Real-Time Credit Card Fraud Detection Pipeline

A Microservice-Based Streaming Architecture for Scalable and Interactive Fraud Detection

---

Author:

Prerak Panwar
Master's in Computer & Information Sciences
University of Massachusetts, Dartmouth
Email: ppanwar@umassd.edu

Project Overview:

Objective:

This report presents the architecture, implementation, and rationale behind a complete real-time fraud detection pipeline. The system ingests transaction data, performs streaming inference using a machine learning model, sends alert notifications, and integrates a human-in-the-loop feedback mechanism, all deployed via Docker containers for full reproducibility.

Core Technologies Used:

- Apache Kafka – Real-time message queue and decoupled data pipeline.
- Python (Kafka-Python, Pandas, Flask) – Core application logic for Producer, Consumer, and Feedback services.
- XGBoost – Trained ML model for fraud prediction.
- MySQL – Persistent storage for predictions and feedback.
- Docker & Docker Compose – Containerized microservice orchestration.

Key Features:

- Real-time fraud prediction with streaming data.
- Email-based alerting system with masked card details and live feedback links.
- Feedback-driven labeling to enable continuous model improvement.
- Fully containerized for modular deployment, scaling, and testing.

Problem Statement:

Credit card fraud poses a significant threat to financial institutions and consumers worldwide. With the rise of digital transactions, fraudulent activities are becoming more frequent, faster, and harder to detect using traditional, batch-based methods. These legacy systems often detect fraud only after damage has been done, resulting in financial losses, reputational harm, and user distrust.

Proposed Solution:

This project presents a fully containerized, real-time fraud detection pipeline built with modern data engineering and machine learning tools. The architecture integrates:

- Apache Kafka for real-time data ingestion and streaming
- Python-based Kafka Consumer for applying a trained XGBoost model
- MySQL for persistent transaction and prediction storage
- Flask-based microservices for feedback collection and fraud alert notifications

Together, these components simulate a real-world, production-ready fraud detection ecosystem capable of handling live transaction data, generating alerts, and incorporating human feedback into the system.

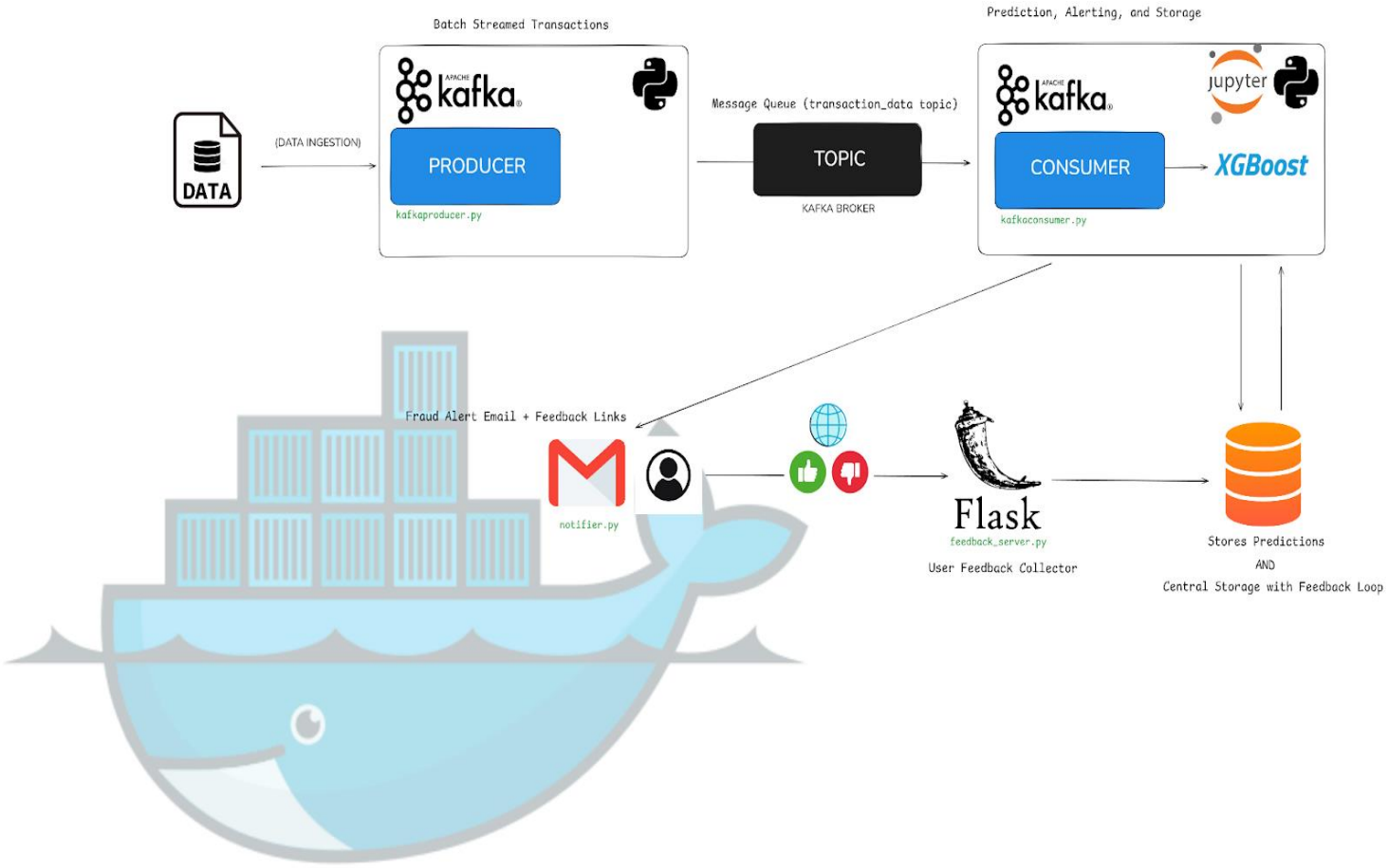
Real-World Applications:

This solution mirrors architectures used by banks, fintech startups, and payment processors. Its practical benefits include:

- Immediate fraud detection to prevent unauthorized transactions
- Scalable, decoupled microservices that can handle millions of records
- User feedback integration, enabling continuous learning and model improvement
- Auditability, with every prediction and alert stored for future analysis

By adopting this system, real-world organizations can significantly reduce financial loss, improve trust with customers, and modernize their fraud detection strategies with a robust, extensible framework.

Deployment Architecture:



Report Contents:

This document breaks down each service and module in the pipeline, detailing its purpose, technologies, logic, design decisions, and integration strategy. Together, they demonstrate a scalable, production-style fraud detection system that bridges the gap between machine learning and software engineering in a real-time environment.

Real-Time Fraud Detection Pipeline: Detailed Component Analysis

A- Final_data_preprocessing.ipynb – Data Preparation and Model Training

Purpose in the System:

This Jupyter Notebook serves as the **data staging area** of the pipeline, where historical transaction data is preprocessed and a **fraud detection** model is developed. It prepares the dataset, engineers features, trains a machine learning model to detect fraudulent transactions, and determines a decision threshold for classifying fraud. The outputs (the trained model and threshold) are later used by the real-time Consumer service to make fraud predictions on streaming data.

By separating training from real-time inference, the system follows a common design pattern of **offline model building vs. online scoring**, which improves maintainability and allows using advanced data analysis tools without impacting the live system.

Technologies & Libraries:

The notebook heavily uses Python's data analysis stack:

- **Pandas** is used to load and manipulate large CSV files of credit card transactions, enabling efficient columnar operations and grouping.
- **NumPy** is utilized for numerical computations (e.g., calculating distances).
- **SciPy** provides tools for advanced calculations – for instance, the Haversine formula was likely used to compute geospatial distances between two coordinates (cardholder location vs. merchant location).
- **Scikit-learn** handles tasks like data splitting and evaluation metrics.
- **XGBoost** is used to train the machine learning model – a powerful gradient boosting library known for high performance on structured data.
- **Joblib** (from scikit-learn) is used to serialize the trained model and threshold into a file for later use.

