HANOI UNIVERSITY OF SCIENCE AND TECHNOLOGY

SCHOOL OF INFORMATION COMMUNICATION TECHNOLOGY

IT4043E

Big Data Storage and Processing

CAPSTONE PROJECT REPORT

Scalable Lakehouse for Financial Analytics

Supervised by:

Dr. Tran Viet Trung

***

Group 3

Name	**Roles**	**ID**
Bui Khac Chien	Batch Processing, Modelling	20220059
Bach Nhat Minh	OLAP Storage, BI Dashboard	20225509
Tran Quang Minh	Streaming, CDC Pipeline	20225512
Vo Ta Quang Nhat	Kubernetes, Batch Ingestion	20225454
Trinh Hoang Anh	Kubernetes, Infer Service	20225470

Hanoi, January 05, 2026

TABLE OF FIGURES

TABLE OF CONTENTS

Abstract

Financial markets generate large volumes of high-frequency and heterogeneous data from multiple sources, including real-time stock prices and global news events. This project presents a Big Data Storage and Processing architecture for integrating and analyzing financial data from sources such as Stooq, Finnhub, and GDELT. The proposed system supports both real-time stream processing and batch processing to enable low-latency analytics and large-scale historical analysis. A multi-layered data storage approach is employed to manage the data lifecycle from raw to processed data. The architecture is designed with a focus on scalability, performance, data quality, and fault tolerance, while also supporting data governance and reproducibility for financial analytics and forecasting.

1. PROBLEM DEFINITION

1.1. Problem

Financial market data naturally exhibits the core characteristics of Big Data when combining real-time stock prices from Finnhub with large-scale textual news data from GDELT.

- **Volume:** High-frequency stock price streams and continuously accumulated global news archives generate massive volumes of financial data.
- **Velocity:** Finnhub provides real-time streaming data, while GDELT updates unstructured text data every 15 minutes, requiring timely ingestion and processing.
- **Variety:** Structured numerical market data must be integrated with semi-structured and unstructured textual data, significantly increasing system complexity.

Despite these characteristics, existing financial data pipelines often lack a unified and scalable architecture that simultaneously addresses **data ingestion, processing, analytics, governance, and machine learning**. In particular, challenges remain in:

- **Data Ingestion & Scalability:** Ingesting heterogeneous real-time and historical data while ensuring scalability and data consistency as data volume grows.
- **Stream & Batch Processing:** Supporting parallel processing of streaming and batch data and managing the data lifecycle from raw to aggregated datasets.

- **Analytics, Machine Learning & Governance:** Enabling low-latency analytical queries, providing reliable training data for machine learning models, and maintaining metadata, data lineage, and data quality for trustworthy financial forecasting.

These gaps motivate the need for a scalable Big Data architecture that can effectively support both analytics and machine learning in the stock market domain.

1.2. Architecture Overview

To address the challenges of processing high-velocity stock market data alongside massive historical news datasets, the system implements a **Lambda Architecture**. This hybrid approach decomposes the problem into three distinct layers: Batch, Speed, and Serving, allowing the system to balance data consistency (accuracy) with low-latency availability (real-time).

