

KUBERNETES

History:

Kubernetes is also called as K8s because (with the number 8 standing for the number of letters between the “K” and the “s”). Hence, it is also called K8s. It was developed by Google using Go language. Later it was donated to CNCF in 2014 (Cloud Native Computing Foundation, it's a software company). The first version of Kubernetes was released in the year 2015.

What is Kubernetes?

Kubernetes is an open-source container orchestration platform which is used to automate many manual processes like deploying, managing and scaling containerized applications.

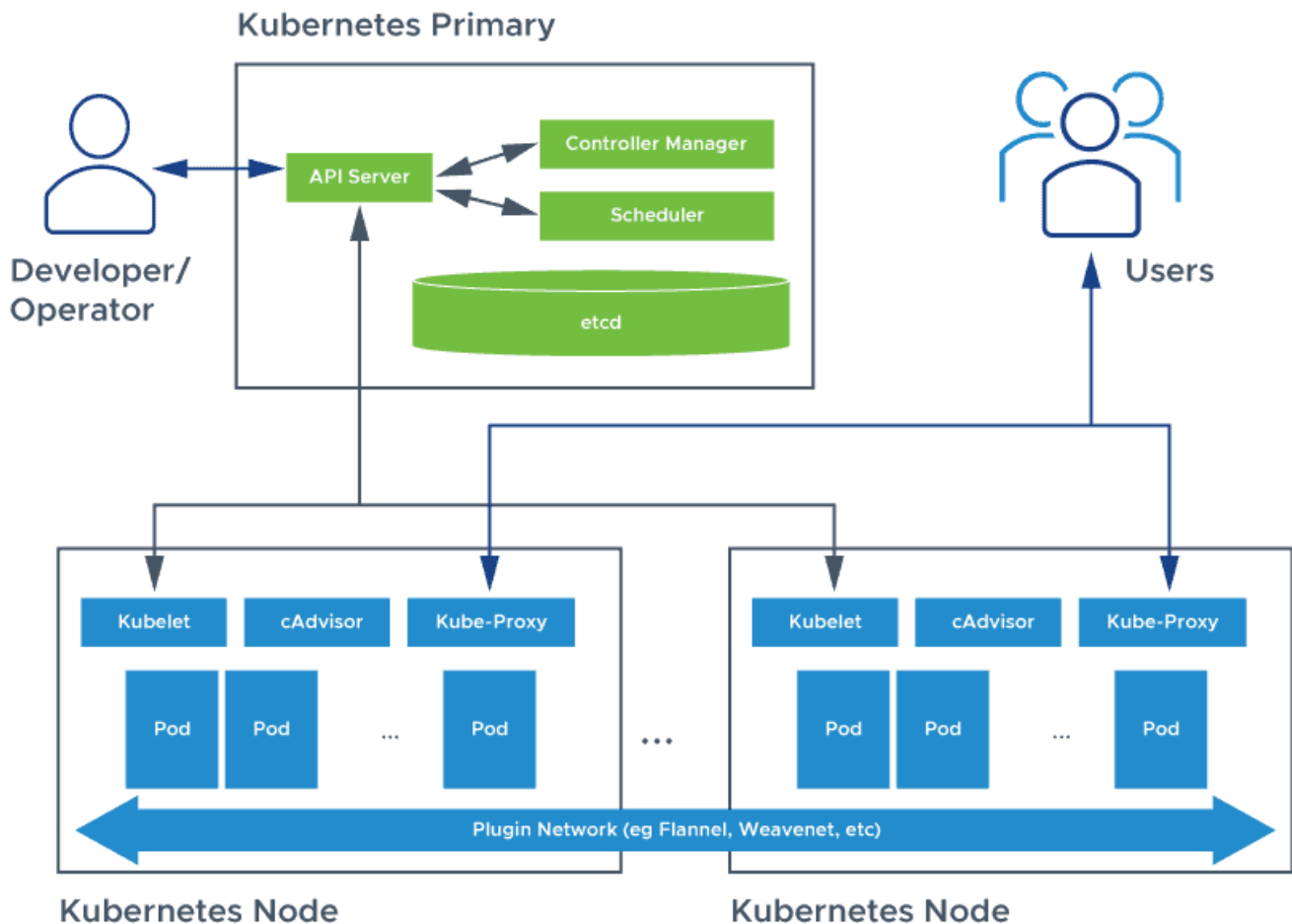
Why do we need Kubernetes?

In Docker we used to pack all the dependencies by creating an image, build the image to create a container and the application gets deployed and can be accessed by the end-user. In production environment, you need to manage the containers that runs the applications and ensure that there's no downtime. However, there are some drawbacks so in order to overcome. Therefore, we moved to Kubernetes.

What is the relationship between Docker and Kubernetes?

The relationship between the Docker and Kubernetes is explained with the help of their respective logos, which is ship and wheel. The docker logo is designed in the form a “ship” containing containers in it and the kubernetes logo is designed as a “wheel” which is used to manage/run/control the containers. The ‘Docker’ is mainly used to create containers to deploy the applications and to control, manage the containers we use ‘Kubernetes’.

KUBERNETES ARCHITECTURE



There are mainly two types of nodes in the Kubernetes same as Docker Swarm. The first one is Manager node (Kubernetes Master) and the other one is Worker node (Kubernetes node). The manager node is also called as “Control plane”.

Below are the components that are available in the manager node and worker node:

A) Manager node

1) API server

2) Control Manager

3) Scheduler

4) etcd

B) Worker node

1) Kubelet

2) Kube-proxy

3) Pod

COMPONENTS OF A MANAGER NODE:

1) API server: API server is abbreviated as Application Programming Interface server and also called as Qube API server.

- It's used to accept the request from the user/operator and stores the request in ETCD.
- It's the only component in the kubernetes cluster who has the ultimate power to communicate with etcd in the entire kubernetes cluster, no other component has the power to do so.

Authenticate user - - > Validate request - - > Retrieve data

Example (a): If we give a command to check the list of pods, it goes as a request to API server. The request gets stored in etcd. etcd checks the data in it and reverts back to API server and API server gives result to the operator/user.

Example (b) : If we give a command to create a pod, now the request from user goes to API server, from API server it goes to scheduler first and the scheduler now checks with two worker nodes and decides where to create. Kubelet gives information to scheduler and based on the information given by kubelet about how many pods are running, based on the less pods running in the worker node, there a new pod is scheduled to create.

Example (c): So in worker node -1 if there are 5 pods running and in worker node -2 if 7 pods are running, then the scheduler schedules the pod to create in worker node -1.

Later the scheduler checks with the kubelet and kubelet gives information to scheduler which again given to API and API gives the result to operator.

2) ETCD: It's like a database to our cluster and it stores all the information about the nodes, pods, configurations, secrets, roles etc. The data is stored in key-value format.

3) Scheduler: The scheduler decides about creating a pod in the cluster either in worker node -1 or worker node -2. The API server tells the scheduler and it takes care of scheduling decisions.

- It searches if any pending tasks are present in etcd.
- If any pending task is found in etcd, it assigns a worker node.
- It's the crucial component in cluster that communicates with the kubelet and decides in which worker node a pod can be created which meet the resource requirements and constraints.
- It always stay in synchronization with the API server to monitor and complete the given tasks.

The following tasks performed by the scheduler:

- Pod creation
- Pod running
- Pod scheduling
- Selecting node
- Kubelet action

Ex: So if 5 pods running in worker node -1 and 7 pods are running in worker node -2, then the scheduler schedules the pod to create in worker node -1 because there's less load.

4) Controller Manager: The controller manager collects the data from the API server and performs the actions.

The following tasks performed by the Controller:

- **Replication controller:** It takes responsibility of creating a pod automatically if it's deleted.
- **Node controller:** It creates a node if it's deleted.

- **Endpoint controller:** In kubernetes there's an endpoint object called Endpoint and it is used to maintain a link between pods and services. It ensures that there's reliable communication is established within the cluster.

Ex: Let's say if a pod is deleted in a node and another one is created (RC) but in order to make the service accessible even after the new one is created, the endpoint controller is used to ensure that the IP addresses are assigned to the pods again and route traffic correctly.

"Each Service has a corresponding Endpoints object that holds the IP addresses and ports of the pods backing the Service".

- **Service Account & Token Controllers:** It creates default accounts and API access tokens for new namespaces.

How a pod is created in Kubernetes?

A request is given by the operator/user to create a pod - - > the request goes to API server - - > API server sends the request to scheduler - - > scheduler communicates with the kubelet as they both have communication between them - - > the kubelet checks about the status of running pods and sends the information to scheduler - - > the scheduler tells the API server that the worker node -1 is having less pods when compared to worker node -2 and tells to create a pod in worker node -1 - - > the API server creates a pod in worker node - - > kubelet tells the API server that the pod is created and at last the API server tells the etcd cluster that the pod is created and the data will be automatically updated in etcd and then we can check it from the etcd cluster.

COMPONENTS OF A WORKER NODE:

1) **Kubelet:** Kubelet is the crucial component in the worker node that takes care of whether pods are running or not. If they're not working properly, kubelet replaces the old one with a new one and since the failed pod can't be restarted, the IP of the pod might also gets changed.

How a pod is created automatically with the help of a controller manager and kubelet?

Here we'll write a rc.yml file (replication controller) and we declare that whenever a pod is deleted a new one needs to be created. We give a command to execute it, the command goes to API server, the request goes from API to controller manager and the controller manager communicates with kubelet and that's how a pod is created. So there's must be controller manager and kubelet, the communication must be there between them and they both can't do their job individually.

2) **Kube-proxy:** Kube-proxy is the component which takes care of all network configuration information of the entire cluster such as pod IP services, deployments etc. In simple words it takes care of networking layer. Like if a NodePort service is created and in order to access the application using the IP address Kube proxy is the one that takes care of it.

IN DETAIL:

3) **Pod:** A pod is a group of one or more containers.

We create a POD - - > CONTAINER GETS CREATED - - > RUNS THE APPLICATION

What is a POD in Kubernetes?

- > A pod is a smallest object in K8's and it is ephemeral (short living object).
- > A pod is like one container or a group of containers.
- > A pod acts as a single instance for our application.
- > We mostly use a single container in a pod but if it's required we can also use multiple containers in a single pod.
- > The containers inside the pod shares same networks, same namespace and storage volumes.
- > We must specify the image name, necessary configuration, resource limits while creating a pod.
- > K8s cannot communicate with the containers, instead they communicate only with pods.

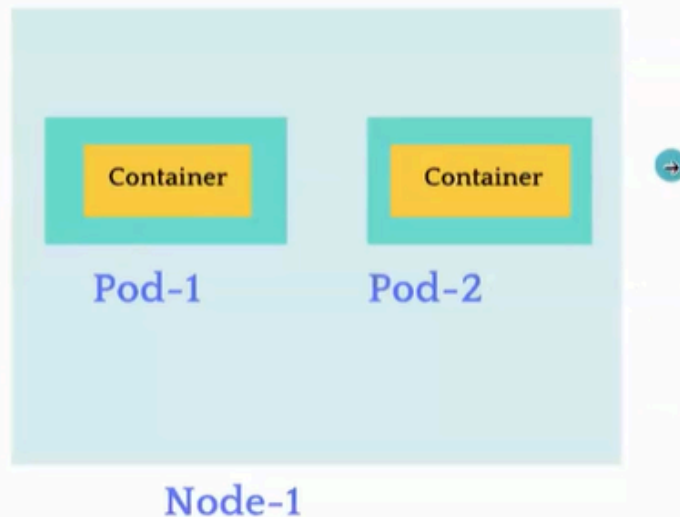
NOTE: In Kubernetes we'll not deploy the containers directly instead there's an object in Kubernetes which is called as a Pod which contains containers.

TYPES OF PODS:

- 1) Single container pod.
- 2) Multi-container pod.

1) Single container pod: In a single node we may have many pods, but we will only have a single container inside the pod. At times we need two containers in a single pod then we need to use the multi container pod.

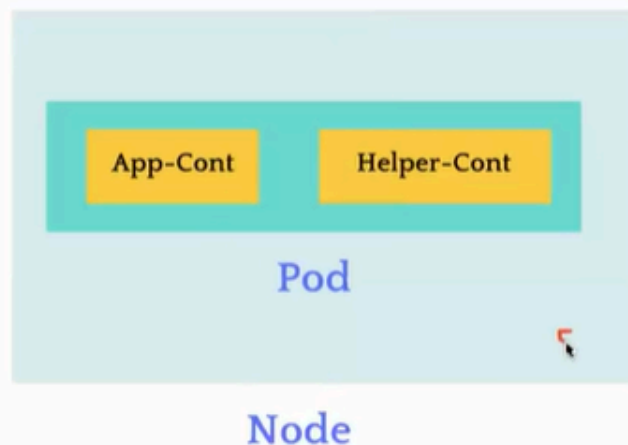
SINGLE CONTAINER POD



Basically, each pods needs only one container. But sometimes we need 2 containers in one pod.

2) Multi container pod: A multi container pod will have two containers inside the pod. Application container and Helper container.

MULTI CONTAINER POD



Helper containers are additional containers included in a pod to perform some tasks that support the primary container which are running in the pod.

> Helper container is used collect the logs information from the application container.

> These can assist in syncing the files or data between containers in a pod. For example if we've a pod, we've an application container and the helper container, helper container is responsible for syncing

configuration files or other shared resources.

> It does Security-related tasks, such as handling secrets, encryption, or authentication.

Pods can be created in two ways.

1) Imperative (Command) : `kubectl create`

2) Declarative (Manifest file)

1) Imperative: The imperative pod creation uses `kubectl` command to create a pod and it is useful for quick creation and modifying the pods.

SYNTAX: `kubectl run pod_name --image=image_name`

`kubectl run pod-1 --image=ubuntu`

`kubectl` > CLI tool to run the command to perform an action, `pod-1` - > name of the pod, `ubuntu` - > image name

2) Declarative: In declarative way, we can use a YAML file to write a manifest file to create a pod. The manifest file extension should be a “`yml`”.

What is Manifest file?

A manifest file is a specification of a Kubernetes API object in JSON or YAML format. In simpler words, a Kubernetes manifest defines the resources (e.g., Deployments, Services, Pods, etc.) that users want to create, and how they want these resources to run inside a Kubernetes cluster.

```
---
apiVersion: v1
kind: Pod
metadata:
  name: pod2
spec:
  containers:
    - name: cont2
      image: nginx
      ports:
        - containerPort: 80
```

Here in the manifest file,

apiVersion: It's the version of the object in api resources.

kind: The type of object from the api resources.

metadata: It's the data about the object.

name: The name of the Pod, which is is the metadata for a Pod.

spec: The info about the container.

containers: Container info.

- **name:** name of the container.

image: Name of the image.

ports: The port number of the pod should work on.

- **containerPort:** The port number which is used to access the application based on the image.

POD COMMANDS:

To create a pod: **kubectl run pod_name --image=image_name**

To see a list of pods: **kubectl get pods/ kubectl get pod/ kubectl get po**

To see a list of pods with full info: **kubectl get po -o wide**

To see a particular POD full info: **kubectl get po pod-name -o wide**

To see a particular POD info in json format: **kubectl get po pod-name -o json**

To see a particular pod full info in YAML format : **kubectl get po pod-name -o yaml**

To describe a pod: **kubectl describe pod pod-name**

To describe all pods: **kubectl describe pod**

To delete a pod : **kubectl delete pod pod-name**

To delete multiple pods: **kubectl delete pod pod-1 pod-2 pod-3**

To delete all pods : **kubectl delete po --all**

To enter into a pod: **kubectl exec -it pod_name -c cont_name bash**

To get pod logs: **kubectl logs pod-name**

To get container logs: **kubectl logs pod-name -c cont-name**

To execute the manifest file first time: **kubectl create -f manifest file name (yaml file)**

To execute the manifest file if anything is updated in the same file: **kubectl apply -f yaml filename**

To get pod details along with labels: **kubectkl get po --show-labels**

KUBERNETES CLUSTERS

Kubernetes in production systems

In real time we don't use single node clusters such as minikube, k3d because it has less CPU, RAM, Memory which won't be sufficient for the organizational requirement. Therefore we prefer one of the most widely used clusters called KOPS (kubernetes operations).

Before KOPS, kubeadm cluster was most widely used, it requires or involves manual efforts for modification, upgradation, whereas KOPS is most widely used for installing Kubernetes and the lifecycle of Kubernetes includes installation, modification and deletion that is managed by KOPS which makes the customer experience smooth and easy.

The difference between Kubernetes and EKS is that, Kubernetes is an open source platform but EKS is a distribution of Kubernetes and cloud-managed cluster it takes care when you ran into any issues and also has some additional plugins and it's a UI based cluster. If you choose two EC2 instances and setup your Kubernetes, there won't be any support from AWS cloud, for issues related to Kubernetes.

There are multiple ways to setup a Kubernetes cluster.

1) Self Managed K8's cluster:

a) mini kube (single node cluster)

> If we've a manager and a worker node in a single server then it's called as single-node cluster.

b) kubeadm (multi-node cluster)

> If we've manager in a different server and worker in a different server then it's a multi-node cluster. Even in production we use multi-node cluster to avoid application getting down in real time.

c) KOPS (Kubernetes Operations)

2) Cloud Managed K8's cluster:

a) AWS EKS (Elastic Kubernetes Services)

b) AZURE AKS (Azure Kubernetes Services)

c) GCP GKS (Google Kubernetes Services)

d) IBM IKE (IBM Kubernetes Services)

MINIKUBE:

- i) It's a platform independent tool used to setup single node cluster in K8s.
- ii) It creates only one node by default and helps to containerize the applications.
- iii) It contains API servers, etcd databases and container runtime.
- iv) We do have the manager and worker in the same node and used to development, testing and experimentation purposes on local.
- v) The installation process is easy when compared to other tools.

NOTE: We don't use single node clusters in real time. This is solely used for practice and experimentation purpose only.

MINIKUBE SETUP REQUIREMENTS/PREREQUISITES:

- > 2 CPUs or more.
- > A minimum of 20 GB of EBS volume.
- > 2GB of free memory
- > Container or virtual manager such as Docker.

You can use the below commands to setup the Minikube cluster or you can also run a script with all the commands.

A) Ubuntu:

```
#!/bin/bash
```

```
apt update -y
```

```
sudo apt install curl wget apt-transport-https -y
```

```
sudo curl -fsSL https://get.docker.com -o get-docker.sh
```

```
sudo sh get-docker.sh
```

```
sudo curl -LO https://storage.googleapis.com/minikube/releases/latest/minikube-linux-amd64
```

```
sudo mv minikube-linux-amd64 /usr/local/bin/minikube
```

```
sudo chmod +x /usr/local/bin/minikube
```

sudo minikube version

sudo curl -LO "https://dl.k8s.io/release/\${curl -L -s
https://dl.k8s.io/release/stable.txt)/bin/linux/amd64/kubectrl"

sudo mv kubectrl /usr/local/bin/

sudo chmod +x /usr/local/bin/kubectrl

**sudo curl -LO "https://dl.k8s.io/\${curl -L -s
https://dl.k8s.io/release/stable.txt)/bin/linux/amd64/kubectrl.sha256"**

sudo echo "\${cat kubectrl.sha256) kubectrl" | sha256sum --check

sudo install -o root -g root -m 0755 kubectrl /usr/local/bin/kubectrl

sudo kubectrl version --client

sudo kubectrl version --client --output=yaml

sudo minikube start --driver=docker --force

#bold ones aren't required.

