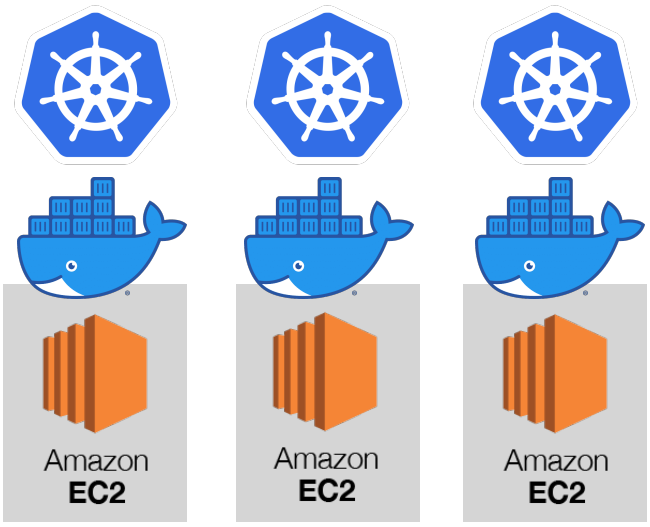
Infrastructure Optimisation

Capstone project

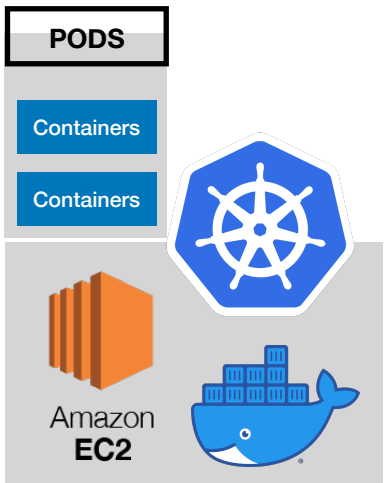
By Manan Patel

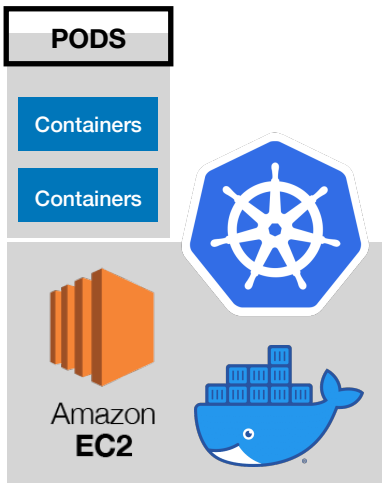


MASTER NODE

WORKER NODE 1

WORKER NODE 2





Introduction

Background/Problem statement:
A popular payment application, **EasyPay** where users add money to their wallet accounts, faces an issue in its payment success rate.
The timeout that occurs with the connectivity of the database has been the reason for the issue. While troubleshooting, it is found that the database server has several downtime instances at irregular intervals. This situation compels the company to create their own infrastructure that runs in high-availability mode.
Given that online shopping experiences continue to evolve as per customer expectations, the developers are driven to make their app more reliable, fast, and secure for improving the performance of the current system.

Project:

We will implement a Cloud Based infrastructure which is fully containerised (docker) and Orchestrated through Kubernetes. The process of setting up Cloud infrastructure - Provisioning & configuration will be automated. The docker will be our container runtime engine and Kubernetes will handle our pods through deployments which will be autoscaled. The Auto scaling will be proportional to the load generated on the application.

Once the Application is deployed, we will conduct a performance test through Blazemeter to evaluate the application/deployment and observe how it behaves under load. If the application pass the load test, we will know that the deployment is capable of handling the load influx. As a result of that we will be able to conclude that the application is reliable, fast, and secure.

We will be including the below mentioned components in our setup:

- AWS (or any cloud provider)
- Git
- Terraform
- Ansible
- Docker
- Kubernetes

Sr	INDEX	Page
[1]	- Prerequisites	4
[2]	- Configurations	5
[3]	- Terraform script	7
[4]	- Ansible Playbook	10
[5]	- Kubernetes manifests	14
[6]	- System provisioning and configuration	20
[7]	- Deploying the pods, services, autoscalling, RBAC & ETCD backup	23
[8]	- Creating and configuring Application Load Balancer in AWS	30
[9]	- Test case- Load testing our application	33
[10]	- Conclusion & concepts used in the project	37

Git repository: https://github.com/ImMnan/Capstone-K8S-Infra.git

[1] Pre-requisites

We will need to make sure that we are equipped with the right components before working on the project.

[1.1] Basic requirements

a.. Linux x86_64 system - Any linux system, however, we will demonstrate on Ubuntu.
b.. AWS account - Any cloud service account however, we will demonstrate the steps as per AWS and writing the code focused on AWS platform.

[1.2] Below are the packages that we want installed into our Linux system, check their availability using these commands:

• Git

```
$ git –version
```

• Python

```
$ python3 –version
```

• Terraform

```
$ terraform –version
```

• Ansible

```
$ ansible –version
```

```
[ec2-user@ip-172-31-83-227 ~]$ terraform --version
Terraform v1.3.7
on linux_amd64
[ec2-user@ip-172-31-83-227 ~]$ ansible --version
[DEPRECATION WARNING]: Ansible will require Python 3.8 or newer on the controller starting with Ansible 2.12. Current
22:44:31) [GCC 7.3.1 20180712 (Red Hat 7.3.1-15)]. This feature will be removed from ansible-core in version 2.12. Dep
setting deprecation_warnings=False in ansible.cfg.
ansible [core 2.11.12]
  config file = None
  configured module search path = ['/home/ec2-user/.ansible/plugins/modules', '/usr/share/ansible/plugins/modules']
  ansible python module location = /home/ec2-user/.local/lib/python3.7/site-packages/ansible
  ansible collection location = /home/ec2-user/.ansible/collections:/usr/share/ansible/collections
  executable location = /home/ec2-user/.local/bin/ansible
  python version = 3.7.15 (default, Oct 31 2022, 22:44:31) [GCC 7.3.1 20180712 (Red Hat 7.3.1-15)]
  jinja version = 3.1.2
  libyaml = True
[ec2-user@ip-172-31-83-227 ~]$ git --version
git version 2.38.1
```

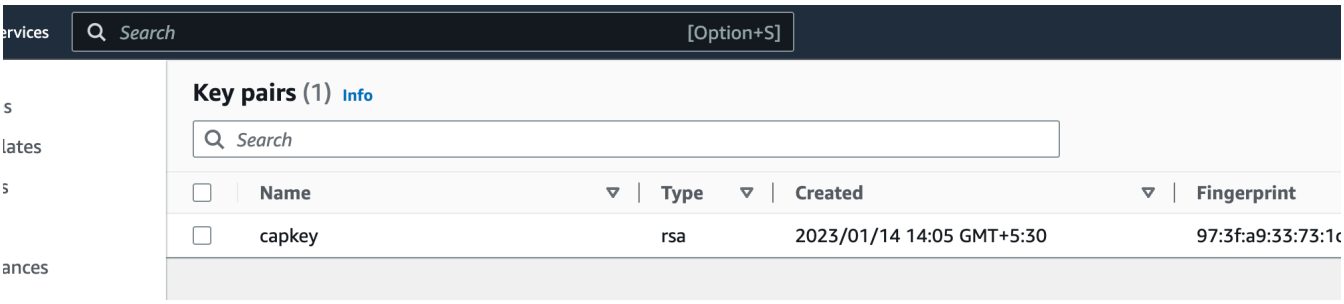
[2] Configurations

[2.1] Git configuration

1. Setup GitHub account, sign in if already a user, else create a new account using signup section.
2. Create a repository to manage our code - Terraform, Ansible-playbooks, Kubernetes yaml files etc. Navigate to repository > Click on New > Name the repository and create it. (I am naming it 'Capstone-K8S-Infra')
3. Copy the https code by clicking on the clone button.
4. Clone the repository on your system using the git clone command.
$ git clone [paste the copied code here]
5. This will create a directory for this specific repository.
6. $ cd [directory name] and move to the directory created by cloning the repository. (i.e. Capstone-K8S-Infra)

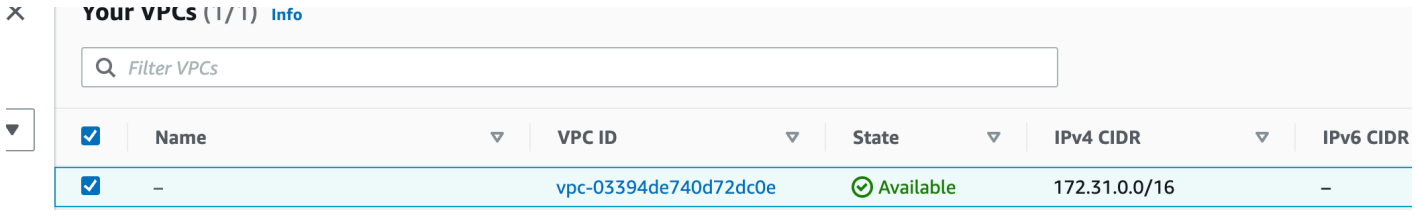
[2.2] AWS keys

1. Login to AWS account, or create one if not available.
2. Navigate to EC2 section > under the Network & Security select Key pairs, or key pairs section displayed on the EC2 Dashboard
3. This will open key pair section, all the created keys can be found here.
4. Click on create key pair to create a new key pair (see the figure below)
5. Name the key pair, select .pem format and RSA as key type. (Setting wpkey as the key name here)
6. Click on create key pair to create it and the system will automatically download the key pair.
7. Change the permission of the key using $ chmod 400 [key-path]
8. Move the key to the Capstone-project directory for further use.



