Name: Tai Wan Kim
Net ID: tk3510
University ID: N10254919
Course: Cloud and Machine Learning
Date: November 21st, 2025

HW4: AI Service in Container

I. Introduction

In this assignment, we train a model and deploy a Flask app that loads the trained model for prediction. Training and inference/app are containerized and managed by a Kubernetes cluster on GCP (GKE). We create a persistent volume that is shared between the training and inference containers so that model weights can be saved by training and later loaded by the app.

II. Model and Dataset

As in assignment 3, I train a simple CNN on MNIST. The Modified National Institute of Standards and Technology (MNIST) dataset is a collection of handwritten digits commonly used to train Optical Character Recognition (OCR) models [1]. The model has the following architecture:

> Input (1×28×28) → Conv(32, 3×3) + ReLU → Conv(64, 3×3) + ReLU → MaxPool(2×2) → Dropout(0.25) → Flatten → FC(128) + ReLU → Dropout(0.5) → FC(10) → Log-Softmax

The code has been adapted from an example provided in PyTorch's GitHub repository [2].

III. Workflow

Here, I detail the steps I took to set up and complete this assignment.

a. Implement Model Training and Flask App

First, I implemented model training and a Flask app. For training, we download the dataset, train the model, and save the weights to the artifacts directory. Because we are running on CPU, we only train for one epoch; the accuracy is still reasonably high for MNIST. Later, the Flask application reads the model weights from the artifacts directory to serve inference. The Flask app exposes an endpoint at the URL "/predict/<testset-id>". Given a test-set sample ID, it predicts the sample's class and returns the result to the user.

b. Build Images

I wrote two Dockerfiles to containerize training and inference. Both are based on the Python slim base image, install the required dependencies, and define entry points that train the model/start the Flask app. The images are pushed to Docker Hub so that they are accessible by the GKE cluster.

c. GKE Setup

Next, I created a GKE cluster and wrote the Kubernetes YAML files. First, I defined a PVC (PersistentVolumeClaim) in pvc.yaml to request persistent storage. This PVC is bound to a PersistentVolume so that any pods that mount it share the same storage. Then, I started a training Job

(train-job.yaml). Once training finishes and the model weights have been written to the shared volume, I start the Flask app via a Deployment (app-deploy.yaml). Finally, I expose the Flask app externally by creating a Service that assigns an external IP and port (app-service.yaml).

d. Frontend Setup

Once the Flask app has been exposed, HTTP requests can be received at its endpoint. We can use curl to test that the API is working correctly. I also implemented a simple web UI that allows users to send requests to the API endpoint. However, the UI only works when CORS is disabled on the client side (use chrome extensions), since the Flask backend does not set CORS headers.

IV. Service Reliability

This project relies on native Kubernetes mechanisms for reliability.

For training, I use a Kubernetes Job with backoffLimit set to 1. If the training pod fails once, the Job controller automatically creates a new pod and retries the run. Once training finishes successfully, the pod transitions to Completed, while the model weights remain safely stored on the PVC.

For inference, the Flask API is managed by a Deployment. The Deployment keeps the desired number of replicas running, and if a pod crashes or its node becomes unavailable, Kubernetes automatically recreates the pod on a healthy node. The Service in front of the Deployment provides a stable endpoint, so clients do not need to be aware of pod restarts or rescheduling.

V. Challenges

While everything ultimately worked as intended, I encountered a few practical challenges along the way.

a. Cross-architecture builds

Because I built the images on my local ARM-based Mac, I had to ensure they would run on Linux/amd64 nodes in GKE. This required using docker buildx with --platform linux/amd64 so that the resulting images matched the cluster's architecture. Getting this right was important; otherwise, pods would fail to start with "exec format" errors due to architecture mismatch.

b. Shared volume design

There was also a design choice around what data should be shared between training and inference. I decided that sharing only the trained model weights via a PVC was sufficient, and that each container could independently download and load the MNIST dataset as needed. This keeps the shared volume small and focused on artifacts that truly need to persist across pods (the weights), while allowing the training and app containers to remain relatively self-contained.