B) RedHat:

yum install docker -y

systemctl start docker

systemctl status docker

curl -LO "https://dl.k8s.io/release/\${curl -L -s
https://dl.k8s.io/release/stable.txt)/bin/linux/amd64/kubectrl"

sudo mv kubectrl /usr/local/bin/kubectrl

sudo chmod +x /usr/local/bin/kubectrl

curl -LO https://storage.googleapis.com/minikube/releases/latest/minikube-linux-amd64

sudo install minikube-linux-amd64 /usr/local/bin/minikube

sudo yum install iptables -y

yum install conntrack -y

minikube start --driver=docker --force

minikube status

KUBECTL:

- > It's a CLI tool which is used to communicate with the Kubernetes cluster.
- > Whenever a command is given by an operator in the form of KUBECTL, it communicates with the Kubernetes API server.
- > We can create pods, we can manage pods, services, deployments and other resources.
- > We can also use KUBECTL for monitoring, troubleshooting, scaling and updating the pods.
- > The configuration of KUBECTL is in \$HOME/.kube directory and the version we're using is 1.28.

SYNTAX:

kubectl [command] [TYPE] [NAME] [flags]

kubectl api-resources - - > Gives list of api resources

Why KOPS?

KOPS means 'Kubernetes Operations'. The reason why we prefer KOPS over KUBEADM or K3D as a multi-node cluster is because, it setup all the infrastructure that is required to run our application. The infrastructure like Instances, networks (VPC), databases, volumes, Security groups, Load balancers, Auto scaling group and much more which all are required for access the application. Therefore we prefer KOPS.

In order to define in which account all the infrastructure has to be installed is done with the help of a service in AWS called as IAM (Identity Access Management). Here we need to create a user and we get the credentials, so these credentials needs to be configured.

KOPS (Kubernetes Operations)

STEP-1: Launch an Instance with t2.micro and 30GB SSD.

STEP-2: Install AWS CLI

- `curl "https://awscli.amazonaws.com/awscli-exe-linux-x86_64.zip" -o "awscliv2.zip"`
- `unzip awscliv2.zip`
- `sudo ./aws/install`

To check version: `/usr/local/bin/aws --version`

To set path: `vim .bashrc` and go to last line and paste this path `export PATH=$PATH:/usr/local/bin/`

```
source .bashrc
```

```
aws --version
```

STEP-3: Install KUBECTL & KOPS

```
curl -LO "https://dl.k8s.io/release/${curl -L -s  
https://dl.k8s.io/release/stable.txt)/bin/linux/amd64/kubectl"
```

```
wget https://github.com/kubernetes/kops/releases/download/v1.24.1/kops-linux-amd64
```

- PERMISSIONS: `chmod +x kops-linux-amd64 kubectl`
- MOVE FILES: `mv kubectl /usr/local/bin/kubectl`
- MOVE FILES: `mv kops-linux-amd64 /usr/local/bin/kops`
- TO SEE VERSION: `kubectl version && kops version`

STEP-4: Create IAM user with Admin Permissions and configure it in any region with table format.

STEP-5: Create INFRA setup

- To CREATE a BUCKET: `aws s3api create-bucket --bucket mustafask.k8s.local --region us-east-1`
- To ENABLE VERSION: `aws s3api put-bucket-versioning --bucket mustafask.k8s.local --region us-east-1 --versioning-configuration Status=Enabled`

Export Cluster data into bucket: `export KOPS_STATE_STORE=s3://mustafask.k8s.local`

Generate-Key: `ssh-keygen`

- To create a Cluster:

```
kops create cluster --name musta.k8s.local --zones us-east-1a --master-size t2.medium --node-size  
t2.micro
```

- To create a Cluster with 2 nodes and 1 master:

```
kops create cluster --name ravi.k8s.local --zones=us-east-1f --node-count=2 --master-count=1 --node-  
size=t2.micro --master-size=t2.medium
```

- To VIEW the Cluster : `kops get cluster`
- To edit the cluster : `kops edit cluster cluster_name`
- To RUN the Cluster : `kops update cluster --name cluster_name --yes --admin`
- To DELETE the Cluster: `Kops delete cluster --name cluster_name --yes`

STEP-6: Create PODS

To CREATE A POD:

To CHECK the PODS: `kubectl get pods`

To CHECK the where the POD IS RUNNING: `kubectl get pods -o wide`

To CREATE a CONTIANER:

* list clusters with: kops get cluster

*edit this cluster with: kops edit cluster musta.k8s.local

*edit your node instance group: kops edit ig --name=musta.k8s.local nodes-us-east-1a

*edit your master instance group: kops edit ig --name=musta.k8s.local master-us-east-1a

Finally configure your cluster with: kops update cluster --name musta.k8s.local --yes --admin

=====

vim pod-nginx.yml

apiVersion: v1

kind: Pod

metadata:

name: nginx

spec:

containers:

- image: nginx

name: nginx

=====

TO DELETE EXISTING POD: kubectl delete pod nginx

TO CREATE A POD: kubectl create -f pod-nginx.yml

TO DEPLOY POD: we need to delete existing pod (kubectl delete pod nginx) & write the code for deployment.

=====

vim deployment-nginx.yml

apiVersion: apps/v1

```
kind: Deployment

metadata:

  labels:

    run: nginx

  name: nginx-deploy
```

```
spec:

  replicas: 2

  selector:

    matchLabels:

      run: nginx
```

```
template:

  metadata:

    labels:

      run: nginx

  spec:

    containers:

      - image: nginx

        name: nginx
```

=====

TO DEPLOY IT: kubectl create -f deployment-nginx.yml --validate=false

TO DELETE THE WHOLE CLUSTER: kops delete cluster --name cluster_name --yes

PICTORIAL REPRESENTATION OF STEP BY STEP PROCEDURE TO IMPLEMENT KOPS (MULTI-NODE CLUSTER)

- 1) Launch an instance with t2.micro and 30GB SSD.
- 2) Install AWS CLI.

The AWS Command Line Interface (AWS CLI) is a unified tool to manage your AWS services. With just one tool to download and configure, you can control multiple AWS services from the command line and automate them through scripts.)

- `curl "https://awscli.amazonaws.com/awscli-exe-linux-x86_64.zip" -o "awscliv2.zip"`
- `unzip awscliv2.zip`
- `sudo ./aws/install`

TO CHECK VERSION: `/usr/local/bin/aws --version`

TO SET PATH: `vim .bashrc` and go to last line and paste this path `export PATH=$PATH:/usr/local/bin/`

3) Install Kubectl and KOPS

- Give x permissions (executable) = **`chmod +x`**
- Move the kubectl and kops to the default path to execute them **`/usr/local/bin/`**

```
[root@ip-172-31-25-79 ~]# ll
total 106708
drwxr-xr-x 3 root root      78 Nov 22 00:11 aws
-rw-r--r-- 1 root root 59382992 Nov 25 10:08 awscliv2.zip
-rw-r--r-- 1 root root 49885184 Nov 25 10:20 kubectl
[root@ip-172-31-25-79 ~]# chmod +x kubectl
[root@ip-172-31-25-79 ~]# ll
total 106708
drwxr-xr-x 3 root root      78 Nov 22 00:11 aws
-rw-r--r-- 1 root root 59382992 Nov 25 10:08 awscliv2.zip
-rwxr-xr-x 1 root root 49885184 Nov 25 10:20 kubectl
[root@ip-172-31-25-79 ~]# mv kubectl /usr/local/bin/
```

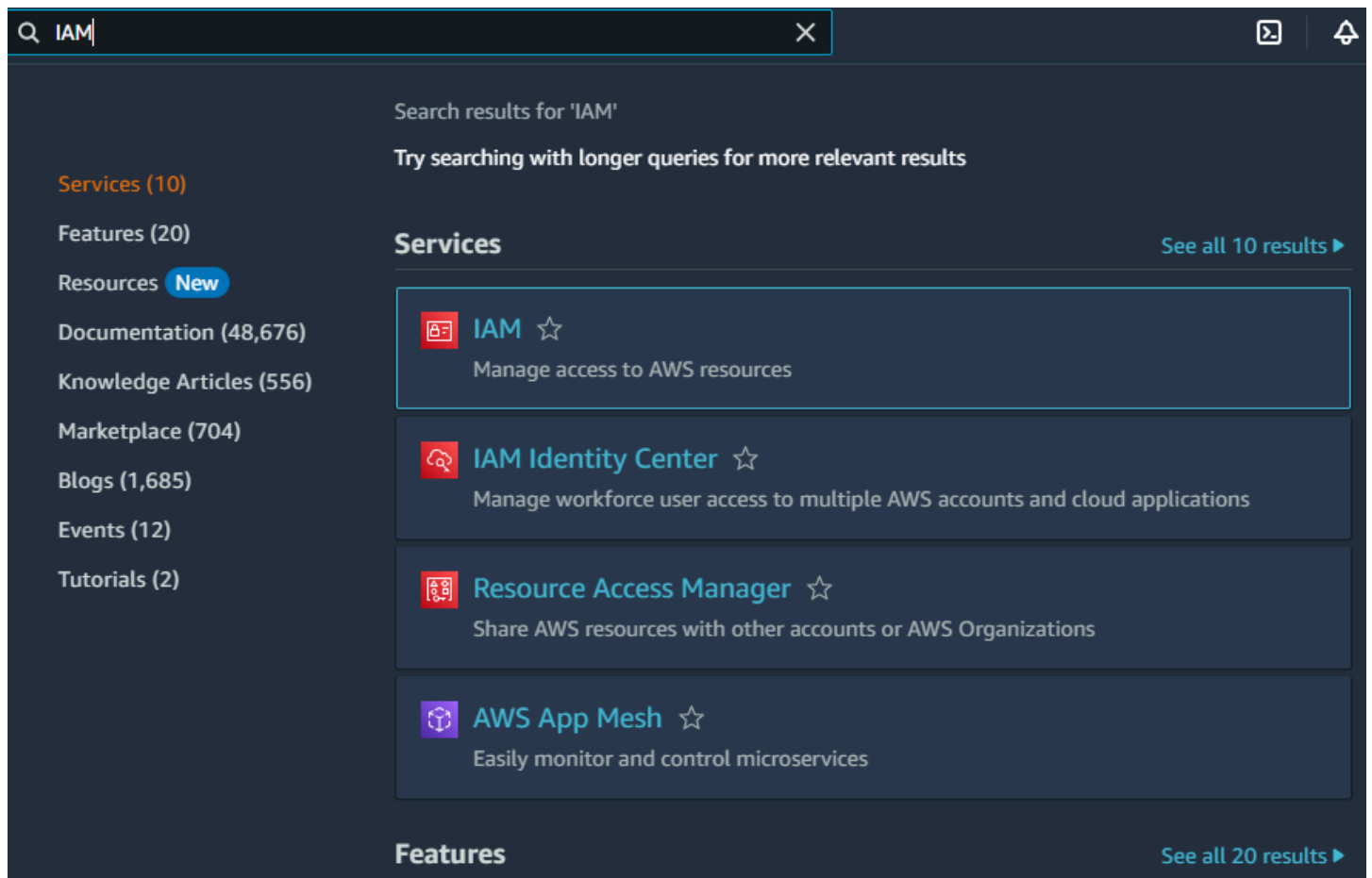
- You can check if both installed by **kubectl version** and **kops version**.

```
[root@ip-172-31-25-79 ~]# kops version
Client version: 1.24.1 (git-v1.24.1)
[root@ip-172-31-25-79 ~]# kubectl version
Client Version: v1.28.4
Kustomize Version: v5.0.4-0.20230601165947-6ce0bf390ce3
The connection to the server localhost:8080 was refused - did you specify the right host or port?
[root@ip-172-31-25-79 ~]#
```

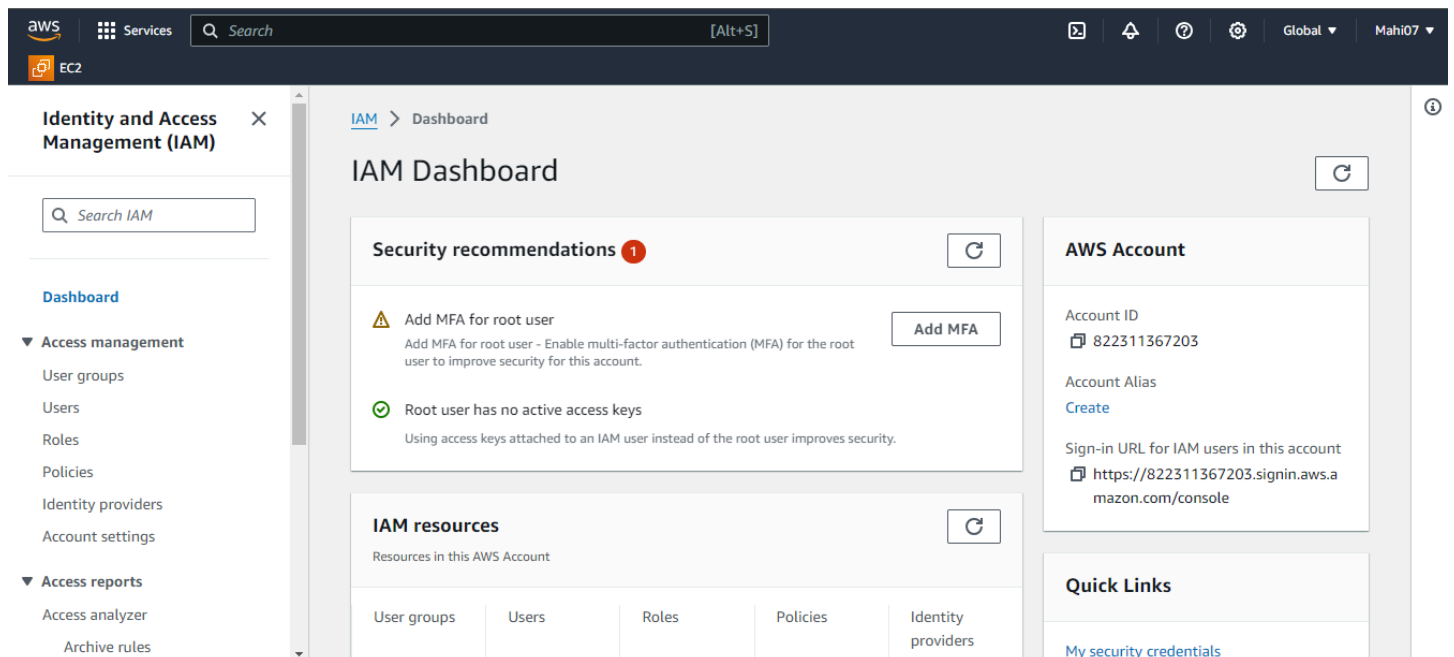
4) Create IAM user with admin permissions and configure it in any region with table format.

In order to define in which account all the infrastructure has to be installed is done with the help of a service in aws called as IAM (Identity Access Management). Here we need to create a user and we get the credentials, so these credentials needs to be configured.

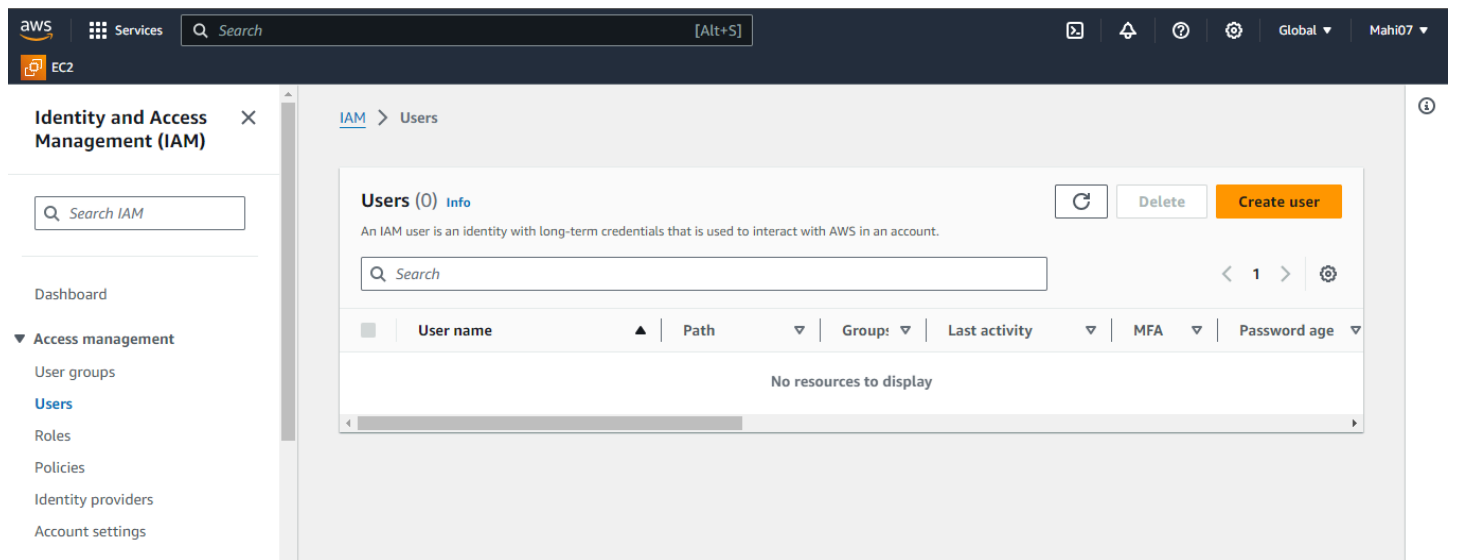
a) Search for **IAM** in global search bar and click on **IAM**.



