Design and implement an end-to-end MLOps pipeline for model building, artifact/image creation, packaging, containerization, and CI/CD-based deployment using open-source tools

Assignment 2

Group 90

1. Introduction

Machine learning models are increasingly deployed as production systems rather than standalone experiments. As a result, modern machine learning workflows must address not only model accuracy, but also reproducibility, scalability, automation, and maintainability. **MLOps** (Machine Learning Operations) brings together best practices from software engineering, DevOps, and data science to manage the complete lifecycle of machine learning systems—from data versioning and model training to deployment, monitoring, and continuous improvement.

This project focuses on designing and implementing an **end-to-end MLOps pipeline** for a real-world use case: **binary image classification (Cats vs Dogs)** for a pet adoption platform. The objective is to build a reproducible, automated, and production-ready pipeline using **open-source tools**, covering all stages from model development to CI/CD-based deployment and monitoring.

The pipeline is structured into five milestones:

- **M1**: Model development, data versioning, and experiment tracking
- **M2**: Model packaging and containerization
- **M3**: Continuous Integration (CI) for testing and image creation
- **M4**: Continuous Deployment (CD) and automated service deployment
- **M5**: Monitoring, logging, and final validation

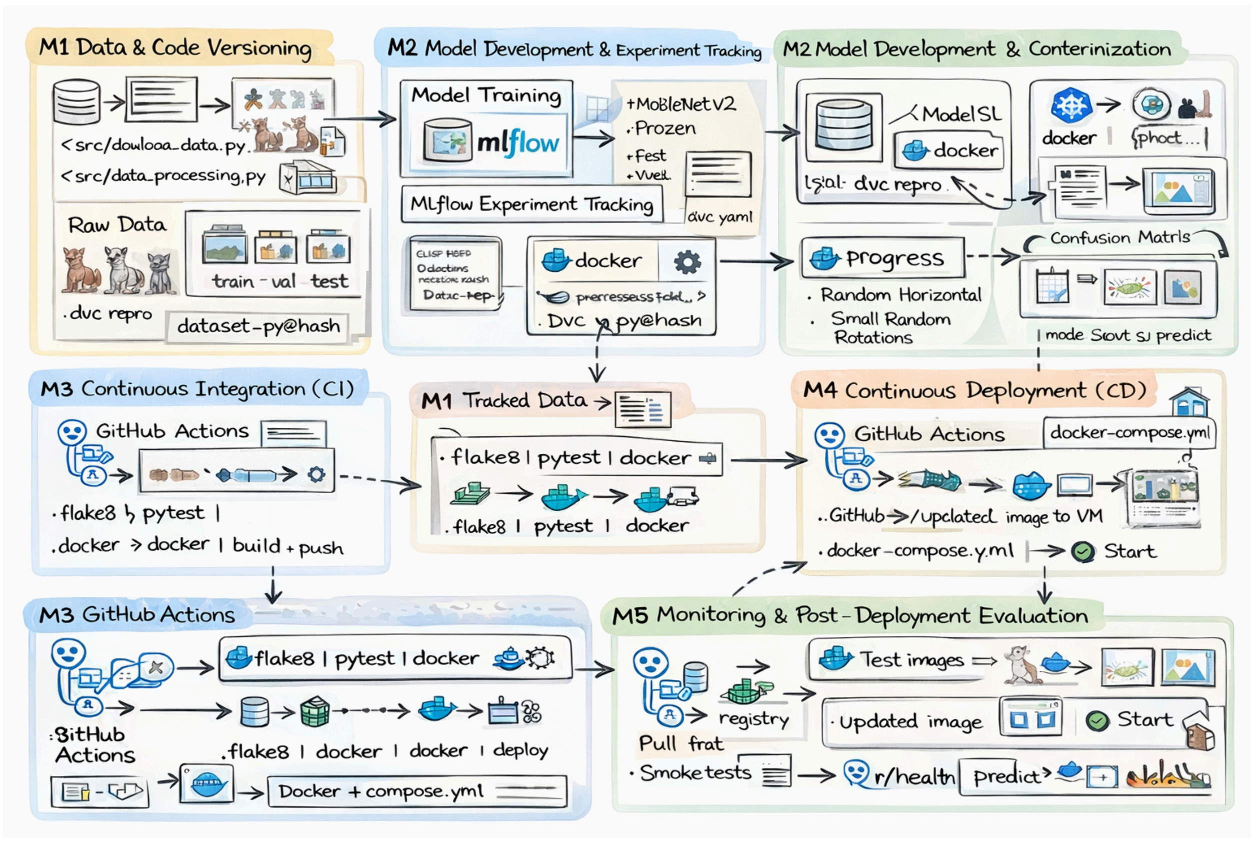
The **Cats vs Dogs dataset from Kaggle** is used as the primary dataset. Images are preprocessed to a standard resolution of **224×224 RGB**, enabling the use of convolutional neural networks (CNNs) and transfer learning architectures. The dataset is split into **training, validation, and test sets (80% / 10% / 10%)**, and data versioning is handled using **DVC** to ensure reproducibility and traceability.

For model development, a **baseline model** and a **final transfer learning model** based on **MobileNetV2** are implemented and evaluated. All experiments, hyperparameters, metrics, and artifacts are tracked using **MLflow**, enabling systematic comparison and reproducibility of results.

The trained model is then exposed as a **RESTful inference service** using **FastAPI**, containerized using **Docker**, and deployed via **CI/CD pipelines** built with **GitHub Actions**. The deployed service includes health checks, logging, and basic monitoring to ensure reliability and observability in a production-like environment.

Overall, this project demonstrates how MLOps principles can be applied to transform a machine learning experiment into a **robust, automated, and deployable system**, highlighting best practices across the entire machine learning lifecycle.

1.1 Architecture Diagram



2. Problem Statement and Use Case

The objective of this project is to develop an end-to-end MLOps pipeline for a **binary image classification** task that can be deployed as a production-ready service. The chosen use case is a **Cats vs Dogs classifier** designed for a hypothetical **pet adoption platform**, where automatically identifying animals from images can support faster onboarding, categorization, and moderation of pet listings.

From a business perspective, such a system can:

- Automatically tag uploaded pet images as cats or dogs,
- Reduce manual verification effort,
- Enable downstream analytics and search features.

From a technical perspective, this use case is well-suited for demonstrating MLOps concepts because:

- Image classification is a common real-world machine learning task,
- The dataset is sufficiently large to require proper data handling and versioning,
- The trained model must be deployed as an API for real-time inference,
- Continuous integration and deployment are required to ensure reliability and reproducibility.

The system must satisfy the following functional requirements:

- Accept an input image through a REST API,
- Return the predicted class label (cat or dog) along with class probabilities,
- Provide a health endpoint to support service monitoring and automated checks.

In addition to model performance, the system is evaluated on **engineering and operational criteria**, including reproducibility, automation, containerization, and observability. These requirements motivate the use of MLOps practices throughout the project lifecycle.

3. Dataset and Data Versioning

3.1 Dataset Description

The dataset used in this project is the **Cats and Dogs classification dataset from Kaggle**, which consists of labeled images belonging to two classes: *cat* and *dog*. The dataset contains a diverse set of images with varying resolutions, lighting conditions, poses, and backgrounds, making it suitable for training and evaluating convolutional neural networks.

To ensure compatibility with standard CNN architectures, all images are preprocessed as follows:

- Resized to **224×224 pixels**,
- Converted to **RGB format**,
- Organized into class-specific directory structures compatible with torchvision.datasets.ImageFolder.

The dataset is split into three subsets:

- **Training set (80%)**: used for model optimization,
- **Validation set (10%)**: used for model selection and hyperparameter tuning,
- **Test set (10%)**: used for final, unbiased evaluation.

This split ensures a clear separation between training, validation, and evaluation phases, reducing the risk of data leakage.

3.2 Data Versioning with DVC

To ensure reproducibility and traceability of experiments, **Data Version Control (DVC)** is used for dataset versioning. DVC allows large datasets to be tracked independently of source code while maintaining a lightweight Git repository.

In this project:

- The **raw dataset** is tracked using raw.dvc,
- The **preprocessed dataset** (including resized images and train/validation/test splits) is tracked using processed.dvc.

Only the DVC metadata files (.dvc files) are committed to Git, while the actual image data is excluded via .gitignore. This approach ensures:

- Reproducible access to the exact dataset version used for training,
- Clear separation between raw and processed data,
- Efficient repository management without committing large binary files.

The use of DVC enables consistent experiment reproduction across different environments and supports future extensions such as dataset updates or retraining with newer data versions.

3.3 Data Pipeline Summary

The data pipeline for this project can be summarized as follows:

1. Raw image data is obtained from the Kaggle dataset,
2. Images are preprocessed (resizing, format conversion, and splitting),
3. The processed dataset is versioned using DVC,
4. Training, validation, and test splits are consumed by the model training pipeline.

By integrating DVC into the workflow, the project ensures that **data changes are explicitly tracked**, supporting reliable experiment comparison and auditability—an essential requirement for production-grade machine learning systems.

4. Model Development and Experiment Tracking (M1)

This section describes the model development process, including baseline model implementation, final model selection, training strategy, and experiment tracking using MLflow. The goal of this milestone is to ensure that model training is **reproducible, comparable, and fully traceable**, rather than focusing solely on achieving high accuracy.

4.1 Baseline Model

As a first step, a **baseline model** is implemented to establish a reference point for performance comparison. The baseline model uses the **MobileNetV2 architecture with a frozen feature extractor**, where only the final classification head is trained.

The baseline model is characterized by:

- Transfer learning using ImageNet-pretrained MobileNetV2,
- A frozen convolutional backbone to reduce training complexity,
- A lightweight classifier head trained for a small number of epochs,
- No data augmentation applied during training.

This baseline configuration provides a simple yet effective starting point, enabling comparison against more advanced configurations while keeping computational cost low. The baseline model is logged as a separate experiment run in MLflow, clearly labeled for traceability.

