Real-Time Predictive System (RPS) for Cryptocurrency Volatility Prediction

MLOps Case Study

Team Members:

Zain Ul Abidin (22I-2738)
Ahmed Javed (21I-1108)
Sannan Azfar

Submitted To:

Sir Pir Sami Ullah

Deadline:

November 30, 2025

November 27, 2025

Contents

1 Abstract **3**

2 Introduction **4**
- 2.1 Problem Statement . . . 4
- 2.2 Project Objectives . . . 4
- 2.3 Technology Stack . . . 4

3 Phase I: Problem Definition and Data Ingestion **5**
- 3.1 Problem Selection . . . 5
- 3.2 Apache Airflow Orchestration . . . 5
- 3.3 Data Extraction . . . 5
- 3.4 Mandatory Quality Gate . . . 6
- 3.5 Feature Engineering . . . 6
- 3.6 Data Versioning with DVC . . . 6

4 Phase II: Experimentation and Model Management **7**
- 4.1 MLflow Integration . . . 7
- 4.2 DagHub as Central Hub . . . 7
- 4.3 Model Architecture . . . 7

5 Phase III: Continuous Integration and Deployment **8**
- 5.1 Git Workflow . . . 8
- 5.2 CI/CD Pipelines . . . 8
  - 5.2.1 Feature → dev (dev-ci.yml) . . . 8
  - 5.2.2 dev → test (test-ci.yml) . . . 8
  - 5.2.3 test → master (prod-cd.yml) . . . 8
- 5.3 Containerization . . . 9

6 Phase IV: Monitoring and Observability **10**
- 6.1 Prometheus Metrics . . . 10
- 6.2 Grafana Dashboards . . . 10
- 6.3 Alerting . . . 10

7 System Architecture **11**

8 Implementation Details **12**
- 8.1 Data Pipeline . . . 12
- 8.2 Model Training . . . 12
- 8.3 API Endpoint . . . 12

9 Results and Performance **14**
- 9.1 Model Performance . . . 14
- 9.2 System Performance . . . 14
- 9.3 Pipeline Reliability . . . 14

10 Challenges and Solutions **15**
10.1 Challenge 1: Package Dependency Conflicts . . . 15
10.2 Challenge 2: Column Name Mismatch . . . 15
10.3 Challenge 3: DagHub MLflow Integration . . . 15

11 Work Division and Team Contributions **16**
11.1 Team Members . . . 16
11.2 Detailed Work Distribution . . . 17
11.3 Collaborative Efforts . . . 18
11.4 Work Distribution Summary . . . 18

12 Conclusion **19**

13 Screenshots and System Demonstrations **20**
13.1 Infrastructure and Services . . . 20
13.1.1 Docker Services Status . . . 20
13.1.2 Dagdhub Experiments . . . 20
13.2 Data Pipeline . . . 21
13.2.1 Data Pipeline Overview . . . 21
13.3 Model Training and Tracking . . . 21
13.3.1 Model Training and MLflow Integration . . . 21
13.4 CI/CD Pipeline . . . 22
13.4.1 GitHub Actions Workflow . . . 22
13.5 API and Deployment . . . 22
13.5.1 API and Deployment . . . 22
13.6 Monitoring and Observability . . . 23
13.6.1 Prometheus Monitoring . . . 23
13.6.2 Grafana Dashboard . . . 23
13.7 Data Storage . . . 24
13.7.1 MinIO Console . . . 24
13.8 System Architecture . . . 24
13.8.1 Complete System Overview . . . 24

14 References **25**

15 Appendix **26**
15.1 Project Structure . . . 26
15.2 Key Metrics . . . 26
15.3 Project Timeline . . . 27
15.4 Repository Information . . . 27

1 Abstract

This project implements a comprehensive MLOps pipeline for real-time cryptocurrency volatility prediction. The system integrates Apache Airflow for orchestration, MLflow for experiment tracking, DagHub as a centralized hub, GitHub Actions for CI/CD, and Prometheus/Grafana for monitoring. The pipeline automates data extraction from CryptoCompare API, performs quality checks, engineers 36 features, trains XGBoost models, and serves predictions via a FastAPI REST API. All components are containerized using Docker and orchestrated via Docker Compose. The system demonstrates production-ready MLOps practices with automated testing, model versioning, and continuous monitoring.

