# Sentiment Analysis on X about Trending Topics

**Group 2**

**May 22, 2025**

| Member | Student ID | Contribution |
|---|---|---|
| Nguyen Phuong Hoai Ngoc | 11224722 | DevOps + Idea |
| Tran Ngoc Son | 11225650 | DS + Monitoring |
| Tran The Duy | | DE + API |
| Hoang Thuy Duong | 11221551 | DS + Monitoring |
| Nguyen Tran Phuong Ngan | 11224591 | Tester + Report + Slide |

# 1. Problem Statement

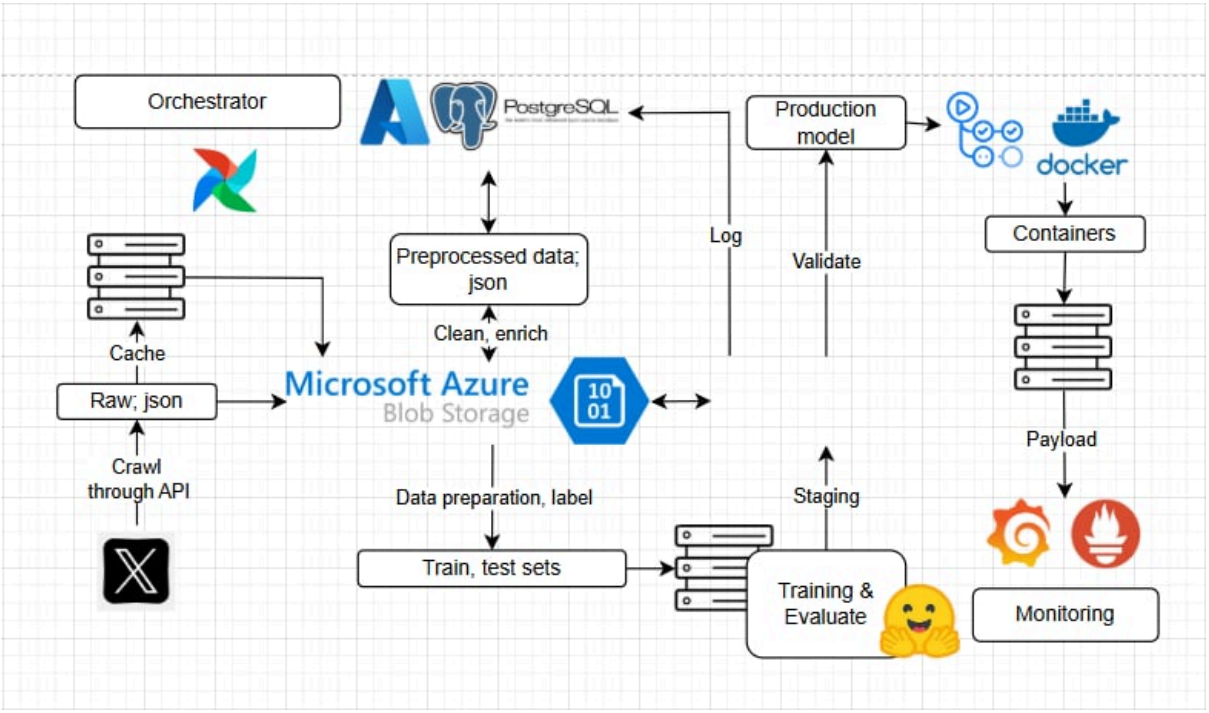
Social media platforms like X are rich sources of public opinion, especially on trending topics. However, manually analyzing the sentiment of millions of tweets is infeasible. Businesses, researchers, and policymakers need automated systems to classify sentiments (positive, neutral, negative) in real-time, enabling them to understand public reactions and make data-driven decisions. The challenge lies in building a scalable, automated machine learning pipeline that can collect, process, and analyze tweets efficiently while ensuring model accuracy, deployment reliability, and performance monitoring. This project aims to address this by implementing an MLOps pipeline for sentiment analysis on X, focusing on automation through CI/CD and a streamlined workflow.

# 2. Outline Report

This report is structured as follows:

- **Problem Statement:** Defines the challenge of sentiment analysis on X and the need for an automated pipeline.
- **Overall Workflow:** Describes the end-to-end process from data collection to monitoring.
- **Database Design:** Details the database structure supporting the workflow.
- **Data Pipeline:** Outlines the data flow and automation using Airflow operators.
- **Model Training:** Explains the fine-tuning process with ModernBERT, logging to MLflow, and setting aliases based on the provided code.
- **CI/CD Implementation:** Explains the Continuous Integration and Continuous Deployment setup, including tools and processes.
- **Challenges and Improvements:** Discusses obstacles faced and proposed solutions.
- **Conclusion:** Summarizes the project outcomes and future directions.