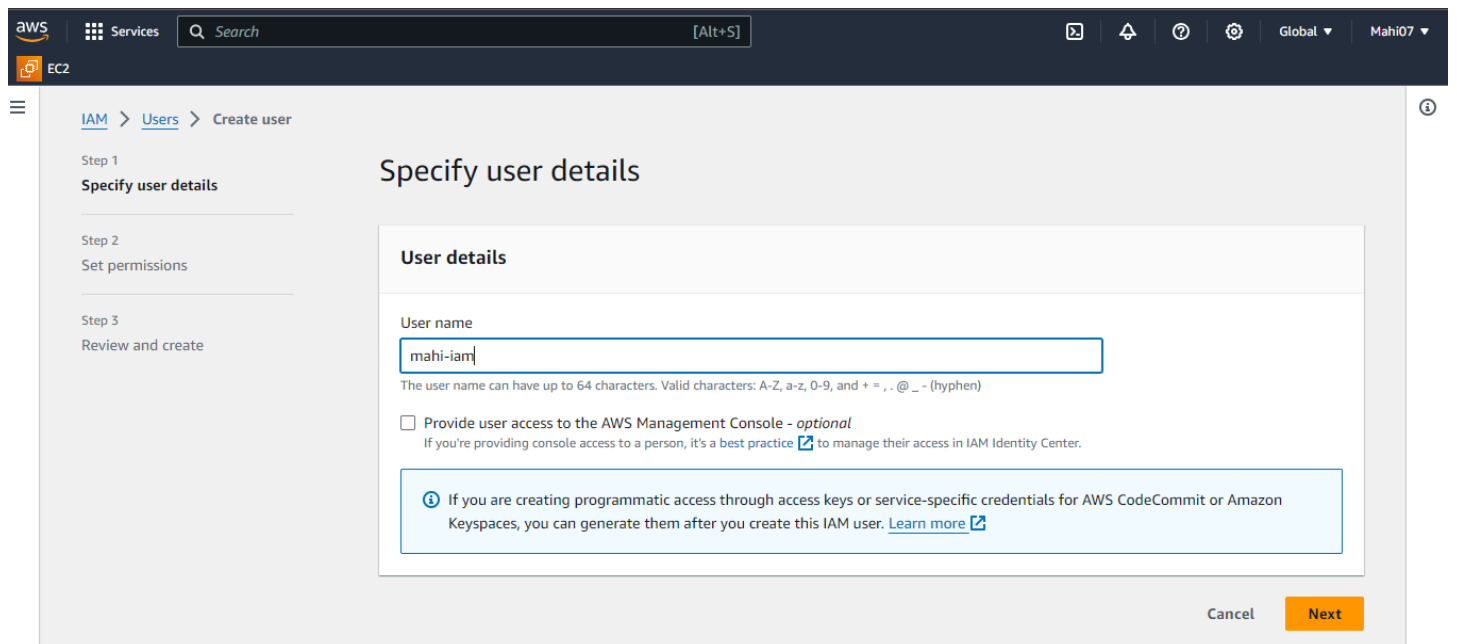
b) You'll be landed on IAM service homepage and select “Users” on the left hand side of the screen.



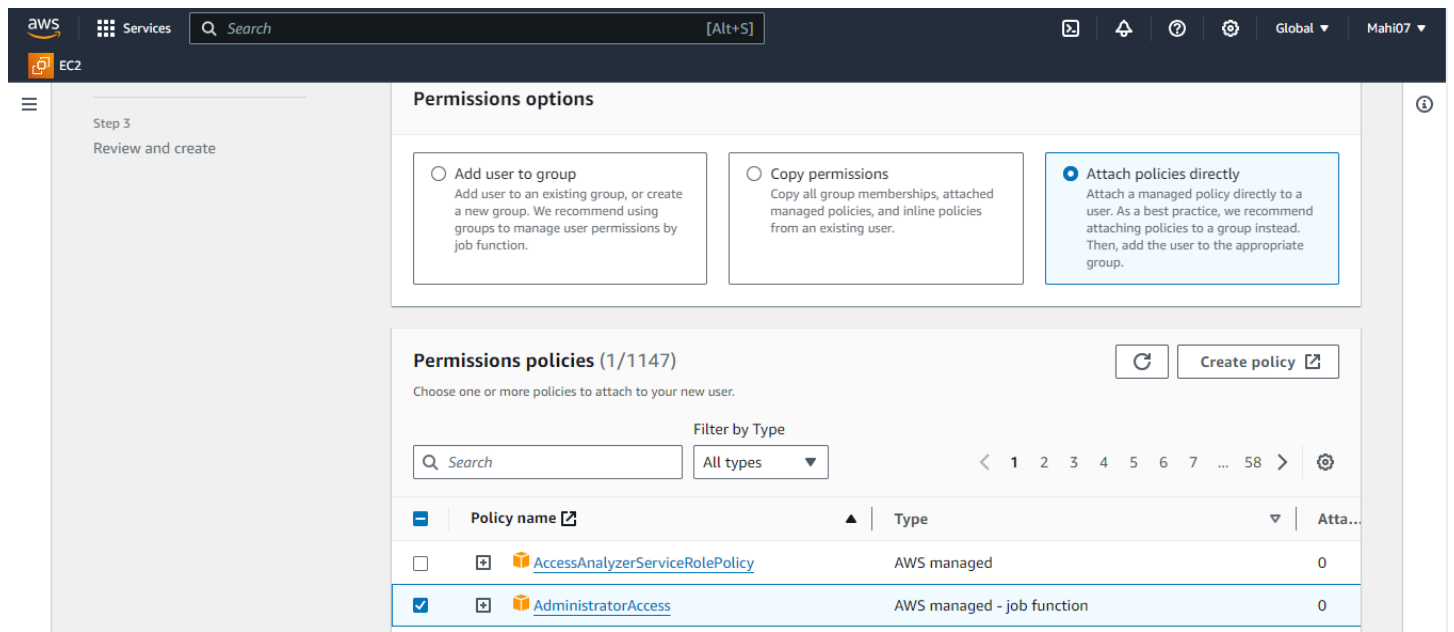
c) Click on “Create user”.



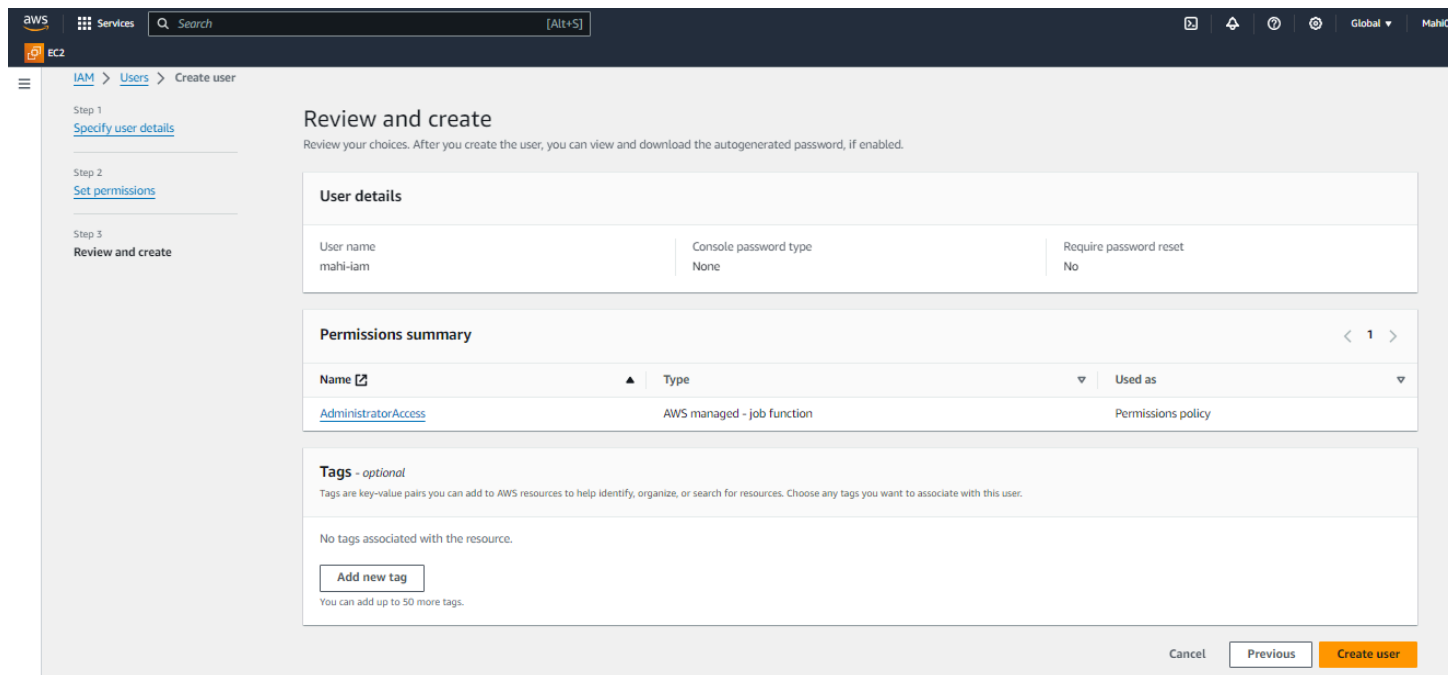
d) Create a username and click on Next.



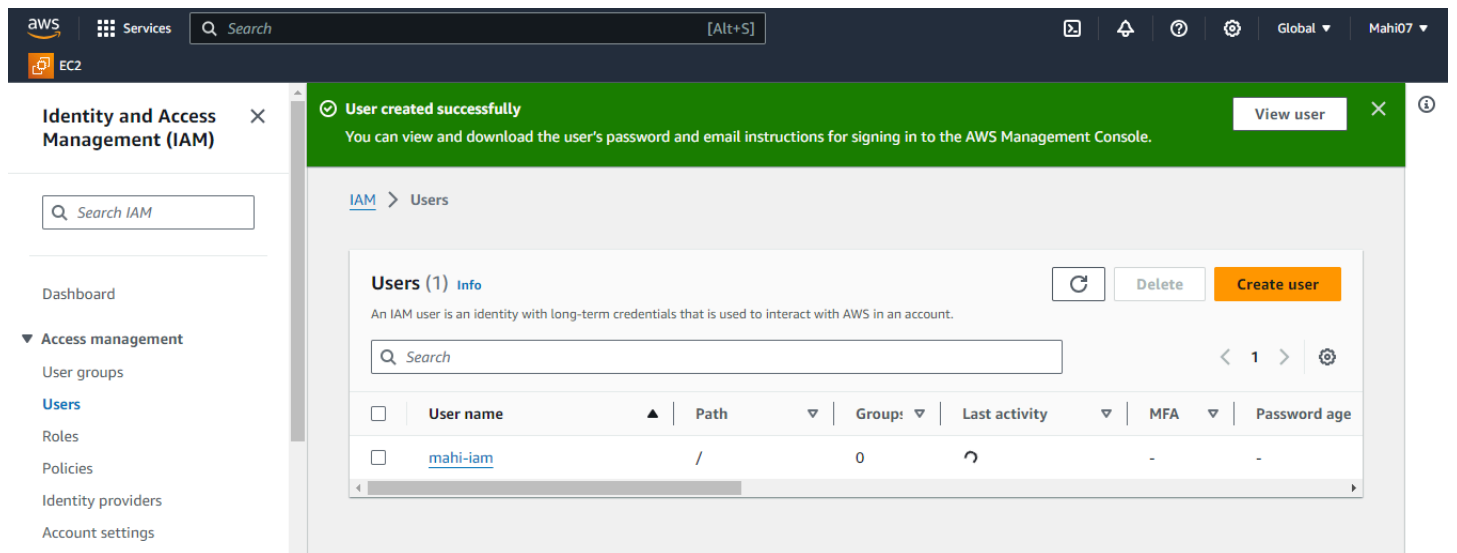
e) Click on **Attach policies directly**, select **AdministratorAccess**, scroll down and click on next at the right bottom as shown above.



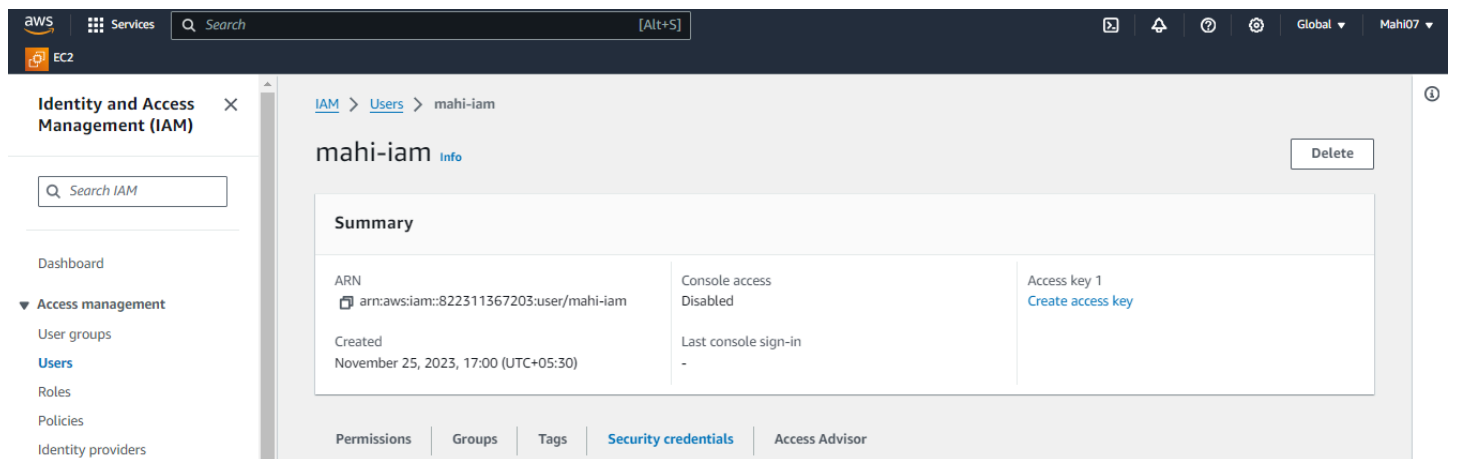
f) Review all the details and click on **“Create user”**.



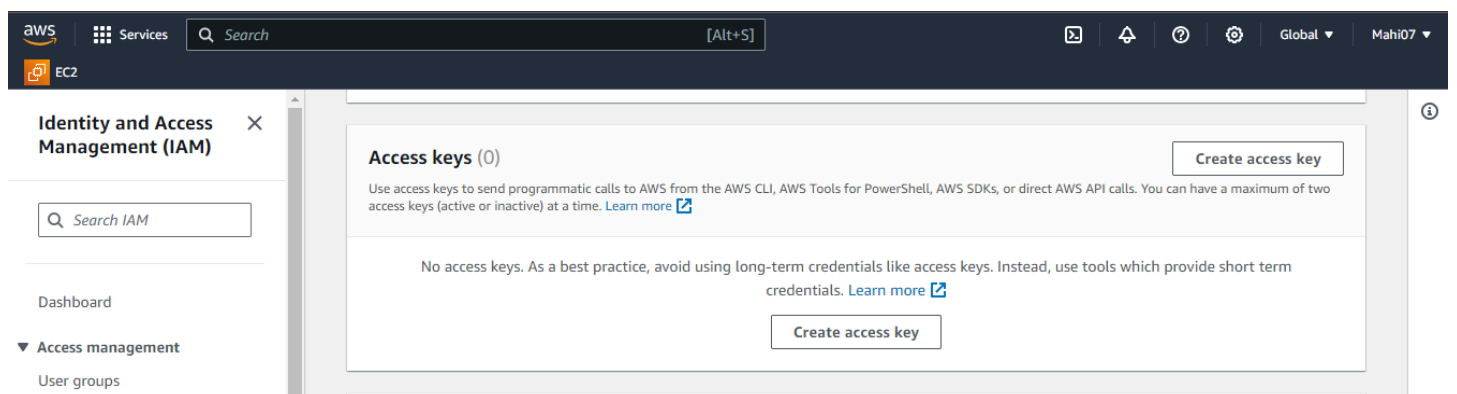
g) A user is created, check the box or click on the username ([mahi-iam](#)).



h) Scroll down and click on **Security credentials**.



i) Scroll down again, go to Access keys and click on **"Create access key"**.



j) Select any one of the following options, check in **'I understand'** and click on **'Next'**.

aws Services Search [Alt+S]

EC2

alternatives

Step 2 - optional
Set description tag

Step 3
Retrieve access keys

Avoid using long-term credentials like access keys to improve your security. Consider the following use cases and alternatives.

Use case

- ☒ **Command Line Interface (CLI)**
You plan to use this access key to enable the AWS CLI to access your AWS account.
- ☐ **Local code**
You plan to use this access key to enable application code in a local development environment to access your AWS account.
- ☐ **Application running on an AWS compute service**
You plan to use this access key to enable application code running on an AWS compute service like Amazon EC2, Amazon ECS, or AWS Lambda to access your AWS account.
- ☐ **Third-party service**
You plan to use this access key to enable access for a third-party application or service that monitors or manages your AWS resources.
- ☐ **Application running outside AWS**
You plan to use this access key to authenticate workloads running in your data center or other infrastructure outside of AWS that needs to access your AWS resources.
- ☐ **Other**
Your use case is not listed here.

Alternatives recommended

- Use [AWS CloudShell](#), a browser-based CLI, to run commands. [Learn more](#)
- Use the [AWS CLI V2](#) and enable authentication through a user in IAM Identity Center. [Learn more](#)

Confirmation

☒ I understand the above recommendation and want to proceed to create an access key.

Cancel Next

k) Enter the a description tag value to the key and click on Create access key.

aws Services Search [Alt+S]

EC2

IAM > Users > mahi-iam > Create access key

Step 1
[Access key best practices & alternatives](#)

Step 2 - optional
Set description tag

Step 3
Retrieve access keys

Set description tag - optional [Info](#)

The description for this access key will be attached to this user as a tag and shown alongside the access key.