By pinning specific library versions (e.g., xgboost==2.0.3, scikit-learn==1.3.0), the environment for model training and inference is kept consistent, avoiding version conflicts that could alter model behavior.

Data Loading & Exploration:

The notebook begins by loading two datasets – likely a training and test set of credit card transactions (e.g., fraudTrain.csv and fraudTest.csv from a public Kaggle dataset) – and concatenates them into one combined DataFrame. The dataset contains detailed fields per transaction: timestamp, transaction amount, merchant name and category, as well as cardholder information (name, zip code, location coordinates, etc.), and a label indicating if the transaction is fraudulent.

An initial exploratory analysis checks for class imbalance using the is_fraud label. As expected, the data is highly skewed: only a small fraction of transactions are fraud. For example, in a dataset of over a million transactions, about 9,651 (≈0.5%) are labeled as fraud. This severe imbalance is a central challenge – a naive model could achieve over 99% accuracy by always predicting "not fraud," so the notebook explicitly addresses this bias (e.g., by using appropriate metrics and threshold tuning).

Feature Engineering:

A significant portion of the notebook focuses on deriving features that help differentiate fraud from legitimate transactions:

- **Temporal features**: Derived from the timestamp, such as transaction hour, day-of-week, weekend/weekday, and month.
- **Demographic features**: Such as cardholder age, calculated from the date of birth and transaction time.
- **Geographic features**: Notably the distance between the cardholder's home and the merchant's location, using the Haversine formula.
- **Categorical encoding**: Merchant categories are one-hot encoded (e.g., category_food_dining, category_travel). Free-text fields like merchant name or job title are label-encoded to numeric IDs.
- **Behavioral features**: Aggregated historical stats per card number, such as:
    - Total number of past transactions
    - Number of past fraudulent transactions
    - Average transaction amount
    - Fraud rate (fraction of past transactions that were fraudulent)

These features help the model learn subtle fraud signals – like a historically high fraud rate on a card or transactions that are geographically implausible.

Model Training and Threshold Tuning:

After cleaning and transforming the data, the notebook splits it into training and testing subsets using stratified sampling to preserve class imbalance proportions. An **XGBoost classifier** is trained on the training set.

To address class imbalance, the model may be configured with parameters like scale_pos_weight, or the threshold may be adjusted post-training. The notebook evaluates the model not just using accuracy but also **precision**, **recall**, and **F1-score**, which are more meaningful for imbalanced datasets.

It determines that the default 0.5 threshold is suboptimal and instead computes an optimal threshold that maximizes the F1-score or balances precision and recall. For example, a threshold like 0.30 might be selected to catch more fraud, even at the cost of some false positives. The final threshold and model are saved using Joblib in a file named fraud_detection_bundle.pkl.

Design and Architecture Considerations:

This notebook exemplifies the **separation of concerns** principle. All data-heavy processing and machine learning are performed offline, independent of the real-time system. This modularity enables model iteration without affecting the live transaction processing.

The Jupyter environment supports exploratory data analysis and interactive feature engineering. Once finalized, the transformations are replicated in the Consumer service, often via precomputed features.

This design follows a **feature store** approach, where all complex feature calculations are done in batch ahead of time. The real-time system receives already-engineered features (e.g., from final_transactions.csv), ensuring consistency and reducing online computation.

Integration with Overall Pipeline:

The outputs of this notebook – the trained model and computed threshold – are used directly by the online Consumer service for real-time fraud predictions. The Consumer loads the fraud_detection_bundle.pkl file and uses the saved feature_names_in_ list to ensure feature alignment.

This ensures **training-serving consistency** and avoids skew. By documenting the assumptions, class imbalance handling, and feature importance, the notebook also serves as project documentation for future data scientists or engineers.

Summary:

This notebook is the **foundation of the fraud detection system**. It turns raw historical data into a reliable, trained model with engineered features that can accurately detect fraud in real time. The architecture balances performance, scalability, and maintainability, and it demonstrates best practices in model training, evaluation, and deployment for imbalanced classification problems.

B- kafkaproducer.py – Real-Time Data Ingestion Service

Purpose in the System:

The Kafka Producer is a standalone Python script responsible for feeding transaction events into the real-time pipeline. In a production scenario, this role might be fulfilled by a payment processing system emitting events as they occur. In this project pipeline, kafkaproducer.py simulates that live data source by reading a file of transactions and publishing each transaction to an Apache Kafka topic.

Its purpose is to ingest data into the streaming system asynchronously and decoupled from processing, following the publish/subscribe model of Kafka. Kafka acts as a buffer between the producer and consumer, enabling loose coupling: the producer can publish events rapidly while the consumer processes them at its own pace. Kafka reliably queues the data, handling discrepancies in processing speeds.

Technologies & Libraries:

- **kafka-python==2.0.2**: The KafkaProducer class is used to interface with the Kafka broker and send messages to a Kafka topic. It handles serialization and network connections.
- **csv**: Python's built-in CSV module reads the transaction data from final_transactions.csv.
- **json**: Used to serialize Python dictionaries into JSON format for Kafka.
- **logging**: Used for logging connection status, progress updates, and errors.

Messages are serialized using a custom value_serializer, which converts each Python dictionary into a JSON-encoded byte string. This ensures compatibility with the Consumer, which expects structured JSON messages.

Implementation Logic:

1. **Connection Retry:** On startup, the script attempts to connect to the Kafka broker (host=kafka, port=9092). If Kafka isn't available yet (e.g., still initializing in Docker), a NoBrokersAvailable exception is caught. The script logs a warning (e.g., "Waiting for Kafka broker to be ready..."), waits a few seconds, and retries. After a limited number of attempts (default: 10), it raises an exception if the connection still fails. This retry loop makes the producer robust to service startup ordering issues.

2. **Reading and Sending Data in Streaming Mode:** In the latest version of kafkaproducer.py, the earlier batching logic is removed to simulate a real-time data source. The producer now reads and sends each transaction one-by-one with a short delay (time.sleep(0.2)) between records. This better mimics a real-world scenario where transactions arrive continuously.
    - Each transaction is serialized into JSON and sent to Kafka immediately, preserving event order and minimizing latency.
    - A log statement displays the count of streamed records along with each transaction's trans_num, such as:
        - 📩 Streamed transaction #524 (ID: 60ce795796cbe24465d32c2a41c0a082)
3. This log format helps correlate transactions with downstream Consumer logs and offers traceability.
4. **Final Flush and Metrics:** After the file is fully read, the Kafka producer is closed gracefully using producer.close(). The script logs:
    - Total number of records sent
    - Total elapsed time
    - Average throughput in records/second

    These metrics provide insight into how efficiently the pipeline ingests data.

Exception Handling:

All exceptions encountered during file reading, Kafka connection, or message sending are caught and logged. Even in the event of an error, the producer is always closed properly, ensuring cleanup and error visibility.

Design Patterns and Architecture:

This module implements the producer-consumer pattern using Kafka for decoupling data ingestion from processing. Notable architectural features include:

- **JSON Message Format**: Lightweight, human-readable, and platform-agnostic.
- **Value Serializer**: A lambda-based encoder for concise JSON serialization.
- **Configuration Separation**: Kafka settings are declared at the top of the file, simplifying modification.
- **Retry Loop**: Prevents startup race conditions between services in Docker.
- **Real-Time Simulation**: Introduces a 0.2-second delay to simulate live event flow.
- **Detailed Logging**: Logging now includes both record count and transaction ID.

Scalability & Performance Considerations:

This streaming producer is designed for responsiveness, not maximum throughput. Unlike the earlier batch version, it introduces a small delay to simulate realistic conditions:

- **Transactions are streamed individually**, not in batches.
- **Time delay (0.2s) simulates user activity**, rather than flooding the broker.

