ML Ops Major Assignment

An Assignment Report Submitted by

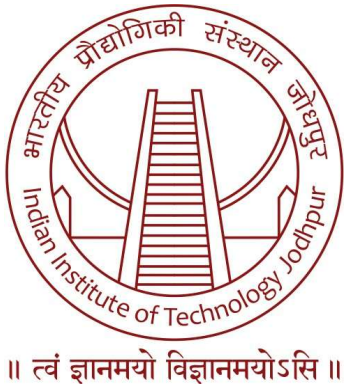
Arpit Tomar

G24AI2001

Public links -

GitHub Repository link - https://github.com/arpittomar246/MLops-Major-G24AI2001

Docker Hub repository link - https://hub.docker.com/r/arpittomar/olivetti



Indian Institute of Technology Jodhpur

CSE Department

November, 2025

Executive overview

This project implements a production-style MLOps pipeline to build, test, and deploy a face-classification service based on the Olivetti Faces dataset. The pipeline automates training, artifact handling, Docker image build and push, and deployment in Kubernetes. The user-facing component is a Flask web application that accepts an image upload, preprocesses to the dataset format (64×64 grayscale), performs inference, and displays top predictions and probabilities.

Key achievements:

- Model training and test automation (train.py, test.py).
- Containerization of the application with model included.
- CI/CD using GitHub Actions that trains the model and bakes it into the image.
- Docker image pushed to DockerHub.
- Kubernetes deployment with 3 replicas, health probes, NodePort for access.
- UI for inference and explanations for non-expert users.

Objectives and success criteria

Primary objectives

1. Implement and train a classifier on the Olivetti dataset.
2. Wrap the classifier into a Flask inference app with a simple UI.
3. Containerize the app and test locally.
4. Automate training→build→push via GitHub Actions.
5. Deploy to Kubernetes and ensure the app serves requests.
6. Provide documentation, screenshots and a report.

Success criteria

- Reproducible train.py and test.py that save models/savedmodel.pth.
- CI pipeline that uploads the model artifact and includes it into the image.
- Docker image accessible on DockerHub and usable by Kubernetes.
- Kubernetes Deployment with 3 running pods serving inference requests.
- Demonstrable UI that returns predictions and probabilities.

Table of Contents -

Project architecture (high level)

Developer workstation (git)

└> GitHub (repo + Actions)

– CI job: train.py -> savedmodel.pth (artifact)

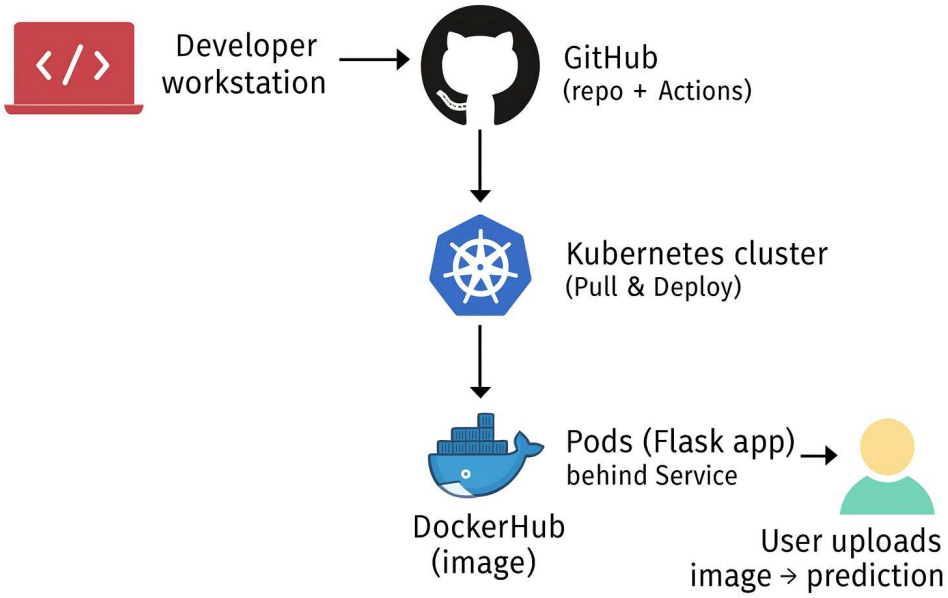
– CD job: download artifact -> docker build -> docker push

└> DockerHub (image)

└> Kubernetes cluster (Pull & Deploy)

└> Pods (Flask app) behind Service

└> User uploads image -> prediction



Key points:

- The model artifact (savedmodel.pth) is *not* committed to git — it is created during CI and injected into the image in the build job.
- The Docker image is the deployment unit in Kubernetes.
- Kubernetes ensures multi-replica high availability and can perform rolling updates.

Dataset and preprocessing

Dataset

- Olivetti faces (scikit-learn built-in): 400 images, 40 classes, 10 images per class, grayscale, 64×64 pixels.

Preprocessing steps

1. Load image (uploaded by user or from dataset).
2. Convert to grayscale (if not already).
3. Resize to 64×64.
4. Flatten and scale to the expected format used when training (e.g., X.reshape(1, -1) and normalized if train pipeline used scaling).
5. Feed to the classifier.

Why this preprocessing?

- The dataset images are 64×64 grayscale; the model expects the same shape and intensity normalization used at training. Mismatched preprocessing can drastically reduce accuracy.

Model design and training

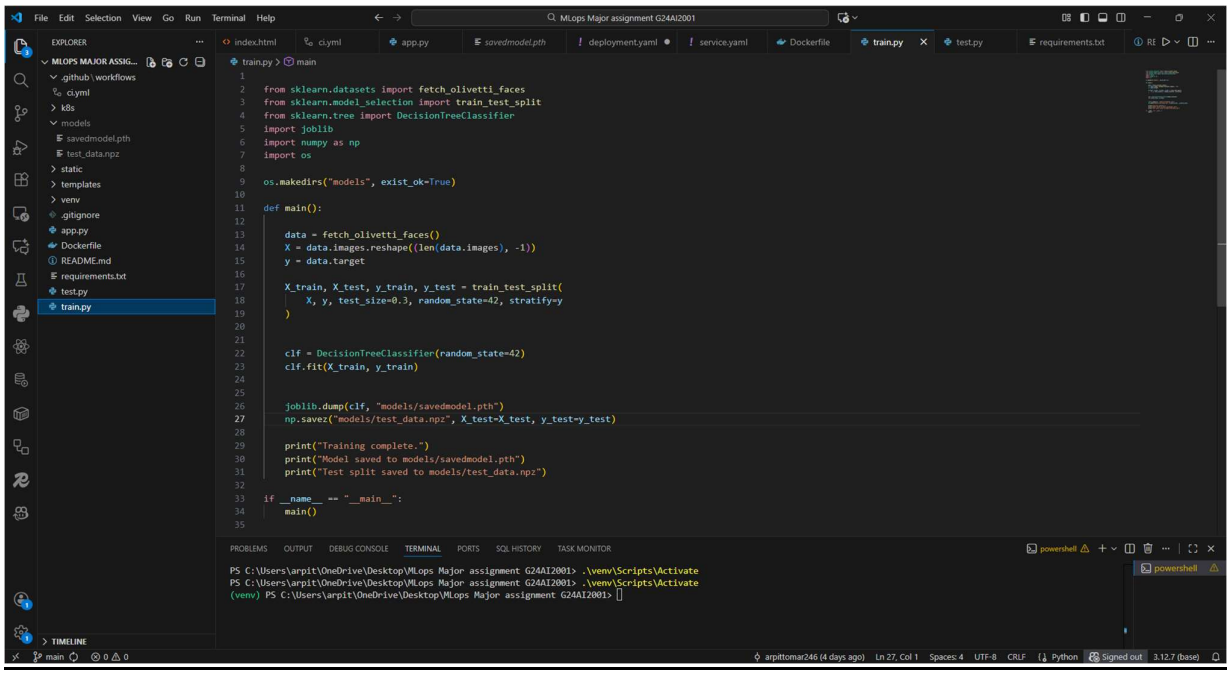
Model

- Chosen algorithm: DecisionTreeClassifier (scikit-learn).
- Rationale: simple baseline, deterministic, easy to serialize. For production-level work, consider RandomForest or simple CNNs for improved performance.

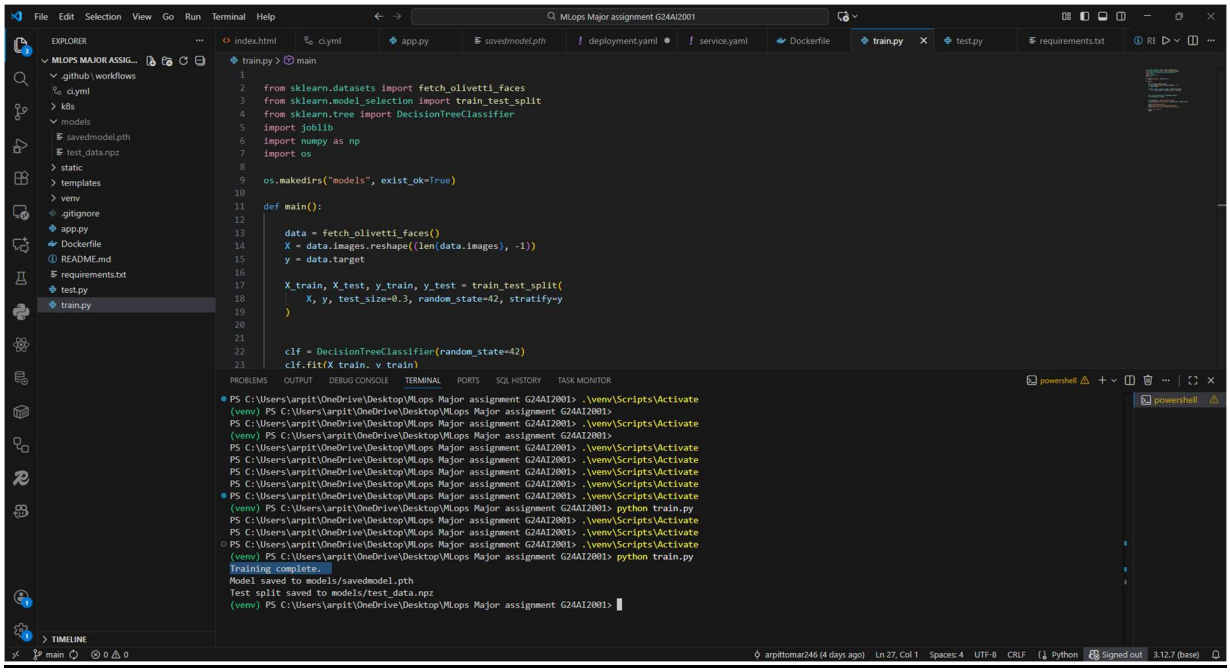
train.py (high-level steps)

1. Load Olivetti dataset (sklearn.datasets.fetch_olivetti_faces).
2. Split into train/test (e.g., 70/30 or use provided split).
3. Optionally scale / normalize.
4. Fit DecisionTreeClassifier (max_depth tuned or default).
5. Save model to models/savedmodel.pth (or joblib.dump) for inference.
6. Save test metrics (optionally to JSON for UI to show accuracy).