# 3. Overall Workflow




*H1. Overall workflow*

The overall workflow for sentiment analysis on X is designed to automate the machine learning lifecycle, leveraging a combination of tools and processes to ensure scalability and efficiency. The workflow is divided into three key stages: **Data Pipeline**, **Operations: CI/CD**, and **Development**. Below is a detailed description of how our team implemented each stage.

## 3.1. Data Pipeline

Our team implemented a robust data pipeline to handle the collection, processing, and storage of tweet data, orchestrated by Airflow. The process is as follows:

- **Data Collection**: Tweets are crawled from the X API every 30 seconds using the `CrawTweetsOperator`, with local caching every 1 hour to manage data volume. Raw JSON data is initially stored in the `DATA_RAW` stage.
- **Data Storage and Processing**: The `PushDataLakeOperator` pushes raw data to Microsoft Azure Blob Storage, where it is cleaned and enriched using a 10:01 split strategy (10% validation, 1% training/evaluation). The `AzureBlobNewFileSensor` triggers the `PullStagingOperator` to move data to the `TWEET_STAGING` table in the SQL Warehouse (PostgreSQL) for validation.
- **Data Preparation and Modeling**: Data is prepared and labeled to create train and test sets, which are then used by the `Training & Evaluate` process to fine-tune

the ModernBERT model. The `SentimentPredictionOperator` applies the model to generate predictions, storing results in the `TWEET_PRODUCT` table alongside the original data.

- **Output and Monitoring**: Processed data and predictions are sent as payloads to Docker containers in the Production environment. Monitoring is enabled through Prometheus and Grafana, ensuring real-time oversight of the pipeline.

This pipeline ensures a continuous flow from raw data ingestion to processed outputs, with Airflow coordinating each step and Azure Blob Storage acting as the central data hub.

## 3.2. Operations: CI/CD

The operations stage focuses on automating the deployment and maintenance of the system using Continuous Integration and Continuous Deployment (CI/CD) with GitHub Actions. Our implementation includes:

- **Continuous Integration (CI)**: In the Dev environment, every push or pull request to the `dev` branch triggers a GitHub Actions workflow to validate code and build a Docker image with `app/main.py` and `requirements.txt`. MLflow logs model artifacts to Azure Blob Storage, and the image is pushed to a Dev repository.
- **Continuous Deployment (CD)**: After successful CI, the workflow deploys the image to the Dev environment using `docker-compose-dev.yml`. Post-deployment tests verify API endpoints . Approved changes are merged into the `main` branch, triggering a Production deployment with `docker-compose-prod.yml`, including scaled replicas and secure ports. Rollbacks are automated via GitHub Actions if issues are detected.

## 3.3. Development

The development stage encompasses the creation and iteration of the model and infrastructure, leveraging a collaborative and iterative approach. Our team's process includes:

- **Model Development**: The ModernBERT model is fine-tuned using preprocessed data from `cleaned_data.csv`, with PyTorch's `DataLoader` and `Trainer`. MLflow tracks experiments, logging metrics and artifacts, while aliases manage version control.

- **Infrastructure Setup**: The Hyper PC with an RTX3060 GPU supports training and inference. PostgreSQL stores tweet data, and Docker containers host the FastAPI app. Airflow orchestrates tasks, with Prometheus and Grafana providing monitoring dashboards.
- **Iteration**: The team iterates based on monitoring insights, adjusting hyperparameters or retraining with balanced datasets to address imbalances.

This stage ensures continuous improvement of the model and infrastructure, driven by real-time feedback and performance metrics.

### 3.4. Introduction to Tool Services in the Workflow

The workflow relies on a suite of tools and services to ensure efficiency and scalability.

