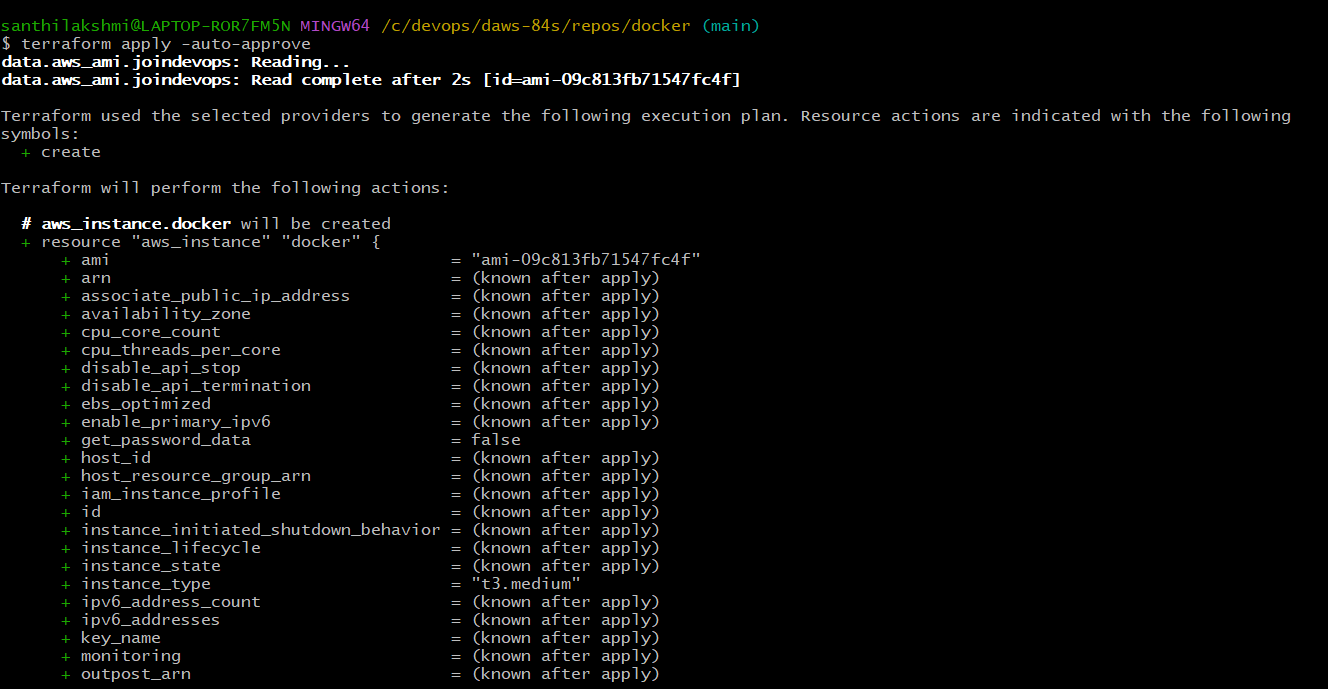
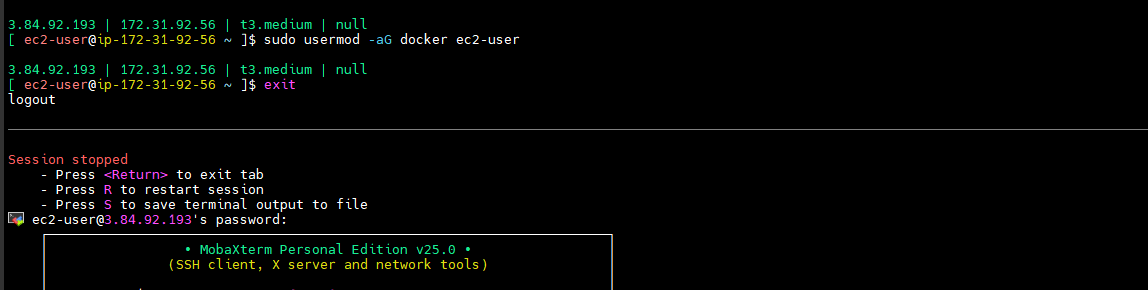
**SESSION - 62**

**--> cd /c/devops/daws-84s/repos/docker**

**--> terraform apply -auto-approve**



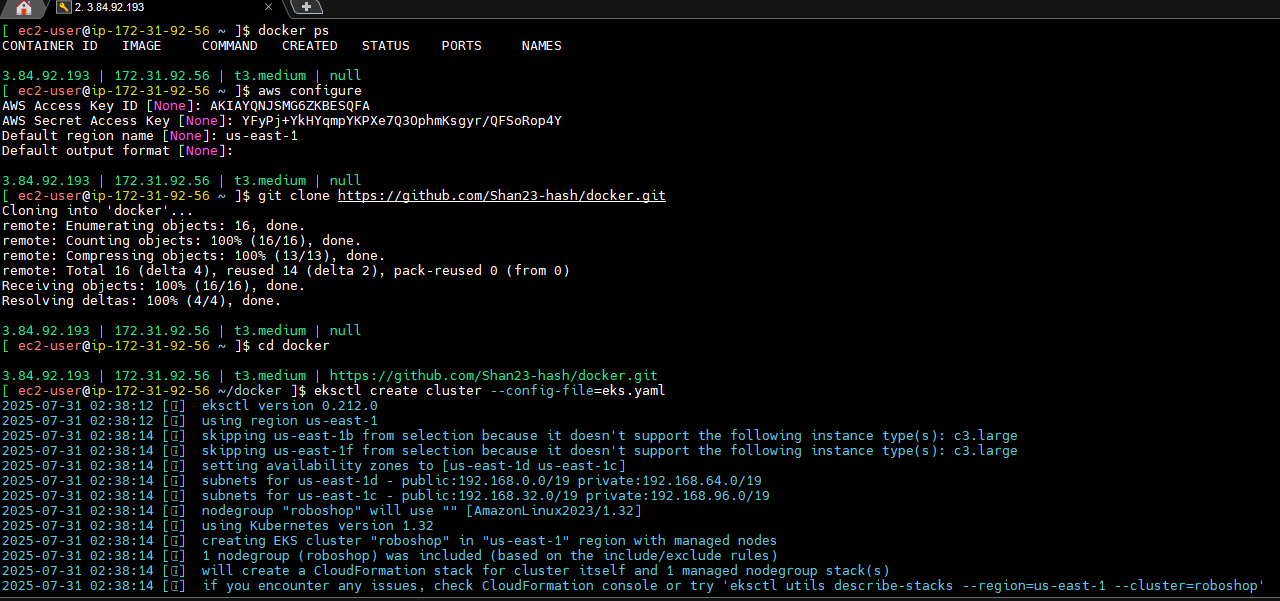
**--> sudo usermod -aG docker ec2-user**

