MLOps Iris Classification – End to End Pipeline

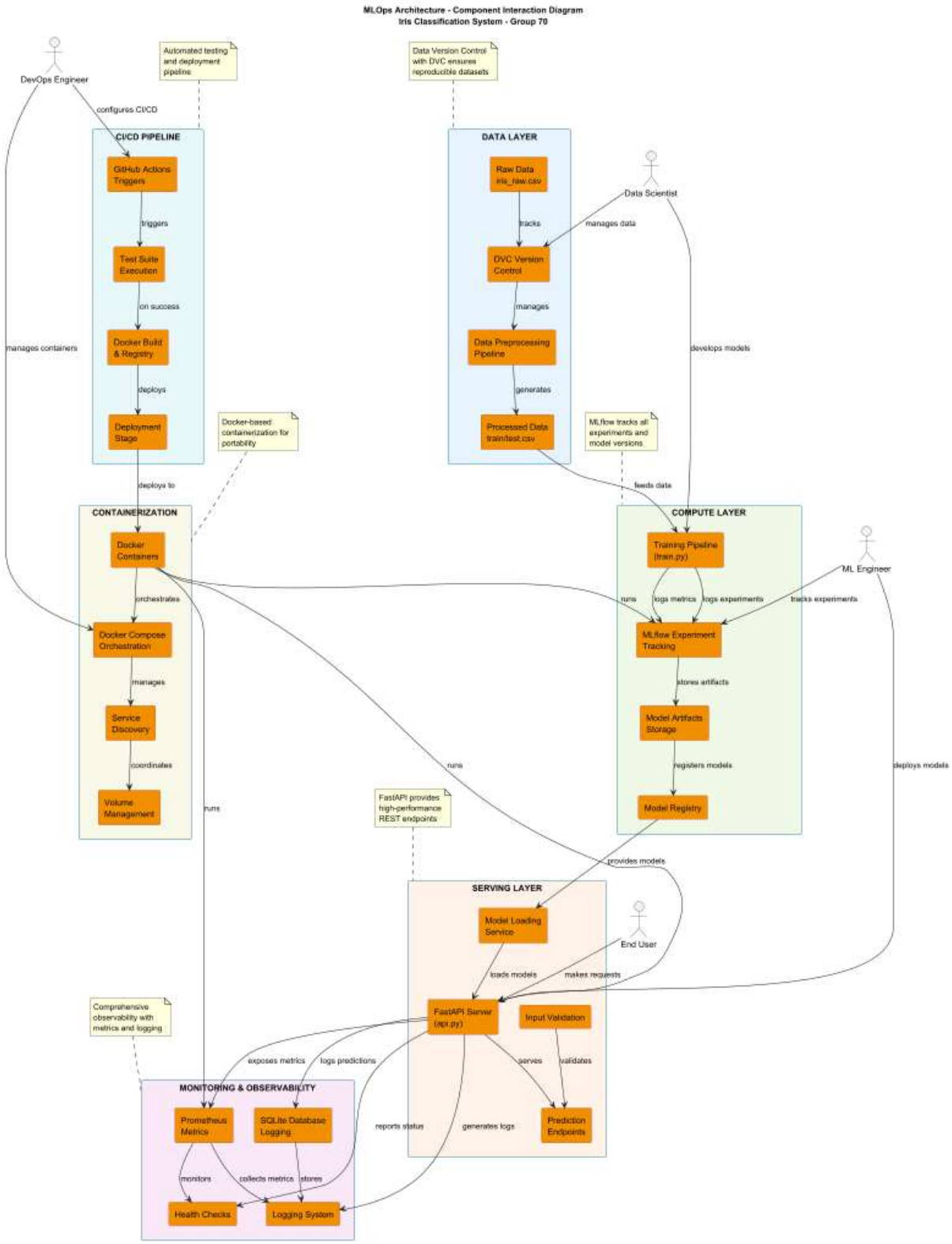
Group No: 70

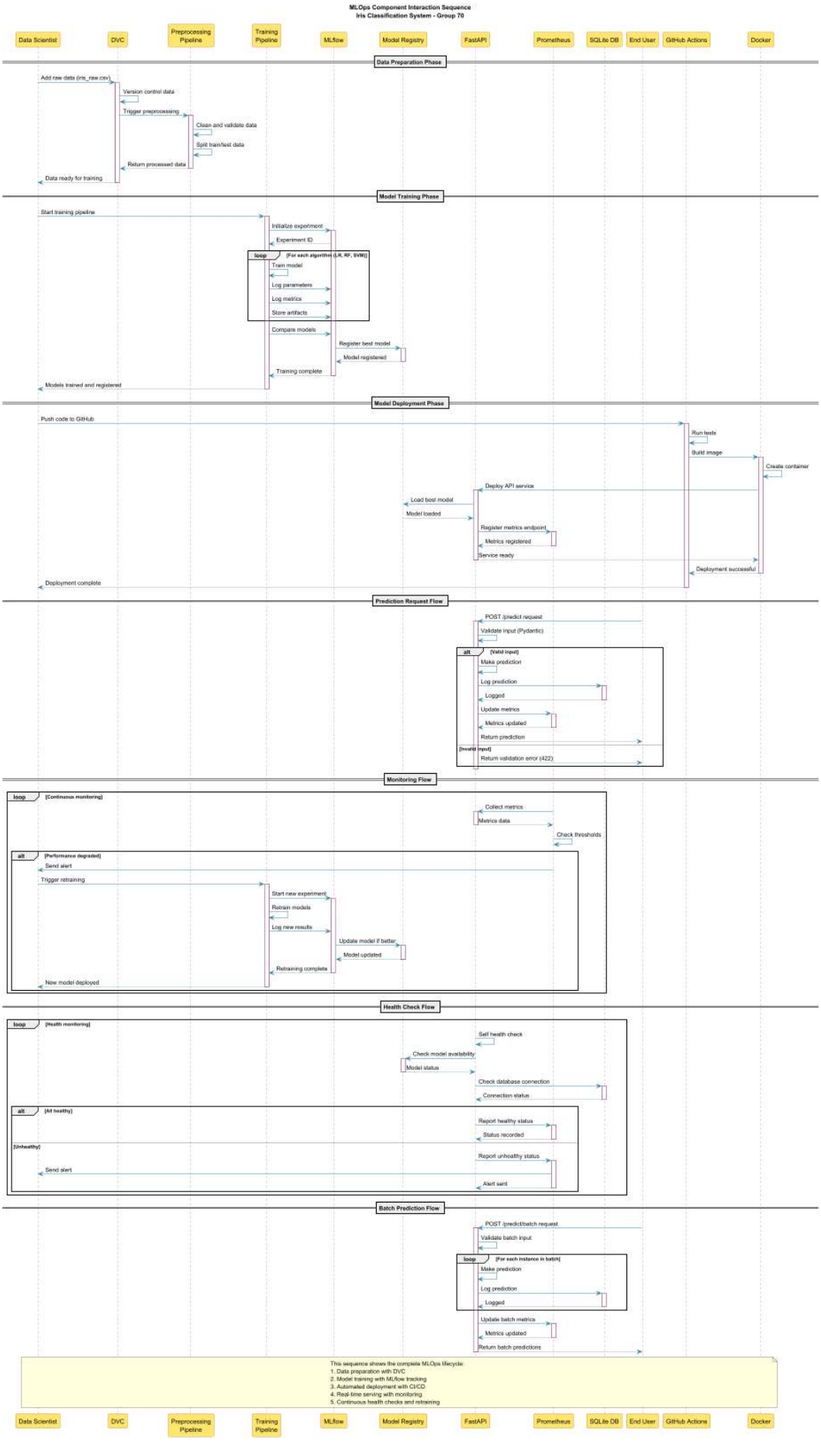
Name	ID	Contribution(%)
Dhiman Kundu	2023ac05129	100
Rina Gupta	2023ac05028	100
Sarit Ghosh	2023ac05131	100
Soumen Choudhury	2023ac05143	100

Assignment Tasks

This project implements a comprehensive MLOps implementation for Iris flower classification with automated training, monitoring, and deployment capabilities addressing the following assignment requirements:

Architecture Diagram:




MLOps Component Interaction Sequence
Iris Classification System - Group 70
Data Scientist
DVC
Preprocessing Pipeline
Training Pipeline
MLflow
Model Registry
FastAPI
Prometheus
SQLite DB
End User
GitHub Actions
Docker
Data Preparation Phase
Add raw data (iris_raw.csv)
Version control data
Trigger preprocessing
Clean and validate data
Split train/test data
Return processed data
Data ready for training
Model Training Phase
Start training pipeline
Initialize experiment
Experiment ID
loop
[For each algorithm (LR, RF, SVM)]
Train model
Log parameters
Log metrics
Store artifacts
Compare models
Register best model
Model registered
Training complete
Models trained and registered
Model Deployment Phase
Push code to GitHub
Run tests
Build image
Create container
Deploy API service
Load best model
Model loaded
Register metrics endpoint
Metrics registered
Service ready
Deployment successful
Deployment complete
Prediction Request Flow
POST /predict request
Validate input (Pydantic)
alt
[Valid input]
Make prediction
Log prediction
Logged
Update metrics
Metrics updated
Return prediction
[Invalid input]
Return validation error (422)
Monitoring Flow
loop
[Continuous monitoring]
Collect metrics
Metrics data
Check thresholds
alt
[Performance degraded]
Send alert
Trigger retraining
Start new experiment
Retrain models
Log new results
Update model if better
Model updated
Retraining complete
New model deployed
Health Check Flow
loop
[Health monitoring]
Self health check
Check model availability
Model status
Check database connection
Connection status
alt
[All healthy]
Report healthy status
Status recorded
[Unhealthy]
Report unhealthy status
Send alert
Alert sent
Batch Prediction Flow
POST /predict/batch request
Validate batch input
loop
[For each instance in batch]
Make prediction
Log prediction
Logged
Update batch metrics
Metrics updated
Return batch predictions
This sequence shows the complete MLOps lifecycle:
1. Data preparation with DVC
2. Model training with MLflow tracking
3. Automated deployment with CI/CD
4. Real-time serving with monitoring
5. Continuous health checks and retraining

Part 1: Repository and Data Versioning

- **GitHub Repository**: Complete project hosted on GitHub with clean structure

https://github.com/SaritGhoshBits25/MLOps_Assignment_Group70

- **Docker Hub Repository**:

https://hub.docker.com/r/wp1412011989/iris-api

- **Demo Video Link:**

https://drive.google.com/drive/folders/15KfNF9Iwzfgb3ljQOudrMe5CuQ60zw4y?usp=drive_link

- **Data Loading & Preprocessing**: Automated data preprocessing pipeline (`src/data_preprocessing.py`)

Directory Structure: Well-organized project structure with separate directories for source code, data, models, tests, and monitoring

```
├── src/                    # Source code
│   ├── api.py              # FastAPI application with monitoring
│   ├── train.py            # Model training with MLflow
│   ├── data_preprocessing.py # Data preprocessing pipeline
│   ├── database.py         # Database operations and logging
│   └── retrain_pipeline.py # Automated retraining pipeline
├── data/                   # Dataset files
│   ├── iris_raw.csv        # Raw iris dataset
│   ├── iris_train.csv      # Training data
│   └── iris_test.csv       # Test data
├── models/                 # Trained model artifacts
├── tests/                  # Test suite
├── monitoring/             # Monitoring configuration
│   └── prometheus.yml      # Prometheus configuration
├── .github/workflows/      # CI/CD pipeline
│   └── ci-cd.yml           # GitHub Actions workflow
├── Dockerfile              # Container configuration
├── docker-compose.yml      # Multi-service orchestration
└── requirements.txt        # Python dependencies
```

Data Version Control (DVC):

This project uses DVC for data versioning and pipeline management. DVC tracks data files and ensures reproducible data processing workflows.

