MLOps Case Study, Building a Real-Time Predictive System (RPS)

Deadline Nov 30, 2025

Context: You are working as an MLOps team for a forward-thinking tech company that needs to move beyond static, periodically trained machine learning models. Your goal is to develop a robust, automated, and continuously monitored **Real-Time Predictive System (RPS)** that handles live data streams and automatically adapts to changing patterns (**concept drift**).

The deliverable is a fully integrated MLOps pipeline that not only trains a model but also serves and monitors it in a production-like environment.

1. Phase I: Problem Definition and Data Ingestion

The first step is selecting a real-world problem and establishing a reliable, automated data feed.

The Challenge (Step 1)

Each group must select *one* predictive challenge centered around **time-series data** from a **free, live external API**.

Domain	**Example Free APIs**	**Predictive Task (Goal)**
Financial	Alpha Vantage, Twelve Data	**Stock/Crypto Volatility:** Predict the short-term volatility or closing price for a specific asset (e.g., next hour).
Environmental	OpenWeatherMap	**Localized Forecasting:** Predict a key weather variable (e.g., temperature, wind speed) for a city 4-6 hours out.

Logistics	Public Transit Data	**ETA/Delay:** Predict the delay or arrival time of a specific transit vehicle at a future stop.

The Orchestration Core: Apache Airflow (Step 2)

Your MLOps pipeline will be structured as a **Directed Acyclic Graph (DAG)** in **Apache Airflow**. This DAG must run on a schedule (e.g., daily) and be responsible for the entire ETL and model retraining lifecycle.

- **Extraction (2.1):** Use a Python operator to connect to your chosen API and fetch the latest live data. The raw data must be saved immediately, stamped with the collection time.
  - **Mandatory Quality Gate:** Implement a strict **Data Quality Check** right after extraction (e.g., check for >1% null values in key columns, or schema validation). If the data quality check fails, the DAG *must* fail and stop the process.
- **Transformation (2.2):** Clean the raw data and perform essential **feature engineering** specific to your time-series problem (e.g., creating lag features, rolling means, or time-of-day encodings).
  - **Documentation Artifact:** Use **Pandas Profiling** or a similar tool to generate a detailed data quality and feature summary report.[1] This report *must* be logged as an artifact to your MLflow Tracking Server (Dagshub).
- **Loading & Versioning (2.3 & 3):**
  - The final, processed dataset must be stored in a cloud-like object storage (e.g., **MinIO**, **AWS S3**, or **Azure Blob Storage**).
  - **Data Version Control (DVC):** Use **DVC** to version this processed dataset. The small `.dvc` metadata file will be committed to Git, but the large dataset file itself must be pushed to your chosen remote storage.

2. Phase II: Experimentation and Model Management

MLflow & Dagshub Integration (Step 4)

The Airflow DAG will trigger the model training script (train.py). This is where robust experimentation and artifact management come into play.

- **MLflow Tracking:** Use **MLflow** within the training script to track every experiment run. Log all **hyperparameters**, key **metrics** (e.g., RMSE, MAE, R-squared), and the final **trained model** as an artifact.
- **Dagshub as Central Hub:** Configure **Dagshub** to act as your remote **MLflow Tracking Server** and **DVC remote storage**. This ensures all three core components—**Code** (Git), **Data** (DVC), and **Models/Experiments** (MLflow)—are linked and visible in a single, collaborative UI.

3. Phase III: Continuous Integration & Deployment (CI/CD)

The code lifecycle must be rigorous, using a professional branching strategy and automated checks.

Git Workflow and CI Pipeline (Step 5)

- **Strict Branching Model (5.1):** Adhere strictly to the **dev**, **test**, and **master** branch model. All new work begins on feature branches and merges into dev.
- **Mandatory PR Approvals (5.3):** Enforce **Pull Request (PR) approval** from at least one peer before merging into the test and master branches.

GitHub Actions with CML (5.1 & 5.2)

Use **GitHub Actions** to automate the CI/CD pipeline, and integrate **CML (Continuous Machine Learning)** for automated reporting.

Merge Event	**CI Action**	**CML Integration**
Feature $\rightarrow$ dev	Run **Code Quality Checks** (Linting) and **Unit Tests**.	N/A
dev $\rightarrow$ test	**Model Retraining Test:** Trigger the Airflow DAG to run a full data and model training pipeline.	Use **CML** to automatically generate and post a **metric comparison report** in the Pull Request comments, comparing the newly trained model's performance against the existing production model in master. The merge should be blocked if the new model performs worse.
test $\rightarrow$ master	**Full Production Deployment Pipeline.**	N/A

Containerization and Deployment (5.4 & 5.5)

- **Docker Containerization (5.4):** The model must be served via a **REST API** (using **FastAPI** or **Flask**) inside a **Docker container**. This is your deployable unit.
- **Continuous Delivery:** The **test $\rightarrow$ master** merge must trigger the final CD steps:

1. Fetch the best-performing model from the MLflow **Model Registry**.
2. **Build** the final Docker image.
3. **Push** the tagged image (e.g., `app:v1.0.0`) to a container registry (e.g., **Docker Hub**).[2]
4. **Deployment Verification:** Run the container on a minimal host (e.g., a simple `docker run`) to verify the service starts and responds correctly to a health check.

4. Phase IV: Monitoring and Observability

A production system is incomplete without monitoring. You must implement tools to ensure the model remains reliable.

Prometheus and Grafana

- **Prometheus:** Embed a **Prometheus data collector** within your FastAPI prediction server. This service must expose metrics endpoints.
  - **Service Metrics:** Collect API inference **latency** and total **request count**.
  - **Model/Data Drift Metrics:** Expose crucial custom metrics, such as the **ratio of prediction requests containing out-of-distribution feature values** (a basic data drift proxy).
- **Grafana:** Deploy **Grafana** and connect it to your Prometheus instance.
  - **Live Dashboard:** Create a dashboard to visualize the key service and model health metrics in real time.
  - **Alerting Enhancement:** Configure a **Grafana Alert** to fire (e.g., log to a Slack channel or file) if the **inference latency** exceeds an acceptable threshold (e.g., 500ms) or if the **data drift ratio** spikes.

Summary of Tools to Integrate

Category	**Tools**	**Purpose in this Project**
Orchestration	**Airflow**	Schedule and automate the entire ETL $\rightarrow$ Training workflow.
Data/Model Mgmt	**DVC, MLflow, Dagshub**	Version data, track experiments, and centralize code, data, and models.
CI/CD	**GitHub Actions, CML, Docker**	Automate testing, model comparison, image building, and deployment.
Monitoring	**Prometheus, Grafana**	Collect service/model metrics, visualize performance, and alert on drift/latency.

The success of your project will be measured by the seamless **automation and integration** of these tools, proving your ability to manage an ML model across its entire lifecycle.