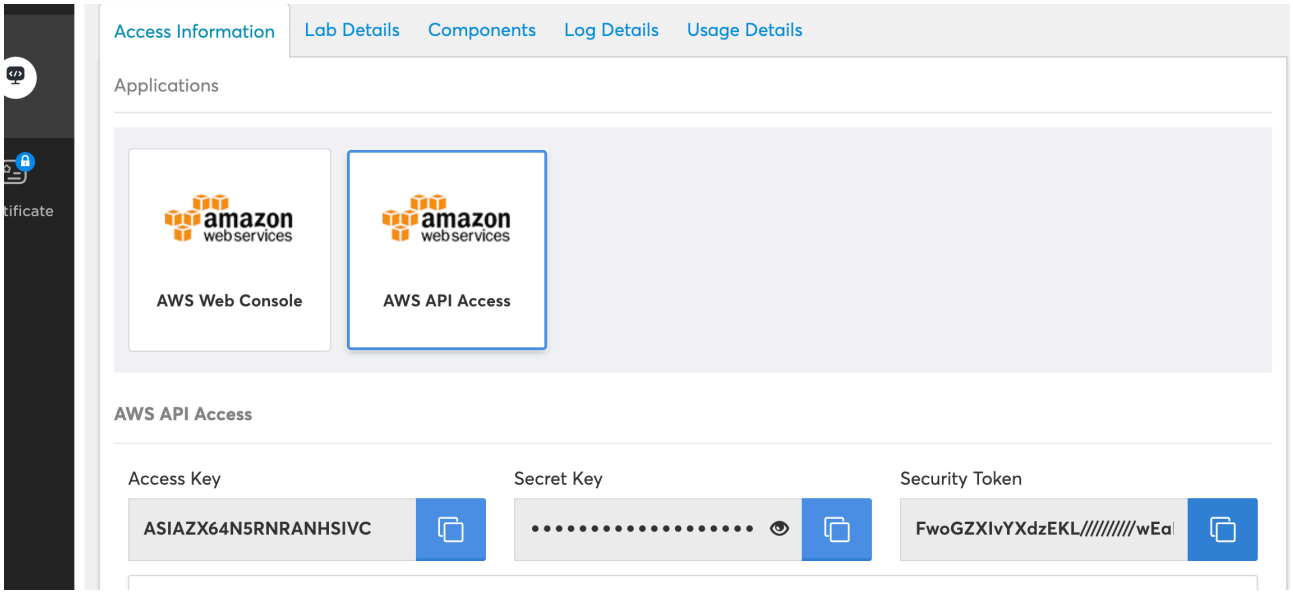
[2.3] VPC ID (AWS)

1. Login to AWS account,
2. Navigate to VPC section > click on VPCs section on the VPC dashboard to open the VPC panel.
3. Simply copy the VPC ID from here.



[2.4] AWS API credentials

1. GO to AWS IAM - Identity and access management
2. Get the keys
3. Create the user, provide admin privileges and copy the keys.
4. You will need all three information:
    a. Access key
    b. Secret key
    c. Token



[2.5] AMI id (AWS)

1. Login to AWS account
2. Navigate to EC2 section > under the images section > click on AMI catalog
3. Here, we will see the available instances and their AMI ids
4. Copy the AMI ID for the choice of image. (We will use the AMI for Ubuntu)

[3] Terraform script

As the title suggests, we will first need a terraform script to provision resources on our AWS cloud.
Navigate to the cloned repository (see [2.1])

```
$ cd Capstone-K8S-Infra
```

Now, let's start with writing a terraform main.tf script

```
$ nano main.tf
```

```
/* Start with adding the local variables, which will be used
throughout the script ami-id of the instance */

locals {
  ami_id = "ami-09e67e426f25ce0d7"
  vpc_id = "vpc-"
  ssh_user = "ubuntu"
  key_name = "capkey"     # the name should match with the name on
cloud
  instance_count = 3
  private_key_path = "/home/ubuntu/Capstone-K8S-Infra/capkey.pem"
}

/* Declare the provider and other required information linked with
it, access key, secret key and token as per AWS
(Any cloud provider you are using) */

provider "aws" {
  region     = "us-east-1"
  access_key = "key"
  secret_key = "key"
  token = "token"
}

/* [4.1] Creating a security group with the name of k8saccess and
setting ingress egress security rules, it will automatically use
the vac id from variables declared in local. We are being open
here, as we want to keep the network open for Kubernetes cluster,
we can narrow it down to specific ports later on */

resource "aws_security_group" "k8saccess" {
  name   = "cap_access"
  vpc_id = local.vpc_id

  ingress {
    from_port   = 22
    to_port     = 22
    protocol    = "tcp"
    cidr_blocks = ["0.0.0.0/0"]
  }
```

```
  ingress {
    from_port   = 0
    to_port     = 0
    protocol    = "-1"
    cidr_blocks = ["0.0.0.0/0"]
  }

  egress {
    from_port   = 0
    to_port     = 0
    protocol    = "-1"
    cidr_blocks = ["0.0.0.0/0"]
  }
}
# Resource creation using local variables - ami, security group
and key.

resource "aws_instance" "web" {
  count = local.instance_count
  ami = local.ami_id
  instance_type = "t2.medium"
  associate_public_ip_address = "true"
  vpc_security_group_ids =[aws_security_group.k8saccess.id]
  key_name = local.key_name

  tags = {
    Name = "cap-ec2"
  }

  /* Setting up connection as we want to use ssh for Ansible
configurations to run. Again using local variables for host ip,
user name, and security key */

  connection {
    type = "ssh"
    host = self.public_ip
    user = local.ssh_user
    private_key = file(local.private_key_path)
    timeout = "4m"
  }

# Just to confirm whether our remote access is working

  provisioner "remote-exec" {
    inline = [
      "hostname"
    ]
  }

/* copying the remote machine ip to our local machine into my
hosts file using local-exec */
```

```
  provisioner "local-exec" {
    command = "echo ${self.public_ip} >> myhosts"
  }
}
```

Created main.tf
Make the changes to this file as per your requirements, each segment has been marked with comments section to provide more information.

Start with replacing the local variables, which will be used throughout the script ami-id of the instance (see [2.5]), use VPS (see [2.3]), key_name (see [2.2]) and key_path (see [2.2])

Declare the provider and other required information linked with it, access key, secret key and token as per AWS (Or any cloud provider you are using) (See [2.4])

[4] Ansible Playbook

[4.1]

The main playbook is prime.yml, it will make sure all the requirements are met for kubernetes and also install kubernetes on our machines.

```
$ mkdir Ansible
$ cd Ansible

$ nano prime.yml

---

- hosts: all
  become: true

  tasks:

  - name: "Initiating Docker installation first as per the OS..."
    import_tasks: docker-install.yml

  - name: "Installing Kubernetes into the instances"
    import_tasks: k8s-install.yaml

- hosts: master
  become: true
  tasks:
  - name: "Installing Kubctl on master node"
    import_tasks: k8s-master.yml
```

Prime.yml calls different yaml files, which we will see one by one.

[4.2] docker-install.yml

This file will be called/imported first by the prime.yml [see 4.1] to install docker and it's requirements to our target system.

```
$ nano docker-install.yml

---

- name: Install aptitude!
  apt:
    name: aptitude
    state: latest
    update_cache: true
```

```
- name: Install required system packages
  apt:
    pkg:
      - apt-transport-https
      - ca-certificates
      - curl
      - software-properties-common
      - python3-pip
      - virtualenv
      - python3-setuptools
    state: latest
    update_cache: true

- name: Add Docker GPG apt Key
  apt_key:
    url: https://download.docker.com/linux/ubuntu/gpg
    state: present

- name: Add Docker Repository
  apt_repository:
    repo: deb https://download.docker.com/linux/ubuntu focal
stable
    state: present

- name: Update apt and install docker-ce
  apt:
    name: docker-ce
    state: latest
    update_cache: true

- name: Install Docker Module for Python
  pip:
    name: docker
```

[4.3] k8s-install.yaml

This yaml configuration file will be called in second stage by prime.yml [see 4.1] to install kubernetes components and configure their requirements into our target system.

```
$ nano k8s-install.yaml
```

```
---

- name: Make the Swap inactive
  command: swapoff -a

- name: install APT Transport HTTPS
  apt:
```

```
    name: apt-transport-https
    state: present

- name: add Kubernetes apt-key for APT repository
  apt_key:
    url: https://packages.cloud.google.com/apt/doc/apt-key.gpg
    state: present

- name: add Kubernetes APT repository
  apt_repository:
    repo: deb http://apt.kubernetes.io/ kubernetes-xenial main
    state: present
    filename: 'kubernetes'

- name: install kubelet
  apt:
    name: kubelet
    state: present
    update_cache: true

- name: install kubeadm
  apt:
    name: kubeadm
    state: present
- name: Creating a daemon json
  copy:
    dest: "/etc/docker/daemon1.json"
    content: |
     {
     "exec-opts": ["native.cgroupdriver=systemd"]
     }
- name: restarting the docker
  ansible.builtin.shell: |
    systemctl enable docker
    systemctl daemon-reload
    systemctl restart docker
```

[4.4] k8s-install.yaml

This yaml file will instal kubectl info the master node. It is called by the prime.yaml in the final stage. (Note:docker, kubernetes and other requirements are already satisfied because the previous yml configurations have run on all hosts, including the master [see 4.2 & 4.3])

```
$ nano k8s-master.yml
```

```
---

- name: install kubectl
  apt:
    name: kubectl
    state: present
    force: yes
```

Once the Ansible-playbook has been run successfully, we will have all our target systems ready with docker, Kubernetes and their dependencies. We will just need to initiate the Kubernetes cluster on these machines.

$ cd ..

To navigate back to the Capstone project directory.

[5] Kubernetes manifests

Now, we will need to construct yaml files based on our Kubernetes deployments, services, RBAC and Autoscalling.

[5.1] Application deployment (deployment & service,)

Create a directory appk8s, we will create the application deployment yaml here.

```
$ mkdir appk8s
$ cd appk8s
```

[5.1.1] Creating application deployment & service

```
$ nano front-end.yaml
```

```
apiVersion: v1
kind: Service
metadata:
  name: frontend
  namespace: appk8s
  labels:
    app: kubesample
    tier: frontend
spec:
  type: NodePort
  ports:
  - port: 80
  selector:
    app: kubesample
    tier: frontend

---
apiVersion: apps/v1
kind: Deployment
metadata:
  name: frontend
  namespace: appk8s
spec:
  selector:
    matchLabels:
      app: kubesample
      tier: frontend
  replicas: 2
  template:
    metadata:
      labels:
        app: kubesample
        tier: frontend
```

```
    spec:
      containers:
      - name: php-redis
        image: gcr.io/google-samples/gb-frontend:v4
        resources:
          requests:
            cpu: 200m
            memory: 400Mi
        env:
        - name: GET_HOSTS_FROM
          value: dns
        ports:
        - containerPort: 80
```

The frontend yaml will create a deployment with the Front end application connected with a service on top of it. The service will be nodeport, which will be connected with our Target group to be used under Application Load balancer setup in our AWS cloud.