VI. Conclusion

For this assignment, I containerized both training and inference and used Kubernetes to deploy a reliable, user-facing AI service. I familiarized myself with the training–deployment–service pipeline and with modularization strategies such as using persistent storage for shared artifacts. I also came to

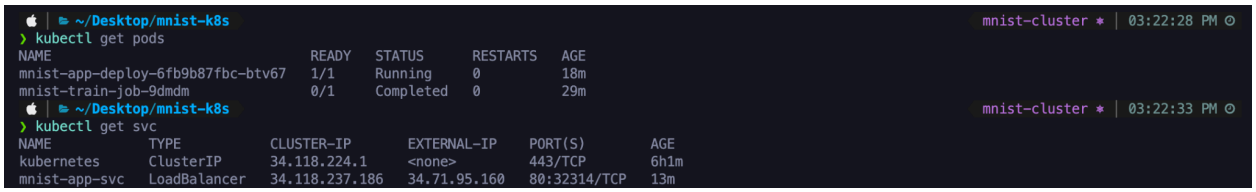
appreciate the benefits of tools like Docker and Kubernetes, which make the service more reliable and easier to manage.

VII. References

1. L. Deng, "The MNIST Database of Handwritten Digit Images for Machine Learning Research [Best of the Web]," in IEEE Signal Processing Magazine, vol. 29, no. 6, pp. 141-142, Nov. 2012, doi: 10.1109/MSP.2012.2211477.
2. PyTorch Foundation, “PyTorch Examples.” [Online]. Available: https://docs.pytorch.org/examples/

VIII. Supplements

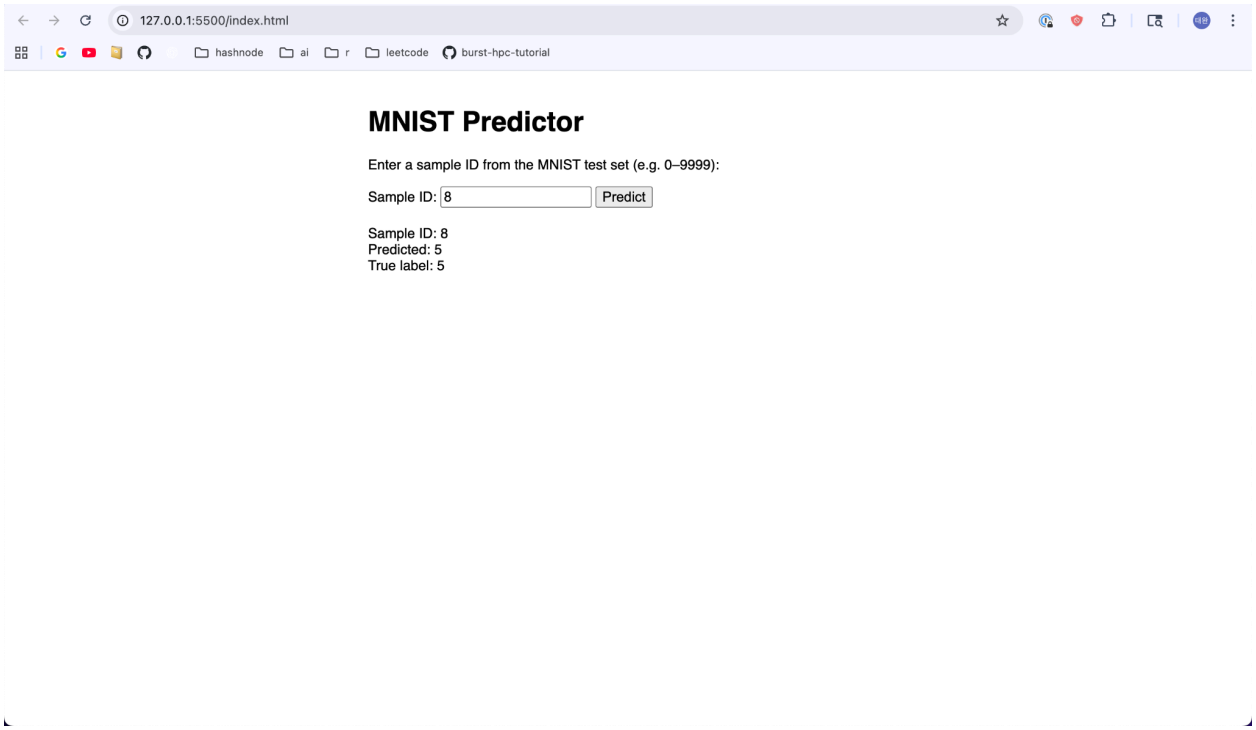
a. Training, Deployment, and Service



b. Flask logs

c. Curl

d. Frontend UI


127.0.0.1:5500/index.html
hashnode
ai
r
leetcode
burst-hpc-tutorial
MNIST Predictor
Enter a sample ID from the MNIST test set (e.g. 0–9999):
Sample ID: 8
Predict
Sample ID: 8
Predicted: 5
True label: 5