**Name:** Atif Ansari

**Roll no:** 04

**Class:** D15B

**Experiment 3**

**Aim:** To understand the Kubernetes Cluster Architecture, install and Spin Up a Kubernetes Cluster on Linux Machines/Cloud Platforms.

**Theory:**

Container-based microservices architectures have revolutionized how development and operations teams test and deploy modern software. Containers allow companies to scale and deploy applications more efficiently, but they also introduce new challenges, adding complexity by creating a whole new infrastructure ecosystem.

Today, both large and small software companies are deploying thousands of container instances daily. Managing this level of complexity at scale requires advanced tools. Enter Kubernetes.

Originally developed by Google, Kubernetes is an open-source container orchestration platform designed to automate the deployment, scaling, and management of containerized applications. Kubernetes has quickly become the de facto standard for container orchestration and is the flagship project of the Cloud Native Computing Foundation (CNCF), supported by major players like Google, AWS, Microsoft, IBM, Intel, Cisco, and Red Hat.

Kubernetes simplifies the deployment and operation of applications in a microservice architecture by providing an abstraction layer over a group of hosts. This allows development teams to deploy their applications while Kubernetes takes care of key tasks, including:

* Managing resource consumption by applications or teams
* Distributing application load evenly across the infrastructure
* Automatically load balancing requests across multiple instances of an application
* Monitoring resource usage to prevent applications from exceeding resource limits and automatically restarting them if needed
* Moving application instances between hosts when resources are low or if a host fails
* Automatically utilizing additional resources when new hosts are added to the cluster
* Facilitating canary deployments and rollbacks with ease

**Necessary Requirements:**

* **EC2 Instance:** The experiment required launching a t2.medium EC2 instance with 2 CPUs, as Kubernetes demands sufficient resources for effective functioning.
* **Minimum Requirements:**
  + **Instance Type:** t2.medium
  + **CPUs:** 2
  + **Memory:** Adequate for container orchestration.

This ensured that the Kubernetes cluster had the necessary resources to function smoothly

Note:

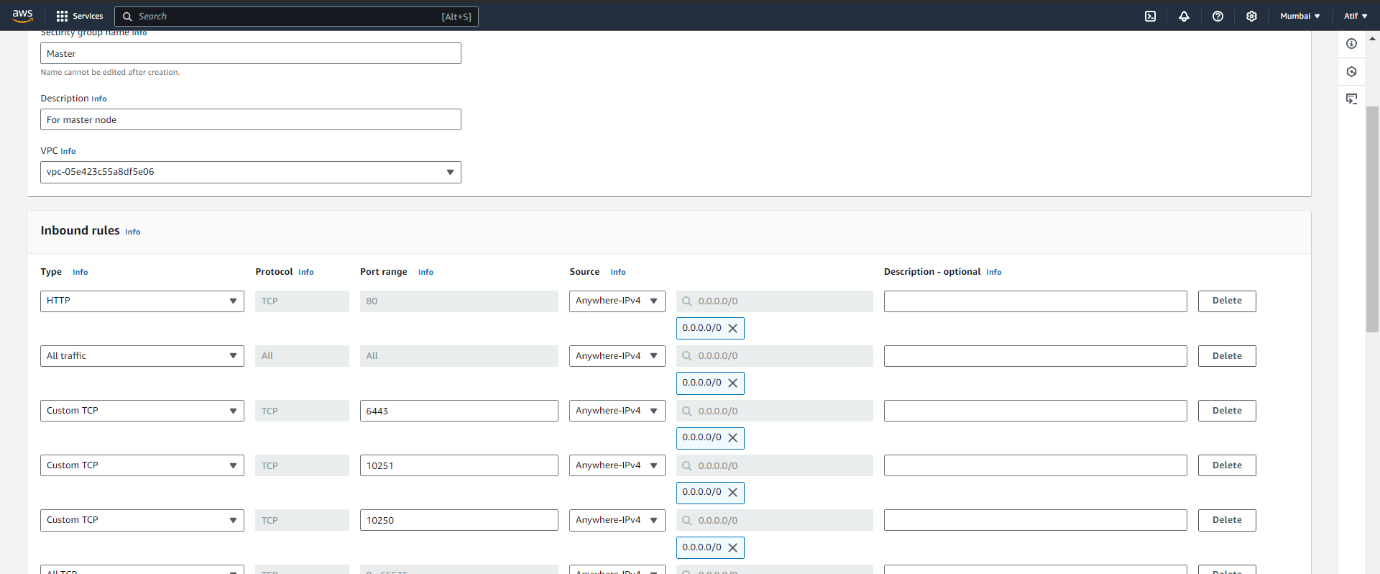
AWS Personal Account is preferred but we can also perform it on AWS Academy(adding some ignores). If You are using AWS Academy Account Errors you will face in kubeadm init command so you have to add some ignores with this command.

in the command if any error occurs in below as the below experiment is performed on Personal Account

