HW-4: Running MNIST Training & Inference on K8s

1. INTRODUCTION

In modern computing, the emergence of virtualization and containers have changed how software is developed,deployed and maintained. As computing needs of quickly growing softwares increases, virtualization and containers are heavily used for efficient use of resources and providing flexibility in deployment. However, managing and orchestrating these containers at scale presents a new set of challenges.This report created as part of *Homework-4* talks about running MNIST digit recognition model training and inference on a K8 cluster while making use of persistent storage.

2. BACKGROUND STUDY

What is Kubernetes?

Kubernetes, (K8s), was initially created to streamline the deployment and management of containerized applications. It was developed by Google and is currently an open source platform. It was based on Google's internal container orchestration system Borg. The main objectives of Kubernetes is:

- Automation: Kubernetes automates the deployment and scaling of containerized applications. This means that we as developers don't need to manually handle things and this also helps reduce human errors.
- Scalability: It does automatic scaling. This allows the applications to handle varying loads dynamically.
- Portability: Kubernetes provides a consistent environment across different infrastructure providers, making it easier to migrate and run applications seamlessly in various environments. *(I experienced this personally while working on the project as running container meant that my teammate who uses a Macbook and me who uses a Windows laptop didn't run into infrastructure issues, which we would usually run into before using containers and Kubernetes)*

Some important Kubernetes concepts and terms:

- Pods are the smallest units which can be deployed on Kubernetes. They consist of one or more than one container that share the same network and storage.
- Nodes are physical or virtual machines in a Kubernetes cluster where containers are being deployed.
- Clusters are a group of nodes.
- Services provide a stable endpoint to access a set of pods which means that we can communicate easily inside a cluster.
- Controllers handle deployment , scaling and updating of application components.

In a nutshell, containerized apps run inside pods, which are the smallest deployable units that run on nodes. Nodes form a cluster where compute, storage, and network resources are combined, shared, and coordinated across pods. Pods are grouped together in a deployment and presented as a service to clients or other applications and services

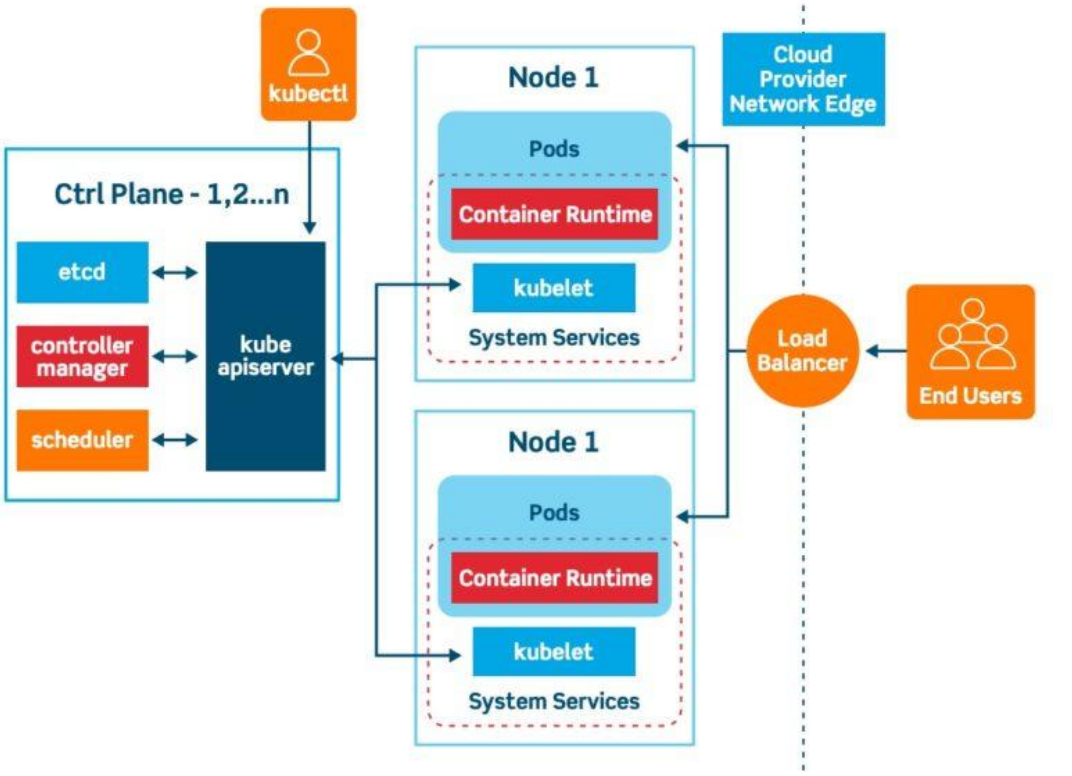


Figure: Kubernetes Architecture

GKE AUTOPILOT V/s STANDARD CLUSTERS

There are two types of clusters available for creation in GCP GKE:
1) Autopilot which is the automatic and hands off version
2) Standard Cluster which requires configuration but offers more control.

In an autopilot cluster the zonal control plane and nodes are managed and provisioned by GKE which means it automatically scales depending on demand. Standards clusters offer more control where the user is responsible for scaling
Autopilot provides a good starting point to quickly deploy applications without spending too much time understanding demands and nodes. It does have limitations with respect to the number of pods per cluster, usage of GKE managed namespace etc. but if someone wants to

quickly setup autopilot is the way to go. Standard clusters offer more control but require deep understanding of your applications demands,traffic and is often the work of a skilled developer, while autopilot is easy to use even for a beginner.

