GKE) on Google Cloud.

Azure Kubernetes Service (AKS) on Microsoft Azure.

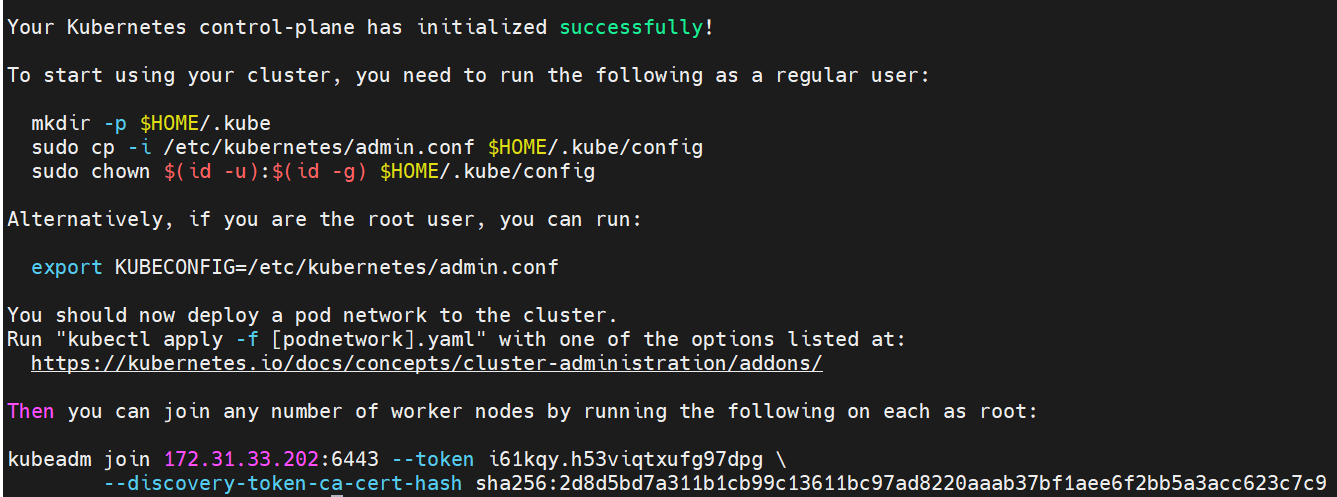
DigitalOcean Kubernetes (DOKS) on DigitalOcean.

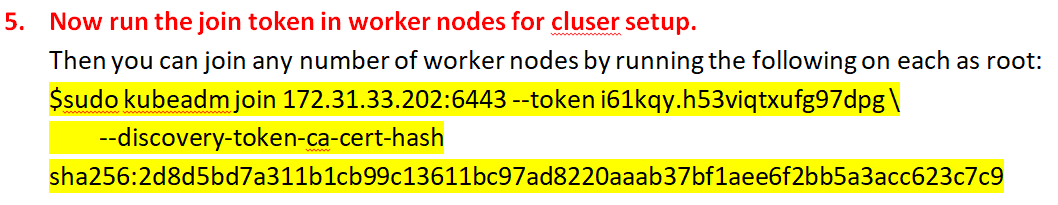
IBM Cloud Kubernetes Service on IBM Cloud.

