

PROJECT REPORT: REAL-TIME FINANCE

FRAUD DETECTION SYSTEM

Module: Data Engineering 2 – Big Data Architecture

Date: 13 February 2026

Team members : Tania Rose Jobi, Vishnu Prem Nair, Sidharth Arakkan, Gaurangi Tyagi

1. INTRODUCTION

The financial sector processes millions of transactions every second. Identifying fraudulent activity manually is impossible due to the sheer volume and speed of data. This project implements a **Hybrid Lambda Architecture** that combines **Batch Processing** (historical analysis) and **Stream Processing** (real-time detection) to identify suspicious financial transactions instantly.

2. SYSTEM ARCHITECTURE

The system is built on a containerized infrastructure using **Docker** and integrates with **Google Cloud Platform (GCP)** for enterprise-grade scalability.

2.1 Core Components:

- **Infrastructure:** Orchestrated using `docker-compose.yml`, running Apache Airflow, Kafka, and Postgres.
- **Batch Layer:** Handles large-scale historical data cleaning, aggregation, and model training using **Apache Airflow**.
- **Streaming Layer:** Uses **Apache Kafka** to ingest live transaction ticks and a Python consumer to detect anomalies in real-time.
- **Serving Layer:** A **Flask Web App** that provides an API for instant fraud prediction using the trained Machine Learning model.

3. DATA GENERATION & INGESTION

Since real financial data is sensitive, we developed a custom generator (`generate_orders.py`) to simulate market activity.

- **Scope:** Generated 6+ hours of data with 1,000+ records per hour.
- **Quality Challenges:** Injected synthetic anomalies like missing values, duplicates, and massive price spikes to simulate fraudulent patterns.
- **GCP Integration:** Historical data is uploaded to **Google Cloud Storage (GCS)** using `gcs_upload.py` for centralized cloud management.

4. THE BATCH PIPELINE (THE “BRAIN”)

The batch pipeline is managed by an Airflow DAG (`de2_batch_pipeline`) which ensures a structured data flow:

1. **Clean & Validate (`clean_validate.py`):** Standardizes time formats (UTC), removes duplicates, and handles null values.
2. **Aggregation (`aggregate.py`):** Summarizes raw ticks into hourly Open-High-Low-Close (OHLC) metrics and total volume.
3. **Feature Engineering (`features.py`):** Calculates critical signals like **3-hour Moving Averages** and **Price Volatility** .
4. **Model Training (`train_model.py`):** Trains a **Random Forest Classifier** with 300 estimators to recognize fraud indicators.

5. REAL-TIME STREAMING & DETECTION

The speed layer ensures no fraudulent transaction goes unnoticed:

- **Kafka Producer:** Streams transaction data into the system, simulating a live financial exchange.
- **Stream Consumer:** Monitors the Kafka topic in 30-second micro-batches.
- **Anomalous Detection:** Uses a configured `spike_threshold_pct` to trigger alerts if a price moves too rapidly within a short window.

6. USER INTERFACE & API

The project includes a production-ready API built with **Flask** (`app.py`):

- **Predict Endpoint:** Receives transaction details in JSON format.
- **Logic:** Loads the saved `rf_model.joblib`, validates the incoming features, and returns a fraud probability score.

7. CONCLUSION

This project successfully demonstrates an end-to-end Big Data pipeline. By combining the stability of batch processing with the speed of real-time streaming, we created a robust system capable of protecting financial transactions. The use of **Docker** and **GCP** ensures that this solution can scale from a local prototype to a global deployment.

8. PERFORMANCE ANALYSIS (LITTLE'S LAW)

To validate the stability of our streaming architecture, we applied Little's Law ($L = \lambda \times W$) to our Kafka consumer. This calculation ensures that our system can handle the ingestion rate without creating a backlog or significant lag.

- Arrival Rate (λ): Our generator produces 1,200 records per hour.
 - $\lambda = 1,200 \text{ records} / 3,600 \text{ seconds} = 0.33 \text{ records per second}$
- Average Processing Time (W): Based on our *stream_consumer.py* logs, the time to process a single micro-batch is approximately 1.5 seconds .
- Average Items in System (L):
 - $L = 0.33 \text{ (Arrival Rate)} \times 1.5 \text{ (Processing Time)}$
 - $L = 0.495 \text{ units}$

Result: Since the value of L is less than 1 , the system is mathematically stable. This proves that the consumer is processing data faster than it is arriving, ensuring zero lag in the fraud detection pipeline and maintaining real-time performance.

Appendix: Technical Stack

- **Languages:** Python 3.11+
- **Libraries:** Pandas, Scikit-learn, Kafka-python, Flask, Joblib
- **Tools:** Apache Airflow, Apache Kafka, Docker, Google Cloud Storage