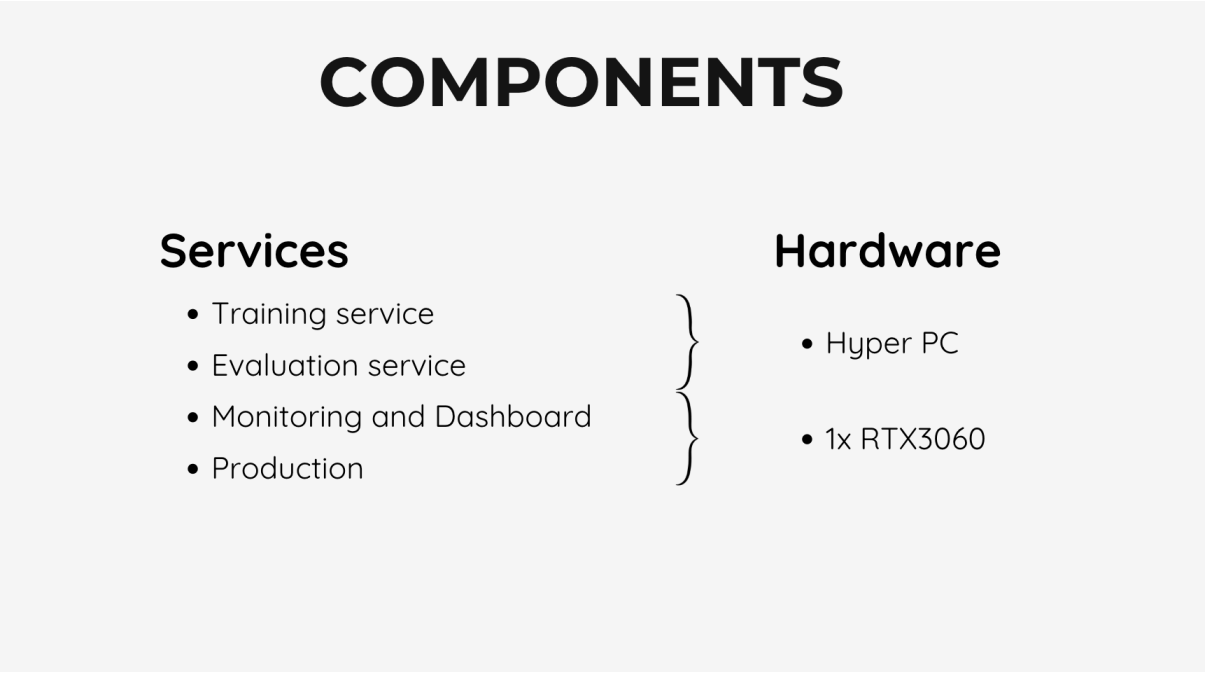
| Tool Services | Details |
|---|---|
| **MLflow** | A platform for managing the machine learning lifecycle, including experiment tracking, model logging, and artifact management. It integrates with Azure Blob Storage to store preprocessed data and model artifacts. |
| **Airflow** | Acts as the orchestrator, scheduling and coordinating tasks such as data collection, processing, training, and deployment through directed acyclic graphs (DAGs). |
| **PostgreSQL** | A robust open-source database used for storing and querying tweet data, integrated with Airflow for persistent storage and validation. |
| **Docker** | Facilitates containerization of the FastAPI application and other services (e.g., MLflow, Prometheus), ensuring portability and consistency across deployment environments. |

| | |
|---|---|
| **Prometheus** | Collects metrics from the deployed model, such as request latency and error rates, enabling real-time performance monitoring. |
| **Grafana** | Provides visualization of metrics collected by Prometheus, offering dashboards for monitoring model health and performance. |

These tools collectively enable a seamless, automated workflow from data ingestion to production deployment and monitoring.

### 3.5. System Components

The system is built on a combination of services and hardware to support the end-to-end workflow:




*H2. System Components*

#### 3.5.1. Services

- **Training Service**: Responsible for fine-tuning the ModernBERT model using the TweetDataset and Trainer from the provided code. This service runs on the Hyper PC, leveraging the RTX3060 for accelerated training.

- **Evaluation Service**: Evaluates model performance on the test set, computing metrics like accuracy using the compute_metrics function, and logs results to MLflow.
- **Monitoring and Dashboard**: Utilizes Prometheus to collect real-time metrics (e.g., latency, accuracy) and Grafana to provide interactive dashboards for visualizing system health and model performance across Dev and Production environments.
- **Production**: Deploys the trained model via Docker containers in the Production environment, serving predictions through the FastAPI app and handling live tweet analysis.

### 3.5.2. Hardware

- **Hyper PC**: A high-performance computing system serving as the primary machine for training, evaluation, and development tasks. It provides the computational power needed for processing large datasets and training deep learning models.
- **1x RTX3060**: A NVIDIA RTX3060 GPU integrated into the Hyper PC, offering 12GB VRAM and CUDA cores for accelerating model training and inference, significantly reducing training time for the ModernBERT model.

These components ensure the system is both scalable and capable of handling the computational demands of sentiment analysis on X.

# 4. Database Design

**1. Tweet_staging – Temporary Storage**

| Column Name | Type | Description |
|---|---|---|
| tweet_id | BIGINT | Unique tweet ID (from Twitter) |
| content | TEXT | Cleaned tweet text |
| type | VARCHAR | Optional tag, e.g. trending, hashtag, user |
| inference_done | BOOLEAN | Whether model inference is completed |
| inferred_at | TIMESTAMP | Time inference was finished |
| moved_to_product | BOOLEAN | If this row was transferred to tweet_product |

**2. tweet_product – Final Sentiment Storage**

| Column Name | Type | Description |
|---|---|---|
| staging_id | BIGINT | FK to tweet_staging.tweet_id (audit + trace) |
| predicted_sentiment | SMALLINT | 0 = Negative, 1 = Neutral, 2 = Positive |
| sentiment_score | FLOAT | Confidence or probability score from the model |
| type | VARCHAR | Same as staging (preserved for analysis) |
| processed_at | TIMESTAMP | When the result was written to product table |