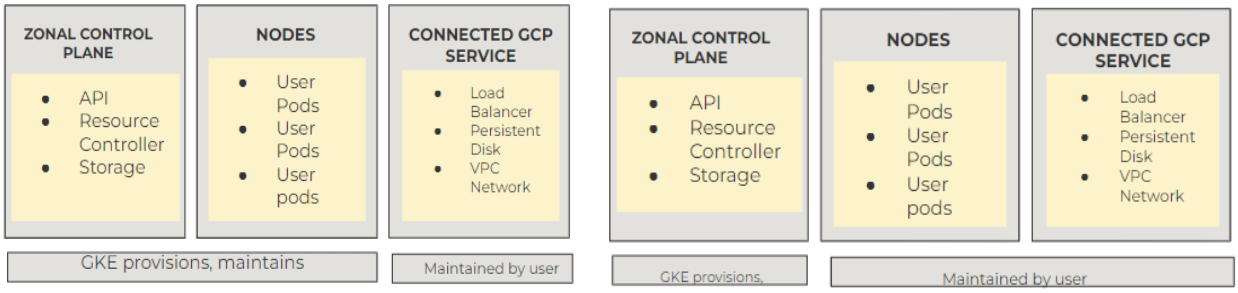


Figure: GKE Autopilot (left) , GKE Autopilot(Right)

Persistent Storage

Persistent storage is the long-term memory of the application and is required when data needs to survive beyond the lifespan of individual containers. Some reasons why persistent storage is needed are:

1.Data Retention: In many applications, data needs to persist even if the application or container is restarted or moved to a different node. Without persistent storage, all that valuable data would be lost each time a container shuts down.

2. Stateful Applications: Some applications have a state that needs to be preserved.If we need to ensure that your application can recover its state after being restarted or moved, we need persistent storage.

3.Scaling Applications: When applications need to be scaled horizontally, persistent storage ensures that each instance of the application has the same data. This ensures that data consistency is maintained to avoid any loss of data integrity.

Key Persistent Storage Concepts wrt Google Kubernetes Engine:

1.Persistent Volumes (PVs): PVs are the abstraction layer that represents physical storage resources in a cluster. They decouple the details of how storage is provided from how it's consumed. A PV in Kubernetes could be a physical disk or even a network-attached storage.

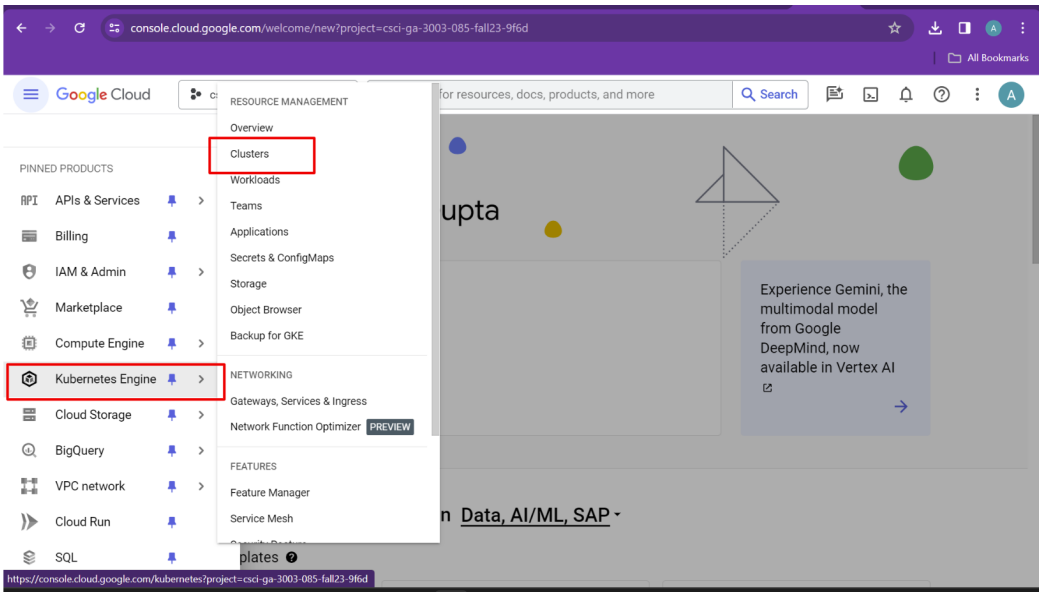
2.Persistent Volume Claims (PVCs):PVCs act as a request for storage by a user or a group of users. They provide a way to claim a specific amount of storage defined by the PV.

3. Storage Classes:These are used to dynamically provision PVs based on predefined templates. Storage Classes enable automatic provisioning, ensuring that applications get the storage they need without manual intervention.