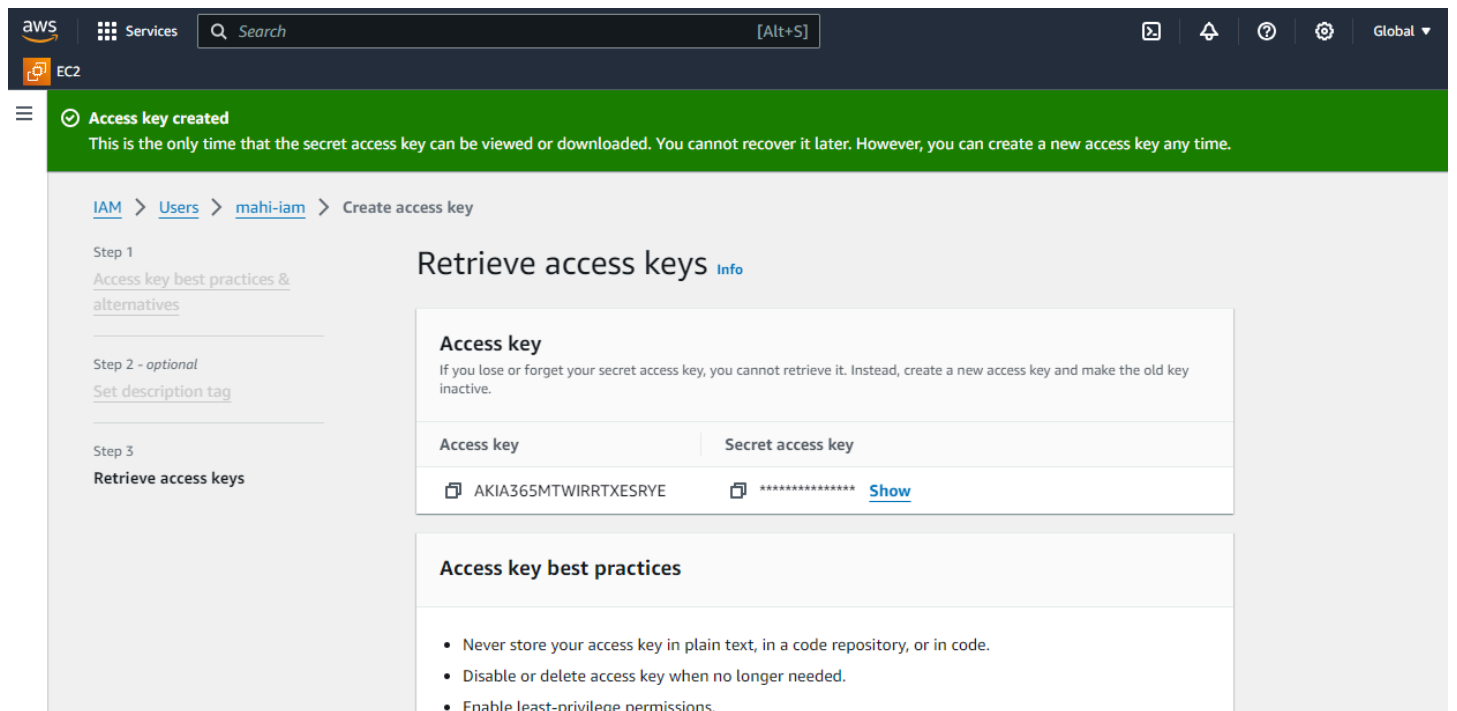
Description tag value
Describe the purpose of this access key and where it will be used. A good description will help you rotate this access key confidently later.

iam-key

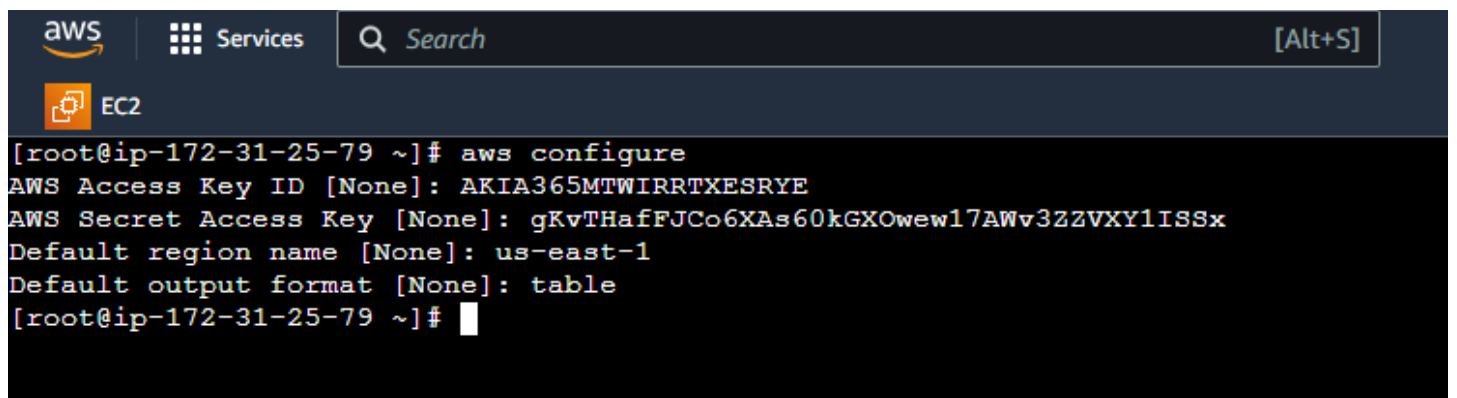
Maximum 256 characters. Allowed characters are letters, numbers, spaces representable in UTF-8, and: _ . : / = + - @

Cancel Previous Create access key

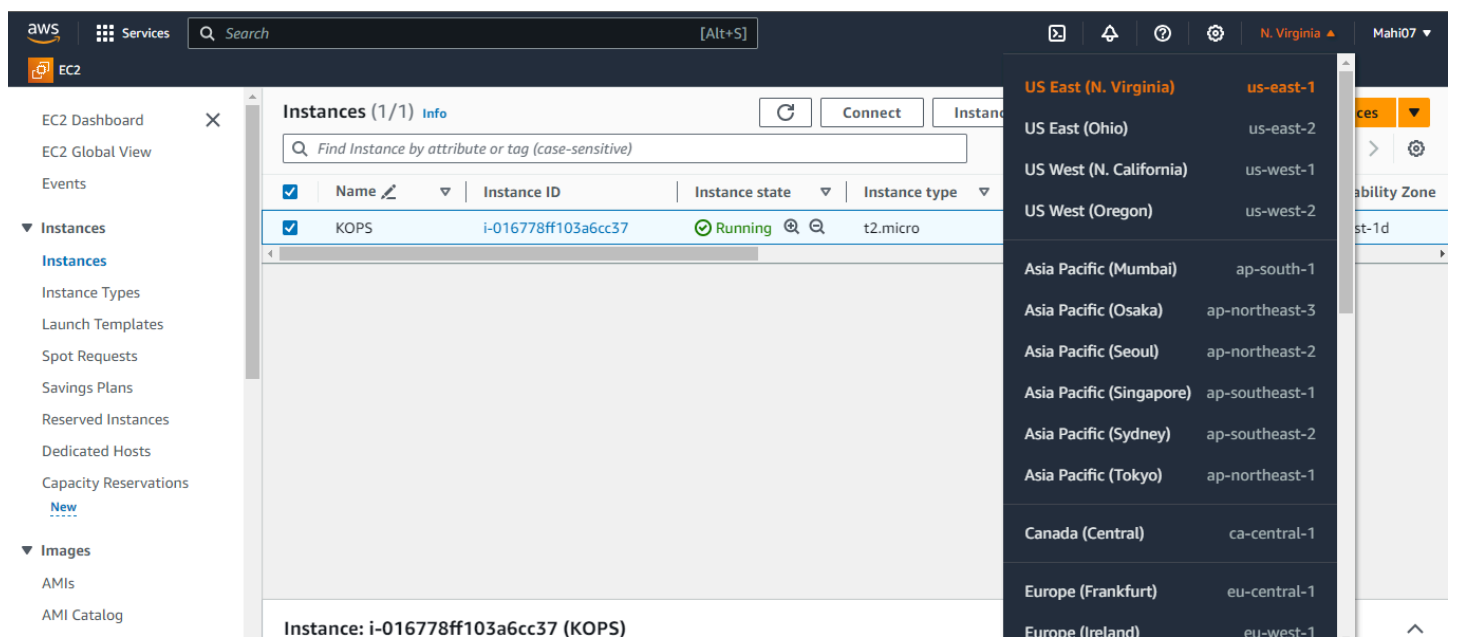
l) Now access key is created, you'll get Access key and Secret access key.



m) It needs to be configured on EC2 instance as shown below.



This image demonstrates how a region needs to be selected based on the availability zone.



So with the help of ‘**aws configure**’ command, the entire KOPS infrastructure setup will be created under my account since it’s my username. Whatever the credentials that you give under “access key and secrets access key” the KOPS infrastructure will be created on that account.

5) Create INFRA setup.

S3 is one of the aws services. It’s abbreviated as Simple Storage Service. All the infrastructure setup information is stored in this S3 service.

TO CREATE BUCKET:

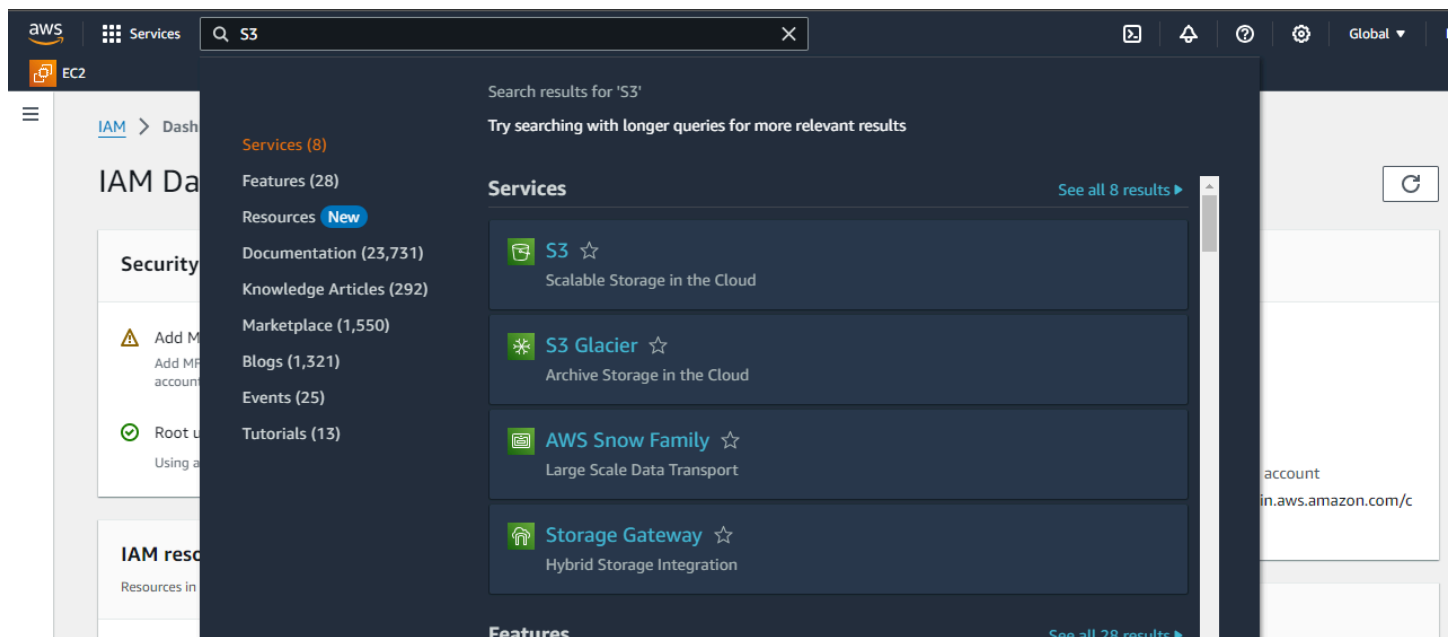
SYNTAX: Cloudprovider Resource Command

aws s3api create-bucket --bucket bucket_name --region region_name

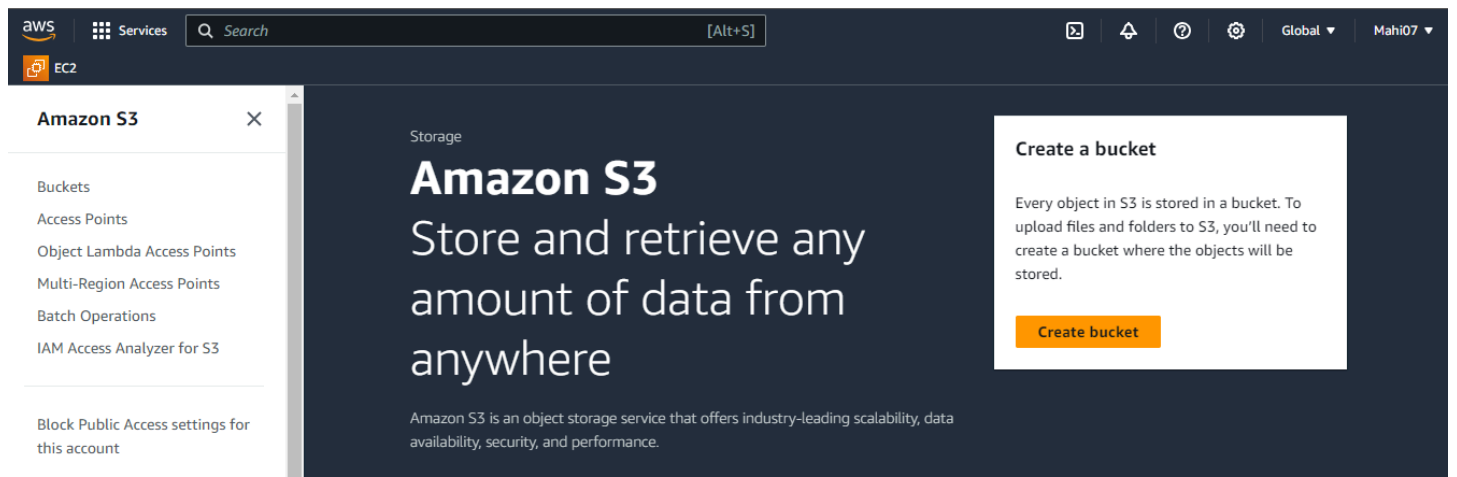
aws s3api create-bucket --bucket mahibucket-1 --region us-east-1

```
[root@ip-172-31-25-79 ~]# aws s3api create-bucket --bucket mahibucket-1 --region us-east-1
-----
|          CreateBucket          |
+-----+
| Location | /mahibucket-1 |
+-----+
[root@ip-172-31-25-79 ~]#
```

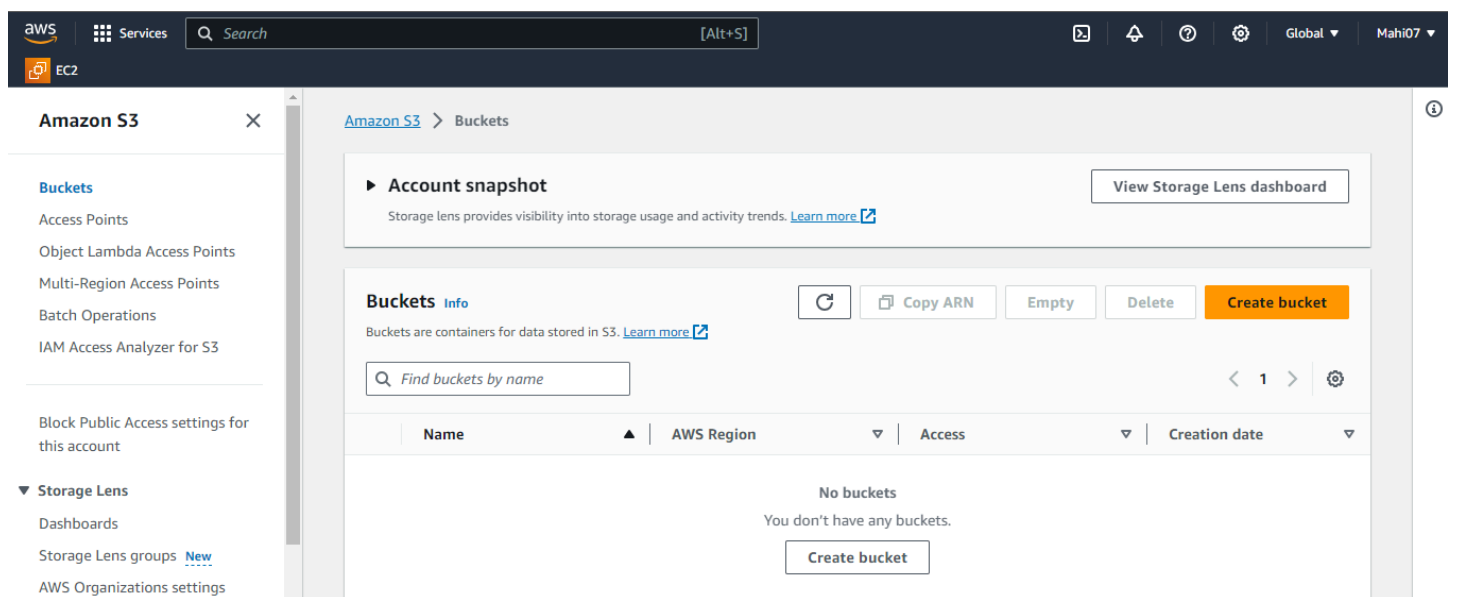
a) Go to AWS console and search for S3 in global search bar.



b) Select S3, click on Buckets on left side of the menu.



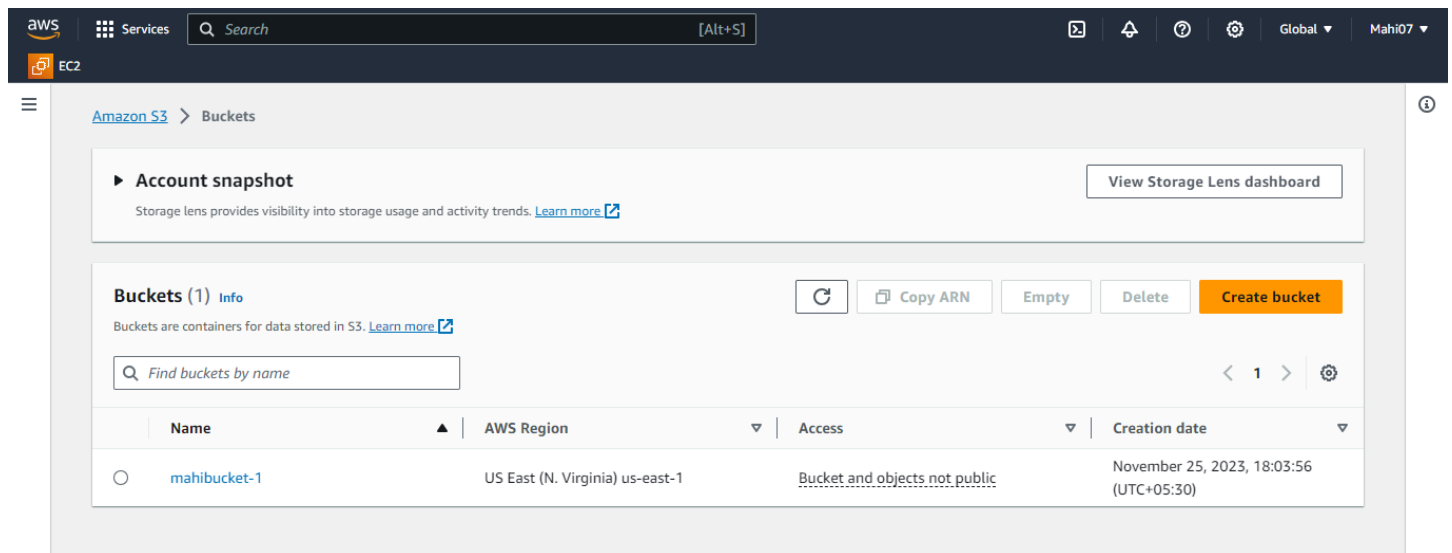
c) If you do not have any buckets it shows No buckets.



d) Create a bucket on EC2 instance using CLI command and then go to aws console, click on refresh icon under buckets and it'll display the results ([mahibucket-1](#)).

```
[root@ip-172-31-25-79 ~]# aws s3api create-bucket --bucket mahibucket-1 --region us-east-1
-----
|          CreateBucket          |
+-----+
| Location | /mahibucket-1 |
+-----+
[root@ip-172-31-25-79 ~]#
```

e) Bucket on AWS console



Here inside the bucket we store the information of Kubernetes cluster such as manager nodes, worker node, pods that are running. In case if the data gets deleted, we've a '**versioning**' concept to protect the data.