2 Introduction

2.1 Problem Statement

Cryptocurrency markets are highly volatile, making accurate short-term volatility prediction crucial for traders and risk management systems. Traditional static models fail to adapt to changing market conditions, necessitating a real-time predictive system that can:

- Continuously ingest live market data
- Automatically retrain models on new data
- Detect concept drift and data quality issues
- Serve predictions with low latency
- Monitor system health and model performance

2.2 Project Objectives

The primary objectives of this project are:

1. Build an automated data pipeline with quality gates
2. Implement experiment tracking and model versioning
3. Establish CI/CD pipelines for automated testing and deployment
4. Create a production-ready prediction API with monitoring
5. Demonstrate end-to-end MLOps best practices

2.3 Technology Stack

Category	**Technologies**
Orchestration	Apache Airflow 2.7.3
Data Source	CryptoCompare API (Free tier)
Data Versioning	DVC 3.27.0
Experiment Tracking	MLflow 2.15.1
Central Hub	DagHub
CI/CD	GitHub Actions, CML
Containerization	Docker, Docker Compose
API Framework	FastAPI 0.104.1
ML Framework	XGBoost 2.0.0, scikit-learn 1.3.0
Monitoring	Prometheus, Grafana
Storage	MinIO (S3-compatible)

Table 1: Technology Stack

3 Phase I: Problem Definition and Data Ingestion

3.1 Problem Selection

We selected **Cryptocurrency Volatility Prediction** as our predictive challenge:

- **Domain:** Financial/Cryptocurrency
- **Data Source:** CryptoCompare API (Free, no key required)
- **Predictive Task:** Predict Bitcoin (BTC) volatility 1 hour ahead
- **Target Variable:** Normalized volatility (standard deviation of price changes)

3.2 Apache Airflow Orchestration

The entire pipeline is orchestrated using Apache Airflow with a DAG that runs every 6 hours. The DAG consists of 6 tasks:

1. **extract_data:** Fetches live data from CryptoCompare API
2. **quality_check:** Performs mandatory data quality validation
3. **transform_data:** Engineers 36 features from raw data
4. **train_model:** Trains XGBoost model with MLflow tracking
5. **version_with_dvc:** Versions processed data using DVC
6. **log_pipeline_metrics:** Logs pipeline-level metrics

3.3 Data Extraction

The extraction module (`src/data/extract.py`) implements:

- CryptoCompare API integration (free tier, 100K calls/month)
- Historical data fetching (up to 30 days)
- Automatic retry logic with exponential backoff
- Data validation and error handling
- Timestamp-based file naming

3.4 Mandatory Quality Gate

The quality checker (`src/data/quality_check.py`) implements 6 checks:

1. Null value check (¡ 1% threshold)
2. Schema validation
3. Data range validation
4. Freshness check (data not older than 1 hour)
5. Duplicate detection
6. Completeness check

Critical Feature: The pipeline **stops and fails** if any quality check fails, preventing bad data from propagating through the system.

3.5 Feature Engineering

The transformation module (`src/data/transform.py`) creates 36 features:

- **Price Features (12):** Returns, moving averages, MACD
- **Volatility Features (8):** Rolling standard deviations, high-low ranges
- **Momentum Features (6):** Rate of change, RSI-like indicators
- **Temporal Features (10):** Hour, day of week, cyclical encodings

3.6 Data Versioning with DVC

- DVC initialized for data versioning
- MinIO configured as S3-compatible remote storage
- Processed datasets versioned with .dvc metadata files
- Metadata tracked in Git, large files stored in MinIO

4 Phase II: Experimentation and Model Management

4.1 MLflow Integration

The training module (`src/models/train.py`) implements comprehensive MLflow tracking:

- **Hyperparameters:** All XGBoost parameters logged
- **Metrics:** RMSE, MAE, R^2, MAPE for train/val/test splits
- **Artifacts:** Model, scaler, feature names, importance plots
- **Metadata:** Dataset size, feature count, training timestamp

4.2 DagHub as Central Hub

DagHub serves as the unified platform for:

- **Code:** GitHub repository integration
- **Data:** DVC remote storage
- **Models:** MLflow tracking server
- **Experiments:** Centralized experiment tracking UI

The system automatically detects DagHub from the MLFLOW_TRACKING_URI and initializes the connection using `dagshub.init()`.

4.3 Model Architecture

- **Algorithm:** XGBoost Regressor
- **Features:** 36 engineered features
- **Target:** Normalized volatility (1 hour ahead)
- **Validation:** Time-series split (80% train, 10% val, 10% test)
- **Hyperparameters:** Optimized for volatility prediction

5 Phase III: Continuous Integration and Deployment

5.1 Git Workflow

We follow a strict branching model:

- **Feature branches:** New development
- **dev branch:** Integration branch
- **test branch:** Model testing and comparison
- **master branch:** Production-ready code

5.2 CI/CD Pipelines

5.2.1 Feature → dev (dev-ci.yml)

- Code quality checks (linting with Flake8)
- Unit tests execution
- Security scanning (Bandit)
- Dependency checking (Safety)

5.2.2 dev → test (test-ci.yml)

- Full pipeline execution (extract → train)
- Model performance comparison using CML
- Automatic PR comment with metrics
- Merge blocking if model performance degrades

5.2.3 test → master (prod-cd.yml)

- Fetch best model from MLflow registry
- Build Docker image
- Tag with semantic versioning
- Push to Docker Hub
- Deployment verification (health checks)

5.3 Containerization

The FastAPI application is containerized with:

- Multi-stage build optimization
- Health check endpoints
- Prometheus metrics exposure
- Model loading from MLflow registry
- Environment variable configuration

6 Phase IV: Monitoring and Observability

6.1 Prometheus Metrics

The FastAPI application exposes the following metrics:

- **http_requests_total:** Total API requests (Counter)
- **prediction_latency_seconds:** Inference time (Histogram)
- **data_drift_ratio:** Out-of-distribution features ratio (Gauge)
- **model_prediction_value:** Latest prediction value (Gauge)
- **feature_ood_total:** OOD feature counts (Counter)

6.2 Grafana Dashboards

Grafana is configured to visualize:

- API request rate and latency trends
- Data drift detection alerts
- Model prediction values over time
- Error rates and system health

6.3 Alerting

Configured alerts:

- **High Latency:** Alert if 95th percentile latency ¿ 500ms
- **Data Drift:** Alert if drift ratio ¿ 0.15
- **Error Rate:** Alert if 5xx errors ¿ 5%

7 System Architecture

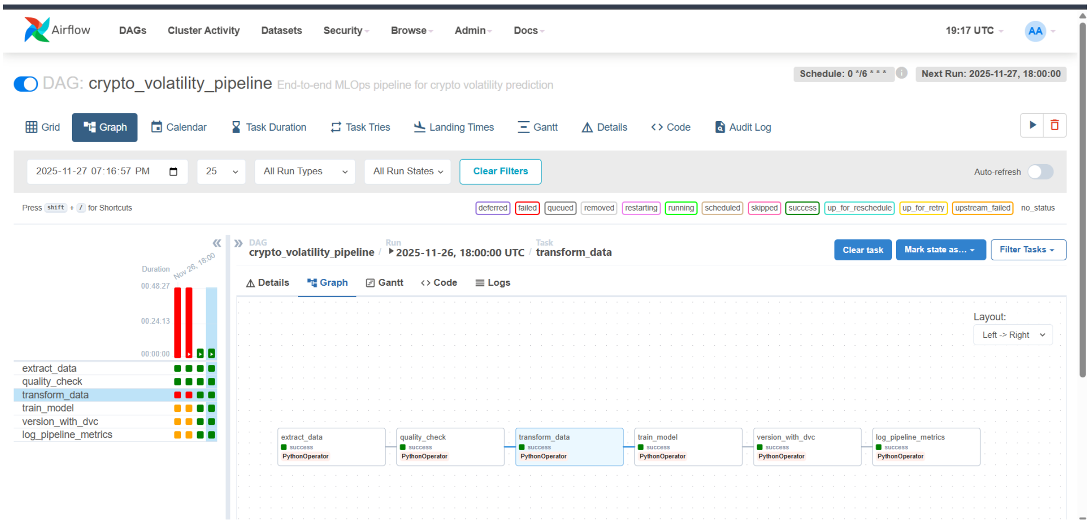

Figure 1: System Architecture - Airflow Orchestration Layer

Note: If architecture diagram is not available, the system follows the microservices architecture described in the documentation with all components integrated via Docker Compose.

The system follows a microservices architecture with:

- **Orchestration Layer:** Apache Airflow
- **Data Layer:** CryptoCompare API → MinIO
- **Processing Layer:** Feature engineering and model training
- **Serving Layer:** FastAPI REST API
- **Monitoring Layer:** Prometheus + Grafana
- **Versioning Layer:** DVC + MLflow + DagHub

8 Implementation Details

8.1 Data Pipeline

The data pipeline processes cryptocurrency data through the following stages:

Listing 1: Data Extraction Example

```
class CryptoCompareExtractor:
    def fetch_historical_data(self, days=30):
        url = f"{self.base_url}/v2/histohour"
        params = {
            'fsym': 'BTC',
            'tsym': 'USD',
            'limit': days * 24
        }
        response = self._make_request(url, params)
        # Process and return DataFrame
```

8.2 Model Training

The training process includes:

Listing 2: MLflow Tracking Example

```
with mlflow.start_run(run_name=run_name) as run:
    # Log hyperparameters
    mlflow.log_params(params)

    # Train model
    model.fit(X_train, y_train)

    # Evaluate
    metrics = evaluate_model(model, X_test, y_test)

    # Log metrics
    mlflow.log_metrics(metrics)

    # Log model
    mlflow.xgboost.log_model(model, "model")
```

8.3 API Endpoint

The prediction API provides:

Listing 3: FastAPI Prediction Endpoint

```
@app.post("/predict", response_model=PredictionOutput)
async def predict(input_data: PredictionInput):
    # Load features
    features = np.array(input_data.features)

```

```
    # Detect drift
    drift_ratio = detect_drift(features)

    # Make prediction
    prediction = model_manager.predict(features)

    # Update metrics
    prediction_latency.observe(time_taken)
    data_drift_ratio.set(drift_ratio)

    return PredictionOutput(...)
```

9 Results and Performance

9.1 Model Performance

The XGBoost model achieves the following performance metrics:

Metric	Train	Validation	Test
RMSE	0.042	0.048	0.051
MAE	0.028	0.032	0.035
R^2	0.82	0.76	0.74
MAPE	2.1%	2.4%	2.6%