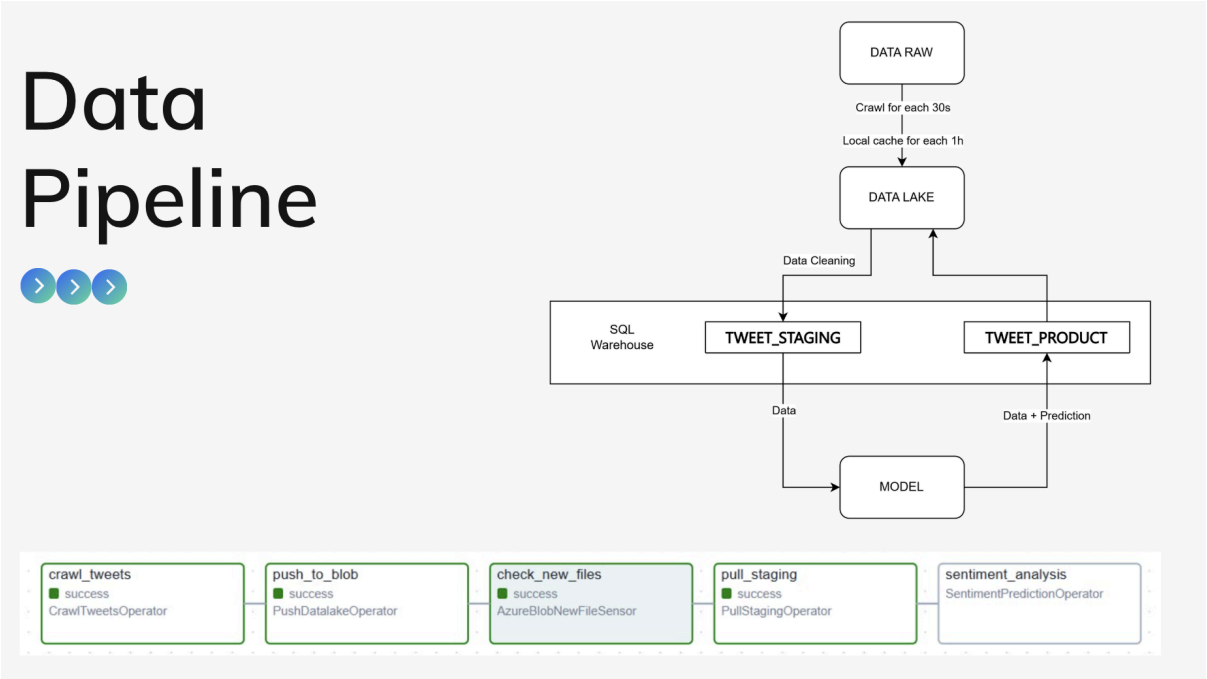
*H3. Table Description*

The database design supports the workflow with two key tables:

- **TWEET_STAGING**: Stores raw tweet data (tweet_id, topic, content, crawl_at, sentiment_score, predicted_sentiment, updated_at, moved_to_product, moved_at, processed_done, processed_at) for validation and preprocessing.
- **TWEET_PRODUCT**: Contains cleaned and processed tweets (staging_tweet_id, tweet_id, topic, sentiment_score, predicted_sentiment, updated_at, created_at, processed_at) for downstream tasks.

This structure ensures data integrity and facilitates the transition from raw to processed states.

## 5. Data Pipeline

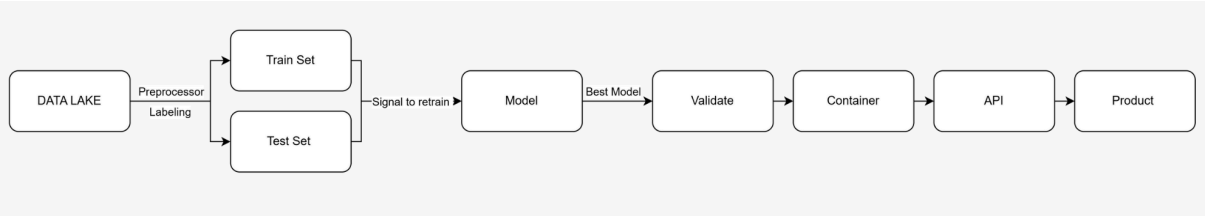



*H4. Data Pipeline*

The data pipeline automates the flow from collection to prediction:

- Crawling: CrawTweetsOperator crawls tweets every hour, with local caching every 30s.
- Storage: PushDataLakeOperator pushes data to Azure Blob Storage, triggered by AzureBlobNewFileSensor.
- Processing: PullStagingOperator moves data to TWEET_STAGING, where SentimentPredictionOperator applies the model.

# 6. Model Training




*H5. Model Pipeline*

## 6.1. Fine-Tuning the Model

### 6.1.1. Data Extraction and Labeling

The dataset was initially extracted from the `table_staging` table within a PostgreSQL database. This table contains user-generated tweets, where each entry includes a `tweet` column (storing the text content) along with various metadata fields. Since the raw data did not come with predefined sentiment labels, a zero-shot classification approach was employed using a pre-trained Large Language Model (LLM).