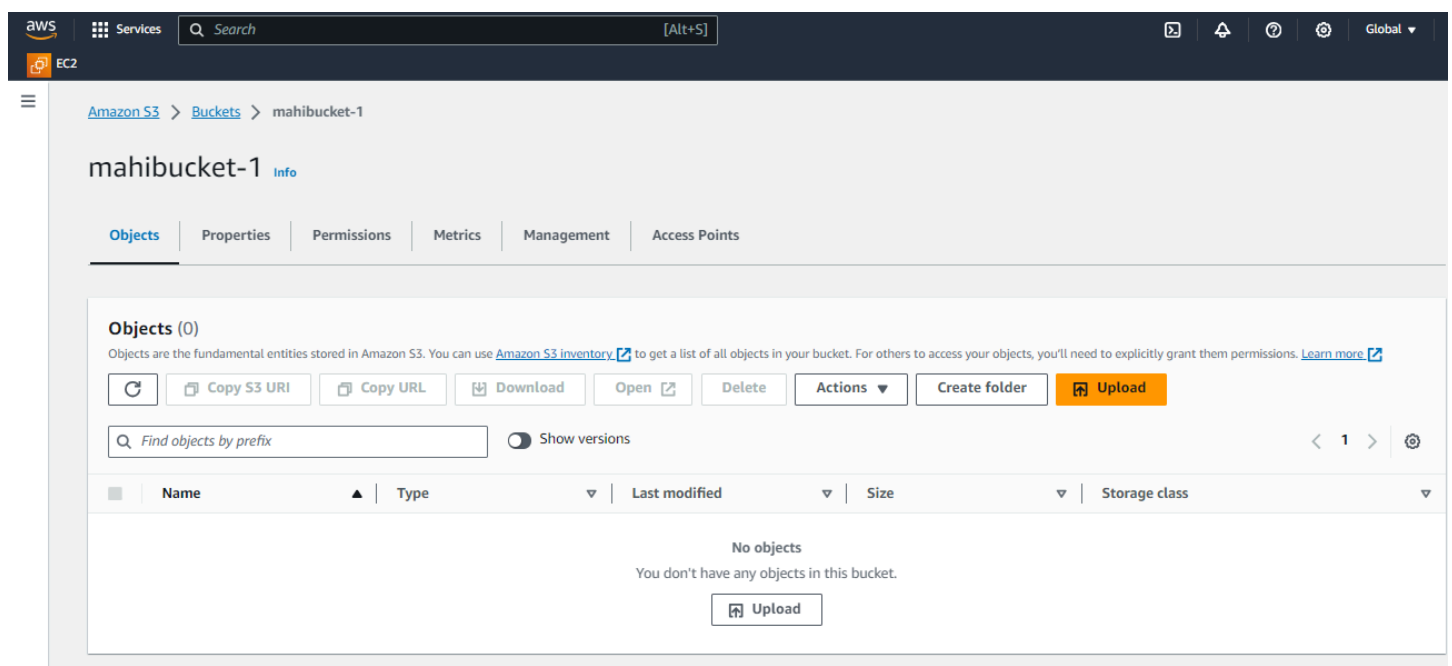
Advantages:

- a) Accidental termination (we can retrieve the file/data)
- b) Maintain multiple versions of a same file.

TO ENABLE THE VERSION:

COMMAND: `aws s3api put-bucket-versioning --bucket mustafask.k8s.local --region us-east-1 --versioning-configuration Status=Enabled`

Go to AWS console and check '**show versions**' option will be added that means versioning is enabled for the bucket.



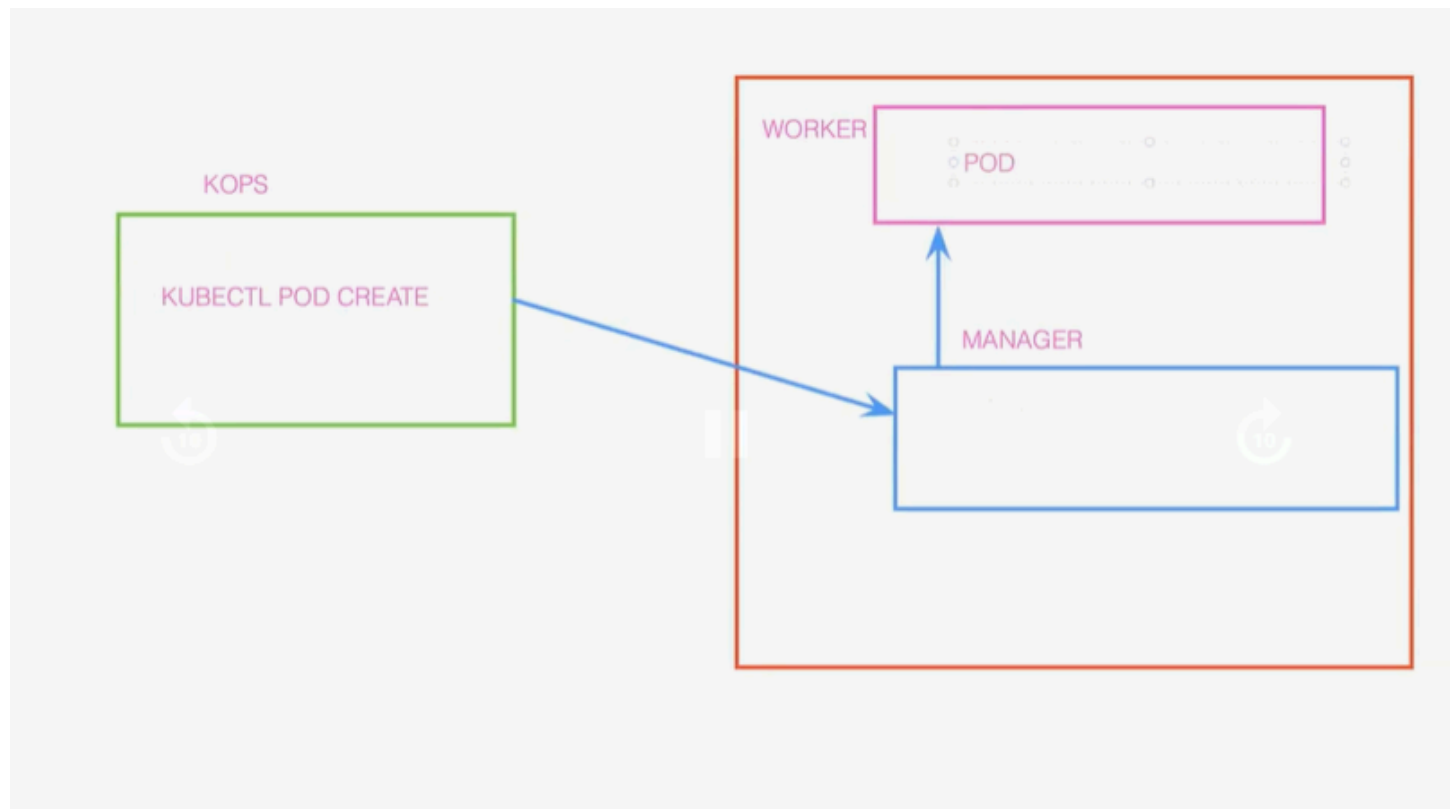
We haven't linked our cluster with the bucket yet. To store all the generated information from cluster inside the bucket we've created we need to link it up.

TO EXPORT CLUSTER DATA INTO BUCKET: `export KOPS_STATE_STORE=s3://mahibucket-1`

GENERATE-KEY:

The reason why we need to generate a key is:

If we give a command to create a Pod through KOPS (server -1), it goes to Manager in the cluster (server-2), manager communicates with the Scheduler to create a pod and scheduler checks with the kubelet and a Pod gets created in the worker node. So, in order to establish a connection between KOPS and Cluster we need a SSH. Therefore we need to generate a key.



Generate a key with command **ssh-keygen**, hit enter three times and now the key is generated.

```
[root@ip-172-31-25-79 ~]# ssh-keygen
Generating public/private rsa key pair.
Enter file in which to save the key (/root/.ssh/id_rsa):
Enter passphrase (empty for no passphrase):
Enter same passphrase again:
Your identification has been saved in /root/.ssh/id_rsa.
Your public key has been saved in /root/.ssh/id_rsa.pub.
The key fingerprint is:
SHA256:t9yq/Hc2e/513zT/U3yTT1yLXDFwmytTaxMDJFEtt7Y root@ip-172-31-25-79.ec2.internal
The key's randomart image is:
+---[RSA 2048]-----+
|      o+=o. |
|      ..o=o|
|      o*+|
|      .+*|
|      S . .o+B=|
|      o o o+EB|
|      . = . oO|
|      . + o. +B|
|      oo=+... B|
+-----[SHA256]-----+
[root@ip-172-31-25-79 ~]#
```

TO CREATE A CLUSTER:

kops create cluster --name mahi99.k8s.local --zones us-east-1a --master-size t2.medium --node-size t2.micro

The cluster name should be in the format of “**name.k8s.local**”. Cluster name: mahi99.k8s.local.

```
[root@ip-172-31-25-79 ~]# kops create cluster --name mahi99.k8s.local --zones us-east-1a --master-size t2.medium --node-size t2.micro
I1125 13:32:33.755495      694 new_cluster.go:251] Inferred "aws" cloud provider from zone "us-east-1a"
I1125 13:32:33.755677      694 new_cluster.go:1168] Cloud Provider ID = aws
I1125 13:32:33.809710      694 subnets.go:185] Assigned CIDR 172.20.32.0/19 to subnet us-east-1a
Previewing changes that will be made:
```

After everything gets installed, cluster configuration gets created.

```
Cluster configuration has been created.

Suggestions:
* list clusters with: kops get cluster
* edit this cluster with: kops edit cluster mahi99.k8s.local
* edit your node instance group: kops edit ig --name=mahi99.k8s.local nodes-us-east-1a
* edit your master instance group: kops edit ig --name=mahi99.k8s.local master-us-east-1a

Finally configure your cluster with: kops update cluster --name mahi99.k8s.local --yes --admin
```

Now, finally configure your cluster with: kops update cluster --name musta.k8s.local --yes --admin

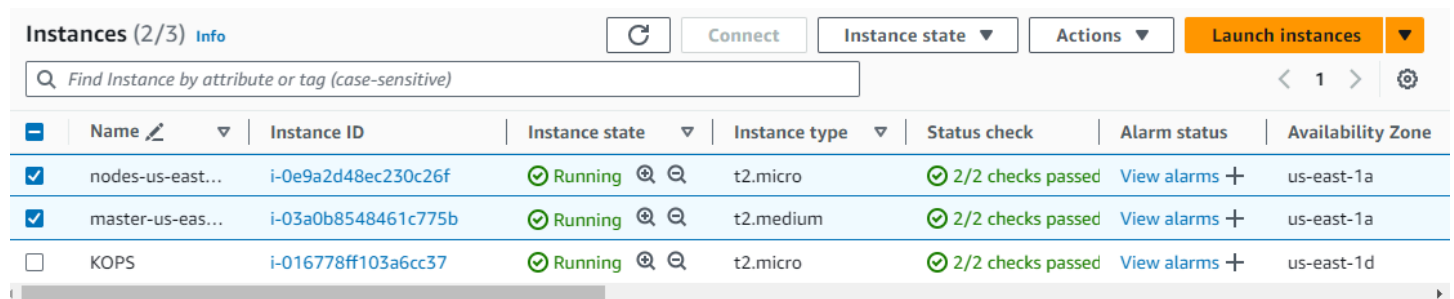
```
kOps has set your kubectl context to mahi99.k8s.local

Cluster is starting. It should be ready in a few minutes.

Suggestions:
* validate cluster: kops validate cluster --wait 10m
* list nodes: kubectl get nodes --show-labels
* ssh to the master: ssh -i ~/.ssh/id_rsa ubuntu@api.mahi99.k8s.local
* the ubuntu user is specific to Ubuntu. If not using Ubuntu please use the appropriate user based on your OS.
* read about installing addons at: https://kops.sigs.k8s.io/addons.

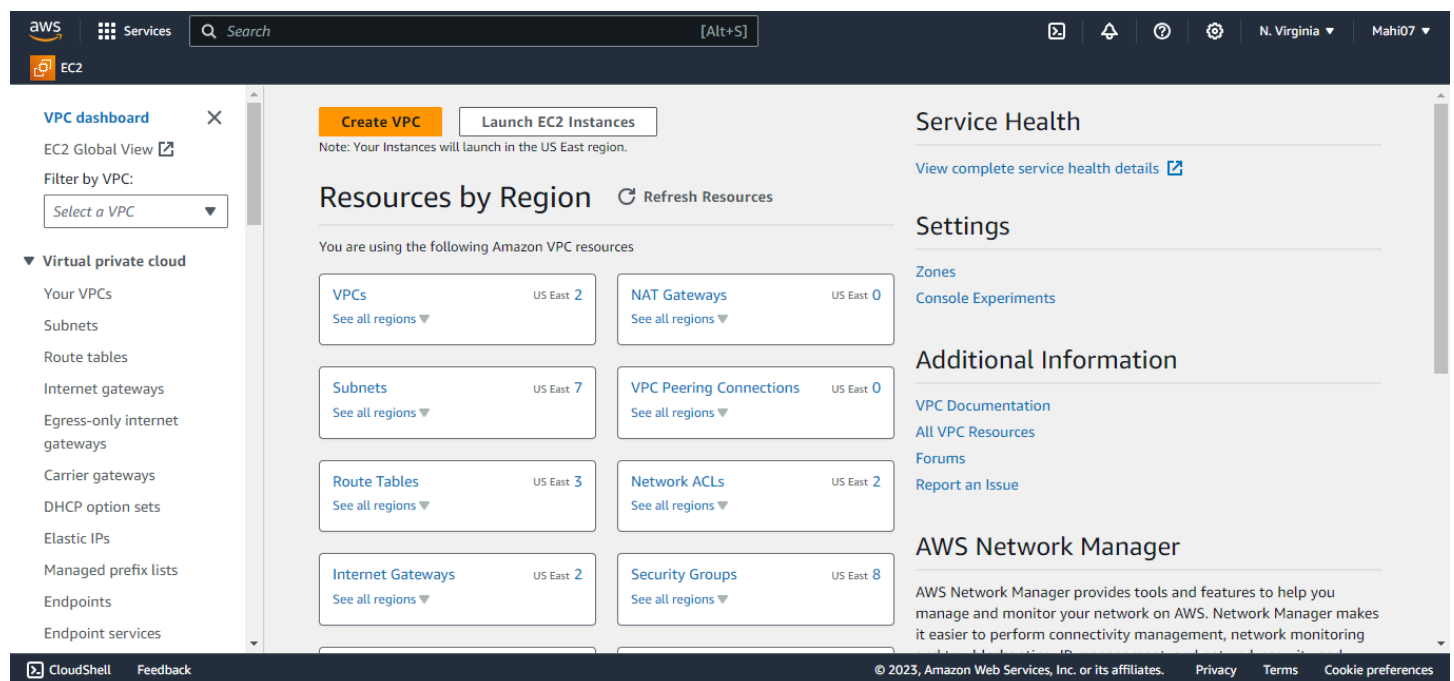
[root@ip-172-31-25-79 ~]#
```

Cluster has started and now you can go to AWS console, refresh the instances list it shows the master and node as shown below.



Instances (2/3) Info							
Find Instance by attribute or tag (case-sensitive)							
	Name	Instance ID	Instance state	Instance type	Status check	Alarm status	Availability Zone
<input checked="" type="checkbox"/>	nodes-us-east...	i-0e9a2d48ec230c26f	Running	t2.micro	2/2 checks passed	View alarms +	us-east-1a
<input checked="" type="checkbox"/>	master-us-eas...	i-03a0b8548461c775b	Running	t2.medium	2/2 checks passed	View alarms +	us-east-1a
<input type="checkbox"/>	KOPS	i-016778ff103a6cc37	Running	t2.micro	2/2 checks passed	View alarms +	us-east-1d

If you search for VPC in global search bar, select it and you'll be landed on this page. You can select the options on the left hand side and check all the infrastructure that's created. To find ASG, Load balancers you can check it on the instance home page without searching any in the search bar.



LABELS, SELECTORS AND NODE SELECTORS

Labels: Label is a identification tag which is used to filter the resources quickly. It's used to organize the kubernetes objects such as pods, nodes etc. You can add multiple labels over the kubernetes objects and add labels like environment, department or anything else according to you. Labels are defined in key-value pairs.

The process of selecting the labels to the objects is called as labels-selectors and those objects in kubernetes can be filtered with the help of label-selectors called as Selectors.

The API currently supports two types of selectors, equality-based and set-based. Label selectors can be made up of multiple selectors that are comma-separated.

Node Selector: Usually, based on the count of pods that are currently running on nodes will be checked and the new pod will be created in the node where it has less number of pods running. A Node selector is used for selecting the nodes. It is used to select a particular node where we need to execute a command.

This is done with the help of labels we mention in the manifest file, we can mention the node label name to create a pod or execute a command in that particular node. So if we run a manifest file, the master node checks the label name and creates a pod in that node.

Ensure that there must be label in the manifest file and if not the manifest file jumps to the next node.

Why Kubernetes services?

Let's say we've deployed an application using Kubernetes deployment. For some reason, if the pod is deleted and a new pod gets automatically created with the help of a kubernetes component called replicaset and the IP address also gets changed for the new pod.

Ex: If the IP address is 172.10.1.20 and the new one might be 172.10.1.22. Then, if the user access the application using the IP address of the old pod which is 172.10.1.20 they wouldn't be able to access it because the IP address is changed to 172.10.1.22 after a new one is created.

i) Service discovery: In order to resolve this issue, we've got Kubernetes services (SVC). We've labels and selectors, so whenever a new pod is created with a new IP address, it still can be accessed if we use the labels to the pod during the creation itself. It helps in tracking the pods and this is called 'Service discovery'.

ii) Expose it to outside world: So, if we create a pod, a container gets created and the application runs inside it and is accessible within the cluster only. However, the reason why we were unable to access the application after creating a pod is because we haven't 'expose' our pod and in order to expose the pod we need to use the kubernetes services.

iii) Load Balancing: When you try to access the application, based on the incoming requests from the outside world, the load balancing distributes the requests equally to the available pods in the cluster.

Types of Kubernetes Services

A service is a method of exposing the pod from the cluster.

Different types of Kubernetes services are the following: Cluster IP, External Name, NodePort and Load Balancer.

