Containerized DL Model Training and Inference on GKE: A Kubernetes-based Approach

Abhishek Chigurupati (ac11950)

1 Introduction

This project focuses on deploying a deep learning (DL) model training and inference pipeline on Google Kubernetes Engine (GKE) using Docker and Kubernetes. The pipeline involves training a convolutional neural network (CNN) on the MNIST dataset and hosting an inference service for real-time digit classification via a web-based interface.

The training process leverages a Docker container with a PyTorch-based model, deployed as a Kubernetes Job. The trained model is stored on a persistent volume to ensure availability during inference. The inference service, hosted using Flask, is exposed through a LoadBalancer to allow public access for user interaction.

By utilizing GKE for orchestration and container management, this project demonstrates a scalable, cloud-native approach to machine learning deployment, integrating training and inference into a single workflow.

2 Problem Statement

Deploying deep learning (DL) models in a scalable and accessible manner is a critical challenge in modern machine learning workflows. This project addresses the need for an end-to-end pipeline that efficiently trains and serves a DL model within a cloud environment.

The primary objective is to develop a robust and reproducible workflow that performs model training and inference on Google Kubernetes Engine (GKE) using Docker and Kubernetes. The pipeline will include the following key components:

- Containerized training of a convolutional neural network (CNN) on the MNIST dataset.
- Persistent model storage using Kubernetes PersistentVolume (PV) and PersistentVolumeClaim (PVC).
- Containerized inference service hosted via Flask, providing real-time digit classification.
- Public exposure of the inference service through a LoadBalancer, allowing user interaction via a web interface.

The solution will leverage GKE's orchestration capabilities to automate container deployment, scalability, and fault tolerance. The main challenge lies in integrating training and inference pipelines within a single Kubernetes cluster while ensuring persistent model access and seamless user interaction.

3 Project Design

3.1 System Architecture

The system architecture follows a microservices approach, with distinct components for training and inference, connected through shared persistent storage. The architecture consists of three main components:

1. **Data Preparation and Training Component**: Responsible for downloading the MNIST dataset and training the CNN model.

2. **Persistent Storage Layer**: Acts as the bridge between training and inference components.

3. **Inference Service Component**: Handles model loading, image processing, and serving predictions.

Each component is deployed as a separate Kubernetes resource, enabling independent scaling and lifecycle management while maintaining connectivity through the shared persistent storage.

3.2 Training Pipeline Design

The training pipeline consists of the following components:

1. **Data Preparation**: A dedicated pod (mnist-downloader) that fetches the MNIST dataset from the PyTorch examples repository and stores it on a shared volume.

2. **Model Training**: A Kubernetes Job that:
   - Runs a PyTorch-based CNN training script in a container
   - Accesses the MNIST dataset from the shared volume
   - Trains the model for one epoch with a batch size of 64
   - Saves the trained model to the persistent volume
   - Terminates upon successful completion

3. **Persistent Storage**: A PersistentVolumeClaim that serves as the bridge between training and inference, storing the trained model file for later use.

3.3 Inference Pipeline Design

The inference pipeline consists of the following components:

1. **Inference Service**: A Kubernetes Deployment that:
   - Runs a Flask application in a container
   - Loads the trained model from the persistent volume
   - Preprocesses uploaded images to match the MNIST format (28x28 grayscale)
   - Performs inference using the loaded model
   - Returns predictions via a REST API

2. **Web Interface**: A simple HTML form that:
   - Allows users to upload handwritten digit images
   - Submits the images to the inference service
   - Displays the predicted digit

3. **LoadBalancer Service**: A Kubernetes Service of type LoadBalancer that:
   - Exposes the inference service to external traffic
   - Provides a stable external IP address
   - Routes incoming requests to the appropriate pods

3.4 Data Flow

The data flow through the system follows these steps:

1. The MNIST dataset is downloaded to a shared volume by the mnist-downloader pod.
2. The training job reads the dataset, trains the model, and saves it to the persistent volume.
3. The inference service loads the trained model from the persistent volume.
4. Users upload handwritten digit images through the web interface.
5. The inference service processes the images and returns predictions.
6. The web interface displays the predicted digits to the users.

This design ensures a clean separation of concerns while maintaining connectivity through the persistent volume, allowing for independent scaling and maintenance of each component.

4 Methodology

4.1 Environment Setup

The environment setup involved creating a GKE cluster with necessary configurations for running containerized machine learning workloads. The following steps were taken:

1. Created a GKE cluster with appropriate node configurations to support ML workloads.
2. Configured kubectl to communicate with the GKE cluster.
3. Set up a project directory structure with separate folders for training and inference services.

