**Infra Optimization**

**- *Saaketh Gunturu***

***Course End Project #1***

***PGP DO - DevOps Capstone Project***

***PGP DO JUL 2022 Cohort 1***

**Introduction:**

*Description: Create a DevOps infrastructure for an e-commerce application to run on high-availability mode.*

This project is to create a DevOps infrastructure for an application that runs of high availability mode. This infrastructure is built in AWS. It contains three EC2 instances which acts as master and worker nodes for kubernetes. These VMs are provisioned in AWS using Ansible ans are installed with Docker and Kubernetes. Once the VMs are ready with Docker and Kubernetes, the cluster setup is established between master and worker nodes.

A spring boot application is taken as an example application for this project. The application has been dockerized and the image is created in master. Using the image, a deployment is created which deployed pods in worker nodes with Nodeport service. This pods also setup with HPA with CPU availability target of 50%. Once the target is reached the autoscaling is kicked in and a new pod is deployed to managed the high availability.

**Tools used:**

1. Ansible - for configuration
2. AWS - EC2 instances
3. Docker
4. Kubernetes - Cluster management
5. Jmeter - Load testing
6. Github - Source control
7. Git - version control

**Github repository location:**

Below are the list of github repository location for the project. The repository contains all the code used to build this project. The document folder contains all the screenshots and a recording of the load test performed.

Repository:

<https://github.com/saakethg7/simplilearn-capstone-prj-1>

Documents:

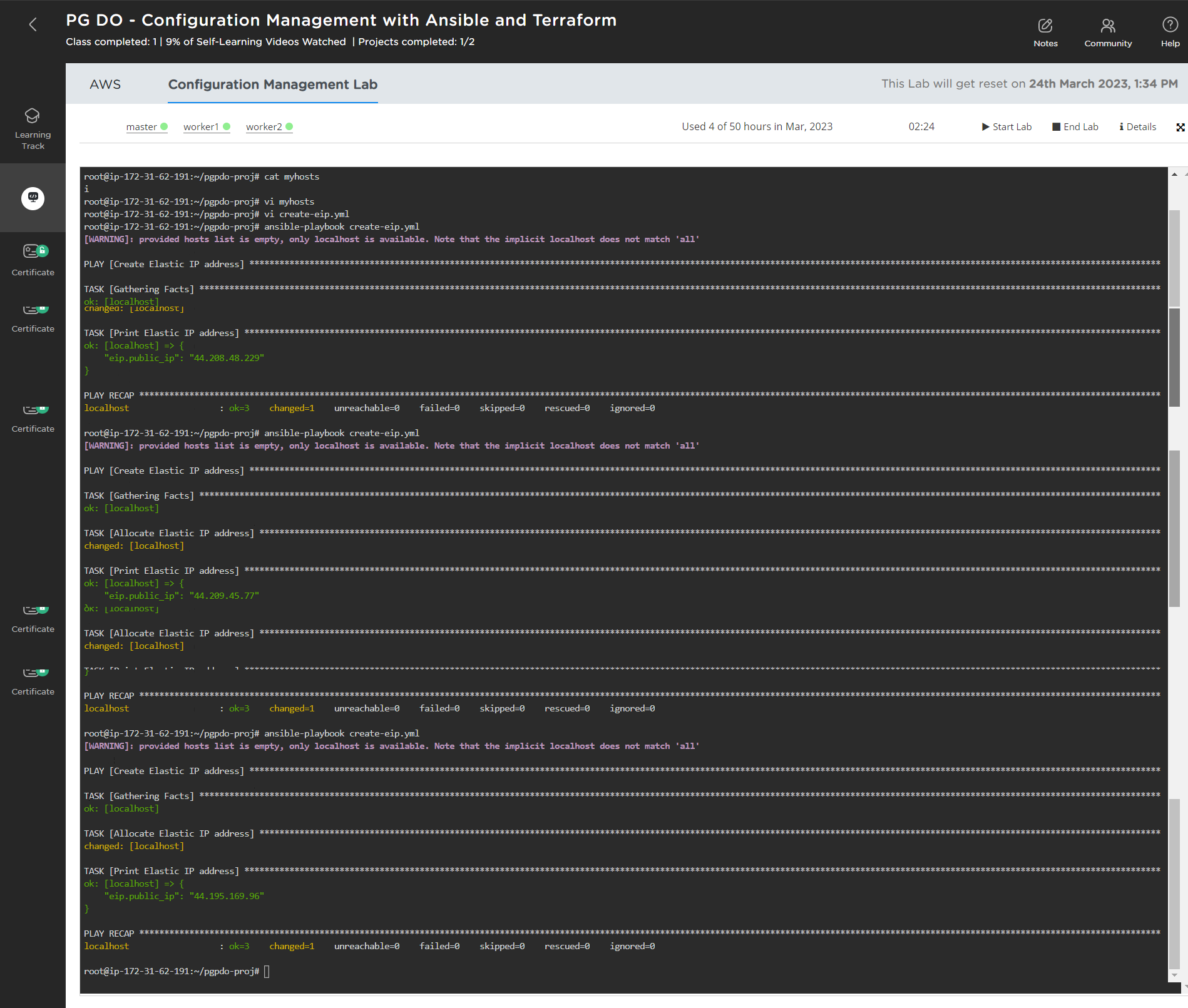
<https://github.com/saakethg7/simplilearn-capstone-prj-1/tree/master/documents>

**Tasks Performed to fulfil the project:**

1. **Provision an EC2 instances**

*Create an Elastic IP address using Ansible:*

To access EC2 instances without any issue, Elastic IP address are created using Ansible.



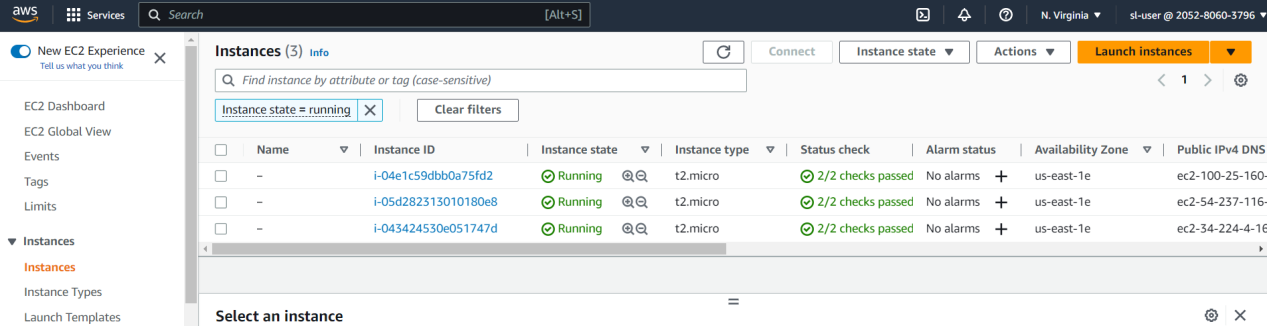
*Provision EC2 instances using Ansible:*

Using Ansible playbook, the EC2 instances are created. Three EC2 instances are created one works as master node and the rest works are worker nodes in kubernetes