1) **Cluster IP:** This service is used to access the application within the cluster only and not exposed externally. It provides a stable IP address and DNS name for the pods within the cluster.

a) Manifest file for POD:

```

---
apiVersion: v1
kind: Pod
metadata:
  name: pod-1
  labels:
    app: swiggy
spec:
  containers:
    - name: cont-1
      image: shaikmustafa/cycle
      ports:
        - containerPort: 80
~
~

```

b) Manifest File for ClusterIP:

```

---
apiVersion: v1
kind: Service
metadata:
  name: thala
spec:
  type: ClusterIP
  selector:
    app: swiggy

  ports:
    - port: 80
      targetPort: 80
~

```

c) Execution result of Manifest file:

```
[root@ip-172-31-47-67 manifests]# kubectl get svc
```

NAME	TYPE	CLUSTER-IP	EXTERNAL-IP	PORT(S)	AGE
aws	ClusterIP	100.66.101.42	<none>	80/TCP	31m
gcp	ClusterIP	100.66.188.180	<none>	80/TCP	9m23s
kubernetes	ClusterIP	100.64.0.1	<none>	443/TCP	3h2m

2) **External Name:** This service is similar to Cluster IP (internally within the server) but it does have a DNS name (like amazon.com or facebook.com) instead of labels and selectors.

If we compare the above image, it only has a Cluster IP and this one has a Cluster IP and also a external dns name which can be used to access the application by exposing the service with externalName but we need to buy SSL certificates either from Go Daddy or Route 53.

Manifest File for ExternalName:

```
---
apiVersion: v1
kind: Service
metadata:
  name: gcp
spec:
  type: ExternalName
  externalName: mahi99.k8s.devops.com
  selector:
    app: uber

  ports:
    - port: 80
      targetPort: 80
```

Execution result of Manifest file:

```
[root@ip-172-31-47-67 manifests]# kubectl apply -f service.yml
Warning: resource services/gcp is missing the kubectrl.kubernetes.io/last-applied-configuration annotation which is required by kubectl apply. k
ubectl apply should only be used on resources created declaratively by either kubectl create --save-config or kubectl apply. The missing annota
tion will be patched automatically.
service/gcp configured
[root@ip-172-31-47-67 manifests]# kubectl get svc
NAME         TYPE          CLUSTER-IP    EXTERNAL-IP    PORT(S)    AGE
aws          ClusterIP     100.66.101.42 <none>         80/TCP     37m
gcp          ExternalName  <none>         mahi99.k8s.devops.com  80/TCP     16m
kubernetes   ClusterIP     100.64.0.1    <none>         443/TCP    3h9m
```

3) **NodePort:** This service is used to expose a service on a static port on each node in the cluster. This service is accessible both within the cluster and externally using the node's IP address and the NodePort. The NodePort should be selected between the range from 30000 to 32767.

Here **80:31232/TCP**, under PORT (S) defines that the application can be accessed externally with the help of NodePort. If you do not assign the port, it gets automatically assigned within the range if we do not mention in the manifest file.

Manifest file for NodePort:

```
---
apiVersion: v1
kind: Service
metadata:
  name: thala
spec:
  type: NodePort
  selector:
    app: swiggy

  ports:
    - port: 80
      targetPort: 80
      nodePort: 31212
~
```

Execution result of Manifest file:

```
[root@ip-172-31-47-67 manifests]# kubectl create -f service.yml
service/gcp created
[root@ip-172-31-47-67 manifests]# kubectl get svc
NAME                TYPE          CLUSTER-IP    EXTERNAL-IP    PORT(S)          AGE
gcp                  NodePort      100.70.132.74 <none>         80:31232/TCP     7s
kubernetes           ClusterIP     100.64.0.1    <none>         443/TCP          3h19m
[root@ip-172-31-47-67 manifests]#
```

4) **Load Balancer:** This service provides a way to expose the service externally using a cloud provider's load balancer. This service is used when there's an incoming high traffic load, requires automatic scaling and load balancing capabilities.

The main use of load balancer is it distributes the traffic equally among the worker nodes.

Ex: In case if we've 2 worker nodes, if 4K people are accessing the application and in case if the requests go to only one server (worker node -1) the server may get terminated. If we use the load balancer the requests first hit by the LB and it distributes the traffic equally among the available servers. It improves the accessibility of application to many users without any latency.

Therefore, if the LB service is used to expose the application for access then it creates a Nodeport and a DNS name to access the application. You can use either the NodePort or DNS to access the application.

Manifest file of LoadBalancer:


```

---
apiVersion: v1
kind: Service
metadata:
  name: thala
spec:
  type: LoadBalancer
  selector:
    app: swiggy

  ports:
    - port: 80
      targetPort: 80

```

Execution result of Manifest file:

```

[root@ip-172-31-47-67 manifests]# kubectl create -f service.yml
service/thala created
[root@ip-172-31-47-67 manifests]# kubectl get svc

```

NAME	TYPE	CLUSTER-IP	EXTERNAL-IP	PORT(S)	AGE
kubernetes	ClusterIP	100.64.0.1	<none>	443/TCP	4h2m
thala	LoadBalancer	100.65.118.142	a6fbbbc7043c3430caa0c02f99e569c4-438340116.us-east-1.elb.amazonaws.com	80:32077/TCP	6s

If you take a load balancer (LB) service to expose our service, a load balancer gets created in the aws cloud as shown below.

EC2 > Load balancers

Load balancers (1/2) ↻ Actions ▼ Create load balancer ▼

Elastic Load Balancing scales your load balancer capacity automatically in response to changes in incoming traffic.

< 1 > ⚙

<input type="checkbox"/>	Name	DNS name	State	VPC ID	Availability Zones
<input type="checkbox"/>	api-jang-k8s-local-hee...	api-jang-k8s-local-hee5ob...	–	vpc-0ef9d37c5eacba9a9	us-east-1b (use1-az1)
<input checked="" type="checkbox"/>	a6fbbbc7043c3430caa...	a6fbbbc7043c3430caa0c0...	–	vpc-0ef9d37c5eacba9a9	us-east-1b (use1-az1)

Once you click on the created load balancer, then you'll move to this page. Once the status turns to 2 of 2 instances in service then you can either access the app from NodePort or DNS name.

EC2 > Load balancers > a6fbbbc7043c3430caa0c02f99e569c4

a6fbbbc7043c3430caa0c02f99e569c4 Actions ▼

▼ Details

Load balancer type Classic	Status 2 of 2 instances in service	VPC vpc-0ef9d37c5eacba9a9	Date created November 30, 2023, 16:35 (UTC+05:30)
Scheme Internet-facing	Hosted zone Z35SXDOTRQ7X7K	Availability Zones subnet-0685bce9014f75e09 us-east-1b (use1-az1)	

NOTE: In the manifest file, under spec, type and ports must be in a same line or else it'll throw an error. For further reference while creating manifest file you can follow the above images.

Components of services

A service is defined with the help of a Manifest file where we define it's properties and specifications.

- 1) Selector: A selector is used to select the label for the objects that we define in our manifest file.
- 2) Port:
- 3) Target Port: It's
- 4) Type: It defines the type of service that we are defining like cluster IP, NodePort, Load balancer or external name.

SERVICE COMMANDS:

To create a service:

To check a specific service: **kubectl get svc service_name**

To check list of services: **kubectl get service [or] kubectl get svc**

```
[root@ip-172-31-47-67 manifests]# kubectl get svc
NAME      TYPE      CLUSTER-IP      EXTERNAL-IP      PORT(S)      AGE
dhoni     LoadBalancer  100.67.72.139    ad26752283da24644b194cac8e9f5a51-1589652433.us-east-1.elb.amazonaws.com  80:32132/TCP  5s
kubernetes ClusterIP    100.64.0.1       <none>            443/TCP      4h44m
thala     ClusterIP    100.70.64.226    <none>            80/TCP      13m
```

To get full info about a service: **kubectl get svc service_name -o wide**

To describe a service: **kubectl describe svc service_name**

```
[root@ip-172-31-47-67 manifests]# kubectl describe svc gcp
Name: gcp
Namespace: default
Labels: <none>
Annotations: <none>
Selector: app=uber
Type: NodePort
IP Family Policy: SingleStack
IP Families: IPv4
IP: 100.70.132.74
IPs: 100.70.132.74
External Name: mahi99.k8s.devops.com
Port: <unset> 80/TCP
TargetPort: 80/TCP
NodePort: <unset> 31232/TCP
Endpoints: <none>
Session Affinity: None
External Traffic Policy: Cluster
Events: <none>
```

To delete service: **kubectl delete svc service_name**

To delete all services: **kubectl delete svc --all**

NAME	TYPE	CLUSTER-IP	EXTERNAL-IP	PORT(S)	AGE
dhoni	LoadBalancer	100.67.72.139	ad26752283da24644b194cac8e9f5a51-1589652433.us-east-1.elb.amazonaws.com	80:32132/TCP	5s
kubernetes	ClusterIP	100.64.0.1	<none>	443/TCP	4h44m
thala	ClusterIP	100.70.64.226	<none>	80/TCP	13m

```
[root@ip-172-31-47-67 manifests]# kubectl delete svc --all
service "dhoni" deleted
service "kubernetes" deleted
service "thala" deleted
```

Replicas in Kubernetes

Before Kubernetes other tools didn't provide important and customized features like 'scaling' and 'replication'. However, when Kubernetes was introduced, replication and scaling were the premium features that increased the popularity and usage of this container orchestration tool.

Why replicas in Kubernetes?

Earlier to this concept, we used a service to expose a pod to access the application. However, in case if there's heavy traffic and if the pod gets deleted, container also gets deleted and we can't access the application, replicas will be useful to overcome this drawback.

Here the self-healing isn't available. To overcome this drawback of not accessing the application when the pod is deleted we use some of the Kubernetes replicas like RC, RS, Deployments, DaemonSets etc.

1) **Replication:** Replication means that in case if a pod gets deleted a new pod will be created automatically and this helps to reduce the downtime of an application.

Ex: If there are 3 pods that are running and the incoming requests goes equally to all the pods but in case for some reason if a pod gets deleted then a new one will be created with the help of replication.

2) **Scaling:** Scaling means in case if the load is increased on the application, then the kubernetes creates new pods according to the load on the application. Due to this the new traffic would be routed to the new pod and this will avoid the termination of application.

Ex: If a pod's CPU utilization or memory utilization reaches to 60% (traffic), before the pod gets terminated a new pod needs to be created. The application will be accessed on the new pod with the data of same old pod and the traffic goes to the old pod and not to the new pod.

KUBERNETES CONTROLLERS

In Kubernetes, we've different types of Kubernetes controllers such as ReplicationController, ReplicaSet, Deployment, DaemonSet etc which are used to manage the lifecycle of a pods within a cluster.

The list of controllers in Kubernetes are the following:

- | | |
|------------------------------------|--|
| 1) Replication controller | 10) Node Controller |
| 2) Replicaset | 11) Persistent Volume Claim (PVC) Controller |
| 3) Deployment | 12) EndpointSlice controller |
| 4) Daemonset | 13) Ingress controller |
| 5) Job | 14) Service controller |
| 6) CronJob | 15) Vertical Pod Autoscaler (VPA) |
| 7) StatefulSet | 16) Namespace controller |
| 8) Horizontal Pod Autoscaler (HPA) | 17) Pod Disruption Budget (PDB) controller |
| 9) Vertical Pod Autoscaler (VPA) | 18) Resource Quota controller |

1) Replication Controller: Replication controller is an object in Kubernetes which is the first version of controller which works only on equality-based selectors only.

In Kubernetes architecture, the controller will only check if a certain number of pods are running or not as per the requirement and it doesn't take care whether they're running equally or unequally in the worker nodes. In case if a pod gets deleted a new one will be created and in order to create that the controller will tell to the scheduler and the scheduler will communicate with kubelet to get the what's happening with the worker nodes about the count and all and then the scheduler decides in which worker node (if there are 2 or more) a pod needs to be created.

- Replication controller can run specific number of pods as per the requirement.
- It'll ensure that all the pods are up and running.

- This will create a pod if the count is less and delete a pod if the count is more as per the requirement. We need to mention the replicas in the YAML file to create the pods.
- It manages the pod-life cycle.
- If a pod is failed, terminated or deleted, replication controller will create a new one. These use labels to identify the pods that they are managing.
- In case if the node gets deleted, whatever the nodes are present in the deleted node, those pods will be created in another node.

In the manifest file of '**rc.yml**', replicas (2) indicates the number of pods needs to be created. We mention the template inside the specifications of replication controller. The selector is declared as flm and also same as app: flm as the label. Whenever a pod is deleted, two new pods will be created if the selector matches (flm) with the label (flm) with the help of the template. The labels must be the same under selector and also under the labels. If not replication controller doesn't work and no pods will be created.

So, if we **create a pod** - - > **it creates a container** - - > **application is accessed**. In the same way if we **create a replication controller** - - > **it creates a pod**, and - - > **the pod creates a container** and - - > **the application can be accessed**.

Therefore, in the manifest file, we mention container details inside the specifications (**spec**) of a pod and in the same way we mention pod details under the specifications (**spec**) of a controller which is under template and inside the template we mention the specifications of the container.

There's no problem with the RC but RS has been introduced.

You can scale up or scale down the replicas in order to create the required number of pods with the help of a replication controller.

Commands of Replication Controller:

- 1) To check the list of replication controllers: **kubectl get rc**
- 2) To check the full info of a replication controller: **kubectl get rc -o wide**
- 3) To get full info about all replication controllers: **kubectl describe rc**

This will give full info as shown below.

- 4) To delete a replication controller:

a) kubectl delete rc rc_name

If you delete the controller, all the pods will be deleted you can't be able to access the application at all.

However, if you want to access the application even the RC is deleted then you need to use the below command.

b) kubectl delete rc my-rc --cascade=orphan

If you delete the pods directly you can't access the application as we deleted the RC above.

5) To scale up the replicas: **kubectl scale rc rc_name --replicas count**

kubectl scale rc my-rc --replicas 5

To increase we need to give the overall count including the running pods. If we've 2 pods already running and if we need 3 more pods so we need to mention the replicas as 5 then, another 3 pods will be created.

6) To scale down the replicas: **kubectl scale rc rc_name --replicas count**

kubectl scale rc my-rc --replicas 2

To decrease a certain number of pods, we need to give the remaining count from the overall pods running. If we've 5 pods overall running and if you want to delete 3 pods so you need to give 2 so the 3 pods will be deleted.

2) ReplicaSet: ReplicaSet works is an object in Kubernetes and it's an advanced version of replication controller which works on both equality based controllers and also on set based controllers.

RC vs RS: Both works on a same mechanism which is creating pods whenever they get deleted but the only difference is ReplicaSet works on both equality-based controllers and set-based controllers.

Difference between RC and RS in manifest file:

ReplicationController

apiVersion: **v1**

kind: **ReplicationController**

selector: **directly label (app: flm)**

ReplicaSet

apiVersion: **apps/v1**

kind: **ReplicaSet**

selector: **matchLabels and then label**

ReplicaSet Commands:

1) To check the list of replica sets: **kubectl get rs**

2) To check the full info of a replica set: **kubectl get rs -o wide**

3) To get full info about all replication controllers: **kubectl describe rs**

4) To delete a replicaset: **kubectl delete rs rs_name**

5) To get rs in yaml: **kubectl get rs -o yaml (check this out)**

Why Deployment?

ReplicaSet is a lower-level object which only focuses on maintaining a desired number of replica pods. We've a drawback, where we can't rollback to a specific version and also we don't have rolling updates. Hence, we moved deployment as it's a higher-level object which provide functionality as mentioned rollback and rolling updates.

3) Deployment:

- Deployment has all the features same as replicaset and also additional features like rolling updates and rollback to a specific version is it's specialty.
- Deployment will create a replicaset, replicaset creates a pod, a pod creates a container and container runs the application.
- If a deployment is deleted, it deletes the replicaset and then replicaset will delete the pods and we can access the application.
- The best part of deployment is it has no downtime and you can update the container image and also configuration of the application.
- It also provides a features such as versioning, where it helps to track the history of the updates of the application.
- It has a pause feature which helps to suspend the updates to the application.
- Scaling can be done manually or automatically based on metrics such as CPU utilization or requests per second.

Deployment Commands:

1) To create a deployment: **kubectl create deployment dep_name --image=image_name --replicas=3**

kubectl create deployment dep-1 --image=shaikmustafa/dm --replicas=3

2) To check the list of deployment : **kubectl get deployment**

3) To check the full info of a specific deployment: **kubectl get deployment dep_name -o wide**

4) To get full info about all deployments: **kubectl describe rs**

5) To delete a deployment: **kubectl delete deployment dep_name**

6) To update an image: **kubectl set image deployment/dep_name cont_name=image_name**

kubectl set image deployment/flm cont-1=shaikmustafa/dm or nginx

Rollout Commands:

7) To check the status of rollout: `kubectl rollout status deployment/dep_name`

kubectl rollout status deployment/flm

8) To check history of deployment: `kubectl rollout history deployment/dep_name`

kubectl rollout history deployment/flm

Whenever a rollout is done the order will be changed as shown below.

9) To rollout to a specific version: `kubectl rollout undo deployment/dep_name --to-revision=number of version`

kubectl rollout undo deployment/flm --to-revision=2/3 likewise

10) To pause a deployment: `kubectl rollout pause deployment/dep_name`

kubectl rollout pause deployment/flm

After pausing the deployment, the pods will be in running, deployment will also be in running. However, no new updates will be reflected on the application.

11) To resume a deployment: `kubectl rollout resume deployment/dep_name`

kubectl rollout resume deployment/flm

Deployment Manual Scaling Commands: Imperative and declarative.

12) To scale up the deployment manually: `kubectl scale deployment /dep_name --replicas=count`

kubectl scale deployment/flm --replicas=3

13) To scale down the deployment manually: `kubectl scale deployment /dep_name --replicas=count`

kubectl scale deployment/flm --replicas=1

Why Scaling?

Usually, if the incoming requests (incoming traffic) are more than the pod capacity, then the pod may get deleted and can't access the application and there will be a downtime. However, since we've RC, RS and deployment this issue was resolved as they create new pods. But this is a manual way of scaling because we do mention the replicas either through imperative or declarative.

Deployment Automatic Scaling: Auto scaling can be done in two ways. VPA and HPA.

4) Vertical Scaling/Vertical Pod Autoscaler (VPA): The vertical scaling is nothing but maintaining a single pod and increasing its CPU and memory based on the future upcoming requests as per the client requirement.

Scenario: Let's say we've a pod with 2 CPU's and 4 GB RAM, and it can only handle 1000 requests. If a client says that in the next week we'll get 5000 incoming requests for the same pod and since it can't withhold the load we'll be increasing the CPU and memory prior to 8 CPU's and 16GB RAM. Here we're increasing the pod capacity not pods.

Here in case if the pod gets crashed, a new pod will be created but there will be a downtime. Let's say the downtime is 3 minutes and during this downtime if users has to access the application, the request doesn't go nowhere, as we don't have another pod in this vertical scaling concept and as we don't have a load balancer implemented to distribute the requests to another pod during this downtime. This is drawback we've so we don't use this scaling during real time and we only prefer horizontal scaling.

5) Horizontal Scaling/Horizontal Pod Autoscaler (HPA): The horizontal scaling refers to maintaining multiple pods and creation of multiple pods whenever the CPU utilization is reached more than the set threshold limit.

In easy way, if the threshold limit is exceeded a new pod is created and if it is less it'll be deleted.