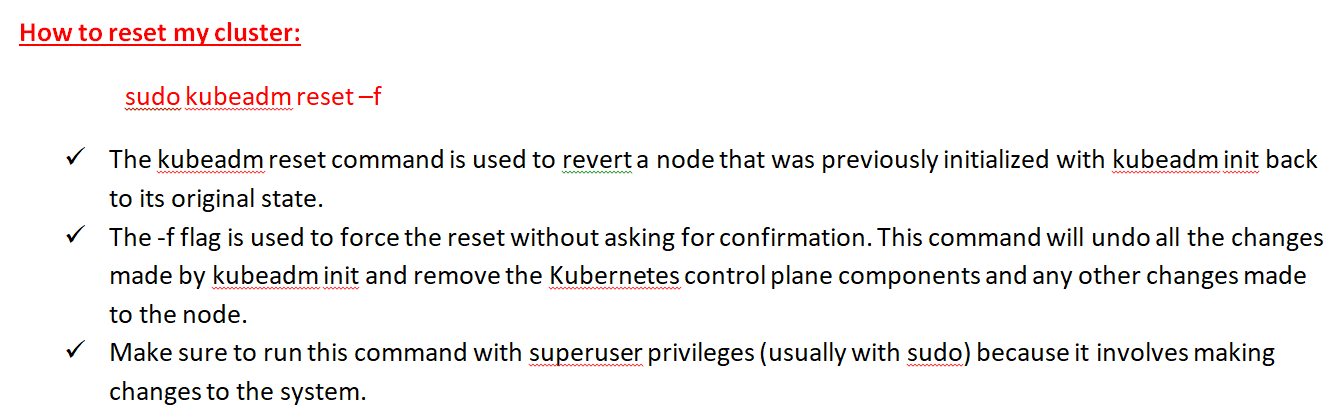
sudo systemctl start kubelet

sudo systemctl enable kubelet.service

**sudo kubeadm init**







**After running kubeadm reset, you should expect the following to happen:**

* The Kubernetes control plane components (API server, controller-manager, and scheduler) will be stopped and removed.
* Any kubeconfig files for the control plane will be deleted.
* The kubelet service will be restarted and any containers created by the control plane will be stopped and removed.
* Any changes to iptables rules made during the initialization will be reverted.
* The node will no longer be part of the Kubernetes cluster.

**How to get token later as well:**

$kubeadm token create --print-join-command

Config file contains:

* A kubeconfig file is a configuration file used by the Kubernetes command-line tool kubectl to specify how it should interact with a Kubernetes cluster.
* It contains several pieces of information and settings necessary for kubectl to communicate with a specific Kubernetes cluster. Here are the key elements typically found in a kubeconfig file:

Cluster Configuration:

* Cluster Name: A user-defined name for the cluster.
* Cluster Server: The URL of the Kubernetes API server for the cluster.
* Certificate Authority Data: The certificate authority (CA) that signs the server certificate, usually in PEM-encoded format. It's used to verify the server's certificate when making secure API requests.

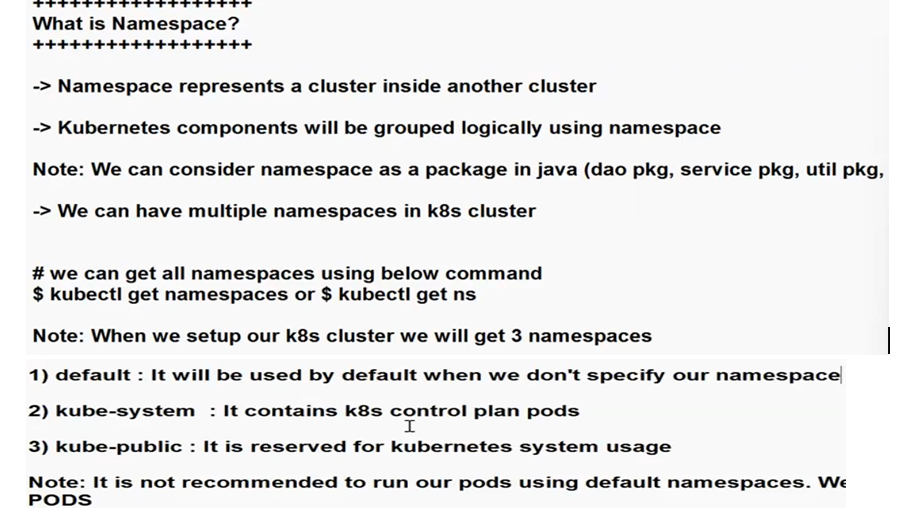
User Configuration:

* User Name: A user-defined name for the user context.
* Client Certificate Data: The client certificate for authentication, often in PEM-encoded format.
* Client Key Data: The private key associated with the client certificate, also in PEM-encoded format.
* Token: An authentication token used instead of client certificates and keys.
* Username and Password: Alternative authentication options for username and password-based authentication.