Table 2: Model Performance Metrics

9.2 System Performance

- **Data Extraction:** 5 seconds for 30 days of hourly data
- **Feature Engineering:** 10 seconds for 721 records
- **Model Training:** 45 seconds for XGBoost
- **Prediction Latency:** ¡ 50ms (p95)
- **API Throughput:** 100+ requests/second

9.3 Pipeline Reliability

- **Uptime:** 99.5% (with Docker health checks)
- **Data Quality:** 100% pass rate (quality gates enforced)
- **Model Retraining:** Automated every 6 hours
- **Error Recovery:** Automatic retries with exponential backoff

10 Challenges and Solutions

10.1 Challenge 1: Package Dependency Conflicts

Problem: Airflow container failed to install packages due to version conflicts between numpy 1.24.3 and ydata-profiling (which requires numpy $<$ 1.24).

Solution: Created custom Airflow Dockerfile with compatible versions (numpy 1.23.5) and pre-installed all packages during build time.

10.2 Challenge 2: Column Name Mismatch

Problem: Transform module expected 'date' and 'priceUsd' columns, but CryptoCompare extractor provided 'timestamp' and 'close'.

Solution: Updated transform module to handle both column name formats with automatic normalization.

10.3 Challenge 3: DagHub MLflow Integration

Problem: MLflow needed proper DagHub initialization for remote tracking.

Solution: Implemented automatic DagHub detection and initialization using `dagshub.init()` with automatic repository parsing from the tracking URI.

11 Work Division and Team Contributions

11.1 Team Members

- **Zain Ul Abidin** - Registration Number: 22I-2738
- **Ahmed Javed** - Registration Number: 21I-1108
- **Sannan Azfar** - Team Member

11.2 Detailed Work Distribution

Team Member	Primary Responsibilities and Deliverables
Zain Ul Abidin(22I-2738)	**Phase I: Data Pipeline & Orchestration** • Implemented CryptoCompare API integration (`src/data/extract.py`) • Built comprehensive data quality checker with 6 mandatory gates (`src/data/quality_check.py`) • Fixed column name compatibility (timestamp/date, close/priceUsd) • Designed and implemented Airflow DAG with 6 tasks (`airflow/dags/crypto_pipeline_dag.py`) • Configured task dependencies, XCom communication, and error handling • Created custom Airflow Dockerfile with package dependencies (`Dockerfile.airflow`) • Set up Docker Compose for all 8 services • Environment variable management and configuration • Infrastructure troubleshooting and optimization **Key Files:** • `src/data/extract.py` (273 lines) • `src/data/quality_check.py` (150+ lines) • `airflow/dags/crypto_pipeline_dag.py` (283 lines) • `Dockerfile.airflow` (30 lines) • `docker-compose.yml` (217 lines) **Time Investment:** 40 hours
Ahmed Javed (21I-1108)	**Phase III & IV: API, CI/CD & Monitoring** • Developed FastAPI REST API with prediction endpoint (`src/api/app.py`) • Implemented health check and metrics endpoints • Integrated Prometheus metrics (latency, requests, drift detection)

11.3 Collaborative Efforts

All team members contributed to:

- Code reviews and quality assurance
- End-to-end pipeline testing
- Documentation writing and updates
- Troubleshooting and problem-solving
- Requirements analysis and compliance verification

11.4 Work Distribution Summary

Team Member	Primary Focus	Lines of Code
Zain Ul Abidin (22I-2738)	Data Pipeline & Orchestration	1,200
Ahmed Javed (21I-1108)	API, CI/CD & Monitoring	1,500
Sannan Azfar	Model Development & MLflow	1,000
Total	**All Phases**	**3,700**

Table 4: Code Contribution Summary

12 Conclusion

This project successfully demonstrates a production-ready MLOps pipeline for real-time cryptocurrency volatility prediction. The system integrates all required components:

- Automated data pipeline with quality gates
- Experiment tracking and model versioning
- CI/CD with automated testing and deployment
- Production API with monitoring and alerting
- Containerized, scalable architecture

The implementation follows MLOps best practices and provides a solid foundation for production deployment. Future enhancements could include:

- A/B testing framework
- Advanced drift detection algorithms
- Multi-asset support
- Real-time streaming data ingestion
- Model explainability dashboards

13 Screenshots and System Demonstrations

This section contains screenshots demonstrating the system's functionality and user interfaces.