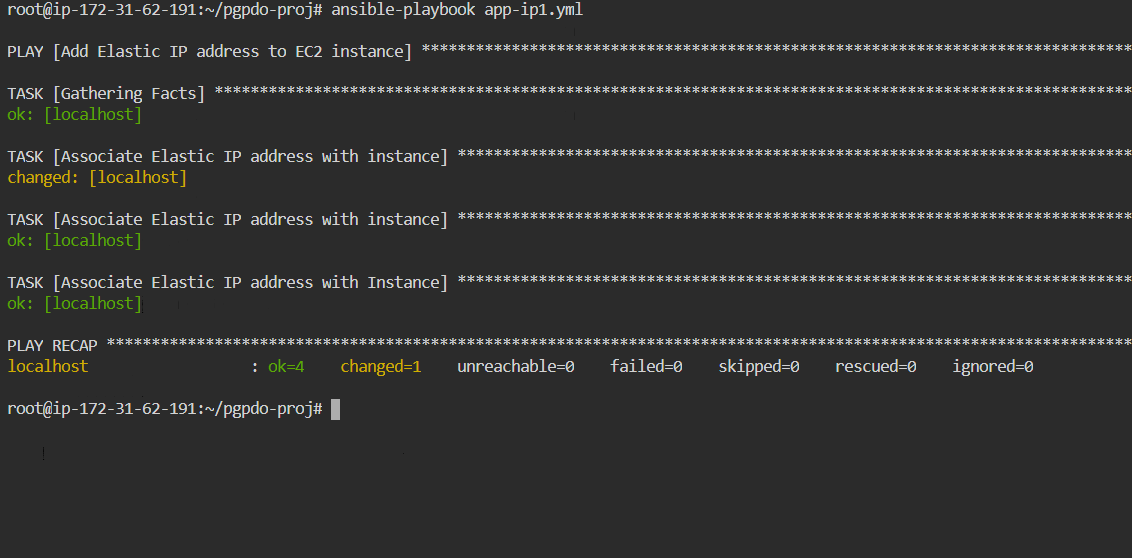


Once the Ansible playbook is run, the Instances are visible in AWS



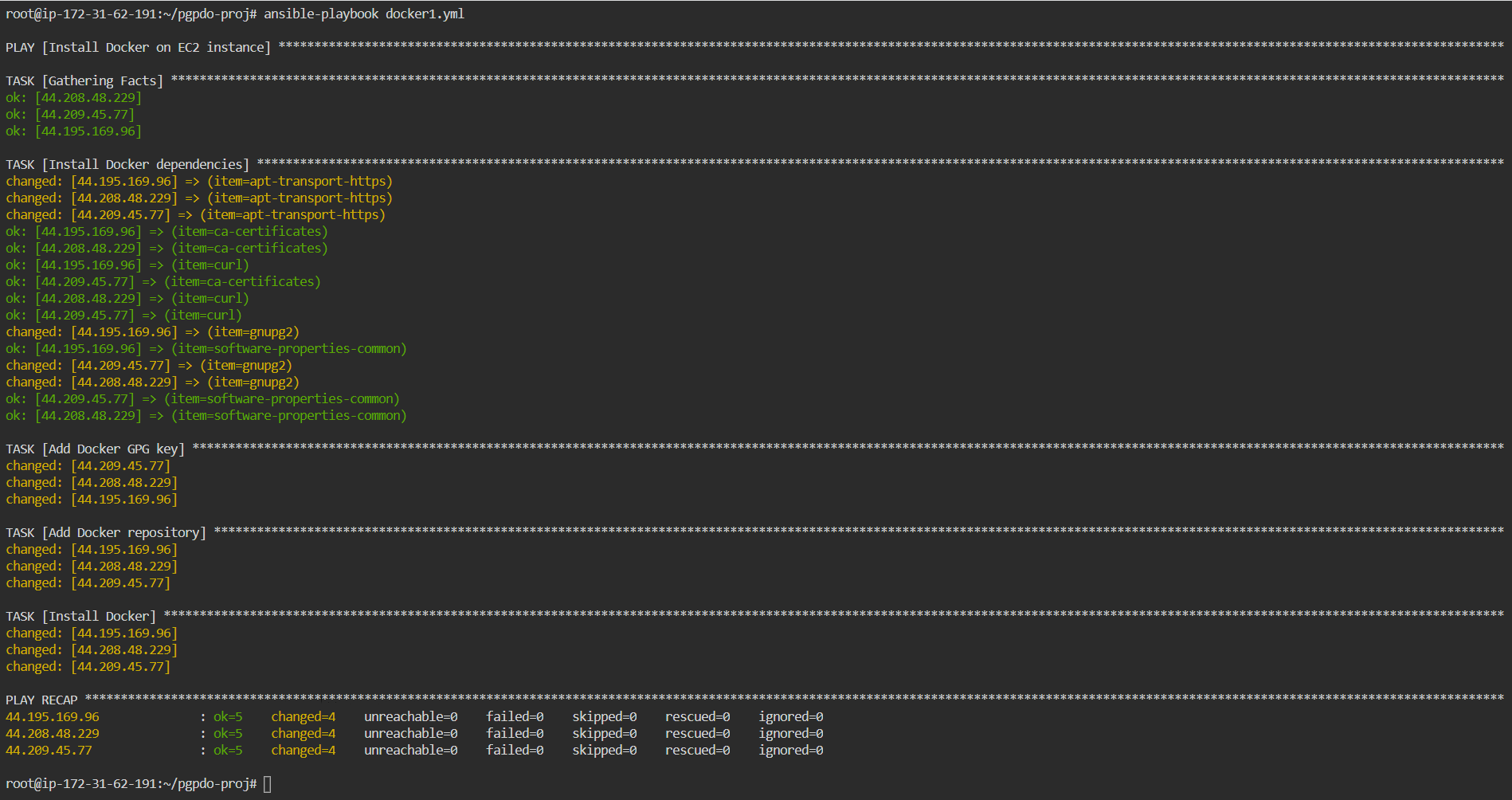
***Add Elastic IP address to EC2 instances using Ansible***

Using another playbook, the Elastic IP address added in first step are assiociated to the EC2 instances.

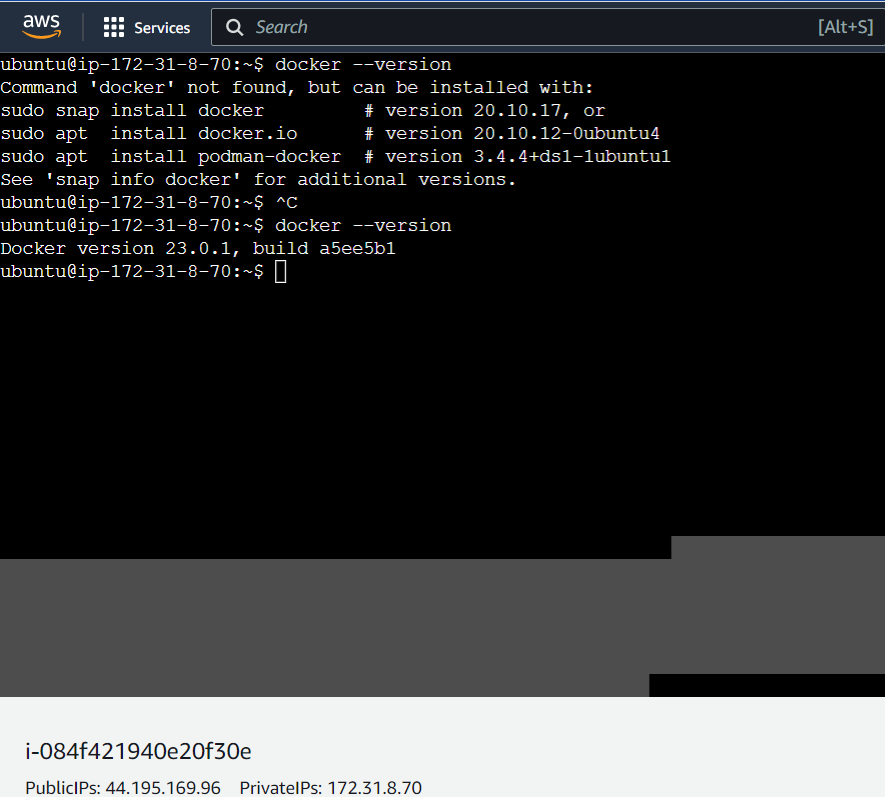


1. **Install Docker using Ansible**

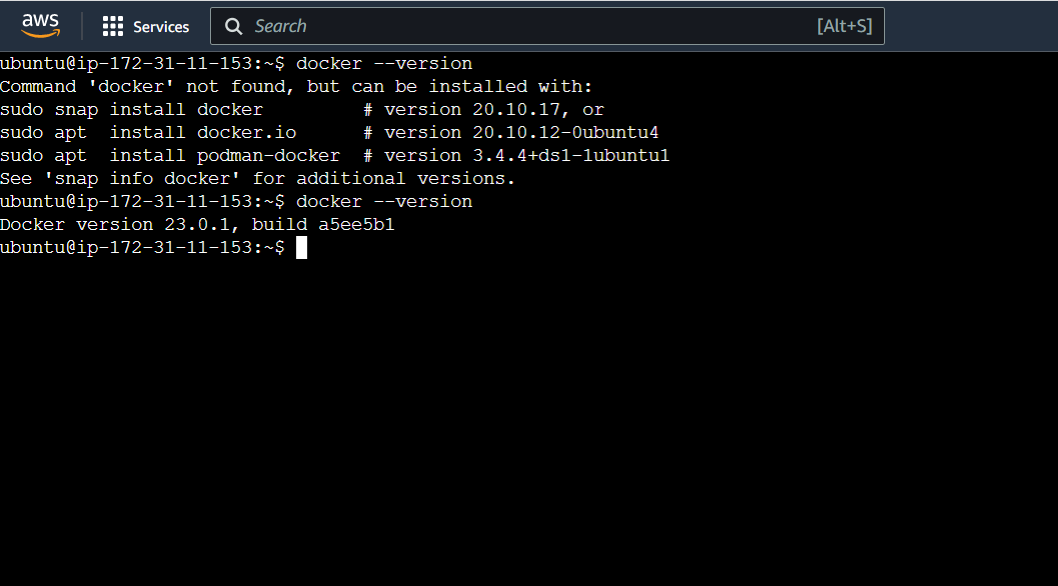
To work with Kubernetes, docker is required. Using Ansible playbook, the docker is installed in all three EC2 instances.