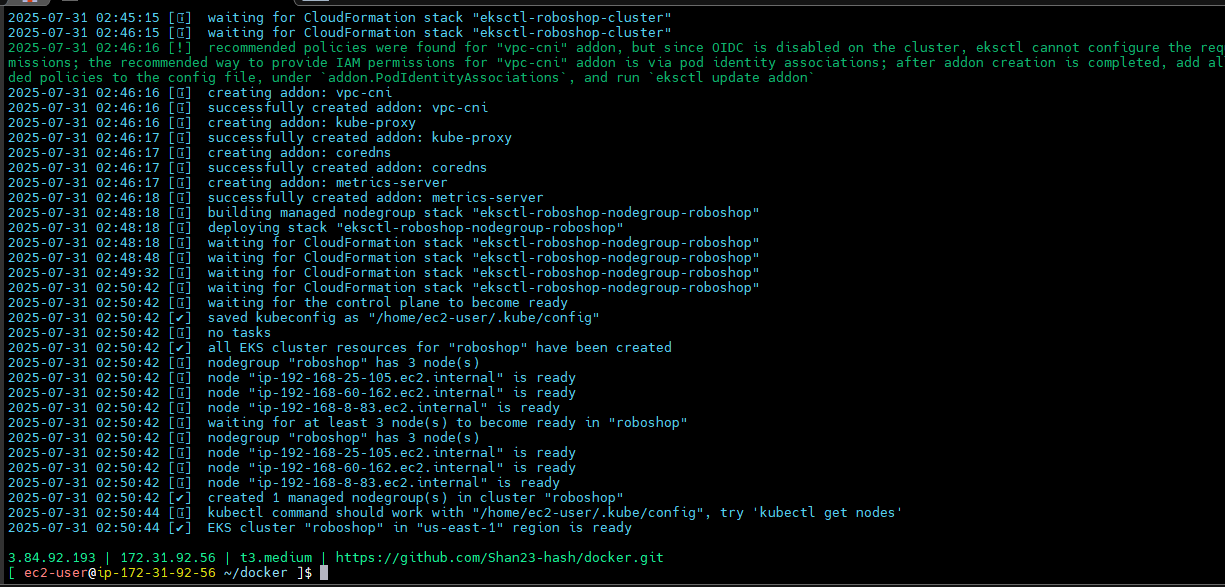


**--> aws configure**

**--> git clone <https://github.com/Shan23-hash/docker.git>**

**--> eksctl create cluster --config-file=eks.yaml**





**-->** Volumes is two times generally hard disk and cloud.

--> PVE means physical represention of the disk.

**On-premise**

**===================**

--> Disk, NFS

PV --> cluster

PVC --> namespace level

EKS admins control cluster level objects…

as a roboshop devops engineer, you need a disk to be created for your application,

we will raise a ticket for this. GB, filesystem type(ext4), etc..it is approved by roboshop team lead/delivery lead. Storage team also checks this and it is approved their team leader..

then storage team creates the disk…

provide these disk details to EKS admin, then they create PV for us and tell us the name.

--> They will create equivalent pv and they will give to us.

--> We will create pvc and pod

--> if you are the admin you can do PV creation that and all but disk creation for these and all will be there separate team.

**Disk Details Submission:**  
We first provide the **disk/storage requirements** (size, type, etc.) to the **EKS admin team**.

**Persistent Volume (PV) Creation:**  
The admin team will **create an equivalent PV** based on the disk details and share the **PV name** with us.

**Persistent Volume Claim (PVC) and Pod Setup:**  
Once the PV is ready, we will:

Create a **PVC** that matches the PV’s specifications.

Mount the PVC in the respective **pod definition**.

**Note (For Admins):**  
If you are part of the **EKS admin team**, you can handle PV creation.  
However, **actual disk creation** (e.g., EBS volume provisioning) is typically handled by a **separate infrastructure or cloud team**.

PV --> K8 resource, physical representation of the actual storage.

PVC --> it is the claim done by pods to mount the storage

SC --> k8 object used to create the volume dynamically…

· **PV (Persistent Volume):**  
A Kubernetes resource that represents a **physical storage unit** in the cluster. It is provisioned by an administrator or dynamically using a StorageClass and abstracts the details of how the storage is provided (e.g., NFS, AWS EBS, etc.).

· **PVC (Persistent Volume Claim):**  
A **request made by a pod** for storage. It specifies the size, access mode, and other requirements. PVCs allow pods to **mount and use PVs** without needing to know the details of the underlying storage.

**· SC (StorageClass):**  
A Kubernetes object that enables **dynamic provisioning** of Persistent Volumes. It defines the type of storage (e.g., SSD, HDD), reclaim policy, and the provisioner (e.g., AWS EBS, GCE PD) to be used for automatic volume creation when a PVC is made.

**EBS static provisioning**

**=======================**

1. Install EBS drivers

2. Provide access to EC2 instance through role, EBSCSIDriverPolicy

3. create volume in same az as in EC2 instance

4. create PV, create PVC, create pod with nodeSelector option

**Dynamic**

**=============**

--> We are created storage class

--> Whoever will create pv that peoples can create volume.

--> volume created Storage class and pvc these two will go create pv.

--> We mentioned the pod.

--> Wallet is storage physical,pepresented by pv

--> climing mother

--> clime will do pvc abnormally pod will take

Kid --> Mother --> Father --> Wallet

Pod --> PVC --> PV --> Storage

Kid --> Mother --> UPI

Pod --> PVC

--> EFS(elastic file sgaring) - efs is nothing but file sharing like nfs(network file sharing) your drive can be everywhere in the internet.

Let us install deivers

--> efs drivers eks -- check in chrome

**EFS STATIC PROVISIONING**

**=======================**

1. Install drivers

2. Give permissions to nodes, EFSCSIDriverPolicy

3. create EFS volume

4. Allow port no 2049 in EFS SG from EC2 SG

5. Create PV, PVC and mount to pod

**--> kubectl kustomize \**

**"github.com/kubernetes-sigs/aws-efs-csi-driver/deploy/kubernetes/overlays/stable/ecr/?ref=release-2.1" \**

**> private-ecr-driver.yaml**

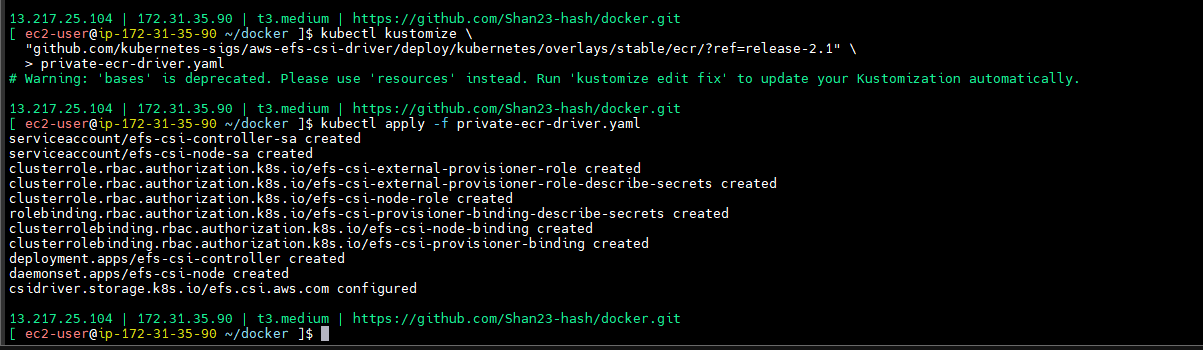
**--> cd**

**--> kubectl kustomize \**

**--> vim private-ecr-driver.yaml** -- all resources are there here.

**--> kubectl apply -f private-ecr-driver.yaml** -- all drivers was installed

--> We can gives permission for nodes.



--> IAM --> ROLES --> eksctl -> add permissions --> attach policies - AmazonEFSCSIDriverPolicy -- > Add permissions

--> Node was ready

--> Go to EFS inside

--> If you did one node it will update automatically remaining 2.