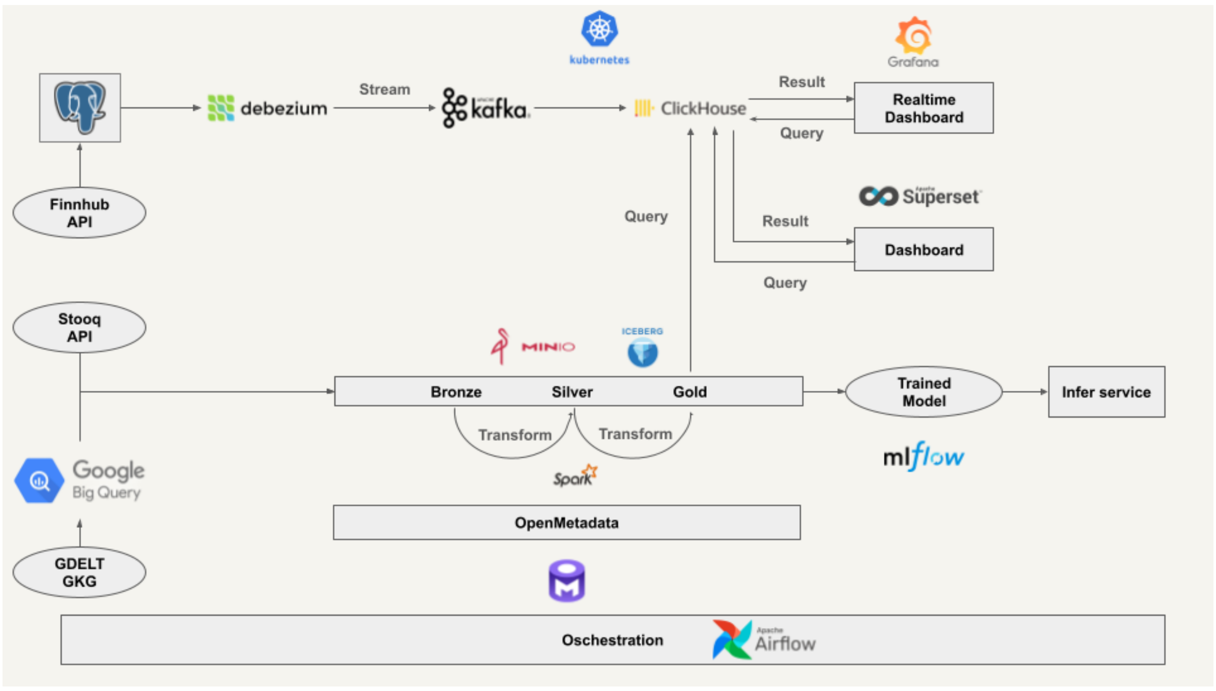

Figure 1: Lambda Architecture

The architecture is organized into four logical stages:

- **Hybrid Data Ingestion:** The system employs a bifurcated strategy to handle heterogeneous data sources. High-volume historical data (GDELT, Stooq) is ingested directly into the Bronze Layer of Lakehouse for bulk processing, while high-velocity real-time data (Finnhub) flows through a transactional buffer and Change Data Capture (CDC) pipeline to ensure immediate availability and data integrity.

- **Batch Layer (Master Dataset):** Utilizing a **Lakehouse** pattern, this layer manages the immutable master dataset stored in an object store (MinIO). It processes large-scale historical data using distributed computing (Apache Spark) to create a refined "Medallion" structure (Bronze, Silver, Gold), ensuring a comprehensive and accurate view of historical trends.
- **Speed Layer (Real-time Views):** This layer handles new data streams that have not yet been processed by the Batch Layer. It leverages a streaming backbone (Apache Kafka) to capture and process market ticks in near real-time, minimizing the latency gap between data generation and availability.
- **Unified Serving Layer:** To provide a single access point for analytics, the architecture unifies outputs from both the Batch and Speed layers into a high-performance OLAP engine (ClickHouse). This allows downstream applications, such as dashboards (Superset, Grafana) and machine learning models to predict and query both historical depth and real-time freshness seamlessly.

2. ARCHITECTURE and DESIGN

2.1. Data Ingestion

The Data Ingestion layer serves as the entry point of the architecture, responsible for reliably collecting data from heterogeneous sources and routing it to the appropriate processing pipelines. The system implements a hybrid strategy, utilizing **Apache Airflow** for orchestration to handle high-volume historical batches.

2.1.1. Data Source

The system integrates three primary financial data providers:

- **Stooq API (Historical):** Provides daily OHLCV (Open, High, Low, Close, Volume) market data. This dataset serves as the authoritative baseline for historical volatility and return calculations (Scope: NVDA, MSFT; Jan 2020 – Present).
- **GDELT Project (Contextual):** A high-volume database of global events and news. The system ingests relevant subsets of the **GDELT 2.0 Global Knowledge Graph (GKG)** to correlate market movements with global sentiment.