After installation, you can check the Docker version check in AWS EC2 instance

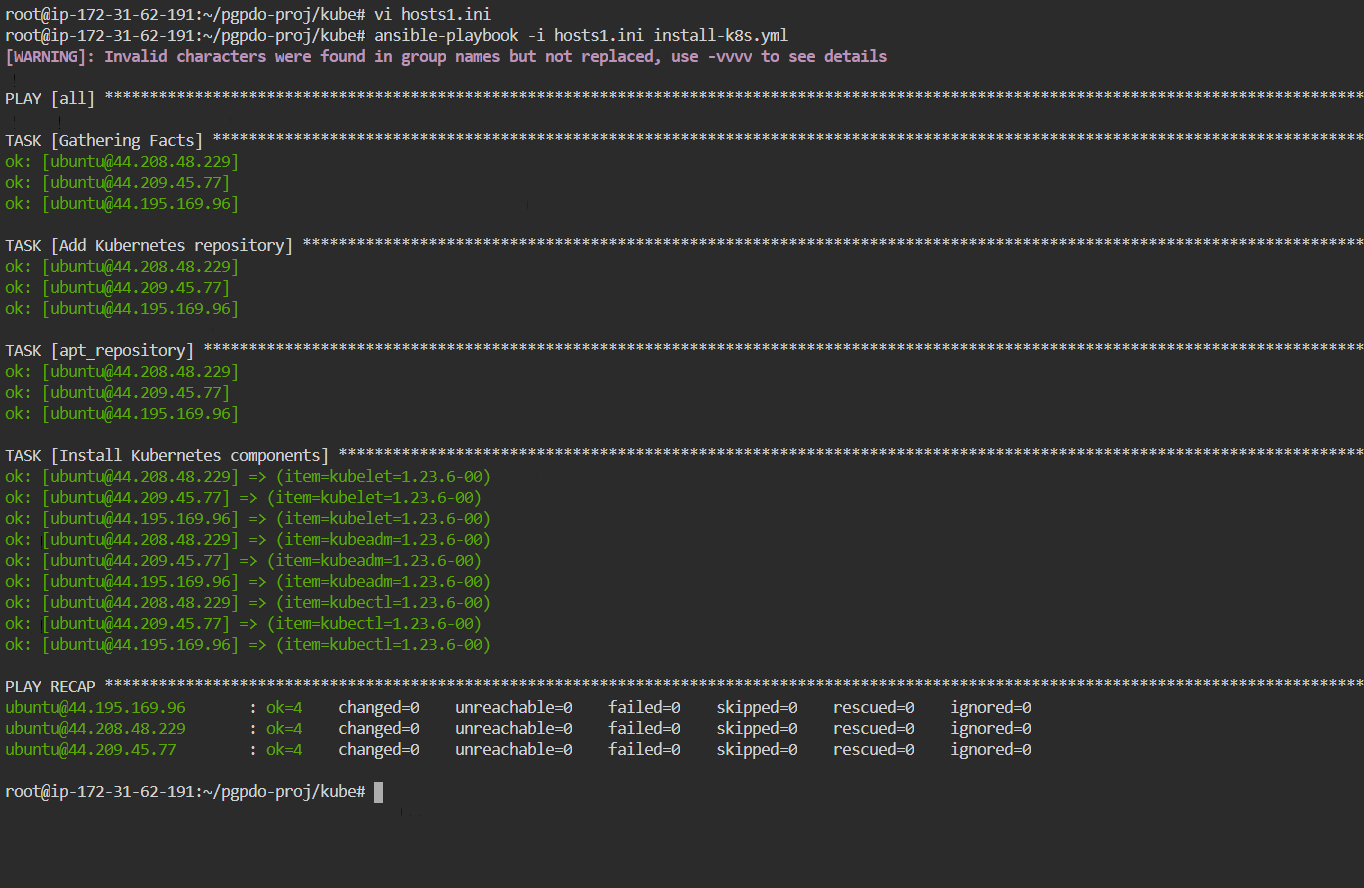


WorkerNode-1 

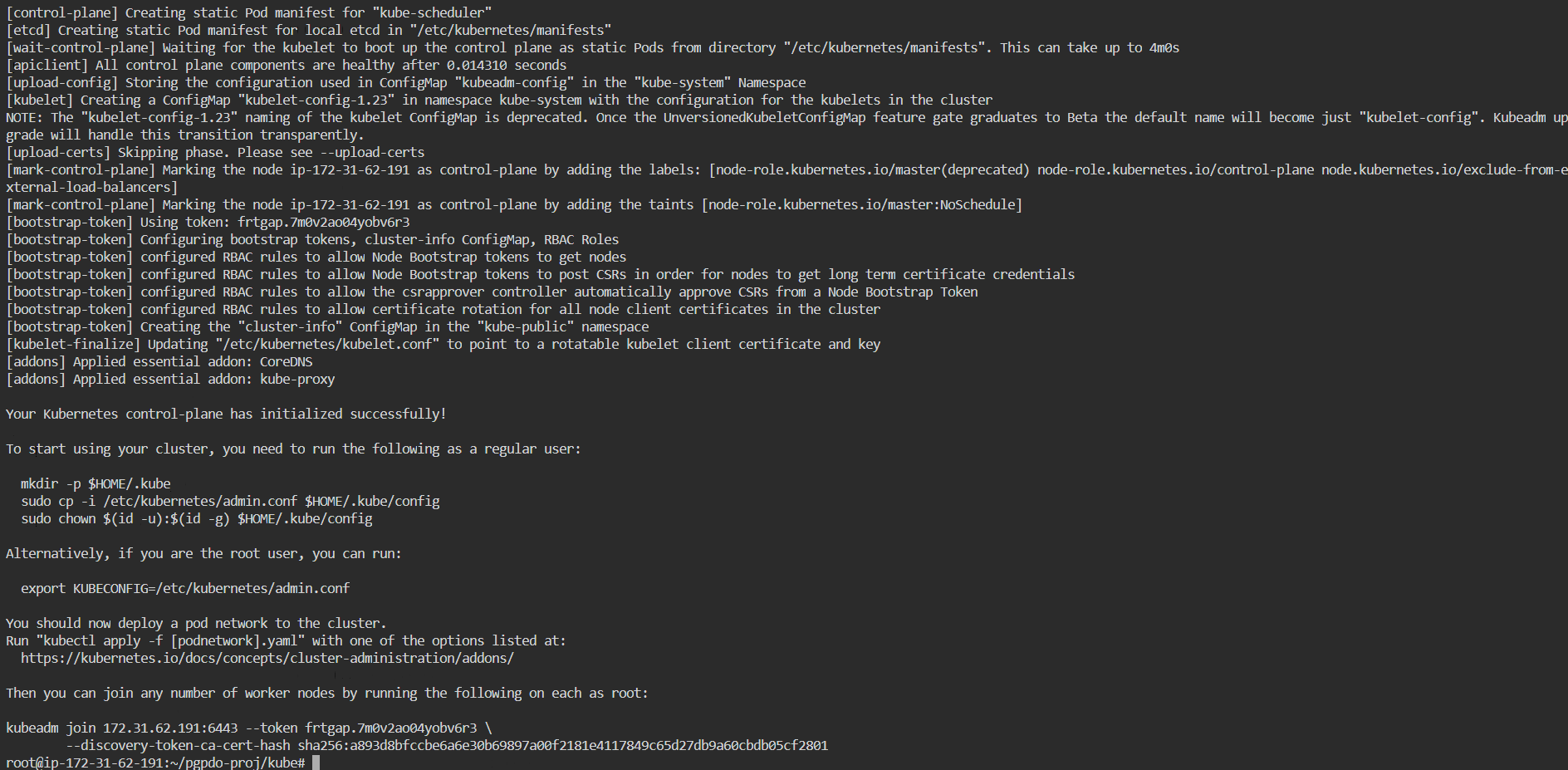
WorkerNode2

1. **Install Kubernetes**

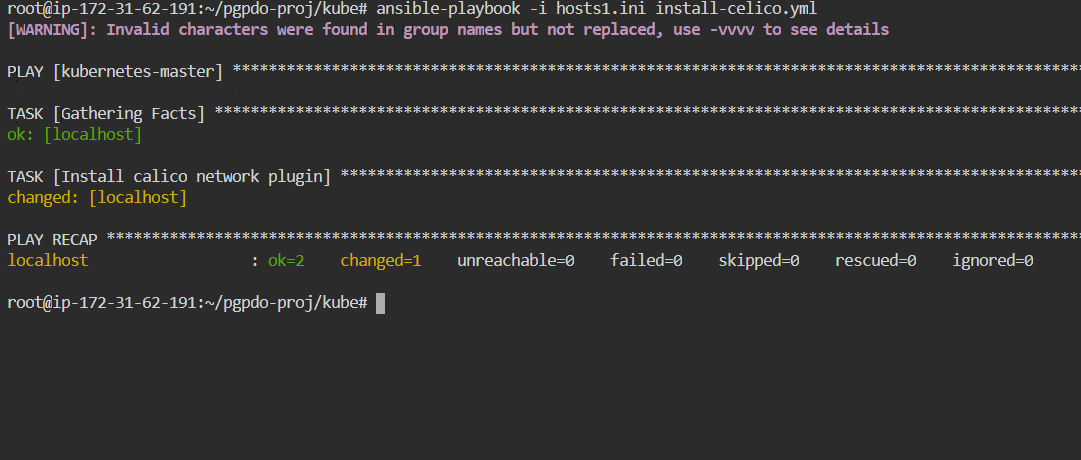
Kubernetes is installed in three nodes using Ansible Playbook