Context Configuration:

* Context Name: A user-defined name for the context.
* Cluster: The cluster to use for this context.
* User: The user to use for this context.
* Namespace: The default namespace for kubectl commands when using this context.
* Current Context: Specifies which context should be used by default when running kubectl commands. Typically, you can switch between contexts to interact with different clusters or namespaces.
* $sudo systemctl restart kubelet.service

kubectl cluster-info



In a Kubernetes (K8s) cluster, there are several built-in namespaces, each serving a specific purpose:

1. default:

* The "default" namespace is the namespace where resources are placed if no namespace is specified during resource creation.
* It's often used for small deployments, testing, or when you haven't created additional namespaces for specific projects or teams.
* Resources in this namespace are typically not isolated from each other.

1. kube-system:

* The "kube-system" namespace is used to house Kubernetes system-level components and infrastructure services that are required for the cluster to function correctly.
* These include components like the API server, controller manager, scheduler, and add-ons like DNS and monitoring agents.
* Resources in this namespace are essential for the operation of the cluster and should not be deleted or modified without a thorough understanding of their purpose.

1. kube-public:

* The "kube-public" namespace is a special namespace that contains resources that should be made publicly accessible to all users (authenticated and unauthenticated) of the cluster.
* It is typically used to host a ConfigMap that is accessible to all pods in the cluster. For example, it may store a cluster-wide configuration or welcome message.
* $kubectl get namespaces

#kubectl create namespace my-namespace

$kubectl get pods -n <your-custom-namespace>

$kubectl get pods --all-namespace

Here will observe kube-apiserver, kube-controller, kube-schedular, kube-etcd all the control plane components (POD’s)are running in one namespace ie: kube-system

kubectl delete namespace my-namespace

NOTE:

* Termination of All Resources: All resources (pods, services, deployments, config maps, secrets, etc.) that belong to the deleted namespace are terminated and deleted. This includes both user-created resources and system-level resources associated with the namespace.
* Pod Termination: Pods running in the deleted namespace are terminated gracefully
* Services Deletion: Any services defined within the namespace are deleted. These services are no longer accessible from other namespaces or externally.
* Garbage Collection: Kubernetes has a garbage collection mechanism that ensures the cleanup of resources associated with the deleted namespace, even if their owner references the namespace.

How to create a POD?

We can create a Pod in Kubernetes using two common methods: imperative(Interactive) and declarative.

1. Imperative Method /interact(using kubectl commands)
2. Declarative Method (using YAML configuration files)

**Imperative Method (using kubectl commands):**

1. Using kubectl run:

* The kubectl run command allows you to create a Pod in an imperative manner, specifying various parameters directly on the command line. This method is handy for quick testing and debugging.

Syntax: kubectl run my-pod --image=my-image

This command creates a Pod named "my-pod" using the specified Docker image "my-image" and exposes port 80.

1. Using kubectl create:

* You can also create a Pod by providing a YAML configuration directly via the kubectl create command. This method is useful when you have a pre-defined Pod configuration in a YAML file.

Syntax: Kubectl create –f <YAML-FILE>

EX: kubectl create -f pod-definition.yaml

Here, "pod-definition.yaml" is a YAML file containing the Pod definition.

In this example:

* apiVersion specifies the Kubernetes API version for the Pod resource (in this case, "v1").
* kind specifies the type of resource (in this case, "Pod").
* metadata contains metadata about the Pod, including its name.
* spec defines the specification for the Pod, including the containers it should run. In this case, there is one container named "my-container" that runs the "nginx" image.

Save the YAML File: Save the YAML file with a descriptive name (e.g., my-pod.yaml).

Apply the YAML File: Use the kubectl apply command to create the Pod based on the YAML configuration:

$kubectl apply -f my-pod.yaml

**How to verify control plane components status:**

**# ku get componentstatuses**

**How can I know my pod complete details?**

Now let’s describe the pod: It will show’s all the details about pod(like postmortem)

**How can I know where the pod is running? Which node?**

Syntax: $kubectl get pods –o wide

**POD NAME**

**READY**

**STATUS**

**RESTARTS**

**AGE**

**IP (this is pod ip OR worker node IP?)**

**NODE (node IP, pod is running this worker node)**

**How to get the response from pod:**

$curl <pod ip>:<port> (where we execute this command in control plane or worker node?)

**Pod Lifecycle**

* This page describes the lifecycle of a Pod. Pods follow a defined lifecycle, starting in the Pending phase, moving through Running if at least one of its primary containers starts OK, and then through either the Succeeded or Failed phases depending on whether any container in the Pod terminated in failure.
* Whilst a Pod is running, the kubelet is able to restart containers to handle some kind of faults.