[5.1.2] Creating database/backend deployment & service

```
$ nano redis-master.yaml
```

```
apiVersion: v1
kind: Service
metadata:
  name: redis
  namespace: appk8s
  labels:
    app: hello
    tier: backend
    role: master
spec:
  ports:
  - port: 6379
    targetPort: 80
  selector:
    app: redis
    tier: backend
    role: master
---
apiVersion: apps/v1
kind: Deployment
metadata:
  name: redis-master
  namespace: appk8s
spec:
  selector:
    matchLabels:
      app: redis
      role: master
```

```
      tier: backend
  replicas: 1
  template:
    metadata:
      labels:
        app: redis
        role: master
        tier: backend
    spec:
      containers:
      - name: master
        image: redis
        resources:
          requests:
            cpu: 100m
            memory: 100Mi
        ports:
        - containerPort: 6379
```

This Yaml will create a Redis master deployment, which is connected to a service on the top layer, so that our frontend deployment can connect to this Redis master deployment for data requests.

```
$ nano redis-slave.yaml
```

```
apiVersion: v1
kind: Service
metadata:
  name: redis-slave
  namespace: appk8s
  labels:
    app: redis
    tier: backend
    role: slave
spec:
  ports:
  - port: 6379
  selector:
    app: redis
    tier: backend
    role: slave
---
apiVersion: apps/v1
kind: Deployment
metadata:
  name: redis-slave
  namespace: appk8s
spec:
  selector:
    matchLabels:
```

```
      app: redis
      role: slave
      tier: backend
  replicas: 2
  template:
    metadata:
      labels:
        app: redis
        role: slave
        tier: backend
    spec:
      containers:
      - name: slave
        image: gcr.io/google_samples/gb-redisslave:v1
        resources:
          requests:
            cpu: 100m
            memory: 100Mi
        env:
        - name: GET_HOSTS_FROM
          value: dns
        ports:
        - containerPort: 6379
```

This Yaml file will create a deployment for redis workers which will be again connected with a service layer on top. Which will help our redid-master deployment to connect with the redis workers for processing requests made by the frontend layer.

Now that our application (frontend and backend is ready to go) we can create the yaml files for setting RBAC and HPA.

$ cd ..

[5.2] Kubernetes - Roles, Rolebining and Horizontal pod autoscaling.

Let's create a directory for RBAC and HPA yaml files.

```
$ mkdir k8s-rbac-hpa
$ cd k8s-rbac-hpa
```

[5.2.1] Creating a RBAC file for roles and rolebinding in our cluster

```
$ nano rbac.yaml
```

```
apiVersion: rbac.authorization.k8s.io/v1
kind: Role
metadata:
```

```
  namespace: appk8s
  name: podcrud
rules:
- apiGroups: [""] # "" indicates the core API group
  resources: ["pods"]
  verbs: ["get", "watch", "list", "delete", "create", "update"]

---
apiVersion: rbac.authorization.k8s.io/v1
kind: RoleBinding
metadata:
  name: podcrud-binding-mukesh-appk8s
  namespace: appk8s
subjects:

- kind: User
  name: mukesh # "name" is case sensitive
  apiGroup: rbac.authorization.k8s.io
roleRef:
  # "roleRef" specifies the binding to a Role / ClusterRole
  kind: Role #this must be Role or ClusterRole
  name: podcrud # this must match the name of the Role or
ClusterRole you wish to bind to
  apiGroup: rbac.authorization.k8s.io
```

This Yaml file will create a role with the required verbs, additionally it will bind the role to a user (named mukesh in this case). We can see the role and rolebindings are associated with the namespace appk8s, which is the namespace for all deployment, services, pods in this project.

[5.2.1] Creating a HPA, for auto scaling our pods/deployment in our cluster

```
$ nano hpa-scalling.yaml

apiVersion: autoscaling/v1
kind: HorizontalPodAutoscaler
metadata:
  name: hpa-frontend
  namespace: appk8s
spec:
  targetCPUUtilizationPercentage: 50
  minReplicas: 2
  maxReplicas: 10
  scaleTargetRef:
    name: frontend
    kind: Deployment
    apiVersion: apps/v1
```

Now we have roles, rolebinding and horizontal pod autoscaling ready to be deployed in our kubernetes cluster.

Everything seems to be ready now, we just need to start our initialisation process and test our application.

Let’s navigate back to the Capstone project main directory.

```
$ cd ..
```

Now can push all our files (provisioning, configuration and manifests) to our Git repository. So that we can use the scripts through other systems when required. (We will see this in later pages)

```
$ git add .
$ git commit -am”project-capstone”
$ git push
```

[6.0] System provisioning and configuration

[6.1] Terraform, init, plan, apply

Everything is updated now. We can run main.tf using terraform, which will automatically run the main.tf

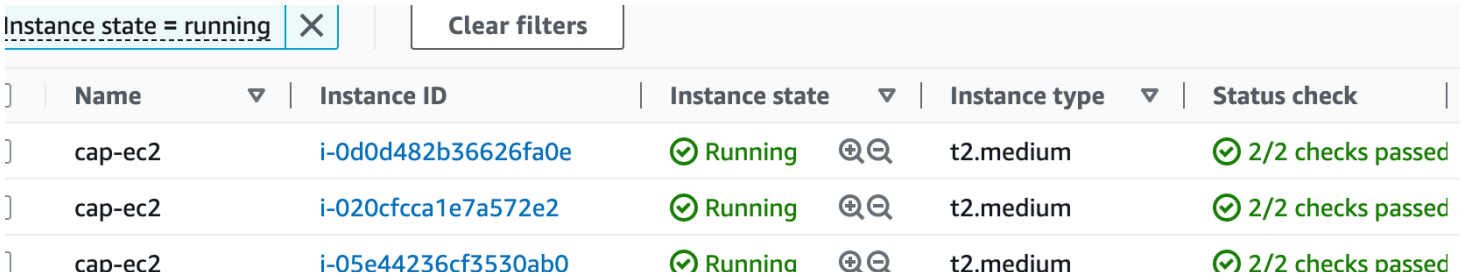
```
$ terraform init
$ terraform validate
$ terraform plan
$ terraform apply –auto-approve
```

```
[ec2-user@ip-172-31-83-227 Capstone-K8S-Infra]$ terraform apply --auto-approve

Terraform used the selected providers to generate the following execution plan. Resource actions
  + create

Terraform will perform the following actions:

  # aws_instance.web[0] will be created
  + resource "aws_instance" "web" {
```

```
aws_instance.web[2] (local-exec): Executing: ["/bin/sh" "-c" "echo 54.227.149.192 >> m
aws_instance.web[2]: Creation complete after 1m42s [id=i-09e207edeaa3a87d9]

Apply complete! Resources: 4 added, 0 changed, 0 destroyed.
[ec2-user@ip-172-31-83-227 Capstone-K8S-Infra]$ ls
Ansible  app-k8s  capkey.pem  k8s-rbac-hba  main.tf  myhosts  terraform.tfstate
[ec2-user@ip-172-31-83-227 Capstone-K8S-Infra]$ cp myhosts Ansible/
[ec2-user@ip-172-31-83-227 Capstone-K8S-Infra]$ cd Ansible/
[ec2-user@ip-172-31-83-227 Ansible]$ ls
docker-install.yml  k8s-install.yaml  k8s-master.yml  myhosts  prime.yml
```

Now, we can see that 4 resources has been added, 3 are our Ec2 and 1 is a security group, that we are using for these group of EC2s. We can also see a myhosts file that contains the public IPs of the created instances (check main.tf for more info).

Now, as seen in the screenshot, copy the myhosts file and capkey.pem to Ansible directory as we will use it as a host file and key to further provision and configure our EC2.



[6.2] Ansible-playbook

```
$ cp myhosts Ansible/
$ cp capkey.pem Ansible/
$ cd Ansible
```

We will copy both myhosts file and the amazon key capkey.pem to the Ansible directory, as we will use them while initiating ansible-playbook command (You can use the absolute path if you do not wish to copy the file, since we are going to use the local path, we are coping the file to the directory).

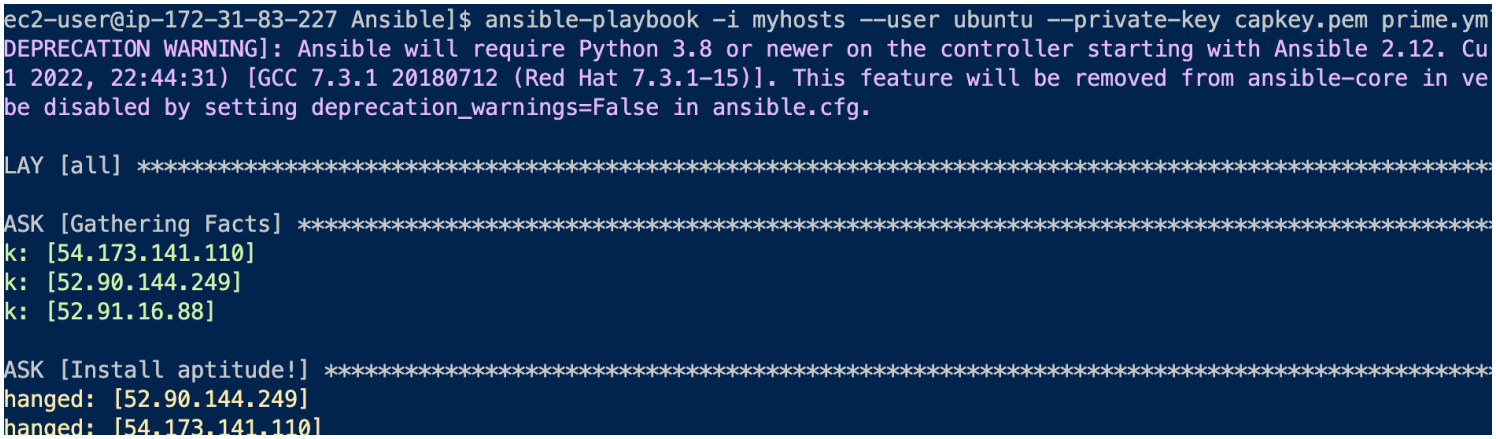
Open myhosts file to edit it and add [master] tag

```
$ nano myhosts
```

Just add [master] on top of one of the IP, and keep others unlabelled. So that the when ansible-playbook runs, it will install kubectl on this node as master.