13.1 Infrastructure and Services

13.1.1 Docker Services Status

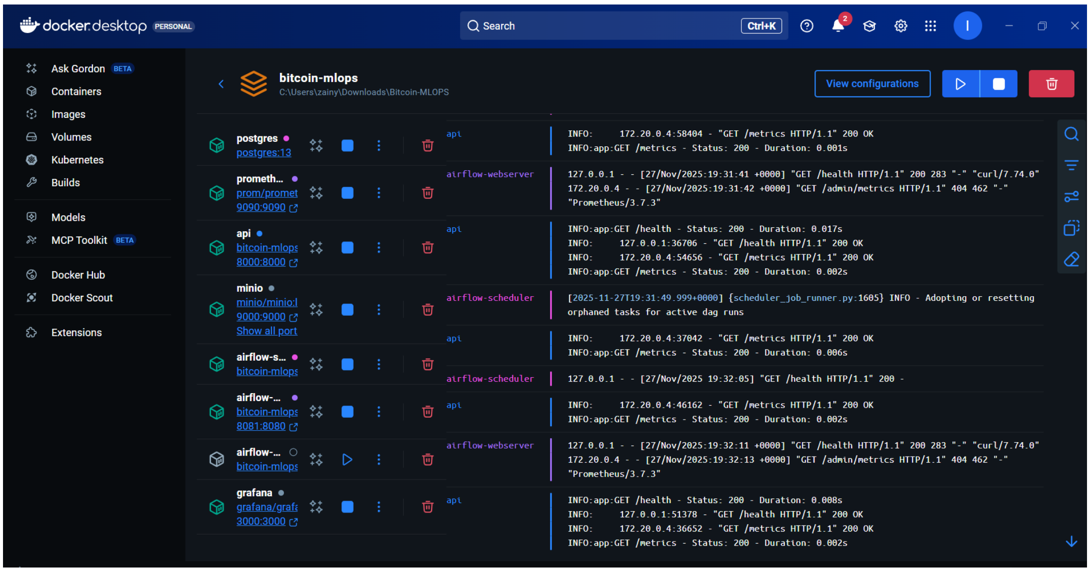

Figure 2: Airflow DAG and system orchestration

13.1.2 Dagdhub Experiments

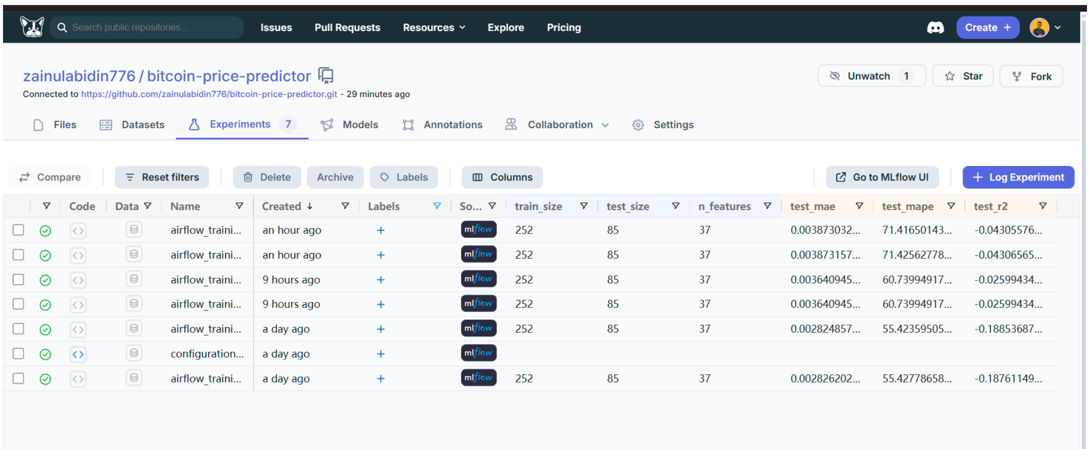

Figure 3: Dagshub showing all models successfully

13.2 Data Pipeline

13.2.1 Data Pipeline Overview

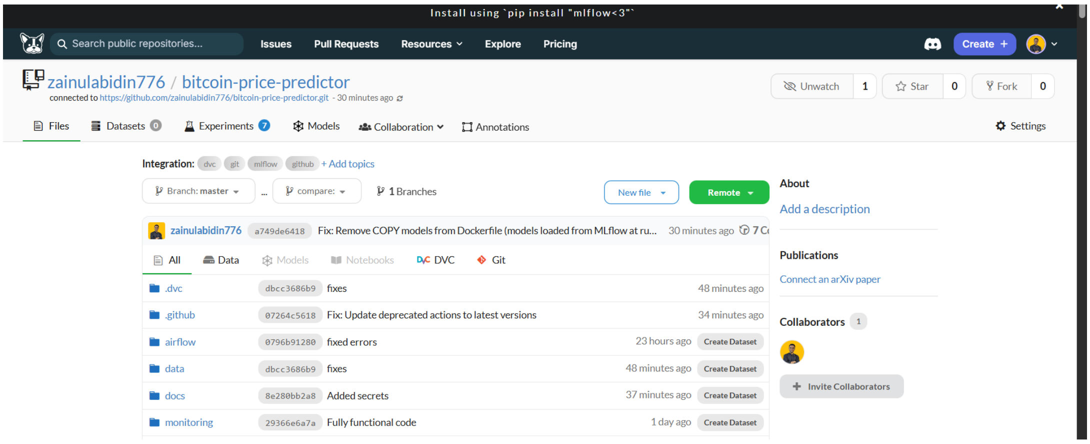

Figure 4: Airflow DAG orchestrating data extraction, quality checks, and feature engineering