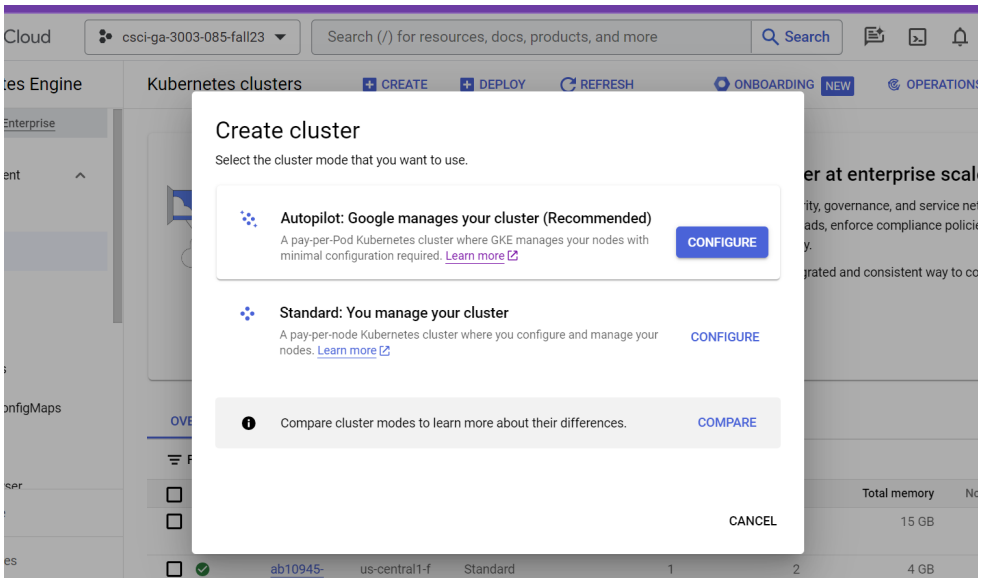
3. CREATING A KUBERNETES CLUSTER ON GCP

I created a cluster using Autopilot using the GCP user interface.

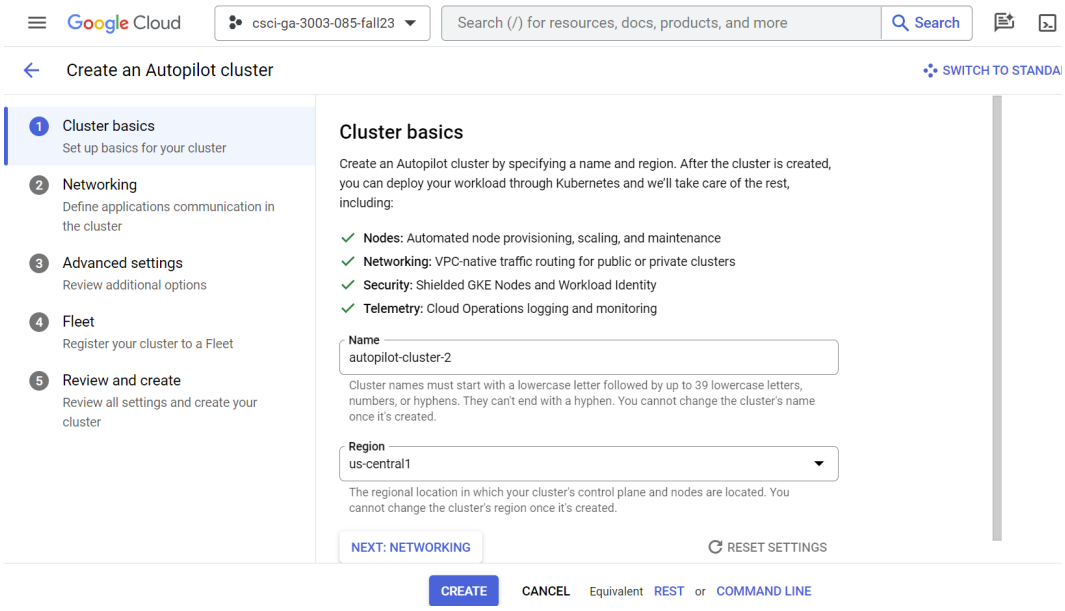
1) Select Clusters



2) Create an autopilot cluster



3) Select cluster settings.



Using command line-

```
gcloud beta container --project "csci-ga-3003-085-fall23-9f6d" clusters create-auto "ag8733-cluster1" --region "us-central1" --release-channel "regular" --network "projects/csci-ga-3003-085-fall23-9f6d/global/networks/csci-ga-3003-085-fall23-net" --subnetwork "projects/csci-ga-3003-085-fall23-9f6d/regions/us-central1/subnetworks/csci-ga-3003-085-fall23-subnet-02" --cluster-ipv4-cidr "/17" --binauthz-evaluation-mode=DISABLED
```

I created an autopilot cluster called ag8733-cluster1, which automatically scales the nodes up and down.