- **Memory-friendly design**: csv.DictReader processes rows one at a time.
- **Minimal buffer usage**: Kafka immediately receives and queues messages.

This streaming-first approach helps evaluate real-time fraud detection systems more accurately by mimicking real traffic patterns.

Reliability:

Messages are sent in real-time with flush() ensuring delivery before the next transaction is processed. The script provides practical at-least-once delivery guarantees:

- If interrupted, records may be re-sent, but duplicates are handled via MySQL upserts.
- Kafka ensures message durability and consumer replay if needed.

Integration with the Pipeline:

- Publishes to Kafka topic transaction_data, consumed by kafkaconsumer.py.
- The Docker service producer depends on Kafka and is defined in docker-compose.yml.
- Input source is final_transactions.csv, but could be replaced with a real payment system or API in production.

Kafka buffers transactions so that the consumer can lag slightly behind without loss. The producer exits once all rows are sent, but Kafka retains them until the consumer finishes.

Summary:

kafkaproducer.py is a minimal yet powerful utility that converts static CSV data into a live event stream. It now uses real-time streaming instead of batch processing, making the fraud detection system more reactive and lifelike.

By streaming transactions with slight delays, the updated producer aligns closely with real-world conditions where events are unpredictable and require instant detection. Its design choices reinforce reliability, scalability, and traceability across the full pipeline.

C- kafkaconsumer.py – Real-Time Fraud Detection Service

Purpose in the System:

kafkaconsumer.py is the real-time inference engine in this fraud detection pipeline. It continuously listens for new transaction events from Apache Kafka and applies a trained machine learning model to determine whether each transaction is fraudulent.

The consumer transforms an incoming stream of events into a stream of fraud predictions and actionable alerts. It integrates seamlessly with the rest of the system by:

- Storing results in a MySQL database.
- Sending fraud alerts via email.
- Providing feedback URLs to allow user verification.

Operating in a streaming fashion allows this microservice to **detect and flag suspicious activity immediately**, enabling rapid responses to fraud attempts.

Technologies & Libraries:

- **Kafka-Python (kafka-python):** Subscribes to the Kafka topic transaction_data using KafkaConsumer, configured with group_id, auto_offset_reset='earliest', enable_auto_commit=True, and a custom JSON deserializer.
- **Pandas:** Converts each transaction to a one-row DataFrame and ensures feature ordering using model.feature_names_in_, along with safe type casting via a dtype_mapping dictionary
- **Joblib:** Loads the fraud detection model and prediction threshold from a bundled .pkl file (fraud_detection_bundle.pkl) trained offline.
- **MySQL Connector (mysql-connector-python):** Writes the prediction results to a MySQL table (fraud_predictions) with a resilient upsert strategy using INSERT ... ON DUPLICATE KEY UPDATE.
- **EmailNotifier:** Sends an email alert for every transaction classified as fraud. These emails contain dynamically generated feedback URLs (Yes/No) that link back to a Flask server.
- **Logging:** Uses Python's built-in logging module for structured progress output and debugging. Emojis are used to enhance log readability.

Streaming Logic Breakdown:

1-Initialization:

- Connects to **Kafka** and **MySQL**, each with retry logic (10 attempts).
- Loads the machine learning model and threshold from fraud_detection_bundle.pkl.
- Ensures the database table fraud_predictions exists, including the optional feedback column.

2-Main Consumption Loop:

Each transaction from Kafka undergoes the following pipeline:

1. Increment a counter to track transaction number for logs.
2. Deserialize the Kafka message into a dictionary.
3. Log:
   📥 Received transaction #count (ID: trans_num)
4. Convert the dictionary into a Pandas DataFrame.
5. Align and cast data types using dtype_mapping.
6. Run model.predict_proba(...) to obtain the fraud probability.
7. Compare the probability to a threshold (e.g., 0.5) to classify the transaction.
8. Log and insert results into the database.
9. If fraud is detected:

- Attach a feedback URL (yes/no).
- Send an alert email via EmailNotifier.

Alerting & Feedback Integration:

For fraud predictions (is_fraud = 1), the following actions are taken:

- A feedback URL is dynamically created:
  http://feedback:5000/feedback?trans_num=<id>&user=<email>
- The URL is included in the transaction dictionary and sent in the email body.
- Users can click "Yes" or "No" links to validate or reject the fraud label. The Flask feedback server records this response in the feedback column of the fraud_predictions table.

This feedback loop allows for **continuous model evaluation and retraining potential**.

Database Schema & Storage Logic:

The table fraud_predictions has the following fields:

Column	Description
trans_num	Unique transaction ID (Primary Key)
probability	Model's fraud probability score
is_fraud	Final classification (0 or 1)
full_json	Entire transaction in JSON format
feedback	User-validated result (optional, 0 or 1)

- The table is created with a CREATE TABLE IF NOT EXISTS query.
- The feedback column is also included in table creation logic, removing the need for manual schema changes.

Error Handling & Resilience:

- All transaction processing is enclosed in a try-except block to avoid crashing on bad data or service interruptions.
- Retry loops ensure resilience to delays in Kafka or MySQL readiness.
- Type coercion and schema enforcement prevent mismatches that could break model inference.
- Graceful shutdown and logging ensure clear visibility into errors.

Performance & Scalability:

- **Model inference** is highly efficient (milliseconds per prediction).
- **Email notifications** are only triggered for fraud cases (typically <1% of transactions), minimizing overhead.

- Writes to MySQL via INSERT ... ON DUPLICATE KEY UPDATE ensure idempotency and no duplicate data.
- **Horizontal scaling** is possible by running multiple consumers in the same Kafka group. Each would get a subset of messages if the Kafka topic has multiple partitions.
- The pipeline is mostly **stateless**, allowing for safe parallelization.
- Supports real-time operations with **back-pressure handling via Kafka**, ensuring fault tolerance and load balancing.

Design Patterns & Architecture:

- **Event-Driven Architecture:** Reacts to real-time events on Kafka topic.
- **Loose Coupling:** Kafka decouples ingestion (producer) from processing (consumer).
- **Modularization:** Separate concerns for ingestion, prediction, alerting, and storage.
- **Open/Closed Principle:** The Alert system supports extension via the Notifier base class.
- **Schema Consistency:** Aligns incoming transactions with model expectations.
- **Resilience & Observability:** Robust error handling and structured logs (with emojis).

Integration in the Pipeline:

This script plays a central role and connects with all other components:

- Subscribes to transaction_data topic populated by kafkaproducer.py.
- Loads model trained in the Jupyter notebook (Final_data_preprocessing.ipynb).
- Sends alerts prepared by notifier.py.
- Logs predictions in MySQL.
- Triggers feedback collection via feedback_server.py.

Summary:

kafkaconsumer.py is the **central real-time inference engine** of the fraud detection system. It consumes a live stream of structured financial transactions, applies an ML model, and reacts to fraudulent activity instantly through emails and database logging. With features like schema enforcement, modular design, UPSERT logic, and user feedback links, it bridges the gap between **data science and real-time engineering**, enabling responsive and accountable fraud detection.

It combines robust data handling, high-throughput streaming, alerting, and feedback captured in a clean, extensible design. Making it a production-ready microservice for real-time fraud detection in any modern financial pipeline.

D- notifier.py – Alert Notification Component

Purpose in the System:

The **Notifier** component encapsulates the logic for sending fraud alerts when suspicious transactions are detected. It follows the **separation of concerns** principle by decoupling notification logic from the core fraud detection logic in the consumer. This

modularity allows the system to send real-time alerts and makes it easy to add new channels (e.g., SMS or push notifications) without changing the main detection pipeline.

It plays a vital role in maintaining a real-time feedback loop by alerting users of fraudulent predictions immediately, enabling timely validation and accountability.