Kubeadm init in Master is done to initiate the Master node

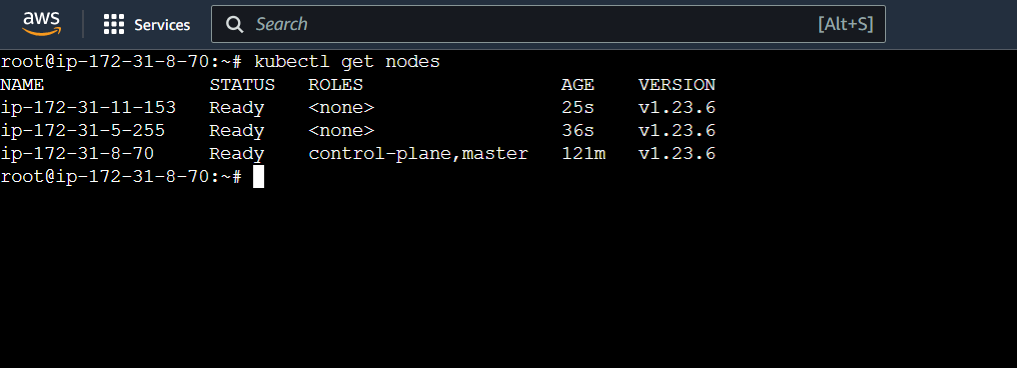


Install Celico driver to establish connection between worker nodes with master. This installation is done using Ansible



Joining Nodes to Master

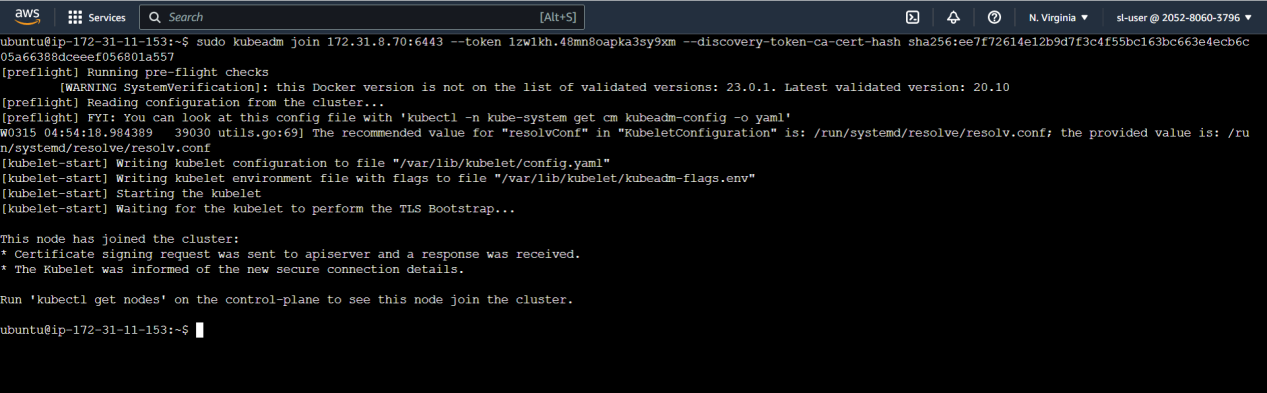
Worker nodes are joined with Master node using Kubeadm join command



Worker node 1



Worker node -2



1. **Configure Application in pod**

For this project, a spring boot application is being used and base application. This application is cloned from git and then an image is created out of it. As a part of build process, one unit test (HelloWorldConfigurationTests) is also executed which you can see in the below image

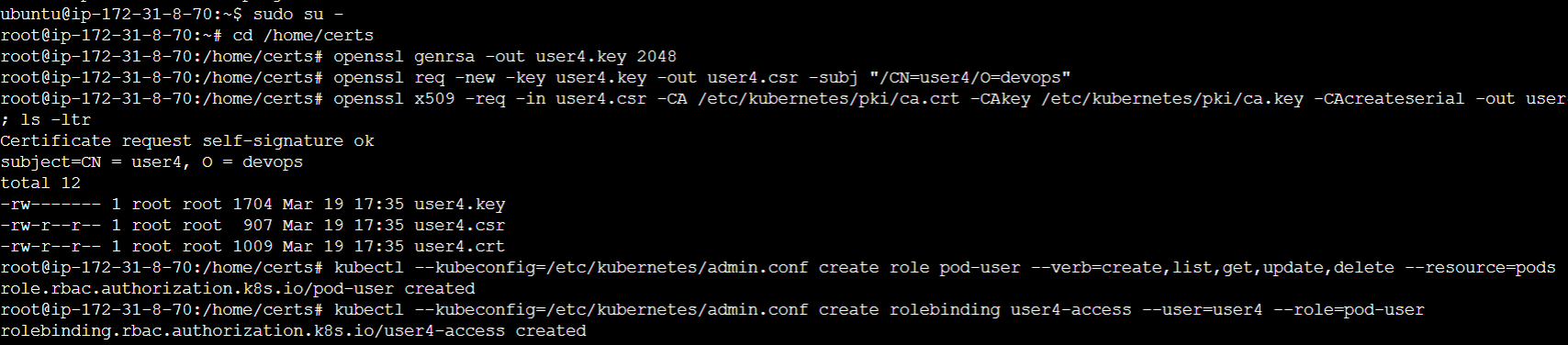


Docker image is created for the application once it is build successfully

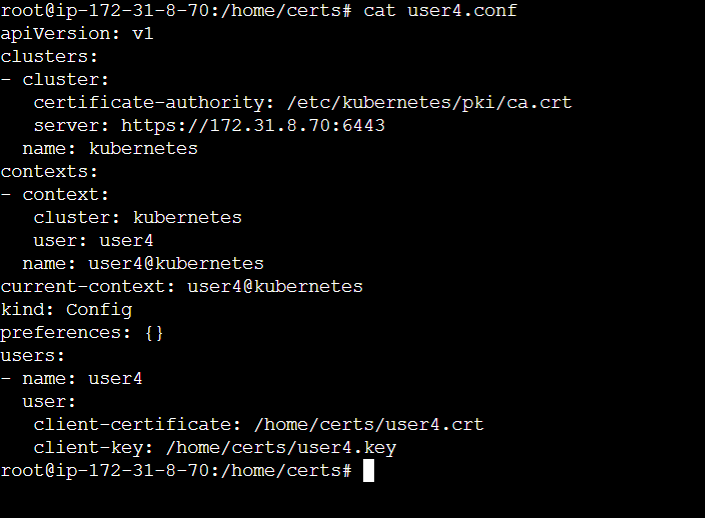


1. **Create a new user with permissions to create, list, get, update, and delete pods**

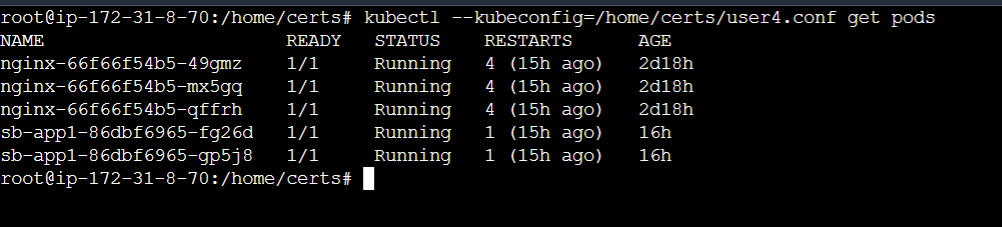
A new user user4 is created with permissions to create, list, get update and delete pods