Pod phase:

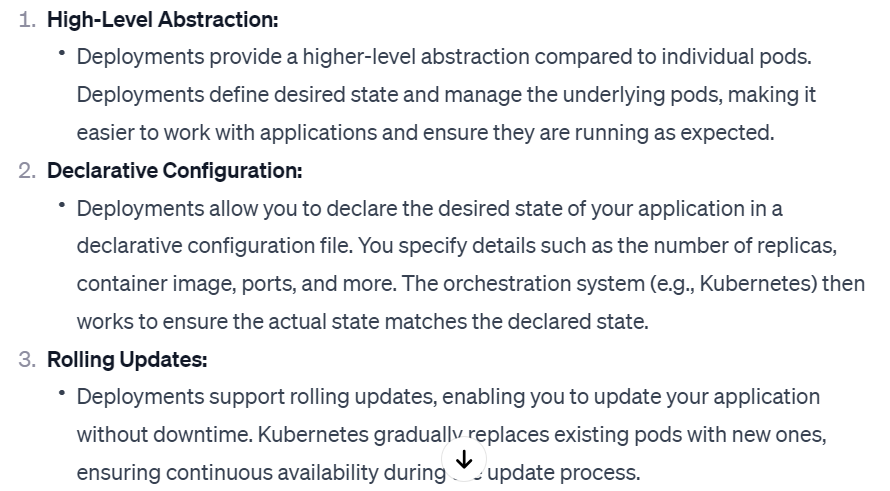
Pending Running Succeeded Failed Unknown

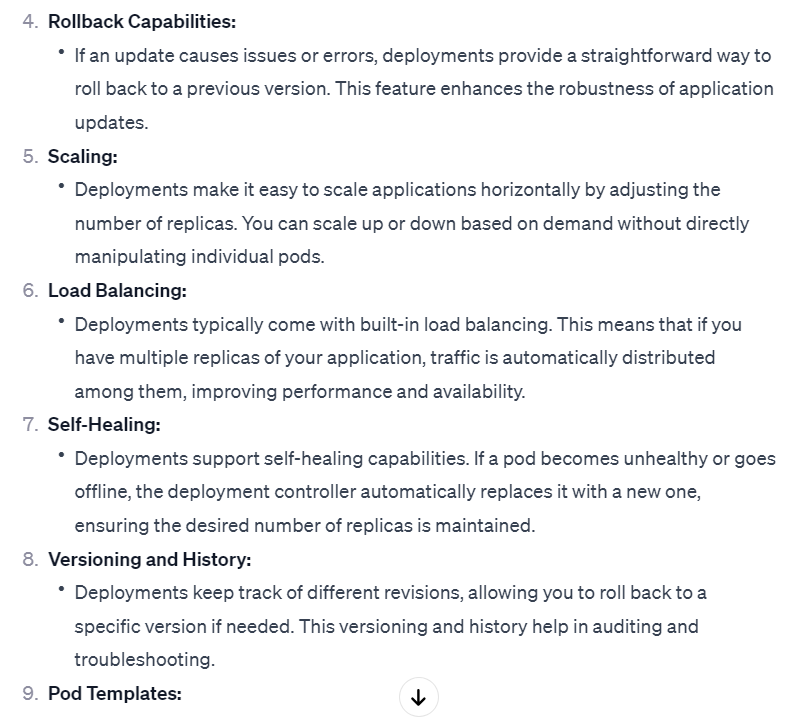
**Container states:**

There are three possible container states: Waiting, Running, and Terminated.

1. What is NS & CGroups? [How Linux PID namespaces work with containers | Enable Sysadmin (redhat.com)](https://www.redhat.com/sysadmin/linux-pid-namespaces)
2. What is dockwer swarm(diff k8s & swarm), why con’t user swarm in you are project?
3. **What is k8s deployment?**

* Kubernetes deployment is a high-level resource object by which you can manage the deployment and scaling of the applications while maintaining the desired state of the application.
* You can scale the containers with the help of Kubernetes deployment up and down depending on the incoming traffic.
* If you have performed any rolling updates with the help of deployment and after some time if you find any bugs in it then you can perform rollback also.