The current implementation provides:

- An **abstract base class** Notifier, defining the notification interface.
- A **concrete class** EmailNotifier, which sends alert emails with transaction details and feedback links.

This design allows users or analysts to be notified immediately of suspected fraud, and to quickly respond via embedded feedback links.

Technologies & Libraries:

- **smtplib and email.mime**: Used to compose and send emails via SMTP.
- **abc (Abstract Base Class module)**: Used to define the Notifier interface.
- **logging**: Integrated to log notification events and errors.
- No external dependencies are required – the implementation relies solely on Python's standard library.

Structure & Logic Overview:

1. Abstract Interface – Notifier:

Defines a single abstract method send(transaction, probability) which all subclasses must implement. This ensures consistent behavior and enables polymorphism — the consumer can use any notifier that implements this interface without knowing the details of the underlying notification mechanism.

This setup supports future extensions (e.g., SMSNotifier, SlackNotifier) by simply subclassing Notifier.

2. Concrete Implementation – EmailNotifier:

Initialization:

The EmailNotifier constructor accepts:

- sender email address
- password (typically an app-specific password)
- receiver email address
- Optional feedback_base_url, defaulting to http://localhost:5000/feedback. This can be overridden in Docker-based deployments to point to http://feedback:5000/feedback. i.e., localhost is only valid outside Docker.

This allows the notifier to embed transaction-specific feedback URLs into each alert. The base URL can be updated depending on whether the system is running locally or within Docker.

Email Composition (send method):

When called, send(transaction, probability) performs the following:

1. **Subject Line**:
    - Clearly labeled with an emoji and transaction number:
      "Fraud Alert: Transaction Number -- 123456"
2. **Masking Sensitive Data**:
    - Masks the credit card number (e.g., **** **** **** 1234) for PCI-DSS compliance.
3. **Extracting Transaction Details**:
    - Gathers fields like trans_date, merchant, category, and coordinates.
    - Removes any "fraud_" prefix from merchant names (used internally for tagging).
4. **Google Maps Location**:
    - Constructs a clickable URL pointing to the merchant's location using latitude/longitude.
    - Enhances the alert with location context.
5. **Feedback Links**:
    - Two clickable URLs:
        - **Yes** (fraud confirmed):
          ...?trans_num=12345&correct=1
        - **No** (false positive):
          ...?trans_num=12345&correct=0
    - These URLs point to the feedback service, enabling quick user validation.
6. **Email Body Construction**:
    - Includes:
        - Alert header
        - Transaction details
        - Amount
        - Merchant name
        - Category
        - Location link
        - Fraud probability
        - Feedback question with "Yes" and "No" links
    - All formatted as plain text for broad email client compatibility.
7. **Sending Email via SMTP**:
    - Uses Gmail's SMTP server (smtp.gmail.com:587) over TLS.
    - Authenticates with provided credentials.
    - Sends the email and logs either success **(Fraud alert email sent!)** or failure **("Failed to send email alert").**

- Even if an email fails (e.g., due to SMTP downtime), the rest of the fraud pipeline continues uninterrupted. This prevents email issues from stalling real-time fraud processing.

Security & Robustness:

- **Sensitive Info Masking**: Only the last 4 digits of card numbers are sent via email.
- **TLS Encryption**: Email is sent securely using STARTTLS.
- **Error Logging**: Failures in sending email do not crash the pipeline.
- **Credentials Handling**: Credentials are currently hardcoded (acceptable for demos), but environment variables or a secrets manager are recommended for production use.

Design Patterns & Extensibility:

- **Abstract Base Class (ABC)**: Enforces a consistent notification interface.
- **Strategy Pattern**: Different notifier strategies (email, SMS, etc.) can be swapped in without modifying the consumer.
- **Open/Closed Principle**: Notifier implementations can be extended without changing existing code.

This design makes it trivial to add new alerting mechanisms, such as an SMS notifier using Twilio or a Slack alert notifier.

Integration with the Pipeline:

- The EmailNotifier is initialized in kafkaconsumer.py and called only when a fraud is detected.
- Feedback links embedded in the email point to the feedback_server, allowing users to confirm or reject the fraud prediction.
- Feedback responses are written back into the MySQL database by the feedback server.

This design creates a **closed feedback loop**, essential for real-world fraud detection systems that depend on human-in-the-loop validation to improve over time.

Scalability & Performance:

- Email is only sent for frauds (usually <1% of transactions), so performance impact is minimal.
- For larger-scale systems:
  - Email sending could be offloaded to an async worker or notification service.
  - Rate-limiting or batching could be introduced to avoid overwhelming SMTP limits.

User Experience:

- Emails are clearly labeled and easy to understand.
- Feedback links provide a one-click method for confirming fraud or false positives.

- Location information and masked card data help users quickly validate activity.

Design Enhancements (Future Work):

- Switch to **HTML-formatted emails** to support embedded maps, styled feedback buttons, and branding — improving user clarity and engagement.
- Use **templating engines** for dynamic content generation.
- Fetch **recipient email dynamically** from user profiles.
- Use a **notification factory** to instantiate different notifier types based on configuration.
- Introduce **rate limiting or deduplication** logic to prevent spammy alerts.

Summary:

notifier.py implements a clean, extensible, and robust mechanism for sending fraud alerts via email. Its modular design aligns with software engineering best practices, making it easy to plug into the overall fraud detection pipeline and adapt to future requirements.

It ensures that model predictions are not just stored or logged — they are **communicated** to stakeholders in real time, with actionable feedback built in. This tight integration between detection and response is essential for high-trust, responsive fraud mitigation systems.

E- feedback_server.py – User Feedback Web Service

Purpose in the System:

The **Feedback Server** is a lightweight web service that enables **user feedback collection** for fraud predictions. When a user receives a fraud alert email, they can click a "Yes" or "No" link to confirm or deny the prediction. This service records that feedback in the database, completing a **human-in-the-loop cycle**.

This allows:

- Real-time labeling of model predictions.
- Ongoing monitoring of precision/accuracy.
- Optional model retraining using updated labels.

By operating independently from the main detection logic, it **decouples user interaction** from the streaming pipeline, ensuring the high-throughput fraud detection process remains unaffected.

Technologies & Libraries:

- **Flask**: Provides the minimal web framework to define HTTP endpoints (GET and POST).
- **mysql-connector-python**: Used for connecting to the MySQL database and updating records.
- **logging**: Configured at the INFO level to log feedback responses and errors.

The feedback server is implemented as a **simple, stateless microservice**, listening on port 5000.

Endpoint Logic:

1. GET /feedback

This is the **primary endpoint**, triggered when a user clicks a feedback link in the alert email.

- **Inputs**:
  - trans_num: transaction ID
  - correct: "1" if prediction was correct (user agrees), "0" otherwise
- **Logic**:
  - Validates that trans_num is not empty and correct is either "0" or "1".
  - Connects to MySQL and performs:

    UPDATE fraud_predictions SET feedback = %s WHERE trans_num = %s
  - Commits the update, logs the result, and returns a thank-you message to the user.
- **Response**:
  - On success:
    ✅ **Thank you! Feedback recorded for transaction <trans_num>**
  - On invalid input:
    "Invalid request" (HTTP 400)"
- **Security Consideration**:
  - Assumes feedback link is unique per user.
  - No authentication is required beyond possession of the feedback URL.

2. POST /feedback *(Optional/Future Use)*

A secondary endpoint designed to support API or frontend integrations.

- **Inputs (JSON body)**:
  {
  "trans_num": "12345",
  "feedback": 1
  }
- **Logic**:
  - Validates required fields.
  - Executes the same SQL update as the GET route.
  - Returns a structured JSON response confirming the update.
- **Use Case**:
  - Ideal for web forms, mobile apps, or dashboards needing POST support.