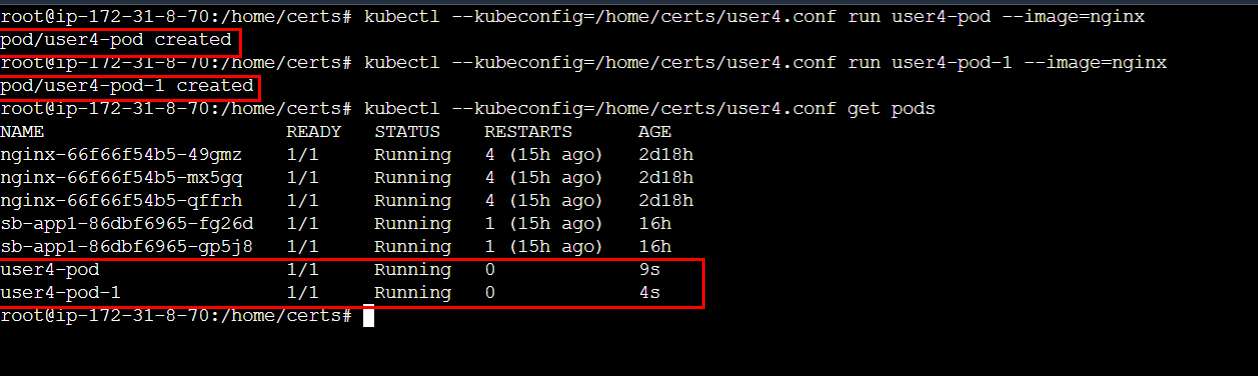
User4.conf file is created like admin.conf



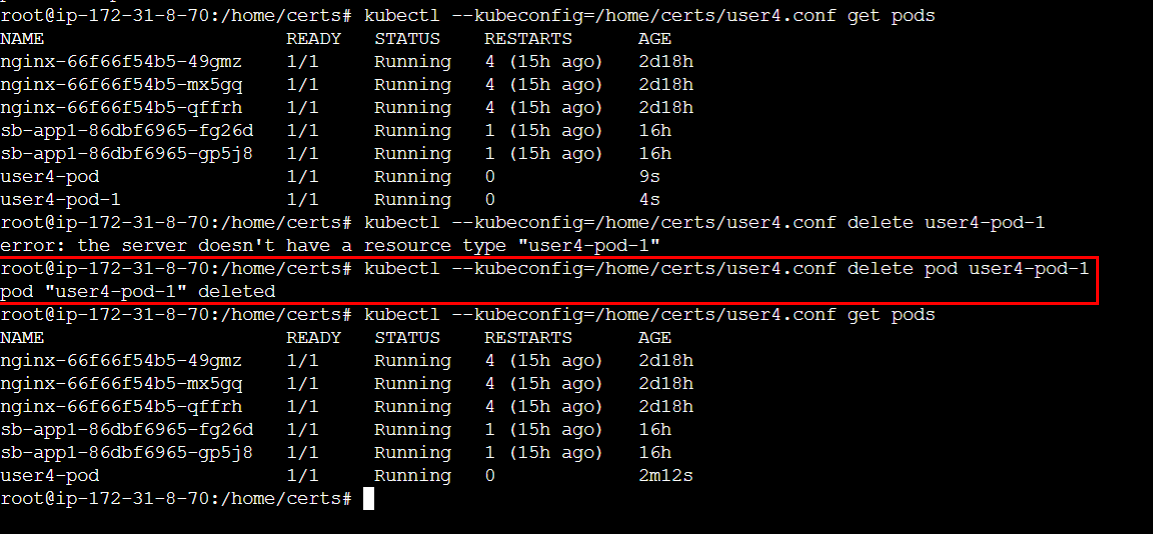
Using the User4.conf as config, pods can be access



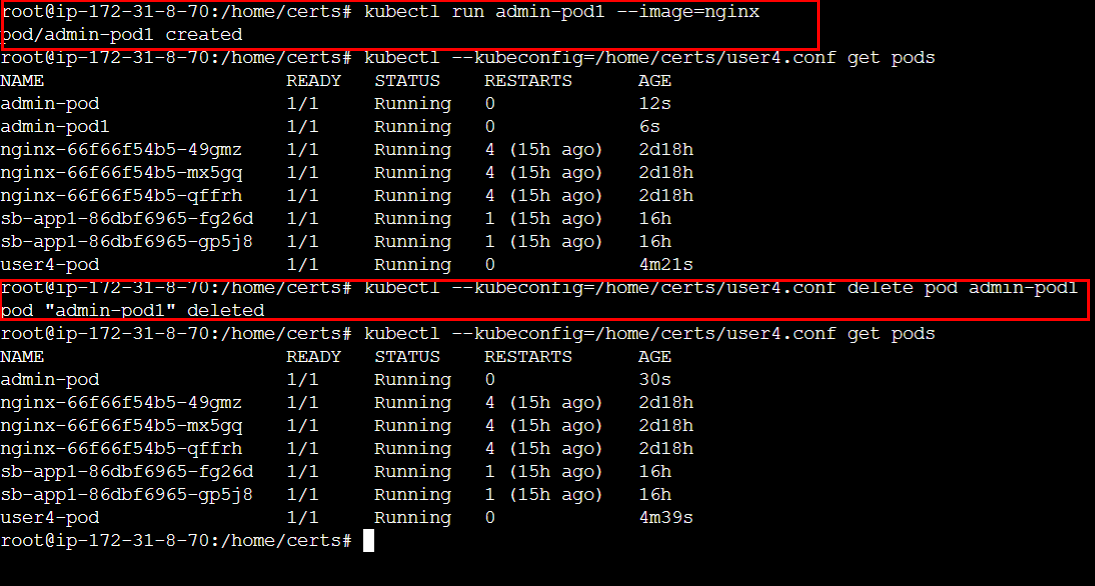
Using the user4.conf, pods are created



Using user4.conf as configuration, the pod is deleted



Not only the pods created using user4.conf but also other pods can be deleted using user4. Here is the example. A pod named admin-pod1 is created with normal user. And it is deleted using user4



With this we can conclude that a user created with create,list,get,update,delete permissions can do all actions to the pods using the the user.

1. **Take snapshot of ETCD database**

Install etcdctl using below commands in master node

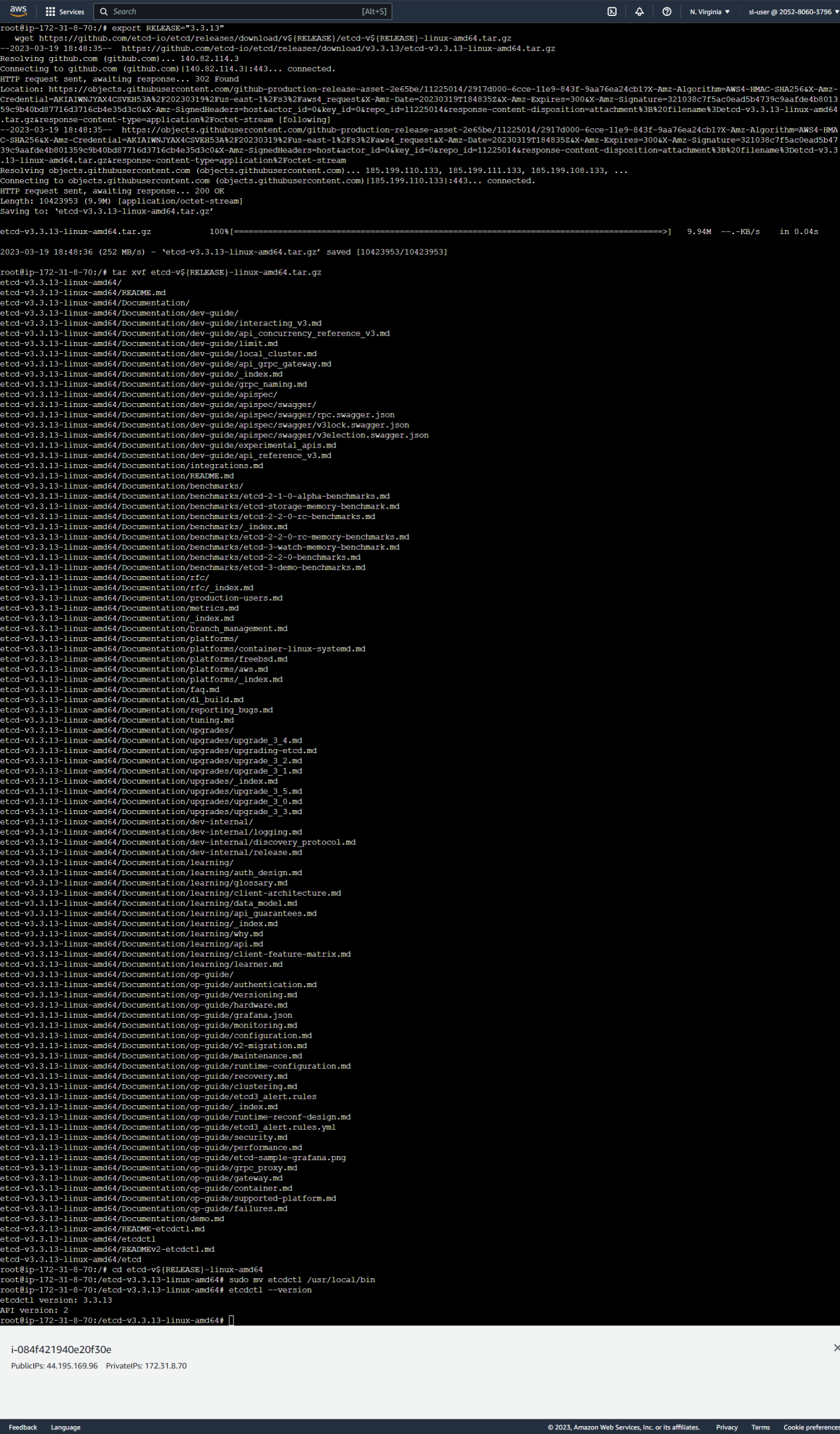
export RELEASE="3.3.13"