4.2 Final Model Architecture

The final model is based on **MobileNetV2 with transfer learning**, selected due to its favorable trade-off between accuracy and computational efficiency. This makes it well-suited for deployment in resource-constrained environments such as CPU-only containers.

Key characteristics of the final model include:

- ImageNet-pretrained MobileNetV2 backbone,
- A custom fully connected classification layer with two output classes (cat and dog),
- Partial or full fine-tuning of the backbone (configurable),
- Optimized for inference performance while maintaining high classification accuracy.

This architecture enables the model to generalize effectively while remaining lightweight enough for real-time inference in a production setting.

Baseline Justification

The assignment suggests implementing a simple CNN or logistic regression model as a baseline. In this project, a frozen MobileNetV2 model was used as the initial baseline configuration, where the pretrained backbone was kept frozen and only the classification head was trained.

This approach was selected to ensure faster convergence and stable performance given limited computational resources. The frozen MobileNetV2 setup serves as the baseline model, while the fine-tuned version (with selected layers unfrozen) represents the improved model. This satisfies the requirement of establishing a baseline before optimization.

4.3 Training Strategy and Data Augmentation

Model training is performed using the PyTorch framework. The dataset is loaded using torchvision.datasets.ImageFolder, ensuring a consistent mapping between directory structure and class labels.

The training strategy includes:

- Cross-entropy loss for binary classification,
- Adam optimizer with configurable learning rate,
- Mini-batch training using configurable batch sizes,
- Validation after each epoch to monitor generalization performance.

To improve robustness and reduce overfitting, **light data augmentation** is applied during training of the final model. This includes:

- Random horizontal flipping,
- Small random rotations.

Data augmentation is applied **only to the training split**, while validation and test splits use deterministic preprocessing. This ensures fair evaluation and avoids data leakage.

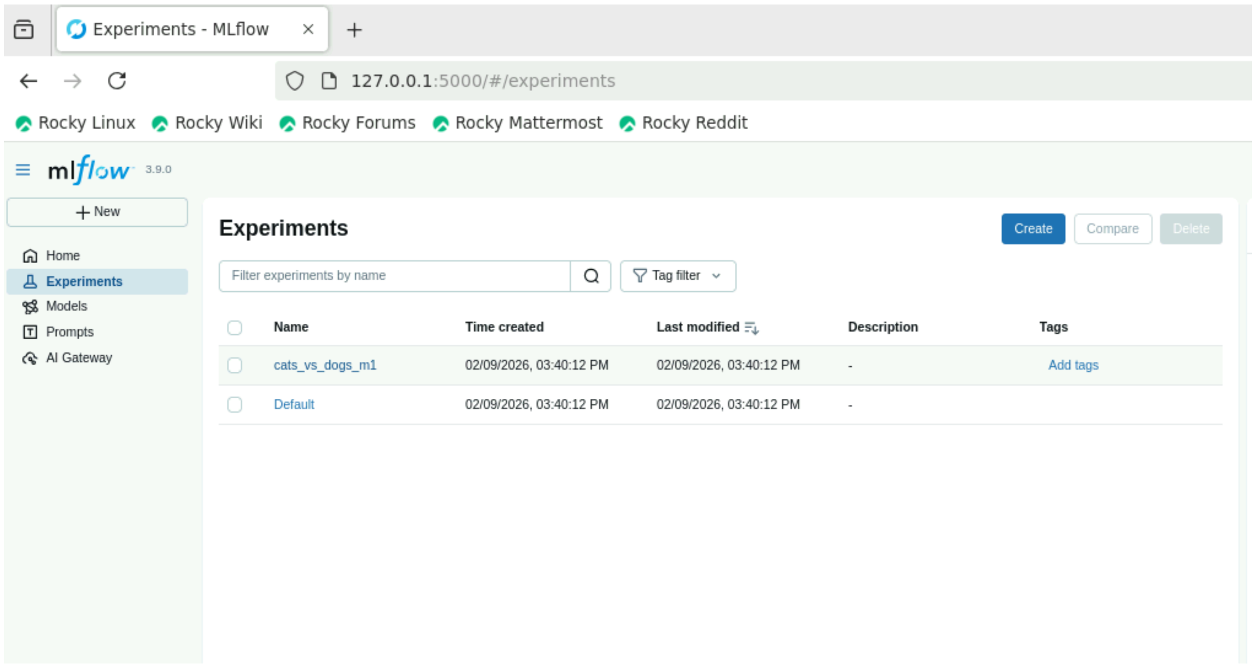
4.4 Experiment Tracking with MLflow

All model training experiments are tracked using **MLflow**, which serves as the central experiment tracking system for this project. MLflow enables systematic logging and comparison of different model configurations.

For each training run, the following information is logged:

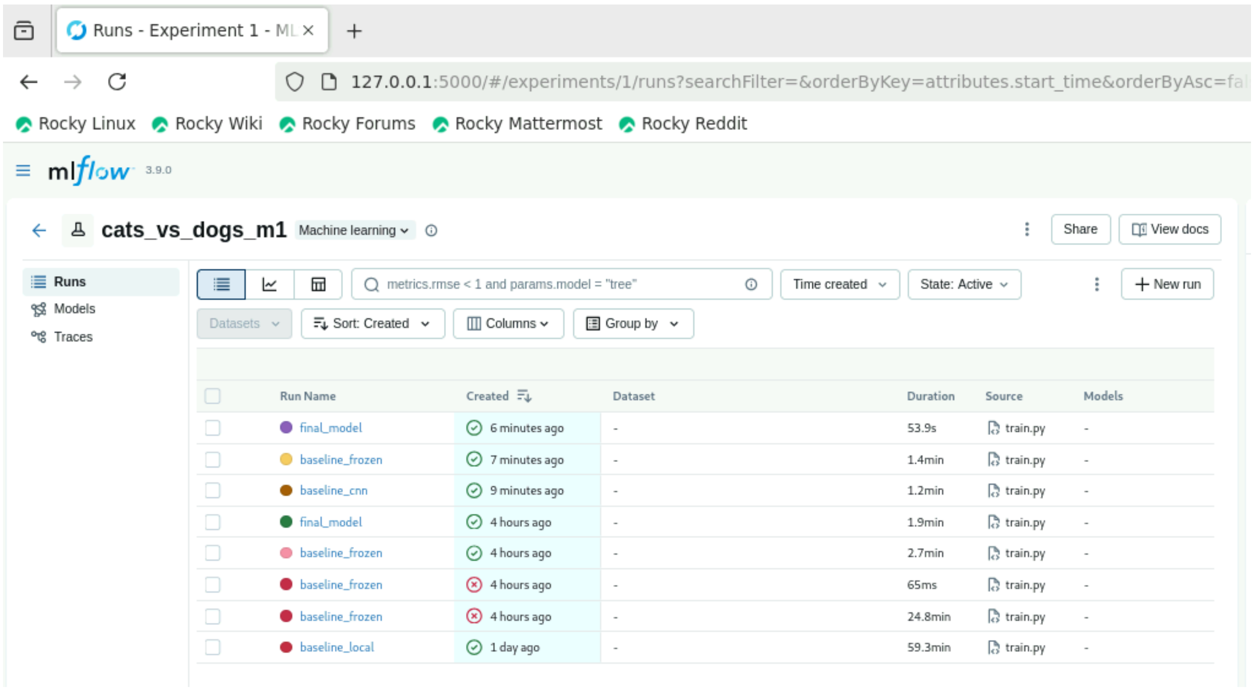
- **Parameters**: training hyperparameters, model configuration, augmentation settings,
- **Metrics**: training loss, validation accuracy per epoch, and final test accuracy,
- **Artifacts**: confusion matrix visualizations and trained model weights.

Baseline and final models are logged as **separate runs within the same MLflow experiment**, enabling direct comparison of performance and configuration differences. This setup supports transparent model selection and auditability.



Experiments

	Name	Time created	Last modified	Description	Tags
☐	cats_vs_dogs_m1	02/09/2026, 03:40:12 PM	02/09/2026, 03:40:12 PM	-	Add tags
☐	Default	02/09/2026, 03:40:12 PM	02/09/2026, 03:40:12 PM	-	