Save it and run the playbook.

```
$ ansible-playbook -i myhosts --user ubuntu --private-key
capkey.pem prime.yml
```



```
TASK [install kubeadm] *********************************************************
changed: [54.84.133.74]
changed: [52.203.86.48]
changed: [52.90.114.68]

TASK [Creating a daemon json] **************************************************
changed: [52.203.86.48]
changed: [52.90.114.68]
changed: [54.84.133.74]

TASK [restarting the docker] ***************************************************
changed: [52.203.86.48]
changed: [54.84.133.74]
changed: [52.90.114.68]

PLAY [master] ******************************************************************

TASK [Gathering Facts] *********************************************************
ok: [52.90.114.68]

TASK [install kubectl] *********************************************************
ok: [52.90.114.68]

PLAY RECAP *********************************************************************
52.203.86.48               : ok=15   changed=13   unreachable=0    failed=0
52.90.114.68               : ok=17   changed=13   unreachable=0    failed=0
54.84.133.74               : ok=15   changed=13   unreachable=0    failed=0
```

We can see that Ansible has done it's job, we have docker, kubernetes installed along with their requirements and basic configuration on our target systems.

Now, we can go ahead and Setup Kubernetes cluster on these machines and deploy our application.

[7] Deploying the pods, services, autoscalling, RBAC & ETCD backup

---

[7.1] Basic configuration

Most of the configurations have been taken care by our Ansible-playbook, however, there are some basic steps to be followed before we can start using Kubernetes cluster.

SSH into these EC2s and start initialisation of the Kubernetes cluster.
Run these commands on all nodes:

```
$ sudo mkdir /etc/docker

$ cat <<EOF | sudo tee /etc/docker/daemon.json
{
  "exec-opts": ["native.cgroupdriver=systemd"],
  "log-driver": "json-file",
  "log-opts": {
            "max-size": "100m"
  },
  "storage-driver": "overlay2"
}
EOF
```

In Masternode:

```
$ sudo kubeadm init
```

```
Your Kubernetes control-plane has initialized successfully!

To start using your cluster, you need to run the following as a regular user:

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

Alternatively, if you are the root user, you can run:

  export KUBECONFIG=/etc/kubernetes/admin.conf

You should now deploy a pod network to the cluster.
Run "kubectl apply -f [podnetwork].yaml" with one of the options listed at:
  https://kubernetes.io/docs/concepts/cluster-administration/addons/

Then you can join any number of worker nodes by running the following on each as ro

kubeadm join 172.31.31.99:6443 --token dq7qup.cz1wy3wbm2sogqbr \
        --discovery-token-ca-cert-hash sha256:a460511271d3aebc05f0d732493604ae1d56b
6f509896aaf6a4d0d
```

Copy the kubeadm join command which will be printed at the end of the output for later use.

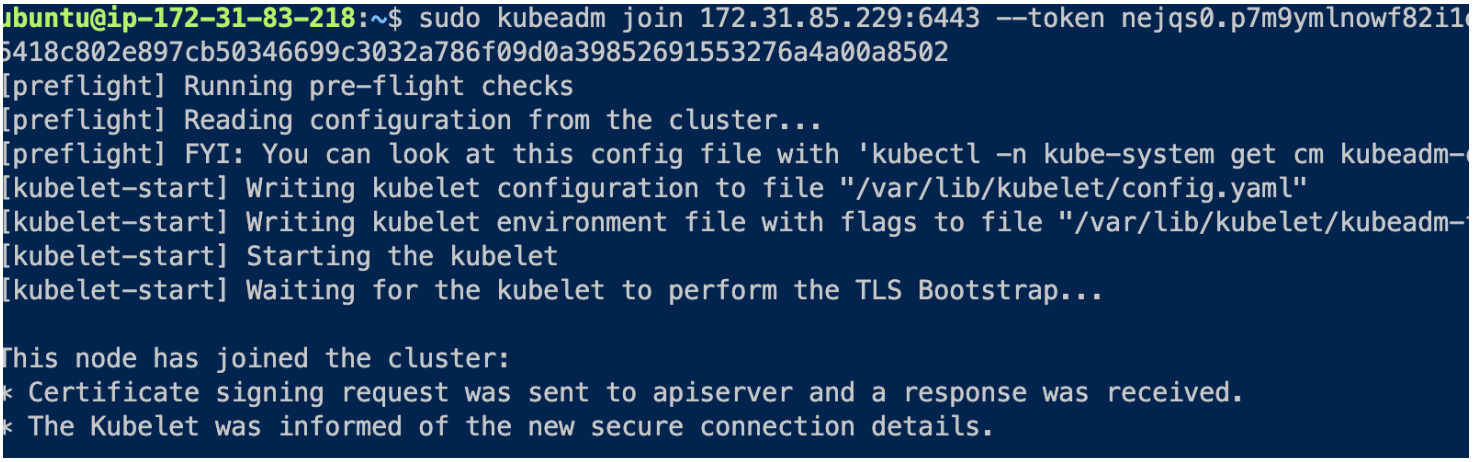
```
$ mkdir -p $HOME/.kube
$ sudo cp -i /etc/kubernetes/admin.conf $HOME/.kube/config
$ sudo chown $(id -u):$(id -g) $HOME/.kube/config
```

Install Container Network Interface (CNI)

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

Now switch to Worker-nodes:
Run the kubeadm join command

```
$ sudo kubeadm join –token******
```



Once the command is successful, you can see the output confirming that the node has joined the cluster.

[7.2] Start deploying the Application

In the Master node:

Let’s create the namespace, which we will use for further deployment.

```
$ kubectl create ns appk8s
```

Now, let’s Install git and clone our **Capstone-K8S-Infra** repository to use it for deployment.

```
$ sudo apt update
$ sudo apt install git
```

```
$ git clone https://github.com/ImMnan/Capstone-K8S-Infra.git
```

Let's navigate to our **~/Capstone-K8S-Infra/appk8s** directory and first deploy the application on our Kubernetes cluster.

$ kubectl apply -f frontend.yaml

$ kubectl apply -f redis-master.yaml

$ kubectl apply -f redis-slave,yaml

```
ubuntu@ip-172-31-85-229:~/Capstone-K8S-Infra/app-k8s$ ls
front-end.yaml  redis-master.yaml  redis-slave.yaml
ubuntu@ip-172-31-85-229:~/Capstone-K8S-Infra/app-k8s$ kubectl create ns appk8s
namespace/appk8s created
ubuntu@ip-172-31-85-229:~/Capstone-K8S-Infra/app-k8s$ kubectl apply -f front-end.yaml
service/frontend created
deployment.apps/frontend created
ubuntu@ip-172-31-85-229:~/Capstone-K8S-Infra/app-k8s$ kubectl apply -f redis-master.ya
service/redis created
deployment.apps/redis-master created
ubuntu@ip-172-31-85-229:~/Capstone-K8S-Infra/app-k8s$ kubectl apply -f redis-slave.yam
service/redis-slave created
deployment.apps/redis-slave created
ubuntu@ip-172-31-85-229:~/Capstone-K8S-Infra/app-k8s$ kubectl get deploy -n appk8s
NAME           READY   UP-TO-DATE   AVAILABLE   AGE
frontend       2/2     2            2           31s
redis-master   1/1     1            1           23s
redis-slave    2/2     2            2           15s
ubuntu@ip-172-31-85-229:~/Capstone-K8S-Infra/app-k8s$ kubectl get pods -n appk8s
NAME                            READY   STATUS    RESTARTS   AGE
frontend-7cc6d85f84-hhhck       1/1     Running   0          41s
frontend-7cc6d85f84-zr7b6       1/1     Running   0          41s
redis-master-6d66bd4cd4-5zsx5   1/1     Running   0          32s
redis-slave-777cf6d6d7-dpfz7    1/1     Running   0          25s
redis-slave-777cf6d6d7-xqtg2    1/1     Running   0          25s
ubuntu@ip-172-31-85-229:~/Capstone-K8S-Infra/app-k8s$ kubectl get svc -n appk8s
NAME          TYPE        CLUSTER-IP       EXTERNAL-IP   PORT(S)        AGE
frontend      NodePort    10.96.188.228    <none>        80:30908/TCP   53s
redis         ClusterIP   10.109.182.235   <none>        6379/TCP       45s
redis-slave   ClusterIP   10.107.8.52      <none>        6379/TCP       38s
ubuntu@ip-172-31-85-229:~/Capstone-K8S-Infra/app-k8s$
```

See the deployment/pods and services created based on the above manifest files

```
$ kubectl get deployment -n appk8s
$ kubectl get pods -n appk8s
$ kubectl get svc -n appk8s
```

[7.3] Applying the RBAC and Horizontal pod autoscaling to this deployment

Navigate to the Capstone-k8s-Infra/k8s-rbac-hpa directory.

```
$ cd ..
$ cd k8s-rbac-hpa

$ kubectl apply -f hpa-scaling.yaml

$ kubectl apply -f rbac.yaml
```

```
ubuntu@ip-172-31-85-229:~/Capstone-K8S-Infra$ cd k8s-rbac-hba/
ubuntu@ip-172-31-85-229:~/Capstone-K8S-Infra/k8s-rbac-hba$ ls
hpa-scaling.yaml  rbac.yaml
ubuntu@ip-172-31-85-229:~/Capstone-K8S-Infra/k8s-rbac-hba$ kubectl apply -f hpa-scaling.yaml
horizontalpodautoscaler.autoscaling/hpa-frontend created
ubuntu@ip-172-31-85-229:~/Capstone-K8S-Infra/k8s-rbac-hba$ kubectl apply -f rbac.yaml
role.rbac.authorization.k8s.io/podcrud created
rolebinding.rbac.authorization.k8s.io/podcrud-binding-mukesh-appk8s created
ubuntu@ip-172-31-85-229:~/Capstone-K8S-Infra/k8s-rbac-hba$ kubectl get hpa -n appk8s
NAME           REFERENCE             TARGETS         MINPODS   MAXPODS   REPLICAS   AGE
hpa-frontend   Deployment/frontend   <unknown>/50%   2         10        2          19s
ubuntu@ip-172-31-85-229:~/Capstone-K8S-Infra/k8s-rbac-hba$ kubectl get roles -n appk8s
NAME      CREATED AT
podcrud   2023-01-16T10:09:15Z
ubuntu@ip-172-31-85-229:~/Capstone-K8S-Infra/k8s-rbac-hba$ kubectl get role-binding -n appk8s
error: the server doesn't have a resource type "role-binding"
ubuntu@ip-172-31-85-229:~/Capstone-K8S-Infra/k8s-rbac-hba$ kubectl get rolebinding -n appk8s
NAME                           ROLE          AGE
podcrud-binding-mukesh-appk8s   Role/podcrud   42s
ubuntu@ip-172-31-85-229:~/Capstone-K8S-Infra/k8s-rbac-hba$
```