- **Finnhub API (Near Real-time):** Supplies high-frequency stock quotes and market news. The system polls this API at 5-minute intervals to drive the streaming analytics pipeline.

These data sources differ in terms of update frequency, data volume, and timeliness requirements, which motivates the adoption of a hybrid ingestion strategy.

2.1.2. Batch Data Ingestion

High-volume historical data (GDELT and Stooq) is processed via a batch pipeline orchestrated by **Apache Airflow**.

Direct-to-Lake Ingestion:

- Raw financial data is ingested directly into the data lake by scheduled Apache Spark jobs orchestrated by Airflow and stored in MinIO, bypassing relational databases.
- The data is first landed in the Bronze layer in its **raw form**, ensuring immutability, decoupling ingestion from processing, and enabling reliable historical reprocessing.

2.1.3. CDC-Based Real-time Ingestion

For the Finnhub data stream, the system employs a **Change Data Capture (CDC)** pattern to ensure transactional consistency and low latency.

a. **Operational Data Store (PostgreSQL)** Data fetched from Finnhub is first persisted into a **PostgreSQL** database. This database acts as an intermediate staging buffer (Operational Data Store), providing:

- **Persistence:** Acts as a short-term backup and a source of truth for the latest market states.
- **Deduplication:** Ensures only valid unique records enter the streaming pipeline.

b. **Streaming via Debezium & Kafka**

To propagate data downstream without coupling, **Debezium** monitors PostgreSQL's write-ahead logs (WAL).

- **Event Capture:** Every INSERT or UPDATE in PostgreSQL is instantly captured as an event.
- **Kafka Publishing:** These change events are serialized and published to **Apache Kafka** topics. This mechanism guarantees at-least-once delivery to

the analytics engine (ClickHouse) and visualization layers while isolating the ingestion logic from the consumption logic.

2.2. Batch Processing

Batch processing serves as the core mechanism for handling large-scale historical data and scheduled updates. This layer processes financial data from Stooq and global news events from GDELT, ensuring high throughput and fault tolerance using Apache Spark.

2.2.1. Batch Data Ingestion

Unlike traditional architectures that rely on a monolithic database for staging, this system ingests data directly from external sources into the data lake.

- **Event Data:** Extracted from the GDELT 2.0 Global Knowledge Graph via Google BigQuery, focusing on news tone, themes, and organizations. Although GDELT is updated every 15 minutes, the ingestion job is **scheduled daily by Airflow** to batch and append new event data into the Bronze layer of MinIO.
- **Market Data:** Historical stock price data (OHLCV) is fetched from the Stooq API, which updates once per day. Airflow schedules a **daily ingestion job** to load new market data directly into the Bronze layer of MinIO.

2.2.2. Medallion Architecture for Batch Processing

The batch data pipeline follows a strict **Medallion Architecture** implemented on Apache Iceberg tables and powered by Apache Spark.

a. **Bronze Layer (Raw):**
Stores immutable raw data ingested directly from **GDELT** and **Stooq**. GDELT data is stored in its original unstructured or semi-structured form, while Stooq data is stored as raw OHLCV records, both partitioned by date (and ticker for Stooq) to optimize ingestion performance.

b. **Silver Layer (Cleaned & Enriched)**
   - Uses Apache Spark for distributed data transformations.
   - **GDELT processing includes:**
     - Entity extraction from XML/HTML content to map news events to stock tickers using Regex and alias mapping.

        - Temporal normalization by converting timestamps from UTC to US Eastern Time (ET).
        - Deduplication based on composite keys (date, ticker, source URL).
    - **Stooq processing includes:**
        - Calculation of technical indicators such as **EMA, MACD, and RSI**.

c. **Gold Layer (Aggregated & Feature-Driven)**
    - Provides business-level aggregates, such as daily sentiment scores aligned with stock price movements, optimized for read-heavy queries from the visualization layer.
    - Build Machine Learning features for training model.

2.2.3. Storage and Table Format

All batch data is stored in MinIO, an S3-compatible object storage. The system adopts Apache Iceberg as the table format to provide ACID transactions, schema evolution, and time-travel capabilities directly on the data lake. Table metadata is managed using the Iceberg Hadoop Catalog, which eliminates complex file-level management and ensures consistency during concurrent read and write operations.