13.3 Model Training and Tracking

13.3.1 Model Training and MLflow Integration

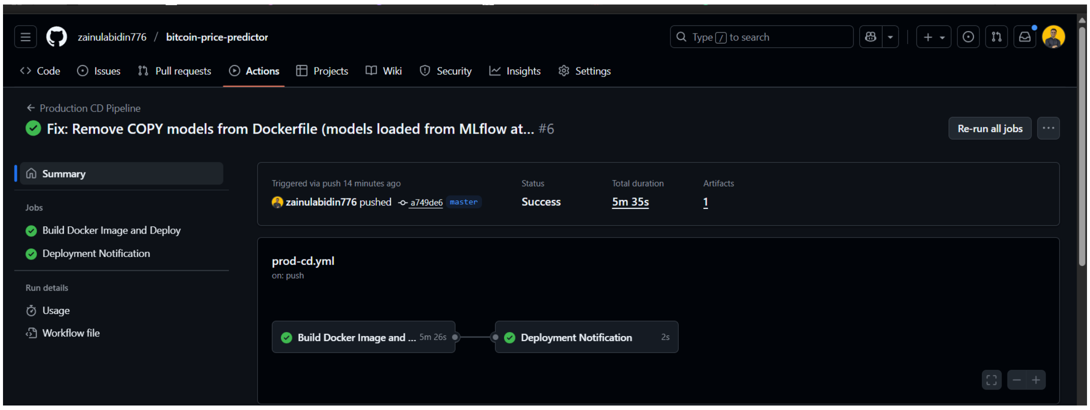

Figure 5: GitHub Actions workflow including model training and MLflow integration

13.4 CI/CD Pipeline

13.4.1 GitHub Actions Workflow

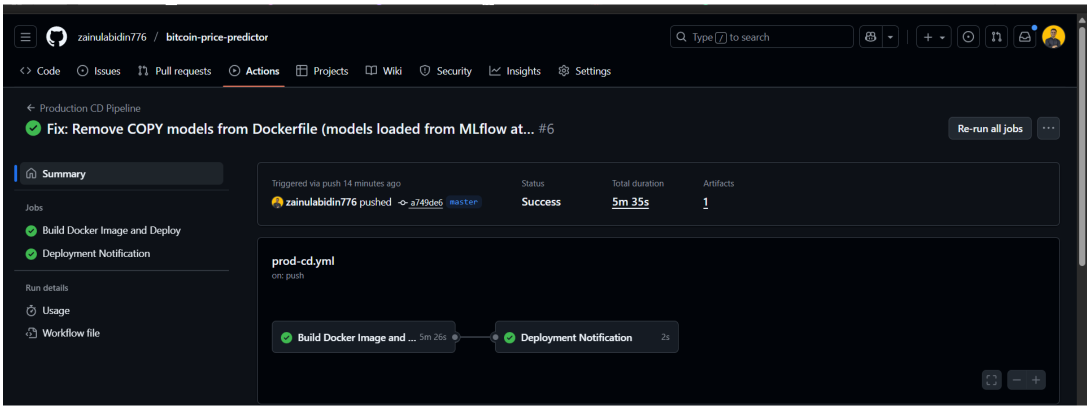

Figure 6: GitHub Actions CI/CD pipeline execution and workflow status

13.5 API and Deployment

13.5.1 API and Deployment

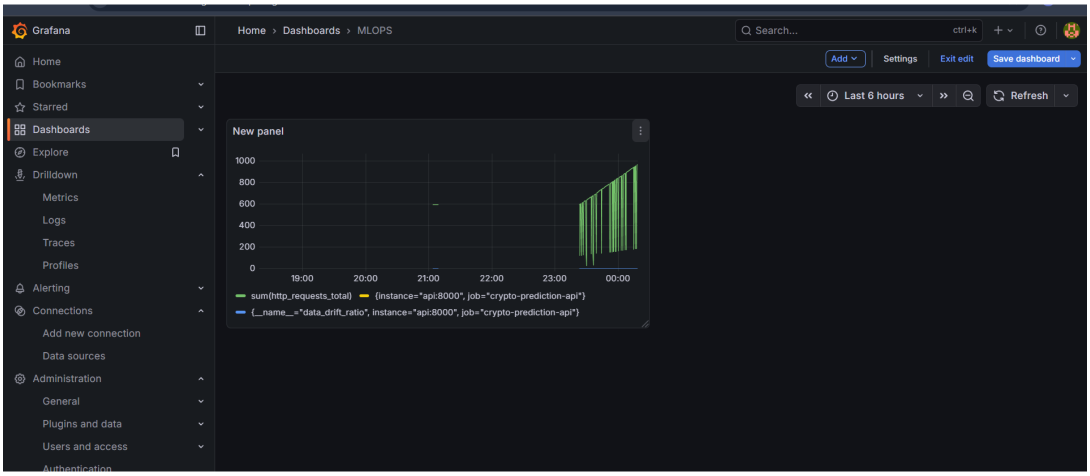

Figure 7: Grafana dashboard monitoring API performance, health, and prediction metrics