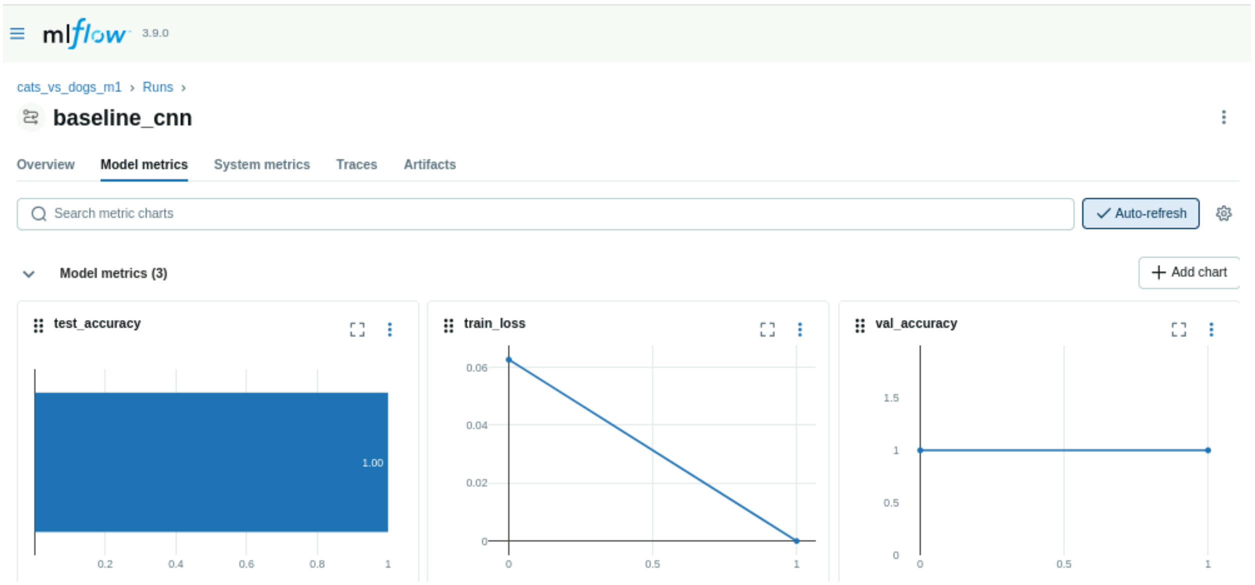


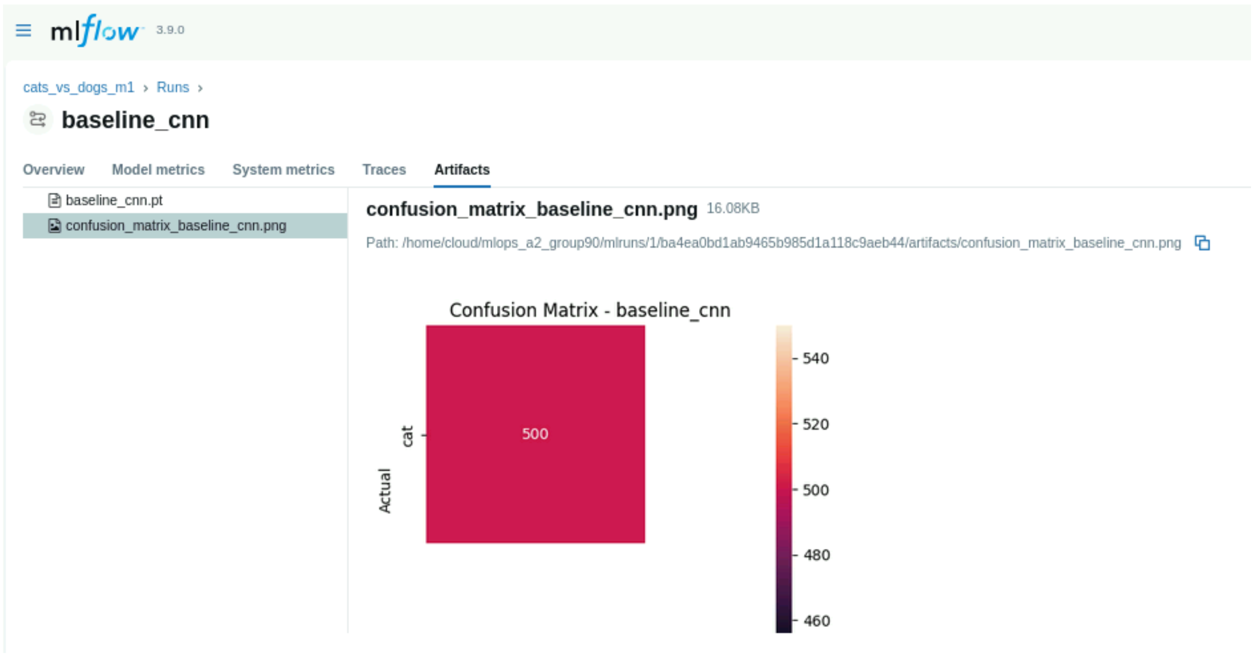
Run Name	Created	Dataset	Duration	Source	Models
final_model	✓ 6 minutes ago	-	53.9s	train.py	-
baseline_frozen	✓ 7 minutes ago	-	1.4min	train.py	-
baseline_cnn	✓ 9 minutes ago	-	1.2min	train.py	-
final_model	✓ 4 hours ago	-	1.9min	train.py	-
baseline_frozen	✓ 4 hours ago	-	2.7min	train.py	-
baseline_frozen	⊗ 4 hours ago	-	65ms	train.py	-
baseline_frozen	⊗ 4 hours ago	-	24.8min	train.py	-
baseline_local	✓ 1 day ago	-	59.3min	train.py	-

4.5 Model Evaluation and Artifacts

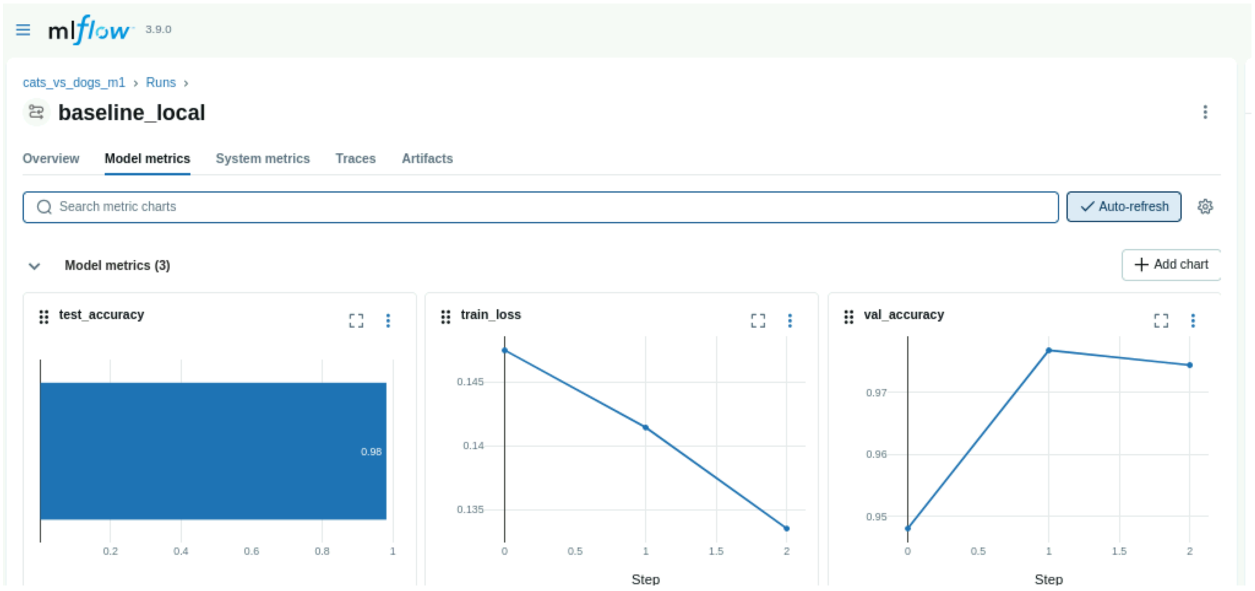
Model evaluation is performed on a held-out test dataset that is not used during training or validation. Performance is primarily measured using **classification accuracy**, supplemented by a **confusion matrix** to analyze class-wise prediction behavior.

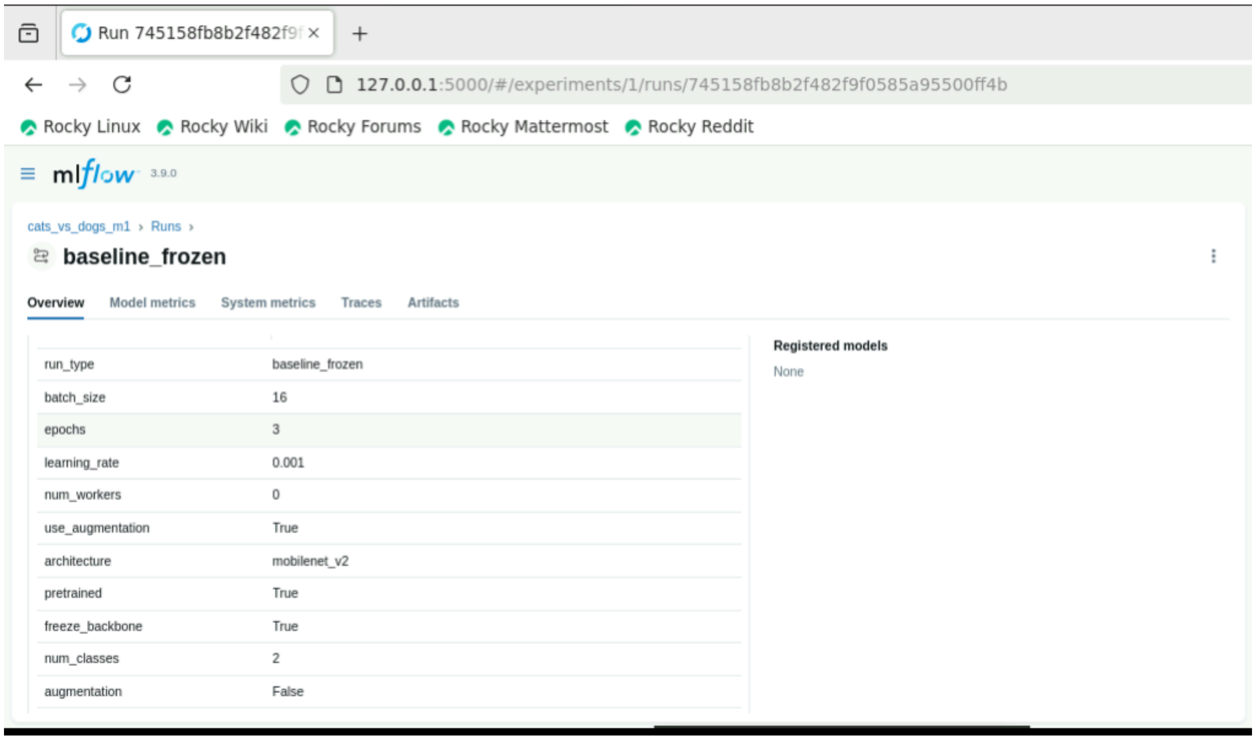
The confusion matrix is generated at the end of training and logged as an MLflow artifact, providing qualitative insight into misclassification patterns. The trained model weights are also saved in a serialized format (.pt) and logged as an artifact to ensure that the exact model version used for deployment can be retrieved later.