The LLM was tasked with classifying each tweet into one of three sentiment categories: "positive", "negative", or "neutral". The model was prompted accordingly for each tweet, enabling sentiment annotation without the need for a manually labeled dataset.

After labeling, the dataset underwent a cleaning process to improve quality. This involved the removal of:

- Tweets with missing text.
- Tweets containing corrupted or unreadable content.

To prepare the data for training and evaluation, the cleaned dataset was divided into two subsets:

- **Training set**: 80% of the data.
- **Evaluation set**: 20% of the data.

To maintain the original distribution of sentiment labels across both sets, stratified sampling was used. This technique ensures that each sentiment class ("positive", "negative", "neutral") is proportionally represented in both subsets. Stratified sampling reduces the risk of introducing class imbalance during model development and enables more reliable evaluation.

### 6.1.2. Tokenization and Input Preparation:

Tokenization of the tweet data was carried out using the AutoTokenizer class provided by the Hugging Face *Transformers* library. To match the architecture of the models evaluated in this study, two separate tokenizers were used:

- For ModernBERT, the tokenizer from the answerdotai/ModernBERT-base model was utilized.
- For BERTBase, the widely used bert-base-uncased tokenizer was applied.

To standardize input lengths and support efficient batch processing, a maximum sequence length of 128 tokens was enforced. The preprocessing pipeline involved the following steps:

- **Truncation**: Tweets longer than 128 tokens were truncated to fit the maximum length.
- **Padding**: Tweets shorter than 128 tokens were padded using a designated special padding token to ensure uniform length.
- **Token Conversion**: The resulting tokens were transformed into numerical indices corresponding to the tokenizer's vocabulary.
- **Tensor Conversion**: All inputs were converted into PyTorch tensors to be compatible with the downstream model training pipeline.
- **Attention Masking**: For each tokenized input, an attention mask was generated. This binary mask identifies which tokens are real (1) and which are padding (0), allowing the model to focus only on meaningful content during training and inference.

This preprocessing pipeline ensured that all textual inputs were consistently formatted and efficiently processed in batches during model training and evaluation.

#### 7.1.3.Model Configure and Training:

The training configuration included the following hyperparameters, selected based on preliminary experiments:

• **Epochs**: 3 epochs for ModernBERT and 4 epochs for BERTBase, allowing sufficient iterations for convergence while minimizing overfitting.

• **Batch Size**: A per-device training batch size of 16 and an evaluation batch size of 64 were used, balancing memory constraints and gradient stability.

• **Learning Rate**: An initial learning rate of $2\times10-5$ was applied, providing a stable starting point for fine-tuning the pre-trained weights.

• **Learning Rate Scheduler:** A linear scheduler with a warm up period of 500 steps was implemented. During the warmup phase, the learning rate increased linearly from zero to the initial value, followed by a linear decay to zero over the remaining steps, aiding in gradient stabilization.

• **Optimizer:** The AdamW optimizer was employed with parameters $\beta 1 = 0.9$, $\beta 2 = 0.999$, and $\epsilon = 1 \times 10^{-8}$, ensuring robust weight updates.

• **Evaluation Strategy:** Model evaluation was performed at the end of each epoch, using the evaluation set to compute accuracy and monitor performance.

The training process involved minimizing the cross-entropy loss between the predicted sentiment probabilities and the ground truth labels. The models' weights were updated iteratively, with gradients computed using backpropagation. To prevent overfitting, early stopping was not explicitly implemented but could be considered in future iterations based on validation loss trends. The fine-tuned models were saved locally in the model/ModernBERT and model/BERTBase directories, containing the model weights, configuration files, and tokenizer files.

## 6.2. Logging to MLflow

To ensure reproducibility and systematic monitoring of model development, MLflow was employed for experiment tracking and metric logging. A dedicated experiment, named `SentimentAnalysisExp`, was created within the MLflow tracking server to organize all fine-tuning runs of the ModernBERT and BERTBase models.

Each training run was logged with a unique identifier, composed of the model name and a timestamp, allowing for precise traceability of individual experiments. This structure made it possible to compare different runs and track improvements over time.

**Metrics and Artifacts Logged**

During the evaluation phase, key classification metrics were calculated on the evaluation set and logged to MLflow:

- **Accuracy**
- **Precision**
- **Recall**
- **F1 Score**

These metrics provided a comprehensive performance overview across the three sentiment categories: *positive*, *negative*, and *neutral*.

Additionally, a confusion matrix was computed for each model to analyze the relationship between predicted and actual labels. This matrix was logged as a visual artifact, structured as follows:

- **Rows**: Represent true sentiment labels.
- **Columns**: Represent predicted sentiment labels.

Logging the confusion matrix allowed for a detailed inspection of misclassification patterns and class-specific performance weaknesses.