2.2.4. Batch Processing Outputs

The final output consists of structured Iceberg tables in the Gold layer. These tables serve as the source of truth for downstream analytics and are synchronized with **ClickHouse** to power low-latency dashboards in Apache Superset.

2.3. Stream Processing

The streaming architecture bridges the gap between periodic data ingestion and real-time analytics. It utilizes a **Change Data Capture (CDC)** pattern to propagate updates from the operational database (PostgreSQL) to the analytical warehouse (ClickHouse) with low latency.

2.3.1. Ingestion Strategy: Micro-batch to Stream

Stock market data is initially ingested via Apache Spark jobs (finnhub_to_postgres.py) running on scheduled intervals. While the ingestion is batch-oriented, the downstream propagation is event-driven. Data is written to the PostgreSQL staging table, serving as the single source of truth and the trigger for the streaming pipeline.

2.3.2. Change Data Capture (CDC) with Debezium

To decouple ingestion from analytics, the system uses **Debezium** for PostgreSQL.

- **Mechanism:** Debezium monitors the PostgreSQL Write-Ahead Log (WAL).
- **Operation:** Every INSERT operation into the staging table is captured as a change event.
- **Configuration:** The connector is configured (via sourcedb-cdc.json) to listen specifically to the finnhub_stocks table and serialize events into JSON format.

2.3.3. Message Brokering with Apache Kafka

Apache Kafka serves as the intermediate buffer to handle high-throughput event logs.

- **Topic Structure:** Debezium publishes events to the dbserver1.public.finnhub_stocks topic.
- **Retention:** Kafka ensures data durability and allows for potential replay of events in case of downstream failures.

2.3.5. Real-time Storage and Access

ClickHouse acts as the serving layer for real-time data.

- **Engine:** The destination table uses the MergeTree engine, optimized for time-series queries.
- **Visualization:** Grafana and Superset connect directly to this ClickHouse table to render live dashboards (realtime_stock_dashboard.json), enabling immediate visualization of market trends as soon as data is committed to PostgreSQL.

2.4. Serving Layer

ClickHouse serves as the central data warehouse, employing a hybrid storage strategy to balance query performance and data freshness.

- **Batch Integration (Zero-Copy & Optimized Storage):**
  - ➢ **Direct Lake Access:** The system utilizes the **Iceberg Table Engine** to create the warehouse.gold_ml_features table. This enables ClickHouse to query Parquet files stored in MinIO directly without data duplication (Zero-Copy).
  - ➢ **High-Performance Serving:** Aggregated analytical datasets are stored in warehouse.gold_stock_analytics using the **MergeTree** engine. This

ensures low-latency retrieval for complex analytical queries required by the dashboards.

- **Speed Integration:**
    - ➢ Real-time market data is ingested into the warehouse.stock_realtime table. This table is optimized for high-frequency writes and time-series aggregations, allowing the system to handle continuous data streams from the ingestion pipeline.

2.4.1. Speed Layer + Batch Layer Integration with Clickhouse

The Serving Layer provides a unified interface for downstream analytics, integrating both historical batch data and real-time streaming data into a high-performance OLAP engine.

2.4.2. Serving Real-time Metrics (Stream to Clickhouse)

To support operational monitoring, the system maintains a dedicated real-time table populated by the streaming pipeline.

- **Table:** warehouse.stock_realtime.
- **Function:** Stores raw tick-level data including timestamp, price, volume, and symbol.
- **Query Logic:** The data is queried using specialized ClickHouse aggregation functions such as argMin (for Open price) and argMax (for Close price) to dynamically generate OHLC (Open-High-Low-Close) candlesticks on 1-minute intervals.