--> Elastic file system --> amazon EFS --> File system

**EBS vs EFS**

**===========**

1. EFS can be anywhere in network, EBS should be in same AZ as in EC2

2. EBS can be accessed only by one instance at a time, EFS can be accessed by multiple instances

3. EBS is faster than EFS.

4. EBS is mainly used for OS and DB, EFS is used to store objects and files.

5. EBS size is fixed, EFS is elastic it automatically grows upto 48TB.

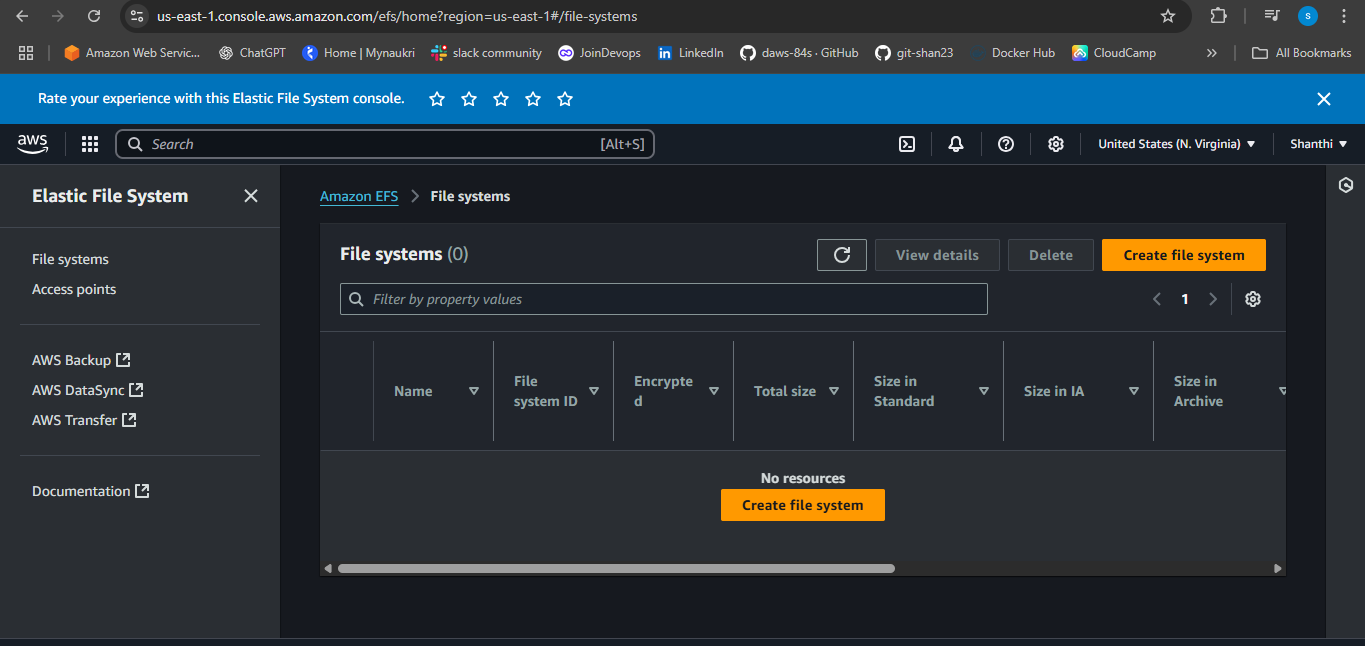
6. EBS filesystem we can select ext4, EFS filesystem is already decided by AWS as NFS

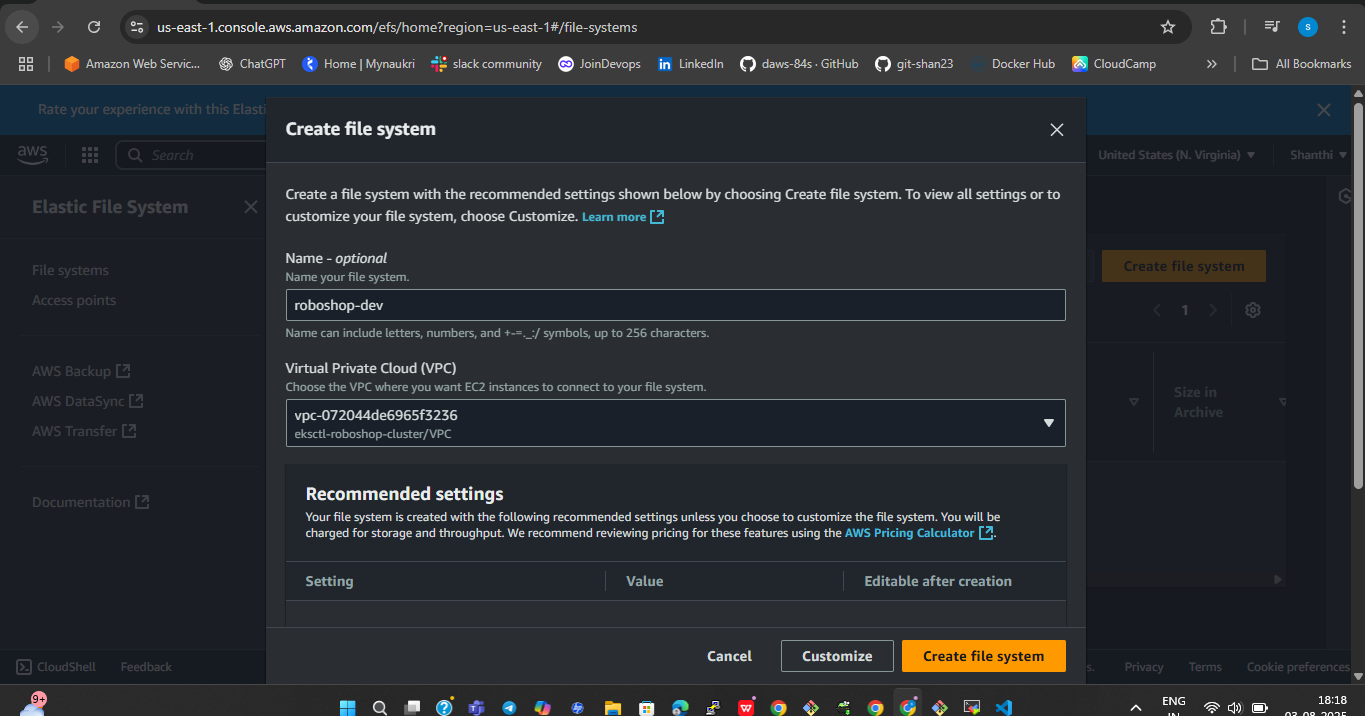
7. EBS will not have any SG attached, but EFS is in network, so there will be SG attached and 2049 should be allowed.