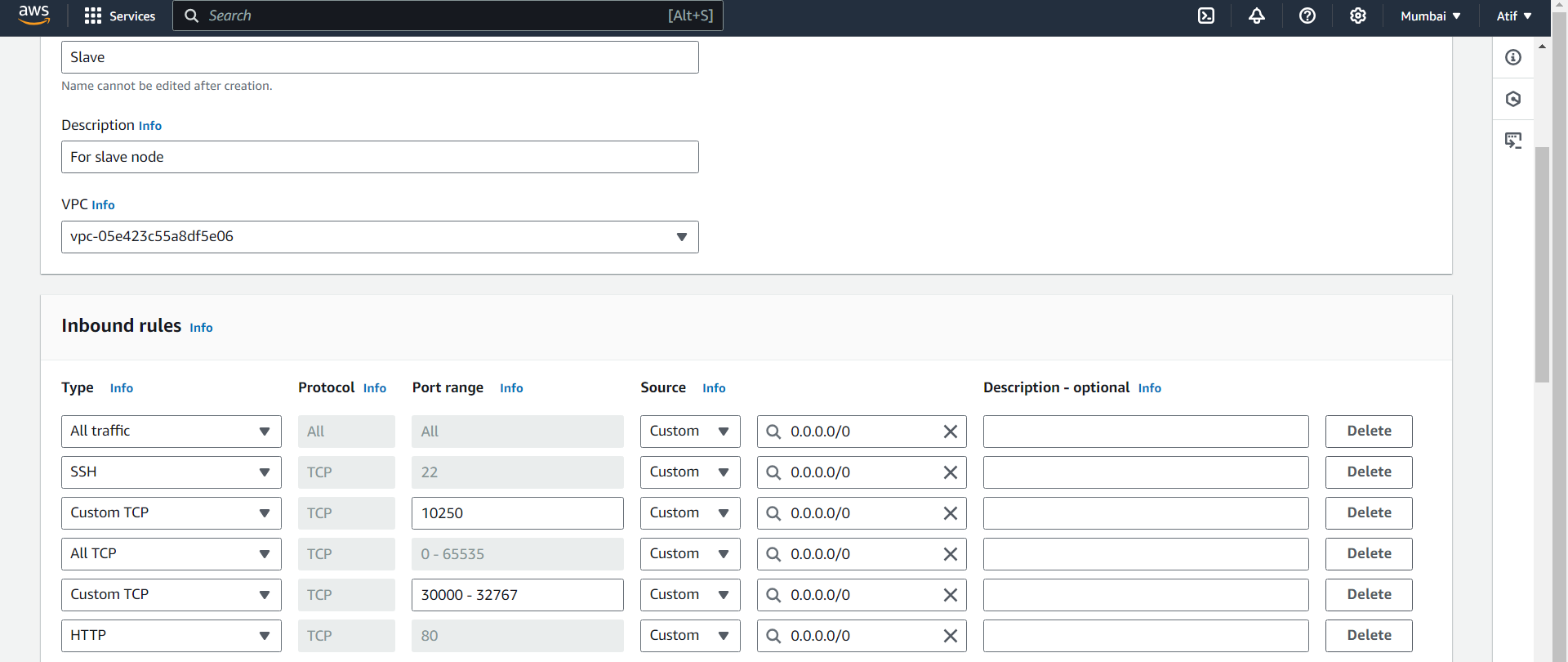
**Prerequisites :**

**Create 2 Security Groups for Master and Nodes and add the following rules inbound rules in those Groups.**

**Master:**



**Node :**



**Step 1:** Log in to your AWS Academy/personal account and launch 3 new Ec2 Instances.

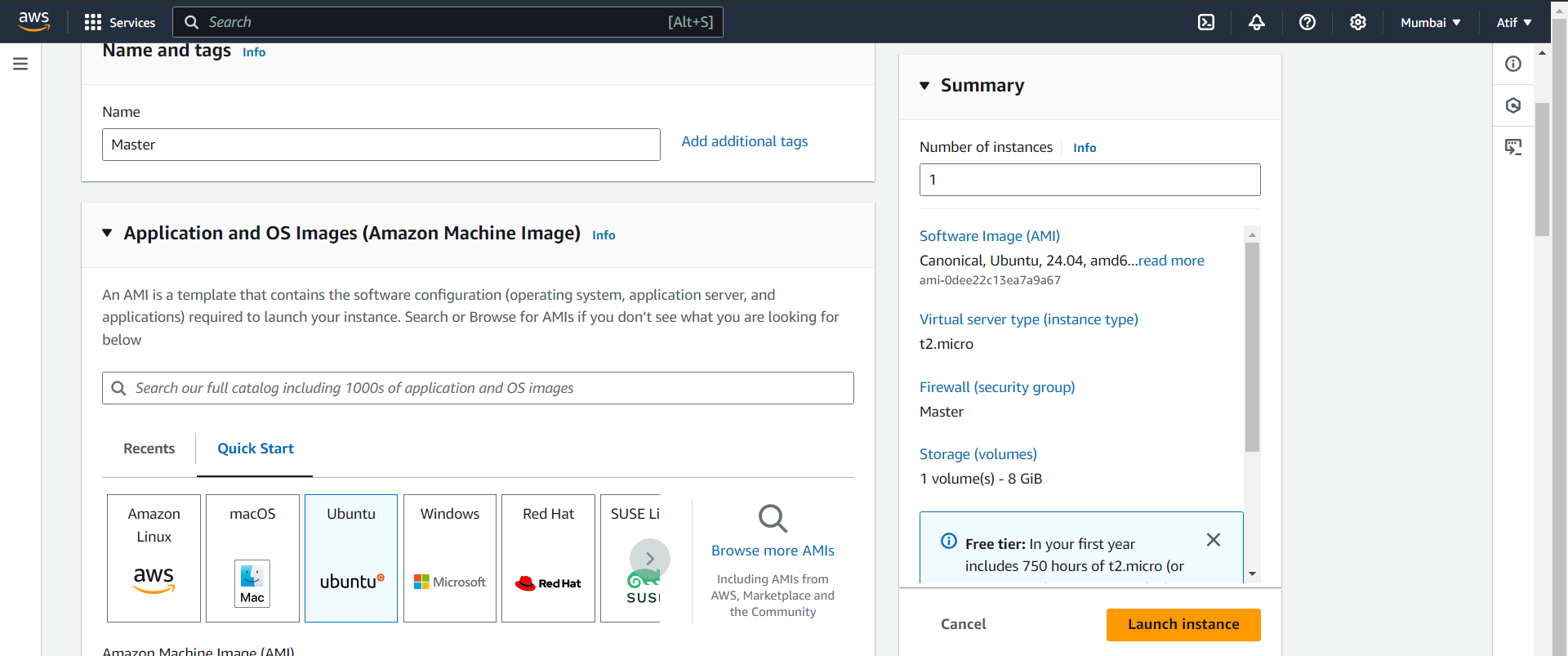
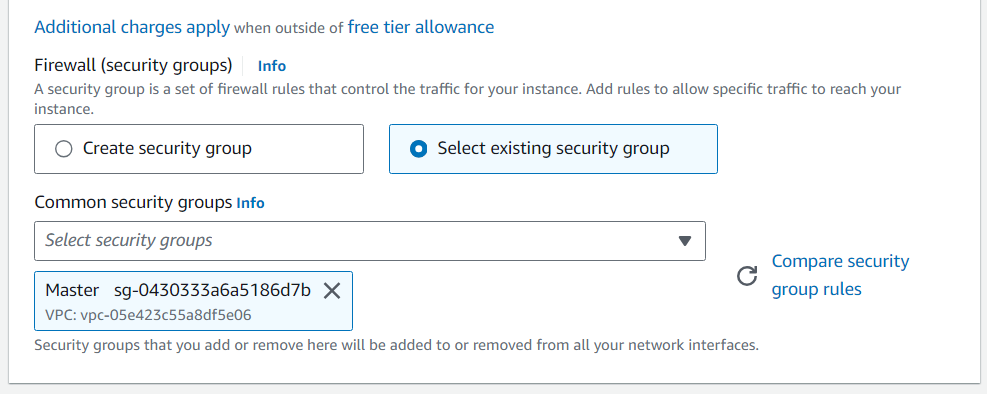
Select Ubuntu as AMI and t2.medium as Instance Type and create a key of type RSA with .pem extension and move the downloaded key to the new folder.We can use 3 Different keys or 1 common key also.