**Storage and Accessibility**

All logged metrics and artifacts—including performance scores and confusion matrices—were stored in the MLflow tracking server. This centralized logging setup provided:

- Easy access to past experiments
- Support for side-by-side model comparison
- A foundation for iterative model improvements

The use of MLflow thus played a critical role in enabling scalable experimentation and transparent evaluation throughout the model development lifecycle.

## 6.3. Setting Aliases

To streamline the deployment process and maintain model quality, a single alias named "champion" was introduced to represent the best-performing model within the experiment lifecycle. This alias-based mechanism enabled seamless updates to the production model without manual intervention.

The training and evaluation workflow was orchestrated using a Directed Acyclic Graph (DAG). This DAG ensured a well-defined, repeatable sequence of operations that included data preprocessing, model training, evaluation, and deployment. Each step was executed in a structured manner, supporting modular experimentation and reproducibility.

The F1 score was selected as the primary evaluation metric due to its ability to balance precision and recall, making it particularly suitable for the imbalanced sentiment classification

task. After each training cycle, both the ModernBERT and BERTBase models were evaluated on a held-out test set.

The evaluation followed this procedure:

- The F1 score of the latest trained model was computed.
- This score was compared with that of the current **"champion"** model.
- If the new model achieved a higher F1 score, it was promoted to **"champion"** by reassigning the alias.

**Production Deployment**

Once the champion model was identified, it was deployed to a production environment for real-time inference. The deployment process involved:

- **Exporting model artifacts**, including learned weights, tokenizer configurations, and metadata.
- **Deploying the model as a FastAPI**, enabling external systems to submit tweet data and receive sentiment predictions in real time.
- **Integrating the API into the operational pipeline**, ensuring that only the most accurate model—the current **“champion”**—was used for inference.

This deployment strategy ensured consistent use of the highest-performing model, providing a robust and scalable solution for real-time sentiment classification on streaming or batch tweet data.

# 7. Monitoring

Monitoring is a critical aspect of the system, ensuring the reliability and performance of both Dev and Production environments.

## 7.1. Monitoring Tools

- Prometheus: Continuously scrapes metrics from the FastAPI app, Airflow, and Docker containers. Key metrics include:
    - Request Latency: Average time to process a prediction request.
    - Throughput: Number of requests processed per second.
    - Error Rate: Percentage of failed requests.

    - Model Accuracy: Periodic evaluation accuracy logged by MLflow and exposed as a metric.
- Grafana: Provides customizable dashboards to visualize Prometheus metrics. Dashboards include time-series graphs for latency, heatmaps for error distribution, and gauges for real-time accuracy, accessible to the team for decision-making.

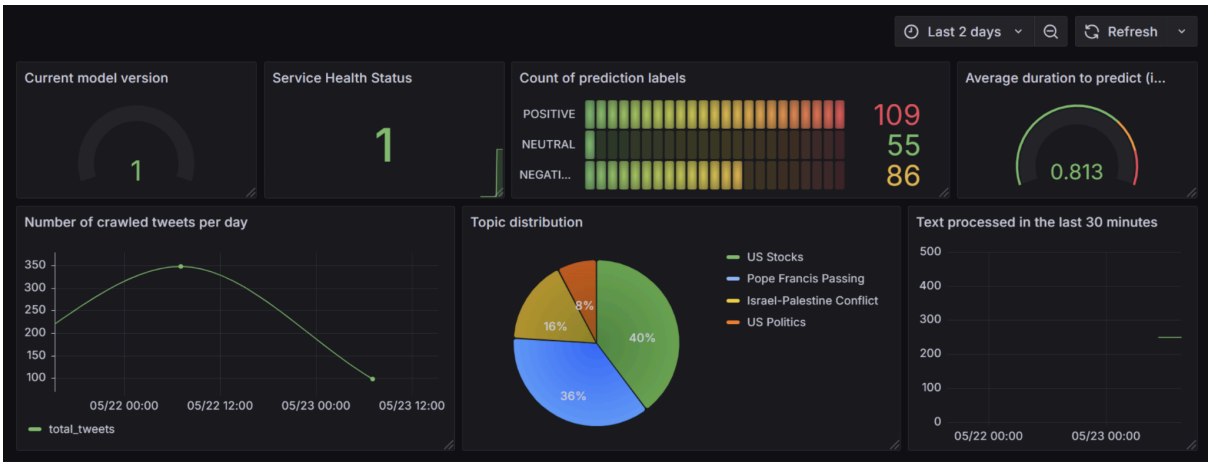
## 7.2. Monitoring Process

The monitoring process was meticulously designed and implemented by our team to ensure comprehensive oversight of the system's performance and data integrity.

Initially, a FastAPI application was developed to serve the trained ModernBERT model, incorporating essential performance metrics such as latency, accuracy, and prediction label counts to facilitate real-time evaluation of model efficacy. This application serves as the foundation for monitoring, providing a structured interface to expose these metrics. The dashboard, constructed using Grafana, was subsequently integrated with two primary data sources to enhance its analytical capabilities.