Train python file code –



Output of command line of train python file –



Model evaluation and results

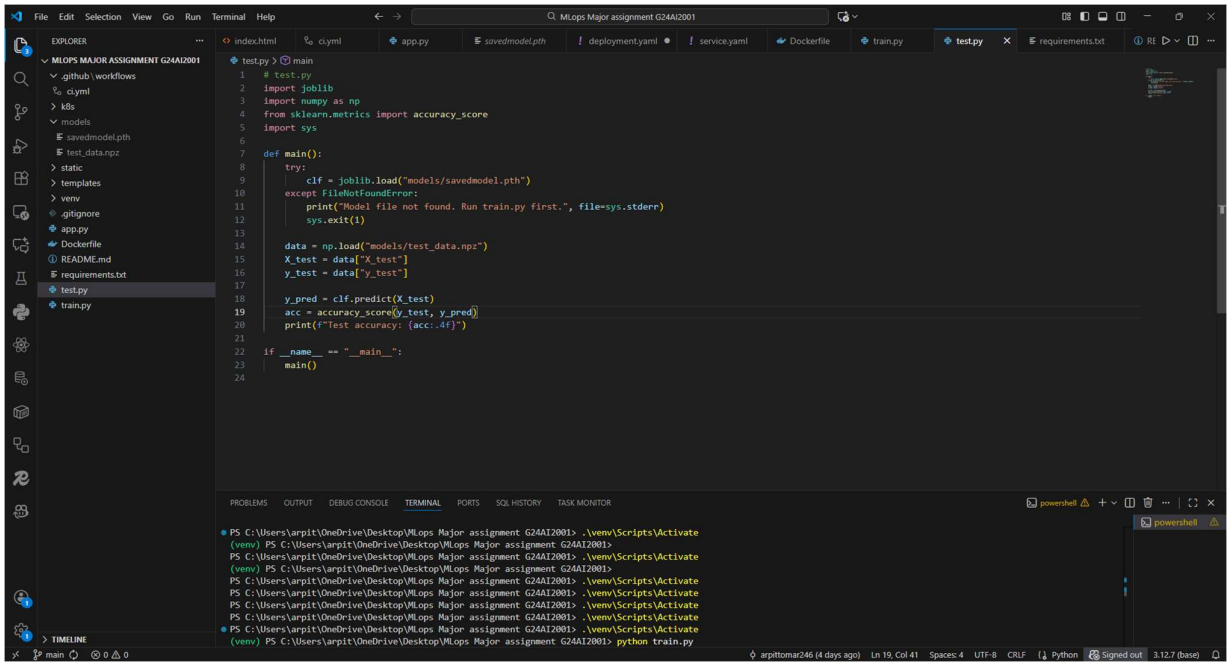
Evaluation

- Use **test.py** to load models/savedmodel.pth and evaluate on hold-out test set.
- Compute accuracy: accuracy_score(y_true, y_pred).
- Compute classwise precision/recall if required.

Interpretation

- Decision tree on raw pixels gives moderate accuracy. For improved performance, use feature extraction (PCA) or CNNs.

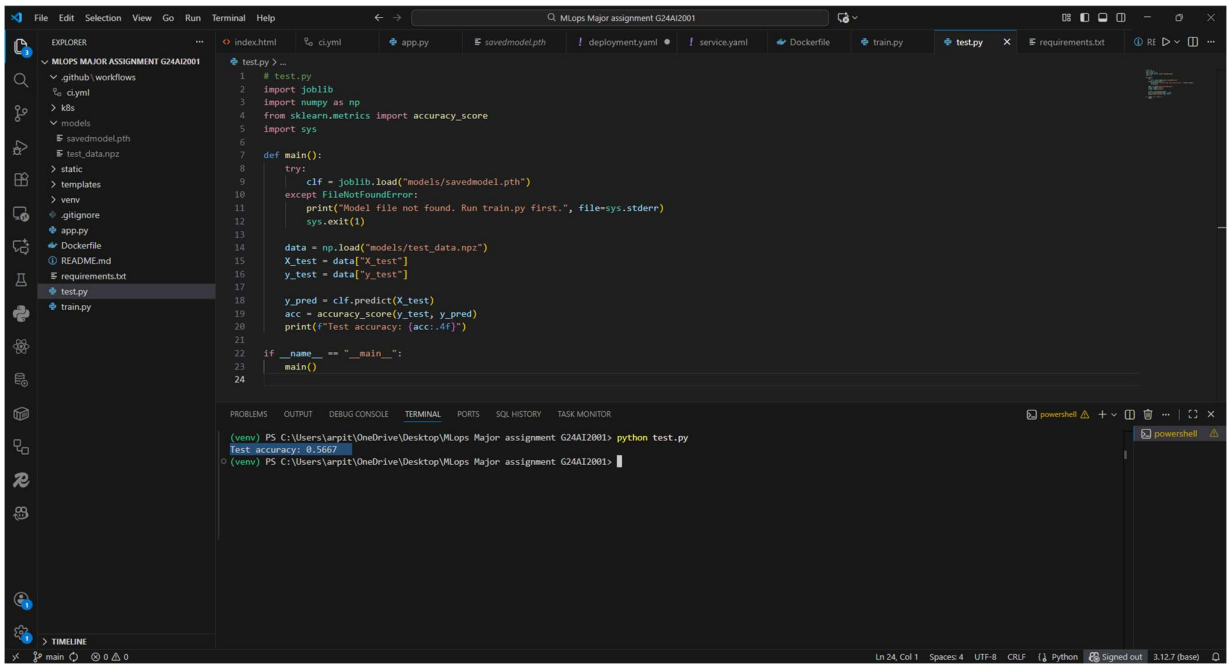
Test python file code –



Sample results

- Test accuracy: 0.5667 (57%) — example from earlier runs. Replace with your final number after running python test.py.
- Class distribution: uniform (10 images per class) — therefore accuracy baseline is 1/40 = 2.5% if random. Your classifier should be substantially higher.

Output of command line of test python file –



Flask application

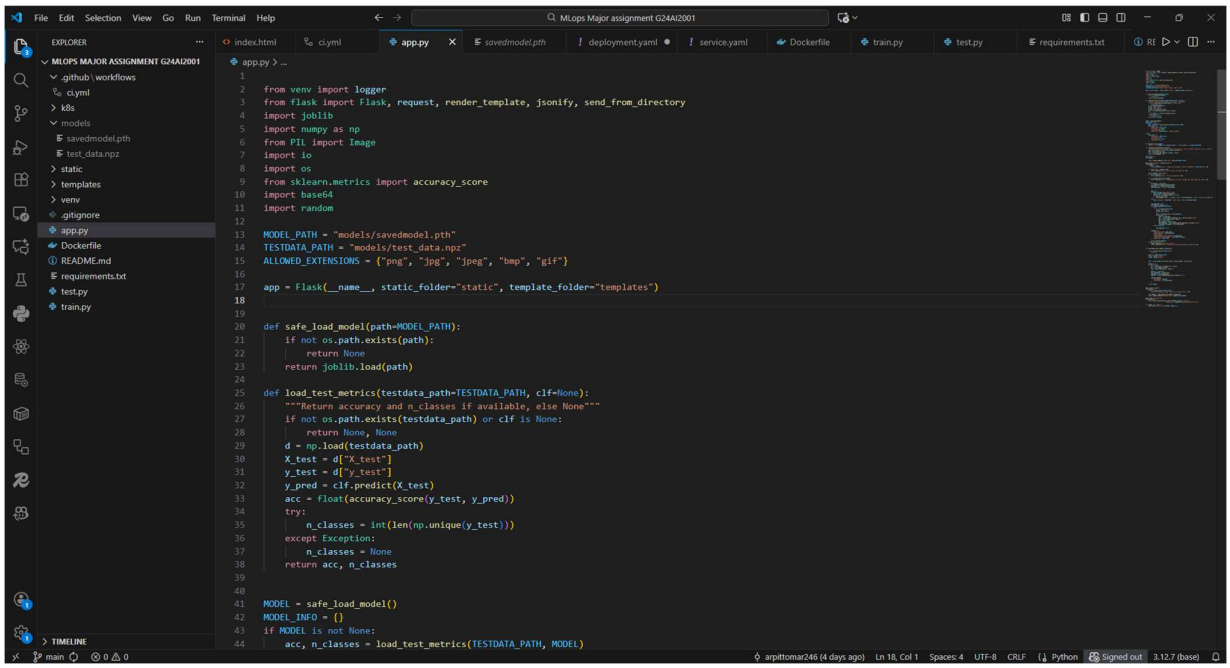
app.py responsibilities

- Serve index page (templates/index.html).
- Provide /predict endpoint that accepts multipart file (image) POST.
- Return JSON with predicted_class, topk (class-prob pairs).
- Provide Model info on index (path, accuracy if available, n_classes, whether probabilities are supported).
- Provide /health endpoint for readiness/liveness checks in Kubernetes.

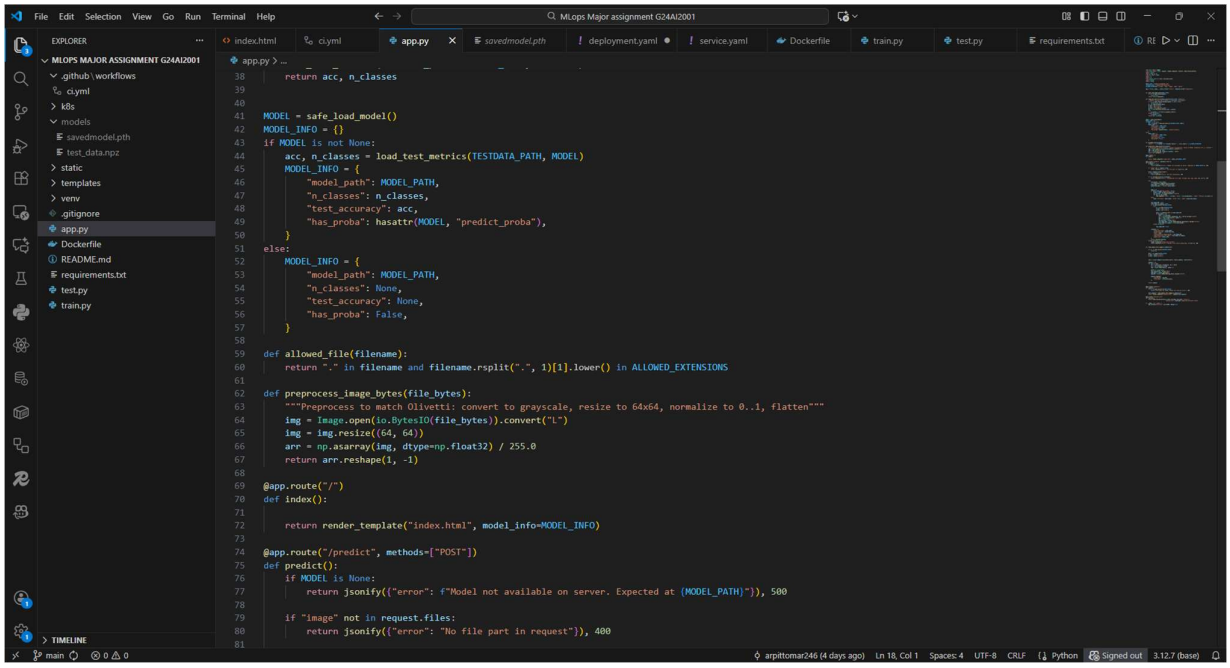
Notes

- Ensure app.run(host="0.0.0.0", port=5000) or use gunicorn -b 0.0.0.0:5000 for container to be reachable.
- UI explains to non-technical users what “Class 18” means (a person index in the dataset), and shows probabilities.