Note: A minimum of 2 CPUs are required so Please select t2.medium and do not forget to stop the

instance after the experiment because it is not available in the free tier.

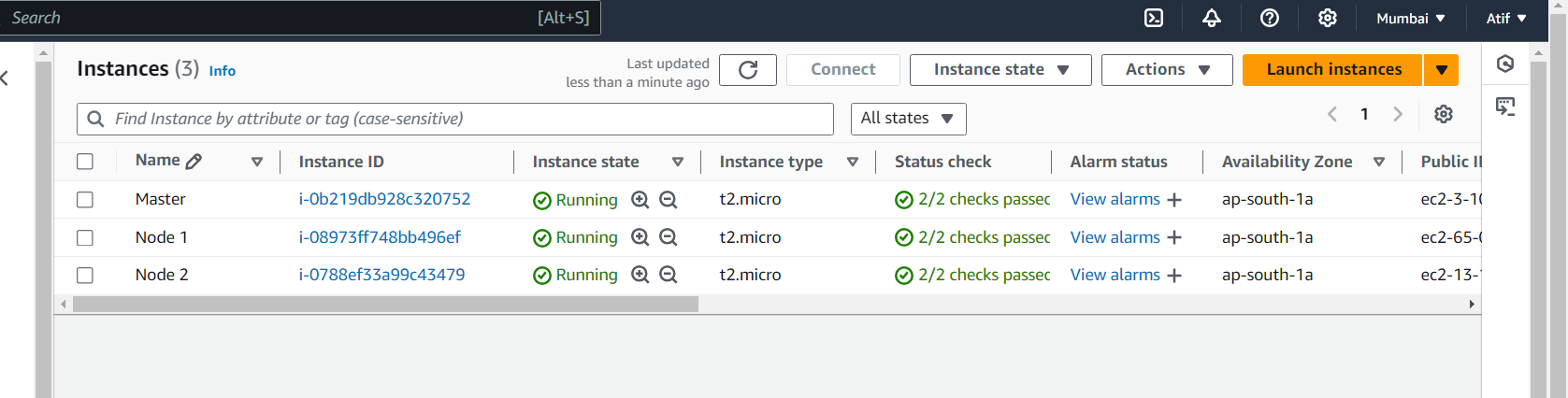
**Also Select Security groups from existing.**

