information technology institute

Project Team Members:

Muhammed Abdalsattar

Abdelrahman Mousa

Heba Magdy

Mohamed Osama

Aya Faisal

Mohamed Osama

Nariman Abdo

Supervised by :

Eng / Abdelrahman Mahmoud

Acknowledgment

I would like to express my sincere gratitude for the guidance, resources, and perseverance that were instrumental in the successful development of this Real-Time Fraud Detection System on Google Cloud Platform. This project was a journey through complex architectural design, iterative implementation, and rigorous real-world debugging, and it would not have been possible without a combination of powerful tools and dedicated effort.

First and foremost, I extend my thanks to the guiding principles of modern data engineering that shaped this project's architecture. The interactive process of questioning, designing, and refining—from initial concepts to a final, robust pipeline—was fundamental to its success. The ability to engage in a technical dialogue, challenge assumptions, and systematically solve problems was an invaluable part of the development experience.

I am especially grateful for the powerful and integrated ecosystem provided by the Google Cloud Platform. The seamless interoperability between services
like **Pub/Sub**, **Dataflow**, **Firestore**, **Cloud Run**, **BigQuery**, **Dataproc**, and **Cloud Composer** provided the foundation upon which this entire system was built. The access to serverless technologies and managed services allowed the focus to remain on data logic and business value, rather than on infrastructure management.

My gratitude also goes to the detailed logging, error reporting, and diagnostic tools within the Google Cloud ecosystem. The journey through troubleshooting IAM permissions, network configurations, service quotas, and Spark version incompatibilities was challenging, but the clarity of the error messages provided a critical feedback loop that made solving these complex issues possible. This project is a testament to the power of systematic, log-driven debugging.

I am also thankful for my own persistence and dedication throughout the challenging phases of this project. The determination to push through complex technical hurdles—from initial design to the final successful execution of the ETL pipeline—was the driving force behind this achievement. This project has not only resulted in a powerful technical solution but has also significantly enhanced my skills in cloud architecture and distributed data processing.

Finally, this project stands as a result of a unique collaboration between human vision and technological capability. I am proud of the work accomplished and hope that this system will

serve as a robust and scalable solution for fighting financial fraud and improving security in the digital economy.

Thank you.

Chapter1: Introduction to systems and components

1. Introduction

In the modern financial landscape, fraud detection is a mission-critical capability. With the explosion of digital transactions—driven by e-commerce, mobile wallets, and online banking—the threat of fraudulent activity has never been greater. Incidents such as identity theft, unauthorized access, and payment fraud can cause significant financial and reputational damage.

Traditional fraud detection approaches typically depend on static rules or manual verification, which are neither scalable nor fast enough to detect sophisticated fraud attempts in real time. This has created a growing need for dynamic, intelligent, and scalable solutions.

This project presents a Real-Time Fraud Detection System built using big data frameworks and machine learning techniques. The system is capable of ingesting high-velocity transaction streams, analyzing them in real time, detecting anomalies, and triggering alerts with minimal delay. The goal is to help financial institutions **mitigate risks by making fraud detection faster, smarter, and more adaptable.**

Abstract

The objective of this project is to create a real-time fraud detection pipeline capable of identifying suspicious financial transactions as they occur. The architecture integrates multiple technologies: Apache Kafka for real-time data ingestion, Apache Spark Structured Streaming for scalable processing, and machine learning models for predictive fraud classification.

The solution is deployed on Google Cloud Platform (GCP), utilizing its managed services for data storage, compute scalability, and alerting. Historical transaction data is used to train classification models that learn the difference between legitimate and fraudulent behaviors. The

model continuously scores live transactions based on factors like location, frequency, and transaction amount.

Once anomalies are detected, the system generates immediate alerts and pushes them for further investigation, enabling financial teams to take timely action.

By combining stream processing, AI, and cloud infrastructure, this system delivers high-speed, automated fraud detection that scales with demand and continuously improves over time.

Motivation

The digital transformation of financial services has introduced new attack surfaces for fraudsters. As transactions move to online and mobile platforms, the complexity and scale of fraudulent activity have escalated.

Organizations lose billions annually due to fraud, and delayed responses only worsen the impact. Traditional tools, such as rules-based engines or manual auditing, are no longer sufficient. These methods can’t keep pace with the rapid flow of data or adapt to constantly evolving fraud patterns.

This project is driven by the need to:

1. Detect fraudulent behavior in real time to enable instant response.
2. Handle massive transaction throughput without degrading performance.
3. Stay ahead of fraud techniques by continuously updating predictive models.

Machine learning offers a compelling solution, as it can learn patterns from historical fraud cases and generalize to detect new ones. When combined with real-time data pipelines, it becomes possible to intercept fraud before damage occurs.

Problem Statement

The financial sector is under constant threat from fraudulent activities that evolve in both technique and frequency. The current challenges include:

- Latency in detection: Most systems detect fraud after the transaction is complete, making it hard to prevent damage.
- Manual or semi-automated checks: This leads to bottlenecks, inefficiencies, and poor scalability.
- Fixed rule sets: Easy for fraudsters to manipulate or circumvent.
- Lack of adaptability: Traditional systems can't evolve quickly enough to counter new fraud patterns.
- Scalability issues: Many tools can't handle the increasing transaction volume seen in today's digital platforms.

This project addresses those challenges by delivering a fraud detection system that:

- Ingests transaction data in real time using Kafka.
- Processes and scores data using Spark and ML models.
- Utilizes GCP for elastic infrastructure and reliable storage.
- Automatically triggers alerts when fraud is suspected.
- Continuously learns from new data to enhance detection over time.

This proactive, scalable, and intelligent system positions financial institutions to act faster and smarter in the fight against fraud.

1.1. Project Purpose & Business Goal

This document outlines the architecture and implementation of the real-time fraud detection system. The primary business goal is to identify and flag potentially fraudulent financial transactions as they occur, minimizing financial loss and protecting customer accounts. The system provides both real-time alerts and a historical data warehouse for batch analytics and model improvement.

1.2. Key Features

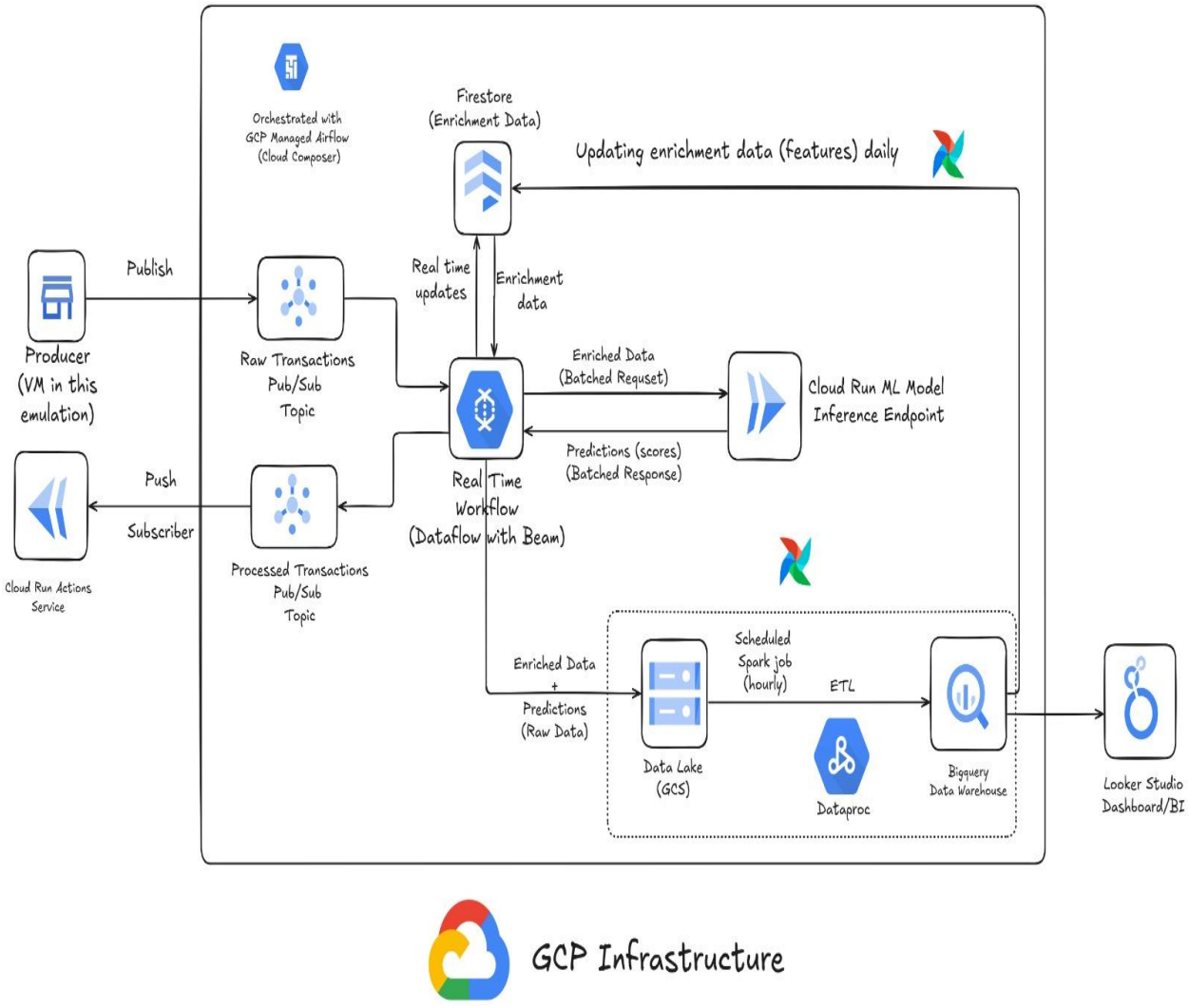
- **Real-Time Inference:** Sub-second transaction scoring using a deployed machine learning model.
- **Scalable Ingestion:** Built to handle high volumes of transaction events using Google Pub/Sub.
- **Real-Time Data Enrichment:** Enriches incoming transactions with low-latency lookups from a NoSQL database (Firestore).
- **Automated Batch Analytics:** A comprehensive data warehouse in BigQuery for BI, reporting, and future model retraining, populated by automated, hourly ETL jobs.
- **Data Activation:** A "reverse ETL" pipeline keeps the real-time data store (Firestore) fresh with the latest information from the data warehouse.
- **End-to-End Monitoring:** Integrated monitoring and alerting using Google Cloud's operations suite.