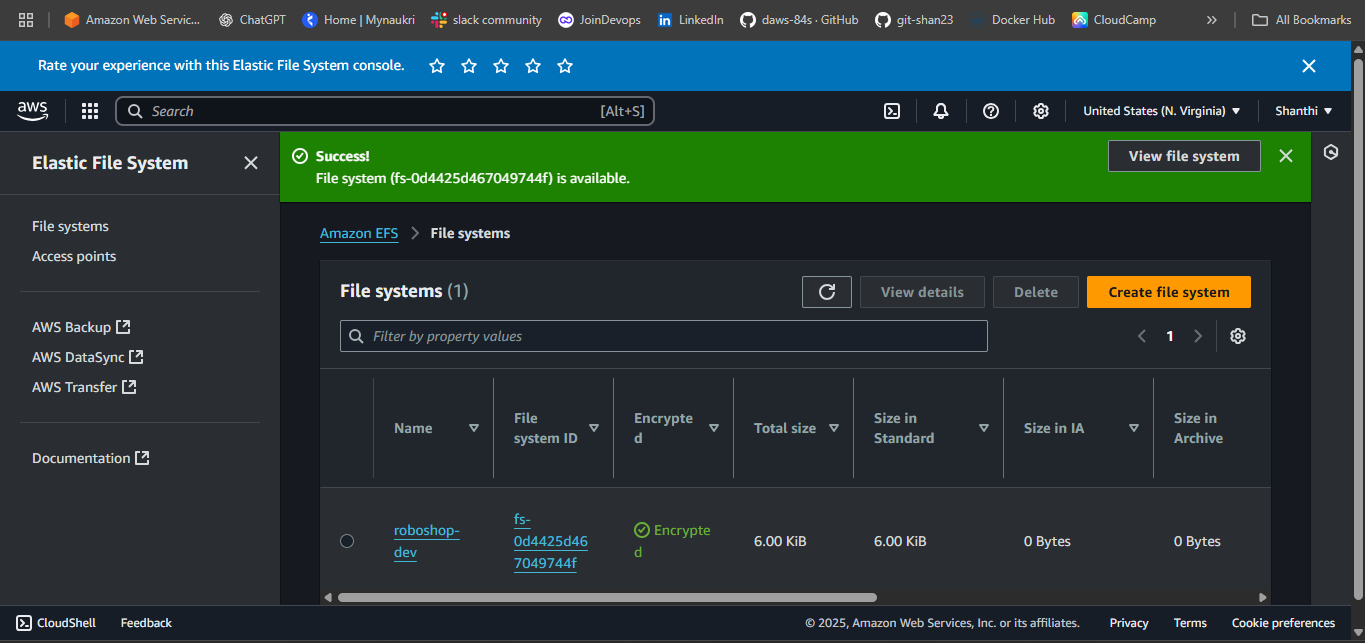
**--> Interview question**

|  |  |  |
| --- | --- | --- |
| **Feature** | **EBS** | **EFS** |
| Storage Type | Block | File |
| Access | Single instance | Multiple instances |
| Scalability | Limited by instance | Elastic and scalable |
| Performance | Low latency, high performance | Scalable with throughput and IOPS |
| Use Cases | Databases, OS, demanding applications | Shared files, web servers, collaborative workloads |
| Replication | Within AZ | Across AZs |

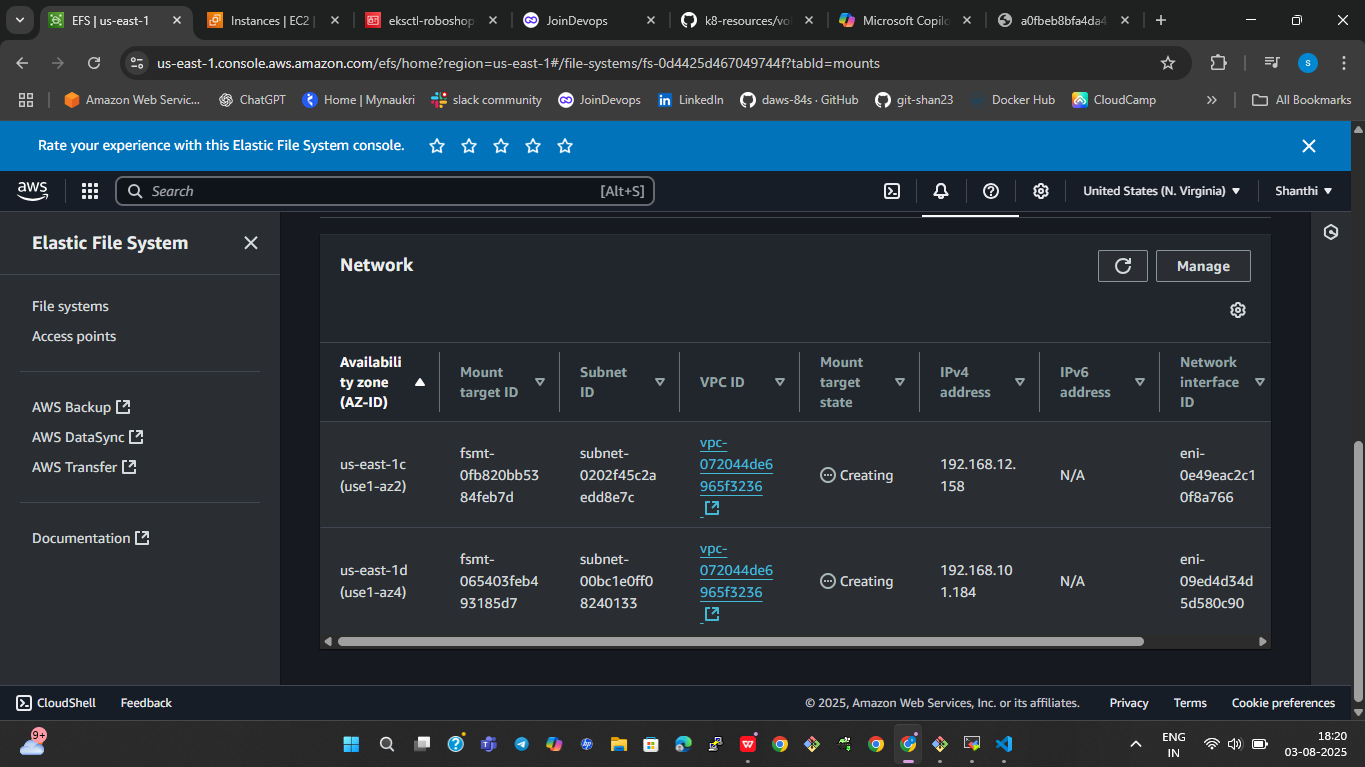
Create EFS --> Create file system ==> roboshop-dev taking --> dev server --> Create file system.







--> VPC there in 2 avilability zone.

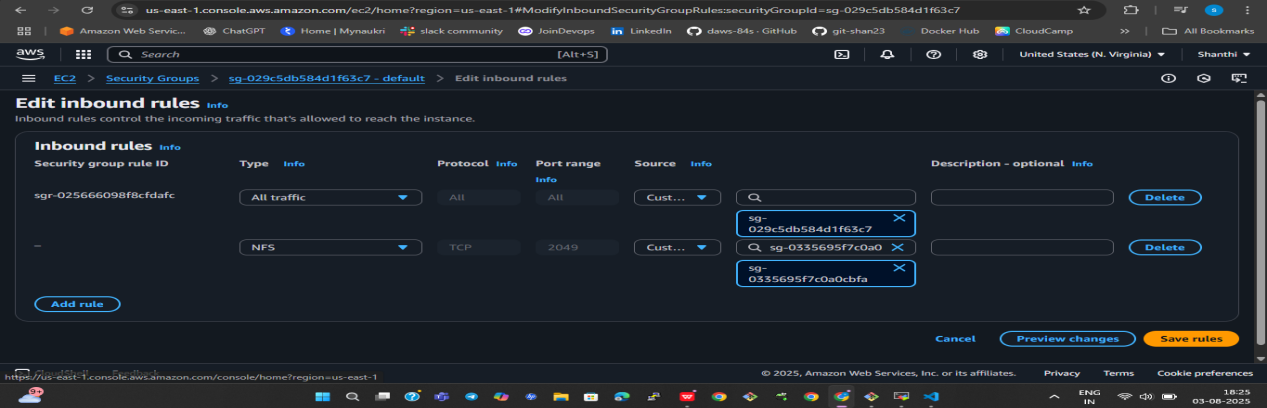


--> VPC created automztically EKS

--> For 1a,1c two poits was created. One secirity group also will attach

--> Security group is default security group

--> Go to security group --> edit inbounded rules --> Add rule --> NFS --> allow instance for secutiry group (take sg and give here) - save changes



**volumes/03-ebs-dynamic.yaml**

apiVersion: v1

kind: PersistentVolumeClaim

metadata:

name: ebs-dynamic

spec:

accessModes:

- ReadWriteOnce

storageClassName: roboshop-ebs

resources:

requests:

storage: 4Gi

---

apiVersion: v1

kind: Pod

metadata:

name: nginx

labels:

purpose: ebs-dynamic-demo

project: roboshop

spec:

nodeSelector:

topology.kubernetes.io/zone: us-east-1c

containers:

- name: nginx

image: nginx

volumeMounts: # docker run -v host-path:container-path

- name: persistent-storage

mountPath: /usr/share/nginx/html

volumes:

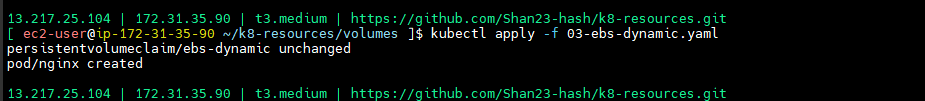
- name: persistent-storage

persistentVolumeClaim:

claimName: ebs-dynamic

--> push and pull the code

**--> kubectl apply -f 03-ebs-dynamic.yaml**



**volumes/04-efs-static.yaml**

apiVersion: v1

kind: PersistentVolume

metadata:

name: efs-static

spec:

capacity:

storage: 5Gi

volumeMode: Filesystem

accessModes:

- ReadWriteOnce

storageClassName: "" # This should be empty in static provisioning

persistentVolumeReclaimPolicy: Retain

csi:

driver: efs.csi.aws.com

volumeHandle: fs-0bce55c2a5841e18c (take from we are created one volume take that one)

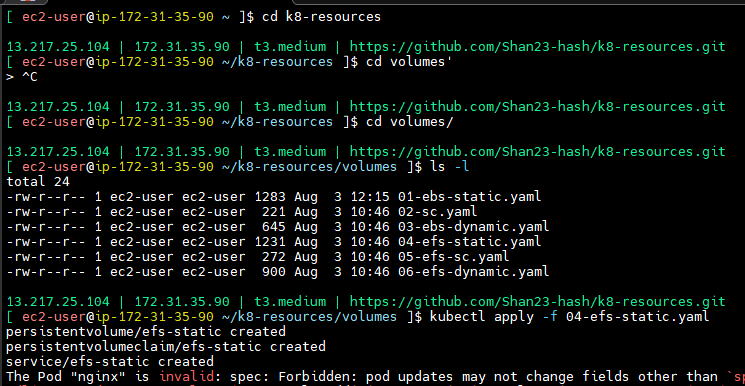
--> Push and pull the code

--> cd ../k8-resources/volumes

**--> kubectl apply -f 04-efs-static.yaml**

**--> kubectl get pv**

---> pv created next pvc



**volumes/04-efs-static.yaml**

apiVersion: v1

kind: PersistentVolume

metadata:

name: efs-static

spec:

capacity:

storage: 5Gi

volumeMode: Filesystem

accessModes:

- ReadWriteOnce

storageClassName: "" # This should be empty in static provisioning

persistentVolumeReclaimPolicy: Retain

csi:

driver: efs.csi.aws.com

volumeHandle: fs-0bce55c2a5841e18c (take from we are created one volume take that one)

---

apiVersion: v1

kind: PersistentVolumeClaim

metadata:

name: efs-static

spec:

accessModes:

- ReadWriteOnce

storageClassName: ""

resources:

requests:

storage: 5Gi

---

apiVersion: v1

kind: Pod

metadata:

name: nginx

labels:

purpose: efs-static-demo

project: roboshop

spec:

containers:

- name: nginx

image: nginx

volumeMounts: # docker run -v host-path:container-path

- name: persistent-storage

mountPath: /usr/share/nginx/html

volumes:

- name: persistent-storage

persistentVolumeClaim:

claimName: efs-static

---

apiVersion: v1

kind: Service

metadata:

name: efs-static

labels:

purpose: efs-static-demo

project: roboshop

spec:

type: LoadBalancer

selector:

purpose: efs-static-demo

project: roboshop

ports:

- protocol: TCP

port: 80 # service port

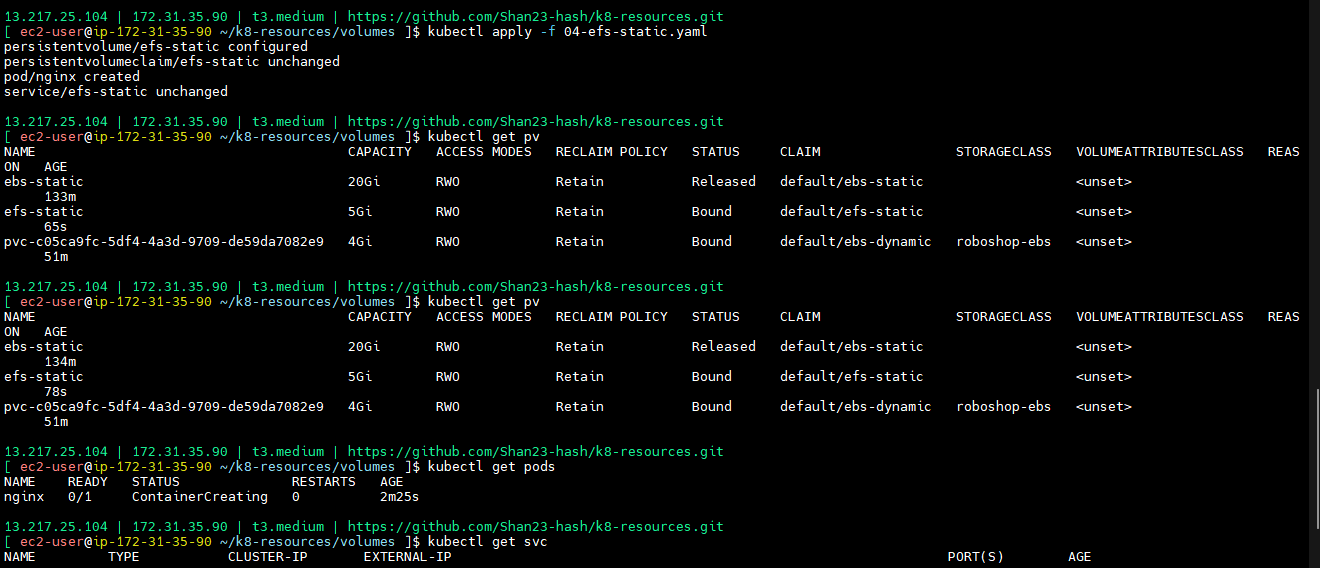
targetPort: 80 # container port

--> Push and pull the code

**--> kubectl apply -f 04-efs-static.yaml**

**--> kubectl get pods**

**--> kubectl get svc**



--> take load balancer externap ip check in browser

**--> kubectl exec -it nginx -- bash**

**--> cd /usr/share/nginx/html**

**--> echo “<h1>Hi, This is from EFS static</h1>” > index.html**

**--> kubectl delete pod nginx**

**--> kubectl apply -f 04-efs-static.yaml**

**--> kubectl get pods**

--> data will not go anywher safe in volume.

**Dynamic Provisioning**

--> Dynamic means not created file inthis we have access point that will create.

--> File system is elastic

--> if you want access point one on one you can take

--> access point will create dynimically

--> create storage class first

**volumes/05-efs-sc.yaml**

kind: StorageClass

apiVersion: storage.k8s.io/v1

metadata:

name: efs-roboshop

provisioner: efs.csi.aws.com

reclaimPolicy: Retain

parameters:

provisioningMode: efs-ap

fileSystemId: fs-0807041eac3ddddd8

directoryPerms: "700"

basePath: "/roboshop" # /roboshop/cart

--> Give file system id

--> bash path also need to give

--> remaining all default

--> push and pull the code

The Amazon EFS Container Storage Interface (CSI) driver for Amazon Elastic Kubernetes Service (EKS) supports two primary provisioning modes for persistent storage:

**Dynamic Provisioning:**

This mode automates the creation of Persistent Volumes (PVs) and Persistent Volume Claims (PVCs) within the EKS cluster.

When a PVC is created, the EFS CSI driver dynamically provisions an Amazon EFS access point for each PV.

This approach simplifies storage management, especially in dynamic application environments, as the driver handles the underlying EFS access point creation and mapping to the PV.

It requires a pre-existing Amazon EFS file system, which is specified in the StorageClass parameters.

Dynamic provisioning with EFS CSI driver is particularly useful for applications requiring ReadWriteMany access mode, where multiple pods on different nodes need to access the same shared storage simultaneously.