**Master:**

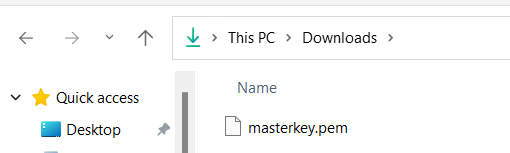
 

**Do Same for 2 Nodes and use security groups of Node for that.**

**Step 2:** After creating the instances click on Connect & connect all 3 instances and navigate to SSH Client.

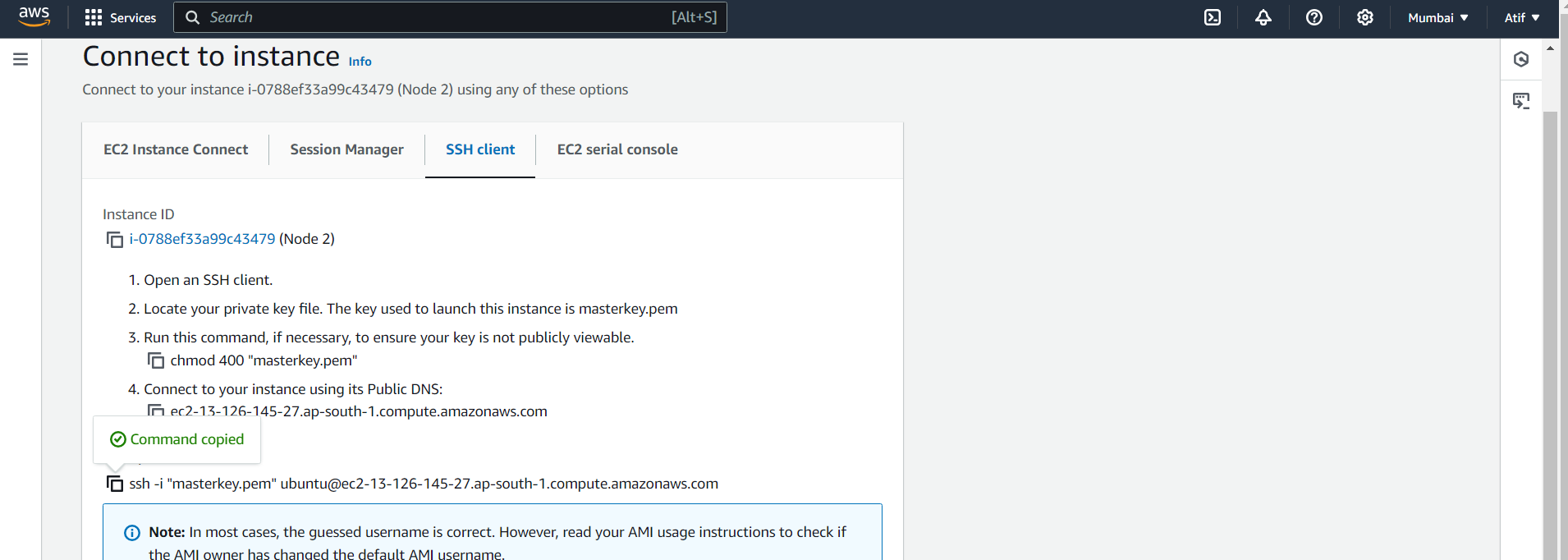


**(Downloded Key)**

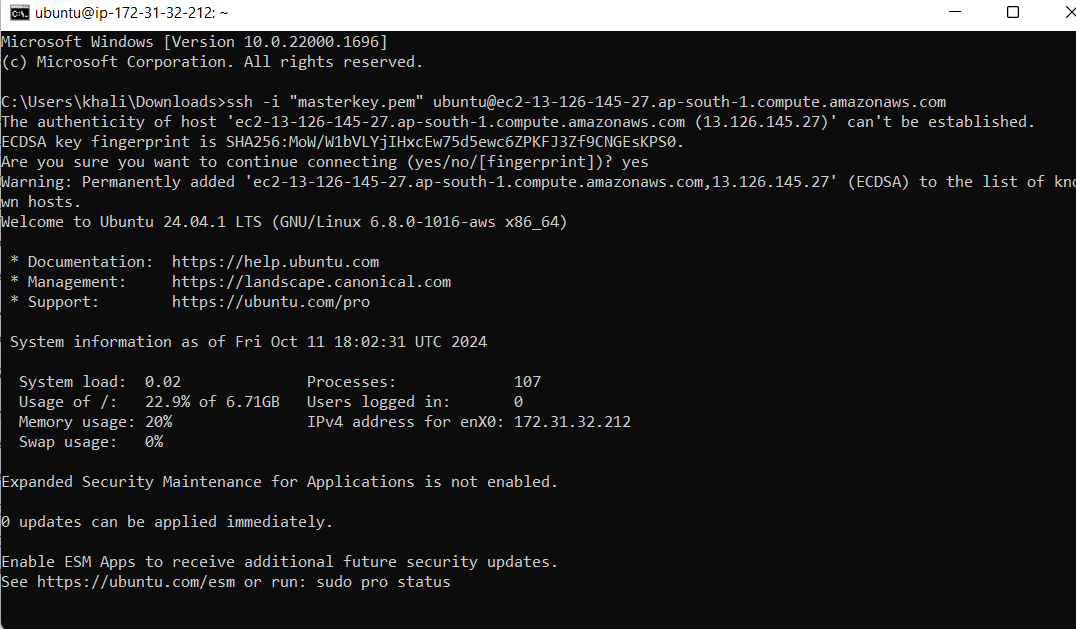
****

**Step 3:** Now open the folder in the terminal 3 times for Master, Node1& Node 2 where our .pem key is stored and paste the Example command (starting with ssh -i …..) in the terminal

Master:



Successful Connection:



**Step 4:** Run on Master,Node 1,and Node 2 the below commands to install and setup Docker in Master, Node1, and Node2.

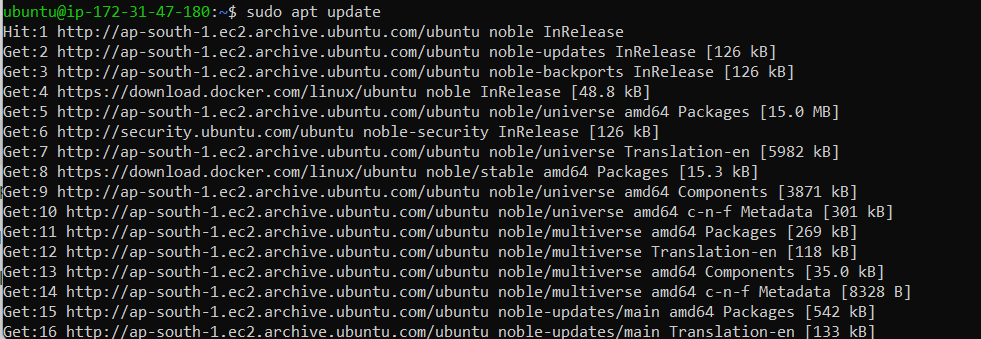
**curl -fsSL https://download.docker.com/linux/ubuntu/gpg | sudo apt-key add - curl -fsSL https://download.docker.com/linux/ubuntu/gpg | sudo tee**

