Homework 4: AI Service in Container

1. Project Overview

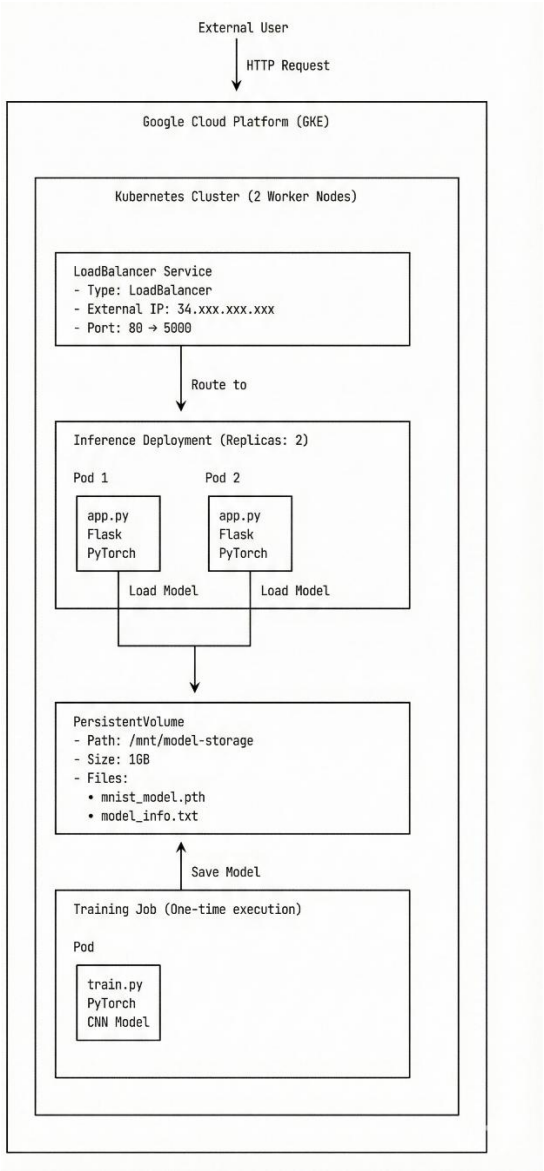
1.1. Objective

This project implements a complete machine learning system for handwritten digit recognition using containerization and orchestration technologies. The system is deployed on GCP using GKE, implementing the full workflow from model training to production deployment.

1.2. System Architecture

The system consists of two main phases, training phase and inference phase. During training phase, a k8s job runs a containerized training script, training a CNN model on the MNIST dataset. And it will save the trained model to a Persistent Volume. During inference Phase, a k8s deployment manages inference service pods (2 replicas). It will load the trained model from the Persistent Volume, and provides a web interface and REST API for digit recognition.

1.3. System Architecture Diagram



2. Complete Pipeline from Code to Cloud Deployment

2.1. Layer 1: Code Layer

The foundation of the system are two Python scripts that handle model training and inference. The training script train.py defines a CNN architecture MNISTNet, which consists of two convolutional blocks followed by fully connected layers. After training on the MNIST dataset for 5 epochs, the script saves the complete model to the path /mnt/model_storage/mnist_model.pth. This path is crucial because it represents the mount point where the persistent volume will be attached in the k8s environment.

The inference script app.py implements a Flask web service that provides both a user interface and REST API endpoints. When the service starts, it loads the saved model from the same path /mnt/model_storage/mnist_model.pth, sets the model to evaluation mode with model.eval(). The Flask application provides three endpoints: a root endpoint / that serves an HTML interface, a /predict endpoint that accepts image uploads and returns JSON predictions, and a /health endpoint used by k8s for health checking.

The key connection between these two scripts is the shared storage path. Both scripts reference /mnt/model_storage/ as if it were a regular directory, but this path will later be mapped to a k8s Persistent Volume, enabling the trained model to persist beyond the lifecycle of individual containers and be accessible to multiple pods simultaneously.

2.2. Layer 2: Docker Containerization to Packaging the Environment

Docker transforms our Python scripts into containers. The Dockerfile serves as a recipe that Docker follows step-by-step to create an image containing everything needed to run our application: the base operating system, Python runtime, required libraries, and our code.

For the training module, the Dockerfile begins with FROM python:3.9-slim, which establishes a minimal Python 3.9 environment as the foundation. The WORKDIR /app instruction sets the working directory inside the container. The build process then copies requirements.txt and executes RUN pip install -r requirements.txt to install PyTorch and other dependencies. Finally, it copies the train.py script into the container.