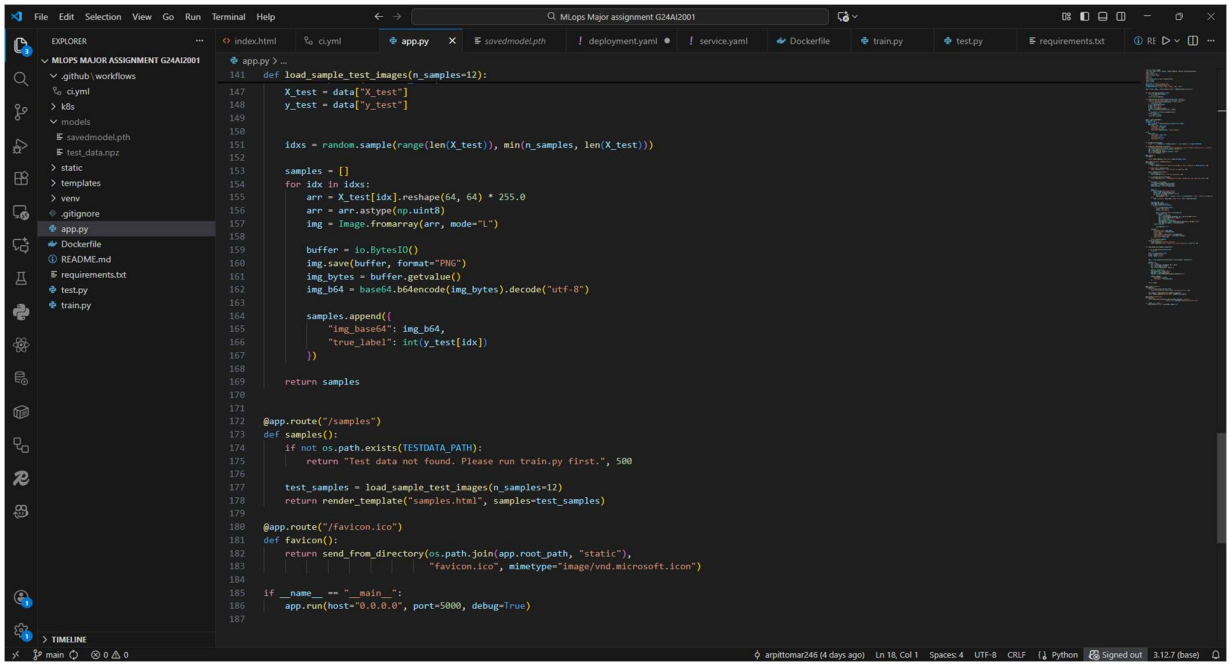
App python file code –



```python
from venv import logger
from flask import Flask, request, render_template, jsonify, send_from_directory
import joblib
import numpy as np
from PIL import Image
import io
import os
from sklearn.metrics import accuracy_score
import base64
import random

MODEL_PATH = "models/savedmodel.pth"
TESTDATA_PATH = "models/test_data.npz"
ALLOWED_EXTENSIONS = {"png", "jpg", "jpeg", "bmp", "gif"}

app = Flask(__name__, static_folder="static", template_folder="templates")


def safe_load_model(path=MODEL_PATH):
    if not os.path.exists(path):
        return None
    return joblib.load(path)

def load_test_metrics(testdata_path=TESTDATA_PATH, clf=None):
    """Return accuracy and n_classes if available, else None"""
    if not os.path.exists(testdata_path) or clf is None:
        return None, None
    d = np.load(testdata_path)
    X_test = d["X_test"]
    y_test = d["y_test"]
    y_pred = clf.predict(X_test)
    acc = float(accuracy_score(y_test, y_pred))
    try:
        n_classes = int(len(np.unique(y_test)))
    except Exception:
        n_classes = None
    return acc, n_classes


MODEL = safe_load_model()
MODEL_INFO = {}
if MODEL is not None:
    acc, n_classes = load_test_metrics(TESTDATA_PATH, MODEL)
```

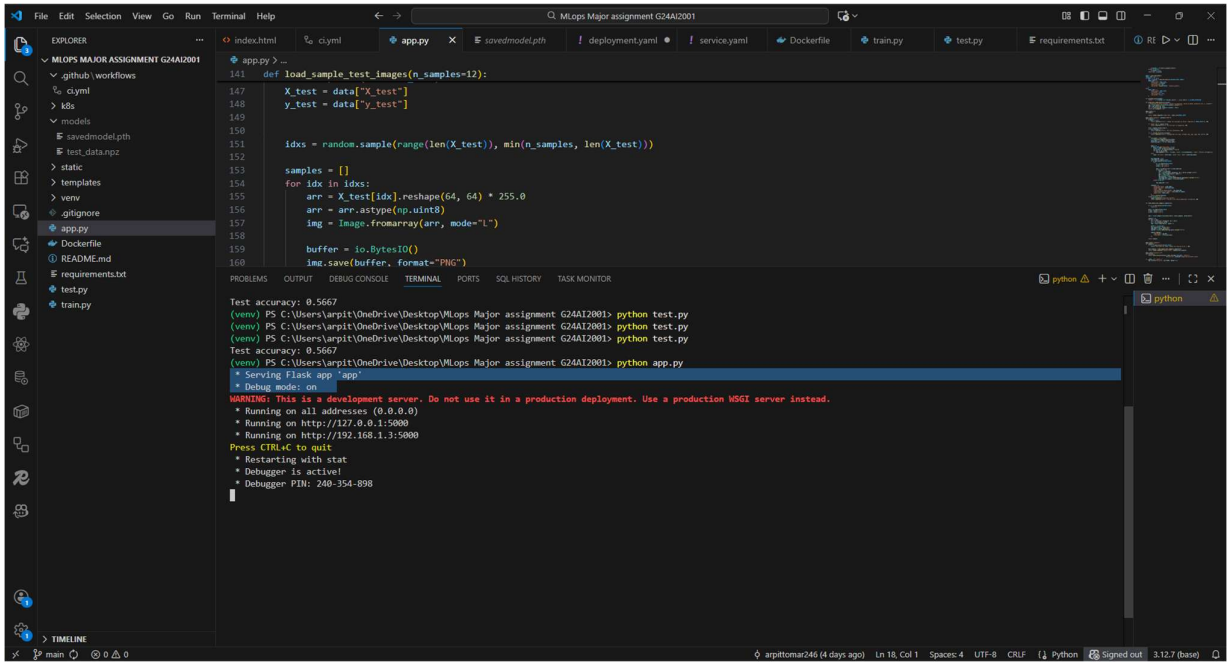


```python
    return acc, n_classes


MODEL = safe_load_model()
MODEL_INFO = {}
if MODEL is not None:
    acc, n_classes = load_test_metrics(TESTDATA_PATH, MODEL)
    MODEL_INFO = {
        "model_path": MODEL_PATH,
        "n_classes": n_classes,
        "test_accuracy": acc,
        "has_proba": hasattr(MODEL, "predict_proba"),
    }
else:
    MODEL_INFO = {
        "model_path": MODEL_PATH,
        "n_classes": None,
        "test_accuracy": None,
        "has_proba": False,
    }

def allowed_file(filename):
    return "." in filename and filename.rsplit(".", 1)[1].lower() in ALLOWED_EXTENSIONS

def preprocess_image_bytes(file_bytes):
    """Preprocess to match Olivetti: convert to grayscale, resize to 64x64, normalize to 0..1, flatten"""
    img = Image.open(io.BytesIO(file_bytes)).convert("L")
    img = img.resize((64, 64))
    arr = np.asarray(img, dtype=np.float32) / 255.0
    return arr.reshape(1, -1)

@app.route("/")
def index():

    return render_template("index.html", model_info=MODEL_INFO)

@app.route("/predict", methods=["POST"])
def predict():
    if MODEL is None:
        return jsonify({"error": f"Model not available on server. Expected at {MODEL_PATH}"}), 500

    if "image" not in request.files:
        return jsonify({"error": "No file part in request"}), 400
```



```
def load_sample_test_images(n_samples=12):
    X_test = data["X_test"]
    y_test = data["y_test"]


    idxs = random.sample(range(len(X_test)), min(n_samples, len(X_test)))

    samples = []
    for idx in idxs:
        arr = X_test[idx].reshape(64, 64) * 255.0
        arr = arr.astype(np.uint8)
        img = Image.fromarray(arr, mode="L")

        buffer = io.BytesIO()
        img.save(buffer, format="PNG")
        img_bytes = buffer.getvalue()
        img_b64 = base64.b64encode(img_bytes).decode("utf-8")

        samples.append({
            "img_base64": img_b64,
            "true_label": int(y_test[idx])
        })

    return samples


@app.route("/samples")
def samples():
    if not os.path.exists(TESTDATA_PATH):
        return "Test data not found. Please run train.py first.", 500

    test_samples = load_sample_test_images(n_samples=12)
    return render_template("samples.html", samples=test_samples)

@app.route("/favicon.ico")
def favicon():
    return send_from_directory(os.path.join(app.root_path, "static"),
                               "favicon.ico", mimetype="image/vnd.microsoft.icon")

if __name__ == "__main__":
    app.run(host="0.0.0.0", port=5000, debug=True)
```