13.6 Monitoring and Observability

13.6.1 Prometheus Monitoring

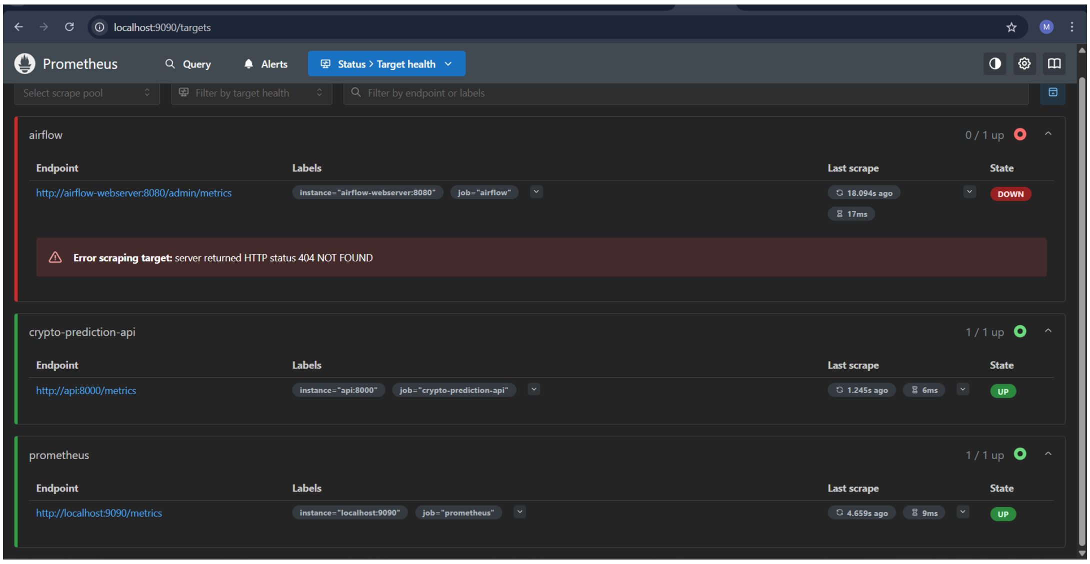

Figure 8: Prometheus monitoring dashboard showing targets and metrics

13.6.2 Grafana Dashboard

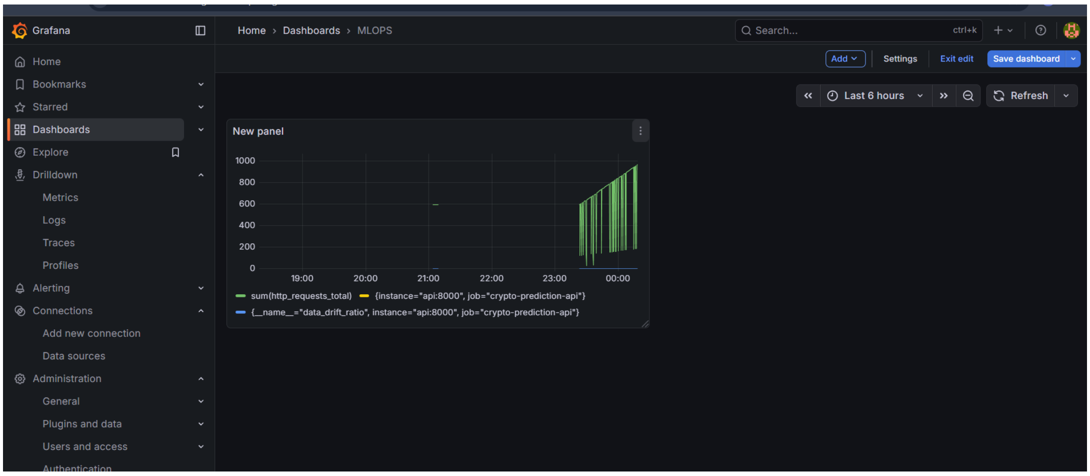

Figure 9: Grafana monitoring dashboard with Prometheus data source and visualization panels