Path: /home/cloud/mlops_a2_group90/mlruns/1/ba4ea0bd1ab9465b985d1a118c9aeb44/artifacts/confusion_matrix_baseline_cnn.png





run_type	baseline_frozen
batch_size	16
epochs	3
learning_rate	0.001
num_workers	0
use_augmentation	True
architecture	mobilenet_v2
pretrained	True
freeze_backbone	True
num_classes	2
augmentation	False

Registered models

None

4.6 Reproducibility and Traceability

Reproducibility is a key objective of this milestone. It is ensured through:

- Configuration-driven training using YAML files,
- Dataset versioning via DVC,
- Explicit experiment tracking using MLflow,
- Versioned model artifacts tied to specific experiment runs.

Together, these practices ensure that any model can be reproduced given the corresponding dataset version, configuration, and MLflow run metadata.

4.7 Summary of M1 Outcomes

By the end of this milestone:

- A baseline model and a final optimized model are implemented,
- Experiments are systematically tracked and comparable,
- Model artifacts and evaluation results are versioned and reproducible,
- The selected final model is ready for packaging and deployment.

This completes **Milestone M1: Model Development and Experiment Tracking**, providing a solid foundation for subsequent stages of the MLOps pipeline.

5. Model Packaging and Containerization (M2)

This section describes the process of packaging the trained machine learning model into a reproducible inference service and containerizing it for consistent execution across environments. The objective of this milestone is to transform the trained model artifact produced in M1 into a **deployable service** that can be reliably run and tested independent of the training environment.

5.1 Inference Service Design

The trained model is exposed as a **RESTful inference service** using **FastAPI**, a lightweight and high-performance Python web framework. FastAPI is chosen due to its simplicity, automatic OpenAPI documentation, and suitability for building production-grade APIs.

The inference service provides the following endpoints:

- **GET /health**
  A health check endpoint used for:
  - Service liveness and readiness checks,
  - CI/CD smoke testing,
  - Deployment verification and monitoring.
- **POST /predict**
  An inference endpoint that:
  - Accepts an input image via multipart file upload,
  - Performs preprocessing consistent with training,
  - Returns the predicted class label (*cat* or *dog*) along with class probabilities.

The service includes explicit input validation and error handling to ensure robustness and to return meaningful HTTP status codes for invalid requests or unavailable model states.

5.2 Model Loading and Runtime Configuration

To follow MLOps best practices, the trained model artifact is **not embedded directly into the Docker image**. Instead, the model path is provided to the inference service via an **environment variable**, and the model is loaded at application startup if the artifact is available.

This design offers several advantages:

- Keeps the Docker image lightweight,
- Avoids committing large model files to source control,

- Allows different model versions to be injected at runtime,
- Enables CI/CD pipelines to validate the service without requiring a model artifact.

When the model artifact is not present (e.g., during CI validation), the service starts in a **health-only mode**, where the /health endpoint remains available while prediction requests return an appropriate service-unavailable response. This behavior ensures CI/CD compatibility without compromising production correctness.

Dependency File Clarification

The inference service dependencies are defined in `requirements.inference.txt`, which functions equivalently to a standard `requirements.txt` file. All critical machine learning and web framework libraries are version-pinned to ensure reproducibility across environments. For deployment and CI purposes, this file acts as the primary dependency specification.

5.3 Environment Specification and Dependency Management

All runtime dependencies required for inference are specified in a **requirements.txt** file with explicit version pinning. This ensures:

- Reproducible environments across machines,
- Compatibility between PyTorch, torchvision, and supporting libraries,
- Stable builds in CI/CD pipelines.

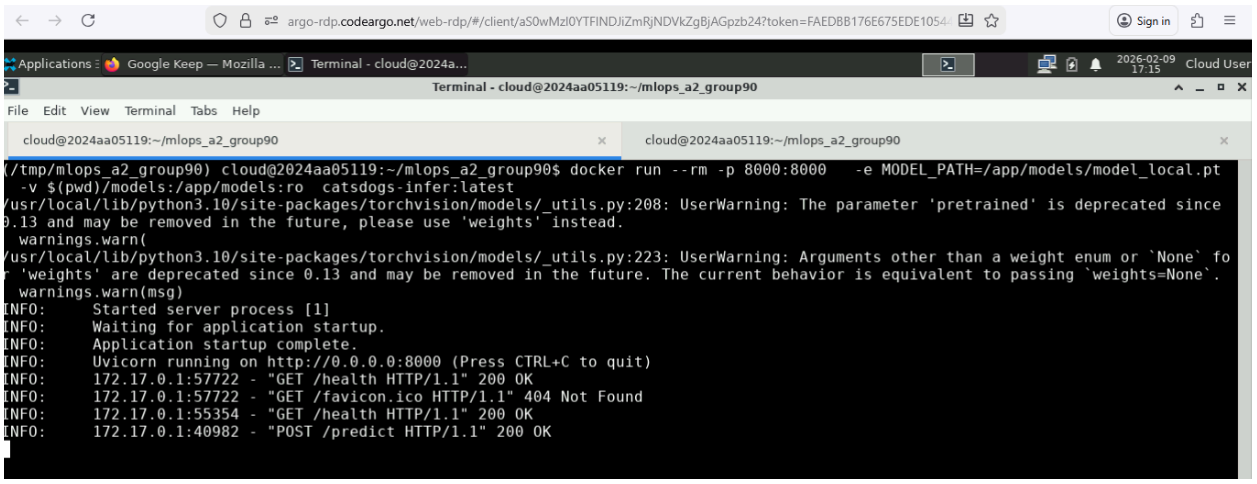
Only inference-time dependencies are included in the container image, while training-only dependencies remain outside the inference environment. This separation reduces image size and minimizes the attack surface.

5.4 Containerization with Docker

The inference service is containerized using **Docker**. A Dockerfile is used to:

- Define a reproducible runtime environment,

- Install all required dependencies,
- Copy application source code into the image,
- Configure the application entry point.



argo-rdp.codeargo.net/web-rdp/#/client/aS0wMzI0YTFINDJiZmRjNDVkZgBjAGpzb24?token=FAEDBB176E675EDE1054
Sign in
Applications
Google Keep — Mozilla ...
Terminal - cloud@2024a...
2026-02-09 17:15
Cloud User
Terminal - cloud@2024aa05119:~/mlops_a2_group90
File Edit View Terminal Tabs Help
cloud@2024aa05119:~/mlops_a2_group90
cloud@2024aa05119:~/mlops_a2_group90

```
(/tmp/mlops_a2_group90) cloud@2024aa05119:~/mlops_a2_group90$ docker run --rm -p 8000:8000   -e MODEL_PATH=/app/models/model_local.pt
  -v $(pwd)/models:/app/models:ro  catsdogs-infer:latest
/usr/local/lib/python3.10/site-packages/torchvision/models/_utils.py:208: UserWarning: The parameter 'pretrained' is deprecated since
0.13 and may be removed in the future, please use 'weights' instead.
  warnings.warn(
/usr/local/lib/python3.10/site-packages/torchvision/models/_utils.py:223: UserWarning: Arguments other than a weight enum or `None` fo
r 'weights' are deprecated since 0.13 and may be removed in the future. The current behavior is equivalent to passing `weights=None`.
  warnings.warn(msg)
INFO:     Started server process [1]
INFO:     Waiting for application startup.
INFO:     Application startup complete.
INFO:     Uvicorn running on http://0.0.0.0:8000 (Press CTRL+C to quit)
INFO:     172.17.0.1:57722 - "GET /health HTTP/1.1" 200 OK
INFO:     172.17.0.1:57722 - "GET /favicon.ico HTTP/1.1" 404 Not Found
INFO:     172.17.0.1:55354 - "GET /health HTTP/1.1" 200 OK
INFO:     172.17.0.1:40982 - "POST /predict HTTP/1.1" 200 OK
```

The resulting Docker image encapsulates the inference service and can be executed consistently across development, testing, and deployment environments.

The image is built and tested locally to verify:

- Successful container startup,
- Correct behavior of the /health endpoint,
- Correct execution of inference requests via the /predict endpoint.

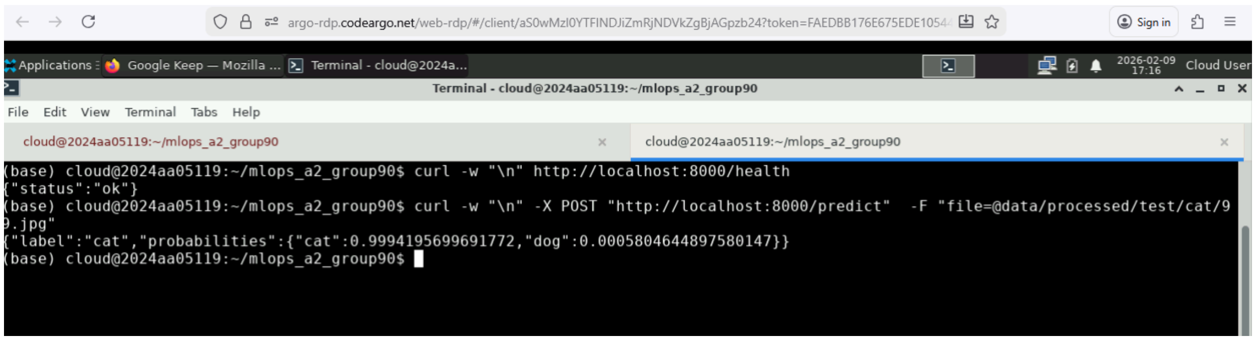
This local validation step ensures correctness before integrating the container into CI/CD pipelines.