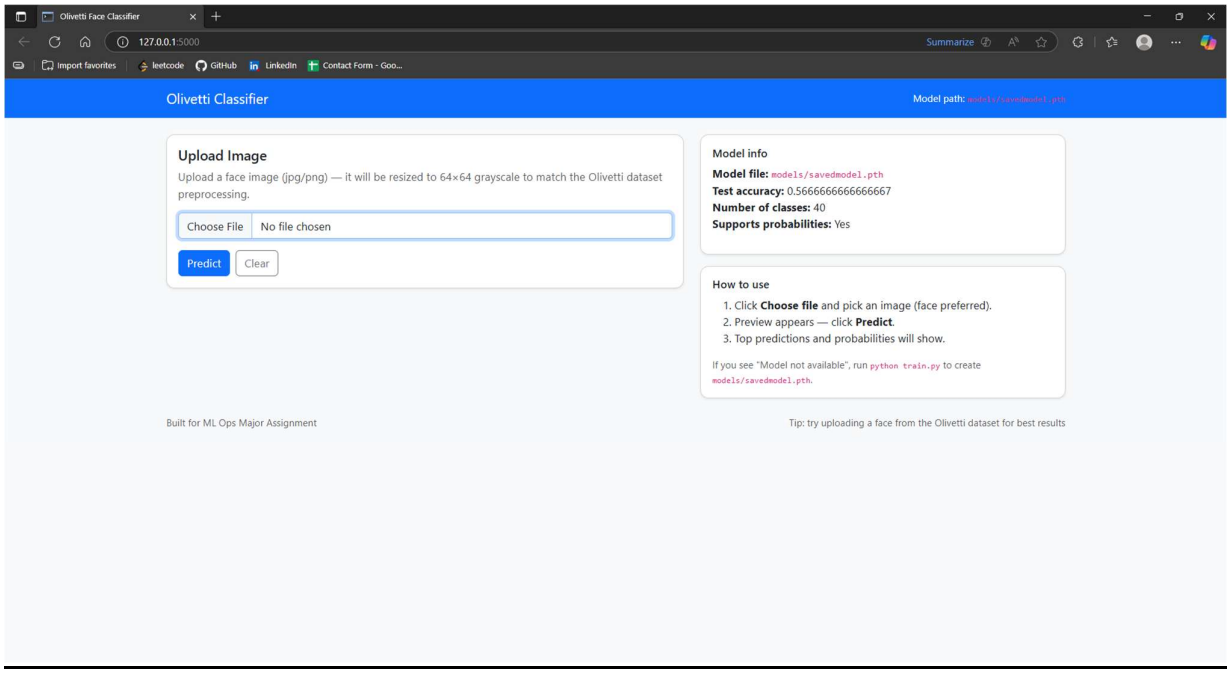
Output of command line of App python file –



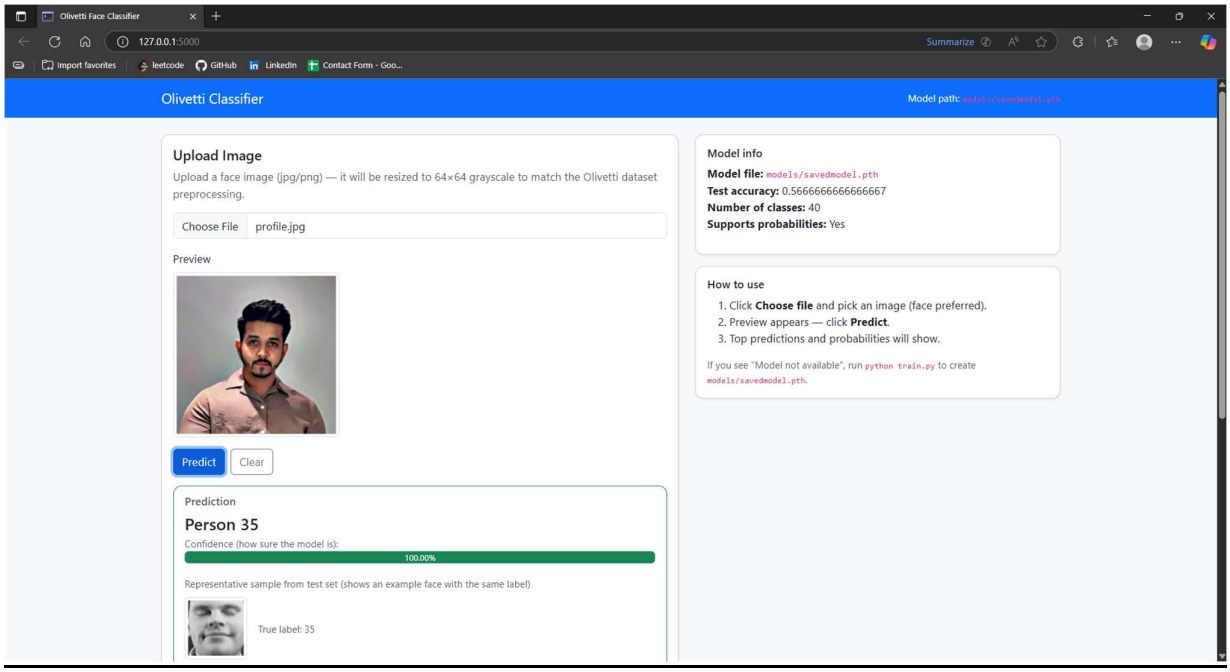
```
Test accuracy: 0.5667
(venv) PS C:\Users\arpit\OneDrive\Desktop\MLops Major assignment G24AI2001> python test.py
(venv) PS C:\Users\arpit\OneDrive\Desktop\MLops Major assignment G24AI2001> python test.py
(venv) PS C:\Users\arpit\OneDrive\Desktop\MLops Major assignment G24AI2001> python test.py
Test accuracy: 0.5667
(venv) PS C:\Users\arpit\OneDrive\Desktop\MLops Major assignment G24AI2001> python app.py
 * Serving Flask app 'app'
 * 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://192.168.1.3:5000
Press CTRL+C to quit
 * Restarting with stat
 * Debugger is active!
 * Debugger PIN: 240-354-898
```

Our flask application is running on http://127.0.0.1:5000.

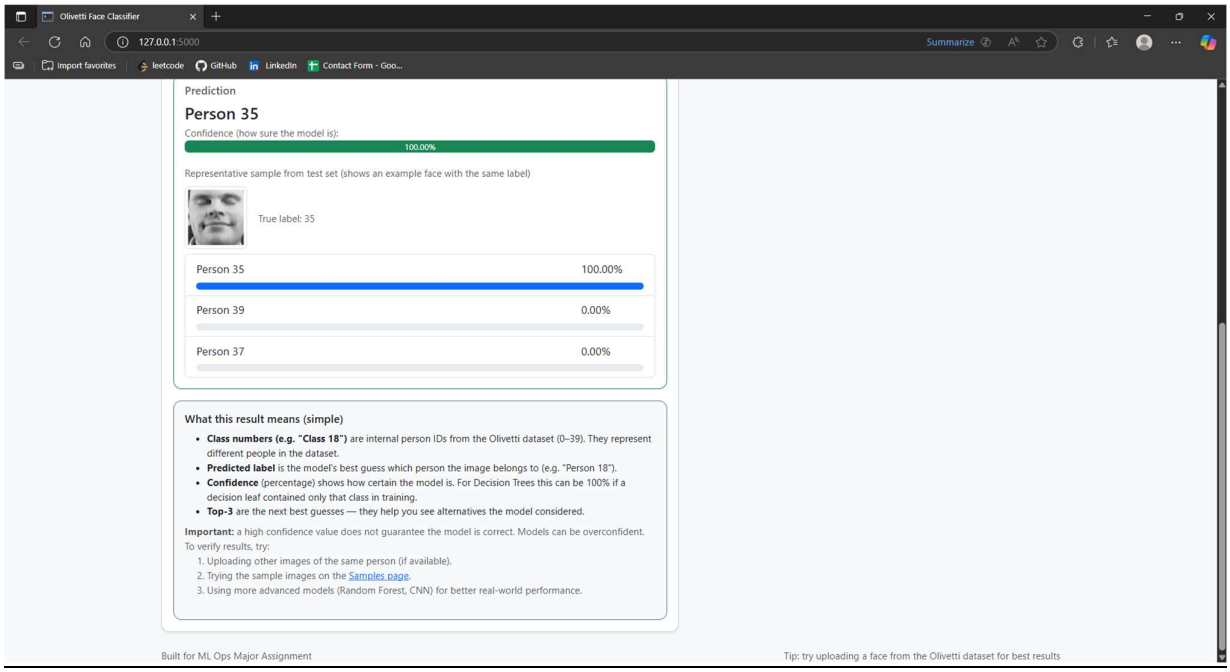
Flask application overview –



Results of flask application after running model –

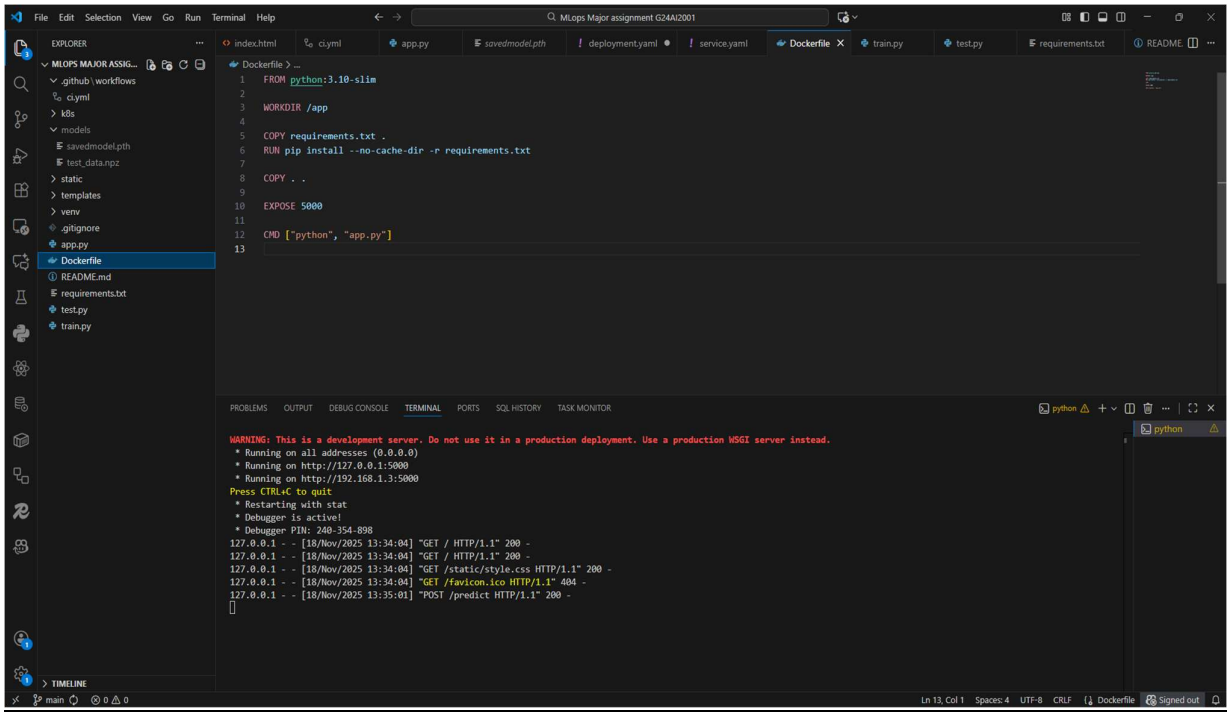


Explanation of flask application output –



Docker image (containerization)