**/etc/apt/trusted.gpg.d/docker.gpg > /dev/null**

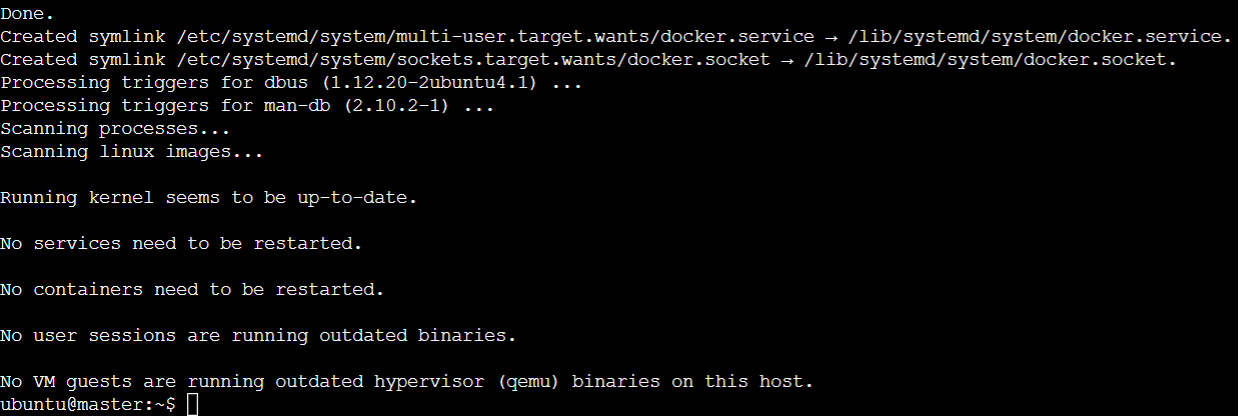
**sudo add-apt-repository "deb [arch=amd64] https://download.docker.com/linux/ubuntu**

**$(lsb\_release -cs) stable"**

**sudo apt-get update**



**sudo apt-get install -y docker-c**



**sudo mkdir -p /etc/docker**

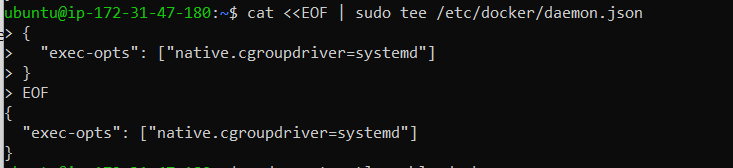
**cat <<EOF | sudo tee /etc/docker/daemon.json**

**{**

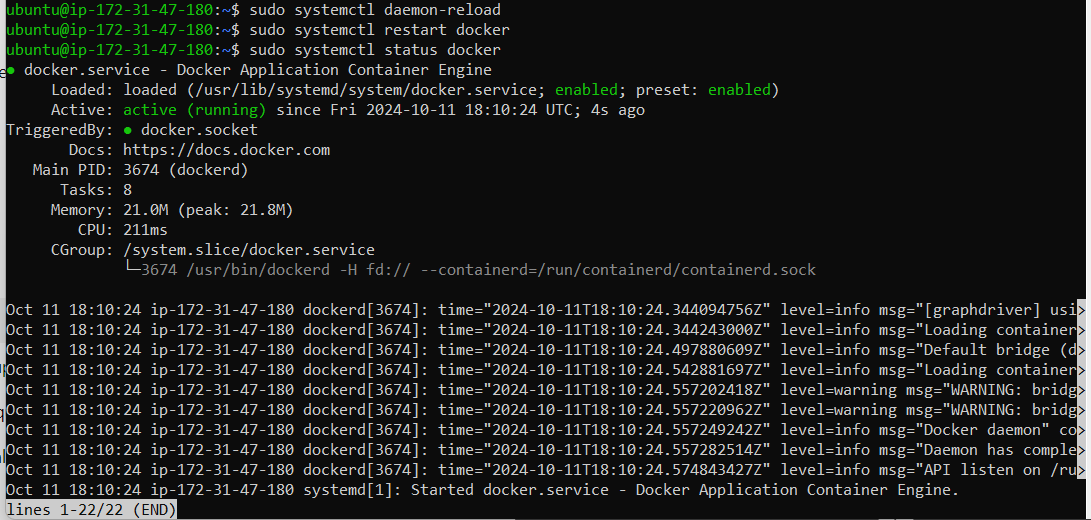
**"exec-opts": ["native.cgroupdriver=systemd"]**

**}**

**EOF**



**sudo systemctl enable docker sudo systemctl daemon-reload sudo systemctl restart docker**

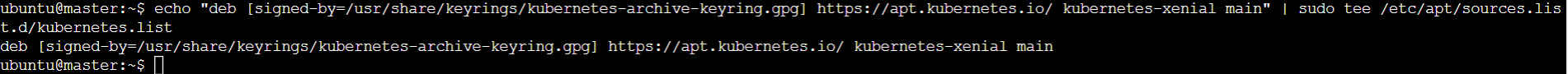


**Step 5:** Run the below command to install Kubernets.

**curl -fsSL https://pkgs.k8s.io/core:/stable:/v1.31/deb/Release.key | sudo gpg --dearmor -o**

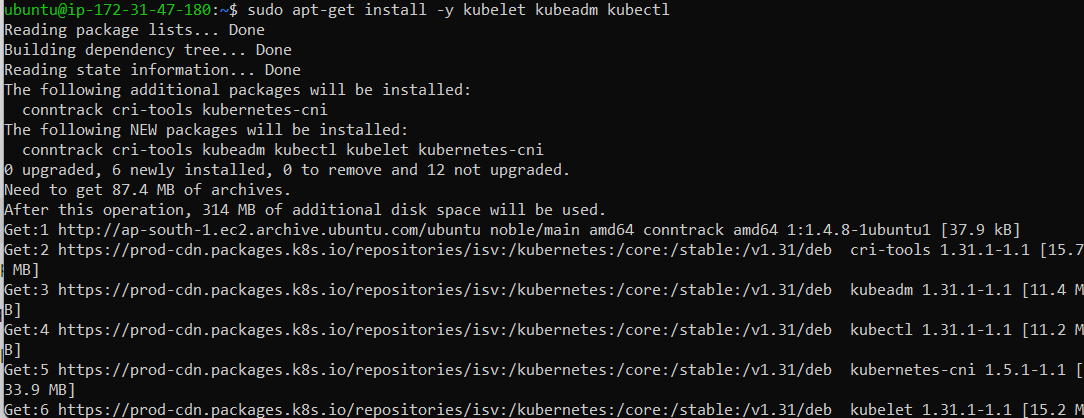
**/etc/apt/keyrings/kubernetes-apt-keyring.gpg**