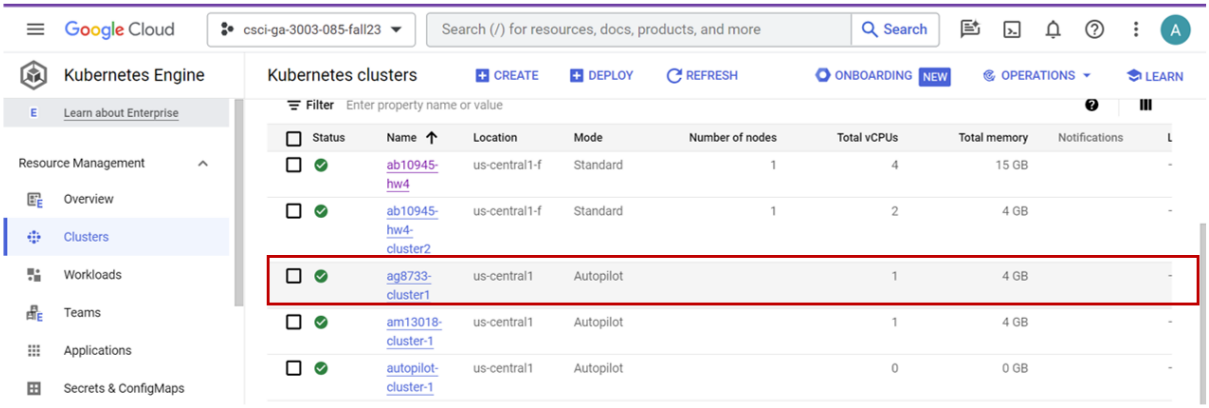


Figure: Created cluster ag8733-cluster1

4. SETTING UP PVC

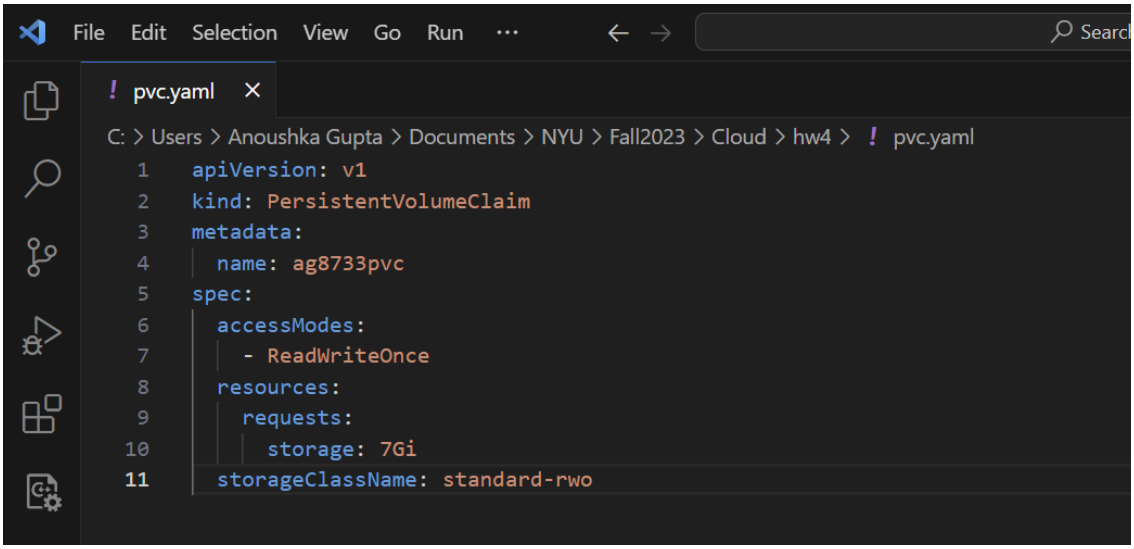


Figure: pvc.yaml

- apiVersion: SPecifies the API version of the resource, here I am using the core/v1 version of the Kubernetes API
- Kind : this defines the type of resource. Here, it's a PersistentVolumeClaim.
- The name of the PVC has been set using the metadata field. I have set it as *ag8733pvc.*
- accessModes: Describes the desired access mode for the persistent volume. ReadWriteOnce, means the volume can be mounted as read-write by a single node.
- Resources are set to 7Gi meaning that I'm requesting for a 7GB storage.

- storageClassName: set to “standard-rwo."

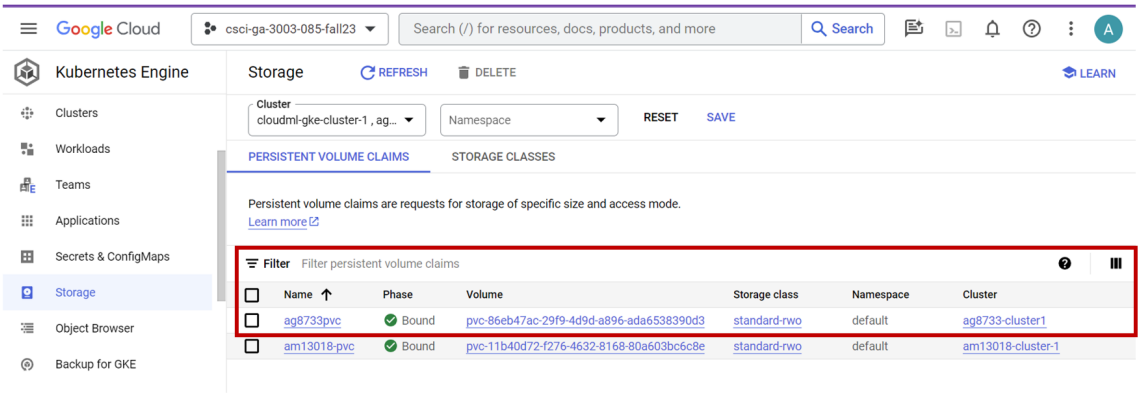


Figure: Shows my persistent storage which has been bound.

5. TRAINING THE MODEL

The main goal of this exercise was to create a containerized version of the training script which can do the training for the MNIST recognizer model and then save the weights of the model to persistent storage so that they can be later used during inference.