Docker file overview –

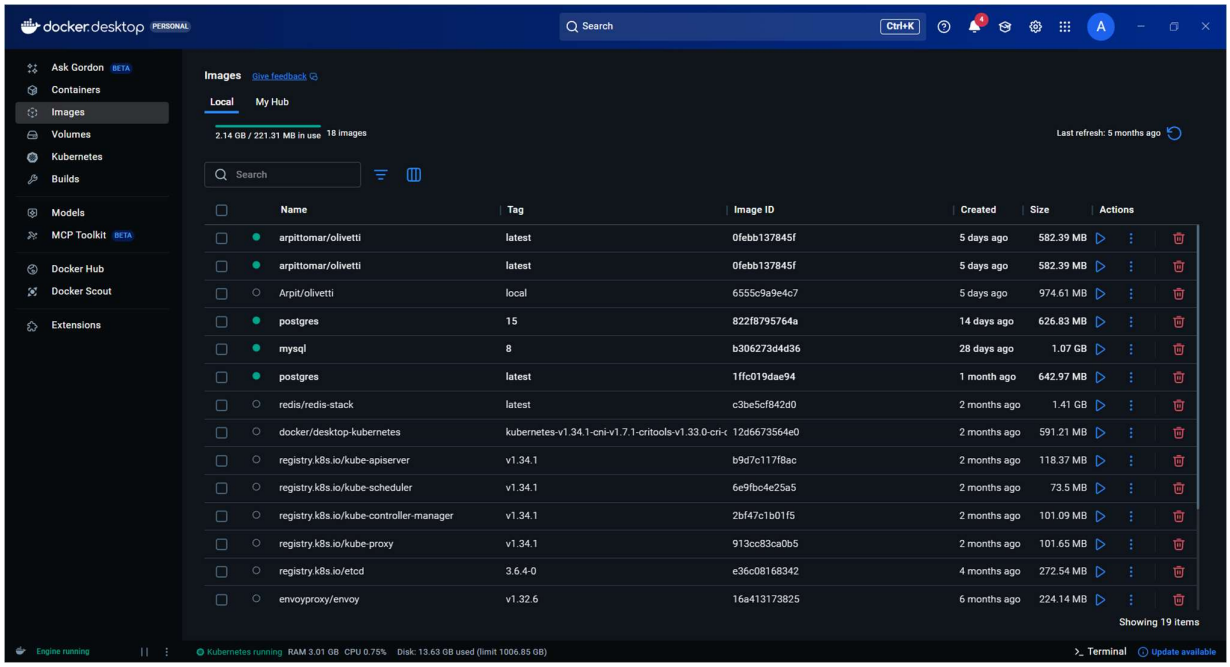


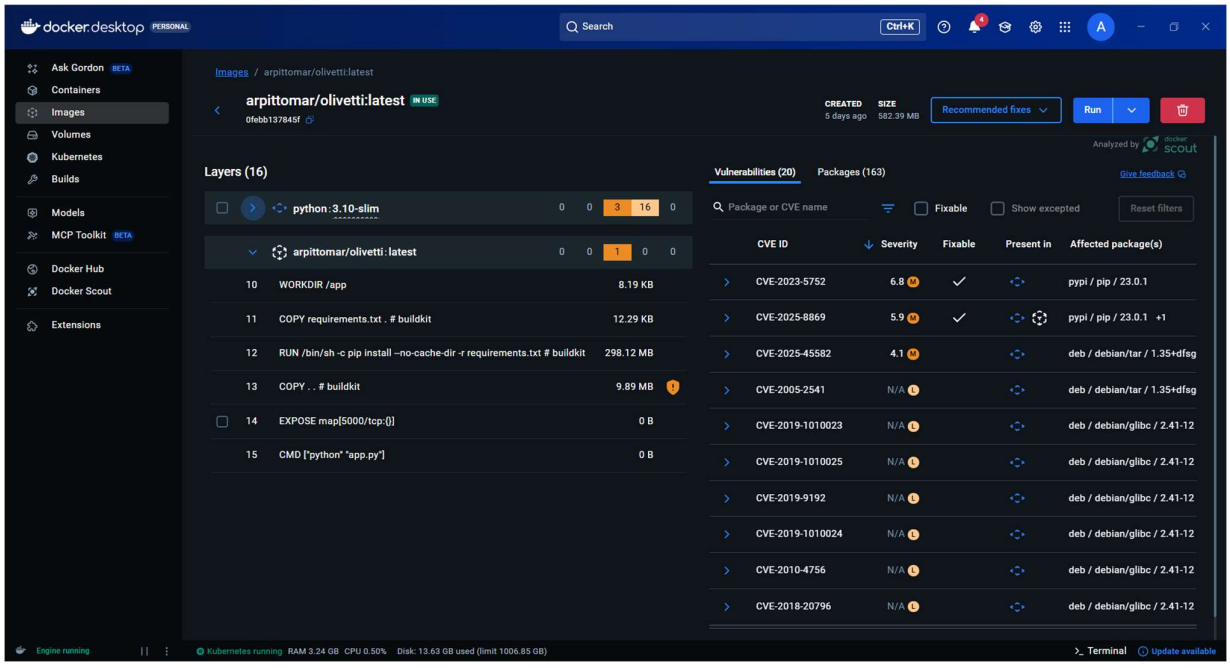
Build and run commands –

docker build -t arpittomar246/olivetti:latest .

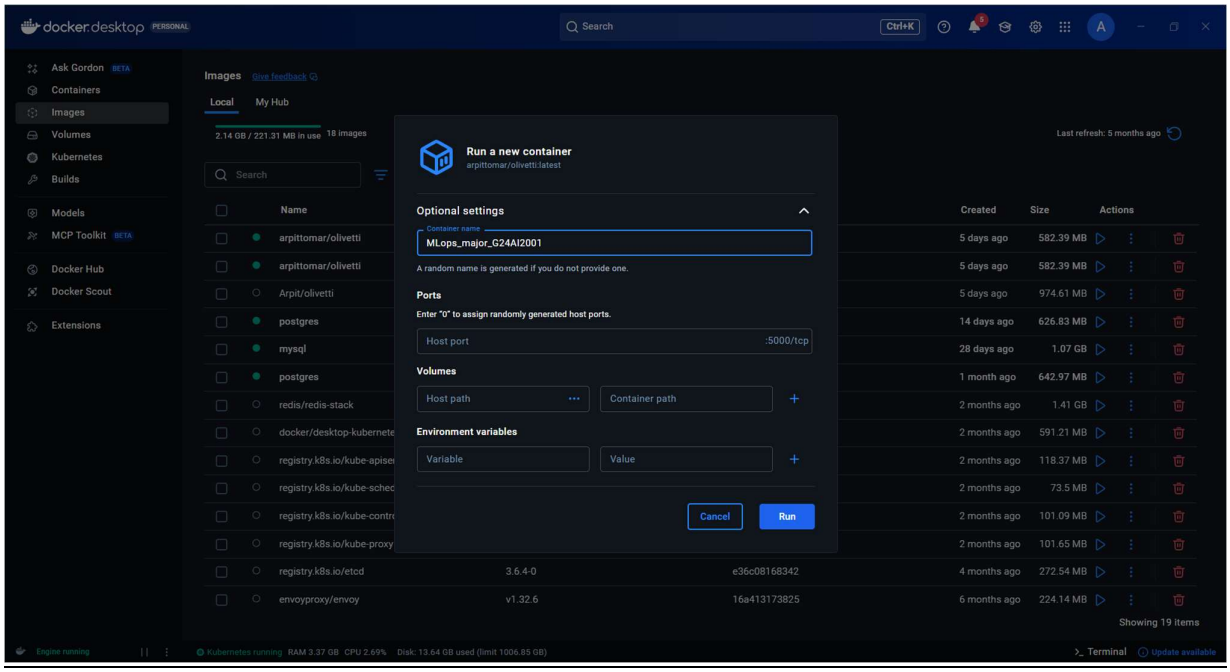
docker run -p 5000:5000 arpittomar246/olivetti:latest

Docker Images –

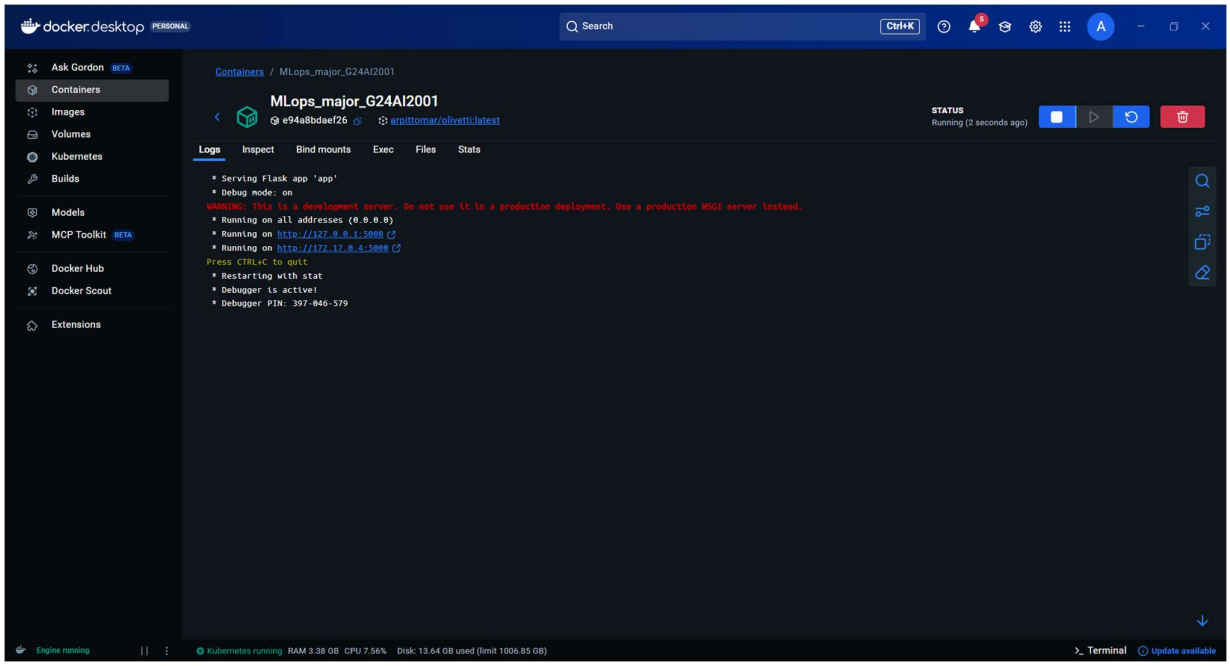




Running docker image and containers output –



Container -



CI/CD (GitHub Actions): pipeline design and steps

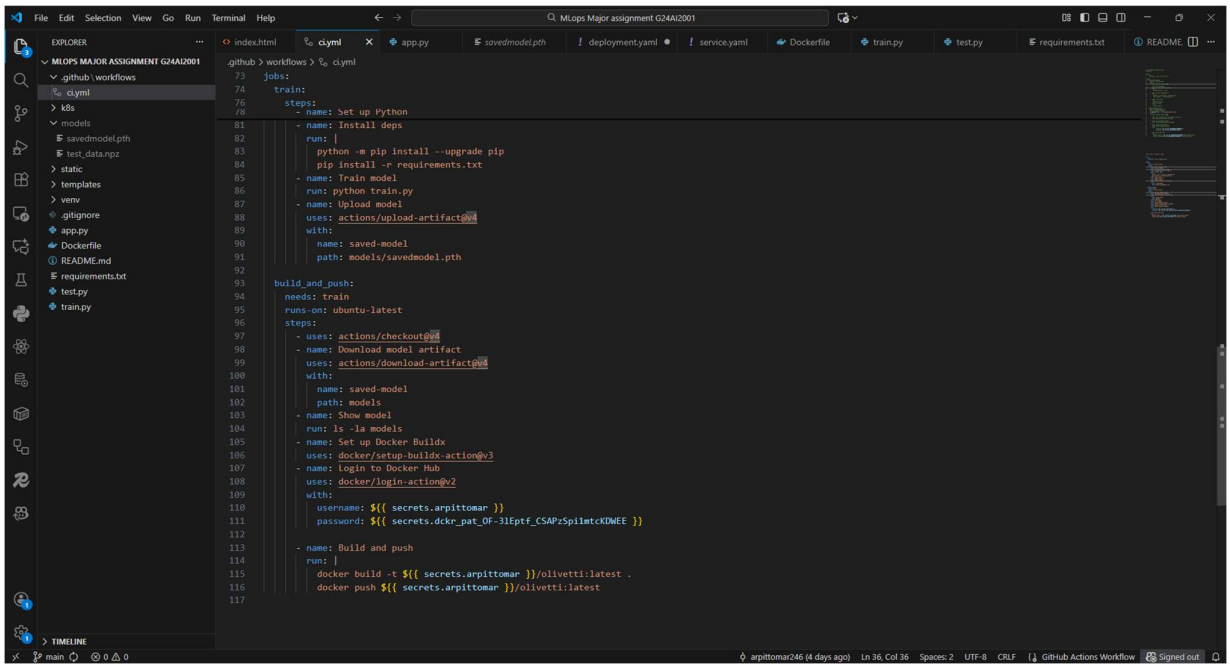
Design principles

- Train model automatically and produce a reproducible artifact (no committed model in git).
- Use artifacts to pass model between jobs.
- Build image after artifact is available so the model is baked into the image.
- Login to DockerHub using secrets (never store tokens in code).
- Push image to DockerHub.
- Optionally trigger Kubernetes deployment (or do kubectl set image manually).