2.4.3. Dashboards

Visualization is decoupled into two purpose-built tools targeting different analytical needs:

- **Operational Dashboard (Grafana):**
    - ➢ **Title:** Stock Stream.
    - ➢ **Focus:** Real-time market microstructure and price discovery.
    - ➢ **Key Components:** Live Candlestick charts with volume overlays, powered by sub-second queries against warehouse.stock_realtime.
- **Strategic Dashboard (Apache Superset):**
    - ▪ **Title:** Quant Trader Dashboard.
    - ▪ **Focus:** Market sentiment analysis, risk assessment, and trend validation.

- **Key Components:**
  - **Market Mood:** Tracks Sentiment Score trends for specific assets (e.g., NVDA, MSFT).
  - **Hype Cycle:** Correlates Price vs. Sentiment Sum to identify market anomalies.
  - **Risk Radar:** Monitors Volatility Tracking to assist in risk management decisions.

2.4.4. Model Training, Evaluation, and Registration with Mlflow

Training and Validation:

Model training is **orchestrated by Apache Airflow and scheduled to run monthly**, allowing the model to periodically adapt to newly accumulated market and sentiment data. The training pipeline builds a deep learning–based stock **return prediction model** using enriched features from the **Gold layer** stored in MinIO.

- **Data Preparation**: Technical indicators and daily sentiment features are loaded, scaled, and transformed into fixed-length time-series sequences using a sliding-window (look-back) approach.
- **Modeling & Optimization**: A GRU-based neural network is employed to capture temporal dependencies in financial time series. Hyperparameters (number of layers, hidden units, dropout, optimizer, learning rate) are automatically tuned using Bayesian Optimization (Keras Tuner).
- **Validation Strategy**: A **walk-forward validation** scheme is applied to simulate realistic sequential retraining and evaluation on time-series data.
- **Experiment Tracking**: All parameters, metrics, and artifacts are logged to **MLflow**, with early stopping applied to mitigate overfitting.
- **Evaluation & Registration**: Model performance is assessed using both regression metrics (MSE, RMSE, MAE, R^2) and financial metrics (direction accuracy, IC, Sharpe ratio). The best-performing model and associated preprocessing artifacts are **registered in the MLflow Model Registry** for downstream use.

Inference Service:

For online inference, the deployed service **retrieves the production-ready model directly from the MLflow Model Registry**. The selected model version is containerized and **deployed as a scalable inference service on Kubernetes**, enabling consistent and reusable predictions for downstream applications such as visualization dashboards and decision-support systems.

3. IMPLEMENTATION DETAILS

This section describes the practical implementation of the proposed system, focusing on the technologies used, deployment environment, and the realization of data ingestion, batch processing, streaming processing, and system monitoring.

3.1. Technology Stack and Deployment Environment

The system operates on a containerized infrastructure managed by Kubernetes, utilizing Helm charts for reproducible deployments across environments. The implementation integrates the following core components and specific versions:

- **Orchestration & Workflow:** Apache Airflow (v2.x) manages DAGs for batch processing and ingestion scheduling.
- **Data Lake Storage:** MinIO serves as the S3-compatible object storage for the Medallion Architecture (Bronze, Silver, Gold layers).
- **Batch Processing:** Apache Spark (v3.5.1) with pyspark is configured with IcebergSparkSessionExtensions to manage Apache Iceberg tables.
- **Streaming & CDC:** Confluent Kafka (v7.x) coupled with Debezium captures Change Data Capture (CDC) events from PostgreSQL.
- **Analytical Storage:** ClickHouse (deployed as a cluster with 2 Shards and 2 Replicas) coupled with ClickHouse Keeper provides low-latency storage for serving visualizations.
- **Metadata Management:** OpenMetadata aggregates lineage and metadata from Airflow, Spark, and database services.
- **Machine Learning:** TensorFlow (v2.15.0) and Keras Tuner are used for model training, with MLflow (v2.10.0) handling experiment tracking and artifact storage.
- **Backend & Frontend:** The application backend is Python-based (FastAPI), interacting with a React frontend.

The deployment environment is based on Docker for containerization and Kubernetes for orchestration. Kubernetes provides automated deployment, scaling, and fault tolerance for all system components. Helm charts are used to manage application configuration and enable reproducible, version-controlled deployments across different environments. This design allows the system to remain cloud-agnostic while fully leveraging cloud-native principles.