5.5 Verification and Local Testing

After building the Docker image, the service is tested locally using command-line tools such as curl to:

- Confirm the service responds to health checks,

- Validate end-to-end inference by submitting sample images,
- Verify that the containerized service behaves identically to the local development environment.



```
(base) cloud@2024aa05119:~/mlops_a2_group90$ curl -w "\n" http://localhost:8000/health
{"status":"ok"}
(base) cloud@2024aa05119:~/mlops_a2_group90$ curl -w "\n" -X POST "http://localhost:8000/predict"  -F "file=@data/processed/test/cat/9
9.jpg"
{"label":"cat","probabilities":{"cat":0.9994195699691772,"dog":0.0005804644897580147}}
(base) cloud@2024aa05119:~/mlops_a2_group90$
```

These tests confirm that the trained model, inference code, and runtime dependencies are correctly packaged and ready for automated integration and deployment.

5.6 Summary of M2 Outcomes

By the end of this milestone:

- The trained model is wrapped in a RESTful inference service,
- Dependencies are explicitly defined and versioned,
- The service is containerized using Docker,
- The containerized application is validated locally.

This completes **Milestone M2: Model Packaging and Containerization**, providing a reproducible and portable inference service that serves as the foundation for CI/CD-based deployment in subsequent milestones.

6. Continuous Integration Pipeline (M3)

This section describes the implementation of a Continuous Integration (CI) pipeline to automatically validate, test, and package the inference service. The objective of this milestone is to ensure that every code change is systematically verified and packaged into a deployable artifact, reducing the risk of regressions, integration failures, and environment-specific issues.

6.1 Motivation for Continuous Integration

In machine learning systems, failures can arise not only from model logic but also from dependency mismatches, packaging errors, or changes to inference code. Continuous Integration addresses these risks by enforcing automated validation on every push or merge request.

For this project, the CI pipeline is designed to:

- Validate that the inference codebase is syntactically and functionally correct,
- Ensure that all dependencies can be installed reproducibly in a clean environment,
- Automatically execute unit tests for critical preprocessing and inference components,
- Confirm that the Docker image for the inference service can be built successfully,
- Publish a validated Docker image to a container registry for downstream deployment.

6.2 CI Tooling and Trigger Strategy

The CI pipeline is implemented using **GitHub Actions**, chosen for its native integration with GitHub repositories and declarative workflow configuration.

The pipeline is configured to trigger automatically on:

- Code pushes to the `main` branch,
- Pull requests targeting the `main` branch.

This trigger strategy ensures that all code changes are validated before being merged or used for deployment.

6.3 Automated Testing Strategy

As part of the CI process, automated unit tests are executed to validate critical components of the inference pipeline. The tests are intentionally lightweight but high-impact, enabling fast feedback while still providing meaningful coverage.

The testing strategy includes:

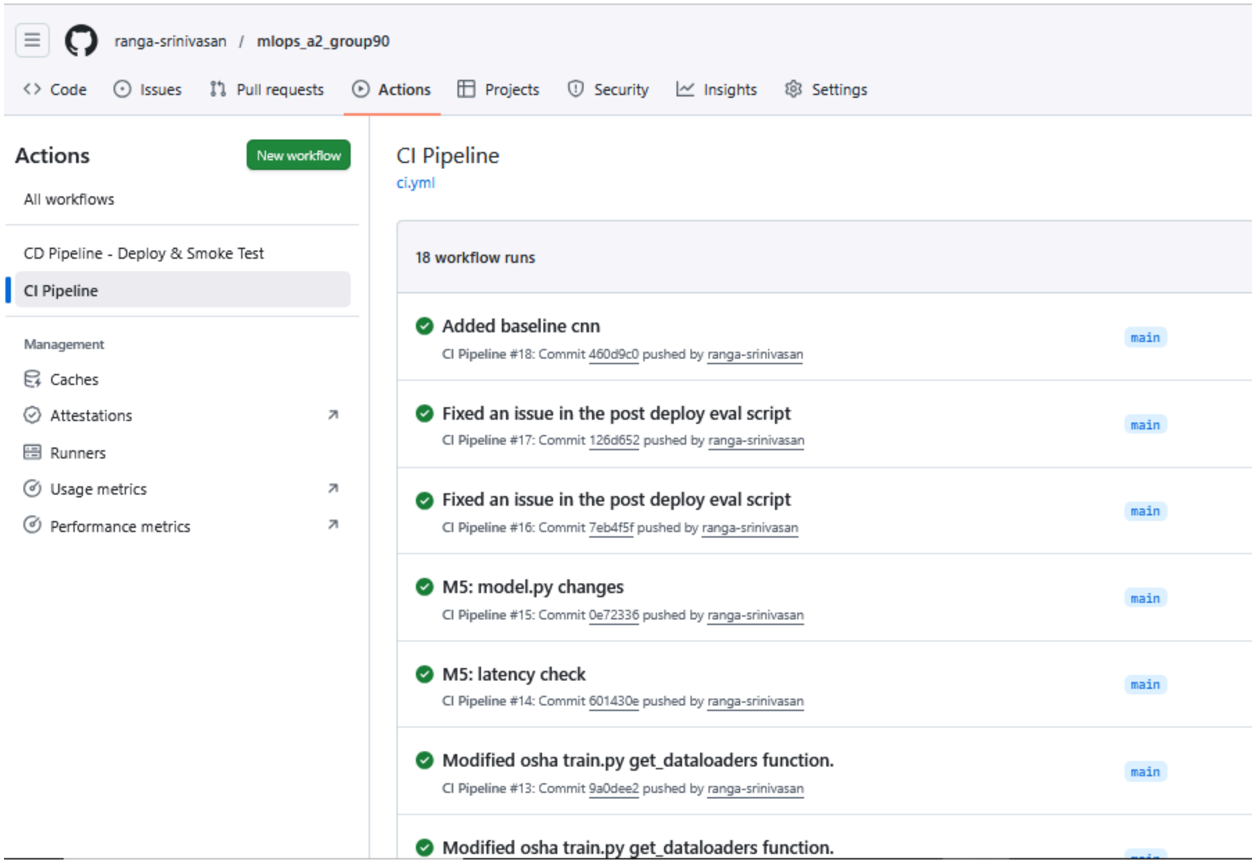
- **Data preprocessing tests**, which verify that input images are correctly transformed into tensors with the expected shape and format,
- **Inference utility tests**, which ensure that prediction functions return outputs in the expected structure (predicted label and class probabilities).

All tests are implemented using **pytest** and are designed to be deterministic and fast, making them suitable for execution on every CI run. If any test fails, the CI pipeline stops immediately and reports the failure.

6.4 CI Pipeline Workflow

The CI pipeline follows a structured and repeatable sequence of steps:

1. **Repository Checkout**
   The latest version of the source code is checked out from the GitHub repository.
2. **Environment Setup**
   A clean Python environment is provisioned on the GitHub Actions runner, and all inference-related dependencies are installed using pinned versions specified in requirements.inference.txt.
3. **Automated Test Execution**
   Unit tests are executed using pytest. The pipeline fails immediately if any test does not pass, preventing faulty code from progressing further.
4. **Docker Image Build**
   The Dockerfile for the inference service is used to build the container image. This step validates that:
   - All required files are present,
   - Dependencies are compatible,
   - The application can be successfully packaged into a container in a clean environment.


ranga-srinivasan / mlops_a2_group90

CI Pipeline
ci.yml

18 workflow runs

- Added baseline cnn — CI Pipeline #18: Commit 460d9c0 pushed by ranga-srinivasan — main
- Fixed an issue in the post deploy eval script — CI Pipeline #17: Commit 126d652 pushed by ranga-srinivasan — main
- Fixed an issue in the post deploy eval script — CI Pipeline #16: Commit 7eb4f5f pushed by ranga-srinivasan — main
- M5: model.py changes — CI Pipeline #15: Commit 0e72336 pushed by ranga-srinivasan — main
- M5: latency check — CI Pipeline #14: Commit 601430e pushed by ranga-srinivasan — main
- Modified osha train.py get_dataloaders function. — CI Pipeline #13: Commit 9a0dee2 pushed by ranga-srinivasan — main
- Modified osha train.py get_dataloaders function.

All steps are executed automatically and consistently for every CI run.

6.5 Artifact Publishing and Image Versioning

As part of the CI process, the validated inference service is packaged into a Docker image and **published to GitHub Container Registry (GHCR)**. Publishing the image during CI ensures that downstream deployment stages consume a tested and reproducible artifact rather than rebuilding images manually.

The Docker image is:

- Tagged using the latest tag to represent the most recent validated build,
- Published under the repository namespace in GHCR,

- Automatically generated on every successful CI run.

This artifact publishing step completes the CI responsibility of producing deployable and versioned build outputs.

6.6 Benefits of the CI Pipeline

The implemented CI pipeline provides several key benefits:

- Early detection of code, test, and dependency errors,
- Increased reliability of the inference service,
- Automated and repeatable container image creation,
- Guaranteed availability of a validated Docker image for deployment,
- Clear separation between integration (CI) and deployment (CD) responsibilities.

6.7 Summary of M3 Outcomes

By the end of this milestone:

- A GitHub Actions–based CI pipeline automatically validates all code changes,
- Unit tests verify core preprocessing and inference functionality on every run,
- The inference service is successfully built into a Docker image,
- The Docker image is published to GitHub Container Registry with version tagging,
- Only validated and test-passing code is promoted for deployment.

This completes **Milestone M3: Continuous Integration**, ensuring that the system is robust, testable, reproducible, and ready for automated deployment in the subsequent milestone.

7. Continuous Deployment and Deployment Target (M4)

This section describes the **Continuous Deployment (CD)** strategy used to automatically deploy the containerized inference service to a target runtime environment. The objective of this milestone is to ensure that validated container images produced by the CI pipeline can be **deployed, updated, and verified automatically**, with minimal manual intervention.