- **Data tracking**: Raw iris dataset (`data/iris_raw.csv`)
- **Pipeline**: Automated data preprocessing pipeline

- **Remote storage**: Local remote for data versioning

DVC Files Structure

```
├── .dvc/
│   └── config            # DVC configuration
├── data/
│   ├── .gitignore        # Git ignores data files
│   ├── iris_raw.csv.dvc  # DVC tracks raw data
│   ├── iris_train.csv    # Generated by pipeline
│   └── iris_test.csv     # Generated by pipeline
├── dvc.yaml              # Pip`eline definition
```

Part 2: Model Development & Experiment Tracking

Multiple Models: Implementation of classification models:

- Logistic Regression
- Random Forest
- Support Vector Machine (SVM)

Model Information:

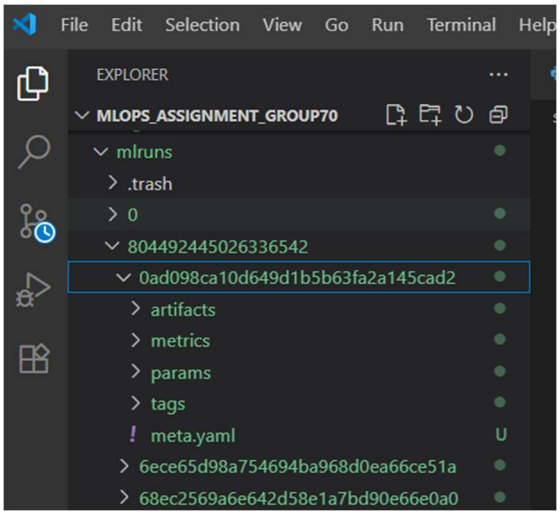
- **Dataset**: Iris flower classification
- **Features**: Sepal length/width, Petal length/width
- **Classes**: Setosa, Versicolor, Virginica
- **Models**: Logistic Regression, Random Forest, SVM
- **Evaluation**: Accuracy, Precision, Recall, F1-score

MLflow Integration:

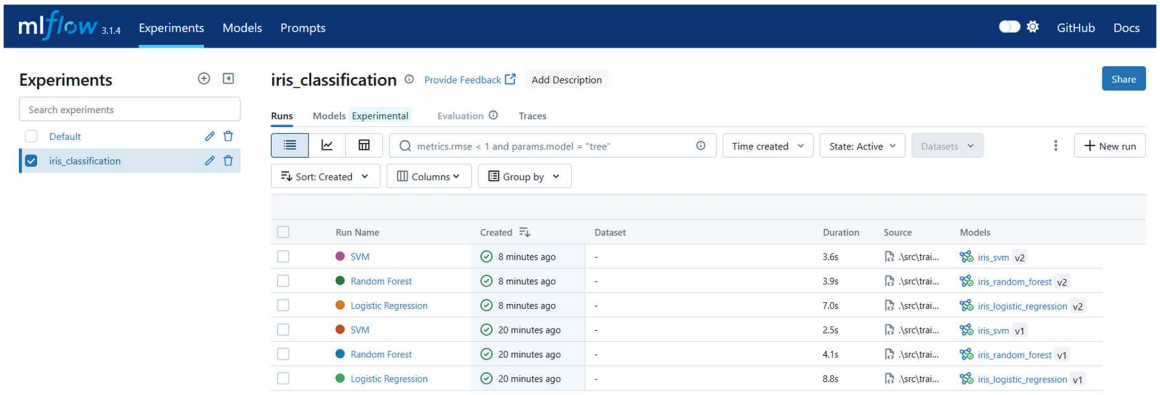
- Experiment tracking with parameters and metrics (`src/train.py`)
- Model versioning and artifact storage
- Model registry for best model selection

Following directories are created by MLflow under mlruns/<experiment-id>/<run-id>/:

- params/: model hyperparameters
- metrics/: performance metrics
- artifacts/: saved models
- tags/: metadata like model name or author