4.2 Persistent Storage Configuration

A critical component of the architecture was the persistent storage setup that allows model artifacts to be shared between training and inference services:

1. Created a Kubernetes PersistentVolumeClaim with 20Gi capacity to store the trained model.
2. Applied the PVC configuration using kubectl, as shown in Figure 1:

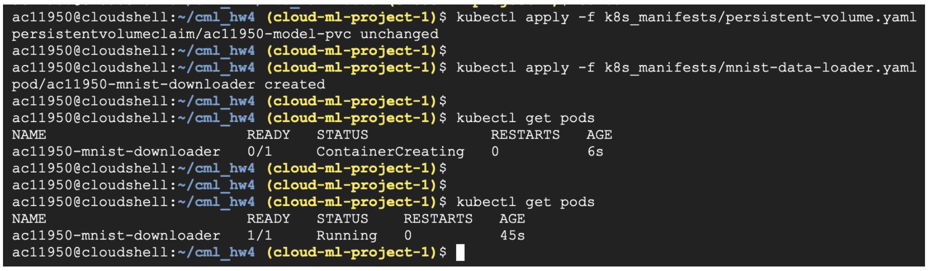


Figure 1: Setting up Persistent Volume Claim for model storage

4.3 Docker Image Creation

Two separate Docker images were created for the training and inference components:

4.3.1 Training Image

The training image was built with PyTorch and necessary dependencies:

1. Created a Dockerfile specifying Python 3.10-slim as the base image.
2. Added the training script and requirements file.
3. Built and pushed the image to DockerHub as shown in Figure 2:

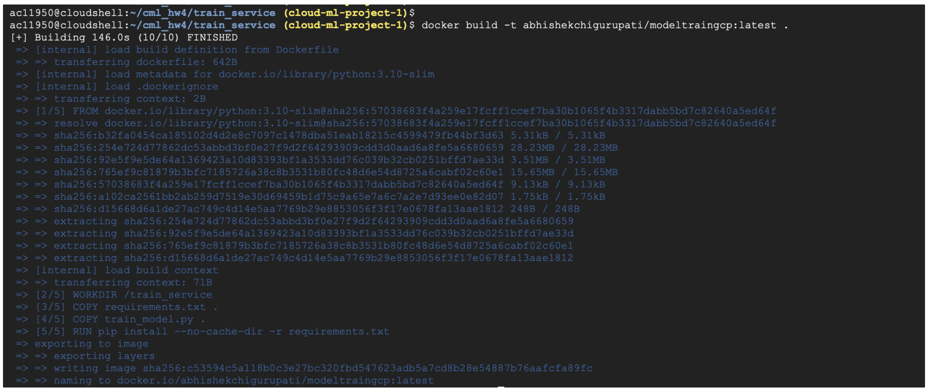


Figure 2: Building the training image with Docker

4. Successfully pushed the training image to DockerHub repository, as verified in Figure 3:

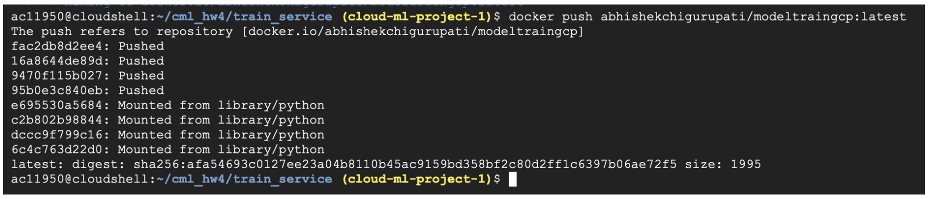


Figure 3: Pushing the training image to DockerHub

4.3.2 Inference Image

The inference service was containerized with a Flask application:

1. Created a Dockerfile with Python 3.10-slim as the base image.
2. Added the Flask application code, model loading logic, and an HTML template.
3. Built and pushed the image to DockerHub, as shown in Figure 4:

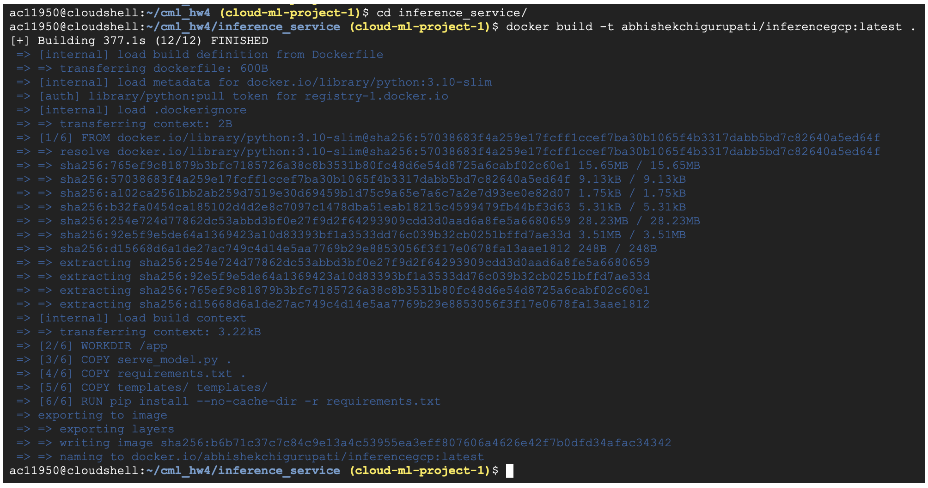


Figure 4: Building the inference service Docker image

4. Successfully pushed the inference image to DockerHub, as demonstrated in Figure 5:

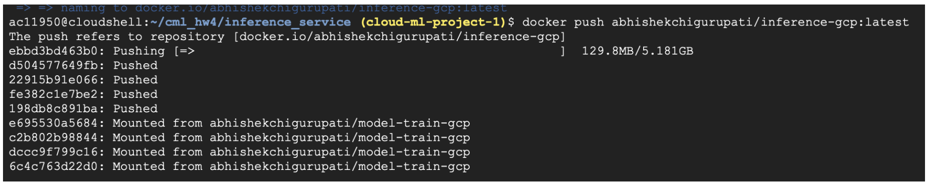


Figure 5: Pushing the inference image to DockerHub

4.4 MNIST Data Preparation

To provide the MNIST dataset for training, a separate data loader pod was created:

1. Configured a Kubernetes manifest for a pod that downloads the MNIST dataset.
2. Applied the configuration and verified the pod was running, as shown in Figure 6:

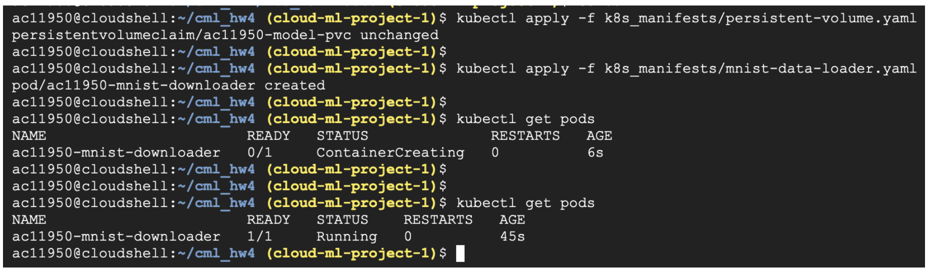


Figure 6: MNIST data downloader pod deployment

5 Technologies and Components

5.1 Google Kubernetes Engine (GKE)

Google Kubernetes Engine (GKE) is a managed Kubernetes service that simplifies the deployment, management, and scaling of containerized applications on Google Cloud. For this project, GKE was selected as the deployment platform due to several key advantages:

1. **Cluster Types**: GKE offers two primary cluster types:
   - **Standard clusters**: Provide more control over the cluster and node configuration. In this project, a Standard cluster was used to have fine-grained control over resources and configurations.
   - **Autopilot clusters**: Fully managed and optimized by Google Cloud, reducing operational overhead but with less customization.
2. **Scalability**: GKE can automatically scale both the cluster and the individual workloads based on demand.
3. **Integration**: Seamless integration with other Google Cloud services like Cloud Storage, Cloud IAM, and Cloud Monitoring.
4. **Cost Efficiency**: The ability to use preemptible VMs and automatic bin-packing of containers reduces operational costs.

The GKE cluster for this project was configured with 3 nodes in the us-central1-c region, each with 5.1 GB of memory, providing sufficient resources for both training and inference workloads, as shown in Figure 7.

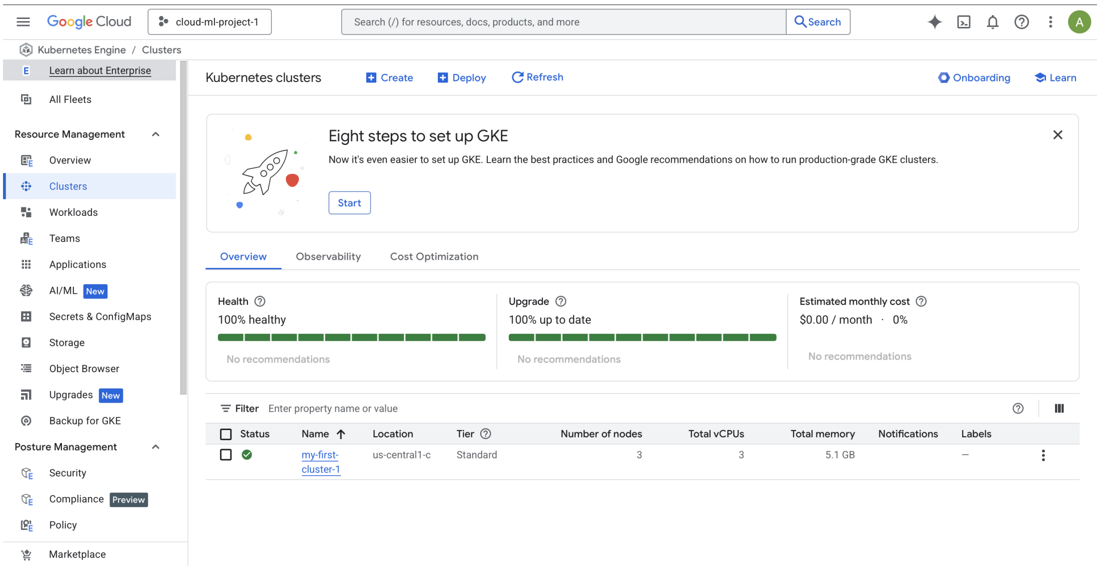


Figure 7: Google Kubernetes Engine console showing the cluster configuration with 3 nodes of Standard tier

5.2 Kubernetes Architecture

This project heavily relies on Kubernetes as the orchestration platform for containerized machine learning workloads. Kubernetes provides several key benefits in the ML workflow:

1. **Workload Orchestration**: Kubernetes schedules containers across a cluster of machines, ensuring efficient resource utilization and high availability.
2. **Declarative Configuration**: The entire infrastructure is defined as code through YAML manifests, enabling reproducible deployments.
3. **Service Discovery**: Kubernetes services enable microservices to communicate with each other through a stable network endpoint.
4. **Scaling**: Horizontal pod autoscaling can adjust the number of inference pods based on CPU usage or custom metrics.