Design Patterns & Architecture:

- **Microservice Pattern**:
  The feedback server is a standalone RESTful service with a single responsibility: handle user input and update the database. This separation improves modularity and testability.
- **Stateless Design**:
  Each request opens a new DB connection, performs the update, and closes the connection, allowing the service to scale horizontally.
- **Defensive Programming**:
  Input validation ensures only legitimate feedback is recorded, avoiding malformed data and SQL errors.

Scalability & Performance:

- **Expected Load**: Minimal. Only one feedback request per fraud alert, and typically <1% of total transactions are flagged as fraud.
- **Performance**: Very high. A single feedback request results in a simple primary key update, a fast and efficient database operation.
- **Throughput**: Can handle hundreds of requests per second if needed. Easily horizontally scalable if usage increases.

Security Considerations:

- **Current Design**:
  - No authentication; feedback is authorized by possession of the feedback URL.
  - Uses parameterized queries to prevent SQL injection.
- **Potential Improvements**:
  - Include secure tokens in feedback URLs.
  - Use hashed or encrypted identifiers.
  - Log user identity if tracking is needed.

Integration with the Pipeline:

- **Data Flow**:
  - The kafkaconsumer.py detects fraud, stores the prediction in MySQL, and includes a feedback link in the email.
  - The user clicks the link, triggering the feedback server.
  - The feedback server updates the feedback field in the fraud_predictions table.
- **Decoupled Interaction**:
  - The consumer continues processing Kafka messages without waiting for user feedback.
  - Feedback is logged asynchronously and can be queried later for monitoring or model retraining.

Use in Model Evaluation & Retraining:

- The feedback stored in MySQL enables:
    - Calculation of precision and recall based on real user validation.
    - Labeling of false positives and false negatives.
    - Retraining the model using updated, user-labeled data.

Deployment Notes:

- **Runs on**: 0.0.0.0:5000 (container listens on all interfaces)
- **Docker-Compose**: Port 5000 is mapped from container to host.
- **Email Integration**: If the feedback links use localhost, they assume the user is on the same machine. For production, links should use the server's public IP or domain.

Design Limitations:

- No authentication on feedback links — relies on obscurity and link uniqueness.
- No user identity tracking — only records the feedback, not who submitted it.
- No UI — feedback confirmation is a simple plaintext response.
- No rate limiting or spam protection — could be added if needed.

Possible Enhancements:

- Add HTML responses with styled "Thank you" pages.
- Include feedback tracking by user/email.
- Add token-based link security.
- Expand POST endpoint to support additional metadata.
- Add a small UI for manual feedback or admin corrections.

Summary:

feedback_server.py is a **critical connector between detection and learning**. It enables real-world validation of fraud predictions by letting users confirm or refute alerts via simple, secure feedback links.

Despite its small size, it plays a big role in transforming a one-way detection pipeline into an interactive learning system, paving the way for **continuous learning**, human-in-the-loop validation, and more trustworthy fraud detection.

F- Dockerfile-producer – Containerizing the Kafka Data Producer

Purpose:

The Dockerfile-producer defines how to build a Docker image for the **Kafka Producer microservice**. Its goal is to encapsulate the kafkaproducer.py script along with all required dependencies and the dataset (final_transactions.csv) into a **self-contained, portable container**. This ensures consistent behavior across environments, removes manual setup steps, and enables reproducible data ingestion workflows in a microservice architecture.

By containerizing the producer:

- You eliminate dependency mismatches (Python version, package versions, etc.).
- You simplify deployment in distributed systems.
- You gain the ability to replay or restart data ingestion with a single command.

Technologies Used:

- **Docker**: Container runtime used for packaging and deployment.
- **Base Image**: python:3.9-slim, a lightweight Debian-based image with Python 3.9.
- **Project Files Included**:
    - kafkaproducer.py: The Kafka data ingestion script.
    - final_transactions.csv: The static dataset to stream.
    - requirements.txt: Contains all Python dependencies, shared across components.

Dockerfile Breakdown & Logic:

```
FROM python:3.9-slim

WORKDIR /app

COPY requirements.txt .

RUN pip install --no-cache-dir -r requirements.txt

COPY kafkaproducer.py .

COPY final_transactions.csv .

CMD ["python", "kafkaproducer.py"]
```

Step-by-step Explanation:

1. **FROM python:3.9-slim**
    - Uses a minimal Python 3.9 base image to reduce image size and improve security.
    - Ensures consistent Python runtime across environments.
2. **WORKDIR /app**
    - Sets the working directory inside the container.
    - All subsequent operations (COPY, RUN, CMD) are executed relative to /app.
3. **COPY requirements.txt .**

- Copies the dependencies list first to leverage Docker's layer caching.
- If requirements don't change, subsequent builds will reuse this layer.

4. **RUN pip install --no-cache-dir -r requirements.txt**
   - Installs Python libraries (e.g., kafka-python, pandas, json, etc.).
   - --no-cache-dir avoids storing cached wheels, keeping the image slim.
5. **COPY kafkaproducer.py . and COPY final_transactions.csv .**
   - Adds the producer script and dataset into the container.
   - Embeds the data into the image, making it self-contained for reproducibility.
6. **CMD ["python", "kafkaproducer.py"]**
   - Sets the default command to run when the container starts.
   - Uses exec form (JSON array) for better performance and signal handling.

Design Considerations:

- **Image Self-Containment**: Includes all code and data to run independently without mounting external volumes.
- **Layer Caching**: Requirements are copied and installed before application code, following Docker best practices.
- **Lightweight & Secure**: Uses a slim Python base image and no unnecessary packages or tools.
- **Portability**: Can run identically across environments with Docker installed.

Efficiency Notes:

- **Unused Dependencies**: The shared requirements.txt includes packages not used by the producer (e.g., xgboost, scikit-learn). This increases image size slightly but simplifies maintenance by sharing dependencies across components.
- **Build Performance**: Copying the dataset into the image increases build time and size but simplifies deployment. For frequently changing data, mounting volumes or pulling from an external source would be better.

Integration with Docker Compose:

- In docker-compose.yml, the producer service:
  - **Builds from**: Dockerfile-producer
  - **Depends on**: Kafka (depends_on: kafka)
  - **Does not expose ports**: Since it only sends data to Kafka, no external communication is needed.
- **Execution Flow**:
  - On docker-compose up, the producer connects to Kafka and begins sending transactions.
  - After sending all records, it logs throughput stats and exits.
  - Can be restarted anytime to replay the dataset (e.g., docker-compose restart producer).

Microservice Architecture Context:

- The producer is packaged as a **standalone microservice**, independently deployable and restartable.
- It can be scaled, updated, or swapped without affecting other parts of the pipeline (e.g., Consumer or Feedback services).
- Adheres to the **Containerization Design Pattern**:
    - Immutable infrastructure: behavior is determined by image state, not runtime configuration.
    - Reproducibility: ensures the same environment is used across dev, staging, and production.

Scalability & Extensibility:

- **Replayable**: Restarting the container replays the same transaction stream from the beginning.
- **Extensible**: Can be adapted to stream different datasets or simulate live ingestion from an API.
- **Parallelization**: Could scale horizontally with multiple producer instances (with care to partition data).

Performance Considerations:

- **Runtime Speed**: Minimal overhead from Docker; data is streamed as fast as possible.
- **Build Time**: Slower due to CSV inclusion and broad requirements list.
- **Startup Time**: Fast, thanks to slim image and pre-installed packages.

Security Considerations:

- **Runs as root** by default (inherited from base image). For production, consider adding a non-root user.
- **No ports exposed**: No direct attack surface; connects only to internal Kafka broker.
- **Hardcoded configuration**: Kafka broker address is embedded in the script. Can be moved to environment variables for flexibility.

Deployment Options:

- **Local Testing**: Ideal for local dev environments using Docker Compose.
- **Cloud Deployment**: Easily portable to ECS, GCP Cloud Run, Kubernetes, etc.
- **CI/CD Integration**: Can be built and published as an image to a container registry for automated deployments.