1) I made the necessary changes in the pytorch/examples mnist.py file.
   I train the model for one epoch and then save the model weights in the */mnt* directory with the name *ag8733_model.pth*
2) I then create a dockerfile which uses the pytorch base image and then runs the *train.py* script. I use this command inside the dockerfile to run the script-

```
CMD python train.py --batch-size 32 --epochs 1 --no-cuda
--no-mps --save-model
```

3) Build the docker file using docker build command and then tag the image.
4) Create a repository on dockerhub
5) Push this train image to docker hub using-

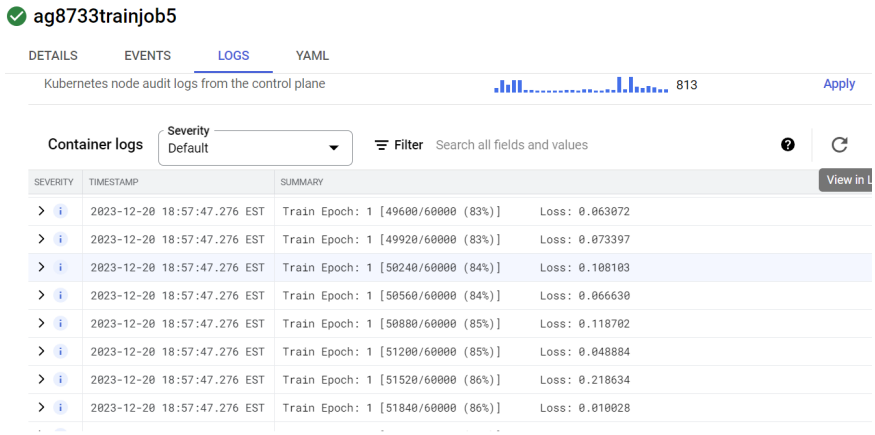
```
Docker push anoushkagupta/mnisttrain:v3
```

```python
    dataset1 = datasets.MNIST('../data', train=True, download=True,
                              transform=transform)
    dataset2 = datasets.MNIST('../data', train=False,
                              transform=transform)
    train_loader = torch.utils.data.DataLoader(dataset1,**train_kwargs)
    test_loader = torch.utils.data.DataLoader(dataset2, **test_kwargs)

    model = Net().to(device)
    optimizer = optim.Adadelta(model.parameters(), lr=args.lr)

    scheduler = StepLR(optimizer, step_size=1, gamma=args.gamma)
    for epoch in range(1, args.epochs + 1):
        train(args, model, device, train_loader, optimizer, epoch)
        #test(model, device, test_loader)
        scheduler.step()

    if args.save_model:
        print("Saving model!")
        torch.save(model.state_dict(), "/mnt/ag8733_model.pth")
```

Figure: changes made to the train.py file to just perform training and save weights

RUNNING THE TRAINING JOB ON GCP:

1. I used train.yaml to start a job which makes use of the persistent storage which was created earlier.
2. Use command **kubectl -f apply train.yaml** to run, which creates the job in the K8 cluster.
3. This job uses the image I had pushed to dockerhub.
4. Once this job finishes, the container dies but as weights are saved in persistent storage they persist.



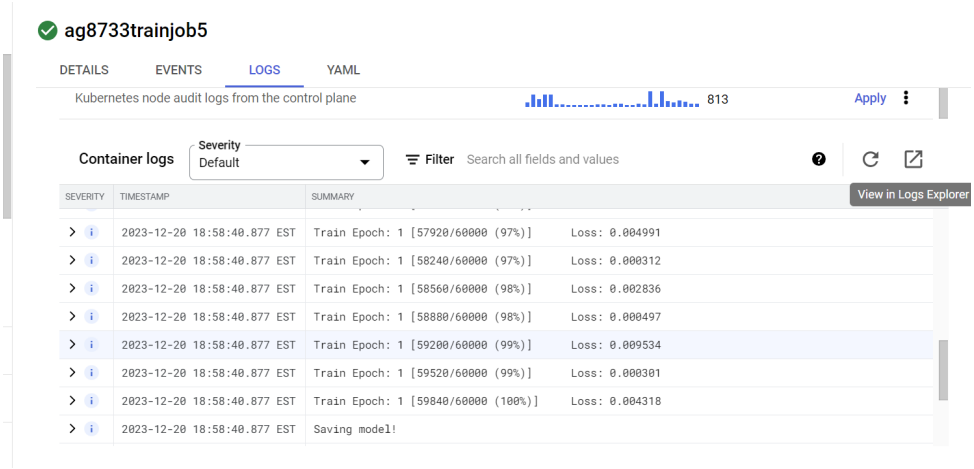


Figure: Training Logs

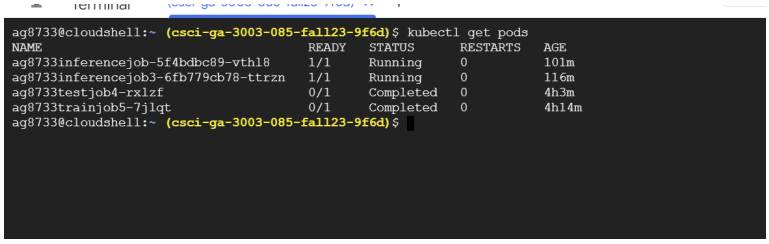


Figure: kubectl get pods shows the train job has been completed

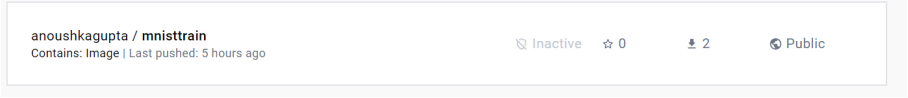


Figure: The Dockerhub repository for the training image

6. INFERENCE

For inference i used the flask app I created for the Final project and use the same templates for deploying the inference service of HW-4.