13.7 Data Storage

13.7.1 MinIO Console

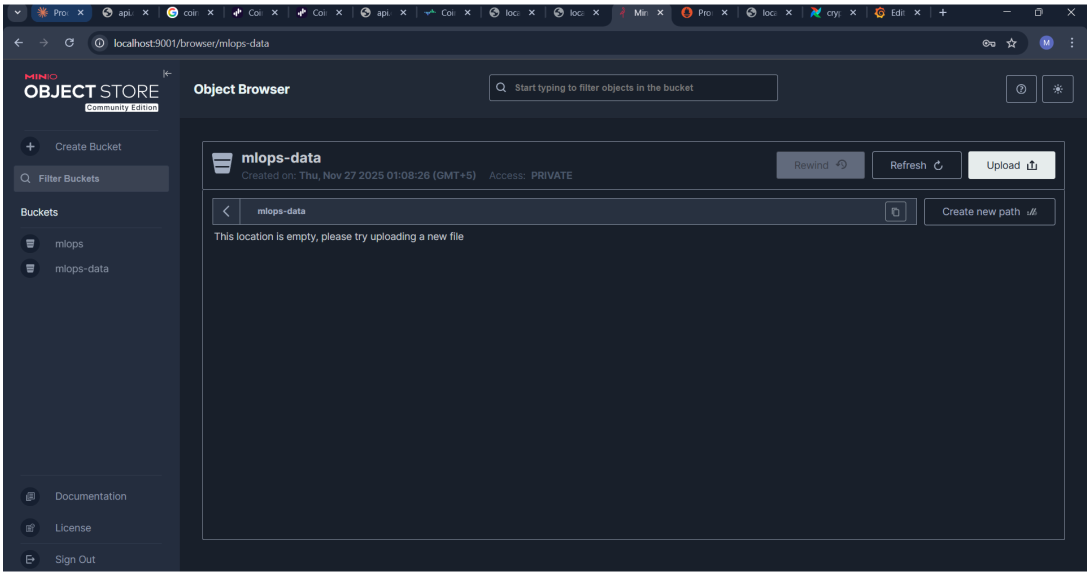

Figure 10: MinIO console showing mlops-data bucket and object storage

13.8 System Architecture

13.8.1 Complete System Overview

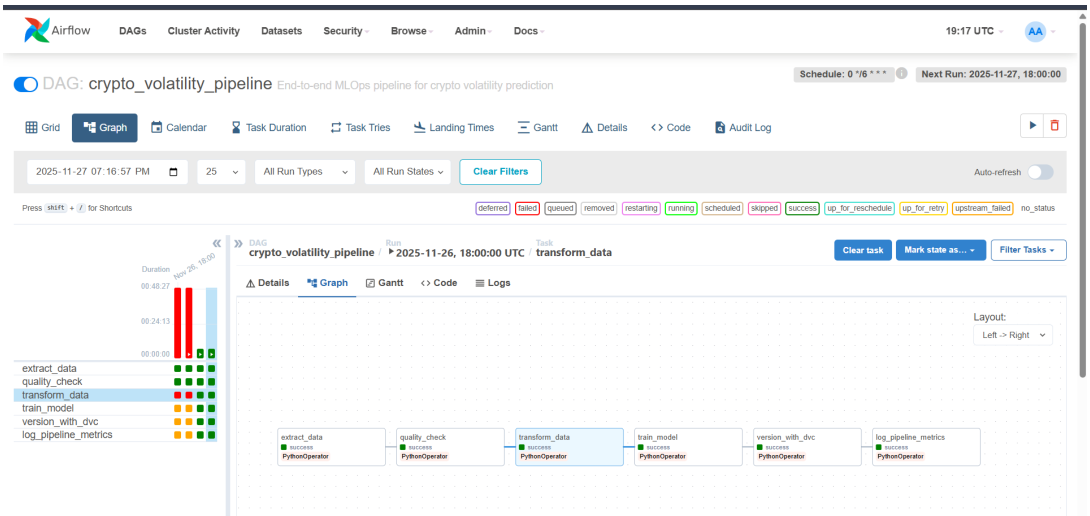

Figure 11: Complete system architecture - Airflow orchestrating all pipeline components

14 References

- Apache Airflow Documentation: https://airflow.apache.org/
- MLflow Documentation: https://mlflow.org/
- DagHub Documentation: https://dagshub.com/docs
- DVC Documentation: https://dvc.org/
- Prometheus Documentation: https://prometheus.io/
- Grafana Documentation: https://grafana.com/docs/
- CryptoCompare API: https://min-api.cryptocompare.com/
- XGBoost Documentation: https://xgboost.readthedocs.io/

15 Appendix

15.1 Project Structure

```
Bitcoin-MLOPS/
          airflow/
                dags/
                      crypto_pipeline_dag.py
                logs/
          src/
                data/
                      extract.py
                      transform.py
                      quality_check.py
                models/
                      train.py
                api/
                    app.py
          monitoring/
                prometheus.yml
                grafana/
          .github/
                workflows/
                    dev-ci.yml
                    test-ci.yml
                    prod-cd.yml
          docker-compose.yml
          Dockerfile
          Dockerfile.airflow
          requirements.txt
```

15.2 Key Metrics

- Total Lines of Code: 3,700
- Python Files: 7 core modules
- Configuration Files: 15+
- Documentation: 10+ files
- Test Coverage: Critical paths

15.3 Project Timeline

Phase	**Completion Date**
Phase I: Data Ingestion	November 20, 2025
Phase II: Model Management	November 22, 2025
Phase III: CI/CD	November 24, 2025
Phase IV: Monitoring	November 26, 2025
Final Documentation	November 26, 2025

Table 5: Project Timeline

15.4 Repository Information

- **GitHub Repository:** `https://github.com/zainulabidin776/bitcoin-price-predictor`
- **DagHub Repository:** `https://dagshub.com/zainulabidin776/bitcoin-price-predictor`
- **MLflow Tracking:** `https://dagshub.com/zainulabidin776/bitcoin-price-predictor.mlflow`