Similarly, the inference Dockerfile follows the same pattern but copies app.py instead and includes EXPOSE 5000 to declare that the Flask application listens on port 5000. Its CMD instruction CMD ["python", "app.py"] starts the Flask web server when the container launches. When we execute docker build -t gcr.io/cloudml01/mnist-training:latest ., Docker reads the Dockerfile line by line, creating layers. The image name encodes the registry location (gcr.io), project ID (cloudml01), image name, and tag.

2.3. Layer 3: Kubernetes Orchestration

Kubernetes orchestrates container execution through YAML configuration files. The PersistentVolumeClaim (pvc.yaml) requests 1GB storage. When applied, Kubernetes provisions a Google Cloud persistent disk and binds it to model-storage-pvc, creating storage that persists independently of containers.

The training Job (training-job.yaml) specifies image: gcr.io/cloudml01/mnist-training:latest and mounts the PVC at /mnt/model-storage. When applied, Kubernetes pulls the image, creates a pod, mounts the volume, and executes the Dockerfile's CMD (python train.py). The script runs, saves the model to the mounted volume, and the Job completes.

The inference Deployment (inference-deployment.yaml) maintains 2 pod replicas continuously.

Each pod runs the inference image with the same volume mounted as read only. The Deployment includes livenessProbe (HTTP GET to /health every 10 seconds and restart container if fails 3 times) and readinessProbe (remove from load balancer if unhealthy, but don't restart). These probes enable automatic self-healing.

The LoadBalancer Service (inference-service.yaml) uses selector app: mnist, component: inference to route traffic to inference pods. GCP automatically provisions a cloud load balancer with an external IP. The configuration port: 80, targetPort: 5000 forwards external requests on port 80 to Flask listening on port 5000 inside containers.

The architecture's elegance is in its layered separation: code uses standard file paths, Docker packages the code without Kubernetes knowledge, and Kubernetes orchestrates everything by connecting volumes and network traffic. Each layer can be modified independently as long as interfaces remain consistent.

3. Self-Healing Mechanisms

Kubernetes provides multiple layers of self-healing capabilities that automatically detect and recover from failures without manual intervention. Our MNIST inference service demonstrates four key self-healing mechanisms working together to ensure high availability.

3.1. Liveness Probe

```yaml
livenessProbe:
  httpGet:
    path: /health
    port: 5000
  initialDelaySeconds: 30
  periodSeconds: 10
  timeoutSeconds: 5
  failureThreshold: 3
```

The liveness probe monitors whether containers are still running properly. This configuration tells Kubernetes to wait 30 seconds after container startup, then send HTTP GET requests to the /health endpoint every 10 seconds. If the endpoint fails to respond within 5 seconds, it counts as a failure. After 3 consecutive failures, Kubernetes automatically kills and restarts the container. This mechanism handles scenarios where the application process becomes unresponsive or enters a deadlocked state - conditions where the container is technically running but unable to serve requests. The /health endpoint in our Flask application returns a simple JSON response indicating service status, allowing Kubernetes to verify that the application is functioning correctly.

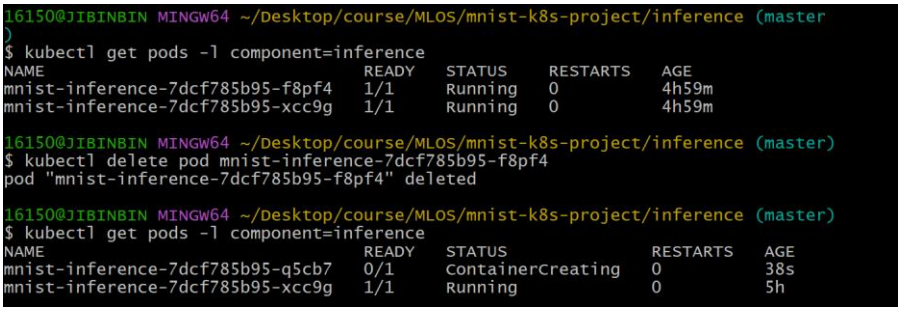
3.2. Readiness Probe

```yaml
readinessProbe:
  httpGet:
    path: /health
    port: 5000
  initialDelaySeconds: 10
  periodSeconds: 5
  timeoutSeconds: 3
  failureThreshold: 3
```

While liveness probes determine if a container should be restarted, readiness probes control whether a pod should receive traffic. The readiness probe checks more frequently, like every 5 seconds, with a shorter timeout. When a pod fails readiness checks, Kubernetes removes it from the Service's endpoint list, preventing new requests from being routed to that pod. Critically, the pod is not restarted - it remains running while marked as "not ready." This distinction is important because some issues are temporary like loading a large model into memory and don't require a full container restart. In our deployment, the readiness probe ensures that pods only receive traffic after successfully loading the model, preventing errors during the startup phase when the model is being loaded from the persistent volume.