The Flask app uses the weights stored in persistent storage to carry out inference and this is provisioned using the infer.yaml file.

I then create an endpoint by using service.yaml.

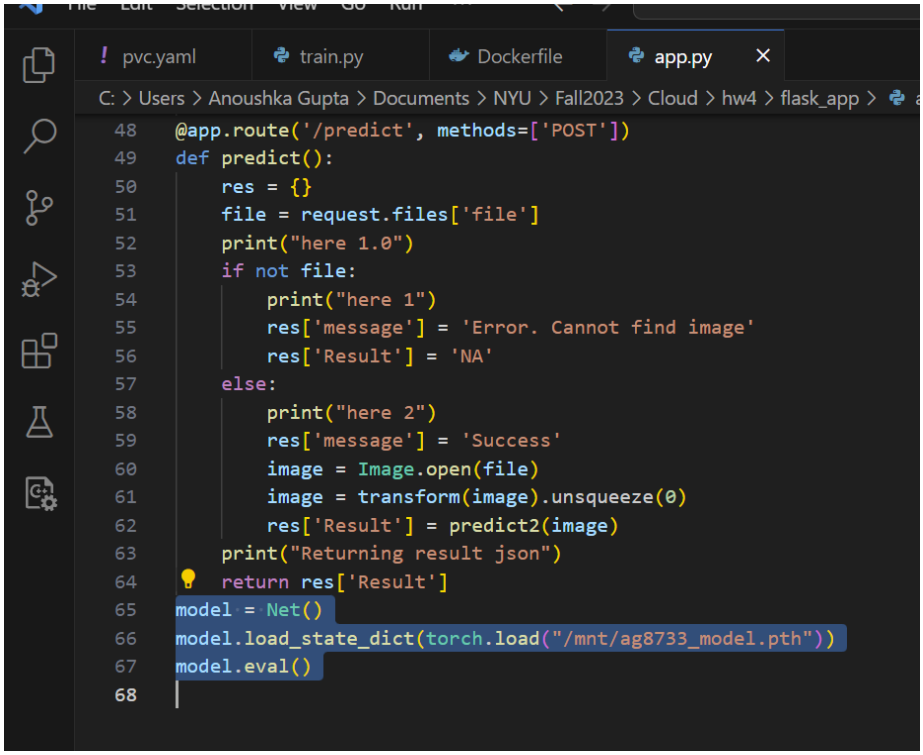


Figure: Flask app which uses the weights in the persistent storage

The Dockerfile just runs this flask application.
The inference.yaml runs this on a pod. The name of the job is ag8733inferencejob.

```
ag8733@cloudshell:~ (csci-ga-3003-085-fall23-9f6d)$ kubectl get pods
NAME                                READY   STATUS        RESTARTS   AGE
ag8733inferencejob-5f4bdbc89-vth18  1/1     Running       0          109m
ag8733inferencejob3-6fb779cb78-dr86b  0/1     Pending       0          14s
ag8733inferencejob3-6fb779cb78-ttrzn  1/1     Terminating   0          123m
ag8733testjob4-rxlzf                0/1     Completed     0          4h11m
ag8733trainjob5-7jlqt               0/1     Completed     0          4h21m
ag8733@cloudshell:~ (csci-ga-3003-085-fall23-9f6d)$
```

I then use service.yaml to create an end point which listens on port 8000.
The service.yaml is run using **kubectl -f apply service.yaml** command.

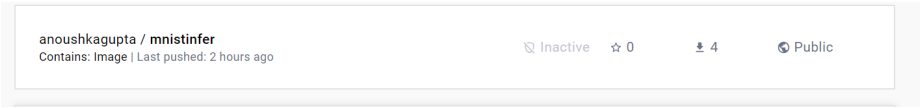


Figure: The DockerHub repository for the inference image

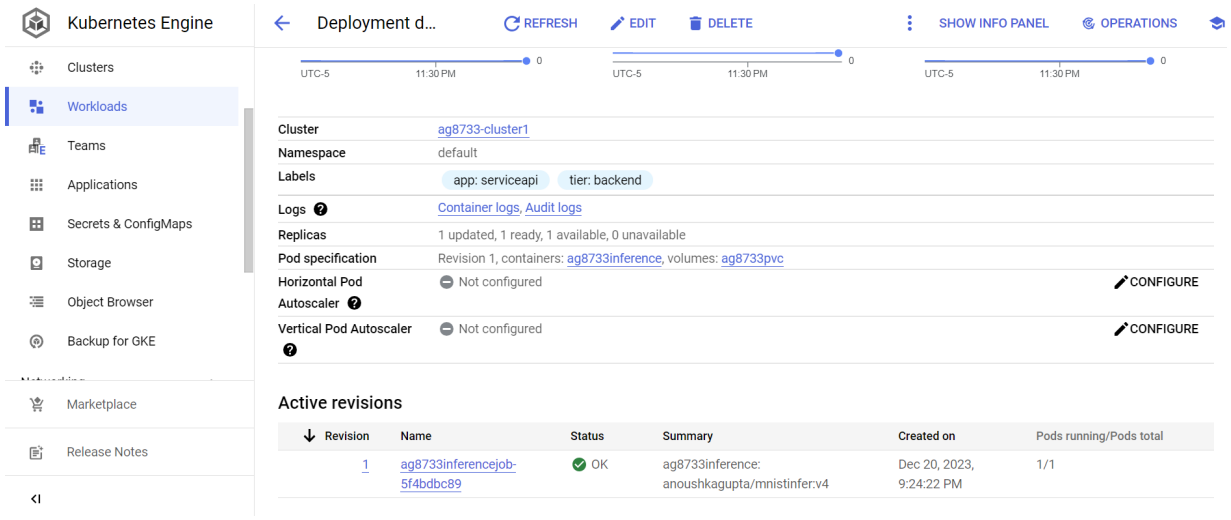


Figure: Dashboard of the inference job, We can see the volume that is mounted the no of pods running, namespace, cluster etc.