5.3 Persistent Volume Claims (PVC)

PVCs are essential for managing stateful workloads in Kubernetes, particularly for machine learning pipelines where model artifacts need to be shared between training and inference services:

1. **Data Persistence**: Unlike ephemeral pod storage, PVCs provide durable storage that survives pod restarts.
2. **Shared Access**: Both training jobs and inference deployments can mount the same volume, enabling model sharing.
3. **Storage Classes**: GKE provides different storage class options (Standard, SSD, etc.) to match performance requirements.
4. **Dynamic Provisioning**: Kubernetes can automatically provision underlying storage resources based on the PVC request.

The PVC in this project is configured with ReadWriteMany access mode, allowing multiple pods to simultaneously read from and write to the volume, which is critical for the handoff between training and inference components.

5.4 MNIST Dataset

The Modified National Institute of Standards and Technology (MNIST) database is a widely used dataset for training and testing machine learning systems in the field of computer vision. For this project, the MNIST dataset was chosen due to several factors:

1. **Standardization**: MNIST is a well-established benchmark in the machine learning community.
2. **Size and Simplicity**: With 60,000 training images and 10,000 test images of handwritten digits (0-9), the dataset is substantial enough for meaningful training but small enough for rapid iteration.
3. **Accessibility**: The dataset was cloned from the PyTorch examples repository using:

   ```
   git clone https://github.com/pytorch/examples.git
   ```

4. **Preprocessing**: The images are already preprocessed (normalized to 28x28 pixels, centered, and grayscale), reducing the data preparation overhead.

To make the dataset available within the Kubernetes environment, a dedicated pod (mnist-downloader) was created to download and store the data on a shared volume. This approach ensures that the dataset is readily accessible to the training job without having to bundle it within the container image, keeping the images lightweight and deployment efficient.

6 Implementation Details

6.1 Training Job Deployment

The training process was implemented as a Kubernetes Job with the following components:

1. Created a Kubernetes Job manifest specifying the training image, volume mounts, and resource requirements.

2. Deployed the training job to the GKE cluster using kubectl:

```
ac11950@cloudshell:~/cml_hw4 (cloud-ml-project-1)$ kubectl logs job/ac11950-train-job
100.0%
100.0%
100.0%
100.0%
Training started!

Train Epoch: 1 [0/60000 (0%)]   Loss: 2.315593
Train Epoch: 1 [2560/60000 (4%)]        Loss: 1.714868
Train Epoch: 1 [5120/60000 (9%)]        Loss: 1.156443
Train Epoch: 1 [7680/60000 (13%)]       Loss: 0.446512
Train Epoch: 1 [10240/60000 (17%)]      Loss: 0.254767
Train Epoch: 1 [12800/60000 (21%)]      Loss: 0.283148
Train Epoch: 1 [15360/60000 (26%)]      Loss: 0.220180
Train Epoch: 1 [17920/60000 (30%)]      Loss: 0.206010
Train Epoch: 1 [20480/60000 (34%)]      Loss: 0.226763
Train Epoch: 1 [23040/60000 (38%)]      Loss: 0.172745
Train Epoch: 1 [25600/60000 (43%)]      Loss: 0.259373
Train Epoch: 1 [28160/60000 (47%)]      Loss: 0.136659
Train Epoch: 1 [30720/60000 (51%)]      Loss: 0.259224
Train Epoch: 1 [33280/60000 (55%)]      Loss: 0.114079
Train Epoch: 1 [35840/60000 (60%)]      Loss: 0.190651
Train Epoch: 1 [38400/60000 (64%)]      Loss: 0.147105
Train Epoch: 1 [40960/60000 (68%)]      Loss: 0.173118
Train Epoch: 1 [43520/60000 (72%)]      Loss: 0.183478
Train Epoch: 1 [46080/60000 (77%)]      Loss: 0.187034
Train Epoch: 1 [48640/60000 (81%)]      Loss: 0.051830
Train Epoch: 1 [51200/60000 (85%)]      Loss: 0.125717
Train Epoch: 1 [53760/60000 (89%)]      Loss: 0.200703
Train Epoch: 1 [56320/60000 (94%)]      Loss: 0.097123
Train Epoch: 1 [58880/60000 (98%)]      Loss: 0.012073
Training ended!

Done with training!
The model is saved at /mnt/ac11950_model.pt
ac11950@cloudshell:~/cml_hw4 (cloud-ml-project-1)$
```