3.3. ReplicaSet

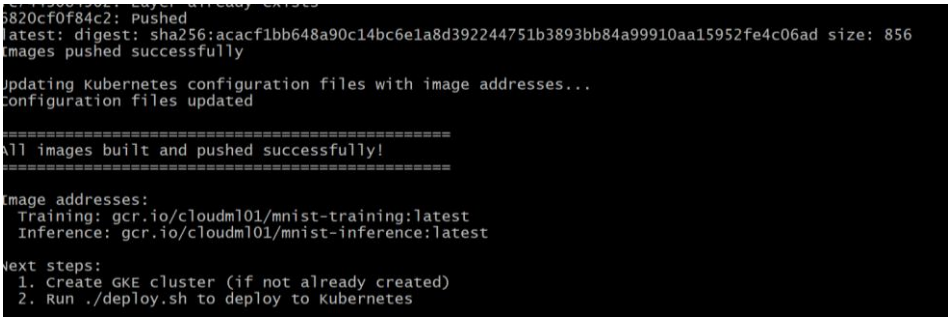
Here I specifie “replicas: 2”, which creates a ReplicaSet that continuously ensures exactly 2 pods are running. If a pod crashes, gets deleted, or its node fails, the ReplicaSet controller immediately creates a replacement pod. This happens automatically without any manual intervention. We tested this mechanism by manually deleting one of the inference pods:



4. Deployment Process and Results

4.1. Automated Deployment Scripts

The build.sh script automates the containerization phase. It authenticates Docker with Google Container Registry, builds both training and inference images from their respective Dockerfiles, pushes the images to GCR, and automatically updates the Kubernetes YAML files to reference the correct image paths with our project ID. This automation is essential because manually building and pushing images, then editing multiple YAML files, is error-prone and time-consuming.



The deploy.sh script orchestrates the Kubernetes deployment. It sequentially applies the PVC configuration, launches the training Job and waits for completion, deploys the inference Deployment and Service, and finally displays the external IP address for accessing the service. The script includes

wait conditions to ensure each step completes successfully before proceeding to the next, preventing race conditions where the inference service might try to load a model that hasn't been trained yet.

```
Epoch [5/5]
  Batch [100/469], Loss: 0.0402, Acc: 98.80%
  Batch [200/469], Loss: 0.0437, Acc: 98.68%
  Batch [300/469], Loss: 0.0446, Acc: 98.61%
  Batch [400/469], Loss: 0.0457, Acc: 98.59%
Train - Loss: 0.0454, Acc: 98.59%
Test - Loss: 0.0345, Acc: 98.95%

================================================
Final Model Evaluation
================================================
Test Loss: 0.0345
Test Accuracy: 98.95%

Saving model to: /mnt/model-storage/mnist_model.pth
Model saved successfully!
Training completed! Model saved successfully.
================================================

[3/4] Deploying inference service...
deployment.apps/mnist-inference created
service/mnist-inference-service created
Inference service deployed

Waiting for inference service to be ready...
error: timed out waiting for the condition on deployments/mnist-infer
ence
```

```
#10 [6/6] COPY app.py .
#10 DONE 0.0s

#11 exporting to image
#11 exporting layers 0.0s done
#11 exporting manifest sha256:1c5822db7d1b8a0b322895f3df5d27dd1308fbf0fd85ef15
1d9196c74e6b4073 done
#11 exporting config sha256:018b9e37576ce464105fcfbcd9e730383a36b7a188aa5687c8
9129d0309b3ffd done
#11 exporting attestation manifest sha256:532bedc0a8b649e4376e01fbc655f2a6acdf
fc08fc697a88f0fbd404f5c4b27d 0.0s done
#11 exporting manifest list sha256:78d975c5fd5d5019fa494ecb0021cd42f1dd68cb317
b2faa98373e8f26a6812e done
#11 naming to gcr.io/cloudml01/mnist-inference:latest done
#11 unpacking to gcr.io/cloudml01/mnist-inference:latest 0.0s done
#11 DONE 0.1s
```

4.2. User Interaction

With the service deployed, we accessed the web interface at http://35.223.215.201