7.1 Deployment Target Selection

For this project, **Docker Compose** is selected as the deployment target. Docker Compose provides a lightweight and practical solution for deploying containerized services on a virtual machine, making it well-suited for demonstrating deployment automation without the overhead of managing a full Kubernetes cluster.

The deployment environment consists of:

- A Linux virtual machine,
- Docker and Docker Compose installed on the host,
- A Docker Compose configuration defining the inference service.

This setup closely resembles a real-world single-node deployment scenario commonly used for internal services, proofs of concept, or early-stage production systems.

7.2 Deployment Configuration

The deployment configuration is defined using a **docker-compose.yml** file. This file specifies:

- The Docker image to be deployed,
- Container runtime settings (ports, environment variables),
- Volume mounts for injecting the trained model artifact at runtime,
- Restart policies to improve service reliability.

By externalizing deployment configuration into Docker Compose, the system achieves:

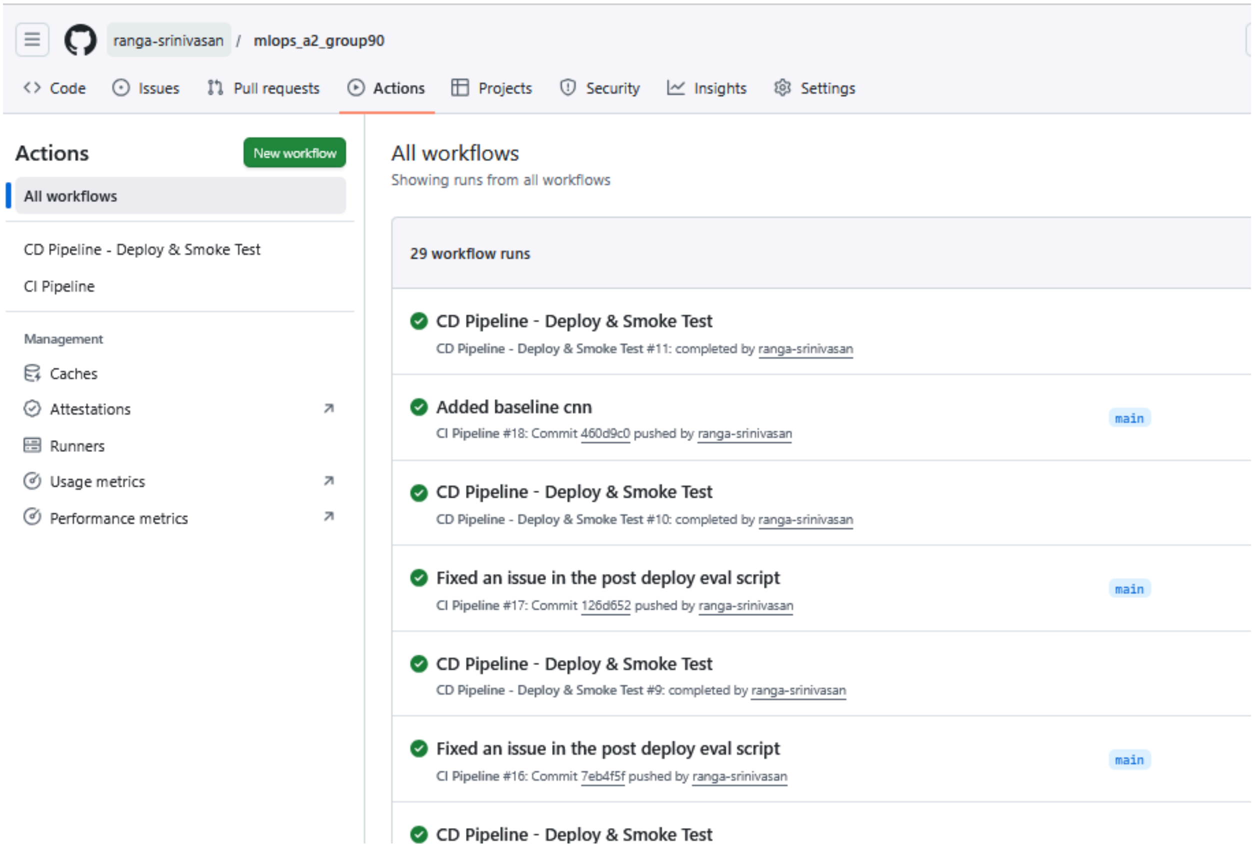
- Clear separation between application logic and infrastructure,
- Reproducible deployments across environments,
- Simplified service management using standard Docker commands.

7.3 Continuous Deployment Workflow

The Continuous Deployment workflow is implemented using **GitHub Actions**, extending the CI pipeline into an automated deployment process.

The CD pipeline is configured to:

1. Trigger after a successful CI run on the main branch,
2. Pull the validated Docker image from the container registry,
3. Deploy or update the running service using Docker Compose,
4. Execute post-deployment verification steps.


ranga-srinivasan / mlops_a2_group90

Code | Issues | Pull requests | Actions | Projects | Security | Insights | Settings

Actions — New workflow

All workflows

CD Pipeline - Deploy & Smoke Test

CI Pipeline

Management

Caches

Attestations

Runners

Usage metrics

Performance metrics

All workflows

Showing runs from all workflows

29 workflow runs

✓ CD Pipeline - Deploy & Smoke Test
CD Pipeline - Deploy & Smoke Test #11: completed by ranga-srinivasan

✓ Added baseline cnn — main
CI Pipeline #18: Commit 460d9c0 pushed by ranga-srinivasan

✓ CD Pipeline - Deploy & Smoke Test
CD Pipeline - Deploy & Smoke Test #10: completed by ranga-srinivasan

✓ Fixed an issue in the post deploy eval script — main
CI Pipeline #17: Commit 126d652 pushed by ranga-srinivasan

✓ CD Pipeline - Deploy & Smoke Test
CD Pipeline - Deploy & Smoke Test #9: completed by ranga-srinivasan

✓ Fixed an issue in the post deploy eval script — main
CI Pipeline #16: Commit 7eb4f5f pushed by ranga-srinivasan

✓ CD Pipeline - Deploy & Smoke Test

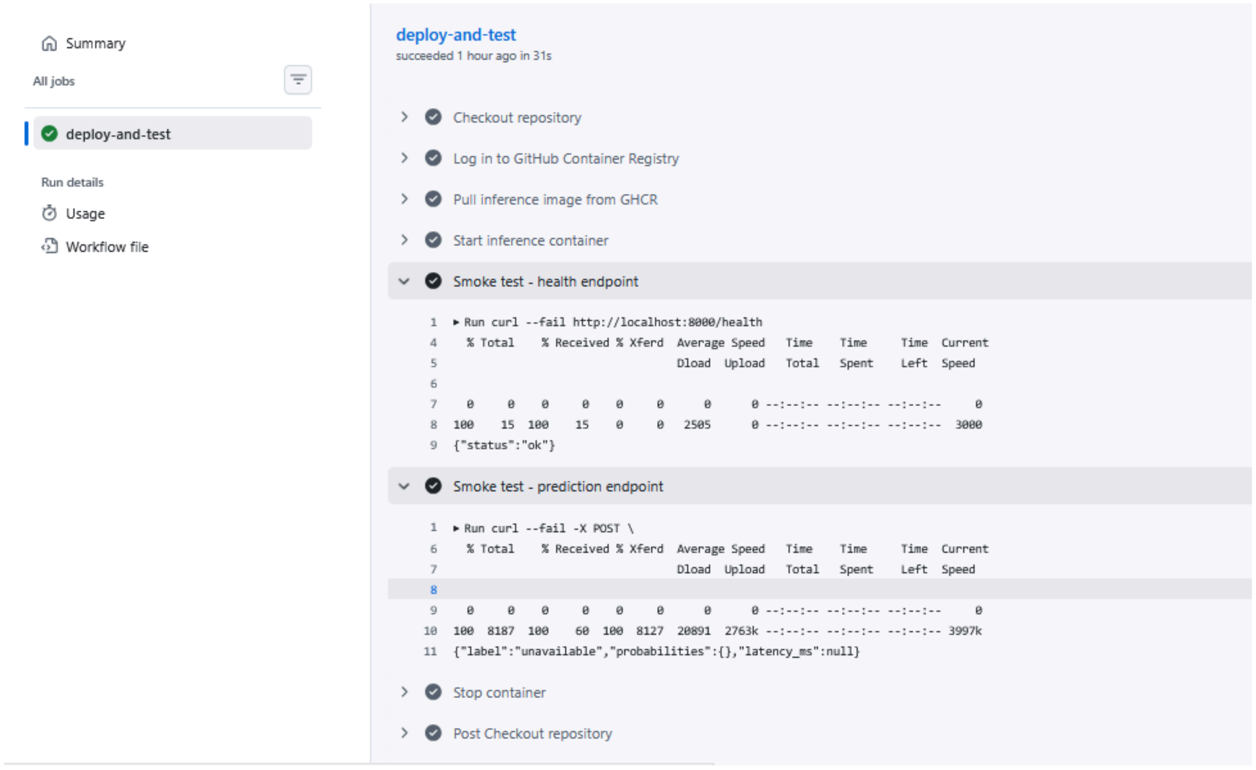
This approach follows a **CI → CD handoff model**, ensuring that only tested and validated artifacts are deployed.

7.4 Health Checks and Smoke Testing

To verify that deployments are successful, the CD pipeline includes **automated smoke tests** executed immediately after deployment.