Check the RBAC and HPA

```
$ kubectl get roles -n appk8s
$ kubectl get rolebinding -n appk8s

$ kubectl get hpa -n appk8s
```

[7.4] RBAC testing- roles and role-binding to a user

Let's first create a user named "mukesh" as the role-binding is bounded to user 'mukesh'

```
$ useradd mukesh
```

Once the user is created, we can check the user's access within our Kubernetes cluster.

```
$ kubectl get pods -n appk8s --as mukesh
```

Will pass as the role allows to show/list/update/create/delete the pods in appk8s namespace. As

```
$ kubectl get deploy -n appk8s --as mukesh
$ kubectl get nodes —as mukesh
```

Both will fail as the user does not have access to list nodes or deployment.

```
ubuntu@ip-172-31-31-99:~/Capstone-K8S-Infra/k8s-rbac-hba$ kubectl get nodes
NAME               STATUS   ROLES           AGE     VERSION
ip-172-31-18-245   Ready    <none>          4m8s    v1.26.1
ip-172-31-22-255   Ready    <none>          4m13s   v1.26.1
ip-172-31-31-99    Ready    control-plane   6m41s   v1.26.1
ubuntu@ip-172-31-31-99:~/Capstone-K8S-Infra/k8s-rbac-hba$ kubectl get nodes --as mukesh
Error from server (Forbidden): nodes is forbidden: User "mukesh" cannot list resource "nodes" in API g
e cluster scope
ubuntu@ip-172-31-31-99:~/Capstone-K8S-Infra/k8s-rbac-hba$ kubectl get pods -n appk8s  --as mukesh
NAME                           READY   STATUS    RESTARTS   AGE
frontend-7cc6d85f84-qsb2j      1/1     Running   0          3m3s
frontend-7cc6d85f84-rqpds      1/1     Running   0          3m3s
redis-master-6d66bd4cd4-84cll  1/1     Running   0          2m40s
redis-slave-777cf6d6d7-7w2rw   1/1     Running   0          2m53s
redis-slave-777cf6d6d7-mb6sb   1/1     Running   0          2m53s
ubuntu@ip-172-31-31-99:~/Capstone-K8S-Infra/k8s-rbac-hba$ kubectl get deploy -n appk8s --as mukesh
Error from server (Forbidden): deployments.apps is forbidden: User "mukesh" cannot list resource "dep
PI group "apps" in the namespace "appk8s"
```

[7.5] Creating Back-up/ Restore point for ETCD

Now since the deployment is complete, let's backup ETCD.

Users mostly interact with etcd by putting or getting the value of a key. We do that by using **etcdctl**, a command line tool for interacting with etcd server. In this section, we are downloading the etcd binaries so that we have the **etcdctl** tool with us to interact.

1) Create a temporary directory for the ETCD binaries.

```
$ mkdir -p /tmp/etcd && cd /tmp/etcd

$ curl -s https://api.github.com/repos/etcd-io/etcd/releases/
latest | grep browser_download_url | grep linux-amd64 | cut -d '"'
-f 4 | wget -qi -
$ cd etcd-*/
$ mv etcd* /usr/local/bin/
$ cd ~
$ rm -rf /tmp/etcd
```

```
ubuntu@ip-172-31-31-99:/tmp/etcd$ curl -s https://api.github.com/repos/etcd-io/etcd/releases/latest | g
download_url | grep linux-amd64 | cut -d '"' -f 4 | wget -qi -
ubuntu@ip-172-31-31-99:/tmp/etcd$ ls
etcd-v3.5.7-linux-amd64.tar.gz
ubuntu@ip-172-31-31-99:/tmp/etcd$ tar xvf *.tar.gz
etcd-v3.5.7-linux-amd64/
etcd-v3.5.7-linux-amd64/README.md
etcd-v3.5.7-linux-amd64/READMEv2-etcdctl.md
etcd-v3.5.7-linux-amd64/etcdutl
etcd-v3.5.7-linux-amd64/etcdctl
etcd-v3.5.7-linux-amd64/Documentation/
etcd-v3.5.7-linux-amd64/Documentation/README.md
etcd-v3.5.7-linux-amd64/Documentation/dev-guide/
etcd-v3.5.7-linux-amd64/Documentation/dev-guide/apispec/
etcd-v3.5.7-linux-amd64/Documentation/dev-guide/apispec/swagger/
etcd-v3.5.7-linux-amd64/Documentation/dev-guide/apispec/swagger/v3election.swagger.json
etcd-v3.5.7-linux-amd64/Documentation/dev-guide/apispec/swagger/rpc.swagger.json
etcd-v3.5.7-linux-amd64/Documentation/dev-guide/apispec/swagger/v3lock.swagger.json
etcd-v3.5.7-linux-amd64/README-etcdutl.md
etcd-v3.5.7-linux-amd64/README-etcdctl.md
etcd-v3.5.7-linux-amd64/etcd
ubuntu@ip-172-31-31-99:/tmp/etcd$ ls
etcd-v3.5.7-linux-amd64  etcd-v3.5.7-linux-amd64.tar.gz
ubuntu@ip-172-31-31-99:/tmp/etcd$ cd etcd-*/
ubuntu@ip-172-31-31-99:/tmp/etcd/etcd-v3.5.7-linux-amd64$ mv etcd* /usr/local/bin/
mv: cannot move 'etcd' to '/usr/local/bin/etcd': Permission denied
mv: cannot move 'etcdctl' to '/usr/local/bin/etcdctl': Permission denied
mv: cannot move 'etcdutl' to '/usr/local/bin/etcdutl': Permission denied
ubuntu@ip-172-31-31-99:/tmp/etcd/etcd-v3.5.7-linux-amd64$ sudo !!
sudo mv etcd* /usr/local/bin/
ubuntu@ip-172-31-31-99:/tmp/etcd/etcd-v3.5.7-linux-amd64$ cd ~
ubuntu@ip-172-31-31-99:~$ rm -rf /tmp/etcd
```

The etcd server is the only stateful component of the Kubernetes cluster. Kuberenetes stores all API objects and settings on the etcd server.
Backing up this storage is enough to restore the Kubernetes cluster's state completely.

Take a snapshot of the etcd datastore using etcdctl:

```
$ sudo ETCDCTL_API=3 etcdctl snapshot save snapshot.db --cacert /
etc/kubernetes/pki/etcd/ca.crt --cert /etc/kubernetes/pki/etcd/
server.crt --key /etc/kubernetes/pki/etcd/server.key
```

```
ubuntu@ip-172-31-31-99:~$  sudo ETCDCTL_API=3 etcdctl snapshot save snapshot.db --cacert /etc/kubernetes/pki/etcd/c
a.crt --cert /etc/kubernetes/pki/etcd/server.crt --key /etc/kubernetes/pki/etcd/server.key
{"level":"info","ts":"2023-01-20T12:43:53.559Z","caller":"snapshot/v3_snapshot.go:65","msg":"created temporary db f
ile","path":"snapshot.db.part"}
{"level":"info","ts":"2023-01-20T12:43:53.569Z","logger":"client","caller":"v3@v3.5.7/maintenance.go:212","msg":"op
ened snapshot stream; downloading"}
{"level":"info","ts":"2023-01-20T12:43:53.569Z","caller":"snapshot/v3_snapshot.go:73","msg":"fetching snapshot","en
dpoint":"127.0.0.1:2379"}
{"level":"info","ts":"2023-01-20T12:43:53.644Z","logger":"client","caller":"v3@v3.5.7/maintenance.go:220","msg":"co
mpleted snapshot read; closing"}
{"level":"info","ts":"2023-01-20T12:43:53.658Z","caller":"snapshot/v3_snapshot.go:88","msg":"fetched snapshot","end
point":"127.0.0.1:2379","size":"5.0 MB","took":"now"}
{"level":"info","ts":"2023-01-20T12:43:53.658Z","caller":"snapshot/v3_snapshot.go:97","msg":"saved","path":"snapsho
t.db"}
Snapshot saved at snapshot.db
```

View that the snapshot was successful:

```
$ sudo etcdutl snapshot status snapshot.db
```

```
ubuntu@ip-172-31-31-99:~$ sudo etcdutl snapshot status snapshot.db
6e126a88, 2153, 994, 5.0 MB
```

Note: Important Note: If you are backing up and restoring the cluster do not run the status command after the backup this might temper the backup due to this restore process might fail.

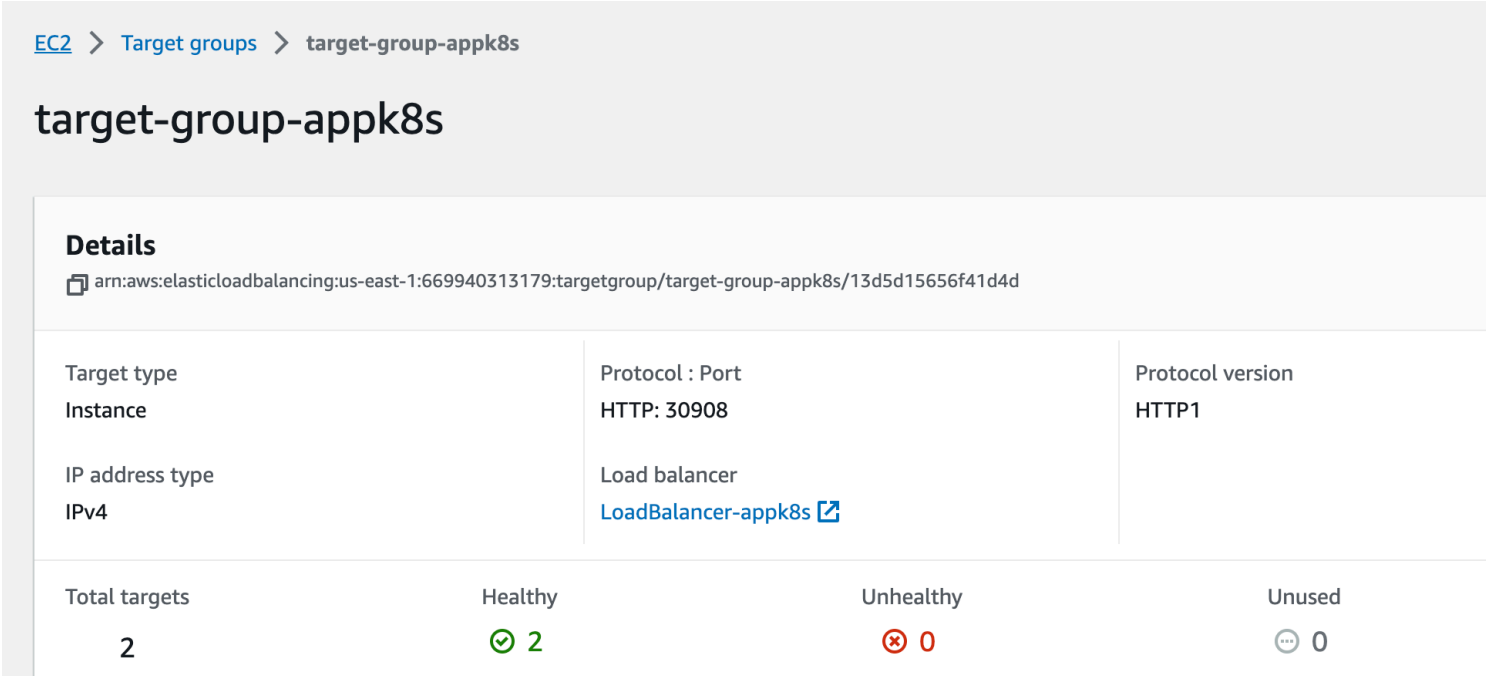
Backing-up The Certificates And Key Files:
Zip up the contents of the etcd directory to save the certificates too

```
$ sudo tar -zcvf etcd.tar.gz /etc/kubernetes/pki/etcd
```