2. System Architecture

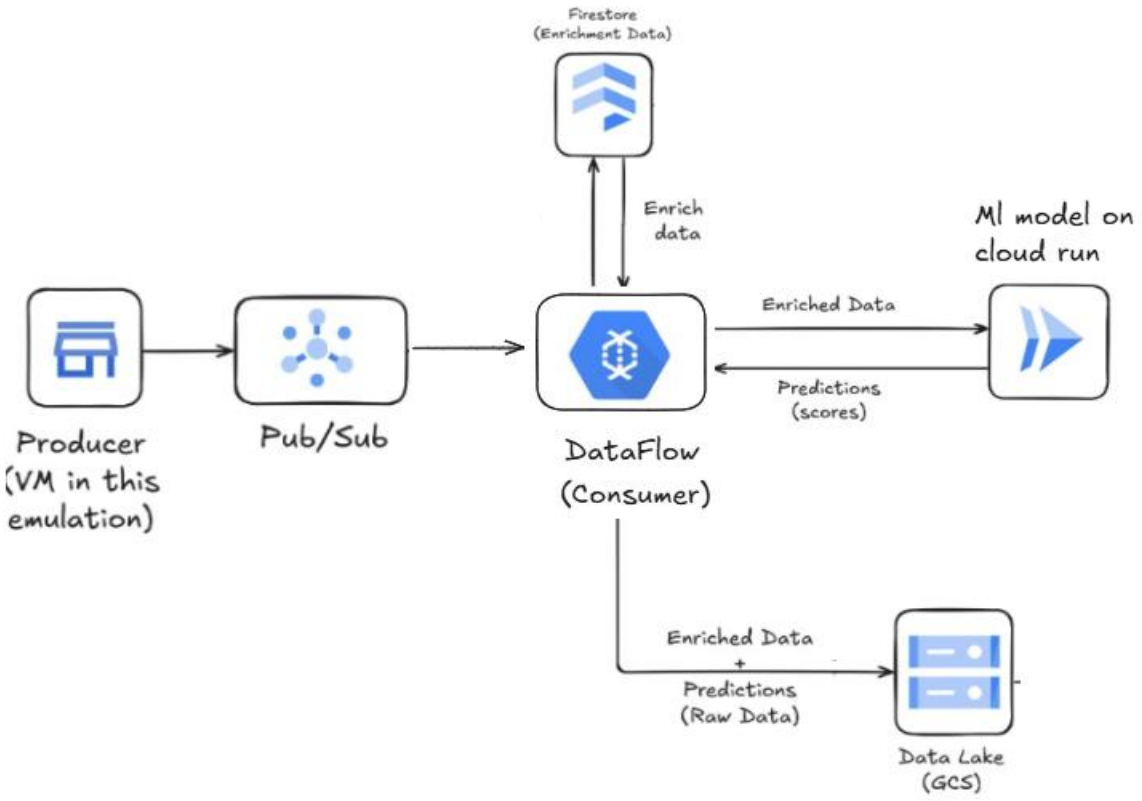
2.1. High-Level Diagram

The system is composed of three interconnected pipelines that create a complete data lifecycle.

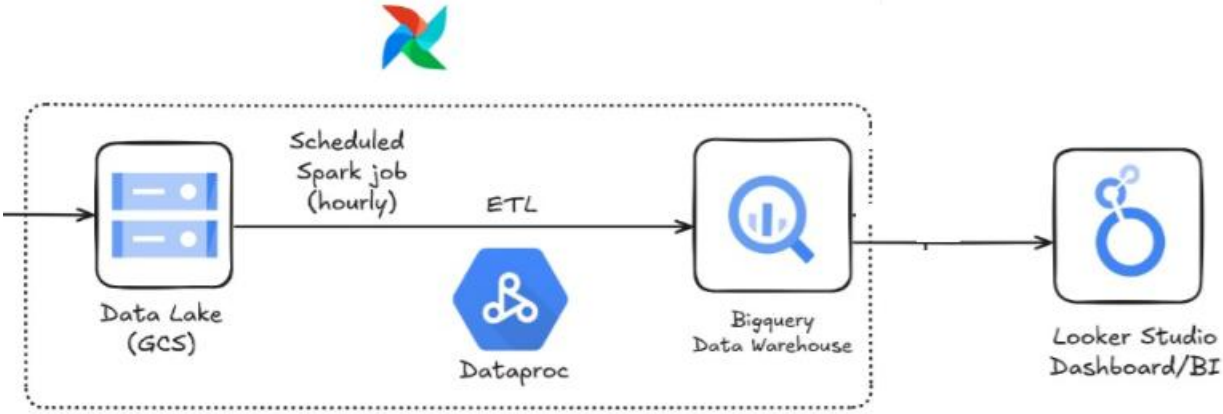
--Arch:



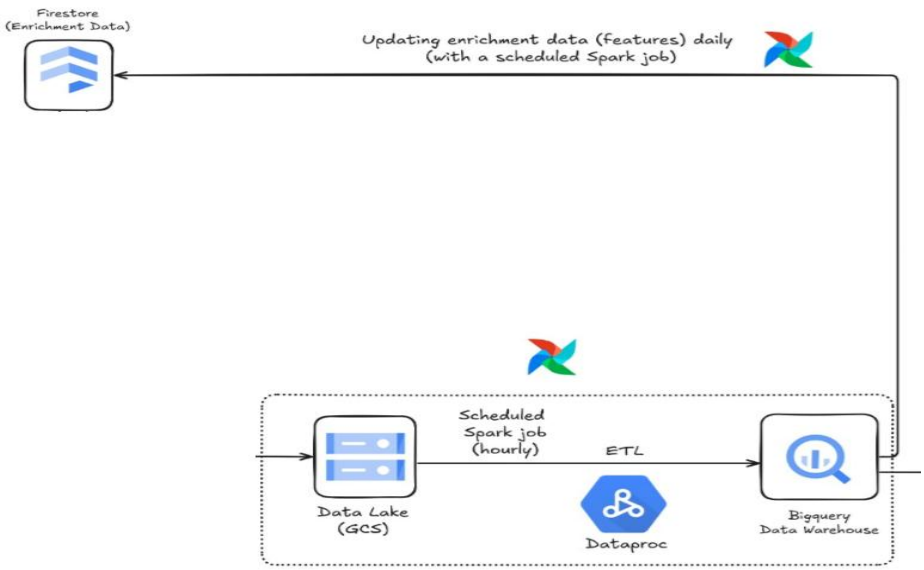
1. **Real-Time Streaming Pipeline (Inference)**



2. **Batch Analytics Pipeline (ETL)**



3. **Firestore Refresh Pipeline (Reverse ETL)**



Overarching Service:

- **Cloud Monitoring:** Provides observability across all components.

2.2. Google Cloud Platform Services

- **Pub/Sub**: A fully managed messaging service used for ingesting the high-velocity stream of raw transaction data.
- **Dataflow**: A serverless, managed Apache Beam service used for the real-time pipeline to preprocess data, call Firestore for enrichment, and invoke the ML model.
- **Firestore**: A NoSQL document database used as the low-latency enrichment store, providing real-time access to user, card, and merchant data.
- **Cloud Run**: A serverless platform for deploying and scaling the containerized machine learning model API.
- **Cloud Storage (GCS)**: Acts as the scalable data lake, storing historical, enriched transaction data in a partitioned Parquet format.
- **Cloud Composer**: A managed Apache Airflow service used to orchestrate all batch workflows, including the hourly GCS-to-BigQuery ETL and the daily BigQuery-to-Firestore refresh.
- **Dataproc**: A managed service for running Apache Spark jobs. Used for the heavy-lifting of batch ETL transformations and the reverse ETL to Firestore.
- **BigQuery**: A serverless, petabyte-scale data warehouse that stores the final, structured data in a Star Schema for analysis.
- **Looker Studio**: The business intelligence and data visualization tool used to build dashboards on top of BigQuery.
- **Cloud Monitoring**: Provides system-wide monitoring, logging, and alerting for all pipeline components.

3. Component Deep Dive

3.1. Producer

- **Purpose:** To simulate a realistic stream of financial transactions.
- **Implementation:** A custom Python script running on a Google Compute Engine (GCE) VM, orchestrated by Cloud Composer.
- **Output:** Publishes flat JSON transaction messages to a Pub/Sub topic.

3.2. Real-Time Ingestion: Pub/Sub

- **Purpose:** To act as a highly scalable and durable buffer, decoupling the producer from the real-time processing layer.
- **Configuration:** A single Pub/Sub topic (e.g., fraud-transactions) receives all incoming transaction messages.

3.3. Real-Time Processing & Enrichment: Dataflow (Consumer)

- **Purpose:** To consume transactions, enrich them with contextual data, get a fraud prediction, and land the complete record in the data lake.
- **Implementation:** A streaming Apache Beam pipeline running on the Dataflow managed service.
- **Workflow:**
    1. **Consume:** Reads messages from the Pub/Sub topic.
    2. **Enrich:** Makes API calls to Firestore to retrieve detailed data for the user_id, card_id, etc.
    3. **Predict:** Sends the enriched data to the Cloud Run model endpoint and receives a prediction score.
    4. **Sink:** Writes the final, fully enriched record (raw data + predictions) as a Parquet file to the GCS data lake.

3.4. Real-Time Data Store: Firestore

- **Purpose:** To serve as a low-latency, key-value database for real-time enrichment.
- **Data Model:** NoSQL collections for users, cards, etc., where the document ID is the natural key for millisecond-level lookups.

3.5. ML Model Serving: Cloud Run

- **Purpose:** To provide a scalable, stateless HTTP endpoint for serving the fraud detection model.
- **Implementation:** An ML model packaged in a Docker container and deployed as a Cloud Run service, exposing a /predict endpoint.

3.6. Data Lake: Google Cloud Storage (GCS)

- **Purpose:** To be the immutable, long-term storage for all historical transaction data.
- **Structure:** Data is stored in Parquet format and organized using **Hive-style partitioning**.
    - **Path:** gs://[your-bucket]/.../year=YYYY/month=MM/day=DD/hour=HH/

3.7. Batch ETL Orchestration: Cloud Composer

- **Purpose:** To schedule and manage all batch processes in the system.
- **Implementation:** An Apache Airflow environment.
- **Orchestrated Tasks:**
    1. **Producer Job:** Triggers the data generation script on the producer VM.
    2. **Hourly ETL:** Manages the ephemeral Dataproc cluster (create, submit job, delete) for loading data from GCS to BigQuery.
    3. **Daily Firestore Refresh:** Manages the daily job to update Firestore from BigQuery.

3.8. Batch Processing: Dataproc

- **Purpose:** To perform large-scale data transformations.
- **Implementation:** Ephemeral Dataproc clusters running PySpark jobs.
- **Jobs:**
    - **Hourly ETL:** Reads partitioned Parquet files from GCS, performs dimension lookups against BigQuery, and appends enriched data to the Fact_Transactions table.
    - **Daily Firestore Refresh:** Reads data from BigQuery (via a GCS export) and writes updates to Firestore.

3.9. Data Warehouse: BigQuery

- **Purpose:** To serve as the central analytical repository, storing historical data in a structured **Star Schema**.
- **Schema Overview:** Consists of one central fact table (Fact_Transactions) and several dimension tables.

3.10. Business Intelligence: Looker Studio

- **Purpose:** To provide a user-friendly interface for analysts to connect to BigQuery, create dashboards on fraud trends, and generate reports.

3.11. Monitoring: Cloud's operations suite

- **Purpose:** To provide observability into the health and performance of the entire system.
- **Key Metrics Monitored:** Pub/Sub message backlog, Dataflow system latency, Cloud Run request errors, and Cloud Composer DAG success rates.