wget <https://github.com/etcd-io/etcd/releases/download/v${RELEASE}/etcd-v${RELEASE}-linux-amd64.tar.gz>

tar xvf etcd-v${RELEASE}-linux-amd64.tar.gz

cd etcd-v${RELEASE}-linux-amd64

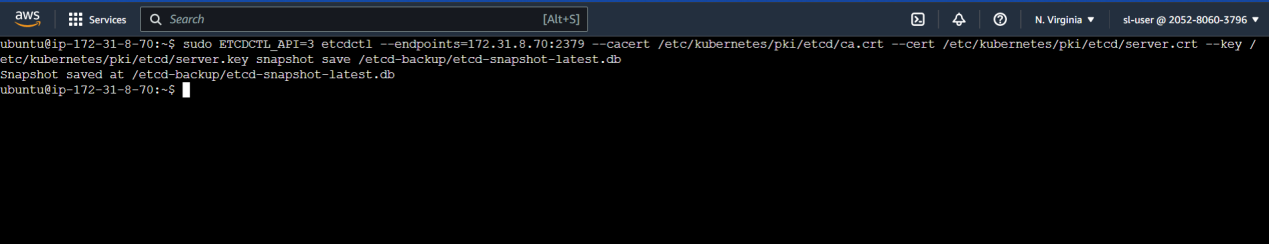
sudo mv etcdctl /usr/local/bin



Take ETCD backup using below command

ETCDCTL\_API=3 etcdctl --endpoints=172.31.8.70:2379 --cacert /etc/kubernetes/pki/etcd/ca.crt

--cert /etc/kubernetes/pki/etcd/server.crt --key /etc/kubernetes/pki/etcd/server.key snapshot save /etcd-backup/etcd-snapshot-latest.db

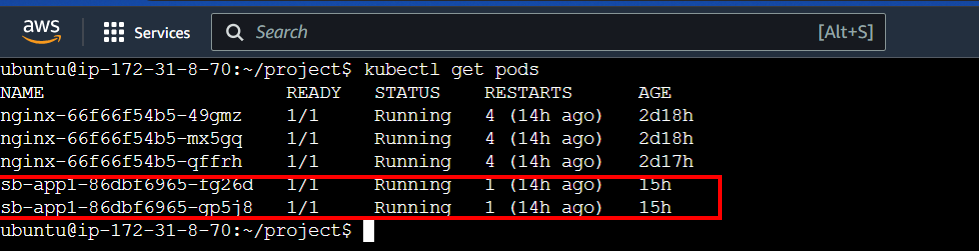


1. **Create a deployment, service and Horizontal auto scaling for the application**

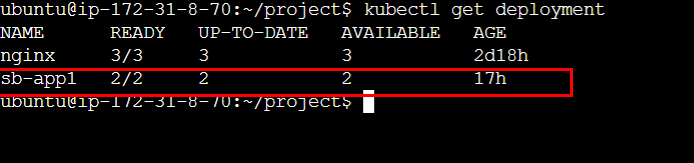
Once the docker image is built for the application, an ansible playbook is used to create a deployment, service and a Horizontal auto scaling for the application. This playbook will create a deployment which in turn creates the pods and distributes among the worker nodes with the application.