The first source, PostgreSQL, was connected to display the contents of the database, including the TWEET_STAGING and TWEET_PRODUCT tables, thereby offering a detailed view of raw and processed tweet data. The second source involved the integration of the FastAPI application, which required an intermediary step through Prometheus to collect and process the exposed metrics effectively. This integration process entailed running the FastAPI application to generate the necessary performance data, followed by the execution of Prometheus to scrape these metrics, and finally connecting the processed data to Grafana for visualization. This methodical approach ensured that the dashboard could reflect both the operational status of the model and the underlying data in a cohesive manner.

## 7.3. Dashboard Monitoring Analysis




*H6. Monitoring Metric*

The dashboard provides a comprehensive view of the system's performance through Grafana, visualizing key metrics collected by Prometheus. Below is an analysis of each chart in the dashboard:

- Current Model Version (Gauge):
    - Displays the current model version in use, currently at version 1. This indicates the system is running the initial deployed model, with no updates applied in the observed period.
- Service Health Status (Gauge):
    - Indicates the health status of the service, currently at 1 (healthy). A value of 1 suggests all critical services (FastAPI, Airflow, MLflow) are operational, with no immediate downtime or failures.
- Count of Prediction Labels (Bar Chart):
    - Shows the distribution of prediction labels over the last couple days: 109 Positive, 55 Neutral, and 86 Negative. This distribution highlights a higher prevalence of positive sentiments, possibly reflecting a bias in the crawled tweet sample or an optimistic trend in recent topics.
- Average Duration to Predict (Gauge):
    - Measures the average time to process a prediction, currently at 0.813 seconds. This latency is within acceptable limits for real-time analysis but suggests room for optimization, especially during peak loads.
- Number of Crawled Tweets per Day (Line Chart):
    - Tracks the number of tweets crawled daily, peaking at 350 on May 22 at 00:00 and declining to 200 by May 23 at 12:00. The downward trend may indicate reduced activity on X or a need to adjust the crawling frequency.

- Topic Distribution (Pie Chart):
    - Illustrates the proportion of topics: 40% US Stocks, 36% Pope Francis Passing, 18% Israel-Palestine Conflict, and 6% US Politics. The dominance of US Stocks and Pope Francis Passing suggests these are current trending topics influencing the dataset.
- Text Processed in the Last 30 Minutes (Time Series):
    - Shows the number of tweets processed in the last 30 minutes, currently at 0. This could indicate a temporary halt in processing, possibly due to a scheduled maintenance or a delay in data ingestion.

# 8. CI/CD Implementation

The CI/CD pipeline is split into two environments—**Dev** and **Production**—to ensure safe and reliable deployment. The process uses GitHub Actions to automate validation, building, and deployment across these environments.

## 8.1. Continuous Integration (CI) - Dev Environment

- **Purpose**: CI validates changes in the Dev environment to catch issues early.
- **Code Validation**:
    - On every push or pull request to the dev branch, a GitHub Actions workflow triggers a test script to validate code quality, syntax, and functionality.
    - Unit tests ensure that the FastAPI application (app/main.py) and model utilities (app/model_utils.py) function correctly.
- **Build and Log Model**:
    - A Docker image is built using the Dockerfile in the Dev environment, incorporating app/main.py and requirements.txt.
    - MLflow logs the model artifacts to Azure Blob Storage or an MLflow server, tagged for the Dev environment.
    - The Docker image is pushed to a Dev-specific repository in Docker Hub or Azure Container Registry.
- **Automation**: The CI pipeline runs automatically on changes to the dev branch, ensuring all changes are thoroughly tested before deployment.

## 8.2. Continuous Deployment (CD) - Dev Environment

- **Purpose**: Deploy validated changes to the Dev environment for further testing.
- **Deployment Process**:

    - Upon successful CI, the GitHub Actions workflow deploys the Docker image to the Dev environment.
    - The Dev environment runs services like the FastAPI app, Airflow, MLflow, and a lightweight monitoring stack (Prometheus and Grafana) for testing purposes.
    - Deployment uses Docker Compose with a docker-compose-dev.yml configuration, specifying Dev-specific settings.
- **Validation**:
    - Post-deployment tests are run to verify the API endpoints and model performance in the Dev environment.
    - Metrics such as request latency and accuracy are monitored using Prometheus and visualized in Grafana.
- **Approval Gate**: Changes in the Dev environment are manually reviewed by the team. If approved, the changes are promoted to the Production environment.