Chapter2: Real-Time Fraud Detection Pipeline

1. Introduction

In the modern digital economy, the velocity and volume of electronic transactions have created a fertile ground for financial fraud. Malicious actors continuously devise sophisticated methods to exploit vulnerabilities, resulting in significant financial losses for both consumers and institutions. The critical challenge lies in the speed of these transactions; post-facto detection is inadequate as it fails to prevent the initial loss. Therefore, there is a pressing need for fraud detection systems that operate in real-time, capable of identifying and flagging potentially fraudulent activity within the seconds between a transaction's initiation and its final approval.

Developing such a system presents several significant technical challenges. Firstly, the system must operate with extremely low latency, as any delay in processing could negatively impact user experience or fail to prevent the fraudulent act. Secondly, it must demonstrate high throughput, reliably processing a massive and often unpredictable stream of events without failure or degradation. Finally, raw transaction data is rarely sufficient for accurate detection. The system must address data complexity by enriching incoming transactions with a rich set of historical and contextual features, such as user behavior patterns, merchant profiles, and geographical data, which are essential for a high-fidelity machine learning model to make an accurate assessment.

1.1 Pipeline Objective

This report details the design and implementation of the central streaming component of an end-to-end fraud detection system. This pipeline serves as the core data processing engine,

engineered to meet the challenges of real-time fraud detection. The primary objectives of this component are to:

- Ingest and Process Events in Real-Time: Reliably consume a continuous stream of raw transaction events from a Google Cloud Pub/Sub topic.
- Enrich Data for Context: Augment each transaction with relevant historical data for both the user and the merchant by performing low-latency lookups against a Firestore database.
- Facilitate Machine Learning Scoring: Prepare the enriched data and dispatch it to a dedicated machine learning model endpoint to obtain a real-time fraud score.
- Decouple Processing from Action: Publish the scored transaction to a second Pub/Sub topic, effectively decoupling the core data pipeline from the downstream business logic that takes real-time action (e.g., blocking the transaction, sending an alert).
- Archive Data for Analytics: Persist the fully enriched and scored transactions to Google Cloud Storage in an efficient, partitioned Parquet format, creating a durable and optimized data source for batch analytics, model retraining, and auditing purposes.

1.2 Pipeline Scope

This part of the report focuses specifically on the streaming data pipeline, which is the main and most critical part of the larger fraud detection architecture. The scope of this system begins when a raw transaction message is published to the input Pub/Sub topic and ends when the processed data is delivered to its two destinations.

The boundaries of this pipeline are therefore defined as follows:

- Input: A stream of transaction messages from a designated input Pub/Sub topic.
- Core Responsibilities: Parsing, validating, enriching, and scoring the transactions.
- Outputs:
    - A real-time stream of scored transactions published to an output Pub/Sub topic.
    - Windowed, time-partitioned batches of scored transactions written as Parquet files to a Google Cloud Storage bucket.

The implementation of the upstream data source, the internal logic of the machine learning model, and the downstream consumer service that subscribes to the final output topic (e.g., a Cloud Run service that triggers alerts) are considered external to this specific pipeline component.

1.3 Technology Stack

The selection of technologies was driven by the requirements for scalability, performance, and maintainability within a serverless, cloud-native ecosystem. The core components of the solution are built on the Google Cloud Platform (GCP):

- Apache Beam on Google Cloud Dataflow: Apache Beam was chosen as the programming model for its powerful, unified framework for defining both streaming and batch data processing pipelines. Google Cloud Dataflow serves as the serverless, fully managed runner for the Beam pipeline, providing automatic scaling of resources based on workload, which is critical for handling fluctuating transaction volumes cost-effectively.
- Google Cloud Pub/Sub: This service is used as the messaging backbone of the system for both data ingestion and output. Its ability to provide scalable, durable, and asynchronous message delivery makes it ideal for decoupling the streaming pipeline from the external data producers and consumers, enhancing the overall resilience and flexibility of the architecture.
- Google Cloud Firestore: Selected as the database for data enrichment, Firestore offers extremely low-latency reads and writes for individual documents. Its NoSQL, document-oriented model is perfectly suited for the rapid key-value lookups required to fetch user and merchant contextual data without introducing significant processing delays.
- Google Cloud Storage (GCS): GCS is used for the long-term, cost-effective archival of processed data. The choice to store data in the columnar Parquet format on GCS creates an optimized data lakehouse foundation that can be directly and efficiently queried by other GCP services like BigQuery.
- Google Cloud Run: While external to the pipeline itself, the architecture is designed for the output Pub/Sub topic to trigger a service hosted on Cloud Run. This serverless container platform is ideal for executing the stateless business logic for the real-time action, as it can scale to zero when idle and scale up near-instantly to handle incoming events.

2. System Architecture and Design Philosophy

2.1 High-Level Pipeline Overview

The real-time fraud detection pipeline is designed as a linear sequence of data transformation stages, orchestrated by Apache Beam and executed on Google Cloud Dataflow. The architecture prioritizes resilience, scalability, and maintainability through a "smart-client, simple-server" approach, where the pipeline orchestrates calls to specialized, independent services for tasks like data enrichment and machine learning inference.

A conceptual diagram of the pipeline's data flow would illustrate the following key stages:

1. Ingestion: The process begins with the ReadFromPubSub transform, which ingests a continuous stream of raw transaction events from the input Pub/Sub topic.
2. Parsing and Validation: Each incoming message, a JSON string encoded in bytes, is immediately passed to the ParseAndTimestampDoFn. This transform is responsible for decoding the message, validating its structure, and extracting the event timestamp. Messages that fail this initial validation are not discarded; instead, they are routed to a dedicated dead-letter queue (DLQ) path on Google Cloud Storage (GCS) for later analysis, ensuring the main pipeline is not halted by malformed data.
3. Enrichment and ML Scoring: Successfully parsed transactions are batched and sent to the PredictFraudBatchDoFn. This is the core intelligence stage of the pipeline, performing two critical functions:
    a. Data Enrichment: It makes a batch-read request to a Firestore database to retrieve pre-computed features and historical context for the users and merchants involved in the transactions.
    b. ML Inference: It then sends the enriched batch of transactions via an HTTP POST request to a decoupled machine learning model endpoint.
    c. Failed batches from this stage (due to an unresponsive model server or missing data in Firestore) are routed to a separate processing DLQ on GCS.
4. Dual-Output Destination: Upon receiving a successful response from the model, the now fully scored and enriched transactions are sent down two parallel paths:
    a. Real-Time Action Path: The data is formatted and published to a second, distinct Pub/Sub topic. This topic serves as an asynchronous handoff to a downstream service responsible for executing business logic (e.g., triggering an alert).
    b. Analytics Archival Path: The data is grouped into fixed time windows (e.g., every two minutes). At the end of each window, the data is written to Google Cloud Storage in a highly optimized, partitioned Parquet format.

This dual-output design effectively separates the real-time operational path from the long-term analytical path, allowing each to be optimized independently.

2.2 Core Principle: Decoupling for Scalability and Resilience

A fundamental design philosophy of this pipeline is the strict decoupling of its core data transformation logic from external services like databases, machine learning models, and business action handlers. This separation is the key to achieving a system that is not only scalable and resilient but also highly maintainable. This principle is realized in three key areas.

2.2.1 Decoupling Data Enrichment

The process of enriching raw transaction data with historical context is externalized to Google Cloud Firestore, a high-performance NoSQL database.

- Mechanism: Rather than embedding user and merchant histories within the pipeline's state, the PredictFraudBatchDoFn queries Firestore in real-time. It collects the unique user and merchant IDs from a batch of transactions and executes a single batch_read operation to efficiently retrieve all necessary contextual data.
- Benefits:
    - Independent Scaling: Firestore is a serverless database that scales its read/write capacity independently of the Dataflow pipeline. If the system experiences a surge in transactions requiring many database lookups, Firestore can absorb this load without requiring the entire Dataflow job to be scaled up, leading to more efficient resource utilization.
    - Data Consistency and Centralization: By using Firestore as a central repository for user and merchant features, we ensure that all services in the ecosystem (including potentially other microservices outside this pipeline) access the same source of truth. This prevents data silos and inconsistencies.
    - Maintainability: The enrichment logic is neatly encapsulated. If new features need to be added (e.g., a user's 90-day transaction history), only the Firestore data model and the query logic within the PredictFraudBatchDoFn need to be updated, without requiring a complete architectural redesign of the pipeline.

2.2.2 Decoupling Machine Learning Inference

The machine learning model is not part of the Dataflow pipeline itself. Instead, it is treated as an independent, external service with a well-defined API.

- Mechanism: The pipeline's PredictFraudBatchDoFn acts as a client to a model server. After enriching a batch of transactions, it sends them as a JSON payload in an HTTP request to a predefined model endpoint URL. It then waits for a response containing the fraud predictions.
- Benefits:
    - Model Agnosticism: This is the most powerful advantage. The data science team can develop and deploy their fraud detection model using any framework or language (e.g., TensorFlow, PyTorch, Scikit-learn) best suited for the task. The pipeline is completely agnostic to the model's internal workings; it only adheres to the API contract. A new, improved version of the model can be deployed at any time without requiring any changes to the data pipeline code.

- Technological Flexibility: This decoupling frees the data engineering team to focus on efficient data transport and the data science team to focus on model development. The pipeline is written in Python using the Beam SDK, while the model could be served via a high-performance C++ or Go server, allowing each component to use the optimal technology.
- Isolated Resource Management: Model serving often has different resource requirements (e.g., GPUs, high memory) than data processing. By deploying the model on a separate service like Vertex AI Endpoints or Cloud Run, its resources can be scaled and configured independently from the Dataflow workers, leading to significant cost and performance optimizations.

2.2.3 Decoupling the Real-Time Action

The final step in the real-time path is not to execute an action, but to hand off the responsibility to another service via a second Pub/Sub topic.

- Mechanism: After a transaction is scored, the pipeline publishes the result as a message to an output Pub/Sub topic. This is a "fire-and-forget" operation. A separate, stateless service (e.g., a Cloud Run instance) is configured with a push subscription to this topic, which triggers it to execute the necessary business logic for each incoming message.
- Benefits:
  - Asynchronous Processing and Ultimate Resilience: This is the most critical resilience pattern in the architecture. If the downstream action-taking service fails, becomes slow, or is taken down for maintenance, the core fraud detection pipeline is completely unaffected. Pub/Sub acts as a durable buffer, retaining the scored transaction messages until the consumer service is healthy again. Pub/Sub's built-in retry mechanisms ensure that the action will eventually be triggered, preventing any data loss due to downstream failures.
  - Simplified Logic and Separation of Concerns: The Dataflow pipeline has a single responsibility: process data. The business logic of what to do with that data (e.g., *if the score is above 0.95, block the card and create a support ticket*) is complex and subject to frequent changes. Isolating this logic in a separate microservice makes it far easier to develop, test, and update without the risk of breaking the complex data processing pipeline.
  - Independent Scalability of Actions: The consumer service, hosted on a serverless platform like Cloud Run, can scale based on the number of actionable events, not the total transaction volume. It can scale to zero if no fraudulent transactions are detected, providing an extremely cost-effective solution, and then scale up instantly to handle a sudden burst of high-risk events.