- MLflow UI accessible at `**http://localhost:5000**`



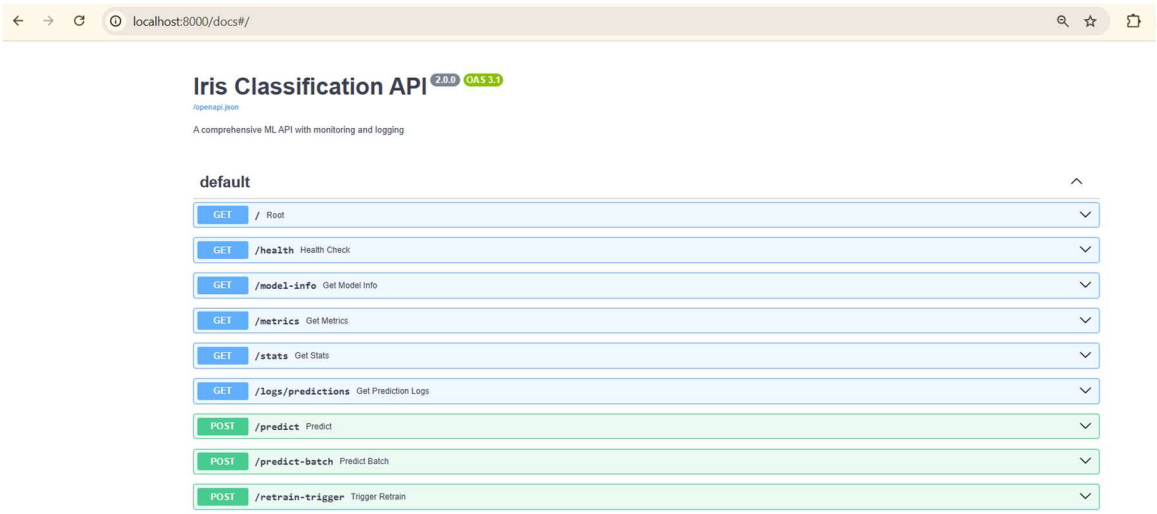
- **Model Selection**: Automated best model selection based on performance metrics

Part 3: API & Docker Packaging

FastAPI Implementation: High-performance REST API (`src/api.py`) with:

- Automatic API documentation with Swagger UI
- GET / - API information and status
- GET /health - Health check endpoint
- POST /predict - Make predictions single predictions
- POST /predict/batch - Batch predictions for batch predictions
- GET /model/info - Current model information

- GET /metrics - Prometheus metrics
- POST /retrain-trigger – Retrain Model

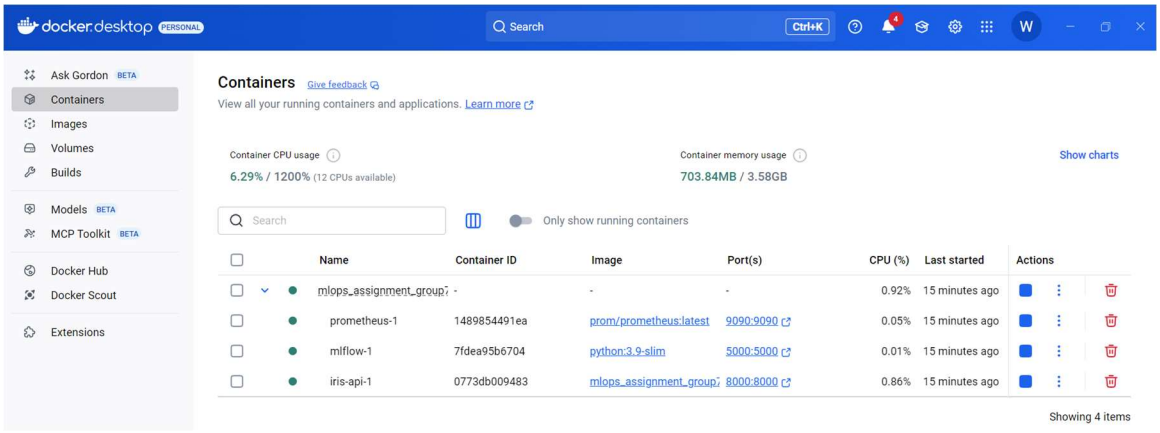


Pydantic schemas are used for request validation.

Docker Containerization: This is fully containerized using Docker.