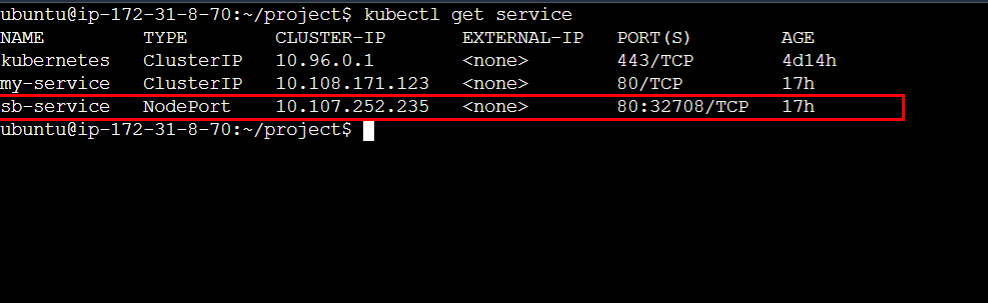
Pods created



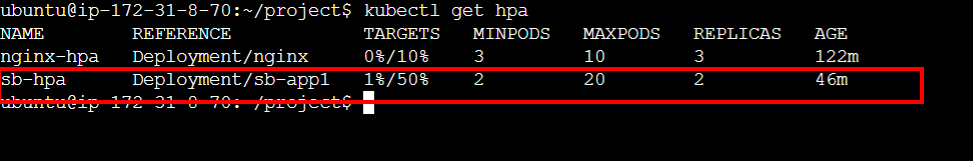
Deployment created



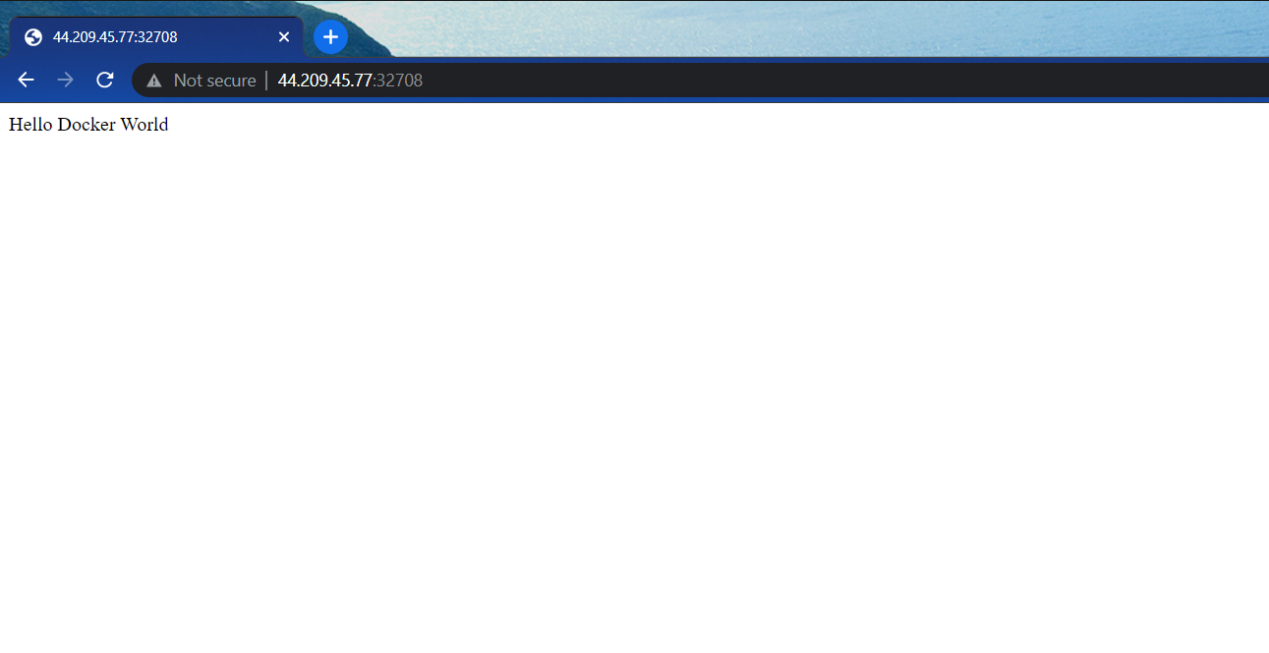
Service created



Horizontal autoscaling



Application accessed through internet

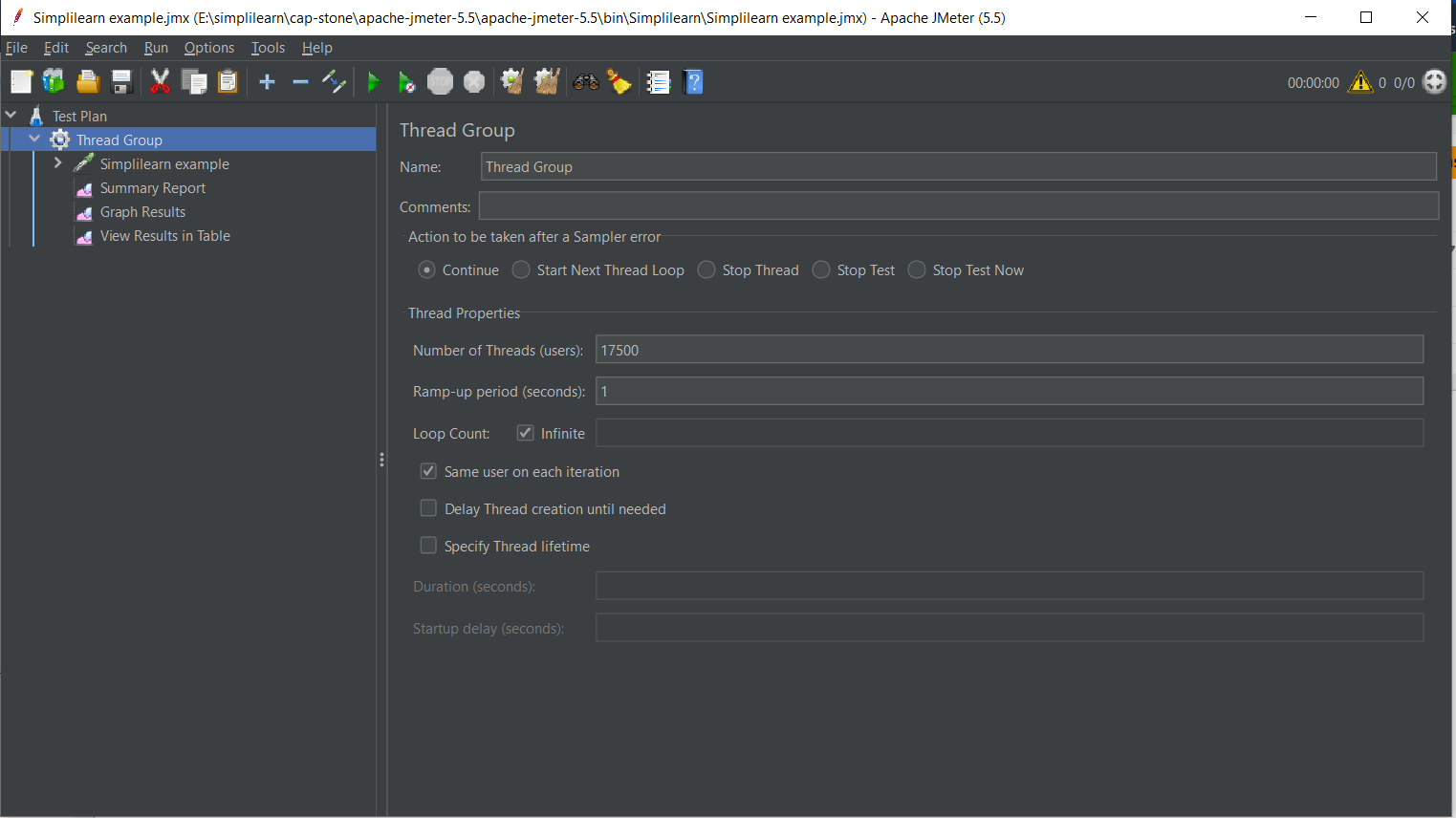


**Unique Selling Point: Autoscaling the application on cpu usage of more than 50%**

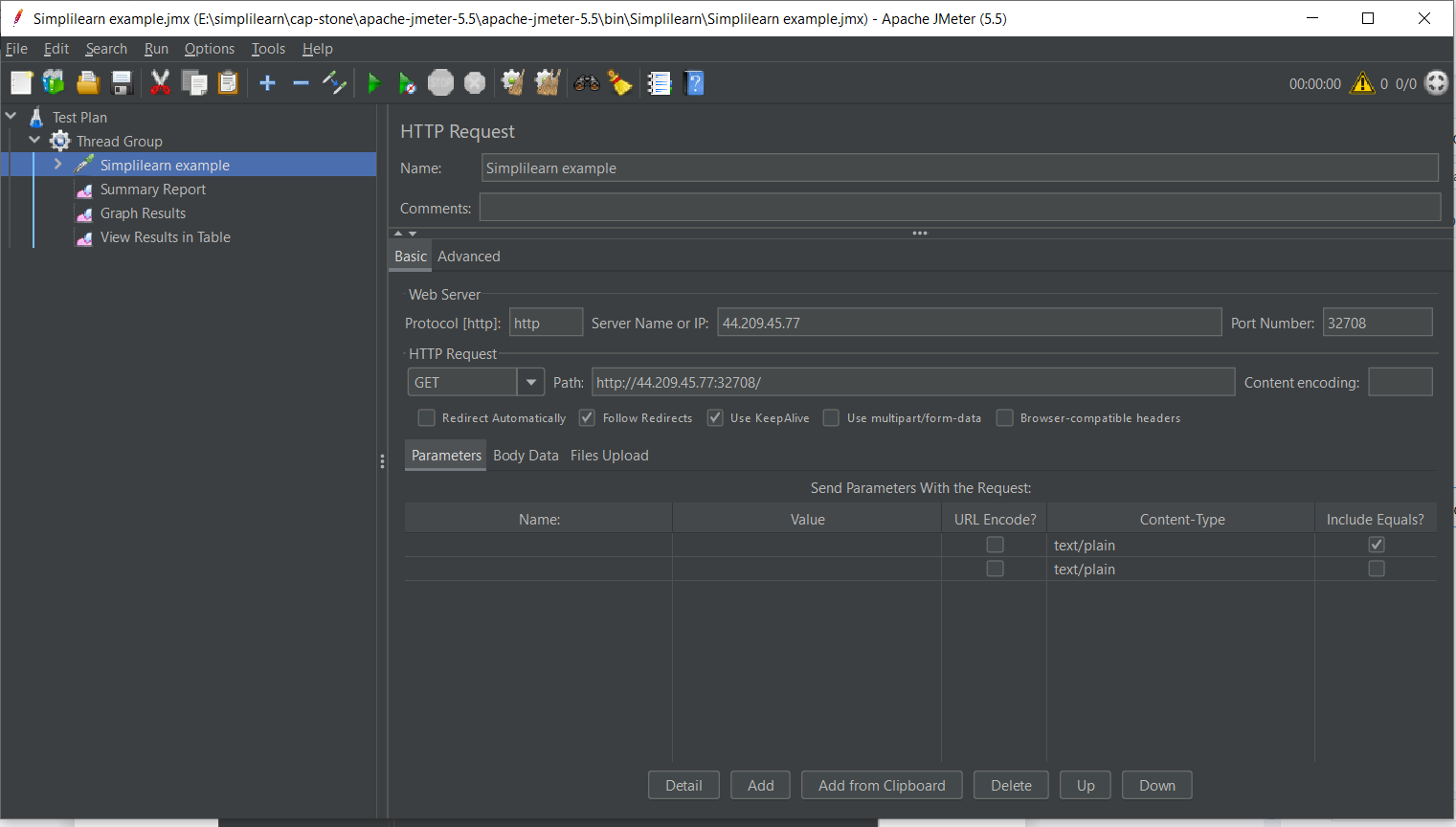
This project is designed in such a way that if the application is overdriven by the request from the users,the application autoscaling mechanism increases the number of pods available which allows to balance the load on the application. For that purpose we have set up the cpu usage at limit of 50%. Once the limit exceeds 50%, then the autoscaling recognizes the load and starts adding more pods to compensate the load.