[8] Creating and configuring Application Load Balancer in AWS

[8.1] Create target group:

1. Login to AWS
2. Navigate to EC2 > Load Balancing > Target group
3. Click on Create target group
4. Choose a target type - Instances
5. Name the target group
6. Select the correct VPC, add port 30908 as target port (because our application is running on port 30908)
7. Click next
8. We will be able to the list of EC2s
9. Select the 2 worker node and create the target group
10. Once the health checkups are done, the nodes/target will be shown under healthy state.

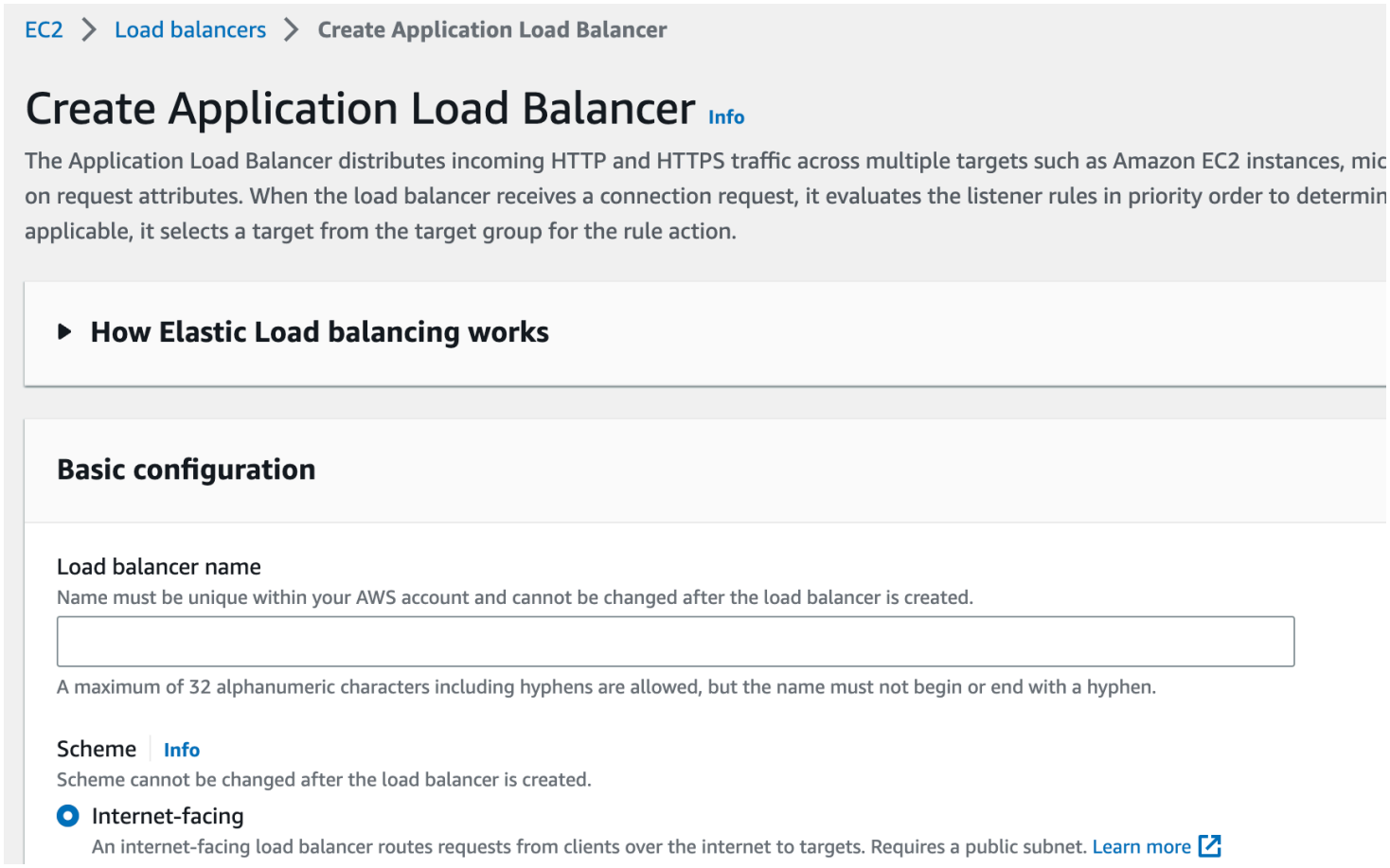


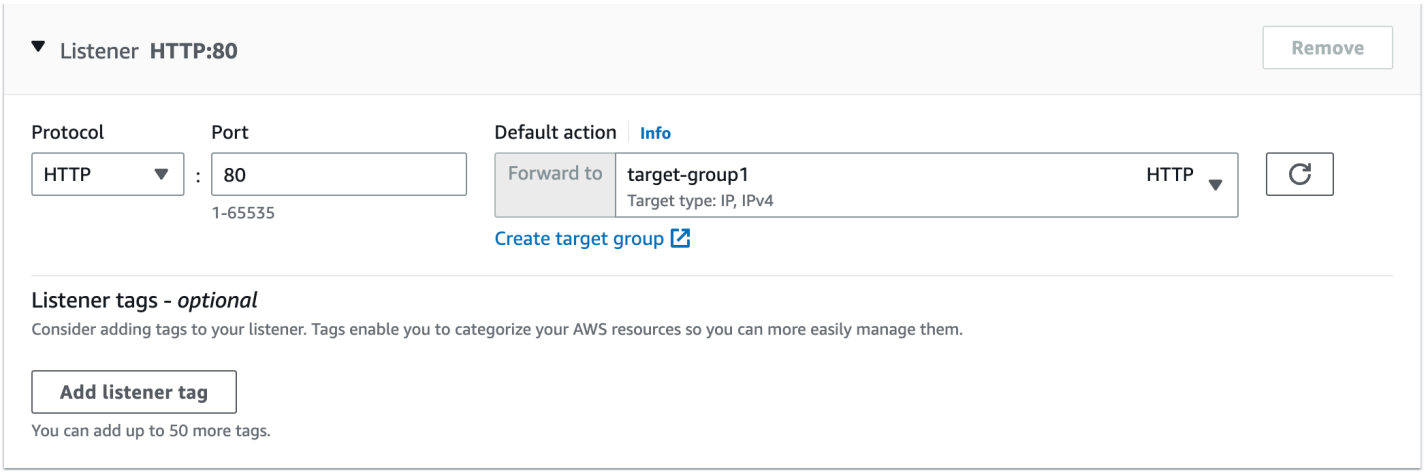
[8.3] Creating Application Load Balancer

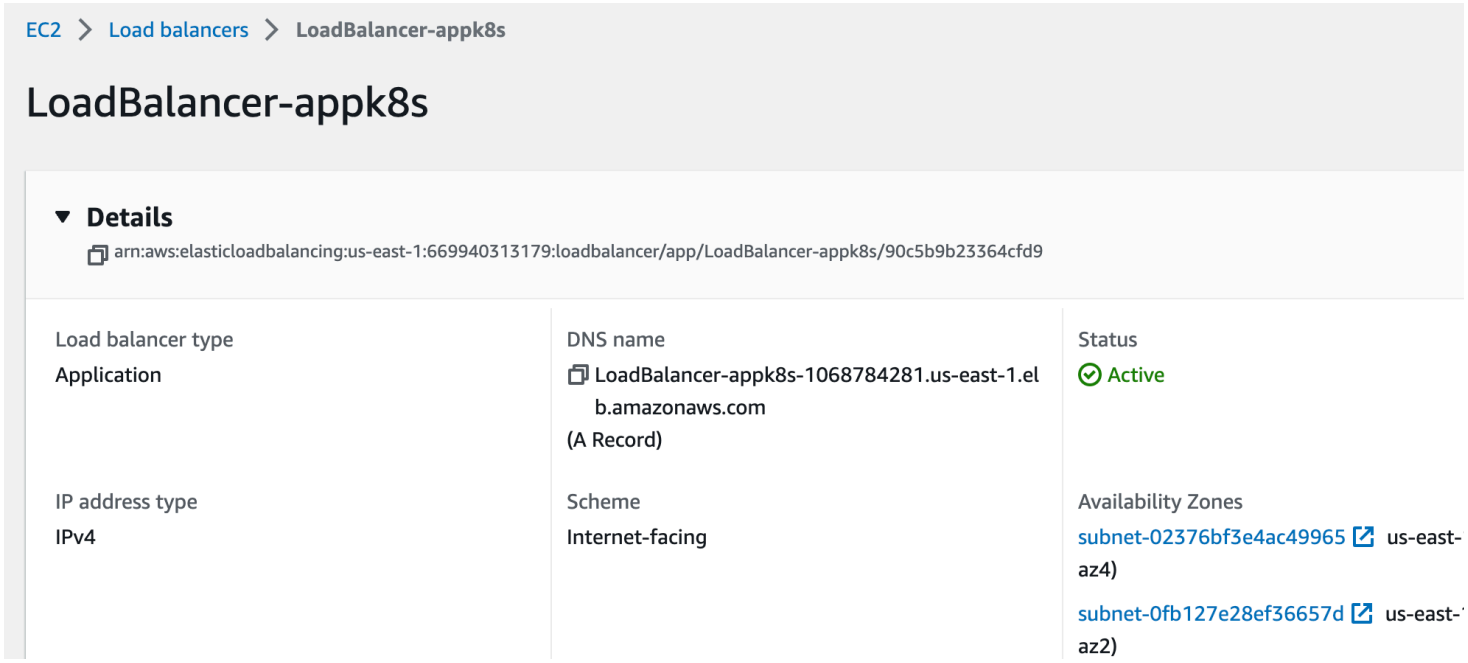
Now, let's create AWS application load balancer:

1. Login to AWS
2. Navigate to EC2 > Load Balancing > Load Balancers
3. Click on create load balancer
4. Select Application load balancer
5. Declare the name of the Load balancer
6. Scheme - Internet facing
7. Select the correct VPC in the network mapping

8. Select the security group, the one used for Kubernetes cluster
9. In Listeners and routing, add the port - 80, in Default action ‘select a target group’ the one we created.
10. Click on create Load balancer



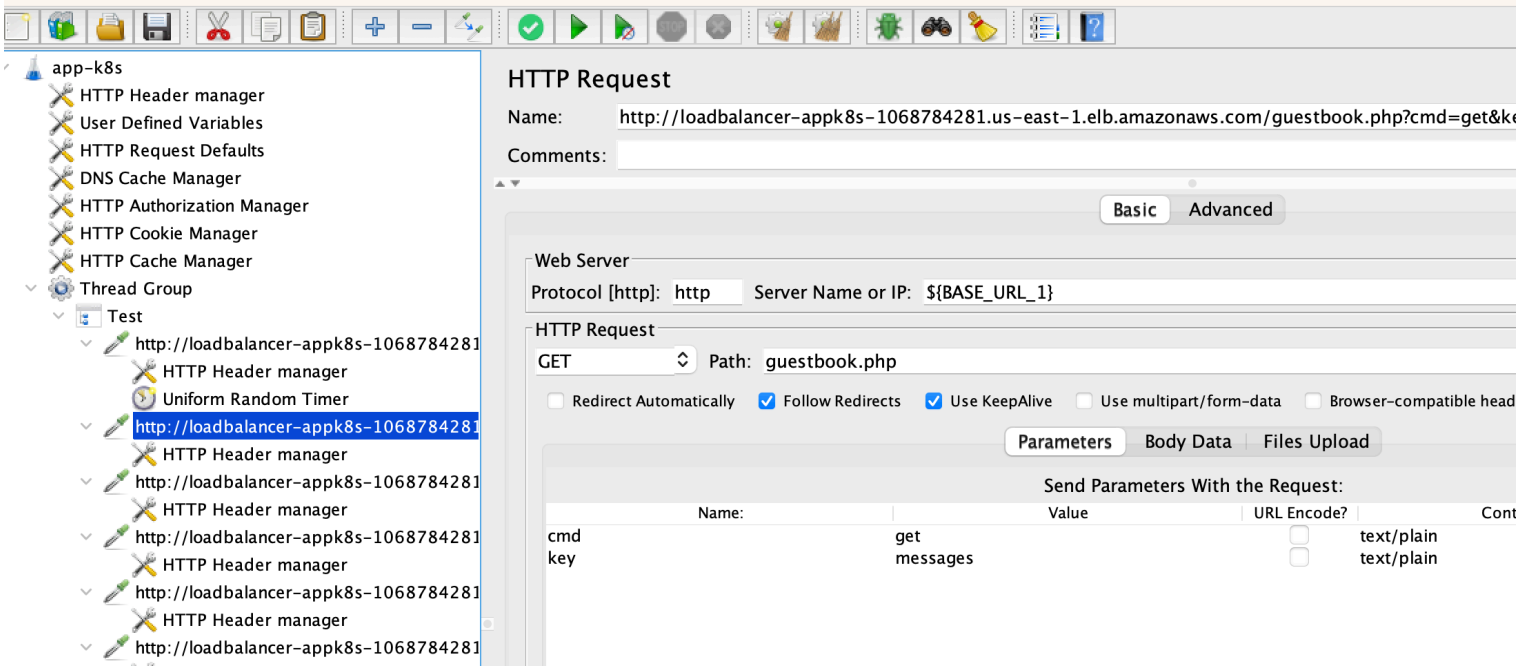




Once the states is active, copy the DNS name to test out the load balancer.
So now we can see that the DNS name (A record) of the load balancer is successfully hitting the EC2s under the target group and the EC2 are directing the traffic to the Kubernetes pods through NodePort service - port 30908.

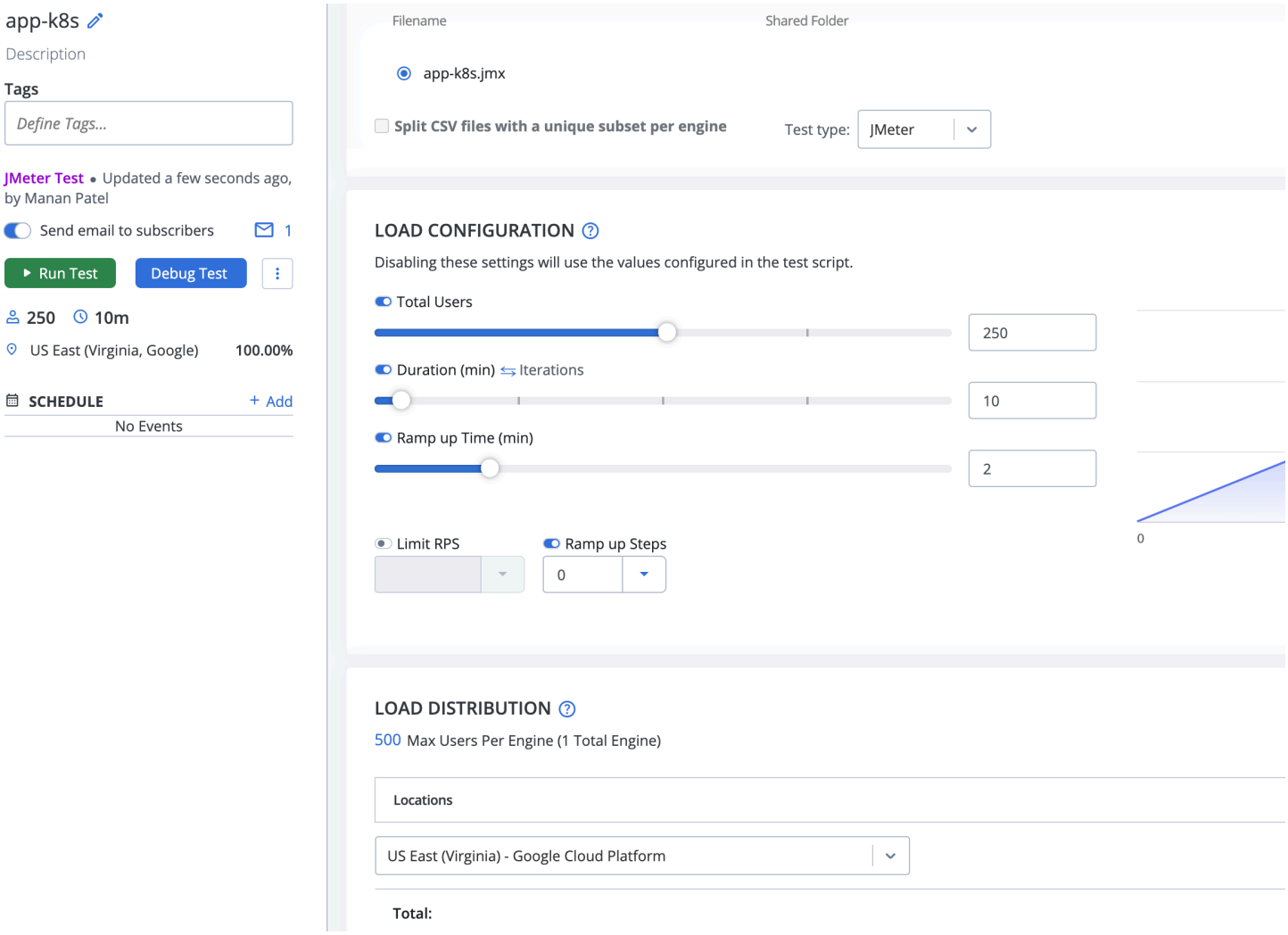
[9] Test case- Load testing our application

We will load test our application to observe the scalability. As we have already configured horizontal pod autoscaling for our application deployment, we can see how it works in real life scenario where there will be load pushed towards the application.



1. We will create a JMeter test, which is a popular tool for creating performance tests named **app-k8s.jmx**
2. The test will GET, POST to the target application (see screenshot or the app-k8s.jmx script in jmeter)
3. Once the test is created in Jmeter, we will load test the application using Blazemeter Performance test.
4. Blazemeter is the industry leading automation testing platform, and we can simply upload the JMeter script and run the test.
5. Login to Blazemeter, **Navigate to Performance test > Create a test > Select performance test.**
   Name the test > Upload the test app-k8s.jmx > set the load configuration (as shown in figure)
6. We are going to run the load of 250 VUs for 10 mins with a ramp-up time of 2 minutes.
7. Now click on Run test button to run the test.

See the screenshot below, showing test configuration in Blazemeter.