**echo 'deb [signed-by=/etc/apt/keyrings/kubernetes-apt-keyring.gpg] https://pkgs.k8s.io/core:/stable:/v1.31/deb/ /' | sudo tee /etc/apt/sources.list.d/kubernetes.list**

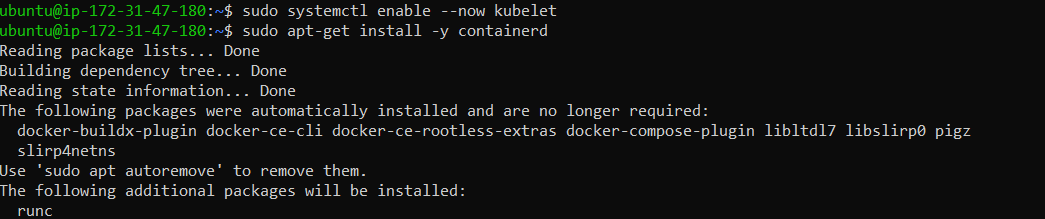
****

**sudo apt-get update**

**sudo apt-get install -y kubelet kubeadm kubectl sudoapt-mark hold kubelet kubeadm kubectl**



**sudo systemctl enable --now kubelet sudo apt-get install -y containerd**

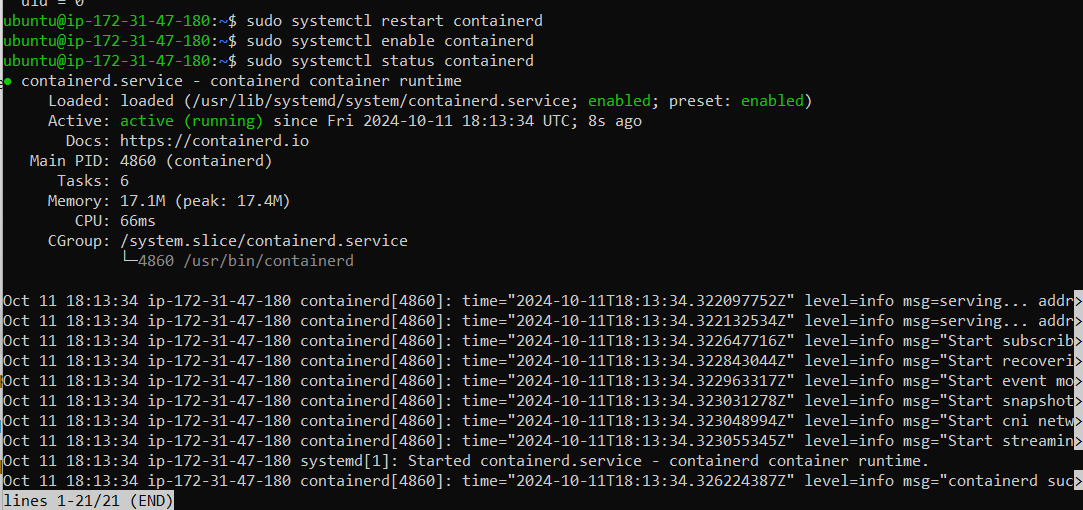


**sudo mkdir -p /etc/containerd**

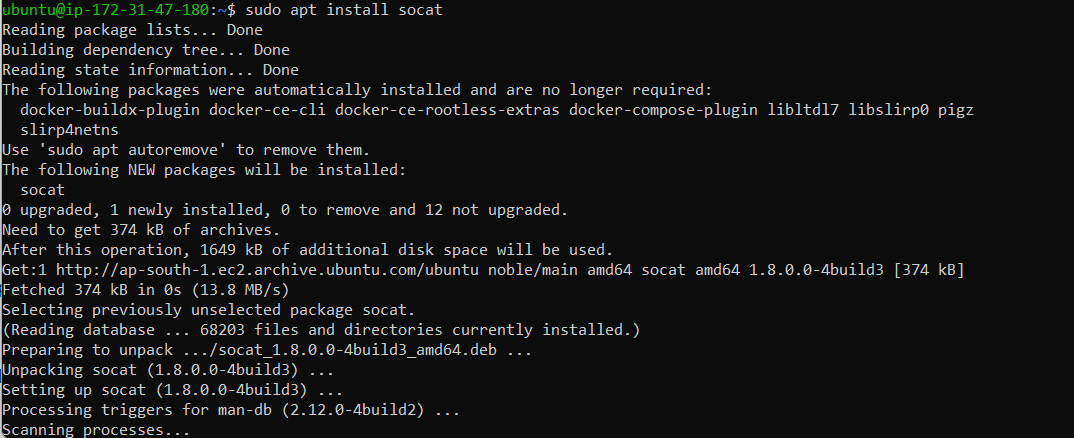
**sudo containerd config default | sudo tee /etc/containerd/config.toml**