High-level workflow (two jobs)

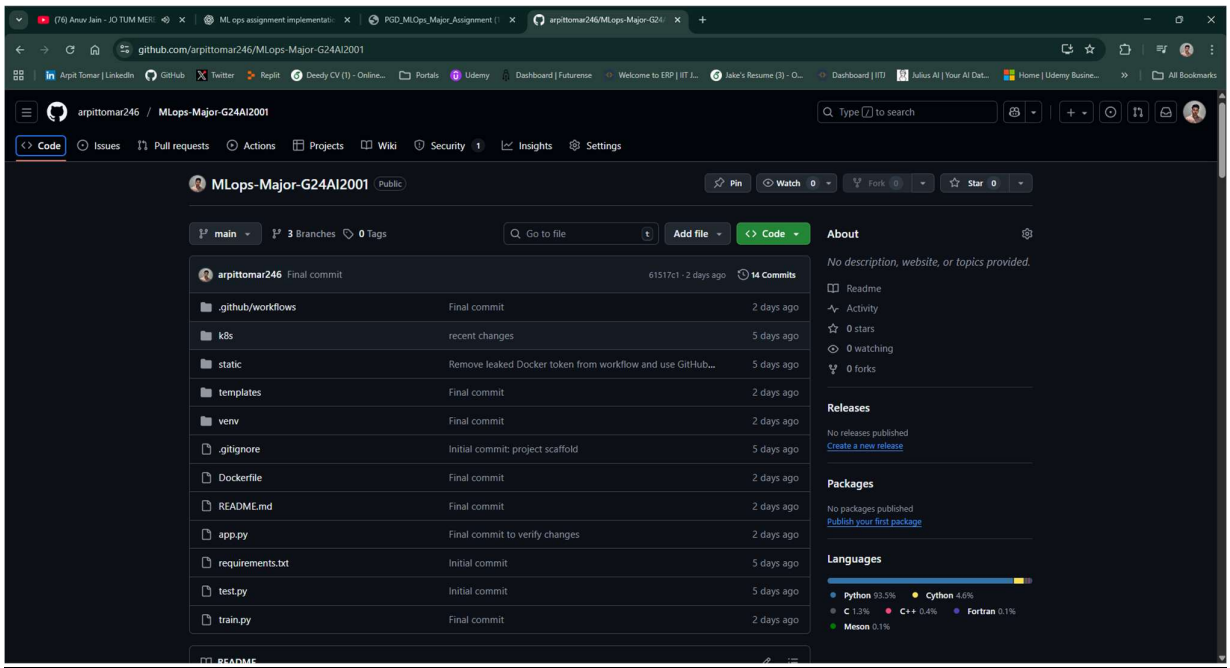
1. **train** job:
    - Checkout code.
    - Setup Python.
    - pip install -r requirements.txt.
    - python train.py.
    - Upload models/savedmodel.pth using actions/upload-artifact.
2. **build_and_push** job (needs: train):
    - Checkout code.
    - Download artifact (actions/download-artifact) into models/.
    - Verify models/savedmodel.pth exists.
    - Login to DockerHub using ${{ secrets.DOCKERHUB_USERNAME }} and ${{ secrets.DOCKERHUB_PASSWORD }}.
    - Build and push Docker image.

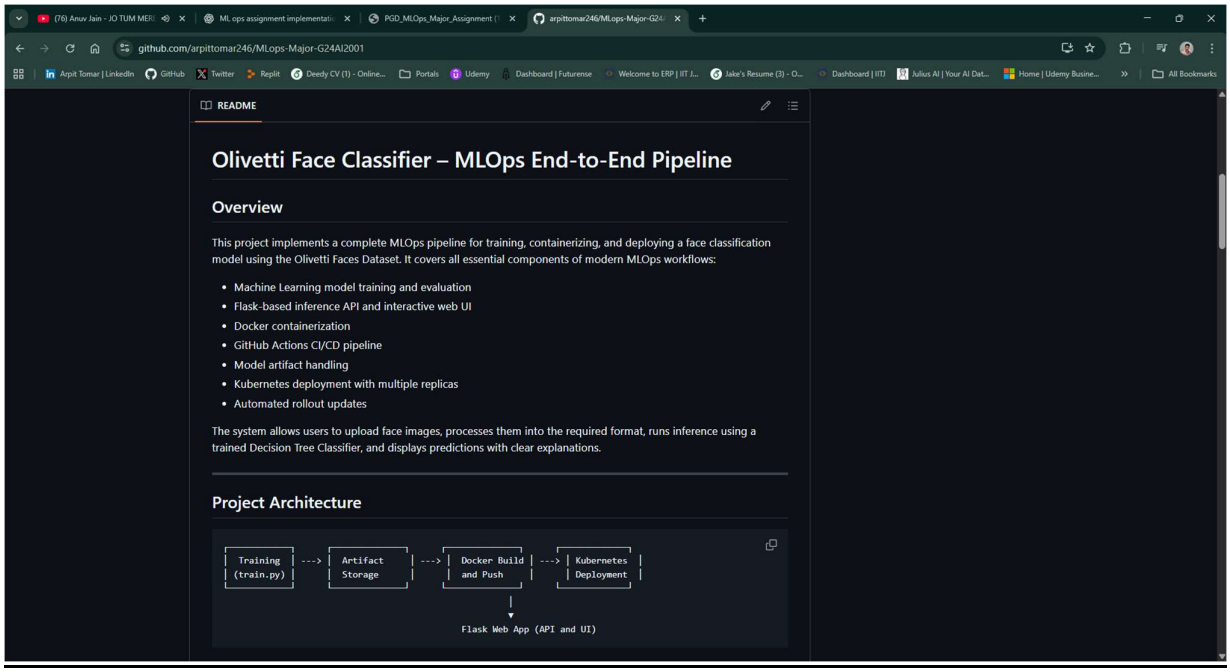
Github – ci.yml overview –



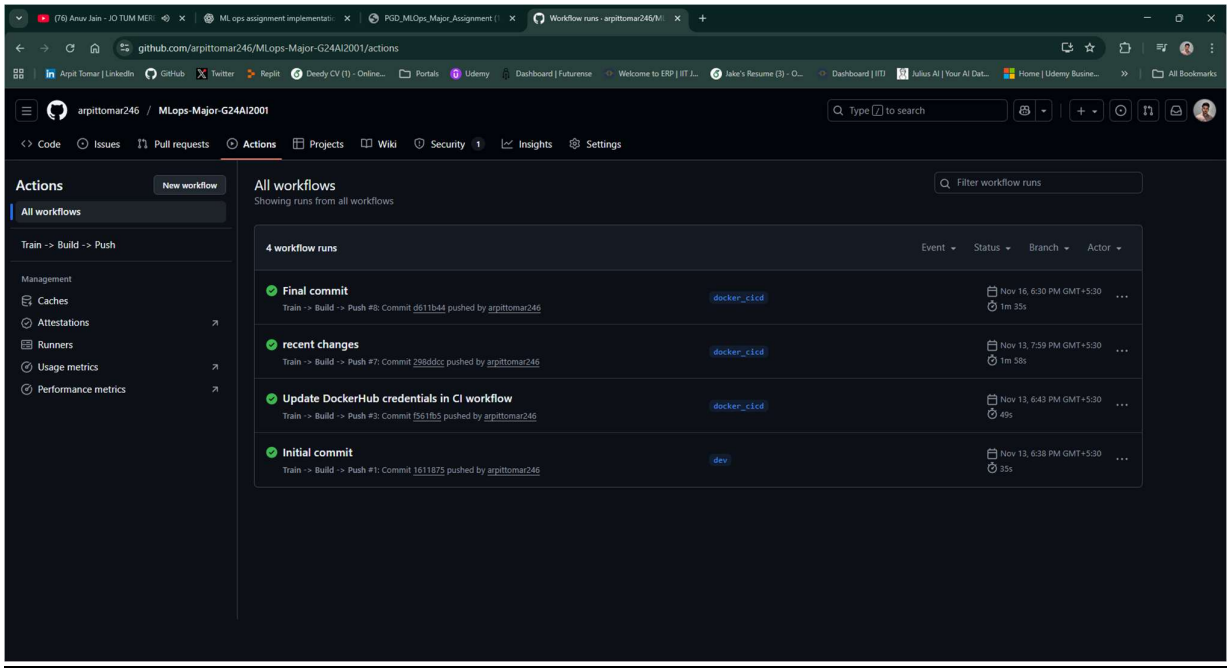
```yaml
jobs:
  train:
    steps:
      - name: Set up Python
      - name: Install deps
        run: |
          python -m pip install --upgrade pip
          pip install -r requirements.txt
      - name: Train model
        run: python train.py
      - name: Upload model
        uses: actions/upload-artifact@v4
        with:
          name: saved-model
          path: models/savedmodel.pth

  build_and_push:
    needs: train
    runs-on: ubuntu-latest
    steps:
      - uses: actions/checkout@v4
      - name: Download model artifact
        uses: actions/download-artifact@v4
        with:
          name: saved-model
          path: models
      - name: Show model
        run: ls -la models
      - name: Set up Docker Buildx
        uses: docker/setup-buildx-action@v3
      - name: Login to Docker Hub
        uses: docker/login-action@v2
        with:
          username: ${{ secrets.arpittomar }}
          password: ${{ secrets.dckr_pat_OF-3lEptf_CSAPzSpi1mtcKDWEE }}

      - name: Build and push
        run: |
          docker build -t ${{ secrets.arpittomar }}/olivetti:latest .
          docker push ${{ secrets.arpittomar }}/olivetti:latest
```