Scenario: Let's say, here we've 5 pods with same configuration which is with 2 CPU's and 4GB RAM.

If the CPU utilization of a pod reaches to 30%, then a new pod needs to be created. If we mention this condition in the command a new pod will be automatically created after the threshold limit is exceeded (>30%) and once the threshold limit is lowered (< 30%) then the created pod will be automatically deleted and we'll have the old pod only. This is how the Auto scaling in terms of CPU utilization.

Here also there would be a bit of delay/downtime during the creation of a new pod, but we've a new pod that'd be creating as a backup. However, during the creation of a new pod, the new incoming requests will be distributed amongst the pods with the help of a load balancer.

Commands based on CPU Utilization:

14) To : `kubectl autoscale deployment/dep_name --min= --max= --cpu-percent=count`

kubectrl autoscale deployment/flm --min=3 --max=6 --cpu-percent=40%

6) DaemonSet:

It's one of the objects in the replicas in Kubernetes which helps in creating a single pod whenever a new worker node is launched/created in the Kubernetes Cluster.

To view the daemonset: **kubectrl get daemonset**

Deployment Strategies in Kubernetes:

The techniques that are used to manage the rollout and scaling of applications within the Kubernetes cluster are called as Deployment strategies.

There are 4 types of deployment strategies in Kubernetes. They are the following.

- 1) Canary Deployment
- 2) Recreate Deployment
- 3) Rolling update Deployment
- 4) Blue-Green Deployment

1) Canary Deployment: If a new version of the application is released into market as a segment wise or phase wise, then it is called as Canary deployment. Like releasing the new version to only a few percentage of users and later based on the reviews, usage and if everything works fine then it is released to few more users.

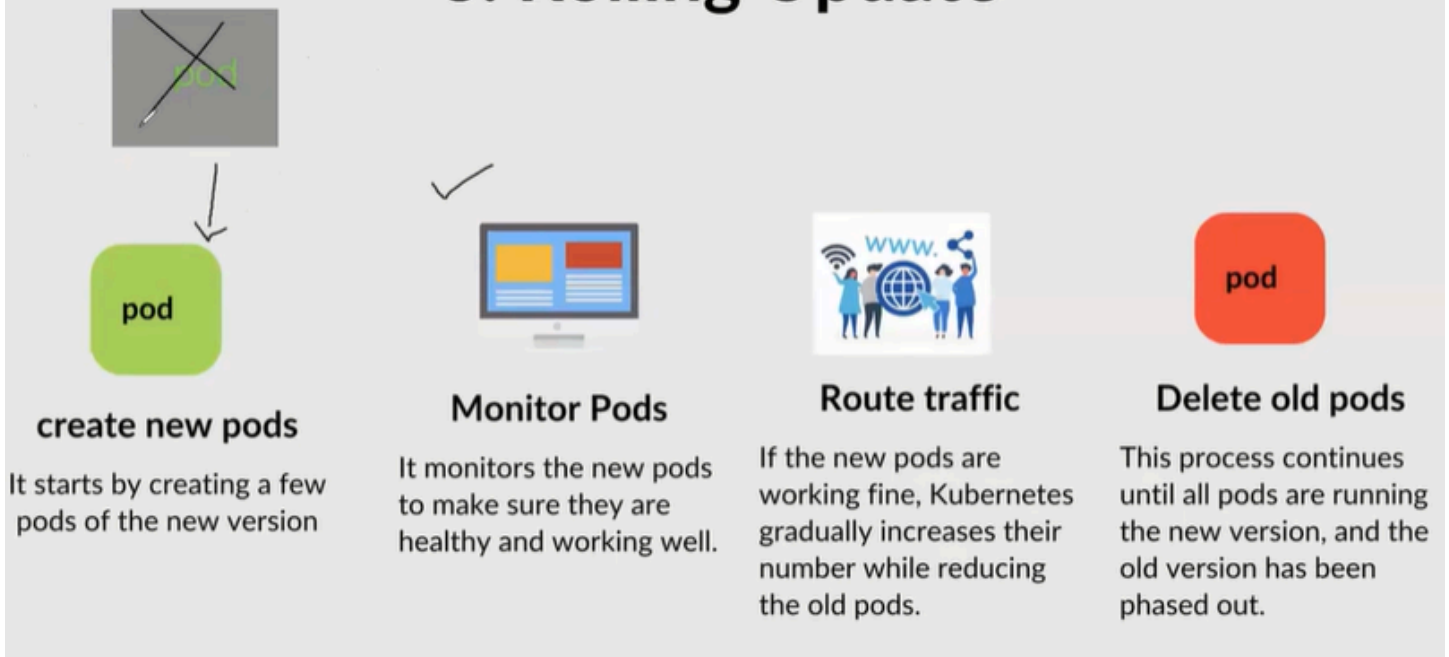
Ex: First 25%, next 25% and then last 50%.

2) Recreate Deployment: If the existing version of the application is completely stopped/terminated and a new version is deployed then it is called as a Recreate deployment strategy. The approach is simple but there's a downtime during the update.

The downtime is during the deployment of a new version to come live after old one is being terminated.

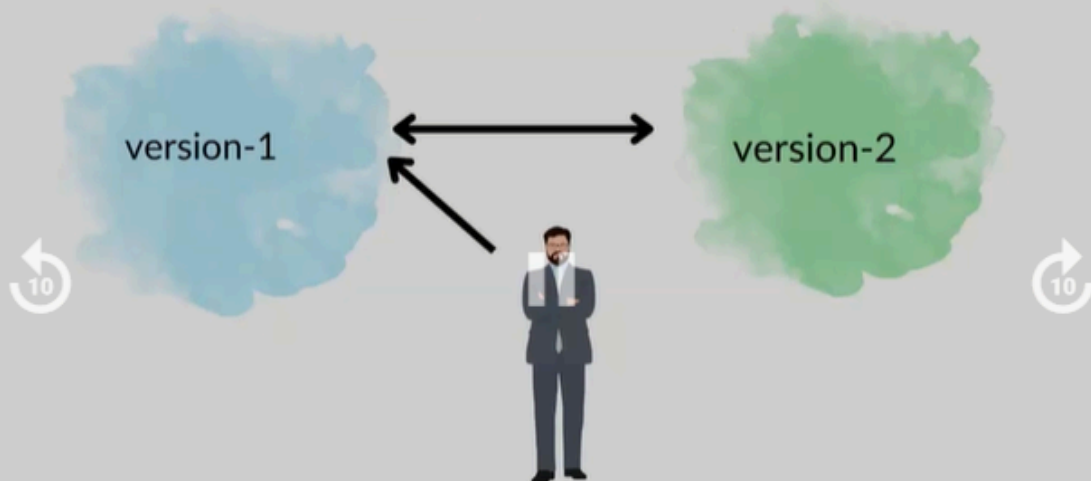
3) Rolling update Deployment: It's a strategy of creating a new pod whenever a old pod is deleted, checking it's health condition, routing the incoming traffic to the new pod and deleting the old pod automatically is called as rolling update deployment strategy.

3. Rolling Update



4) Blue-Green Deployment: It's a strategy of accomplishing a zero-downtime upgrade to an existing application. The "Blue" version is referred to as a currently running copy of an existing application and "Green" version is the new version. Once the green version is ready, the traffic is rerouted to the new version.

4. BLUE-GREEN DEPLOYMENT



A Blue/Green deployment is a way of accomplishing a zero-downtime upgrade to an existing application. The "Blue" version is the currently running copy of the application and the "Green" version is the new version. Once the green version is ready, traffic is rerouted to the new version

Kubernetes namespaces:

A namespace in Kubernetes is used to isolate the environments such as dev, test, prod and QA so that the pods that are created can also be differentiated and it makes the job easier to separate them based on environments.

About Namespaces:

1. Namespaces are used to group the components like pods, services, and deployments.
2. It's used to isolate the different environments such as dev, test, prod and QA to separate the projects or applications related to them.
3. You can have multiple namespaces in Kubernetes cluster and they're logically isolated from one another but they can be connected through configurations.
4. Pods inside the namespace can by default communicate to one another but not with the pods in different namespaces. However, can be done by configuration.
5. One service in a namespace can talk to another service in namespace if the target and sources are used with full name which include services/object name followed by Namespace.
6. In real-time the frontend pods are mapped to one namespace and backend pods are mapped to another namespace.
7. The namespaces are only hidden from one another but aren't fully isolated from each other.
8. The namespace represents as cluster inside the cluster.
9. The names of the resources within one namespace must be unique.
10. If a namespace is deleted all the resources would also be deleted.

Different types of default namespaces in Kubernetes cluster:

The different types of default namespaces are default, kube-public kube-system and kube-node-lease.

1. **Default** : It's the namespace where all the resources like pods, service, deployments we create gets stored which is called as default namespace.
2. **kube-public** : It's the namespace for resources that are publicly available for all.
3. **kube-system** : It's the namespace for resources created by the Kubernetes.
4. **kube-node-lease** : It's one of the default namespace in Kubernetes, where if a node gets down another node will be created and all the objects in that old node will be available in the new node. You can also define as, this namespace holds Lease objects associated with each node. Node leases allow the kubelet to send heartbeats so that the control plane can detect node failure.

How was the traffic handling with VM's, K8s and Ingress improved with ages?

i) Before the introduction of Kubernetes services (the traditional load balancing technique),

So during the early 2000's virtualization technology became widely adopted with solutions like VMware. The enterprises commonly used load balancing technique was traditional load balancing, achieved by deploying software load balancers like NGINX, HAProxy, or hardware-based load balancers like F5 on

physical or virtual servers. These load balancers would handle incoming traffic and distribute it to various backend servers or application instances to ensure high availability and scalability.

In 2006 Amazon launched EC2 (Elastic Compute Cloud), providing virtual servers on demand. Cloud based load balancing (like AWS ELB) was introduced in 2009.

The capabilities that enterprise load balancer provided were the following:

- i) Ratio based:** You can tell the LB to send 5 requests to pod1 and 2 requests to pod2.
- ii) Sticky session:** You can tell if one request goes to pod1 and all remaining also should go to pod1 only.
- iii) Path based:** Send the requests based on the rules configured to specific IP addresses.
- iv) Domain based:** If the request is from it should go to amazon.com and if India amazon.in.
- v) Whitelisting:** Only allow specific customers to access the application
- vi) Blacklisting:** You can block the users/the requests that are unwanted.

ii) After the introduction of Kubernetes services/before the introduction of ingress,

Kubernetes Service-Based Load Balancing type was built to manage traffic within the cluster and integrate with existing *cloud load balancers*.

Why do we need cloud based load balancers?

We use to create deployment, deployment used to create a pod and we use service to expose the the application using cluster IP, NodePort static port (exposes the service on each node's IP at a static port) and Load balancing.

If we expose our service using Kubernetes load balancing type based service we can access the application 'within the cluster and outside of the cluster' as well.

To access within the cluster you can use "*cluster -IP*" and to access the application outside of the cluster we need to expose the application using a "*cloud-based load balancer*" which provides a static IP and DNS name, making your application accessible globally, and is integrated with other cloud services for better scalability, security, and management.

Scalability: The load balancer automatically scales up or scales down it's capacity based on the incoming traffic.

Security: It provides SSL/TLS termination, web application firewall (WAF) that means to provide protection against common web exploits and attacks and integrate with AWS security groups that allows you to control inbound and outbound traffic to your instances.

Management: It takes care of monitoring and logging with AWS CloudWatch and AWS CloudTrail for logging API calls.

The service provides round robin load balancing, like if there are 2 pods and if there are 10 requests that are incoming, it used to send 5 requests to Pod1 and another 5 to Pod2. It was a simple load balancing

provided by the services in Kubernetes.

However, there was an issue, if there are 100's and 1000's of microservices for an application are deployed in a cluster, and to expose them using a Kubernetes Load balancing type we need a load balancer from cloud provider for each individual microservice and it was expensive because they provide static IP addresses and DNS names as well. To overcome this, kubernetes has introduced Kubernetes Ingress.

iii) After the introduction of Ingress,

Kubernetes introduced Ingress API object in the year 2015. It not only solved the problem of cost but also the complex management. Instead of using clod load balancers for each microservice we can now use a single cloud load balancer for an entire application no matter how many microservices you've for an application.

KUBERNETES INGRESS

TECH DEF: Kubernetes Ingress is a resource that manages external access to multiple services within a cluster using a single external load balancer. The ingress controller implements the routing rules and direct the traffic to the appropriate services.

SIMPLE TERMS: Let's say there are hundreds of microservices are running on pods inside the Kubernetes cluster, ingress manages all of them by providing an external access using a single external load balancer for an entire application. With the help of ingress it reduces the multiple load balancers, it takes care of routing rules and lowers the costs.

Here is a simple example or pictorial demonstration or proof of concept of how an Ingress sends all its traffic to one Service:



Shirt and Crisp explanation on how Ingress works in Kubernetes:

Client: Makes a request.

Ingress-Managed Load Balancer: Receives the request and forwards it to the Ingress.

Ingress: Uses routing rules to determine which Service should handle the request.

Service: Routes the request to the appropriate Pods within the cluster.

This setup allows multiple microservices to be accessed through a single entry point (the Ingress-managed load balancer).

DETAILED EXPLANATION:

We've to create a YAML file for ingress resource and choose a desired Ingress Controller as per the organizational requirement like Nginx, HAProxy, F5 etc.

So we configure the routing rules in the Ingress using different approaches, let's say if we chose path based routing, then */path* is mentioned as */bar* in the *YAML* file as per our example. When a user tries to access the application, the *ingress* resource inside the cluster routes the traffic to appropriate *service* (this service can be *deployment* or any other service that's been chosen) and service routes the traffic to the pods and the application or a single microservice that can be related to payments or catalogue can be accessed from the container.

The ingress resource is of no use if there's no ingress controller (load balancer).

For more detailed explanation on Ingress, please refer to the official documentation below:

<https://kubernetes.io/docs/concepts/services-networking/ingress/>

STEP BY STEP PROCEDURE ON HOW TO IMPLEMENT KUBERNETES INGRESS IN A THEROTICAL WAY:

In order to implement the Kubernetes Ingress and route the traffic to the pods, all we need is that, first we need to write a YAML file for Kubernetes Ingress with all the required information (typically change the Ingress name, service name) and install or deploy the Ingress controller (Nginx, F5 or HAProxy etc) on the Kubernetes cluster. After the Ingress controller is setup you'll get an IP address and in order to access the application you need need to add the host, IP address in */etc/hosts*.

STEP BY STEP PROCEDURE OF THE IMPLEMENTATION OF KUBERNETES INGRESS ON MINIKUBE CLUSTER:

1) First setup minikube cluster (for demo, in production we use EKS or AKS).

<https://minikube.sigs.k8s.io/docs/start/?arch=%2Fwindows%2Fx86-64%2Fstable%2F.exe+download>

2) Create a service to expose the application from cluster.


<https://kubernetes.io/docs/concepts/workloads/controllers/deployment/>

3) Create a Kubernetes Ingress resource.

<https://kubernetes.io/docs/concepts/services-networking/ingress/>

To execute the below manifest file on your machine, use **kubectl create -f ingress.yml**, and to check if the ingress is created or not check use **kubectl get ingress**.

You can try it out with path/host/domain based routing based on the requirement or for demo purpose.

service/networking/ingress-wildcard-host.yaml 

```
apiVersion: networking.k8s.io/v1
kind: Ingress
metadata:
  name: ingress-wildcard-host
spec:
  rules:
  - host: "foo.bar.com"
    http:
      paths:
      - pathType: Prefix
        path: "/bar"
        backend:
          service:
            name: service1
            port:
              number: 80
```

4) Install or deploy the Ingress controller in the Kubernetes cluster.

For all Ingress controllers: <https://kubernetes.io/docs/concepts/services-networking/ingress-controllers/>

For Nginx Ingress controller on Minikube: <https://kubernetes.io/docs/tasks/access-application-cluster/ingress-minikube/>

Enable the Ingress controller

1. To enable the NGINX Ingress controller, run the following command:

```
minikube addons enable ingress
```

2. Verify that the NGINX Ingress controller is running

```
kubectl get pods -n ingress-nginx
```

5) After installing the ingress controller on cluster now if you check the list of Ingresses you'll get to see the address for the host you selected in Ingress. **ADD A PICTURE DURING PRACTICE**

6) Since we haven't purchased the domain (domain doesn't exist i.e. foo.bar.com) so we can't access the app directly using the domain name or the IP address (**check during practice whether the app is not accessible with IP as well or not and modify this line later**).

You can update the host file (**/etc/hosts**), add the IP address and the host name so then you can access it, but this is not the real time practice that's implemented in the organization because the domain name requested by the client will be purchased. **ADD A PICTURE OF RESULT.**

PACKAGE MANAGEMENT IN KUBERNETES

A package manager or a package management system is used to keep track of what software installed, install a new software, upgrade to a new version or remove software that previously installed, all these activities are carried out by a package manager on your machine/computer. In general, for different operating systems we use a package managers like **apt** for ubuntu, **yum** for amazon linux/debian/CentOS, **yum** and **rpm** for RedHat, and **Helm** for kubernetes to install mandatory packages.

How did Helm come into existence?

As the Kubernetes platform and ecosystem continued to expand, deploying one and only one Kubernetes configuration file (i.e. a single YAML) was not standard industry practice anymore. It often happened that there would be multiple clusters to deploy and multiple resources to orchestrate within a Kubernetes cluster. So, as the number of YAMLs increased, the complexity of storing these YAMLs also increased.

As the complexity started to increase in Kubernetes, Helm was introduced at the very first KubeCon in 2015.

What is Helm in Kubernetes?

Helm is a package management system or a package manager in Kubernetes that allows you to define, install, and manage complex applications on Kubernetes clusters. It simplifies the deployment and management of applications on Kubernetes, how is that done is we can just maintain one single file and can use it for different environments by just changing the *values* in the ***values.yaml*** file.

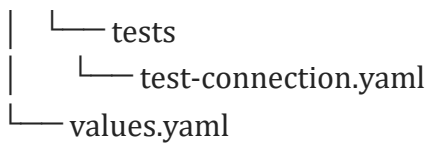
Components of a Helm Chart:

The important components in Helm chart are charts, repository and release.

i) Chart: Helm *charts* are packages of Kubernetes YAM manifest files that define a set of Kubernetes resources that are needed to deploy and run an application or a service on a Kubernetes cluster. is a collection of files that describe related Kubernetes resources such as deployment, service or any other required resources. A helm chart follows a particular directory tree structure as shown below.

Ex: You can use the command “**helm create nginx-demo**” to generate the chart. This will create a chart named nginx-demo with the default files and folders.

```
nginx/
├── charts
├── Chart.yaml
├── templates
│   ├── deployment.yaml
│   ├── _helpers.tpl
│   ├── hpa.yaml
│   ├── ingress.yaml
│   ├── NOTES.txt
│   ├── serviceaccount.yaml
│   └── service.yaml
```



The `nginx/` directory is the root directory of the Helm chart.

The `Chart.yaml` file describes the chart, including its version, name, and other metadata.