3. Performance Optimization with Event Batching

While the decoupled architecture provides scalability and resilience, achieving the low-latency and high-throughput required for real-time fraud detection necessitates further optimization within the pipeline itself. A naive approach of processing each transaction individually would create significant performance bottlenecks, particularly when interacting with external services like the machine learning model. This pipeline addresses this challenge through a deliberate and highly effective strategy of micro-batching events.

3.1 The Role of Micro-Batching for ML Inference

The single most performance-critical interaction in the pipeline is the call to the machine learning model endpoint. Making a separate HTTP request for every single transaction would be prohibitively slow and inefficient. The pipeline mitigates this by grouping transactions into small batches before sending them for prediction.

- Mechanism: This is achieved using Apache Beam's built-in GroupIntoBatches transform. This powerful transform collects individual elements and bundles them into a list (a batch) based on configurable parameters. The pipeline is designed to group transactions after they have been parsed and before they enter the PredictFraudBatchDoFn transform.
- Implementation: The batching behavior is controlled by two key command-line parameters, allowing for flexible tuning of the pipeline's performance characteristics:
    - --ml_batch_size: This integer parameter defines the maximum number of elements to include in a single batch.
    - --allowed_wait_time_seconds: This parameter sets a maximum time limit for the pipeline to wait before sending a batch, even if it has not yet reached the ml_batch_size. This is crucial for ensuring low latency, as it prevents a few trailing transactions from being held up for an extended period during times of low traffic.

The interplay between these two parameters allows an operator to balance the trade-off between latency and throughput. A larger batch size improves throughput but may slightly increase latency, while a shorter wait time minimizes latency at the cost of potentially smaller, less efficient batches.

3.2 Efficiency and Performance Gains

The implementation of micro-batching before the ML inference step yields substantial improvements in overall system performance and efficiency.

- Reduced Network Overhead: Batching dramatically decreases the number of network round-trips to the model server. Instead of establishing a new HTTP connection and sending headers for each transaction, a single request can carry a payload of 50 or 100 transactions. This significantly reduces the cumulative impact of network latency and request/response overhead, which is a major contributor to processing time in distributed systems.
- Improved Model Throughput: Modern machine learning serving frameworks and hardware (including GPUs and TPUs) are highly optimized for vectorized operations. They can perform predictions on a batch of data far more efficiently than they can process a sequence of individual prediction requests. By sending a batch, the pipeline allows the model server to leverage these optimizations, leading to a much higher prediction throughput and better utilization of the serving infrastructure.
- Cost-Effectiveness: In a cloud environment, performance is directly tied to cost. By reducing the number of invocations on the model serving endpoint (e.g., a Vertex AI Endpoint or a Cloud Run service), batching directly lowers operational costs. Furthermore, by improving the efficiency of the model server, it may be possible to provision less powerful (and thus cheaper) hardware to handle the same transaction volume.

3.3 Ensuring Scalability with Fan-out Keys

A potential risk with any grouping operation in a large-scale parallel processing system is the creation of a "hot key." If all elements were given the same key before being passed to GroupIntoBatches, a single Dataflow worker might become overwhelmed trying to process all the elements, creating a bottleneck that negates the benefits of parallelization.

- Mechanism: To prevent this, the pipeline employs a "fan-out" strategy immediately before the batching step. This is implemented via the AddRandomKey transform, a simple DoFn that assigns a random integer key (e.g., from 0 to N-1) to each transaction. The GroupIntoBatches transform is then applied to these key-value pairs.
- Benefit: This technique ensures that the incoming stream of transactions is randomly distributed across a set of keys. As a result, the Dataflow runner can distribute the grouping workload across multiple workers, with each worker being responsible for batching the elements for a subset of the keys. This prevents any single worker from becoming a bottleneck and ensures that the batching stage can scale horizontally as the overall data volume increases. Once the batching is complete, the random key has served its purpose and can be discarded, leaving just the batch of transactions to be processed by the subsequent steps.

4. Data Persistence for Analytics and Auditing

While the primary operational goal of the pipeline is to facilitate real-time actions, a second, equally important objective is to create a durable, long-term repository of all processed transactions. This historical dataset is invaluable for a wide range of offline activities, including the training and validation of future machine learning models, business intelligence reporting, generating compliance reports, and in-depth forensic analysis of fraud patterns. This archival process runs in parallel to the real-time action path, ensuring that analytical requirements do not interfere with low-latency operational needs.

4.1 Windowing and Aggregation

To manage the unbounded nature of a continuous data stream, the pipeline groups transactions into finite, time-based collections using a windowing strategy. This is the first step in preparing the data for batch-oriented storage.

- Mechanism: The pipeline utilizes Apache Beam's FixedWindows transform. This function assigns each transaction to a window of a specific, non-overlapping duration based on its event timestamp. The duration of this window is a configurable parameter, --window_duration_seconds, allowing operators to define the granularity of the output files (e.g., creating a new set of files every two minutes).
- Handling Late Data: In any real-world streaming system, some events may arrive late due to network delays or issues at the source. The pipeline is configured with an allowed_lateness period, which gives these tardy records a grace period to be included in their correct time-based window even after the window has technically closed. The ACCUMULATION_MODE.DISCARDING setting ensures that when the window's results are written, only the new, late-arriving data is processed, preventing redundant writes and ensuring efficiency.

4.2 Partitioned Parquet Storage in GCS

The choice of storage format and structure is critical for optimizing the cost and performance of future analytical queries. The pipeline is designed to write data to Google Cloud Storage (GCS) in a highly efficient, query-friendly manner.

- Mechanism: At the conclusion of each window, the WriteWindowedBatchToGCS transform writes the collected batch of transactions to a file in GCS. The data is stored in the Apache Parquet format, a columnar storage file format that is the de facto standard for large-scale data analytics. The transaction_schema, defined with PyArrow, enforces a consistent and strongly-typed structure for all data written to storage.
- Benefits:

- Query Optimization with Hive-Style Partitioning: The key to this strategy is the directory structure used for output files. The pipeline automatically creates a Hive-style partitioned path based on the transaction's event timestamp: .../year=YYYY/month=MM/day=DD/hour=HH/. This structure is natively understood by modern query engines like Google BigQuery, Apache Spark, and Presto. When a query with a time filter is executed (e.g., "analyze all transactions from the first week of July"), the engine can perform "partition pruning"—it skips reading all directories that do not match the filter, drastically reducing the amount of data scanned, which in turn lowers query costs and improves performance by orders of magnitude.
- Data Compression and Efficiency: The Parquet format offers excellent compression, significantly reducing the data's storage footprint on GCS and thereby lowering costs. Its columnar nature means that queries only need to read the specific columns required, rather than scanning entire rows, which further enhances query speed.

4.3 Parallelized and Fault-Tolerant Writes

A significant challenge in writing windowed data from a parallel system is managing concurrent writes. Multiple Dataflow workers may finish processing their portion of a window's data at the same time and attempt to write to the same output location. The pipeline design elegantly solves this potential for conflict.

- Mechanism: To enable safe, parallel writes, the pipeline employs a two-part strategy:
  - Fan-out Key: Before writing, the AddRandomKey transform is used again to distribute the windowed data across multiple workers, preventing a single worker from becoming a bottleneck for the GCS write operation.
  - Unique Filenames: The WriteWindowedBatchToGCS transform generates a universally unique identifier (UUID) for every file it creates (e.g., data-<uuid>.parquet). This ensures that even when multiple workers are writing to the exact same hourly partition directory simultaneously, each worker writes to its own unique file, completely avoiding any write conflicts or the need for complex file-locking mechanisms.
- Fault Tolerance: The write process is designed for resilience. The WriteWindowedBatchToGCS DoFn wraps its write logic in a try...except block. In the event of a failure—such as a transient network error or a permissions issue preventing a write to GCS—the batch of records is not discarded. Instead, it is captured and emitted to a tagged side output (failed_gcs_write_tag). This failed batch can then be routed to a separate dead-letter queue location, ensuring that no processed data is lost due to sink errors and allowing for later inspection and manual reprocessing.

5. Robustness and Error Handling

In a distributed, high-throughput streaming system, failures are not an exception; they are an expected part of normal operation. Sources of error can range from malformed input data and network timeouts to bugs in downstream services or transient infrastructure issues. A robust pipeline must be designed with the explicit goal of handling these failures gracefully, ensuring both the continuous operation of the service and the prevention of data loss. This pipeline achieves this resilience through a comprehensive, multi-stage Dead-Letter Queue (DLQ) strategy, which isolates problematic data at every critical step of the process.

5.1 Multi-Stage Dead-Letter Queue (DLQ) Strategy

Instead of a single, monolithic error-handling block, the pipeline implements a series of targeted DLQs. Each one is designed to capture a specific class of failure, providing granular insight into the health of the system and preserving the failed data in its respective context for debugging and reprocessing.

5.1.1 Parsing Dead-Letter Queue

The first line of defense is at the ingestion point. The pipeline must be protected from "poison pill" messages—malformed records that cannot be parsed and could otherwise cause the entire pipeline to crash repeatedly.

- Mechanism: The ParseAndTimestampDoFn transform wraps its JSON decoding and timestamp extraction logic in a try...except block. If a message cannot be successfully parsed into a valid JSON object or if it lacks a valid timestamp, the exception is caught. Instead of failing, the transform emits the original, raw byte-string message to a side output tagged as failed_parses.
- Destination: This stream of failed messages is then written directly to a dedicated GCS path specified by the --parse_dlq_path parameter.
- Significance: This mechanism immediately isolates data quality issues originating from upstream producers. It allows the main pipeline to continue processing valid transactions without interruption while preserving the problematic messages for offline analysis.

5.1.2 Processing Dead-Letter Queue

The most complex stage of the pipeline—enrichment and ML inference—is also the most susceptible to errors involving external dependencies. A dedicated DLQ is in place to handle failures at this critical juncture.

- Mechanism: The PredictFraudBatchDoFn uses extensive error handling to manage a variety of potential failures. It routes the entire failing batch of transactions to a side output tagged as failed_batches under several conditions:
    - Missing Enrichment Data: If a transaction's required user or merchant data cannot be found in Firestore.
    - ML Model Request Failure: If the HTTP request to the model endpoint fails due to network issues, timeouts, or server-side errors (e.g., 5xx status codes).
    - Model Validation Error: If the model server returns a 422 Unprocessable Entity status, indicating that the data, while syntactically correct, failed the model's validation logic.
    - Unexpected Errors: Any other unforeseen exception within the transform's logic.
- Destination: The captured batch, along with any relevant error details, is serialized to a JSON string and written to the GCS path specified by --processing_dlq_path.
- Significance: This isolates the pipeline from the unreliability of its external dependencies. A temporary outage of the model server or a missing record in Firestore will not halt the entire system. The pipeline continues to process other valid batches, and the failed data is preserved with full context for later investigation.