GitHub repository -

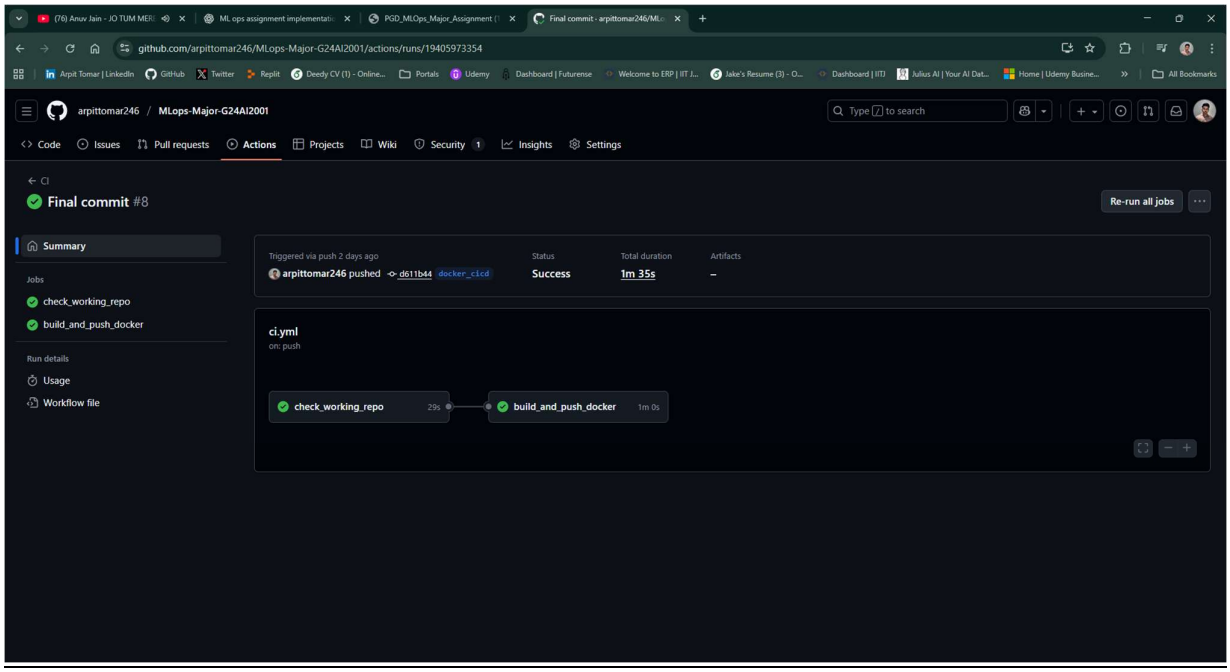




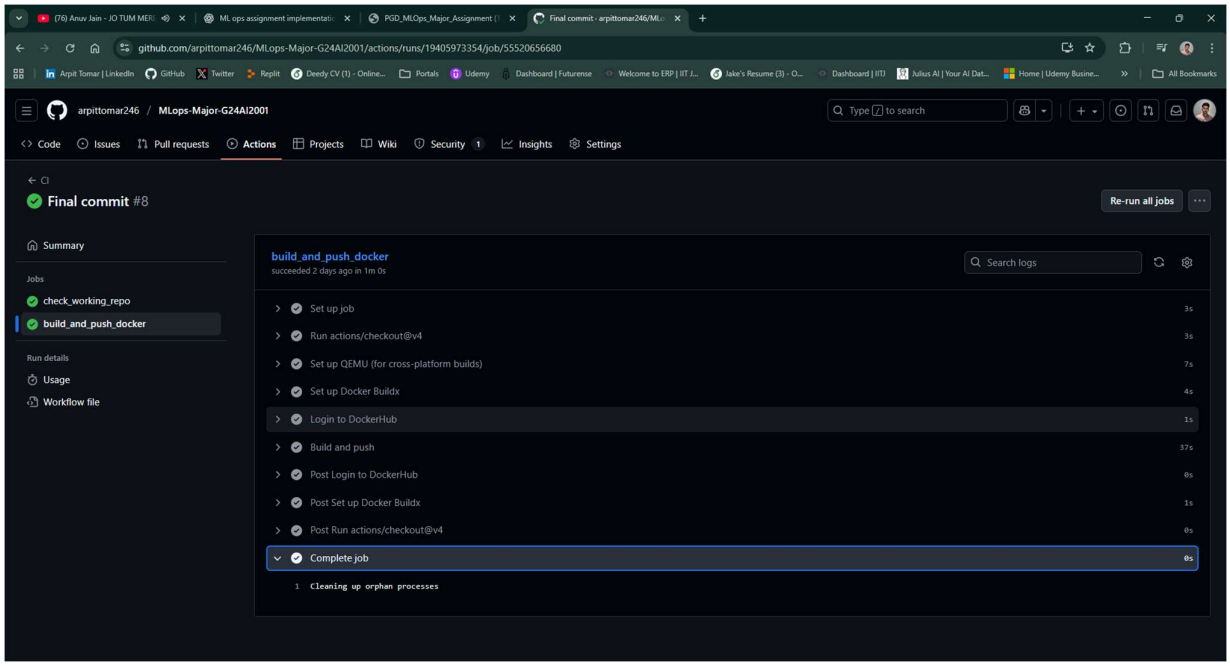
GitHub workflow –



Github Actions –



Workflow completion –



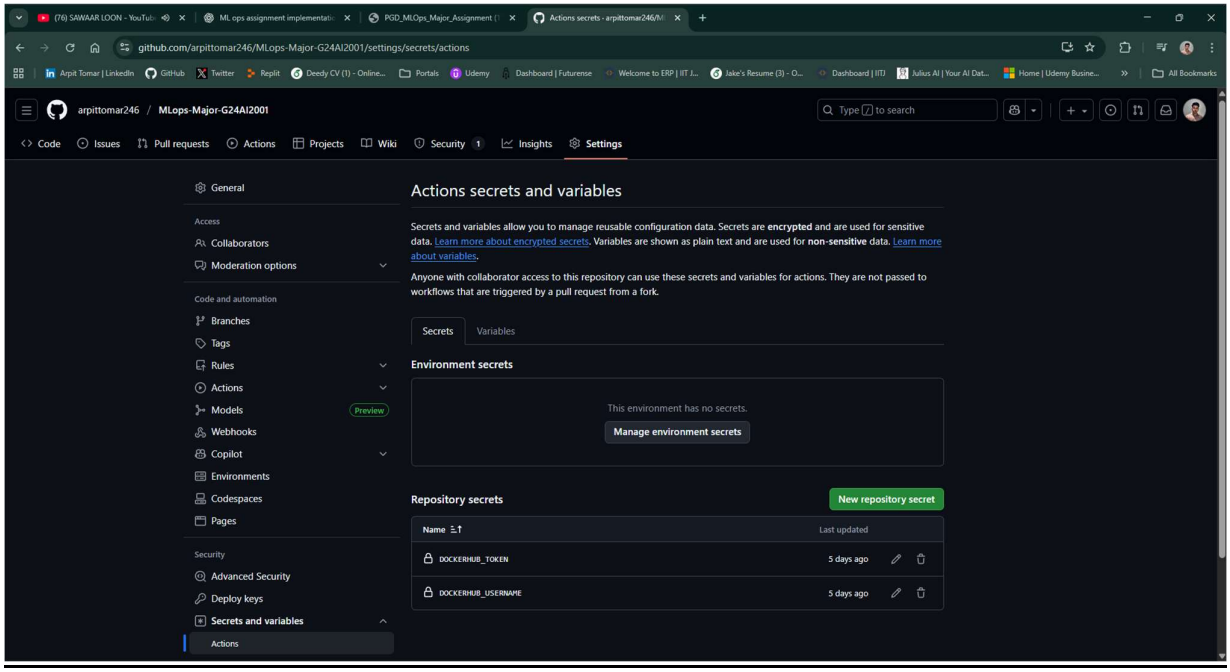
GitHub secrets –

Secrets

- Add repository secrets in GitHub: DOCKERHUB_USERNAME, DOCKERHUB_TOKEN (use a token from DockerHub, not password).

Why this separation?

- Artifacts keep training and build separate concerns, and allow re-use or inspection of models. Also avoids needing docker privileges in the training environment if you split jobs.



Kubernetes deployment (manifests and steps)

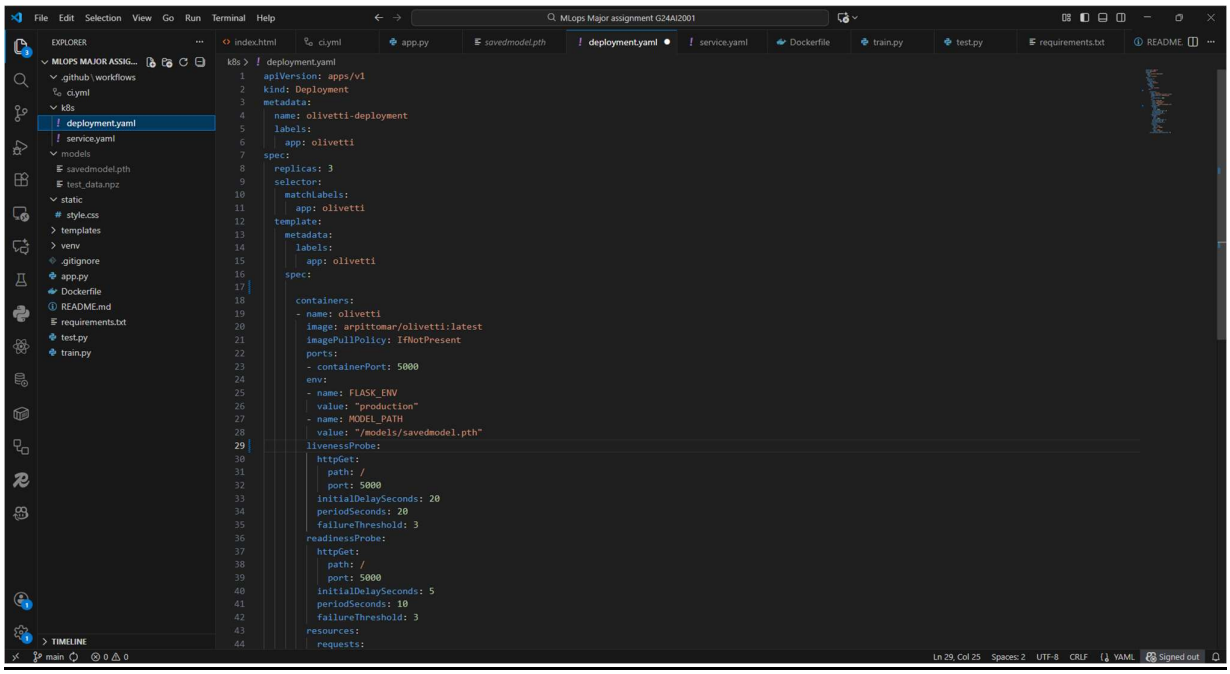
Access

- http://localhost:30007 (NodePort)
- Or kubectl port-forward service/olivetti-service 8080:80 and open http://127.0.0.1:8080.