## 8.3. Continuous Deployment (CD) - Production Environment

- **Purpose**: Deploy thoroughly tested changes from the Dev environment to Production for live usage.
- **Deployment Process**:
    - After approval, a GitHub Actions workflow merges the dev branch into the main branch, triggering the Production deployment.
    - The Docker image is rebuilt with a Production tag to ensure consistency, using the same artifacts logged by MLflow in the Dev phase.
    - The image is deployed to the Production environment using a docker-compose-prod.yml configuration, which includes production-grade settings.
    - Production services include the FastAPI app, Airflow, MLflow, Prometheus, and Grafana, all optimized for high availability and performance.
- **Rollback**:
    - In case of deployment failures, Docker containers in Production can be rolled back to the previous stable version.
    - Rollback is automated via GitHub Actions, triggered by monitoring alerts or manual intervention.
- **Monitoring**:
    - Production monitoring is more robust, with Prometheus collecting detailed metrics and Grafana providing real-time dashboards for team oversight.

### 8.4. Tools and Integration

- **GitHub Actions**: Manages CI/CD workflows, with separate jobs for CI (Dev), CD (Dev), and CD (Production).
- **Docker**: Ensures consistent deployment across Dev and Production environments using containerization.
- **MLflow**: Tracks experiments and logs model artifacts across both environments, ensuring traceability from Dev to Production.

## 9. Challenges and Improvements

### 9.1. Challenges

During the development and deployment of this project, the team encountered several practical challenges that reflect the complexities of implementing an MLOps pipeline in a real-world setting.

One significant challenge was the initial setup of the infrastructure, particularly in configuring the Hyper PC and RTX3060 GPU environment for training and inference. The team faced difficulties in installing compatible drivers and CUDA libraries, which led to delays in starting the training phase, as the system frequently crashed during early runs due to misconfigured GPU settings.

### 9.2. Improvements

To resolve the infrastructure setup issues, the team plans to create a detailed setup guide for the Hyper PC and RTX3060 environment, documenting the exact driver versions, CUDA configurations, and troubleshooting steps to streamline future setups and reduce onboarding time for new team members.

For optimizing environments, the team intends to implement resource isolation by allocating dedicated memory and CPU quotas for Dev and Production environments using Docker resource limits, ensuring that simultaneous operations do not overwhelm the Hyper PC.

## 10. Conclusion

The project successfully implements a CI/CD-driven MLOps pipeline for sentiment analysis on X, addressing the challenge of automating sentiment classification for large-scale social

media data. By leveraging Airflow for orchestration, GitHub Actions for CI/CD across Dev and Production environments, Docker for deployment, and MLflow for model management, the workflow ensures scalability and reliability. The addition of system components (Training, Evaluation, Monitoring, Production) and hardware (Hyper PC, RTX3060) enhances computational capability, while robust monitoring with Prometheus and Grafana ensures performance oversight. Despite challenges such as infrastructure setup, environment optimization, and connectivity issues, the project demonstrates the power of automation in handling big data tasks.

## 10.1. Lessons

Throughout the development and deployment of this sentiment analysis system, our team gained valuable insights:

- **Importance of Automation**: Automating the data pipeline with Airflow and CI/CD with GitHub Actions significantly reduced manual effort and errors, ensuring consistency across environments. However, we learned that careful configuration is crucial to avoid bottlenecks, as seen in the 0 processed texts issued in the last 30 minutes.
- **Monitoring as a Lifeline**: Real-time monitoring with Prometheus and Grafana proved essential for identifying issues like pipeline blockages and latency spikes. We learned the importance of setting up detailed dashboards early and adding custom metrics to catch issues proactively.
- **Data Imbalance Challenges**: The prediction label distribution (109 Positive, 55 Neutral, 86 Negative) highlighted the need for balanced datasets. We learned that adjusting crawling strategies and retraining with representative data are key to improving model fairness.
- **Scalability Requires Planning**: Handling peak tweet volumes revealed the need for scalable infrastructure. We learned that tools like Docker and Azure Blob Storage are critical, but future scalability may require additional technologies like Apache Kafka for real-time streaming.

These lessons have shaped our understanding of MLOps, emphasizing the need for robust automation, proactive monitoring, and continuous iteration to build reliable ML systems.

## 10.2. Real-World Scenarios

The sentiment analysis system developed in this project has wide-ranging applications in real-world scenarios, particularly in contexts where understanding public opinion in real-time is critical:

- **Business Intelligence**: Companies can use this system to monitor customer sentiment on X about their products or services. For example, a tech company could track reactions to a new product launch (e.g., 40% US Stocks discussions) to gauge market reception and adjust marketing strategies accordingly.
- **Crisis Management**: Governments and organizations can leverage the system to detect public sentiment during crises, such as the Israel-Palestine Conflict (18% of topics). Real-time analysis can help identify negative sentiment spikes (e.g., 86 Negative predictions) and inform timely interventions or communication strategies.
- **Market Research**: Market researchers can analyze trending topics like Pope Francis Passing (36%) to understand public interest and emotional responses, providing insights for media, advertising, or content creation strategies.
- **Political Analysis**: Political campaigns can use the system to monitor voter sentiment on X, especially on topics like US Politics (6%), to tailor campaign messages and predict electoral trends based on sentiment distribution.