5.1.3 Formatting and GCS Write Dead-Letter Queues

The final steps of formatting the data and writing it to its destinations are also potential points of failure. The pipeline includes DLQs to ensure that successfully processed data is not lost at the very last moment.

- Mechanism: Both the FormatForPubSubDoFn and the WriteWindowedBatchToGCS transforms include try...except blocks. If a record cannot be converted to the final output schema, or if a network error prevents a Parquet file from being successfully written to GCS, the failing element or batch is emitted to a corresponding side output (failed_format_tag or failed_gcs_write_tag).
- Destination: These outputs are routed to a failure path within the GCS output directory.
- Significance: This provides end-to-end data integrity. It guarantees that valuable, scored data is not discarded simply because of an error in a sink operation, ensuring that every record that successfully passes the core processing logic can be recovered.

5.2 Benefits of the DLQ Approach

This granular, multi-stage DLQ strategy provides three core benefits that are essential for a production-grade streaming system:

- System Stability: The primary benefit is the isolation of failures. By catching, tagging, and routing problematic data to side outputs, the pipeline ensures that the main data path remains clear for valid transactions. This prevents cascading failures and allows the system to maintain high availability and performance even when encountering partial data or dependency issues.
- Data-Loss Prevention: The fundamental principle of this design is to never discard data due to a processing error. Every message that enters the pipeline is accounted for. If it cannot reach the final destination through the "happy path," it is persisted in a DLQ, ensuring it is available for manual review, debugging, or automated reprocessing.
- Enhanced Observability and Debugging: The separated DLQ paths serve as a powerful diagnostic tool. A sudden influx of files in the parse_dlq immediately points to an upstream data quality problem. Files appearing in the processing_dlq suggest issues with the ML model or Firestore. This clear separation of error types drastically reduces the time required to identify the root cause of a problem, enabling developers and operators to debug and **resolve issues more efficiently.**

6. Formulate a Hypothesis

After thoroughly researching the problem, the next step is to formulate a hypothesis. This hypothesis should propose a

potential solution to the identified problem.

Hypothesis: "A real-time fraud detection system that uses machine learning models, big data processing frameworks,

and scalable cloud infrastructure will significantly improve the accuracy and speed of fraud detection, reduce false

positives, and adapt to emerging fraud patterns."

Key Assumptions of the Hypothesis:

1. Machine learning models trained on historical transaction data can identify complex fraud patterns better than

static rule-based systems.

2. Big data technologies such as Apache Kafka and Apache Spark can handle the large volume of transactions

and provide real-time analysis.

3. The system will be able to automatically scale as transaction volumes increase, ensuring consistent

performance.

4. The system will be able to reduce false positives by learning from the actual fraud cases in the training data.

This hypothesis establishes the basis for the project's experimental phase.

6. 1. Conduct an Experiment

This phase involves building and deploying the Real-Time Fraud Detection System to test the hypothesis. The goal is

to implement the system in a real-world environment and collect data to evaluate its performance.

Step 1: Data Ingestion

• Set up script to generate transactions on VM to ingest transaction data from source , online payment gateways and create topic in pub/sub to fetch this data from source

Step 2: Preprocessing

• Google Dataflow is used to clean and preprocess the transaction data. This involves removing duplicates,

handling missing values, and formatting the data so that it can be used in machine learning models.

- For example, transactions may be filtered to remove those with incomplete information, or certain features (such as transaction amount and location) may be transformed to standardize values.

Step 3: Machine Learning Model

• A pre-trained machine learning model (using algorithms such as random forests, gradient boosting, or deep learning) is applied to detect fraudulent activities.

- The model is trained on historical transaction data, where each transaction is labeled as "fraud" or "not fraud."
- Once trained, the model is deployed to predict the likelihood of fraud for each incoming transaction in real time.
- Key features considered by the model may include transaction amount, location, merchant type, time of day, and user history.

Step 4: Data Storage

• The preprocessed and processed data are stored in Google Cloud Storage (GCS) as a data lake, allowing for

scalable and secure storage.

- Google BigQuery is used to store summarized and aggregated data, enabling real-time querying and reporting.

Step 5: Experimentation

• The system continuously monitors and processes transaction data to detect fraud. The key metrics evaluated

during the experiment include:

- Accuracy: The percentage of fraud cases correctly identified by the system.
- False Positive Rate: The number of legitimate transactions that were incorrectly flagged as fraud.
- Real-time Performance: The system's ability to detect fraud within milliseconds of the transaction being processed.

6. 2. Reach a Conclusion

After conducting the experiment, the final step is to analyze the results and determine whether the system meets the

expectations set forth in the hypothesis.

Key Questions to Answer:

1. Has the accuracy improved?

- Compare the fraud detection accuracy of the new system with the baseline accuracy of traditional systems. The goal is to achieve higher accuracy, meaning that more fraud cases are correctly detected.

2. Are false positives reduced?

- Examine whether the machine learning model has successfully reduced the number of false positives, ensuring that fewer legitimate transactions are incorrectly flagged as fraud.

3. Does the system detect fraud in real time?

- Measure the latency between the time a transaction occurs and the time the system flags it as fraud.
- The system must operate within strict time constraints to ensure that fraud can be prevented before the transaction is completed.

4. Can the system scale?

- Assess whether the system can handle increasing transaction volumes without degrading performance.
- The use of cloud infrastructure and distributed processing should allow the system to scale automatically.

Conclusion: Based on the performance metrics, the team can either confirm the hypothesis, adjust the system to improve its performance, or propose new approaches for future experiments. If the hypothesis is confirmed, the RealTime Fraud Detection System will be deployed in a production environment, providing financial institutions with a

scalable, accurate, and real-time fraud detection solution

7. I. High-Level System Architecture

This part of the project implements the core machine learning module for the **real-time fraud detection system**. It is not a monolithic application, but a specialized, self-contained component designed for seamless integration into a larger, **event-driven data pipeline** on **Google Cloud Platform (GCP)**. The architecture emphasizes modularity, portability, and clear data contracts, allowing it to reliably consume training data from a data warehouse and serve low-latency predictions to a real-time data processing framework.

The design is deliberately split into two distinct sub-components to handle the **asynchronous nature of model training** versus the **synchronous demands of real-time serving**.

A. Separation of Concerns: Decoupled Training and Inference

A foundational design principle is the strict separation of the machine learning system into two independently deployable services: the **training pipeline** and the **inference service**.

- **Training Pipeline:** This is the **batch processing** component of the system. Its purpose is to be triggered on a periodic schedule (e.g., weekly) by an orchestrator like Apache Airflow or Cloud Composer. It performs the computationally intensive task of model creation by ingesting a static dataset from a source like **BigQuery**, training a new model, performing hyperparameter tuning, and publishing the resulting model artifact to Google Cloud Storage (GCS).
- **Inference Service:** This is the **real-time, transactional** component. It is a continuously running, stateless web service deployed on **Google Cloud Run**. Its sole responsibility is to provide on-demand fraud predictions with minimal latency. It is designed for high availability and scalability, ready to be invoked by a real-time **Apache Beam** pipeline running on **Google Cloud Dataflow**.

Integration Impact: This decoupled architecture is essential for operational stability. The entire real-time data pipeline does not need to be paused or redeployed to retrain the model. The batch training workflow can execute independently without impacting the uptime of the inference service, which continues to serve predictions using the current production model.

B. Containerization for Portability and Orchestration

Both the training and inference components are containerized using **Docker**. This encapsulates them into portable, reproducible execution units that can be managed by GCP's various container services.

- **Training Container (machine_learning/training_pipeline/Dockerfile):** This container image packages the entire model training workflow. An orchestrator can launch this container as a discrete task (e.g., a Vertex AI Custom Job), passing in configuration via environment variables to execute a training run.
- **Inference Container (machine_learning/inference_service/Dockerfile):** This container image packages the lightweight prediction API. This is the image that is deployed directly to **Google Cloud Run**, creating a scalable, serverless HTTPS endpoint.

Integration Impact: Containerization abstracts the ML module's environment away from the rest of the cloud infrastructure. The orchestrator only needs to know how to run the container and pass environment variables, making the entire system robust and simplifying dependency management across the project.

C. Microservice-Based Inference as a Real-Time Enrichment Step

The inference service is designed as a microservice that functions as a **real-time data enrichment step** within the broader fraud detection pipeline.

Integration Impact: In this system, the data flow is as follows:

1. A stream of financial transactions arrives in real-time, for example, via a **Google Cloud Pub/Sub** topic.
2. An **Apache Beam** pipeline, deployed on **Dataflow**, subscribes to this topic and processes each transaction event.
3. For each transaction, the Beam pipeline makes a synchronous HTTP request to the inference service's **/predict** endpoint hosted on **Cloud Run**.
4. The inference service immediately returns a JSON response containing the fraud score.
5. The Beam pipeline then **"enriches"** the original transaction data with this score. This enriched data can then be routed to its next destination, such as being written to a data lake on GCS for logging, loaded into BigQuery for analytics, or used to trigger real-time alerts.

The use of a standard **RESTful API (FastAPI)** provides a simple, language-agnostic integration point that decouples the data processing logic in Beam from the machine learning model itself.

D. Cloud-Native Design with GCS as the Handoff Layer

Google Cloud Storage (GCS) is used as the persistent handoff layer, or "data contract," between the training and inference components.

- **Upstream to Training:** The training pipeline is configured to read its input DATA_PATH from a designated GCS bucket, where an upstream ETL process places the prepared training dataset.
- **Training to Inference:** After a successful run, the training_pipeline writes the serialized model.joblib file to a well-known GCS path in the ARTIFACTS_DIR. This artifact is the key output of the training step.
- **Model to Live Service:** The inference_service on **Cloud Run** is configured at startup to load its model from this same GCS MODEL_PATH. This decouples the service deployment from model training; the Cloud Run service can be scaled or restarted at any time and will always fetch the designated production model from GCS.

E. Programmatic Control via Configuration Management

The extensive use of environment variables allows the behavior of each component to be controlled programmatically by an external system or developer.

Integration Impact: An orchestration tool can trigger a training run and dynamically set the DATA_PATH. For deployment, a CI/CD pipeline can deploy a new version of the inference_service to **Cloud Run** and configure its MODEL_PATH environment variable to point to a new model file. This makes advanced deployment strategies like A/B testing or canary rollouts of new models straightforward to implement at the infrastructure level.

II. The Training Pipeline

The training pipeline is an automated, end-to-end workflow designed to produce a production-ready model artifact. It is containerized and executed as a batch process, with all experiments and results tracked for full reproducibility.

A. Workflow Orchestration (main.py)

The main.py script serves as the entry point, orchestrating each stage of the pipeline in a predefined sequence: Initialization, Data Preparation, Hyperparameter Optimization (HPO) & Training, Evaluation, and Model Export. This ensures a reproducible and linear workflow where the output of each step feeds into the next.

B. Data Management and Preparation (data_preparation.py)

The pipeline begins by preparing the dataset with a strong emphasis on production-realism and preventing data leakage.