4. LESSONS LEARNED

Lesson 1: Hybrid Data Ingestion Strategies (Batch vs. Stream)

Problem Description

- **Context:** The system handles two distinct data types: high-volume historical batch data (GDELT, Stooq) and high-velocity real-time market ticks (Finnhub).
- **Challenges:** Using a "one-size-fits-all" ingestion approach created performance bottlenecks. Routing massive batch files through a relational database caused unnecessary I/O overhead, while sending raw API streams directly to the Data Lake risked data duplication and integrity issues.
- **System Impact:** High latency in batch loading and potential data inconsistencies in the streaming layer.

Approaches Tried

- **Approach 1: Unified Database Ingestion.** Attempted to route all data through PostgreSQL before archiving.
    - *Trade-off:* Severely degraded performance for batch ingestion; resource wastage on non-transactional data.
- **Approach 2: Direct-to-Lake for All.** Attempted to write API streams directly to MinIO.
    - *Trade-off:* Resulted in duplicate records and lack of immediate consistency for the operational view.

Final Solution

- **Batch Layer (GDELT/Stooq): Direct-to-Lake**. Files are fetched via Web/API and written directly to MinIO (Bronze Layer) as Parquet. This bypasses the transactional overhead entirely.
- **Speed Layer (Finnhub): Transactional Buffering**. Implemented the pipeline: API → Postgres → Debezium → Kafka → MinIO.

Reasoning: PostgreSQL acts as a transactional buffer to enforce Primary Keys (deduplication) and ensure data integrity during network instability. Debezium captures these changes (CDC) for the Data Lake.

Key Takeaways

- **"Don't put a database in the middle unless you have a transactional reason."**
- Use raw storage (Parquet/MinIO) for bulk throughput.
- Use relational databases (PostgreSQL) only when data integrity, deduplication, or immediate operational serving is required.

Lesson 2: Optimizing the Serving Layer for Low-Latency Analytics

Problem Description

- **Context:** The project required both interactive historical analysis (Superset) and sub-second real-time monitoring (Grafana).
- **Challenges:** The initial architecture failed to meet latency requirements for the presentation layer. Dashboards were sluggish, leading to poor User Experience (UX).
- **System Impact:** Dashboard timeout errors and inability to visualize real-time market movements.

Approaches Tried

- **Approach 1: SparkSQL (Direct Query).** Connected Superset directly to MinIO via Spark.
    - *Trade-off:* High latency. Spark is designed for throughput, not interactive queries. Spinning up executors for every dashboard filter caused multi-second delays.
- **Approach 2: PostgreSQL (Serving Layer).** Loaded aggregated data into Postgres.
    - *Trade-off:* Poor scalability for analytics. Being row-oriented, Postgres struggled to aggregate millions of records (e.g., SUM(price)) efficiently compared to column-oriented stores.

Final Solution

- **ClickHouse as the Unified Serving Layer.**
    - **For Batch (Superset):** Aggregated "Gold" data is ingested into ClickHouse. Its column-based architecture allows for sub-second query responses on large historical datasets using SIMD vectorization.

- **For Real-Time (Grafana):** utilized ClickHouse's Kafka Table Engine to ingest streams directly (consuming millions of rows/sec). Grafana queries this directly for live dashboards.

Key Takeaways

- **OLTP vs. OLAP:** Never use a transactional DB (Postgres) for heavy analytical workloads; use a columnar store (ClickHouse).

Lesson 3: Contract-First Development for Parallel Execution

Problem Description

- **Context:** The Data Engineering (DE) team and the Data Visualization (Viz) team needed to work simultaneously to meet the Capstone deadline.
- **Challenges:** The visualization team was blocked because the data pipeline was not yet fully operational. Waiting for the full backend implementation effectively forced a "Waterfall" process in an Agile timeline.
- **System Impact:** Delayed feedback loops on dashboard design and potential integration hell at the end of the project.