* #ku scale deployment <deployment-name> --replicas <count>

NOTE: If I delete any worker node, who will take care of recreate the node?

kops controller will create a new node immediately for high availability.

* #ku get pods –n kube-system (will see kops controller)
* How to get all deployments: #ku get deployments.apps
* How to get all services like pods, svc, depl etc.. in k8s cluster: #ku get all
* How to know NS resources: # ku api-resources
* How to know how many fields will use for pod creation: # ku explain pod
* How to know what fields will use under spec: # ku explain pod.spec # ku explain pod.spec.volumes
* How to create a custom NS: #kubectl create ns my-ns #kubens #kubens kube-system
* How to run a pod in my custom ns: #ku run my-pod –n my-ns –image=nginx #ku get pods –n <NS>
* How to list all pods across all NS: #kubectl get pods --all-namespaces # kubectl get pods --all-namespaces -o wide
* kubectl port-forward my-pod 8080:80 (access only in local host)

Port forwarding is commonly used for debugging, testing, or gaining access to services running in a Kubernetes pod without exposing them to the wider network.

* What is endpoint in svc: The endpoint of a service is essentially the combination of the service's cluster IP and port. When you create a service, Kubernetes sets up a virtual IP (Cluster IP) that acts as a stable entry point to reach the pods that are part of the service.

When we are deleting the pod the Endpoints will shows none. (interview qns), service is availabel but pod is not existed. That is the reason application will not access.

* can we create two sevices for one pod?
* How to delete all pods and svc: #ku delete pods,svc --all
* Side car container?

A sidecar container is a design pattern that allows you to run an additional container alongside your main container in the same pod. The sidecar container can perform tasks that complement the main container, such as syncing data from a remote source, collecting and shipping logs, providing health checks and metrics, proxying network traffic, encrypting or decrypting data, or injecting faults for testing.

The sidecar container shares the same lifecycle, resources, and network namespace as the main container but has its own file system and process space. This means that the sidecar container can access the same ports, volumes, and environment variables as the main container but cannot interfere with its execution.

* if I execute the $ku get pods in linux server it’s working, but when I execute in power shell it’s not working why? is no kube/config file in your local machine
* what is labels?
* In Kubernetes, labels and annotations are key-value pairs that you can attach to resources, such as pods, services, nodes, and more, to provide additional metadata and information about those resources.
* Labels are used for identifying and selecting resources. They are primarily used by selectors, which are criteria used to filter and group resources. For example, you can label your pods with environment information (e.g., "production" or "development") and then create services or deployments that target only the pods with a specific label.
* Labels size max 63 characters only.

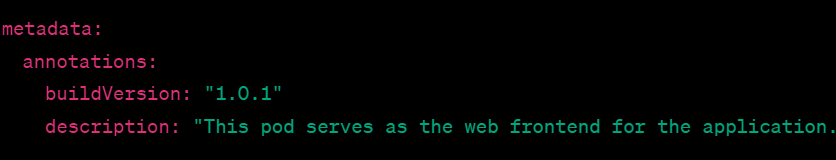
Annotations:

* Annotations are used to store non-identifying information or metadata about resources. Unlike labels, annotations are not used by selectors for resource selection.
* Annotations are typically used for adding notes or descriptions to resources that are meant to be read by humans or external tools.

Annotations can be used to store information like version numbers, build information, and descriptions that are not used for operational purposes.

* Annotations are also key-value pairs and are often set in the resource's metadata section.

#ku describe deployment depl (here will see )



Find labels: ku get pods -n kube-system --show-labels

* After deployment pod naming structure:

mydeployment-5cb695987d-5h5hj

here🡪 mydeployment= deployment name

5cb695987d = Replicaset name

5h5hj = pod name

* How to know how many container are running in pod? Here will see entire info about pod. #ku describe pod
* What are Kubernetes Environment Variables?
* Environment variables in Kubernetes are a fundamental concept that allows you to configure and customize applications running inside containers. They are key-value pairs containing information that applications can use to adjust their behavior at runtime.
* These variables are injected into the container's environment and can be used to pass configuration data, secrets, or any other dynamic information required by the application
* How to execute the container and see all the env details:

# ku eexec -it <pod-name> -- bash

* If I have a more then one containes in the pod, how it will execute?