Flexible Data Ingestion: The read_data function ingests data from either local file systems or GCS, capable of processing a single CSV file or aggregating all .csv files within a directory.

Time-Based Splitting: To prevent data leakage and accurately simulate a production scenario, the dataset is split chronologically. The split_dataset_based_on_time function sorts all

transactions by timestamp before partitioning the data. This ensures the model is trained on past data to predict future outcomes.

Feature Identification: After splitting, the feature set is analyzed to automatically identify numerical and categorical columns for the preprocessing stage.

C. Preprocessing and Feature Engineering (training.py)

All feature transformations are encapsulated within a scikit-learn Pipeline object, guaranteeing that the exact same steps are applied during both training and inference.

- **ColumnTransformer:** A ColumnTransformer applies different transformation chains to different types of columns.
- **Numerical Features:** Missing values are filled using the column's median (SimpleImputer), and features are scaled to have zero mean and unit variance (StandardScaler).
- **Categorical Features:** Missing values are filled with the most frequent value (SimpleImputer). The features are then converted to a numerical format using OneHotEncoder. Crucially, this encoder is configured with handle_unknown="ignore", a vital parameter for production that allows the model to handle new, unseen categories at inference time without crashing.

D. Model Training and Hyperparameter Optimization (training.py)

The core of the pipeline involves training an XGBoost model with an optimization strategy tailored to the business problem of fraud detection.

- **Primary Evaluation Metric: Recall:** The optimization process is explicitly configured to maximize **recall**. This is the most critical design choice for this problem. In fraud detection, the business cost of a **False Negative** (a fraudulent transaction missed by the model) is direct financial loss. The cost of a **False Positive** (a legitimate transaction incorrectly flagged) is a minor user inconvenience. Therefore, the primary goal is to minimize false negatives, which is mathematically equivalent to maximizing recall. The _objective function in the HPO loop is designed specifically to guide the model search towards this goal.
- **Algorithm:** The model is an XGBClassifier, a powerful and efficient gradient boosting implementation.

- **Hyperparameter Optimization (HPO):** The **Optuna** library is used to automatically find the best hyperparameters for the model.
- **Class Imbalance Handling:** The pipeline automatically calculates the scale_pos_weight and passes it to the XGBClassifier. This parameter increases the importance of the minority class (fraud) during training, forcing the model to pay more attention to correctly identifying these crucial instances.
- **Time-Series Cross-Validation:** Within the Optuna HPO loop, a TimeSeriesSplit cross-validator is used. This ensures that during each validation fold, the model is always trained on past data and validated on future data, maintaining chronological integrity.

E. Model Evaluation (evaluation.py)

After training, the final model's performance is measured on the unseen test set. The evaluate_model function calculates a full suite of metrics, including a detailed classification_report, ROC AUC, and Precision-Recall AUC. A confusion matrix is also generated and saved as an image, providing a clear visual representation of the model's performance.

F. Model Export and Versioning (model_exporter.py)

The final step is to persist the entire trained pipeline—including all preprocessing steps and the trained model—into a single model.joblib file. This single-artifact approach is robust and prevents training-serving skew. This artifact is then logged to the **Weights & Biases Artifact Registry**, which versions the model and links it directly to the experiment run that produced it.

III. The Inference Service (Real-Time API)

The inference service is a lightweight, high-performance FastAPI application, containerized and deployed on **Google Cloud Run**. Its sole responsibility is to serve real-time fraud predictions to the Apache Beam pipeline with minimal latency.

A. API Design and Technology (main.py)

The service is built using production-ready Python technologies chosen for performance and reliability. **FastAPI** provides a high-speed web framework, while the **Uvicorn** ASGI server runs the application in production. The service exposes two critical endpoints:

- **Health Check (/health):** A vital endpoint used by **Cloud Run** for liveness probes to ensure the service is running and the model has been successfully loaded into memory.
- **Prediction (/predict):** The core POST endpoint that accepts a batch of transactions and returns the corresponding fraud predictions.

B. Data Contracts and Validation (schemas.py)

The service enforces a strict data contract using Pydantic models to ensure data integrity.

- **TransactionInput:** Defines the detailed schema for a single transaction, including all features and their data types.
- **PredictionInput:** Defines the schema for the request body, which contains a list of TransactionInput objects. This design explicitly enables **batch prediction**, allowing the **Apache Beam** pipeline to group multiple transactions into a single, efficient API call, reducing network overhead and leveraging vectorized computation.
- **Output Schemas:** Define the structure of the API's response, ensuring a consistent, well-formed JSON object is returned.

C. Prediction Logic and Execution (predictor.py, main.py)

The prediction logic is designed for efficiency and consistency with the training process.

- **Model Loading on Startup:** The service uses FastAPI's lifespan context manager to load the model from GCS **only once** when the Cloud Run container starts. The loaded pipeline is stored globally, making it immediately available for all subsequent prediction requests.
- **Dynamic Model Source (utils.py):** The load_model function can load a model from either a local path (for testing) or a GCS URI. In production on **Cloud Run**, it fetches the official model artifact directly from cloud storage.
- **Consistent Preprocessing (predictor.py):** The make_predictions function passes the incoming DataFrame directly to the loaded model_pipeline. Because the entire pipeline was saved as a single artifact, this one call automatically applies the exact same preprocessing steps used during training, guaranteeing consistency and preventing training-serving skew.

D. Error Management (main.py)

The service includes graceful error handling to ensure stability. It returns an HTTP 503 Service Unavailable status if a request arrives before the model is loaded and an HTTP 500 Internal Server Error if any unexpected error occurs during prediction, preventing a single bad request from crashing the service instance.

IV. MLOps and Tooling

The project was developed with a strong emphasis on **MLOps** principles to ensure the entire ML lifecycle is reproducible, version-controlled, and automated.

A. Experiment Tracking with Weights & Biases

All training runs are logged to **Weights & Biases (W&B)**, creating a centralized, auditable record of every model's configuration, code version, and performance. The deep integration with **Optuna** automatically logs every HPO trial, and final evaluation metrics and artifacts (like the confusion matrix) are explicitly logged, providing complete transparency and traceability.

B. Model Registry and Artifact Management

The project leverages the **W&B Artifact Registry** to version-control the trained model.joblib artifacts. W&B automatically links each model to the exact experiment run that produced it, creating an unbreakable chain of lineage. This provides a single source of truth for deployment, enabling a CI/CD pipeline to programmatically fetch a specific model version and deploy it as a new **Cloud Run** revision.

C. Foundational Tooling for Reproducibility

- **Containerization (Dockerfile):** Two separate Dockerfiles, one for training and one for inference, precisely define the execution environment for each component.
- **Dependency Management (requirements.txt):** Each service has its own requirements.txt file with pinned library versions.

This combination guarantees a stable and reproducible environment, solving the "it works on my machine" problem and preventing failures from unexpected dependency updates.

D. Configuration Management for Portability

The project relies entirely on environment variables for configuration, adhering to the Twelve-Factor App methodology. This decouples the application from its configuration, allowing the

same container image to be promoted through different environments (development, staging, production) without any code changes. This is essential for building automated, secure, and flexible CI/CD pipelines.

You are right. Let's transform that summary into a fully detailed technical breakdown, drilling down into the specific implementation details and anchoring every concept to its corresponding code component. This version is designed to be the comprehensive, unabridged technical chapter of your report.

8. Data Model: BigQuery Data Warehouse

8.1. Schema Overview

The data warehouse is designed using a classic Star Schema. This model separates the numerical measurements of business events (facts) from their descriptive context (dimensions), providing an intuitive and high-performance structure for analysis.

The data pipeline was built to ingest raw transactional data from **Google Cloud Storage (GCS)**, process it using **PySpark on Google Dataproc**, and load the cleaned and enriched data into **BigQuery**.

- **Source**: The data originated from a batch job simulating real-time financial transactions in JSON format.
- **Processing Layer**:
    - Implemented with **PySpark**, the ETL logic included data validation, cleansing (e.g., handling missing/null values), schema enforcement, and column transformations.
    - Engineered new features like transaction type flags, merchant category codes, and card anonymization.

- **Orchestration**: The pipeline was orchestrated using **cloud composer** ensuring scheduled, monitored, and repeatable runs with clear logging and retry logic.

8. 2. Dimensional Modeling

Due to the lack of a live **OLTP system** (such as a banking core), I made specific design decisions:

- **Fact Table**:
    - The core of the schema is a **FactTransaction** table that holds transactional metrics like amount, timestamp, card_id, merchant_id, location, and is_fraud.
    - This table is updated incrementally with every pipeline run.
- **Dimension Tables**:
    - **Static Dimensions**: Since no user or merchant update stream was available, I assumed slowly changing dimensions were unnecessary.
    - Dimensions such as **dim_user**, **dim_card**, **dim_merchant**, and **dim_time** were pre-loaded from static CSVs or JSON files and stored in BigQuery.
    - These dimensions contain denormalized descriptive data to support rich analytical queries.

8. 3. No SCD (Slowly Changing Dimensions)

Given the absence of a dynamic source system for users or merchants, **SCD logic was excluded** from the pipeline. Instead:

- Dimensions were **loaded once** and reused across pipeline runs.
- The design assumes that user attributes (e.g., age group, gender, account type) are static over the reporting period.

8. 4. Use Cases Enabled

This data model supported:

- Fraud detection analytics by analyzing user behavior over time.
- Aggregations and reports such as total spend per merchant, peak transaction hours, and card usage patterns.
- Drill-down dashboards for identifying anomalies in specific regions or merchants.