The `templates/` directory contains the Kubernetes YAML manifest templates that define the resources and configurations needed to deploy Nginx on a Kubernetes cluster. These templates are processed by Helm during the deployment process.

In the `values.yaml` file, we specify the values that are used to replace the template directives (deployment to service) in the templates.

ii) Repository: A helm *repository* is a storage environment where various charts of various application can be stored and shared across teams or anyone in the world.

iii) Release: It's an instance of helm chart running in a Kubernetes cluster using helm.

In simple words, an instance of helm chart is nothing but the collection of resources such as deployments, services, pods that are created.

KUBERNETES CONFIGMAPS AND SECRETS

The below use case explanation is used to make you understand the similarity of environment variables and ConfigMap.

USE CASE: A user is trying to access the app and when they try to login to the app the database checks the credentials that were stored (stored during the creation of account) and gives the information to the user whether they're correct or wrong through the app. You get the responses as "wrong credentials, or please try again" from the application.

How do we get the response?

In general the backend of the application i.e. database has connection parameters such as database port, database user, database password, database connection type. The application shouldn't have these details as hard coded (fixed values) and when if these details get changed the user would get null or vague information. In hard coding, fixed values or data are written directly into the source code of a program, whereas in soft coding, data or configuration information is stored in a separate location, such as a configuration file or database, and read into the program at runtime.

The environment variables are the variables that store the app secrets and configuration data. The values of app secrets and configuration data is stored in the text files or secrets or third party secret managers or calling scripts. These environment variables are not hard coded, these are truly dynamic and can be changed based on the environment that your program is running in.

When a user is trying to login to your app with their credentials, environment variables are called during the runtime of your application, so these are retrieved by your running app when needed.

In the same way, in Kubernetes also has ConfigMap that stores the data in it and is used when needed.

SUMMARY: The environment variables or modules are called when needed and can make change of the values during the runtime of an application without changing the source code to see how the configuration changes affects the application. In the same way, Kubernetes ConfigMaps is used/called when needed in the Kubernetes and can be decoupled and can be used in different environment.

What is ConfigMap in Kubernetes?

ConfigMap in Kubernetes is an API object which is used to store non-confidential data in key-value pairs within Kubernetes and this data can be used at later point of time by your application or pod or deployment when needed.

About ConfigMap:

- i) The ConfigMap can only store 1MB of data and not more than that and in order to store large amount of data we've to mount a volume or database or a file service.
- ii) ConfigMap doesn't provide any security and encryption. Therefore we can't store any sensitive data (like ssh keys, tokens, passwords) and in order to provide security and encryption we use Kubernetes Secrets.
- iii) ConfigMap allows you to decouple environment-specific configuration from your container images (like the application configurations would be different for development, testing, production), so that your applications are easily portable.
- iv) Here you can maintain the ConfigMap (YAML file) for each environment and instead of updating or modifying the data always inside the pod we can just make changes in the ConfigMap and it replicates the changes directly inside the pod. So you can decouple the Pod with the related ConfigMap and then it can be used in another environment as per the requirement. So this flexibility is provided by the ConfigMap.

IMPORTANT POINTS ABOUT CONFIGMAPS:

- i) First create a ConfigMap either by command or through a YAML file.
- ii) After creating a ConfigMap we need to insert the data in it and later the data can be injected to the pod.

ii) After injecting the data into the pod, if there is any update in any configuration or if the requirement changes then the ConfigMap can be updated and the data will be changed in Pod as well.

Different ways to create a ConfigMap

Imperative way:

1) Creating a ConfigMap from Literal:

Here we're creating a ConfigMap through `--from-literal` which means creating by passing key values through the command instead of providing key value pair data.

```
kubectcl create cm firstcm --from-literal=name=mahi --from-literal=stream=DevOps --from-literal=domain=engineer DATA = 3
```

```
kubectcl create cm firstcm --from-literal=name=mahi,stream=DevOps,domain=engineer DATA = 1
```

cm = ConfigMap

firstcm = ConfigMap name

--from = We're creating it from literal (providing key value instead of a file with key value pair data)

name, stream, domain = Values

To get a list of ConfigMaps: `kubectcl cm`

To describe a ConfigMap: `kubectcl describe ConfigMap ConfigMap name / kubectcl describe cm cm-name`

To delete a ConfigMap: `kubectcl delete ConfigMap ConfigMap name / kubectcl delete cm cm-name`

2) Creating a ConfigMap using a file:

i) First create a file using any name (new-file)

ii) Insert some data.

iii) Execute the file using "**kubectcl create cm cm-name --from-file=new-file**"

Now a ConfigMap is created and you can check if it's created or not using "*kubectcl get cm*" and can view the data using "*kubectcl describe cm cm-name*".

3) Creating a ConfigMap using a env extension file:

i) First create file with extension env (test.env).

ii) Add some data.

iii) Execute the file using **"kubectl create cm cm-name --from-env=test.env"**

4) Creation of ConfigMap using a directory:

If there are multiple files that has the data, if you want to create a ConfigMap then each file needs a separate ConfigMap. Instead, we can create a directory and create all files in the directory. Then if you create a ConfigMap using the directory, the data from all files would be added.

i) Create a directory.

ii) Create required number of files inside the directory and data.

iii) Now, create a ConfigMap **"kubectl create cm cm-name --from-file=directory name"**.

Why we've chose *--from-file* instead of *--from-folder* is because folder is also file like in Ansible.

Declarative way:

5) Creation of ConfigMap using declarative way using Manifest file:

i) Create a file with yaml extension and write the Manifest file.

apiVersion: v1

kind: ConfigMap

metadata:

name: Demo

data:

DATABASE_PORT: "3306"

DATABASE_URL: "https://example.com"

ii) Execute the file using *"kubectl create -f filename.yml"*.

iii) Check if the ConfigMap is created.

What kind of information can you inject into a Pod?

You can inject configuration files, environment variables, or secrets (like API keys or database credentials) into a Pod using ConfigMaps or Secrets. These changes are reflected in the application by either updating the Pod's environment variables or by mounting the data as files, which the application reads at runtime. This way, the application adapts to new settings without altering its source code.

RUNTIME: For example, if a ConfigMap is mounted as a file in a Pod, the application will open and read that file when it needs the configuration, allowing it to use the latest settings without needing to restart or recompile.

How to inject data into a Pod:

- 1) Create a ConfigMap either using literal, file or any way that you wish and verify if that is created.
- 2) Now create a Manifest file to create a Pod and add the env with it's key and value.

apiVersion: v1

kind: Pod

metadata:

-name: pod-1

spec:

containers:

- name: cont-1

image: nginx

env:

- name: onevalue

valueFrom:

ConfigMapKeyRef:

key: name

name: test

- name: two value

valueFrom:

ConfigMapKeyRef:

key: Stream

name: test

3) Go into the pod and check if the data is injected to the Pod or not.

kubect exec -it pod-name /bin/bash & env | grep test

NOTE: If you want to inject specific key values to the pod then you need to use '*configMapKeyRef*' and if you're directly injecting all the values in configMap then use '*configMapRef*'.

VOLUME MOUNTS

Why Volume Mounts are needed?

The size of a Configmap is 1MB, so to increase the size to store large amounts of data we need Kubernetes Mounts.

Without altering the application code and if you want to inject the data we can use the environment variables with the key values or mounting the data by files to make the desired changes with the help of ConfigMaps by mounting the Volumes to the containers without restarting any containers.

They provide persistent storage for applications, ensuring data survives even if a Pod restarts or moves to a different node.

Use Case: A database running in a Pod needs to store its data persistently. By using a PersistentVolume mounted to the Pod, the database's data is saved even if the Pod is rescheduled, ensuring no data loss.

How do you inject a more than 1MB file to a Pod?

1) First create a file, add some data and then create a ConfigMap using the created file.

2) Now create a Manifest file to create a Pod to inject the data into it, as the data is greater than 1MB in size so we need to use Volumes.

3) Create a Volume and create a folder inside the container to mount the volume to a container (`"/folder-name"`). You've different access modes like read only, write only and read & write only mode.

Below is the script for the Manifest file:

apiVersion: v1

kind: Pod

metadata:


```
name: Pod-1

spec:

  containers:

    - name: cont-1

      image: httpd

      volumeMounts:

        - name: my-volume

          mountPath: "/test-volume"

          readOnly: true

  volume:

    - name: my-volume

      configMapRef:

        - name: demo
```

A detailed explanation of volume and volumeMounts section from the above manifest file.

Volume:

-name: my-volume is the name of the volume.

name: demo is the name of the ConfigMap.

volumeMounts:

-name: my-volume is the volume name.

mountPath: "/test-volume is the folder inside the container in which the my-volume is stored.

readOnly: true means we can only read the data inside the volume.

CONCLUSION: So, the data stored in ConfigMap is stored into 'my-volume' and the volume is mounted within the container (which is in default path) inside a new folder.

Why secrets in Kubernetes?

It is used to store the confidential data and it provides security in the form of encryption of the data and similar to ConfigMaps the data can be used at later point in time by the application when needed. Hence, there's no risk of security in Kubernetes secrets.

Let's say a user created a YAML file for creating the ConfigMap, it is created on K8s cluster and at the same time the API server is saving the information in etcd as well. The data is not secure as the data can be viewed and in order to solve this we use secrets where we've RBAC at the etcd where in order to view or hack the data at the rest they need a decryption key.

Example: Let's consider an example that, we've front-end, backend and database for an application. If we need to change the database credentials, we can't directly change it in the code as they're hard coded values and it's sensitive information. If it's still done, it'll alter the application code which would disturb the running pods, that causes downtime and affect the incoming traffic.

Hence, we use Kubernetes Secrets specifically to store the sensitive data and manage these credentials, which allows you to change the hard coded values without altering the application code.

COMMANDS:

1) To create a secret: `kubectl create secret generic secret-name --from-literal=keyname=value --from-literal=keyname=value` **generic** → is used to encrypt the data.

`kubectl create secret generic test-sec --from-literal=username=mh99--from-literal=password=rj@123`

2) To check list of secrets: `kubectl get secret`

3) To view details of a secret: `kubectl describe secret secret-name`

4) To create a secret using a file: `kubectl create secret generic secret-name --from-file=filename`

Opaque you see when you describe the file is called the encryption technique to encrypt the data.

5) Create a secret using the env file to encrypt: `kubectl create secrett sec-name --from-env-file=filename.`

6) To view the data in yaml file: `kubectl get secret -o yaml.`

How to inject secrets to a Pod:

1) First create a secret using literal or file or env.

2) Next create a file with yml extension and write a Manifest file to create a pod, create an environment variable name, mention the key value of the secret to attach to the created environment variable.

If you want to mention all the key names and it's values, use *secretRef* and if not use *secretKeyRef*.

3) Execute the yaml file to create a Pod and then go inside the pod to check if the data is decrypted or not.

The reason why the data is encrypted outside because to ensure it's security during the storage and transmission and decrypted inside the pod to keep it secured and protected while being used, preventing unauthorized access from outside.

How to mount a Volume to a Pod:

Storing the secret (>1MB) to a volume and attaching the vol to the container inside a Pod.

1) Create a secret using literal or file or env.

2) Next create a file with yml extension, write the Manifest file with details to create a Pod, create a volume, storing the secret data to the volume and mount the volume within the container inside a new folder.

apiVersion: v1

kind: Pod

metadata:

name: pod-1

spec:

containers:

- name: cont-1

image: httpd

volumeMounts:

- name: my-volume

mountPath: "/new-vol"

volumes:

- name: my-volume

secret:

secretName: demo-sec

How do you rename the key name inside a Pod:

If you want to change the name inside a Pod for the key name you created in local, you can do that by adding the below lines of code in the Manifest file.

items:

- *key:* key-name in local

path: the name you want to create inside the Pod/rename the key name inside the pod (This will create a file with a new name and the value inside the key [data] can be accessed).

Below is the script for the Manifest File:

apiVersion: v1

kind: Pod

metadata:

name: pod-1

spec:

containers:

- name: cont-1

image: httpd

volumeMounts:

- name: my-volume

mountPath: "/new-vol"

volumes:

- name: my-volume

secret:

secretName: demo-sec

items:

- *key:* key-name

path: new-name-1

- *key*: key-name

path: new-name-2

KUBERNETES VOLUMES

A pod is a short living object (ephemeral). If a Pod is deleted due to any heavy load or heavy traffic or under any unforeseen circumstances we lose the data. In order to overcome this drawback of data loss we use Kubernetes Volumes. This will help us out in storing the data in the volume even the pod is deleted or moved to another node inside the cluster.

KEY FACTS:

- 1) We attach a volume to a Pod, that'll replicate the data for all the containers inside the Pod.
- 2) A volume resides inside the pod and stores the data of all containers in that Pod.
- 3) If the pod is deleted inside the pod, a new pod gets automatically created and the data from volume is shared to the new created container.
- 4) If the pod is deleted, then the volume also gets deleted which leads to a loss of data for all containers permanently. After the deletion of Pod, a new pod is created (due to replication) with volume but this time volumes don't have the previous data or any data.

The data is lost when a Pod is deleted and also it's not available when a new pod is created, so we got Kubernetes Volumes.

Why Kubernetes Volumes is needed?

If a Pod is deleted, a new pod is created along with volume because of replication. However, the newly created pod doesn't have the previous data or any data, this drawback of unavailability of old data or any data or persistent storage, Kubernetes Volumes comes to the rescue.

Types of volumes:

- 1) Empty Dir
- 2) HostPath
- 3) Persistent Volume (PV)
- 4) Persistent Volume Claim (PVC)

1) Empty Dir:

- This volume is created when the pod is created and it exists as long as the pod.
- There won't be any data available when EmptyDir is created for the first time.
- This volume is used to share the volumes between multiple containers within a pod instead of host machine or any master/worker node.
- Containers within the pod can access the other container's data. However, the mount path can be different for each container.
- If the pod gets crashed then, a new pod gets created, the data in the volume is lost.

Advantage: The data replication is possible, if we create some files in cont-1, it replicates in cont-2 and viceversa.

Disadvantage: If the pod is deleted, the volume also gets deleted. If we've replication, a new pod gets created but there won't be any data. Hence, we chose HostPath over EmptyDir.

2) HostPath:

- This volume type is the advanced version of EmptyDir.
- In EmptyDir the data is stored in volumes, where host machine doesn't have the data of pods or containers.
- In HostPath volume type, it helps to access the data of pods or container volumes from the host machine.
- HostPath replicates the data of volumes on the host machine. Like if there are any changes made from local host machine then the changes can be reflected in pods volumes and vice versa.

3) Persistent Volume:

- Persistent means always available.
- Persistent Volume is an advanced version of EmptyDir and HostPath.
- It doesn't store the data in local server, instead it stores in the cloud or some other place where the data is highly available.
- In previous volume types, if the pod gets deleted the data is lost. However, using this persistent volume the data is not lost and it is shared to other pods or other node's pods.
- PV's are independent of the pod life cycle, even if no pods are using PV's they do still exist.
- With the help of Persistent Volume the data is stored on central location such as EBS, Azure Disks etc.
- One persistent volume is distributed to the entire Kubernetes Cluster and any node or node's pod can access the data from the Persistent Volume accordingly.
- In K8s, PV is a piece of storage in the cluster and is provisioned by the administrator. If a Pod requests a storage via PVC, K8s will search for suitable PV to satisfy the request and if you want to use PV you have to claim that volume with the help of Manifest YAML file. If the correct PV is found then PV is bound to the PVC and it can be used by the Pod and if not the PVC will remain unbound (pending).

4) Persistent Volume Claim:

- PVC is request for storage by user. It is similar to a Pod, pod consumes the node resources and PVC's consume the PV resources. Pods can request specific level of resources (CPU, memory). Claims can request specific size and access modes (e.g., they can be mounted ReadWriteOnce, ReadOnlyMany, ReadWriteMany, or ReadWriteOncePod)
- To get the Persistent Volume you've to claim the volume with the help of PVC.
- When you create a PVC K8s will find the suitable PV to bind them together. After a successful bound to the pod you can mount it as a volume.
- If the pod is deleted, the PV is released and after a new pod is created then the PV is attached automatically again to the newly created Pod.
- Once a user finishes it's work, then the attached volume gets released and can be used for recycling such as new pod creation for future usage.

Overview of all types of volumes:

EmptyDir: This volume is created when a pod is created and data is lost from volume when pod is deleted.

HostPath: In this volume you can mount a directory or file from host machine and if a pod is deleted the data is not lost as you've the data in the local machine.

Persistent Volume (PV): PV is a piece of storage in Cluster which is provisioned by the K8s administrator or dynamically provisioned using storage classes.

Persistent Volume Claim (PVC): PVC is request for storage by user, to get PV you've to claim using PVC and once the suitable PV is found then K8s will bound the PV with the Pod and you can mount it as a volume.

Implementation of no data loss in the Pod using PVC in Kubernetes:

- 1) First create an EC2 instance and setup a cluster (preferably minikube for implementation).
- 2) Create an EBS volume using the aws console.
- 3) Create a manifest file to create a PV.
- 4) Create a manifest file for PVC.
- 5) Create a manifest file for deployment and mount the PV using the PVC in the same file.
- 6) Execute the manifest files for PV, PVC and deployment.

Now if you delete a pod, a new one gets created with the help of replicas mentioned in the deployment and after the new one is created, you get to see the same data in it. This is because of the PV concept of Kubernetes and the EBS volume storage that we used to mount it to the Pod.

Advantage or usage of PVC volume: If a pod is deleted, with the help of replicas using any service such as deployment a new pod gets created and using EBS volume, the data is not lost and even if the instance is deleted, with the help of ASG a new instance gets created and with the help of K8s architecture a new pod gets created and the data is retrieved from EBS with the help of PV that is claimed using PVC.

Below are the manifest files for PV, PVC and deployment:

1) Persistent Volume:

apiVersion: v1

kind: PersistentVolume

metadata:

name: my-pv

spec:

capacity:

storage: 5Gi

accessModes:

- ReadWriteOnce

persistentVolumeReclaimPolicy: Recycle

awsElasticBlockStore:

volumeId: " mention vol -id here"

fsType: ext4

2) Persistent Volume Claim:

Let's say if we've taken an EBS volume of 10Gi and from that storage we've created a PV of 5Gi. Whenever a pod requires a specific amount of storage then PVC claims the desired amount of storage from PV.

apiVersion: v1

kind: PersistentVolumeClaim

metadata:

name: my-pvc

spec:

accessModes:

- ReadWriteOnce

resource:

requests:

storage: 2Gi

3) Deployment:

apiVersion: v1

kind: deployment

metadata:

name: kalki

spec:

replicas: 2

selector:

matchLabels:

app: demo

template:

metadata:

labels: app:demo

spec:

containers:

- name: cont-1

image: nginx

command: ["/bin/bash", "-C", "sleep 10000"]

volumeMounts:

- name: test-vol

mountPath: "/opt/persistent"

volumes:

- name: test-vol

persistentVolumeClaim:

claimName: my-pvc