- Multi-stage Dockerfile for optimized images
- Docker Compose orchestration (`docker-compose.yml`)
- Health checks and restart policies
- Build and run with Docker Compose

```
docker-compose up –build
```



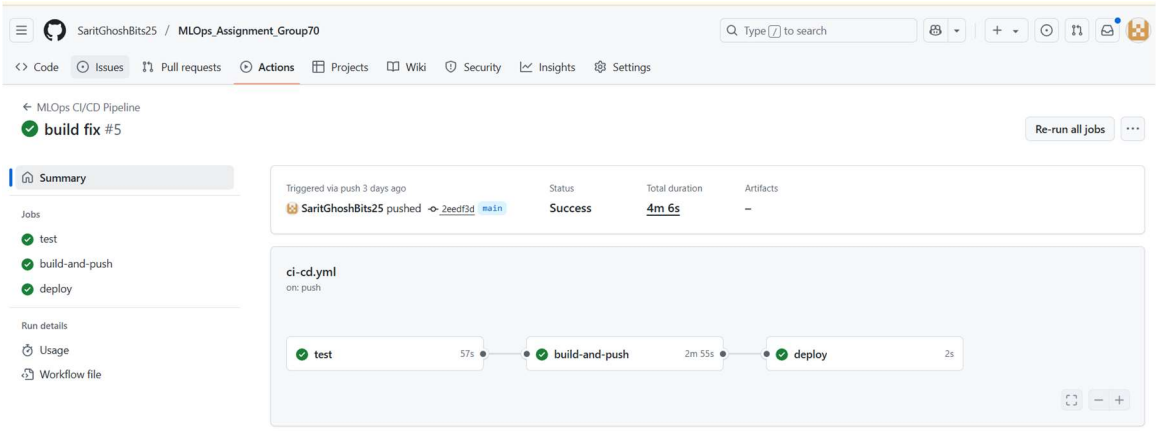
This will start:

- **Iris API:** http://localhost:8000
- **Prometheus:** http://localhost:9090
- **MLflow:** http://localhost:5000

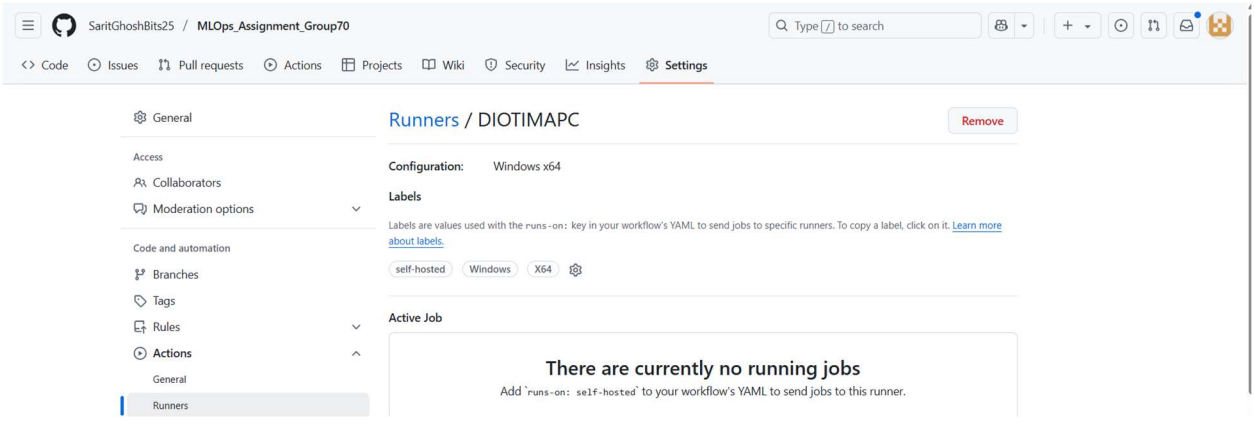
- **JSON Input/Output**: Structured JSON request/response format with validation

Part 4: CI/CD with GitHub Actions

- **Automated Pipeline** (`.github/workflows/ci-cd.yml`):
- **Testing Stage**: Code linting, unit tests, and API health verification
- **Build Stage**: Docker image building and container testing
- **Deploy Stage**: Automated deployment with health monitoring
- **Code Quality**: Automated linting and testing on every push
- **Docker Hub Integration**: Automated image building and registry push