8. 5. Scalability & Cost Optimization

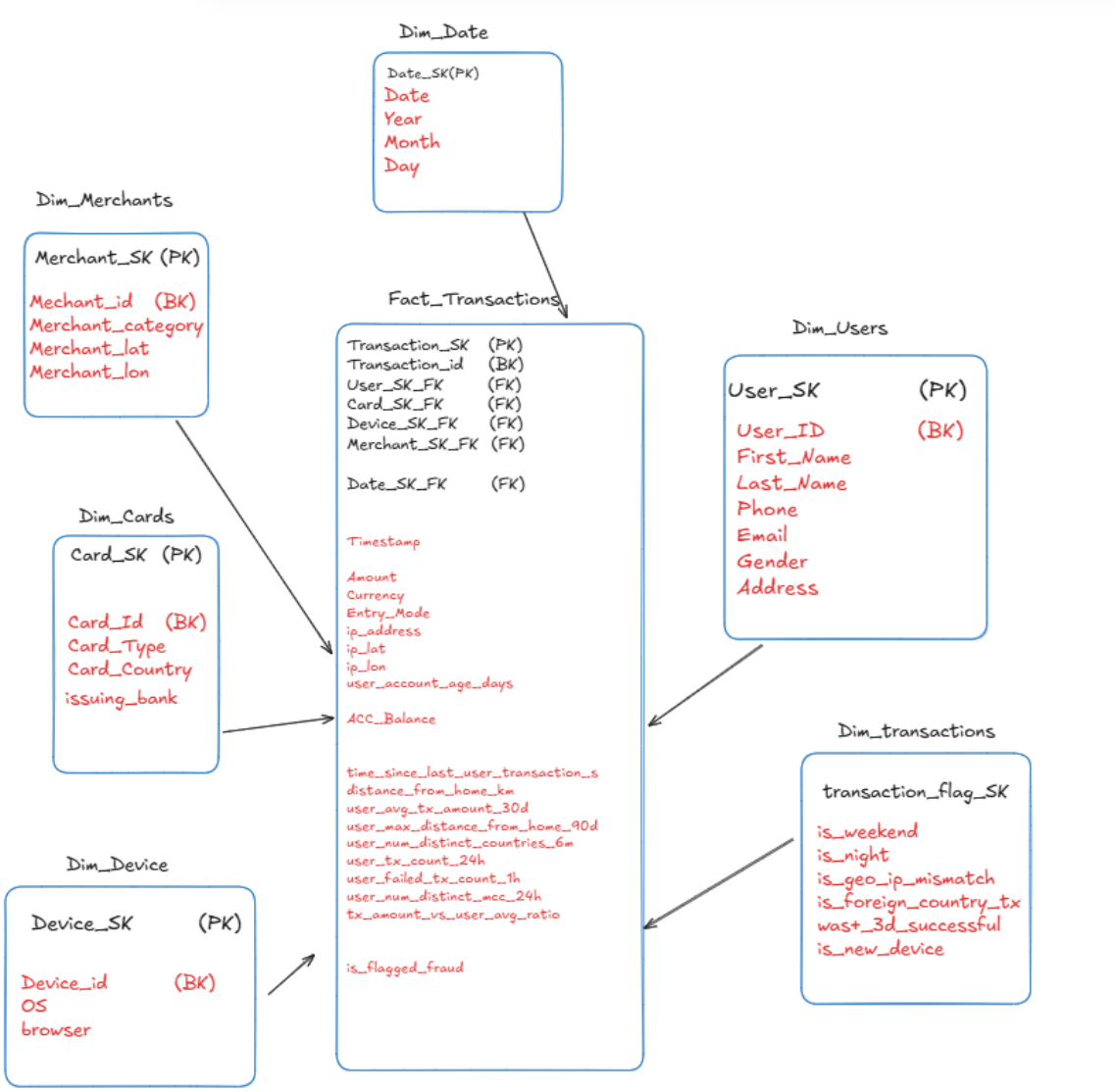
- Used **partitioning and clustering** in BigQuery to optimize query performance and reduce costs (e.g., partitioned by transaction_date, clustered by merchant_id and card_id).
- Ensured all transformations happen outside BigQuery to minimize compute costs on the warehouse.

8.6. Dimension Tables

- **dim_Users:** Stores unique information about each user.
- **dim_Cards:** Stores unique information about each payment card.
- **dim_device:** Stores unique information about each device.
- **dim_Merchants:** Stores unique information about each merchant.
- **dim_Date:** A pre-populated table describing each day (e.g., day of week, month, year, holiday).
- **Dim_transactions:** stores flags for each transaction

8.7. Fact Table

- **Fact_Transactions:** The central table in the schema. Each row represents a single transaction event. It contains foreign keys to all dimension tables and a wide range of numerical measures and pre-calculated features for fraud analysis.



8.9. Data modeling (Star Schema)

9. System Architecture and Orchestration

This section provides a high-level overview of the orchestration component, which serves as the backbone for the real-time fraud detection pipeline. It begins by defining the critical role of orchestration within the system's broader context. Subsequently, it details the core technologies chosen for the implementation, justifying the selection of Apache Airflow on the Google Cloud

Platform (GCP). Finally, it outlines the robust local development environment and the strategy for deploying workflows into the managed production environment, emphasizing the seamless transition from development to operation.

9.1 Introduction to Pipeline Orchestration

In a sophisticated, multi-stage system like the real-time fraud detection pipeline, orchestration is the component responsible for scheduling, executing, monitoring, and managing the complex web of data workflows. The end-to-end process—spanning from the initial ingestion of raw transactional data to the final export of machine learning features—relies on a series of dependent tasks that must be executed in a precise, reliable, and timely manner. Effective orchestration ensures data integrity, provides fault tolerance, and enables scalability.

For this project, **Apache Airflow** was selected as the orchestration engine due to its widespread adoption, inherent scalability, and powerful ecosystem. As an open-source, Python-native platform, it allows for the definition of complex workflows as code, which aligns perfectly with modern data engineering best practices (DataOps). This approach facilitates version control, automated testing, and collaborative development.

To host the production environment, **Google Cloud Composer** was chosen. Cloud Composer is a fully managed workflow orchestration service built on Apache Airflow. This choice abstracts away the underlying infrastructure management, including the setup, scaling, and maintenance of Airflow's components. By leveraging Cloud Composer, the project benefits from deep integration with other Google Cloud services, robust security controls, and the reliability of a managed platform, allowing the development team to focus exclusively on building and optimizing data workflows.

9.2 Core Technologies and Environment

The architecture is built upon a foundation of powerful and integrated services, primarily within the Google Cloud Platform. The local development environment is designed to mirror this production setup closely.

- **Google Cloud Platform (GCP):** The entire pipeline is deployed on GCP, leveraging a suite of its managed services to ensure scalability, reliability, and interconnectivity.
    - **Cloud Composer:** The managed Apache Airflow environment for production orchestration.
    - **BigQuery:** Acts as the central, serverless data warehouse. It is used for storing historical transaction data (Fact_Transactions) and for performing large-scale SQL-based feature engineering calculations.

    - **Google Cloud Storage (GCS):** Serves as a versatile and scalable staging layer. It is used to store raw input data for processing and as an intermediate location for data being exported from BigQuery before it is loaded into Firestore.
    - **Dataproc:** A managed Apache Spark service used to execute the heavy-lifting ETL (Extract, Transform, Load) jobs that process raw hourly transaction data.
    - **Firestore:** A NoSQL document database used to store the final, pre-calculated user and merchant features. Its low-latency read capabilities make it ideal for serving this data in real-time to the fraud detection model's inference service.
- **Apache Airflow:** This is the core engine for defining, scheduling, and monitoring the data pipelines (DAGs). It acts as the central nervous system of the entire batch-processing component of the architecture.
- **Docker:** This containerization technology is fundamental to the local development workflow. It is used to encapsulate the entire Airflow environment, ensuring consistency between developer machines and creating a high-fidelity replica of the production setup.

9.3 Local Development and Deployment Strategy

A key principle of this project is to enable a seamless and efficient development lifecycle. This is achieved through a well-defined local environment that uses Docker to replicate the Cloud Composer setup, minimizing discrepancies and facilitating a smooth transition of code to production.

- **The Dockerfile:** This file defines a custom container image that serves as the foundation for the Airflow environment. It starts with an official apache/airflow base image and systematically builds upon it:
    - It copies the requirements.txt file and installs all necessary Python dependencies, including apache-airflow-providers-google, google-cloud-firestore, and google-cloud-bigquery. This guarantees that the local execution environment has the exact same libraries as the production Cloud Composer environment.
- **The docker-compose.yaml File:** This configuration file orchestrates the multi-container local Airflow cluster. It defines and links all the necessary services, including the Airflow scheduler, webserver, worker, a Postgres database for metadata, and a Redis instance for message brokering. A critical feature of this setup is the use of volumes to mount the local dags, plugins, and logs directories directly into the running containers. This allows developers to modify DAGs or custom plugins on their local machine and see the changes reflected immediately in the Airflow UI without needing to rebuild the Docker image, drastically accelerating the development and testing cycle.

- **Transition to Production:** The workflow is designed for simplicity and reliability. Developers write, test, and validate their DAGs and custom plugins entirely within the local Docker-based environment. Once a feature or bug fix is complete and tested, the code from the dags/ and plugins/ directories is synchronized with the corresponding GCS bucket associated with the project's Cloud Composer instance. Composer automatically detects the changes and updates the workflows. This strategy ensures that the code running in production has been vetted in a nearly identical local environment, significantly reducing the risk of environment-specific deployment failures.

10. Core Data Processing Workflows (DAGs)

This section provides an in-depth analysis of the two primary Directed Acyclic Graphs (DAGs) that constitute the core of the data processing and feature engineering pipeline. These workflows are meticulously designed not only for functionality but also for efficiency, robustness, and maintainability. The discussion will highlight the specific design patterns, data validation techniques, and performance optimizations implemented within each DAG.

10.1 Hourly Transactional ETL (dataproc_hourly_transactions_etl)

This DAG is the primary ingestion mechanism for the system, responsible for processing raw transaction data as it arrives on an hourly basis. Its core function is to orchestrate a serverless Spark job that transforms this raw data and loads it into the main analytical table in the BigQuery data warehouse.

- **Objective:** To reliably and efficiently execute a Spark-based ETL process on an hourly schedule using a Dataproc cluster. The job reads raw, partitioned data from Google Cloud Storage (GCS), performs necessary transformations and enrichments, and appends the processed records into the Fact_Transactions table in BigQuery.
- **Efficiency and Pre-checks:** To prevent unnecessary resource allocation and ensure operational efficiency, the DAG incorporates a critical pre-check:
    - **Fail-Fast Mechanism:** The workflow begins with a **GCSObjectsWithPrefixExistenceSensor**. This sensor actively polls a specific, dynamically generated GCS path (e.g., .../year=2025/month=08/day=04/hour=07/) corresponding to the DAG's execution hour. The subsequent, more resource-intensive Dataproc job is only triggered if and when the source data files are confirmed to exist. This "fail-fast" approach is crucial for cost optimization, as it avoids spinning up a Dataproc cluster—an expensive operation—if the upstream data source is delayed or missing.
    - **Dynamic and Templated Execution:** The core processing task is handled by the **DataprocInstantiateWorkflowTemplateOperator**. Rather than defining the Spark cluster and job configuration directly within the DAG, this operator calls a

pre-configured, reusable Dataproc Workflow Template. This design decouples the orchestration logic from the execution logic. Furthermore, it allows for the dynamic injection of parameters (such as the specific year, month, day, and hour) directly into the Spark job at runtime. This makes the workflow template highly reusable and the DAG code clean and focused on orchestration.

- **Post-Load Validation:** Data integrity is paramount for a fraud detection system. To guarantee the reliability of the data loaded into the warehouse, a post-processing validation step is included:
    - **Data Integrity Check:** After the Dataproc job completes, a **BigQueryCheckOperator** executes a SQL query against the target Fact_Transactions table. This query verifies that records corresponding to the specific processing hour (TIMESTAMP_TRUNC(transaction_timestamp, HOUR) = '{{ data_interval_start }}') have indeed been written. If the query returns a count of zero, the task fails, immediately signaling a data loading issue. This automated check ensures that any downstream processes, including the daily feature calculation, will only run if the required source data is complete and verified.
- **Error Handling and Alerting:** The DAG is configured with a custom on_failure_callback function (task_failure_alert). In the event of any task failure, this function is triggered, logging detailed context about the failure (DAG ID, Task ID, Log URL). This mechanism can be easily extended to send alerts to external systems (e.g., Slack, PagerDuty), enabling proactive monitoring and facilitating a rapid response from the data engineering team.