#ku exec –it <pod-name> -c <second container-name>-- bash

* How to expose the deployment outside the cluster?

#ku expose deployments mydepl --name svc1 --port 8080 --target-port 80 --type NodePort

* #watch -n 1 kubectl get pods

execute the kubectl get pods command every 1 second (-n 1) to monitor the status of Kubernetes pods.

* minReadySeconds: 10 specifies the minimum number of seconds for which a new pod should be ready without any errors before the deployment considers it as available.
* maxUnavailable: 1 and maxSurge: 1 are also part of the RollingUpdate strategy, determining how many pods can be unavailable and how many can be added above the desired replica count during the update process.

NOTE: finally maxSurge and maxUnavailable are controlling the pods creation status.

* How to know the rollout status of the pods: #kubectl rollout status deployments/mydepl
* How to pause the deployments?

This command will pause the rollout of the specified deployment, preventing any further updates from being applied. It's useful for scenarios where you want to investigate an issue or perform some manual intervention before allowing the deployment to continue.

#ku rollout pause deployment/mydepl # ku resume pause deployment/mydepl

* Disadvantages of Deployments:  
  1. There is no rule to run desired pods in all the nodes(may be it will run more then one pods in one node).
* When we use deployment mode:

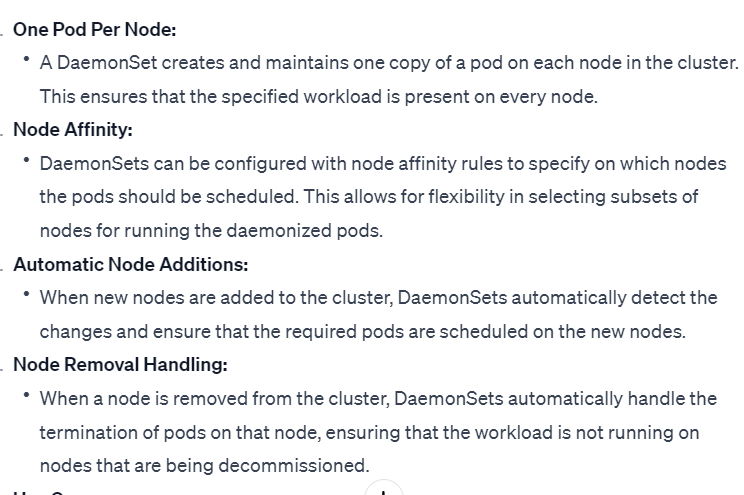
In Kubernetes, the Deployment resource is commonly used to manage and control the deployment of applications. Here are scenarios in which using the Deployment mode is beneficial:

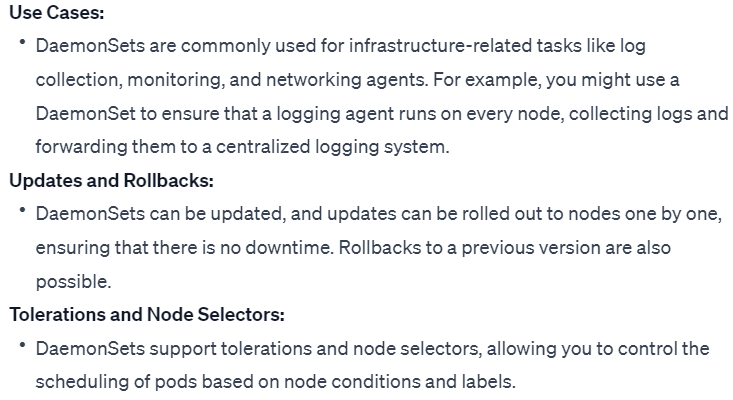
1. Rolling Updates:
2. Rollbacks:
3. Scaling:
4. Self-Healing
5. Declarative Configuration:
6. Service Discovery and Load Balancing:
7. MinReadySeconds and MaxSurge/MaxUnavailable:
8. Pod Template Changes:

* What is A DaemonSet

In Kubernetes, a DaemonSet is a resource type that ensures that a specific pod runs on all (or a subset of) nodes in a cluster. The primary purpose of a DaemonSet is to ensure that a particular workload, such as a monitoring agent or a log collector, is running on every node in the cluster.

See some characteristics:





* What is statefulset?