Figure 8: Deployment of the training job on GKE

3. Monitored the training progress through pod logs, showing the model training with decreasing loss:

```
ac11950@cloudshell:~/cml_hw4 (cloud-ml-project-1)$ kubectl apply -f k8s_manifests/inference-deployment.yaml
deployment.apps/ac11950-inference created
ac11950@cloudshell:~/cml_hw4 (cloud-ml-project-1)$ kubectl get pods
NAME                                 READY   STATUS              RESTARTS   AGE
ac11950-inference-687b797f45-xkvxz   0/1     ContainerCreating   0          6s
ac11950-mnist-downloader             1/1     Running             0          11m
ac11950-train-job-6bbjg              1/1     Running             0          9m56s
ac11950@cloudshell:~/cml_hw4 (cloud-ml-project-1)$
```

Figure 9: Training logs showing model training progress and loss reduction

4. Verified the successful completion of the training job, as shown in Figure 10:

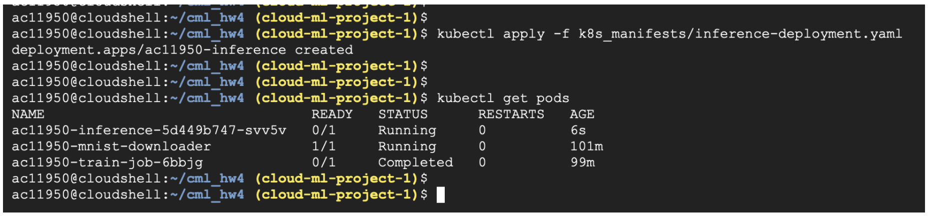
```
abhi@Mac ac11950_cml_hw4 % kubectl get pods

NAME                                 READY   STATUS      RESTARTS   AGE
ac11950-inference-6654f5b8d-xv2kg    1/1     Running     0          3h16m
ac11950-mnist-downloader             1/1     Running     0          6h46m
ac11950-train-job-dpkjs              0/1     Completed   0          6h43m
abhi@Mac ac11950_cml_hw4 %
abhi@Mac ac11950_cml_hw4 %
```

Figure 10: Kubernetes pod status showing completed training job

6.2 Inference Service Deployment

The inference service was deployed as a Kubernetes Deployment with an associated Service:

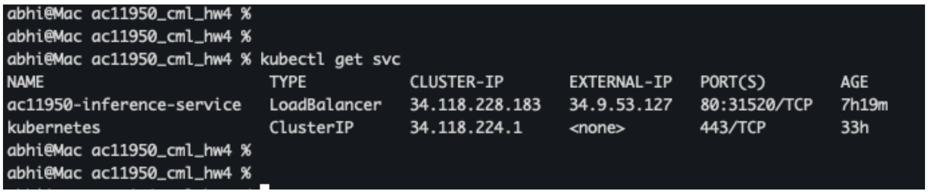
1. Created a Deployment manifest specifying the inference image, volume mounts, and port configurations.
2. Applied the deployment configuration using kubectl:



```
ac11950@cloudshell:~/cml_hw4 (cloud-ml-project-1)$
ac11950@cloudshell:~/cml_hw4 (cloud-ml-project-1)$ kubectl apply -f k8s_manifests/inference-deployment.yaml
deployment.apps/ac11950-inference created
ac11950@cloudshell:~/cml_hw4 (cloud-ml-project-1)$
ac11950@cloudshell:~/cml_hw4 (cloud-ml-project-1)$
ac11950@cloudshell:~/cml_hw4 (cloud-ml-project-1)$ kubectl get pods
NAME                               READY   STATUS      RESTARTS   AGE
ac11950-inference-5d449b747-svv5v  0/1     Running     0          6s
ac11950-mnist-downloader           1/1     Running     0          101m
ac11950-train-job-6bbjg            0/1     Completed   0          99m
ac11950@cloudshell:~/cml_hw4 (cloud-ml-project-1)$
ac11950@cloudshell:~/cml_hw4 (cloud-ml-project-1)$
```

Figure 11: Deploying the inference service using kubectl

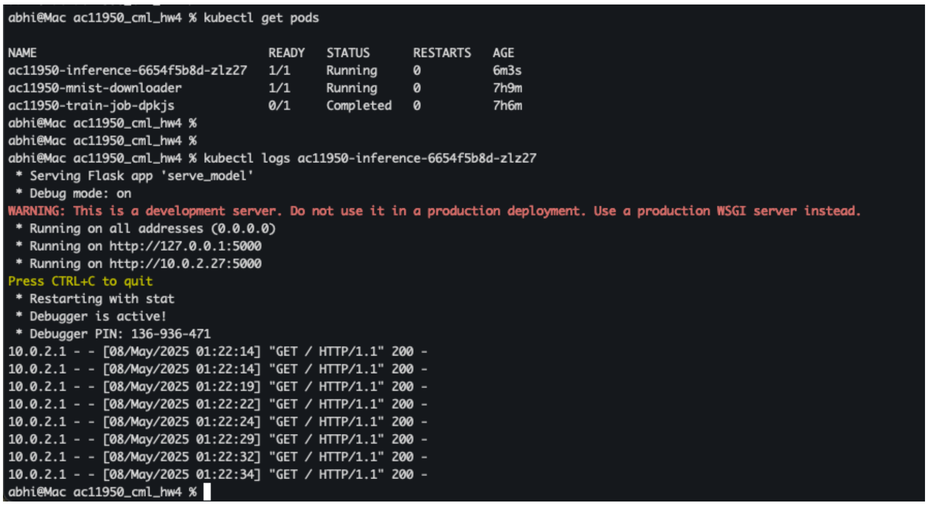
3. Created a Service manifest of type LoadBalancer to expose the inference service.
4. Verified the deployment status and external IP assignment:



```
abhi@Mac ac11950_cml_hw4 %
abhi@Mac ac11950_cml_hw4 %
abhi@Mac ac11950_cml_hw4 % kubectl get svc
NAME                       TYPE           CLUSTER-IP       EXTERNAL-IP   PORT(S)        AGE
ac11950-inference-service  LoadBalancer   34.118.228.183   34.9.53.127   80:31520/TCP   7h19m
kubernetes                 ClusterIP      34.118.224.1     <none>        443/TCP        33h
abhi@Mac ac11950_cml_hw4 %
abhi@Mac ac11950_cml_hw4 %
```