**sudo systemctl restart containerd sudo systemctl enable containerd sudo systemctl status containerd**

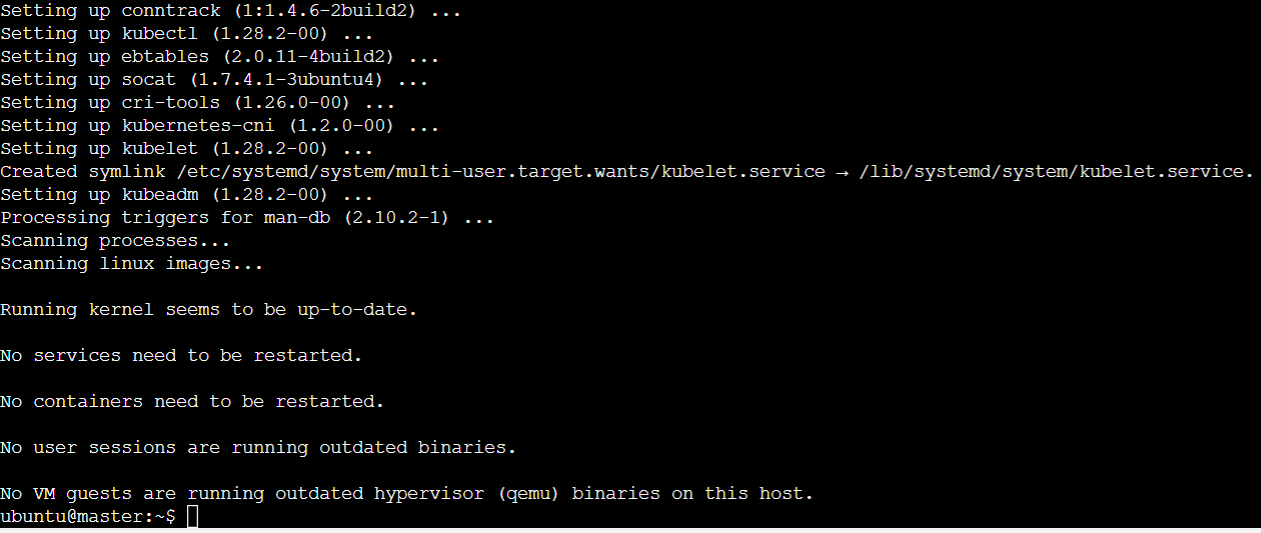


**sudo apt-get install -y socat**



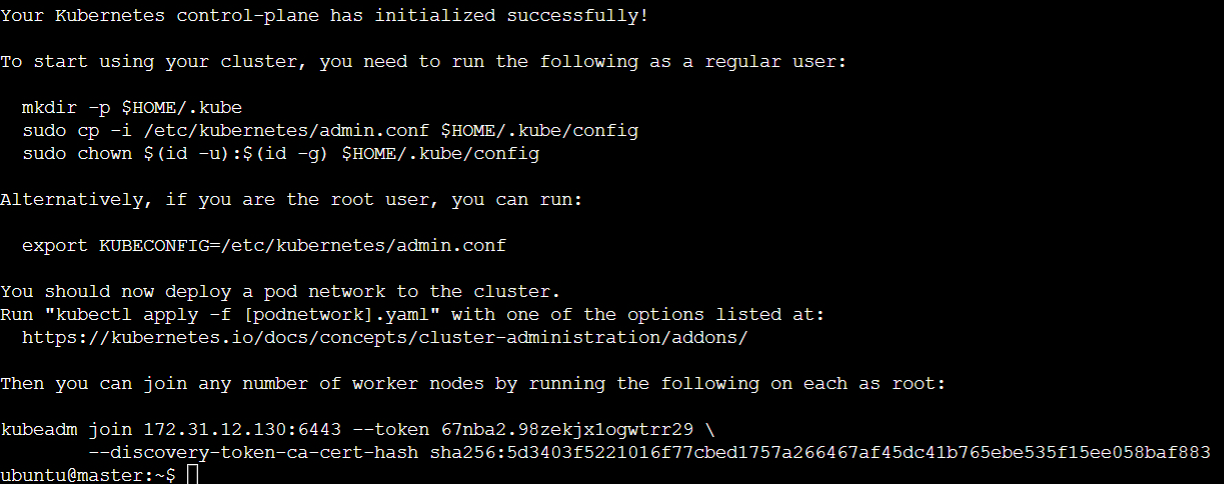
**Step 6:** Initialize the Kubecluster .Now Perform this Command only for Master.

**sudo kubeadm init --pod-network-cidr=10.244.0.0/16**

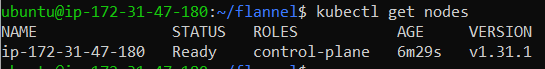
****

**Run this command on master and also copy and save the Join command from above. mkdir -p $HOME/.kube**

**sudo cp -i /etc/kubernetes/admin.conf $HOME/.kube/config sudo chown $(id -u):$(id -g) $HOME/.kube/config**

****

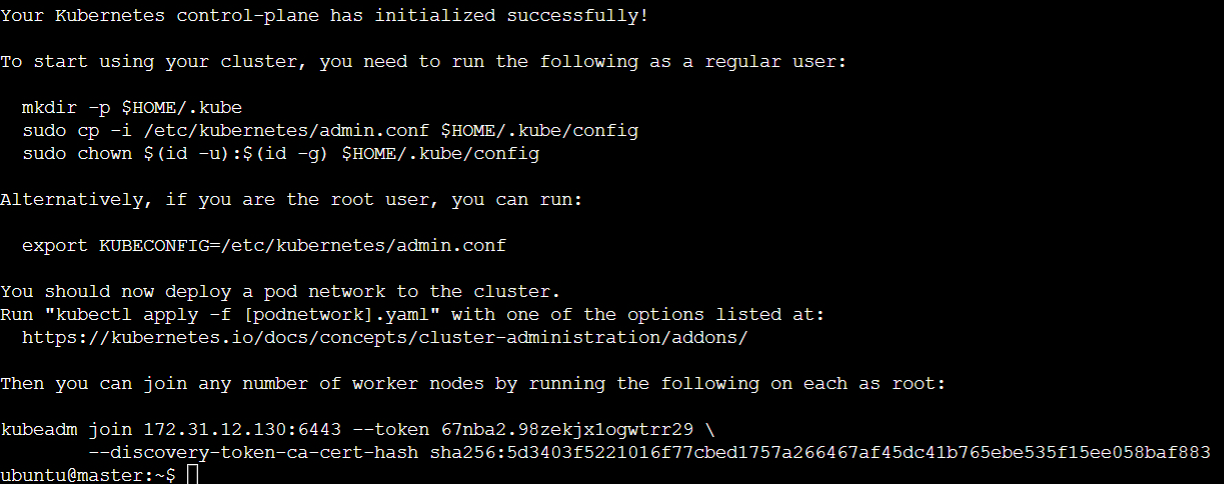
**Step 7: Now Run the command kubectl get nodes to see the nodes before executing Join command on nodes.**



**Step 8: Now Run the following command on Node 1 and Node 2 to Join to master. sudo kubeadm join 172.31.27.176:6443 --token ttay2x.n0sqeukjai8sgfg3 \**