Approaches Tried

- **Approach 1: Sequential Development.** Waited for the DE team to finish the Bronze/Silver/Gold layers before starting dashboard work.
  - *Trade-off:* Inefficient time management; risk of missing the final presentation deadline.
- **Approach 2: Ad-hoc Mocking.** Used random hardcoded values in the visualization tools.
  - *Trade-off:* Integration failed when real data arrived because data types and formats did not match.

Final Solution

- **Schema-Based Contract Development.**
  - The teams agreed on a strict CSV schema (Contract) defining column names, data types, and expected values upfront.
  - The Viz team built dashboards using these "Contract CSVs" (Mock Data) while the DE team built pipelines to produce that exact output.
  - **Integration:** Swapping the mock source for the real ClickHouse/MinIO source required zero changes to the dashboard logic.

Key Takeaways

- Decouple dependencies between backend and frontend teams early.
- **Contract-First:** Defining the output schema *before* writing the processing logic allows parallel development and reduces friction during the integration phase.

Lesson 4: Multi-arch Build and Docker ARGs

Problem Description

- **Context:** The development team utilizes mixed hardware architectures (e.g., Apple Silicon ARM64 and Intel/AMD64).
- **Challenges:** The current Dockerfile configurations (e.g., Backend/Dockerfile, airflow/Dockerfile) rely on hardcoded base images and do not dynamically handle platform-specific dependencies. This leads to build failures or significant performance degradation (via emulation) when developers build images on non-native architectures.
- **System Impact:** Inconsistent development environments and slower CI/CD pipelines due to lack of native architecture support.

Approaches Tried

- **Approach 1: Single Architecture Builds.** We built images solely for linux/amd64.
  - *Trade-off:* Developers on ARM chips (M1/M2) faced slow emulation or crash loops.
- **Approach 2 (Proposed Future Solution): Docker Build arguments (ARG).** Utilizing ARG instructions (e.g., ARG BUILDPLATFORM, ARG TARGETARCH) within the Dockerfile to conditionalize installation steps.

Final Solution

- **Proposed Implementation:** Refactor all Dockerfiles to use **Multi-arch builds** with docker buildx.

Key Takeaways

- **Portability:** Always decouple the build process from the host architecture.
- **Maintainability:** Use ARG to centralize version control (Python, Spark, Java versions) at the top of the Dockerfile rather than burying them in RUN commands.

Lesson 5: Why Helm Charts Are Preferable to Manual kubectl apply

Problem Description

- **Context:** The project requires deploying over 27 distinct Kubernetes resources, including ConfigMaps, Secrets, StatefulSets (Postgres, ClickHouse), and Deployments (Airflow, Spark).
- **Challenges:** Initially, deployment relied on applying raw YAML files using kubectl apply -f <file>. As configurations evolved, managing dependencies (e.g., Postgres must be ready before Airflow) and environment variables became error-prone.
- **System Impact:** High operational risk during updates; lack of versioning for the deployed infrastructure.

Approaches Tried

- **Approach 1: Scripted Manual Application.** We utilized a shell script k8s/deploy_final.sh to apply YAMLs in a specific order and loop sleep commands to wait for pods.
  - *Trade-off:* This was brittle. If a pod crashed or took too long, the script would fail or timeout blindly. It lacked state management (no easy rollback).
- **Approach 2: Helm Charts.** We transitioned to packaging the entire pipeline into a custom Helm chart located in bdsp-pipeline.

Final Solution

- **Detailed Solution:** Created a unified Helm chart (bdsp-pipeline) that templated all Kubernetes resources.
- **Implementation Details:**
  - **Centralized Config:** All variables (images, ports, credentials) are moved to values.yaml, replacing scattered hardcoded values in YAML files.
  - **Release Management:** Deployment is executed via helm-deploy.sh, allowing atomic upgrades.
  - **Conditional Logic:** Enabled features (like postgres.enabled) can be toggled via flags in values.yaml without deleting files.
- **Metrics:** Deployment complexity reduced from executing a 300-line Bash script to a single command: helm upgrade --install bdsp ./bdsp-pipeline.

Key Takeaways

- **Infrastructure as Code (IaC):** Helm treats infrastructure configuration as versioned code.