Figure 12: LoadBalancer service exposing the inference application with external IP

5. Monitored the Flask application logs to verify it was serving correctly:



```
abhi@Mac ac11950_cml_hw4 % kubectl get pods

NAME                                READY   STATUS      RESTARTS   AGE
ac11950-inference-6654f5b8d-zlz27   1/1     Running     0          6m3s
ac11950-mnist-downloader            1/1     Running     0          7h9m
ac11950-train-job-dpkjs             0/1     Completed   0          7h6m
abhi@Mac ac11950_cml_hw4 %
abhi@Mac ac11950_cml_hw4 %
abhi@Mac ac11950_cml_hw4 % kubectl logs ac11950-inference-6654f5b8d-zlz27
 * Serving Flask app 'serve_model'
 * Debug mode: on
WARNING: This is a development server. Do not use it in a production deployment. Use a production WSGI server instead.
 * Running on all addresses (0.0.0.0)
 * Running on http://127.0.0.1:5000
 * Running on http://10.0.2.27:5000
Press CTRL+C to quit
 * Restarting with stat
 * Debugger is active!
 * Debugger PIN: 136-936-471
10.0.2.1 - - [08/May/2025 01:22:14] "GET / HTTP/1.1" 200 -
10.0.2.1 - - [08/May/2025 01:22:14] "GET / HTTP/1.1" 200 -
10.0.2.1 - - [08/May/2025 01:22:19] "GET / HTTP/1.1" 200 -
10.0.2.1 - - [08/May/2025 01:22:22] "GET / HTTP/1.1" 200 -
10.0.2.1 - - [08/May/2025 01:22:24] "GET / HTTP/1.1" 200 -
10.0.2.1 - - [08/May/2025 01:22:29] "GET / HTTP/1.1" 200 -
10.0.2.1 - - [08/May/2025 01:22:32] "GET / HTTP/1.1" 200 -
10.0.2.1 - - [08/May/2025 01:22:34] "GET / HTTP/1.1" 200 -
abhi@Mac ac11950_cml_hw4 %
```

Figure 13: Inference service logs showing the Flask application serving requests

6.3 Web Interface Implementation

The web interface was implemented as an HTML form allowing users to upload handwritten digit images:

1. Created a responsive HTML template with a file upload form.
2. Implemented client-side validation for image uploads.
3. Designed a clean result display area for showing the predicted digit.
4. The completed web interface is shown in Figure 14:

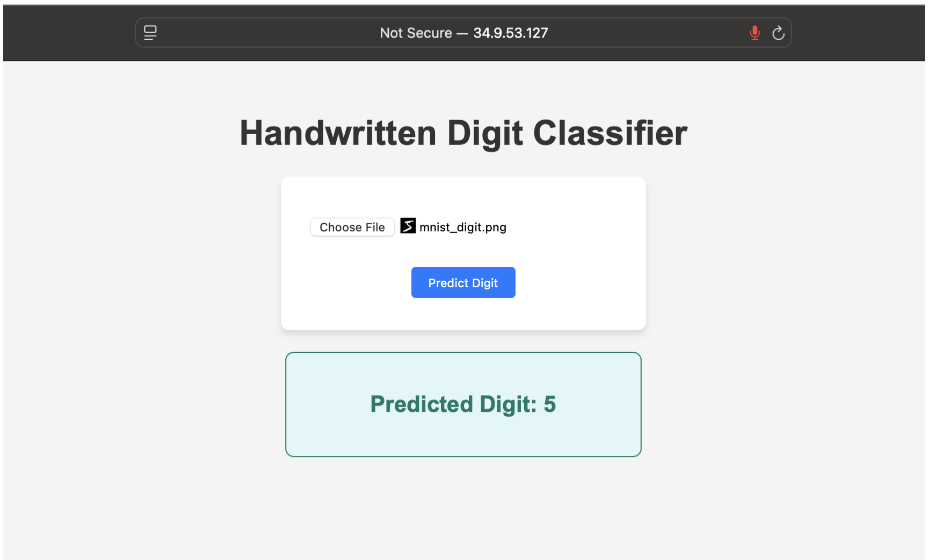


Figure 14: Web interface for handwritten digit classification

5. **The live demo of this service is publicly accessible at `http://34.9.53.127`. Users can visit this URL directly to try the handwritten digit classifier with their own images.**

7 Results and Analysis

7.1 Training Performance

The training job successfully trained a convolutional neural network on the MNIST dataset:

1. The training completed in a single epoch, with the loss decreasing from an initial value of 2.31 to a final value of 0.012.
2. The training logs (Figure 9) show a steady decrease in loss, indicating successful learning.
3. The trained model was saved to the persistent volume at path `/mnt/ac11950_model.pt`, making it accessible for the inference service.