**Static Provisioning:**

In this mode, the Amazon EFS file system and its mount targets are provisioned manually outside of Kubernetes.

A Persistent Volume (PV) is then manually created in Kubernetes, referencing the pre-existing EFS file system.

Pods can then claim this PV using a Persistent Volume Claim (PVC) and mount the EFS volume.

This mode offers more control over the EFS file system configuration and is suitable for scenarios where the EFS volume is already managed or requires specific configurations not supported by dynamic provisioning.

Key Considerations:

**EFS File System:**

Both dynamic and static provisioning require a pre-existing Amazon EFS file system. The CSI driver does not create the EFS file system itself.

**Access Points:**

Dynamic provisioning utilizes EFS access points to provide isolation and manage permissions for individual PVs.\

**Fargate Compatibility:**

While the EFS CSI driver supports dynamic provisioning, there are specific considerations for Fargate pods, as the node agent component is built-in, but the controller might need to run on an EC2-based worker node.

**Permissions:**

Ensure appropriate EFS file system permissions and security group rules to allow NFS traffic from EKS worker nodes to the EFS mount targets.

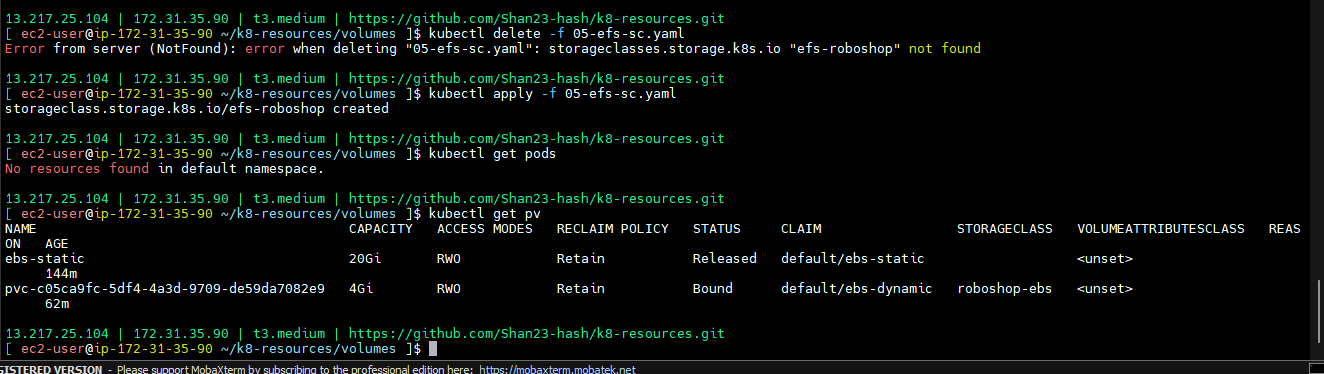
**--> kubectl delete -f 04-efs-static.yaml** (deleting load balancer)

--> Load balacer was created.

--> we can create pod

**--> kubectl apply -f 05-efs-sc.yaml**

**--> kubectl get pv**



**volumes/06-efs-dynamic.yaml**

apiVersion: v1

kind: PersistentVolumeClaim

metadata:

name: efs-dynamic

spec:

accessModes:

- ReadWriteOnce

storageClassName: "efs-roboshop"

resources:

requests:

storage: 5Gi

---

apiVersion: v1

kind: Pod

metadata:

name: nginx

labels:

purpose: efs-dynamic-demo

project: roboshop

spec:

containers:

- name: nginx

image: nginx

volumeMounts: # docker run -v host-path:container-path

- name: persistent-storage

mountPath: /usr/share/nginx/html

volumes:

- name: persistent-storage

persistentVolumeClaim:

claimName: efs-dynamic

---

apiVersion: v1

kind: Service

metadata:

name: efs-dynamic

labels:

purpose: efs-dynamic-demo

project: roboshop

spec:

type: LoadBalancer

selector:

purpose: efs-dynamic-demo

project: roboshop

ports:

- protocol: TCP

port: 80 # service port

targetPort: 80 # container port

--> pvc created

--> storage class name mention mandatory

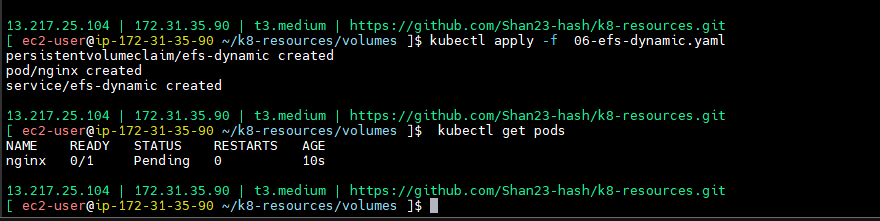
--> push and pull the code

**--> kubectl apply -f 05-efs-sc.yaml**

**--> kubectl get sc** (roboshop was with ratain codes)

--> **kubectl apply -f 06-efs-dynamic.yaml**

**--> kubectl get pods**



**-->** got error delete pod ,amually and try

**--> kubectl delete pod nginx**

1. EFS volume creation --> delete this next

2. PV, PVC creation --> delete these first

--> **kubectl apply -f 06-efs-dynamic.yaml**

**--> kubectl get pods**

--> Pod was created here will get one access point

--> Go to EFS and check

--> data will access only for this application

--> access points will create dynamically

--> EFS --> amazon efs --> access points

**--> kubectl get pv** (same name came in volumes)

--> **kubectl delete -f 06-efs-dynamic.yaml**

stateful applications --> nothing but Data Bases

Deployment --> stateless applications not for stateful applications

Statefulset --> stateful applications.

--> foe statefull set this pv and pvc is mandatory because you need to store the data outside of kubernates that is cluster.

**Deployment vs Statefulset**

**==========================**

1. Deployment is for stateless applications, Statefulset is stateful applications

2. PV and PVC is not mandatory for Deployment, but mandatory for stateful applications

3. Statefulset need headless service... i.e no cluster IP

4. Pods create in orderly manner in statefulset, Once first pod comes to running, then only other pod will create. While deletion reverse order follows

5. Pod identities are preserved in statefulset, because if any pod crashes, statefulset create another pod with same name, so that communication is easy between pods..

1. **deployment.yaml**

apiVersion: apps/v1

kind: Deployment

metadata:

name: nginx

# deployment labels

labels:

purpose: deployment-demo

project: roboshop

app: nginx

spec:

replicas: 3

# These are the labels replica set use to create pod replicas, this should match pod labels

selector:

matchLabels:

purpose: deployment-demo

project: roboshop

app: nginx

# This is pod definition

template:

metadata:

labels:

purpose: deployment-demo

project: roboshop

app: nginx

spec:

containers:

- name: nginx

image: nginx:latest

---

apiVersion: v1

kind: Service

metadata:

name: nginx-svc

labels:

purpose: service-demo

spec:

selector:

purpose: deployment-demo

project: roboshop

app: nginx

ports:

- protocol: TCP

port: 80 # service port

targetPort: 80 # container port

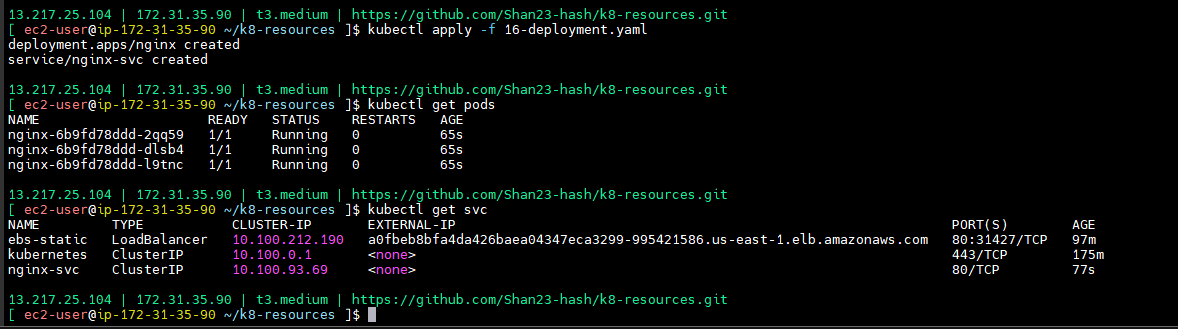
--> push and pull the code

--> **kubectl apply -f 16-deployment.yaml**

**--> kubectl get pods**

--> Now created 3 replicas and one service came

--> **kubectl get svc**



--> pod to pod communication we will hit service.