Observe Autoscalling at the kubernetes pod level

Now, since the test is running we can see the change in our Kubernetes deployment.

```
$ kubectl get deploy -n appk8s
$ kubectl get hpa -n appk8s
$ kubectl get pods -n appk8s
```

We can see the deployment has been scaled up, there are 5 pods running now out of 10. The HPA is at 194% whereas before the test began, it was comfortably at 0%(See screenshots). The first command show that there were limited pod and HPA to 0%, once the test starts, running the command again will show HPA at 194% and Frontend pods are scaled at 5 pods.

```
ubuntu@ip-172-31-85-229:~/Capstone-K8S-Infra/k8s-rbac-hba$ kubectl top pods -n appk8s
NAME                           CPU(cores)   MEMORY(bytes)
frontend-7cc6d85f84-hhhck      1m           26Mi
redis-master-6d66bd4cd4-5zsx5  2m           3Mi
redis-slave-777cf6d6d7-dpfz7   2m           2Mi
redis-slave-777cf6d6d7-xqtg2   2m           2Mi
ubuntu@ip-172-31-85-229:~/Capstone-K8S-Infra/k8s-rbac-hba$ kubectl get deploy -n appk8s
NAME           READY   UP-TO-DATE   AVAILABLE   AGE
frontend       1/1     1            1           57m
redis-master   1/1     1            1           56m
redis-slave    2/2     2            2           56m
ubuntu@ip-172-31-85-229:~/Capstone-K8S-Infra/k8s-rbac-hba$ kubectl get hpa -n appk8s
NAME       REFERENCE              TARGETS   MINPODS   MAXPODS   REPLICAS   AGE
frontend   Deployment/frontend   0%/50%    1         10        1          11m
ubuntu@ip-172-31-85-229:~/Capstone-K8S-Infra/k8s-rbac-hba$ kubectl get hpa -n appk8s
NAME       REFERENCE              TARGETS    MINPODS   MAXPODS   REPLICAS   AGE
frontend   Deployment/frontend   194%/50%   1         10        8          12m
ubuntu@ip-172-31-85-229:~/Capstone-K8S-Infra/k8s-rbac-hba$ kubectl get deploy -n appk8s
NAME           READY   UP-TO-DATE   AVAILABLE   AGE
frontend       5/10    10           5           58m
redis-master   1/1     1            1           58m
redis-slave    2/2     2            2           58m
```

We can also see the CPU cores utilised per pod in our deployment.

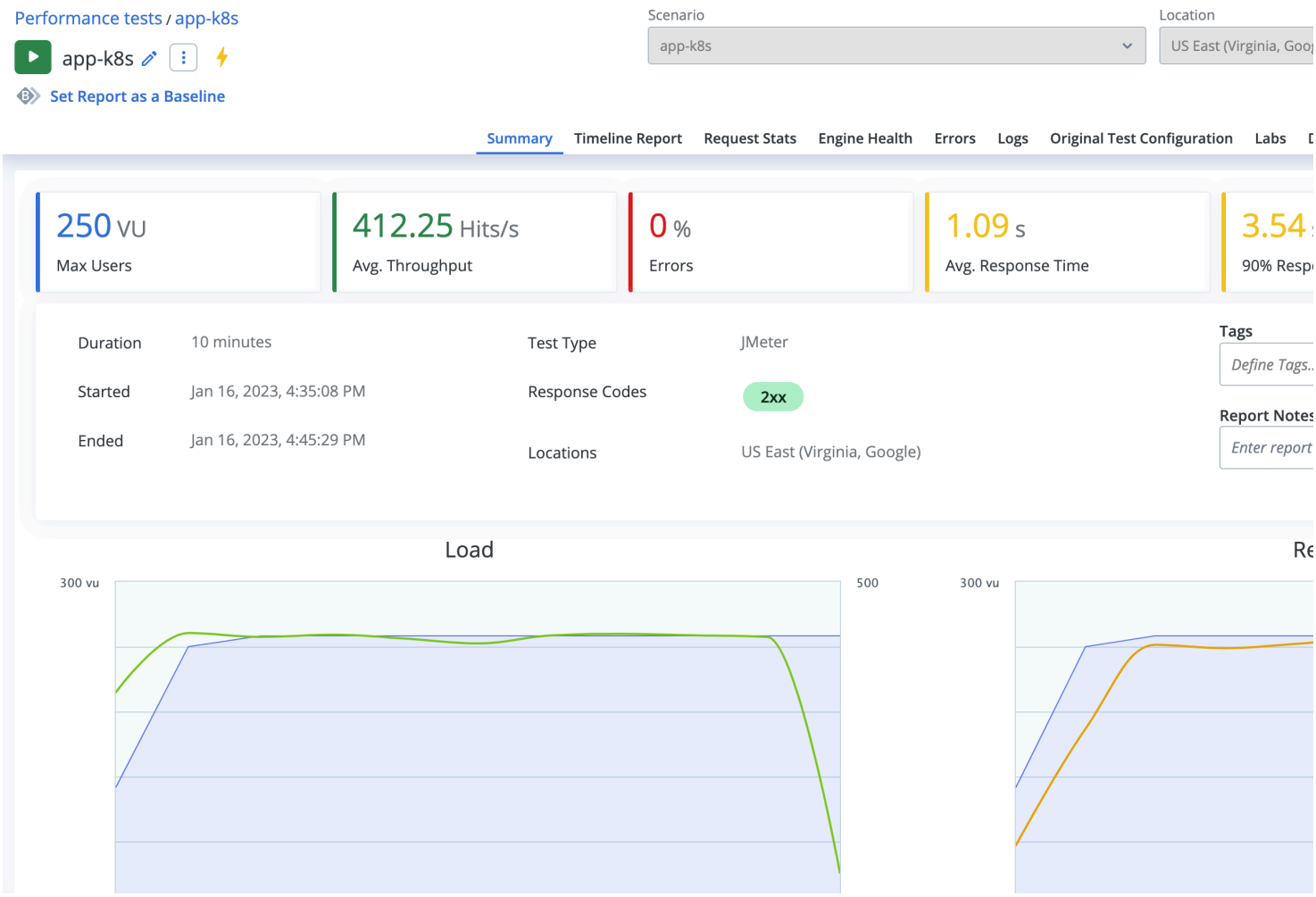
```
$ kubectl top pods -n appk8s
```

```
ubuntu@ip-172-31-85-229:~/Capstone-K8S-Infra/k8s-rbac-hba$ kubectl top pods -n appk8s
NAME                           CPU(cores)   MEMORY(bytes)
frontend-7cc6d85f84-dtl4v      318m         142Mi
frontend-7cc6d85f84-fjfch      319m         135Mi
frontend-7cc6d85f84-hhhck      317m         131Mi
frontend-7cc6d85f84-k262n      320m         136Mi
frontend-7cc6d85f84-thwx7      319m         133Mi
redis-master-6d66bd4cd4-5zsx5  2m           3Mi
redis-slave-777cf6d6d7-dpfz7   8m           2Mi
redis-slave-777cf6d6d7-xqtg2   8m           2Mi
ubuntu@ip-172-31-85-229:~/Capstone-K8S-Infra/k8s-rbac-hba$
```

[9.1] Test results:

As a result of our load testing, we can see that we have used 250VUs, which is generating the throughput of 412 hits/sec with average response time to be just around 1 sec.

This equals to 247,200 hits to this application within the 10 min test, and the Kubernetes cluster is handling the pods very effectively as we cannot see a single error for this test. Not a single timeout or 404 not found error after 250K requests to this application.



This means that the application is available all the time, thanks to our deployment and autoscaling in our Kubernetes cluster. Moreover, the Load balancer is performing efficiently as well, making sure that the load is managed through the target group.

The complete test report is here: (The report is publicly available to go through**)**

https://a.blazemeter.com/app/?public-token=W6DDrwPuMXGXAlCBt3EKcBYZgloFtznQRDTNNCIOauiJbfWPwG#/accounts/1195941/workspaces/1226630/projects/1473373/masters/66301406/summary

[10] Conclusion & concepts used in the project

[10.1] DevOps process

DevOps is a set of practices and tools designed to shorten the life cycle of a software development, integration and deployment process. It is an agile relationship that uses automated processes to bridge the gap between the Development and IT operations.
We are using Terraform, Ansible, Docker & Kubernetes to secure DevOps lifecycle, which will eventually lead us to achieve the main goal of making the app reliable, faster, and secured.

[10.2] Configuration Management

Configuration management is a process for maintaining computer systems, servers, and software in a desired, consistent state. This means that we can ensure that the system environment is the same for all participants to the DevOps process

For this project we use a configuration + infrastructure management tools such as Terraform and Ansible to automatically provision our Cloud infrastructure, EC2/VM, their environment and then configure these Machines to support containers/containerised application and orchestration.

[10.3] Containerised application & Orchestration

A **container** is a portable computing environment. It contains everything an application needs to run, from binaries to dependencies to configuration files.

- Portability
- Efficiency
- Agility
- Faster delivery
- Improved security
- Faster app startup
- Easier management
- Flexibility

On top of that we are using Kubernetes as an orchestration tool for these containerised application.
Kubernetes automates operational tasks of container management and includes built-in commands for deploying applications, rolling out changes to your applications, scaling your applications up and down to fit changing needs, monitoring your applications, and more—making it easier to manage applications.

[10.4] Automation testing

Performance testing is important to prepare your application for normal activity as well as special events. Doing performance testing helps you evaluate the speed, stability, and scalability of an application.

We are using **BlazeMeter** as out automation testing tool, yes, the test is created in Jmeter, but to scale the test and increase the load generation we are using Blazemeter as a testing solution.

Blazemeter is a SaaS-based performance testing tool, fully compatible with JMeter and many other open source load testing tools. Whether you are running automated tests every night as part of your continuous integration workflow or load testing before a special event.
BlazeMeter simplifies and improves script maintenance. BlazeMeter is fully compatible with the latest versions of many open source testing tools, such as JMeter,Gatling, Selenium, and Taurus

BlazeMeter runs in the cloud, so we don't have to rely on our own machine's resources when scaling the number of virtual users. You can easily ramp up to thousands or tens of thousands of virtual users to simulate how your system will perform under heavy usage and surprising traffic spikes.

Moreover, BlazeMeter reports are clear, comprehensive, and insightful.

[10.5] The conclusion for enhancing the EasyPay application

- Faster release due to less issues, containerised application that can be easily deployed and orchestrated through Kubernetes.
- Better engagement and productivity since all teams are equally responsible to the result.
- Automated configuration of systems lead to increased consistencies, compliances and security since the system is automatically provisioned, configured and maintained at desired state.
- Automated testing - the tests can be build before completing the codes and automatically tested, through Blazemeter automation testing pipeline. Tests can be run in schedules/cronjobs as well as the test execution trigger can be linked with Jenkins to automatically initiate testing with a new build.
- Good management of codes with central repository version control. helps teams solve problems, tracking every change by each contributor and helping prevent concurrent work from conflicting. It also helps for rollback in case of an incident caused by the new version deployment.

- ETCD backup/restore point will help team manage incidents. such as losing all control plane nodes. The snapshot file contains all the Kubernetes states and critical information. In order to keep the sensitive Kubernetes data safe, encrypt the snapshot files.

- Automatically deploy and scale micro-services and applications based on the load. The pods/deployments will be scaled based on the incoming traffic/load. Not only manages and enhances the application response + availability but also saves costs, when the incoming traffic is less by stalling the pods/deployment down.