Summary:

Dockerfile-producer defines a **reliable, reproducible environment** for the Kafka Producer microservice. It encapsulates the code, data, and dependencies needed to stream transaction records into Kafka. By adhering to containerization best practices, it contributes to a modular, scalable fraud detection pipeline that is **easy to develop, test, deploy, and maintain**.

G- Dockerfile-consumer – Containerizing the Fraud Detection Consumer

Purpose:

The Dockerfile-consumer defines the container build instructions for the **Kafka Consumer microservice**, which performs real-time fraud detection using a trained machine learning model. Containerizing this component ensures that the Python environment, model file, and dependencies (e.g., Kafka client, MySQL connector, XGBoost) are consistent across development, testing, and production environments.

This approach:

- Eliminates environment inconsistencies.
- Simplifies deployment using Docker Compose or Kubernetes.
- Bundles the model and logic into a reproducible, scalable microservice.

Technologies Used:

- **Docker**: Container platform to build and run the service.
- **Base Image**: python:3.9-slim for a lightweight, minimal footprint.
- **Python Libraries**: Installed via requirements.txt, including kafka-python, xgboost, mysql-connector-python, and others.
- **Project Files Included**:
    - kafkaconsumer.py: Main streaming fraud detection script.
    - fraud_detection_bundle.pkl: Serialized model and threshold.
    - notifier.py: Module responsible for sending fraud alert emails.

Dockerfile Breakdown:

```
FROM python:3.9-slim

WORKDIR /app

COPY requirements.txt .

RUN pip install --no-cache-dir -r requirements.txt

COPY kafkaconsumer.py .

COPY notifier.py .

COPY fraud_detection_bundle.pkl .

CMD ["python", "kafkaconsumer.py"]
```

Explanation:

1. **Base Image** – python:3.9-slim
    - Lightweight, secure image with Python 3.9 pre-installed.
    - Ensures the same environment as model training and preprocessing.
2. **Working Directory** – /app
    - Standardized directory for application files inside the container.
3. **Install Dependencies**
    - COPY requirements.txt followed by pip install --no-cache-dir -r requirements.txt
    - Avoids caching to reduce image size.
    - Docker layer caching means dependencies aren't reinstalled if only source code changes.
4. **Copy Application Files**
    - kafkaconsumer.py: Main fraud detection logic.
    - notifier.py: Email alert system (imported in kafkaconsumer.py).
    - fraud_detection_bundle.pkl: Serialized XGBoost model and threshold, generated offline.
5. **Command**
    - CMD ["python", "kafkaconsumer.py"]: Starts the consumer on container launch.

Key Design Decisions:

Bundling the Model (fraud_detection_bundle.pkl)

- Ensures version alignment between model and consumer code.
- Simplifies deployment: no need to fetch from cloud or mount external storage.
- Promotes reproducibility and model traceability.

Including notifier.py

- Required for email functionality within the consumer.
- Treated as a dependency rather than a separate service.
- Allows modular notification logic with potential for easy substitution or expansion.

Dependency Strategy

- Uses a shared requirements.txt across all services.
- May include unnecessary packages (e.g., Flask, not used by the consumer).
- Slightly increases image size but simplifies maintenance in multi-service projects.

Integration with Docker Compose:

- Defined in docker-compose.yml as the consumer service.
- **Depends on**: Kafka and MySQL (ensures those are up first).
- **No ports exposed**: Does not serve web traffic, only connects to internal services.
- Automatically connects to Kafka (kafka:9092) and MySQL (mysql:3306) using Docker Compose DNS resolution.

Architecture Alignment:

- **Microservice Pattern**: The container isolates the fraud detection logic, allowing it to run and scale independently.
- **Containerization Best Practices**:
    - Reproducible builds.
    - Separation of code and configuration.
    - Lean, single-responsibility container.
- **ML Deployment Strategy**:
    - "Bake the model into the image" approach.
    - Ensures model and logic are tightly coupled and versioned together.

Scalability & Extensibility:

- **Horizontal Scaling**:
    - In Kafka, multiple consumers in the same group can process messages in parallel.
    - This container can be replicated easily to handle higher throughput.
- **Model Updates**:
    - Simply replace fraud_detection_bundle.pkl and rebuild the image.
    - If features change, update dtype_mapping and retrain.
- **Notification Logic**:
    - Easily replaceable thanks to the Notifier interface.
    - Could eventually delegate email sending to a separate microservice if needed.

Performance Considerations:

- **Memory Usage**:
    - Slightly higher due to XGBoost model and Pandas DataFrame operations.
    - Still efficient and handles one transaction at a time.
- **Startup Time**:
    - Installing XGBoost and dependencies increases image build time.

- Runtime is fast; XGBoost makes predictions in milliseconds.

- **Logging**:
  - Logs (including fraud alerts and errors) stream to the container's stdout.
  - Easily collected by Docker or centralized logging systems.

Security Considerations:

- **Runs as root** by default (inherited from base image).
- **Secrets (e.g., email password, DB credentials)** are hardcoded in source code.
  - For production: move to environment variables and configure in docker-compose.yml.
  - Consider non-root users and secrets management best practices

---

Operational Behavior:

- **Designed to run continuously** until manually stopped.
- Handles its own retries for Kafka and MySQL connections.
- If the container exits due to an error, Docker Compose does not auto-restart it by default — can be configured using restart: on-failure.

Comparison with Dockerfile-producer:

Feature	Producer	Consumer
Dataset embedded	final_transactions.csv	(only processes messages)
Model embedded	N/A	fraud_detection_bundle.pkl
Email alerts	N/A	Uses notifier.py
Runtime duration	Infinite (streaming service)	Infinite (streaming service)
Port exposure	None	None

Both Dockerfiles follow a similar structure but differ in purpose: the producer now streams transaction data in real time, while the consumer continuously processes, predicts, and triggers alerts.

Summary:

Dockerfile-consumer defines a robust, portable, and reproducible runtime environment for the **Fraud Detection Kafka Consumer service**. It encapsulates the model, code, and dependencies required to consume transaction messages from Kafka, make fraud predictions using XGBoost, and send real-time alerts.

By containerizing this component:

- Deployment becomes seamless and environment-agnostic.
- Scaling is simplified (via Docker Compose or orchestration platforms).
- The service behaves consistently and is tightly coupled to the exact model it was trained with.

This is a textbook example of **containerized ML inference in production**, following modern best practices for microservice and model deployment.

H- Dockerfile-feedback – Containerizing the Feedback Service

Purpose:

Dockerfile-feedback defines the container environment for the **Feedback Web Service**, which collects user responses to fraud alerts via email. By containerizing this Flask-based application, the system guarantees that:

- The correct Python and Flask versions are used.
- The service is portable across machines.
- It can be integrated seamlessly into a multi-container pipeline with Docker Compose.

This design aligns with a **microservice architecture**, keeping the feedback logic isolated from the Kafka Consumer or Producer services.

Technologies & Structure:

- **Base Image**: python:3.9-slim
  Lightweight, secure base with minimal OS overhead.
- **Web Framework**: Flask
  Used to create a simple HTTP API with GET and POST endpoints for feedback collection.

- **Database Connector**: mysql-connector-python
  Enables writing user feedback into the fraud_predictions MySQL table.
- **Python File Included**:
  - feedback_server.py: Flask app that exposes /feedback endpoint to receive user confirmation on fraud alerts.

Dockerfile Overview:

FROM python:3.9-slim

WORKDIR /app

COPY requirements.txt .

RUN pip install --no-cache-dir -r requirements.txt

COPY feedback_server.py .