7.2 Inference Service Performance

The inference service demonstrated reliable performance for digit classification:

1. The service successfully loaded the model from the persistent volume.
2. The Flask application responded to HTTP requests with low latency, as shown in the logs (Figure 13).

3. The web interface was accessible via the LoadBalancer's external IP address (34.9.53.127), as shown in Figure 15.

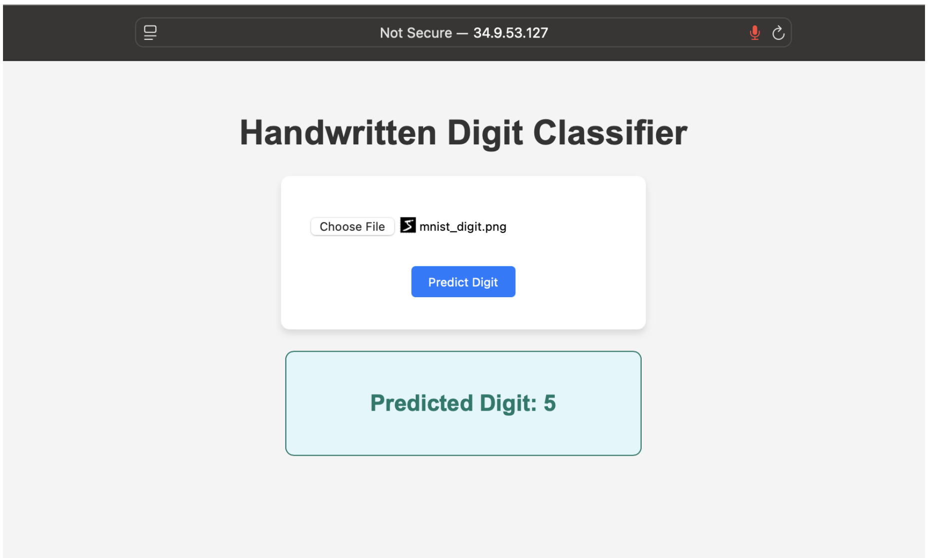


Figure 15: Web interface accessible at external IP 34.9.53.127 with a successful prediction of digit "5"

4. Users could select handwritten digit images for classification, as demonstrated in Figure 16:

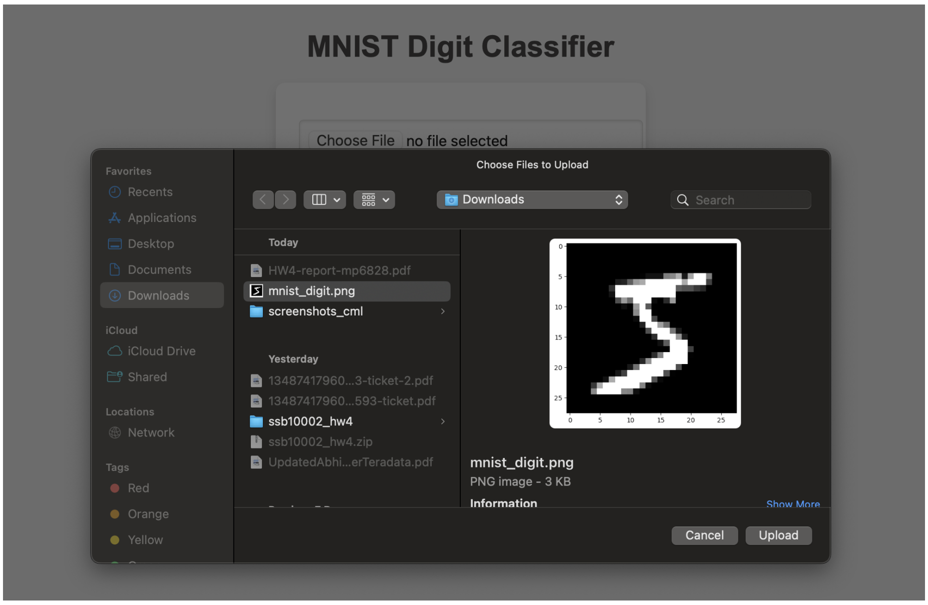


Figure 16: File selection dialog showing a handwritten digit "5" (mnist_digit.png) being chosen for prediction

5. The model correctly identified the uploaded handwritten digit as "5", demonstrating the end-to-end functionality of the pipeline from training to inference.