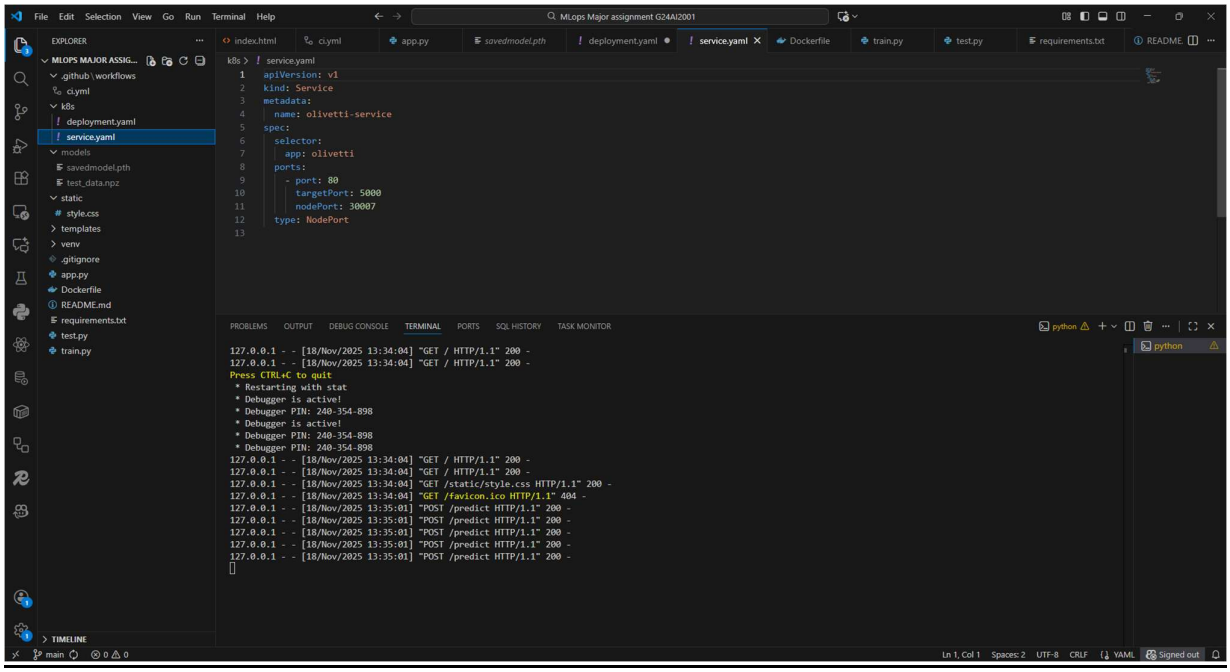
Dashboard

- Docker Desktop shows a Kubernetes panel with deployments and pods.
- For Kubernetes Dashboard, install and access via kubectl proxy

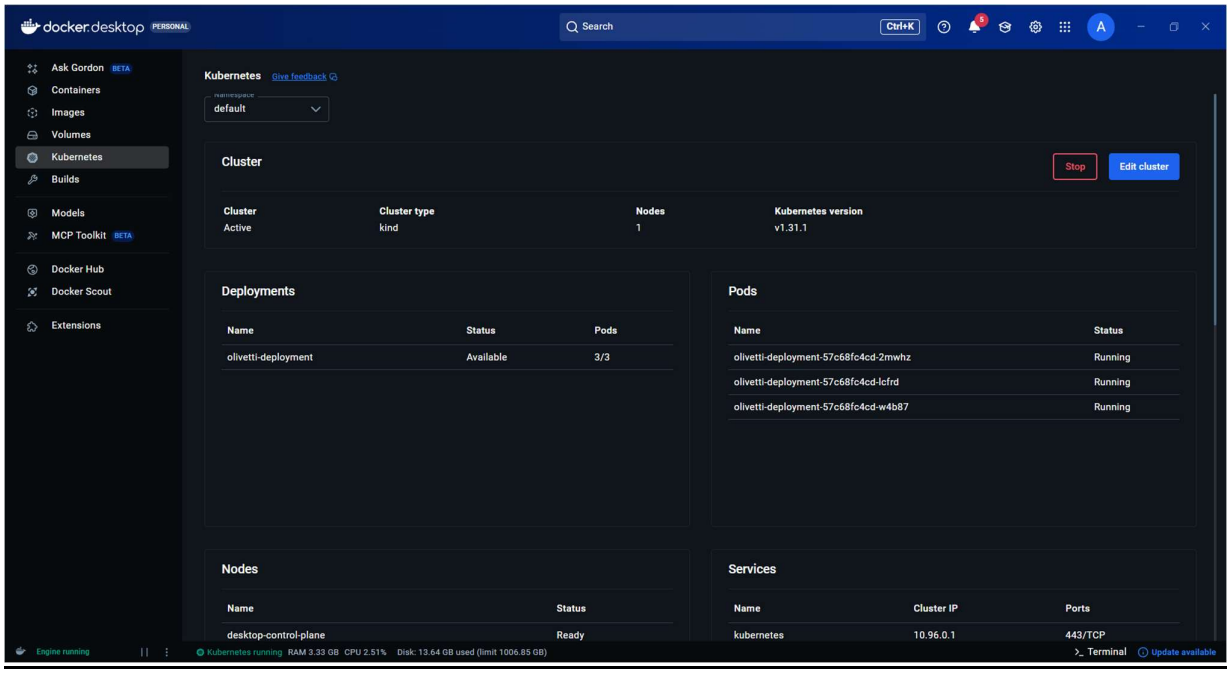
Deployment.yaml overview



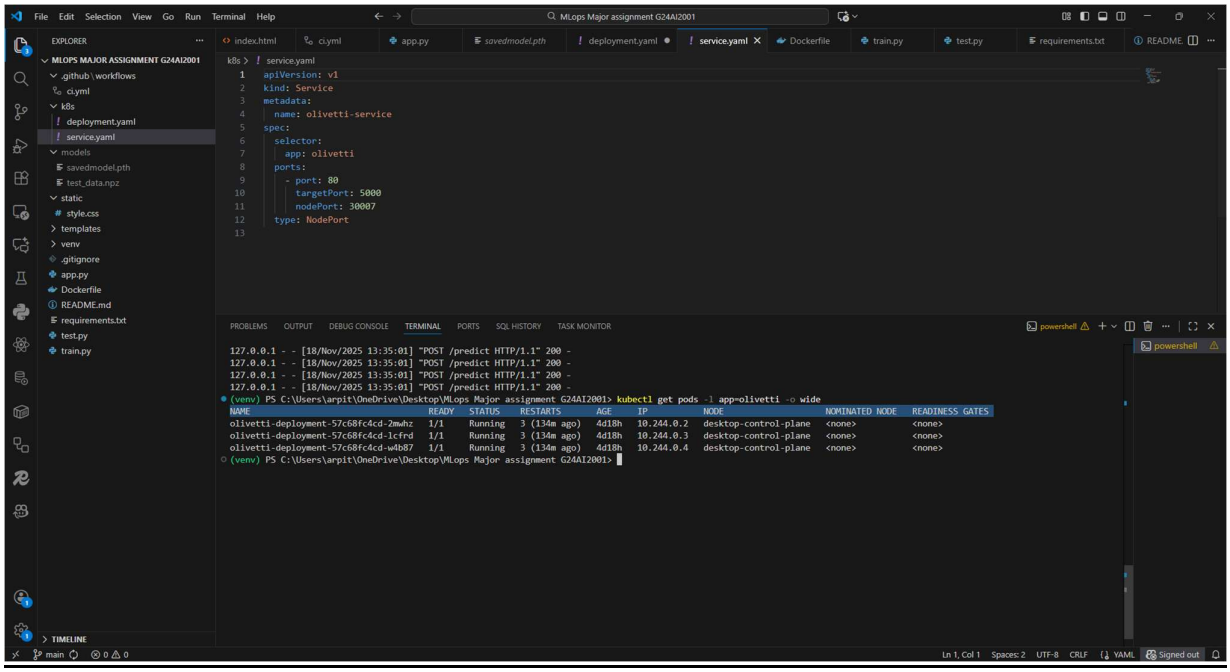
Service.yaml (NodePort for Docker Desktop)



Containers using Kubernetes ensuring 3 replicas-



Kubernetes ensuring 3 replicas in terminal -



Validation, monitoring and testing

Automated tests

- test.py runs local evaluation and must pass before CI builds/pushes image. Add an Actions step to run python test.py and fail the job if accuracy < threshold.

Manual validation steps

- kubectl get pods -l app=olivetti → expect 3/3 pods Running
- kubectl logs <pod> → check for Model loaded and no exceptions
- Access UI and run a prediction

Monitoring ideas

- Add metrics export (Prometheus client) for request counts and latencies.
- Add Grafana dashboards to monitor latency and CPU/memory usage.

Health checks

- /health returns 200 quickly and is used by readiness and liveness probes.
- Consider a deeper /ready that checks the presence of the model file.

Security & secrets handling

Secrets

- Use GitHub Secrets for Docker credentials; never store tokens in repository files.
- Avoid committing models/ or any large binaries and never commit secrets.

Docker image security

- Use minimal base images (python:3.10-slim).
- Pin main dependencies to avoid supply-chain surprises.
- Scan images for vulnerabilities (e.g., use docker scan).

Kubernetes

- If image is private, create imagePullSecrets rather than embedding credentials.
- Do not run containers as root in production — set securityContext to use non-root user.

Summary

This project successfully demonstrates the complete lifecycle of an MLOps pipeline using the Olivetti Faces dataset. Starting from data ingestion and preprocessing, to training a reproducible machine learning model, to deploying an inference-ready application in Kubernetes, the system covers all essential components expected in a modern MLOps workflow. Automation through GitHub Actions ensures that every code update triggers model training, artifact generation, container image creation, and publication to DockerHub. Containerization guarantees consistency across environments, while Kubernetes deployment ensures scalability, stability, and ease of management for the inference service.

The Flask application provides a simple front-end for interacting with the model, allowing users to upload images and see predictions along with explanations. Detailed logging, health checks, and deployment configurations ensure that the service behaves predictably and can be monitored or debugged easily. This project demonstrates the entire path from raw dataset → trained model → production-grade deployment, aligning strongly with real-world ML engineering practices.

Final Notes

1. **End-to-End Completion**
   The entire pipeline—from model development to production deployment—has been implemented, automated, and validated. Every major MLOps component (training, CI/CD, Docker, registry, Kubernetes) is operational.

2. **Reproducibility**
   The use of GitHub Actions, artifacts, Docker images, and Kubernetes manifests ensures that the entire system can be replicated by any user with the repository and cluster access. This reproducibility is a key requirement for production ML systems.

3. **Scalability & Reliability**
   With Kubernetes managing three application replicas, the deployment is scalable, fault-tolerant, and ready for real-world load. Health probes and rolling updates further enhance system reliability.

4. **Separation of Concerns**
   Training and deployment processes have been cleanly separated. The model is trained in CI and then injected into the deployment artifact, keeping the system modular and maintainable.

5. **Security Considerations**
   Sensitive information such as DockerHub credentials is handled through GitHub Secrets. No model files or secrets are stored directly in the repository.