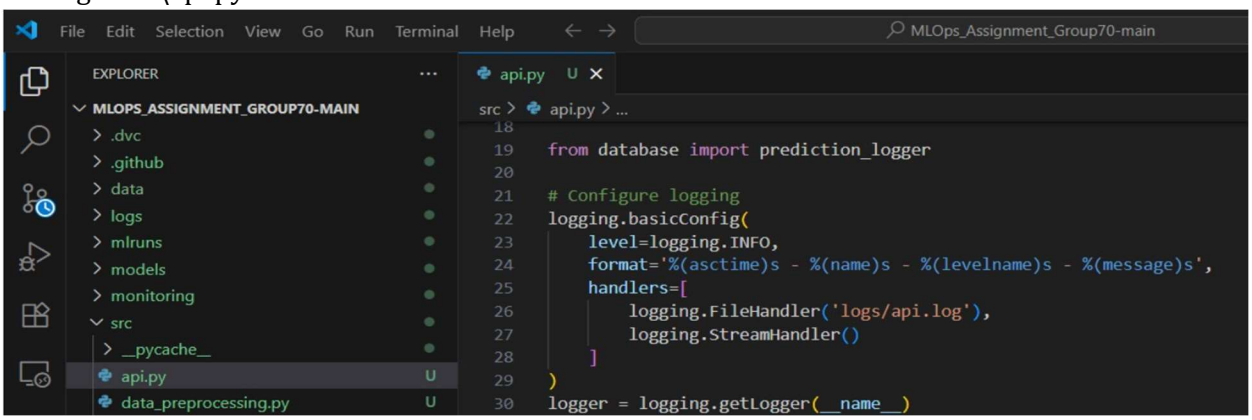
- **Deployment**: Local deployment with self-hosted runner



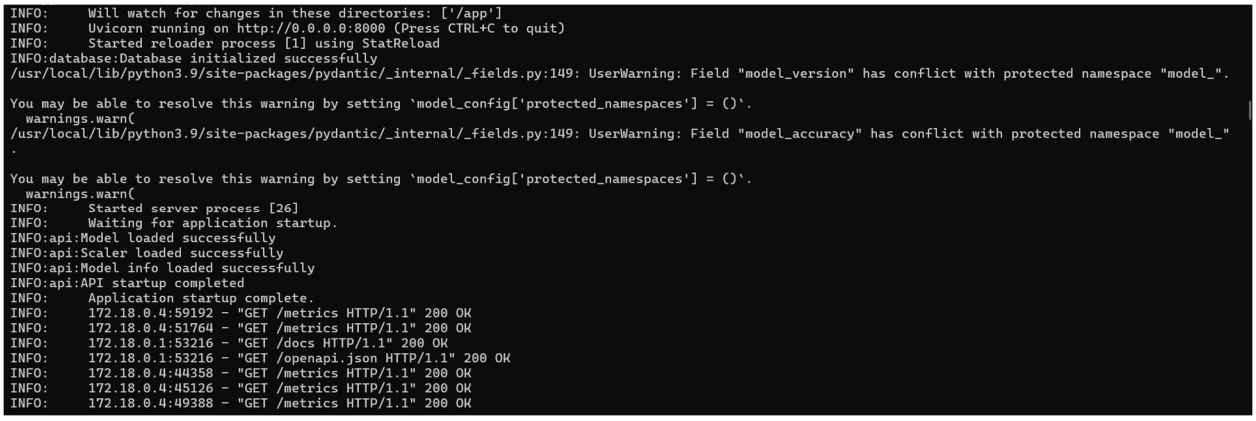
Part 5: Logging and Monitoring

Comprehensive Logging:

- API logs: src\api.py



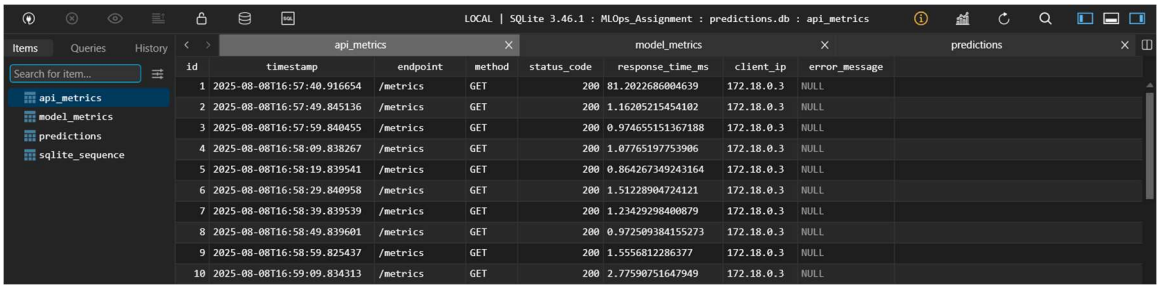
API Logging:



- **Container logs:** docker-compose logs <service-name>
- **MLflow logs:** Available in MLflow UI

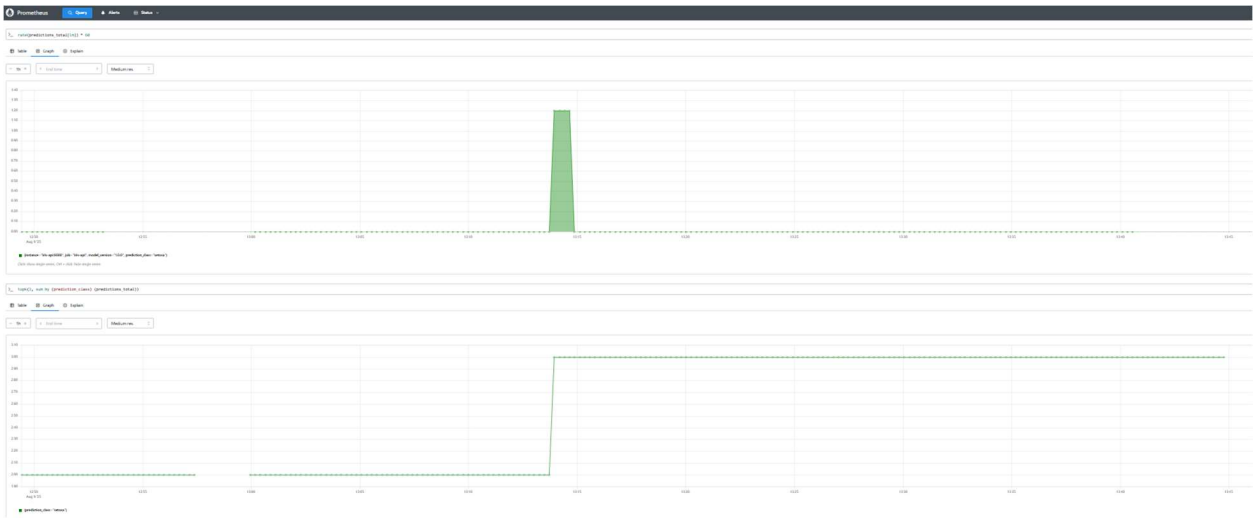
All predictions are logged to SQLite database for persistence storage with (`src/database.py`):

- Prediction ID and timestamp
- Input features and predictions
- Model version and confidence scores
- Request metadata



Monitoring Integration:

- Prometheus metrics endpoint (`/metrics`)
- Custom metrics for predictions, latency, and model performance
- Prometheus configuration (`monitoring/prometheus.yml`)



Health Monitoring: Dedicated health check endpoints for system status

Part 6: Summary + Demo

- **Architecture Documentation**: Comprehensive summary with system architecture
- **Demo Preparation**: Complete setup instructions and API usage examples
- **Video Walkthrough**: Ready-to-demonstrate solution with all components integrated

Bonus Features

Input Validation:

- Pydantic models for request/response validation
- Schema-based input validation with error handling

Prometheus Integration:

- Full Prometheus monitoring setup
- Custom metrics dashboard ready
- Real-time performance monitoring

Model Retraining via API:

The '/retrain' endpoint supports:

- Automated retraining pipeline (`src/retrain_pipeline.py`)
- Performance-based retraining triggers
- Continuous model improvement workflow
- Retraining the best model
- Logging it to MLflow
- Saving the new model to registry