To prove the autoscaling we will be using Jmeter as testing tool to introduce the load on the application.A Jmeter test has been created to call one of the node with a load of more than 15000 users. This triggers the autoscaling and it increases the number of pods. Below screenshots demonstrates the same

**Jmeter test script - thread count**

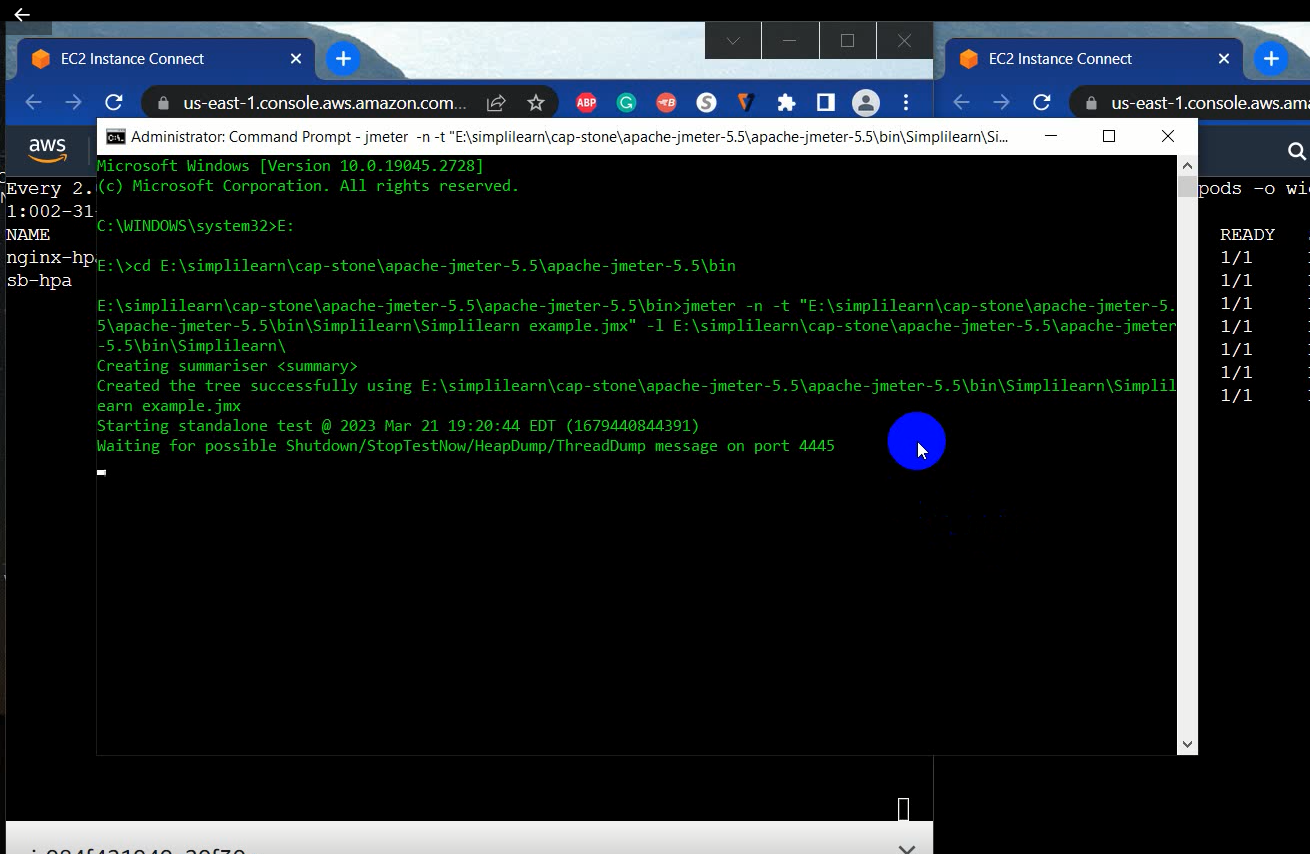


**Test**

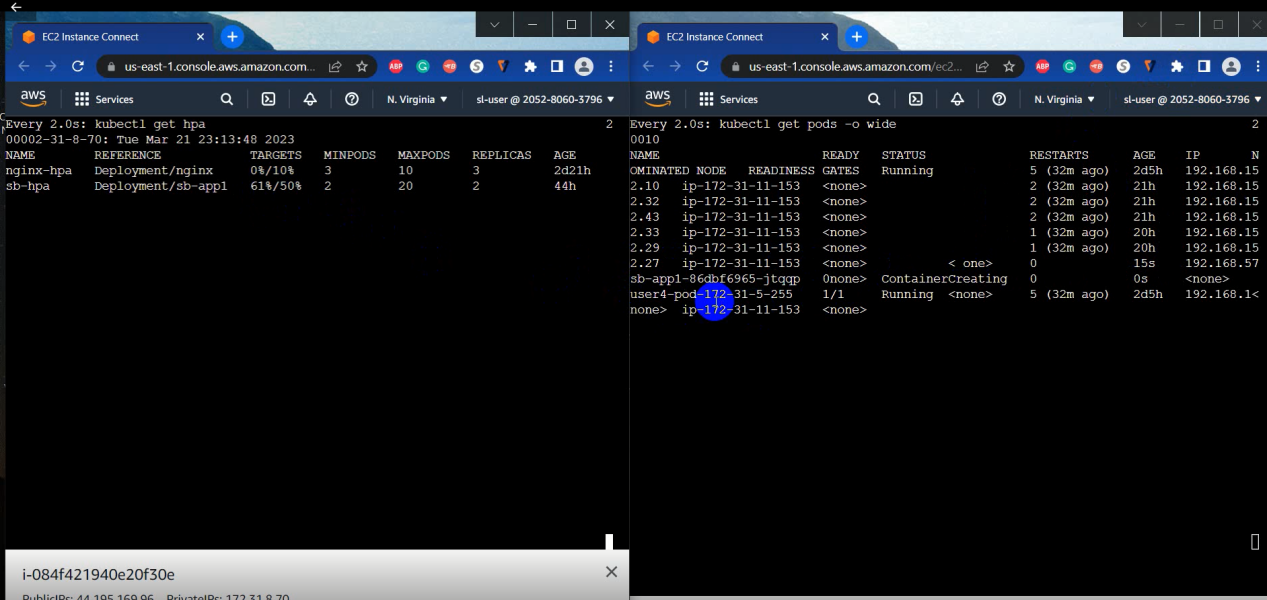


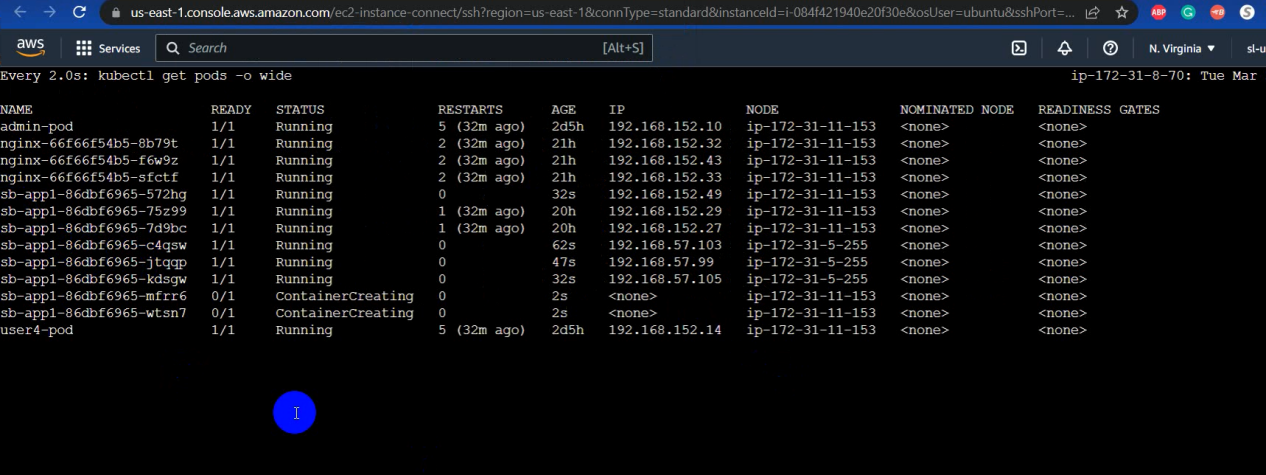
Before starting the load, the hpa and pods. Hpa is at 2% load and the pods count is 2. Once the load test starts, the load is increased and pods count will be increased.

Jmeter test starts



Load starts increasing which increases the pod count





From the above images you can see that open the increase in the load the pods count has been increased.

This proves the Unique selling point of this project if the memory of CPU goes beyond 50%, environments automatically get scaled up and configured.

**Conclusion:**

This project demonstrates the ability for an application that when a load is increased on a kubernetes pod, then the horizontal auto scaling from kubernetes analyses it and automatically scales up the pods and configures the pods. This situation compels the company that this infrastructure that runs in high-availability mode makes the app more reliable, fast, and secure for improving the performance of the current system.