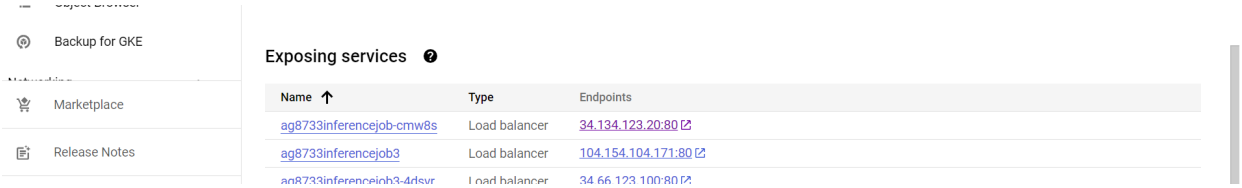
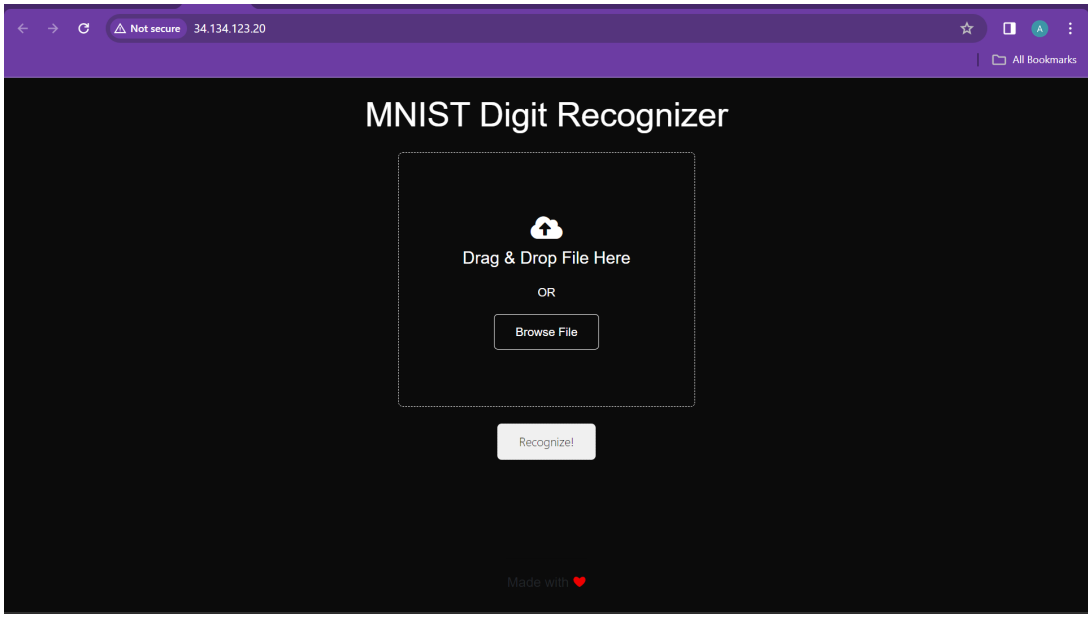