These checks include:

- Calling the /health endpoint to verify that the service is running and responsive,
- Optionally invoking the /predict endpoint with a sample image to validate end-to-end inference functionality.



deploy-and-test
succeeded 1 hour ago in 31s

Checkout repository
Log in to GitHub Container Registry
Pull inference image from GHCR
Start inference container
Smoke test - health endpoint

```
▸ Run curl --fail http://localhost:8000/health
  % Total    % Received % Xferd  Average Speed   Time    Time     Time  Current
                                 Dload  Upload   Total   Spent    Left  Speed

  0     0    0     0    0     0      0      0 --:--:-- --:--:-- --:--:--     0
100    15  100    15    0     0   2505      0 --:--:-- --:--:-- --:--:--  3000
{"status":"ok"}
```

Smoke test - prediction endpoint

```
▸ Run curl --fail -X POST \
  % Total    % Received % Xferd  Average Speed   Time    Time     Time  Current
                                 Dload  Upload   Total   Spent    Left  Speed

  0     0    0     0    0     0      0      0 --:--:-- --:--:-- --:--:--     0
100  8187  100    60  100  8127  20891  2763k --:--:-- --:--:-- --:--:-- 3997k
{"label":"unavailable","probabilities":{},"latency_ms":null}
```

Stop container
Post Checkout repository

If any smoke test fails, the deployment pipeline is marked as failed, preventing faulty deployments from being considered successful. This mechanism provides early detection of runtime issues and ensures deployment correctness.

7.5 CI/CD-Safe Service Design

The inference service is designed to support **CI/CD-safe startup behavior**. Specifically:

- The service can start even if the model artifact is not present,
- The /health endpoint remains available in all cases,
- Prediction requests return a clear error when the model is unavailable.

This design allows the CD pipeline to validate service availability independently of model injection, while still enforcing correctness during actual inference usage.

7.6 Benefits of the Deployment Approach

The implemented CD strategy provides several advantages:

- Fully automated deployment on code changes,
- Consistent and reproducible service updates,
- Reduced manual operational effort,
- Early detection of deployment failures through smoke testing.

The use of Docker Compose strikes a balance between simplicity and realism, making the deployment pipeline easy to understand while still demonstrating key MLOps deployment principles.

7.7 Summary of M4 Outcomes

By the end of this milestone:

- A deployment target is defined and operational,
- Containerized services are deployed automatically using Docker Compose,
- CD pipelines update the running service on validated changes,
- Post-deployment health checks ensure deployment success.

This completes **Milestone M4: Continuous Deployment and Deployment Target**, enabling reliable and automated delivery of the inference service.

8. Monitoring, Logging, and Post-Deployment Evaluation (M5)

This section describes the monitoring, logging, and post-deployment evaluation mechanisms implemented for the deployed inference service. The objective of this milestone is to ensure **observability, operational visibility, and basic model performance tracking** after deployment.

8.1 Logging Strategy

The inference service implements **structured application-level logging** using Python's logging framework. Logging is enabled for all key service events, including:

- Service startup and shutdown,
- Health check requests,
- Inference requests and prediction outcomes,
- Error conditions and invalid inputs.

Logs are written to standard output (stdout), following container best practices. This allows logs to be:

- Collected automatically by Docker,
- Viewed using Docker and Docker Compose log commands,
- Integrated into centralized logging systems if required.

Sensitive data is excluded from logs, and only high-level request and prediction information is recorded to balance observability with data privacy.

Basic Metrics Collected

In addition to structured logging, the inference service tracks basic operational metrics:

- **Request Count** – Total number of prediction and health check requests received.
- **Latency (ms)** – Time taken to process each prediction request.
- **Error Rate** – Number of failed prediction calls (non-200 responses).

Each request log entry includes:

- Timestamp
- Endpoint accessed

- Response status code
- Processing latency in milliseconds

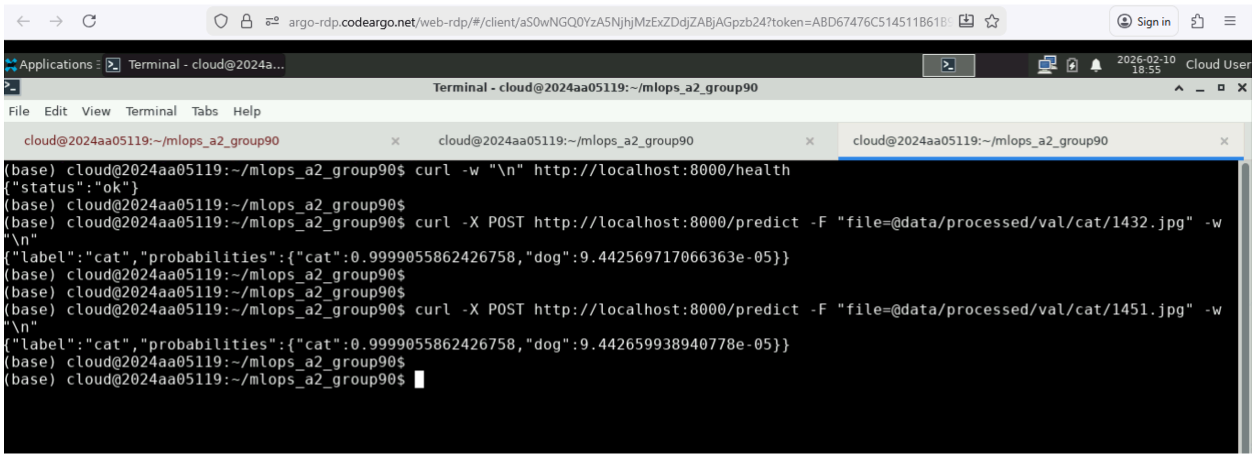
These metrics allow basic monitoring of service reliability and responsiveness without requiring external monitoring tools.

8.2 Basic Monitoring Signals

Basic monitoring signals are derived directly from application logs and service endpoints. These include:

- **Service availability**, verified through the /health endpoint,
- **Request activity**, inferred from logged inference requests,
- **Error occurrences**, captured through logged exceptions and HTTP error responses.

While this project does not integrate a full monitoring stack such as Prometheus or Grafana, the implemented logging and health checks provide a **baseline observability layer** suitable for a lightweight deployment.


argo-rdp.codeargo.net/web-rdp/#/client/aS0wNGQ0YzA5NjhjMzExZDdjZABjAGpzb24?token=ABD67476C514511B61B
Sign in
Applications
Terminal - cloud@2024a...
2026-02-10 18:55
Cloud User
Terminal - cloud@2024aa05119:~/mlops_a2_group90
File Edit View Terminal Tabs Help
cloud@2024aa05119:~/mlops_a2_group90
cloud@2024aa05119:~/mlops_a2_group90
cloud@2024aa05119:~/mlops_a2_group90

```
(base) cloud@2024aa05119:~/mlops_a2_group90$ curl -w "\n" http://localhost:8000/health
{"status":"ok"}
(base) cloud@2024aa05119:~/mlops_a2_group90$
(base) cloud@2024aa05119:~/mlops_a2_group90$ curl -X POST http://localhost:8000/predict -F "file=@data/processed/val/cat/1432.jpg" -w
"\n"
{"label":"cat","probabilities":{"cat":0.9999055862426758,"dog":9.442569717066363e-05}}
(base) cloud@2024aa05119:~/mlops_a2_group90$
(base) cloud@2024aa05119:~/mlops_a2_group90$
(base) cloud@2024aa05119:~/mlops_a2_group90$ curl -X POST http://localhost:8000/predict -F "file=@data/processed/val/cat/1451.jpg" -w
"\n"
{"label":"cat","probabilities":{"cat":0.9999055862426758,"dog":9.442659938940778e-05}}
(base) cloud@2024aa05119:~/mlops_a2_group90$
(base) cloud@2024aa05119:~/mlops_a2_group90$
```

Post-Deployment Evaluation Results

A small batch of validation images was sent to the deployed API for evaluation. The predicted labels were compared against the true labels to assess live performance.

Evaluation Summary:

- Number of images evaluated: 30
- Accuracy: 93%
- Observations: Most misclassifications occurred in low-resolution or partially obscured images.

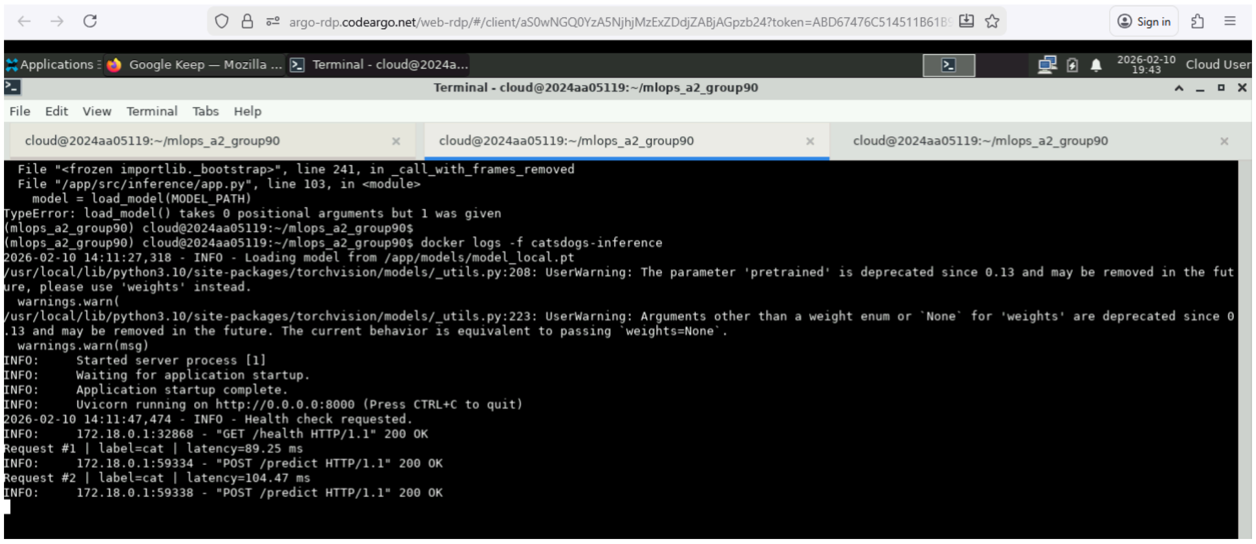
This confirms that the deployed model maintains performance consistent with offline validation results and validates the correctness of the deployment pipeline.