--> Let me create the debug also.

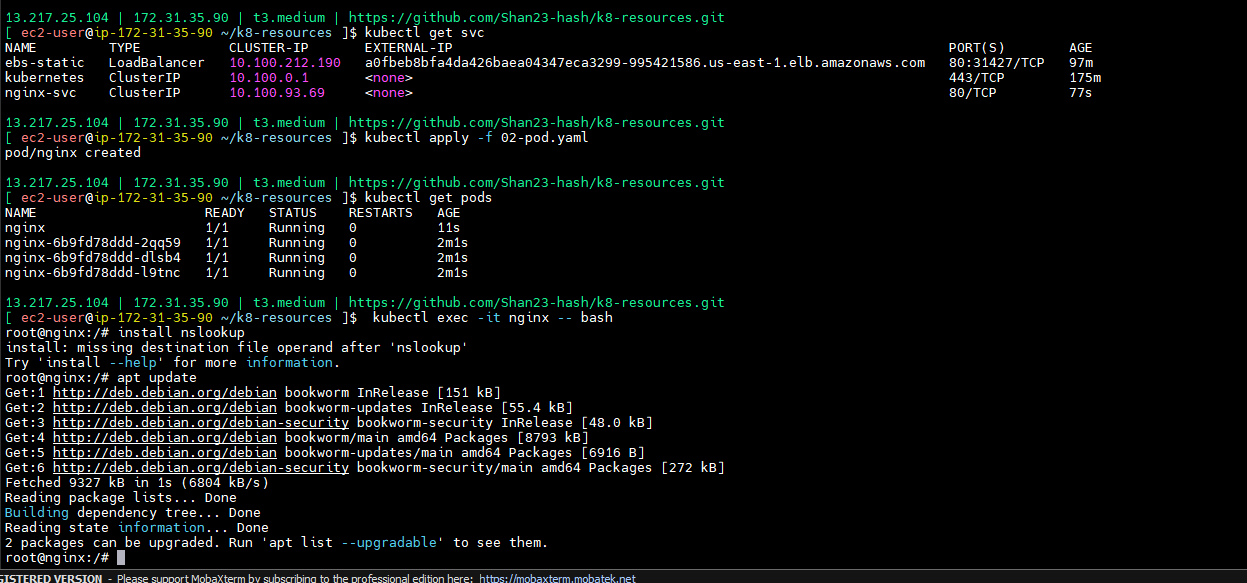
--> **kubectl apply -f 02-pod.yaml**

**--> kubectl get pods**

**--> kubectl exec -it nginx -- bash**

**-->** install nslookup

--> **apt update**

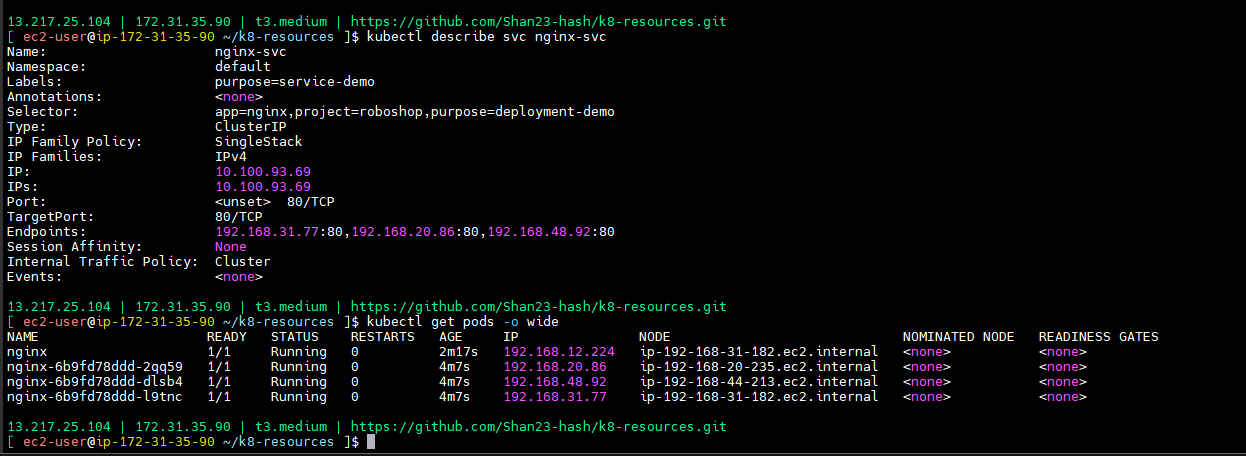


**-->** address came 9.154means nginx cluster IP

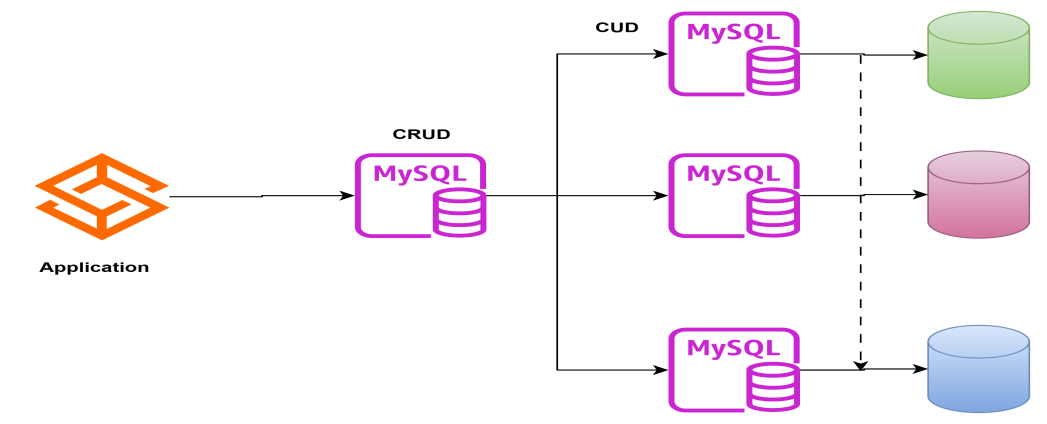
nslookup nginx-svc --> Cluster IP and then see end points

--> **kubectl describe svc nginx-svc**

**--> kubectl get pods -o wide**



**-->** service will send to any one pod.



1. DB node is responsible to find out all the worker nodes and inform them about data changes.

headless service, it is a service without cluster IP

nslookup nginx-headless-svc --> all the endpoint IP

1. **statefulset.yaml**

apiVersion: v1

kind: Service

metadata:

name: nginx-svc-headless

labels:

purpose: statefulset

project: roboshop

spec:

ports:

- port: 80

name: web

clusterIP: None

selector:

purpose: statefulset

project: roboshop

---

apiVersion: v1

kind: Service

metadata:

name: nginx-svc-normal

labels:

purpose: statefulset

project: roboshop

spec:

ports:

- port: 80

name: web

selector:

purpose: statefulset

project: roboshop

---

apiVersion: apps/v1

kind: StatefulSet

metadata:

name: nginx-statefulset

spec:

selector:

matchLabels:

purpose: statefulset

project: roboshop # has to match .spec.template.metadata.labels

serviceName: "nginx-svc-headless" # this should be headless service

replicas: 3 # by default is 1

template:

metadata:

labels:

purpose: statefulset

project: roboshop # has to match .spec.selector.matchLabels

spec:

containers:

- name: nginx

image: nginx

volumeMounts:

- name: www

mountPath: /usr/share/nginx/html

# This is nothing but PVC, you can directly create here

volumeClaimTemplates:

- metadata:

name: www

spec:

accessModes: [ "ReadWriteOnce" ]

storageClassName: "efs-roboshop"

resources:

requests:

storage: 2Gi

1. EBS dynamic provisioning
2. **Kubectl apply -f 02-sc.yaml**
3. **Kubectl get sc**

**-->** cluster ip none what we will cal

--> if cluster ip none is there that is called headless service.