CMD ["python", "feedback_server.py"]

Explanation:

1. **Base Image – python:3.9-slim**
    - Provides a consistent Python environment across all containers.
    - Keeps the image lightweight while supporting modern Python features.
2. **Working Directory – /app**
    - Application code is stored here.
    - Simplifies execution context for running and debugging the app.
3. **Installing Dependencies**
    - COPY requirements.txt .
      Shared requirements file copied in first to leverage Docker's caching.
    - RUN pip install --no-cache-dir -r requirements.txt
      Installs all required Python packages. This includes Flask and mysql-connector-python, but also other unused libraries (e.g., xgboost, kafka-python) since all services share the same requirements list.
4. **Copying Application Code**
    - COPY feedback_server.py .
      Brings in the actual feedback web service.
5. **Command**
    - CMD ["python", "feedback_server.py"]
      Starts the Flask development server. The app is configured to run on 0.0.0.0:5000 so it is accessible via Docker networking.

Design Considerations:

- **Self-Contained**:
  Includes everything needed to run the Flask service — no external dependencies are required after image build.
- **Simplicity**:
  The service runs on Python's built-in Flask development server. This is acceptable for small-scale or demo environments.
- **Modular Deployment**:
  Feedback logic runs in its own container, independently from the fraud detection engine. This allows:
    - Independent updates and scaling
    - Easier testing and debugging.
    - Clear separation of concerns.

- **Shared Requirements File**:
  Even though this service only needs a subset of packages (Flask + MySQL connector), the full requirements.txt is used for simplicity. This slightly increases image size but reduces build configuration complexity.

Networking & Port Mapping:

- Flask runs on 0.0.0.0:5000 inside the container.
- Docker Compose maps this to host port 5000 (or another specified port).
- This allows users to click feedback links from their emails and reach the feedback service directly.

Integration with Docker Compose:

- The service is defined as feedback in docker-compose.yml.
- It **depends on** the MySQL container to ensure the database is available before starting.
- It connects to the MySQL service via hostname mysql, as Docker Compose provides internal DNS.

Extensibility & Production Notes:

Web Server:
For production, replace Flask's built-in server with a WSGI server like Gunicorn for better performance and concurrency:

```
RUN pip install gunicorn

CMD ["gunicorn", "-w", "4", "-b", "0.0.0.0:5000", "feedback_server:app"]
```

- **Security Enhancements**:
  - The service currently uses query parameters without authentication.
  - Add validation tokens or hashed transaction IDs to secure feedback links in production.
- **Custom Requirements File**:
  Consider using a minimal requirements-feedback.txt to reduce build size and install time.
- **Environment Variables**:
  - Move DB credentials or hostnames into environment variables for flexibility and security.

Performance & Scalability:

- **Expected Load**: Very low. Only triggered by user clicks on fraud alerts.
- **Resource Usage**: Minimal — lightweight Flask server and single-row SQL updates.
- **Horizontal Scalability**: Stateless design means the service can scale across instances or containers.

Summary:

Dockerfile-feedback defines a **clean, minimal container** for the feedback collection microservice. It encapsulates a simple Flask app that allows users to confirm or refute fraud predictions by clicking email links. This completes the feedback loop in the real-time fraud detection pipeline.

Benefits of this design:

- Easy deployment using Docker Compose.
- Portable, isolated execution environment.
- Modular architecture supporting continuous improvement of the model based on real user input.

Though small in scope, this service is a **crucial link** between machine learning predictions and human verification, enabling the system to evolve and learn from actual user behavior over time.

I- docker-compose.yml – Orchestrating Multi-Container Services

Purpose:

The docker-compose.yml file defines the blueprint for deploying the **end-to-end fraud detection pipeline**. It orchestrates all key services—Zookeeper, Kafka, MySQL, Kafka Producer, Kafka Consumer, and the Feedback Web Service—into a single, networked system. By using Docker Compose, the entire application can be launched with a single command, ensuring all services are:

- Started in the correct order.
- Connected via a shared internal network.
- Configured consistently across environments.

This Compose file is **essential for reproducibility, deployment, and local testing** of the full real-time data pipeline.

Services Defined

1. Zookeeper

- **Image**: confluentinc/cp-zookeeper
- **Port**: 2181 (mapped to host)
- **Purpose**: Coordinates Kafka brokers.
- **Configuration**: Sets ZOOKEEPER_CLIENT_PORT and ZOOKEEPER_TICK_TIME.
- **Notes**: Stateless in this setup (no volume), suitable for demo purposes.

2. Kafka Broker

- **Image**: confluentinc/cp-kafka
- **Port**: 9092 (mapped to host)
- **Depends on**: zookeeper
- **Environment**:
    - Configured as broker.id=1

- Uses internal DNS (zookeeper:2181)
- Advertises itself as kafka:9092
- Internal topic replication factor is set to 1 (since there's only one broker)

- **Notes**: Core messaging infrastructure for streaming data; loosely coupled with consumers/producers via topics.

3. MySQL Database

- **Image**: mysql:5.7
- **Ports**: Container 3306 → Host 3307
- **Environment**:
  - MYSQL_ROOT_PASSWORD
  - MYSQL_DATABASE=kafka_data
- **Volumes**:
  - Persists data in a named volume mysql_data
- **Purpose**: Stores fraud prediction results and user feedback.
- **Notes**: The only persistent state in the pipeline, necessary for feedback and result analysis.

4. Kafka Producer

- **Build**: From Dockerfile-producer
- **Depends on**: kafka
- **Function**: Reads the static dataset (final_transactions.csv) and publishes transaction records to Kafka.
- **Behavior**:
  - Runs once and exits after sending all data.
  - No exposed ports or volumes.
- **Notes**: Stateless and replayable; designed for bulk simulation of real-time events.

5. Kafka Consumer

- **Build**: From Dockerfile-consumer
- **Depends on**: kafka, mysql
- **Function**:
  - Continuously listens to the transaction_data Kafka topic.
  - Applies ML model to detect fraud.
  - Inserts results into MySQL.
  - Sends alert emails on suspected frauds.
- **Notes**: Runs indefinitely; critical component tying together inference, storage, and notification.

6. Feedback Web Service

- **Build**: From Dockerfile-feedback
- **Depends on**: mysql
- **Ports**: Container 5000 → Host 5000
- **Function**:
    - Exposes a /feedback endpoint via Flask.
    - Updates the fraud prediction records in MySQL based on user input from email links.
- **Notes**: Stateless. Only exposed web-facing service, allowing human interaction with the pipeline.

Orchestration & Startup Logic

- **Startup Order**:
    - Compose uses depends_on to determine startup sequence:
        - Zookeeper → Kafka → MySQL → (then) Producer, Consumer, Feedback.
- **Readiness Handling**:
    - depends_on does **not** wait for full readiness—only for container start.
    - **Retry logic** in kafkaproducer.py, kafkaconsumer.py, and feedback_server.py ensures services wait for Kafka or MySQL to become available before proceeding.

Networking & Communication

- **Default Network**: All services share an internal network.
- **Service Discovery**:
    - Containers can refer to each other by name (kafka, mysql, feedback, etc.).
    - Kafka advertised listener is kafka:9092.
    - MySQL is reached at mysql:3306.
- **Port Exposure**:
    - **Kafka**: 9092 (host) — can be used to connect from CLI tools.
    - **MySQL**: 3307 (host) — allows inspection via GUI tools like MySQL Workbench.
    - **Feedback**: 5000 (host) — accessed by users through email links.

Volumes and Persistence

- **MySQL Data Volume**:
    - Declared as mysql_data
    - Ensures persistence of:
        - Fraud prediction results

        - Feedback labels from users

- **Kafka & Zookeeper**:
    - Do not use volumes in this setup — messages are ephemeral (acceptable for demos).
    - For production durability, volumes for Kafka logs and Zookeeper state would be recommended.