7.3 Kubernetes Resource Management

The Kubernetes orchestration demonstrated effective resource management:

1. Pods were scheduled appropriately across the cluster.
2. The completed training job (Figure 10) remained in the "Completed" state for logging purposes.
3. The inference service scaled to the specified number of replicas and maintained high availability.
4. The LoadBalancer service successfully exposed the application with an external IP of 34.9.53.127 (Figure 12).

7.4 Challenges and Solutions

Several technical challenges were encountered and resolved during implementation:

1. **Image Compatibility:** Resolved the "no matching manifest for linux/amd64" error by building multi-architecture Docker images.
2. **ImagePullBackOff Error:** Fixed by ensuring the DockerHub images were public and using correct image tags.
3. **PVC Mount Issues:** Resolved by verifying the correct path specifications and permissions in the volume mounts.
4. **Model Loading Errors:** Debugged and fixed issues with model serialization format compatibility between training and inference environments.

8 Conclusion

This project successfully demonstrated the deployment of a deep learning model training and inference pipeline on Google Kubernetes Engine using containerization and cloud-native technologies. The implementation achieved all the primary objectives set out in the problem statement.

8.1 Technical Achievements

The following key technical achievements were accomplished:

1. **End-to-End ML Pipeline**: Successfully implemented a complete machine learning workflow from data preparation and model training to inference serving within a Kubernetes environment.
2. **Containerization Strategy**: Leveraged Docker containers to create reproducible, portable, and isolated environments for both training and inference components.
3. **Data Persistence**: Effectively utilized Kubernetes PersistentVolumeClaims to enable seamless data sharing between the training and inference components, demonstrating a solution to the stateful workload challenge in containerized environments.
4. **Public Service Exposure**: Successfully exposed the inference service to the public internet using Kubernetes LoadBalancer, enabling real-time user interaction with the trained model.
5. **Resource Optimization**: Implemented appropriate resource requests and limits to ensure efficient cluster utilization while maintaining performance.
6. **Operational Monitoring**: Established logging and monitoring practices to track both training progress and inference service health.

8.2 Advantages of the Approach

The containerized, Kubernetes-based approach offers several significant advantages over traditional ML deployment methods:

1. **Scalability**: The architecture can easily scale to handle more complex models, larger datasets, or increased inference traffic through Kubernetes' native scaling capabilities.

2. **Reproducibility**: The entire pipeline, including environment configurations, is defined as code through Dockerfiles and Kubernetes manifests, ensuring consistent deployment across environments.

3. **Maintainability**: The microservices approach allows for independent updates to each component without affecting the entire system.

4. **Resilience**: Kubernetes provides self-healing capabilities, automatically restarting failed pods and rescheduling workloads as needed.

5. **Cost Efficiency**: Cloud-native deployments enable efficient resource utilization, with the ability to scale down or up based on demand.

8.3 Future Improvements

While the current implementation successfully demonstrates the concept, several enhancements could be made in future iterations:

1. **Production-Grade Web Server**: Replace the Flask development server with a production-grade WSGI server like Gunicorn or uWSGI.

2. **CI/CD Integration**: Implement a continuous integration and deployment pipeline to automate testing and deployment of model updates.

3. **Horizontal Pod Autoscaling**: Configure HPA to automatically scale the inference service based on CPU utilization or custom metrics like request rate.

4. **Advanced Monitoring**: Integrate with Prometheus and Grafana for comprehensive monitoring of both system metrics and ML-specific metrics.

5. **A/B Testing**: Implement Kubernetes-native A/B testing to compare different model versions using traffic splitting.

6. **Model Versioning**: Implement a system for tracking and managing different versions of trained models.

7. **Security Enhancements**: Add authentication, encrypted communication, and proper access controls for the inference service.

8.4 Broader Impact

The successful implementation of this project demonstrates how modern cloud-native technologies can transform the machine learning deployment workflow. By leveraging containerization and orchestration platforms like Kubernetes, ML practitioners can focus more on model development while ensuring production-grade reliability, scalability, and maintainability. This approach bridges the gap between data science experimentation and production deployment, accelerating the time-to-value for machine learning initiatives.

The architecture pattern demonstrated in this project can be adapted for a wide range of ML use cases beyond image classification, including natural language processing, recommendation systems, and time-series forecasting. The underlying principles of containerization, orchestration, and microservices remain applicable across domains, making this a valuable reference implementation for diverse ML applications.

9 References

1. Google Kubernetes Engine Documentation. (2024). `https://cloud.google.com/kubernetes-engine/docs`
2. Kubernetes Official Documentation. (2024). `https://kubernetes.io/docs/home/`
3. PyTorch Documentation. (2024). `https://pytorch.org/docs/stable/index.html`
4. Flask Documentation. (2024). `https://flask.palletsprojects.com/`
5. The MNIST Database of Handwritten Digits. (2024). `http://yann.lecun.com/exdb/mnist/`
6. Burns, B., et al. (2021). *Kubernetes: Up and Running: Dive into the Future of Infrastructure*. O'Reilly Media.
7. Géron, A. (2023). *Hands-on Machine Learning with Scikit-Learn, Keras, and TensorFlow*. O'Reilly Media.
8. Hausenblas, M., & Schimanski, S. (2022). *Programming Kubernetes: Developing Cloud-Native Applications*. O'Reilly Media.