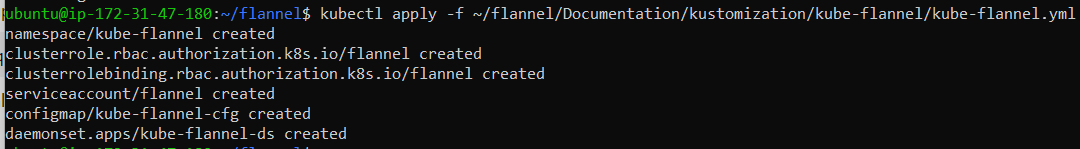
**--discovery-token-ca-cert-hash**

**sha256:d6fc5fb7e984c83e2807780047fec6c4f2acfe9da9184ecc028d77157608fbb6**

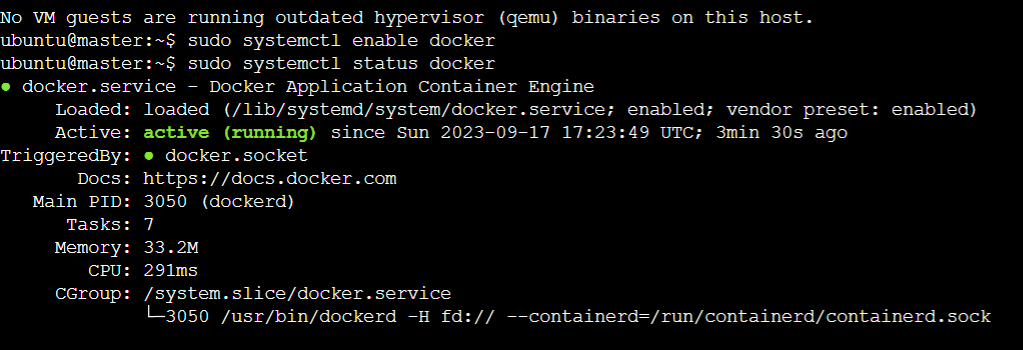
****

**Step 9: Now Run the command kubectl get nodes to see the nodes after executing Join command on nodes. Since Status is NotReady we have to add a network plugin. And also we have to give the name to the nodes.**

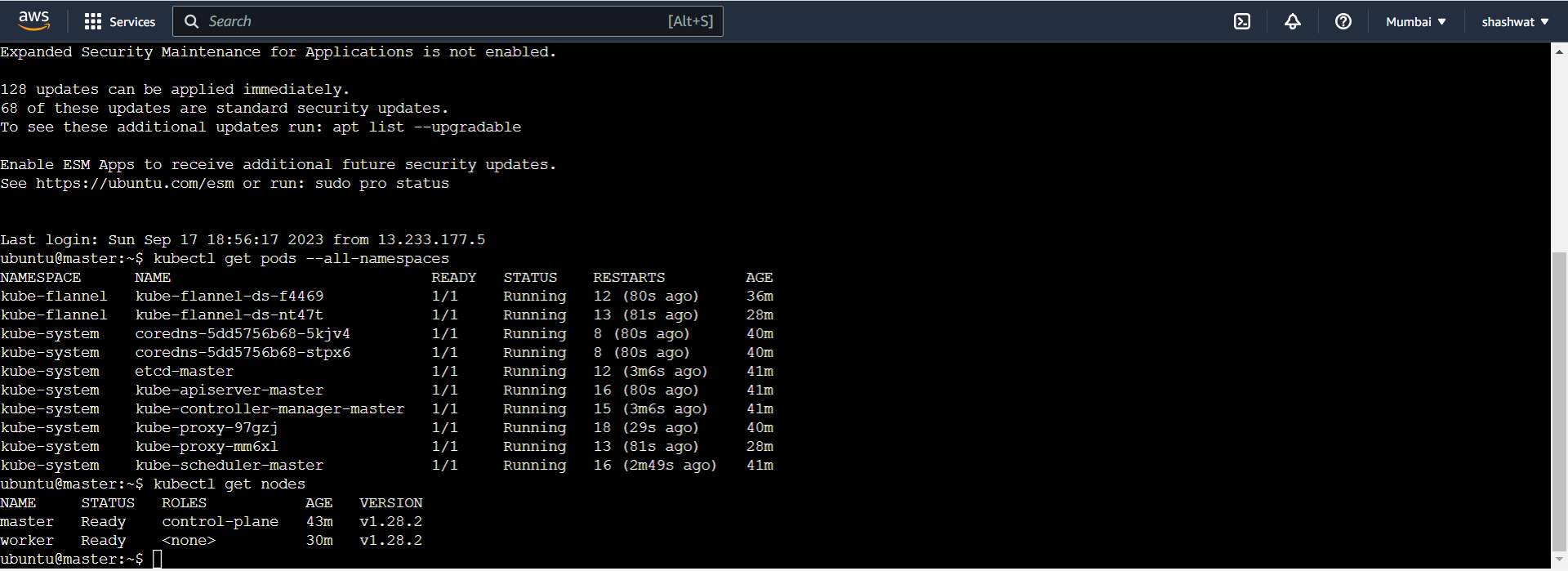
**kubectl apply -f https://docs.projectcalico.org/manifests/calico.yaml**



**sudo systemctl status kubelet**



**Step 10: Run command kubectl get nodes -o wide . And Hence we can see we have Successfully connected Node 1 and Node 2 to the Master.**

****

**Conclusion:** In this experiment, we successfully set up a Kubernetes cluster with one master and two worker nodes on AWS EC2 instances. After installing Docker, Kubernetes tools (kubelet, kubeadm, kubectl), and containerd on all nodes, the master node was initialized and the worker nodes were joined to the cluster. Initially, the nodes were in the NotReady state, which was resolved by installing the Calico network plugin. We also labeled the nodes with appropriate roles (control-plane and worker). The cluster became fully functional with all nodes in the Ready state, demonstrating the successful configuration and orchestration of Kubernetes.