Figure: The exposed services with this inference pods

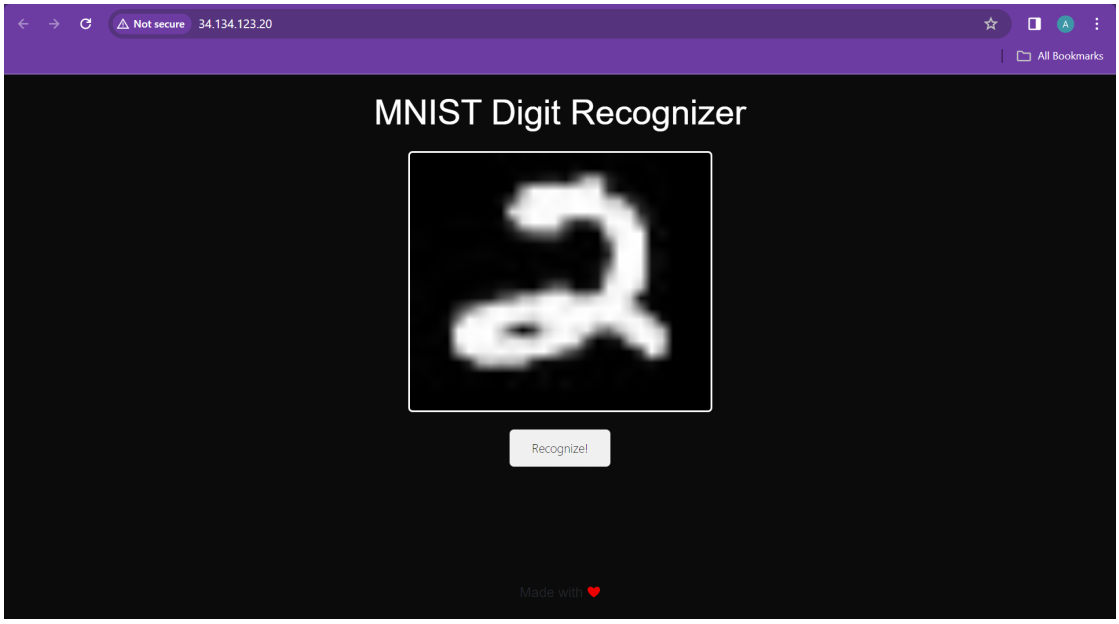
7. DEMO

The end point of the service is http://34.134.123.20/ where you can upload a file and it shows the predicted digit.

UI of the flask app-

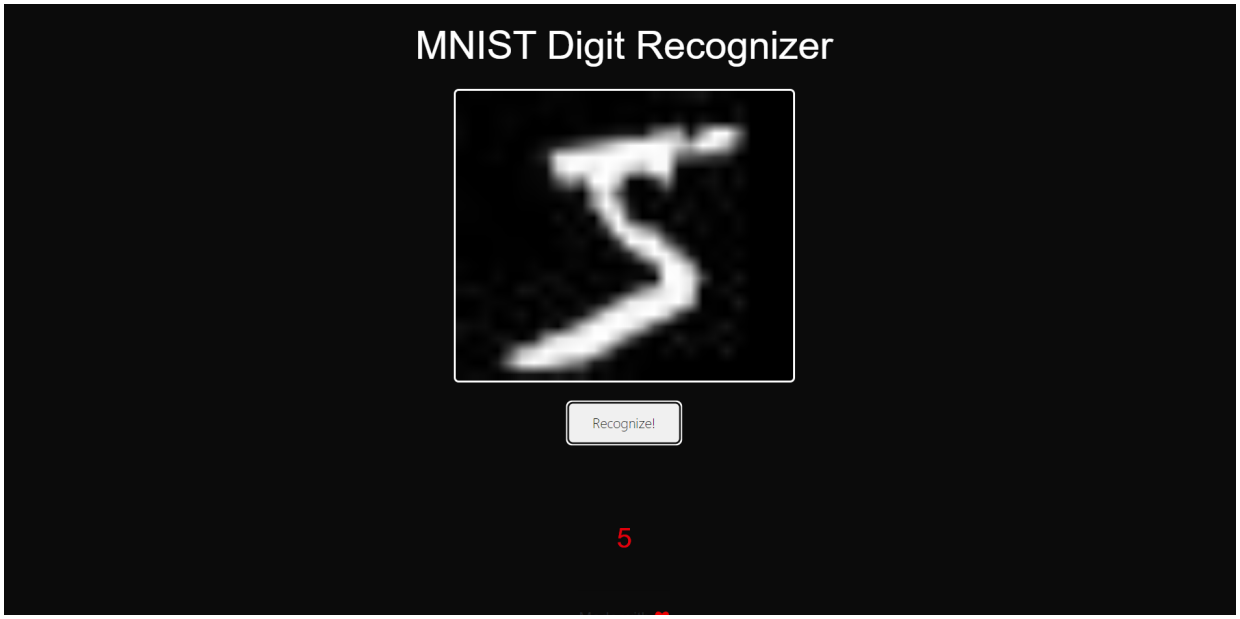


Uploading a MNIST dataset image- Click on the button Recognize to generate results.



The predicted result can be seen on the screen at the bottom.

Trying with another example-



8. CONCLUSION & LEARNINGS

This was a very good deep down into training and saving weights in persistent storage. Some challenges that i encountered included-

1) Setting up a persistent storage and ensuring the weights are actually being stored in the mounted volume. I wasn't specifying the correct paths while storing the model weights and hence they were not being saved on the persistent volume and the weights would die when the training container ended.
2) Requesting Pods. There was a lot of traffic and it was taking a very long time before pods were being assigned to my autopilot cluster and this significantly slowed down my progress.

Learnings-

1) Setting up machine learning training jobs using containers and Kubernetes on GCP.

2) Command line understanding of kubectl and associated commands like apply, get pods, get service etc.

In conclusion I did the following things-

- Created a persistent storage volume claim using pvc.yaml
- Created a container for the training part of code, pushed the image to docker hub, wrote the train.yaml script and then ran a job for training which saves model weights.
- Created a flask app which uses the weights stored in persistent storage for inference.
- Created a deployment or inference.yaml to run the flask application on a pod.
- Expose the endpoint for communication using service.yaml.

REFERENCES

1. Understanding Kubernetes concepts (https://platform9.com/blog/kubernetes-enterprise-chapter-2-kubernetes-architecture-concepts/)
2.Docker Documentation (https://docs.docker.com/)
3. Deploying an app on a GKE cluster (https://cloud.google.com/kubernetes-engine/docs/deploy-app-cluster)
4. Pytorch MNIST example (https://github.com/pytorch/examples/tree/main/mnist)
5. Persistent Volume on GCP (https://cloud.google.com/kubernetes-engine/docs/concepts/persistent-volumes)
6. YAML files for Kubernetes (https://www.cncf.io/blog/2022/03/03/how-to-write-yaml-file-for-kubernetes/)