Architecture & Design Principles

- **Microservice-Oriented**:
    - Each component is isolated in its own container with a single responsibility:
        - Kafka/Zookeeper: Event backbone
        - Producer: Data ingress
        - Consumer: Model scoring and alerts
        - Feedback: User validation
        - MySQL: Central persistent store
- **Decoupling**:
    - Kafka acts as the buffer and glue — enabling asynchronous interaction between Producer and Consumer.
    - Feedback service operates independently via shared DB access.
- **Modularity & Reusability**:
    - Components can be individually restarted or scaled.
    - Model updates only require rebuilding the consumer image.

Deployment Notes

- **Reproducibility**:

Anyone with Docker and this Compose file can bring up the pipeline with:

docker-compose up --build

- **Scalability**:
    - Services like Kafka and Consumer can be scaled by adding brokers or replicas (though additional configuration would be needed).
- **Security Considerations**:
    - Passwords (e.g., MySQL root) are hardcoded in Compose.
    - In production, use .env files or Docker Secrets for sensitive information.
- **Observability**:
    - Logs for each service are visible via docker-compose logs.

- Useful for monitoring streaming performance and debugging (e.g., seeing fraud alerts or email send failures).

Summary

The docker-compose.yml file is the **central orchestration layer** for your real-time fraud detection pipeline. It captures the full system architecture and abstracts away infrastructure complexity, allowing developers and analysts to focus on:

- Model development
- Data ingestion
- Feedback collection

This Compose file turns a multi-component architecture into a **portable, version-controlled, and easily reproducible system**—ready to run with a single command.

J- requirements.txt – Python Dependencies and Environment Management

Purpose:

The requirements.txt file defines the complete set of Python dependencies required to run all components of the fraud detection pipeline. By pinning exact versions for each package, it ensures **reproducibility**, **compatibility**, and **consistency** across development and production environments. This is especially critical in a machine learning context, where models can behave differently under different library versions.

This file acts as a **single source of truth** for the Python environment, enabling seamless Docker image builds and preventing environment drift.

Contents and Key Technologies

Package	Version	Purpose
kafka-python	2.0.2	Kafka client used by both Producer and Consumer for real-time messaging.
pandas	2.0.3	Data manipulation and preprocessing in the Consumer service.
numpy	1.24.3	Numerical computing library; foundational for pandas and ML model input.
mysql-connector-python	8.0.33	MySQL database client used by Consumer and Feedback service for inserts/updates.
joblib	1.2.0	Used to load the serialized machine learning model (fraud_detection_bundle.pkl).

xgboost	2.0.3	Core ML library used to train and run the fraud detection model.
scikit-learn	1.3.0	Used during model training (e.g., train_test_split, metrics); included for compatibility.
flask	2.2.5	Lightweight web framework powering the Feedback Service.

This list covers the full technology stack: data science, messaging, database interaction, and web service—all in one unified environment.

Design Considerations

- **Single Unified File:**
  All components (Producer, Consumer, Feedback) share the same requirements.txt. While some services install unused packages (e.g., XGBoost in Feedback), this design:
  - Simplifies maintenance.
  - Guarantees all dependencies are present in every container.
  - Reduces the risk of missing packages during container builds.
- **Pinned Versions (==)**:
  Ensures deterministic builds and guards against:
  - Breaking changes in newer releases.
  - Model incompatibility due to version mismatch (e.g., pickled XGBoost models requiring matching library version).
  - Subtle performance or behavior shifts (e.g., changes in Pandas indexing behavior between versions).

Environment Reproducibility & Docker Integration

Docker Compatibility:
All Dockerfiles use this file via
RUN pip install --no-cache-dir -r requirements.txt

- This guarantees that every container—no matter where it's built—has an **identical software environment**.
- **Machine Learning Model Reliability:**
  Models trained using specific library versions (e.g., XGBoost 2.0.3, scikit-learn 1.3.0) will load and run reliably in inference mode.

Consistent Across Dev and Prod:
Developers can recreate the exact environment locally with:

pip install -r requirements.txt

- This ensures zero surprises when deploying to Docker or cloud environments.

Performance & Scalability Implications

- The environment is **comprehensive**, not minimal. This increases image size and container build time slightly, but:
    - Ensures all necessary tools are available across components.
    - Reduces development overhead from missing packages.
- **Memory Usage**:
  Libraries like pandas, xgboost, and scikit-learn have non-trivial memory footprints. However, they are necessary for the model's logic and data transformation in the Consumer.
- **Throughput Optimization**:
  Pinned versions avoid unexpected performance regressions or bugs introduced in newer versions of core libraries (e.g., Pandas, Kafka client).

Security & Auditability

- **Version Pinning** helps track and audit the exact set of packages used at any point in time.
- **Vulnerability Management**:
  If a vulnerability is discovered in a specific version (e.g., Flask 2.2.5), it's easy to identify affected environments and upgrade precisely.

Documentation & Stack Transparency

Reading requirements.txt reveals the **technological foundation** of the system:

- **Real-Time Messaging**: kafka-python
- **Data Science & ML**: xgboost, scikit-learn, pandas, numpy
- **Web Interface**: flask
- **Database Interaction**: mysql-connector-python

This acts as de facto documentation of the stack, helpful for onboarding new developers or presenting in academic or professional settings.

Summary

The requirements.txt file is a foundational component of the fraud detection pipeline. It ensures:

- Environment consistency across local machines, Docker containers, and deployment targets.
- Reliability of ML model execution.
- Ease of maintenance and debugging.

By pinning versions and consolidating all dependencies into one file, the project ensures that all parts of the system—from batch preprocessing to real-time inference to user feedback—work together seamlessly.

This approach reflects best practices in machine learning engineering and software deployment, balancing simplicity, reproducibility, and robustness.

Conclusion & Final Thoughts:

This project successfully designed, built, and deployed a **real-time credit card fraud detection system** using a modern data engineering stack. From ingesting transactional data via **Kafka**, to applying an **XGBoost ML model** for live predictions, issuing **email alerts**, and capturing **user feedback**, the system demonstrates an end-to-end streaming pipeline with real-world applicability.

Key Achievements:

- **Streaming Architecture:** Enabled real-time fraud detection using Kafka for asynchronous, scalable communication between services.
- **ML Integration:** Deployed a trained XGBoost model within a consumer microservice to score live transactions with sub-second latency.
- **Notification System:** Integrated a secure, masked-email alerting mechanism to notify stakeholders of suspected fraud.
- **Feedback Loop:** Captured real-world validation data from users via embedded feedback links, closing the machine learning lifecycle.
- **Containerization:** Leveraged Docker and Docker Compose to orchestrate a reproducible, portable microservice ecosystem.

Future Work:

- **Model Retraining Pipeline:** Automate model updates using the stored feedback for periodic retraining.
- **Data Lake Integration:** Stream data to cloud-based data lakes or warehouses for long-term analytics and auditing.
- **Streaming Enhancements:** Add support for Kafka topic partitioning and parallel consumers to improve throughput at scale.
- **User Personalization:** Dynamically send alerts to actual cardholders (not a static email), integrating user profile management.
- **Security Improvements:** Use environment variables and secret managers to secure credentials across containers.

Acknowledgments:

This project was developed as part of the academic curriculum at **UMass Dartmouth**. I would like to express my sincere gratitude to **Professor Ashok Kumar Patel** for his invaluable guidance, mentorship, and encouragement throughout the course of this project. His insights into real-time systems and data engineering were instrumental in shaping the architecture and ensuring the system's robustness.

Special thanks to the broader open-source community and the maintainers of tools like **Apache Kafka**, **XGBoost**, **Flask**, and **Docker**, whose work enabled the rapid development of this end-to-end streaming pipeline.

Final Note:

Fraud detection is a critical and evolving field. This pipeline is a strong foundation that demonstrates not just technical skill but the ability to integrate **data science, engineering, and user experience** into a cohesive, production-grade system.

"Great systems aren't built in silos. They listen, learn, and evolve — just like this one."