--> match labels

--> pod labels

--> all places will maintain same name

--> this is nothing but pvc.

--> storage class is roboshop eps

--> requesting 2GB

--> POd to pod how it will communicate you need to create normal service

--> Remove cluster IP.

--> Normal service,headless service,PVC all should be there

--> if you do all these it will work properly your application in kubernates.

--> Push and pull the code

**--> Kubectl apply -f 17-statefulset.yaml**

**--> Kubectl get pods** (not created will see the reason)

**--> kubectl describe pod sc-headless-0**

**-->** here events are not coming

**--> Kubectl get pvc**

**--> kubectl describe pod sc-headless-0**

**-->** problem is drivers not installed.

**--> kubectl delete -f 17-statefulset.yaml**

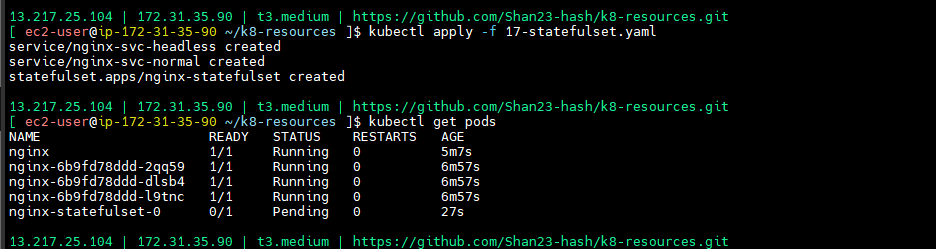
--> Let me take efs roboshop

--> Push and pull the code

**--> kubectl apply -f 17-statefulset.yaml**

**--> kubectl get pods**

**--> kubectl get pvc**



**--> kubectl describe pvc www-sc-headless-0**

--> it is goiing for ebs

**--> kubectl delete -f 17-statefulset.yaml**

**--> kubectl delete pvc www-sc-headless-0**

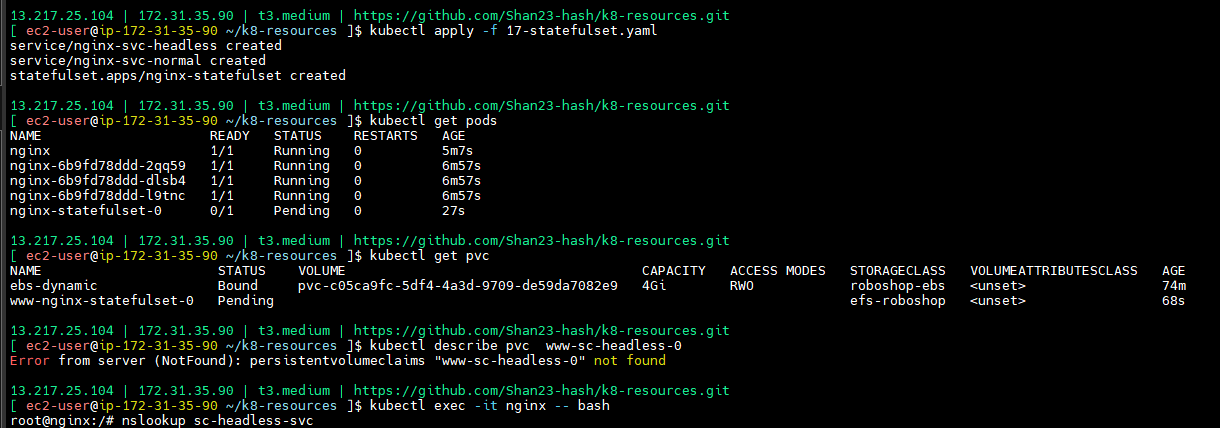
**--> Kubectl apply -f 17-statefulset.yaml**

**--> Kubectl get pvc**

**--> Kubectl get pods**

**--> kubectl exec -it nginx -- bash**

**--> nslookup sc-headless-svc**



--> These all are pod addresses

--> why heaadless created -- if request came, on headless service one nslook up will come, headless is imp For internal communcation.

--> outsides pods will reach normal services

--> this is purely for database stateful set, Our databases no need to keep in kubernates.

--> here names is very important

--> pods will create in orderly manner

--> for databases Pods create in orderly manner in statefulset, Once first pod comes to running, then only other pod will create. While deletion reverse order follows

--> Pod identities are preserved in statefulset, because if any pod crashes, statefulset create another pod with same name, so that communication is easy between pods..

--> POD will create same name because there is dependency between these TWO.

--> POD identity means pod names

**--> stateful vc deplyment (popular interview question)**

Kubernetes Deployments and StatefulSets are both controllers used to manage Pods, but they are designed for different types of applications:

**1. Deployments:**

**Purpose**: Ideal for stateless applications where Pods are interchangeable and do not require stable identities or persistent storage tied to individual instances.

**Pod Management**: Treats Pods as "cattle," meaning they are expendable and can be easily replaced without affecting the application's functionality.

**Scaling**: Scales Pods up and down without any specific ordering.

**Networking:** Pods receive dynamic, non-stable hostnames and IP addresses.

Storage: Typically uses shared PersistentVolumeClaims (PVCs) if persistent storage is required, meaning all Pod replicas share the same volume.

Use Cases: Web servers, microservices, stateless APIs.

**2. StatefulSets:**

**Purpose:**

Designed for stateful applications that require stable, unique identities, ordered deployment, and persistent storage tied to specific Pod instances.

**Pod Management:**

Treats Pods as "pets," meaning each Pod has a unique identity that persists across restarts and reschedulings.

**Scaling:**

Guarantees ordered deployment and scaling of Pods, which is crucial for applications like databases or distributed systems that rely on specific node order.

**Networking:**

Provides stable, predictable hostnames and network identities for each Pod (e.g., pod-name-0, pod-name-1).

**Storage:**

Utilizes volumeClaimTemplates to automatically provision a unique PersistentVolumeClaim (and thus a unique PersistentVolume) for each Pod replica.

**Use Cases:**

Databases (MySQL, PostgreSQL), distributed file systems (Hadoop), message queues (Kafka).

|  |  |  |
| --- | --- | --- |
| Feature | Deployment | StatefulSet |
| Application Type | Stateless applications (e.g., web servers, APIs) | Stateful applications (e.g., databases, distributed systems) |
| Pod Identity | Pods are interchangeable and lack stable identities. | Each pod has a stable, unique network identity and hostname (e.g., <statefulset-name>-<pod-ordinal-index>). |
| Storage | Typically uses ephemeral storage. Persistent volumes can be used but are not tightly integrated with pod identity. | Each pod has its own PersistentVolume (PV) and PersistentVolumeClaim (PVC), ensuring data persistence across restarts. |
| Scaling | Scales out replicas freely without considering order or individual pod identities. | Scales out replicas in a predictable order, maintaining individual pod identities. |
| Updates | Supports rolling updates by default, replacing old pods with new ones gradually. | Supports rolling updates, but updates proceed in reverse order of the pod ordinal index, ensuring consistency for stateful applications. |