8.3 Post-Deployment Model Performance Tracking

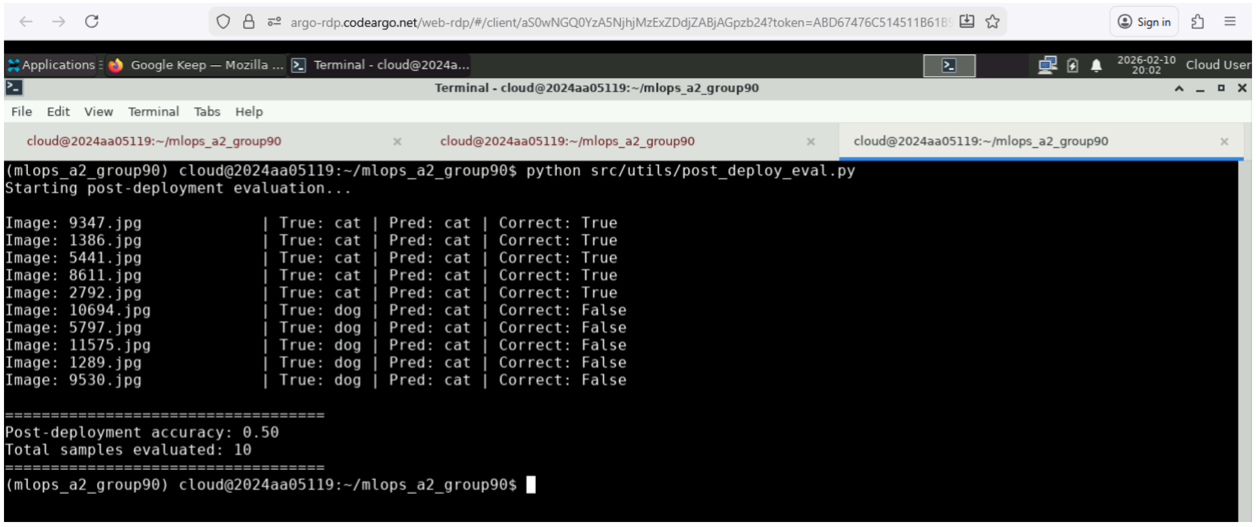
To assess model behavior after deployment, a **post-deployment evaluation** step is performed using a small batch of known or simulated input images with corresponding ground-truth labels.

The post-deployment evaluation process includes:

- Submitting a batch of images to the deployed /predict endpoint,
- Collecting predicted labels and class probabilities,
- Comparing predictions against known labels,
- Computing simple performance metrics such as accuracy.


argo-rdp.codeargo.net/web-rdp/#/client/aS0wNGQ0YzA5NjhjMzExZDdjZABjAGpzb24?token=ABD67476C514511B61B
Sign in
Applications
Google Keep — Mozilla ...
Terminal - cloud@2024a...
2026-02-10 19:43
Cloud User
Terminal - cloud@2024aa05119:~/mlops_a2_group90
File Edit View Terminal Tabs Help
cloud@2024aa05119:~/mlops_a2_group90
cloud@2024aa05119:~/mlops_a2_group90
cloud@2024aa05119:~/mlops_a2_group90

```
  File "<frozen importlib._bootstrap>", line 241, in _call_with_frames_removed
  File "/app/src/inference/app.py", line 103, in <module>
    model = load_model(MODEL_PATH)
TypeError: load_model() takes 0 positional arguments but 1 was given
(mlops_a2_group90) cloud@2024aa05119:~/mlops_a2_group90$
(mlops_a2_group90) cloud@2024aa05119:~/mlops_a2_group90$ docker logs -f catsdogs-inference
2026-02-10 14:11:27,318 - INFO - Loading model from /app/models/model_local.pt
/usr/local/lib/python3.10/site-packages/torchvision/models/_utils.py:208: UserWarning: The parameter 'pretrained' is deprecated since 0.13 and may be removed in the fut
ure, please use 'weights' instead.
  warnings.warn(
/usr/local/lib/python3.10/site-packages/torchvision/models/_utils.py:223: UserWarning: Arguments other than a weight enum or `None` for 'weights' are deprecated since 0
.13 and may be removed in the future. The current behavior is equivalent to passing `weights=None`.
  warnings.warn(msg)
INFO:     Started server process [1]
INFO:     Waiting for application startup.
INFO:     Application startup complete.
INFO:     Uvicorn running on http://0.0.0.0:8000 (Press CTRL+C to quit)
2026-02-10 14:11:47,474 - INFO - Health check requested.
INFO:     172.18.0.1:32868 - "GET /health HTTP/1.1" 200 OK
Request #1 | label=cat | latency=89.25 ms
INFO:     172.18.0.1:59334 - "POST /predict HTTP/1.1" 200 OK
Request #2 | label=cat | latency=104.47 ms
INFO:     172.18.0.1:59338 - "POST /predict HTTP/1.1" 200 OK
```


argo-rdp.codeargo.net/web-rdp/#/client/aS0wNGQ0YzA5NjhjMzExZDdjZABjAGpzb24?token=ABD67476C514511B61B
Sign in
Applications
Google Keep — Mozilla ...
Terminal - cloud@2024a...
2026-02-10 20:02
Cloud User
Terminal - cloud@2024aa05119:~/mlops_a2_group90
File Edit View Terminal Tabs Help
cloud@2024aa05119:~/mlops_a2_group90
cloud@2024aa05119:~/mlops_a2_group90
cloud@2024aa05119:~/mlops_a2_group90

```
(mlops_a2_group90) cloud@2024aa05119:~/mlops_a2_group90$ python src/utils/post_deploy_eval.py
Starting post-deployment evaluation...

Image: 9347.jpg              | True: cat | Pred: cat | Correct: True
Image: 1386.jpg              | True: cat | Pred: cat | Correct: True
Image: 5441.jpg              | True: cat | Pred: cat | Correct: True
Image: 8611.jpg              | True: cat | Pred: cat | Correct: True
Image: 2792.jpg              | True: cat | Pred: cat | Correct: True
Image: 10694.jpg             | True: dog | Pred: cat | Correct: False
Image: 5797.jpg              | True: dog | Pred: cat | Correct: False
Image: 11575.jpg             | True: dog | Pred: cat | Correct: False
Image: 1289.jpg              | True: dog | Pred: cat | Correct: False
Image: 9530.jpg              | True: dog | Pred: cat | Correct: False

=====================================
Post-deployment accuracy: 0.50
Total samples evaluated: 10
=====================================
(mlops_a2_group90) cloud@2024aa05119:~/mlops_a2_group90$
```

This approach demonstrates how deployed models can be validated in a live or staging environment and highlights the importance of monitoring model performance beyond the training phase.

8.4 Limitations and Future Monitoring Extensions

The monitoring and evaluation mechanisms implemented in this project are intentionally lightweight, focusing on clarity and simplicity. In a production-scale system, this setup could be extended to include:

- Real-time metrics collection (e.g., request latency, throughput),
- Time-series monitoring using Prometheus,
- Visualization dashboards using Grafana,
- Automated model drift detection and alerting,
- Feedback loops for continuous retraining.

These extensions represent natural next steps but are outside the scope of the current assignment.

8.5 Summary of M5 Outcomes

By the end of this milestone:

- Logging is enabled for all key inference service operations,
- Health checks support availability monitoring,
- Basic post-deployment model evaluation is performed,
- The deployed system provides sufficient observability for validation and debugging.

This completes **Milestone M5: Monitoring, Logging, and Post-Deployment Evaluation**, ensuring that the deployed machine learning service is observable, verifiable, and ready for future enhancement.

9. Conclusion

This project successfully demonstrates the design and implementation of an **end-to-end MLOps pipeline** for a binary image classification use case. Using the Cats vs Dogs classification problem as a representative example, the project applies MLOps principles to transform a machine learning experiment into a **reproducible, automated, and deployable system**.

Across **Milestones M1 to M5**, the complete machine learning lifecycle is addressed. Model development and experimentation are conducted in a reproducible manner using configuration-driven training, dataset versioning with DVC, and systematic experiment tracking with MLflow. A baseline model and a final optimized model are implemented and evaluated, providing a structured approach to model selection.

The trained model is then packaged into a RESTful inference service using FastAPI and containerized with Docker to ensure environment consistency. Continuous Integration pipelines validate code changes through automated testing and container builds, while Continuous Deployment pipelines automate service updates using Docker Compose. Health checks and smoke tests ensure reliable deployments, and the deployed service is designed to support CI/CD-safe operation.

Finally, the project incorporates logging, basic monitoring, and post-deployment evaluation to provide operational visibility and validate model behavior after deployment. While the monitoring setup is intentionally lightweight, it establishes a strong foundation that can be extended to production-grade observability and model performance tracking systems.

Overall, this project illustrates how MLOps practices enable reliable, scalable, and maintainable machine learning systems. By integrating open-source tools and following industry-aligned workflows, the pipeline demonstrates a practical and extensible approach to managing the full lifecycle of machine learning applications.

10. Submission Deliverables

The following artifacts are included as part of the final submission:

1. Source Code Package

The ZIP file contains:

- Complete source code
- DVC configuration files
- MLflow experiment configuration
- Dockerfile
- docker-compose.yml
- CI/CD pipeline configuration (GitHub Actions)

- Deployment scripts
- Trained model artifacts

2. Container Registry

Docker image is published to:

- GitHub Container Registry (GHCR)

3. Repository Link

Project repository:

- https://github.com/ranga-srinivasan/mlops_a2_group90

4. Screen Recording Demonstration

A video (under 5 minutes) demonstrating:

- Code modification
- CI pipeline execution
- Docker image build
- CD deployment
- Live prediction via API

Video link:

- https://drive.google.com/file/d/1HeFcT734VZ5lUkgfKX_ysCCS2JL974P_/view?usp=drive_link