10.2 Daily Feature Export to Firestore (dwh_export_to_firestore_v2)

This daily batch DAG is responsible for the critical task of feature engineering. It calculates and denormalizes key user and merchant attributes from the historical data in BigQuery and then exports these enriched profiles to Firestore, where they can be accessed with low latency by the real-time inference service.

- **Objective:** To pre-calculate critical features for both users and merchants, ensuring that the real-time component of the fraud detection system has access to fresh, comprehensive data. The workflow exports this data to two separate Firestore collections.
- **Efficiency and Parallelism:** The DAG is designed to minimize its total runtime by executing independent data streams concurrently:
    - **Parallel Execution with TaskGroup:** The logic for processing user data and merchant data is encapsulated into two distinct **TaskGroup** blocks: process_users_to_firestore and process_merchants_to_firestore. Since the calculation and export of user features are independent of the export of merchant

dimensions, Airflow executes these two groups in parallel. This design significantly shortens the overall duration of the DAG run compared to a sequential execution.

    - **Automated Cleanup:** Each task group concludes with a **GCSDeleteObjectsOperator**. This task is responsible for removing the temporary newline-delimited JSON files that were created in GCS during the export process. This automated cleanup ensures that the staging area remains tidy and helps control storage costs.

- **Data Quality and In-Warehouse Pre-computation:** The DAG leverages the analytical power of BigQuery and includes several data quality safeguards:
    - **Source Data Verification:** Both task groups begin with a **BigQueryCheckOperator**. This operator serves as a pre-flight check, ensuring that the source BigQuery tables (e.g., Fact_Transactions for users, Dim_Merchants for merchants) are populated and meet basic quality criteria before any resource-intensive calculations are initiated.
    - **In-Warehouse Feature Engineering:** The most computationally demanding part of the workflow—the calculation of historical user features—is handled by the **BigQueryInsertJobOperator**. This operator executes the complex, templated SQL script get_users_historical_features.sql. By performing these aggregations and calculations directly within the BigQuery engine, the system leverages its immense processing power and avoids the need to pull massive amounts of raw data into Airflow workers for processing, which would be highly inefficient.
- **Modularity and Reusability:** To promote clean code and reusability, the DAG utilizes a custom operator for the final loading step:
    - **Custom GCSToFirestoreOperator:** The logic for reading the JSON files from GCS and writing them to Firestore is encapsulated within a custom, reusable operator. This approach abstracts away the complexity of the loading process, making the DAG definition itself much cleaner, more readable, and focused on the high-level workflow. This operator is a key component of the project's custom plugin, which will be detailed in the next section.

11. Custom Extensions and System Robustness

This section details the custom components developed specifically for this project and discusses the overarching design principles that were implemented to ensure the orchestration system is

efficient, maintainable, and highly reliable. It highlights how these bespoke extensions and adherence to data engineering best practices elevate the solution beyond a standard implementation, creating a production-grade data platform.

11.1 Custom Airflow Plugin: GCS to Firestore

A core component of the daily feature export pipeline (dwh_export_to_firestore_v2) is its ability to load data efficiently from Google Cloud Storage (GCS) into Firestore. While Airflow's Google Provider package offers extensive integrations, a direct, feature-rich operator for this specific task—loading newline-delimited JSON files with schema validation—was not available. To address this gap, a custom Airflow plugin was developed, consisting of a specialized Hook and an Operator.

- **Motivation:** The primary motivation for creating this plugin was to build a reusable, robust, and efficient component that could handle the specific requirements of the project. The desired functionality included batch loading to optimize write performance, strong schema enforcement to guarantee data integrity in the NoSQL database, and seamless integration with Airflow's existing Google Cloud connection management.
- **The Firestore Hook (firestore.py):** The foundation of the plugin is a custom Hook that cleanly encapsulates all direct interaction with the Google Cloud Firestore API.
    - **Authentication and Connection:** The FirestoreHook extends Airflow's GoogleBaseHook, inheriting its powerful and secure mechanism for handling authentication. This allows it to seamlessly use the google_cloud_default connection configured in Airflow, whether it relies on a service account key file (in local development) or Application Default Credentials (in Cloud Composer).
    - **Efficient Batch Operations:** A key feature of the hook is its batch_write method. Firestore performs best when write operations are sent in atomic batches (up to 500 operations per batch). This method takes a list of write instructions (e.g., set, update, delete) and commits them in a single, efficient API call. This approach is significantly faster than writing records one by one and ensures that a batch of writes either succeeds or fails together, preventing partially loaded data.
    - **Helper Methods:** The hook provides a clean API with helper methods like create_batch_set_operations_from_list. These methods abstract away the complexity of constructing Firestore WriteBatch objects, allowing the operator to simply pass a list of data dictionaries.
- **The GCSToFirestoreOperator (gcs_to_firestore.py):** This custom operator orchestrates the end-to-end process, acting as the user-facing component within the DAG.
    - **Orchestration Logic:** The operator is designed to work downstream from a task that provides a list of GCS file paths (such as GCSListObjectsOperator), which it

retrieves from XComs. It then uses the standard GCSHook to download each JSON file, processes the records, and uses the custom FirestoreHook to execute the writes.

- **Schema Enforcement:** A critical feature for ensuring data quality is the operator's built-in schema enforcement. It accepts a schema parameter—a dictionary mapping field names to Python types (e.g., {'user_tx_count_24h': int, 'user_home_lat': float}). Before writing to Firestore, it iterates through each record and attempts to cast the values to their specified types. This process is robust; if a particular field cannot be cast, it logs a warning and leaves the value as is, rather than failing the entire task. This prevents a single malformed record from halting the entire data load while still guaranteeing that well-formed data conforms to the correct types.
- **Intelligent** Batch **Processing:** The operator does not attempt to load all records from all files into memory at once. It processes the records and, before sending them to the hook, intelligently divides them into chunks of 500. It then submits these batches sequentially to the FirestoreHook, adhering to Firestore's API limits and ensuring the process remains memory-efficient.

11.2 Principles of Efficient and Robust DAG Design

Beyond the creation of custom components, the project's DAGs are built upon a foundation of design principles aimed at maximizing reliability, efficiency, and maintainability.

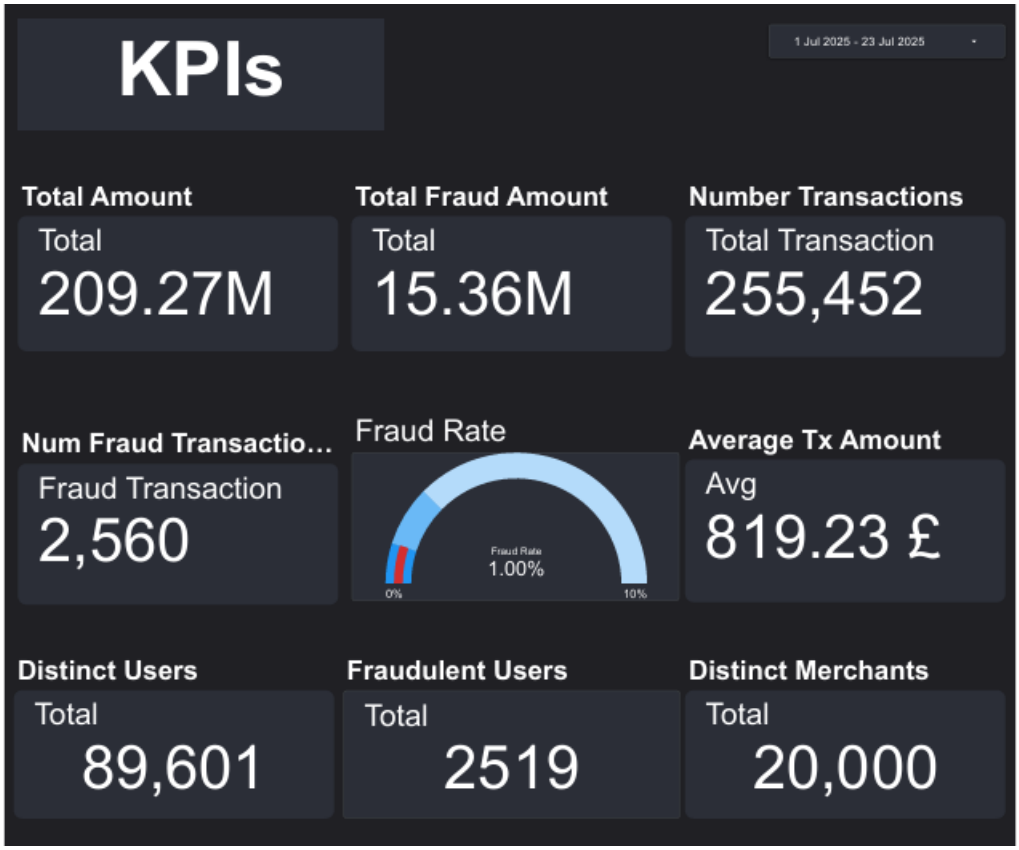
- **Dynamic Configuration:** All environment-specific values, such as GCP project IDs, GCS bucket names, and Firestore collection names, are externalized from the DAG code using **Airflow Variables** (e.g., Variable.get("gcp_project_id", ...)). This practice is fundamental to creating portable workflows. It allows the exact same DAG code to be executed in the local Docker environment and the production Cloud Composer environment simply by setting the appropriate variable values in each context, eliminating the need for environment-specific code changes.
- **Proactive Data Validation:** The workflows embody a "fail-fast" philosophy. Instead of running an expensive job only to have it fail due to missing or invalid source data, pipelines incorporate proactive validation steps. The use of the **GCSObjectsWithPrefixExistenceSensor** in the hourly ETL and the **BigQueryCheckOperator** at the start of the daily feature export are prime examples. These checks verify data availability and quality *before* committing to resource-intensive tasks, saving significant computation time and cost.
- **Code Organization and Maintainability:** The project follows a clean and logical directory structure that separates distinct concerns.

    - **Separation of Logic:** SQL code is kept in dedicated .sql files, custom components reside in the plugins/ directory, and DAG definitions are located in the dags/ folder. This separation makes the codebase easier to navigate, manage, and test.
    - **Readability with TaskGroup:** The use of TaskGroup in the daily export DAG visually and logically bundles related tasks together. This greatly improves the DAG's readability in the Airflow UI, making it easier to understand the workflow's structure and monitor its execution at a glance.
- **Comprehensive Monitoring and SLAs:** The system is designed for operational excellence. Key DAGs include a Service Level Agreement (**sla**) parameter, which instructs Airflow to trigger an alert if a DAG run exceeds its expected completion time. This, combined with the **on_failure_callback** functions, creates a comprehensive monitoring framework that ensures the system meets its performance targets and that the operations team is immediately notified of any deviations or failures, enabling prompt investigation and resolution.

12. Executive Summary & Key Performance Indicators (KPIs)

This section of the dashboard provides a high-level, at-a-glance summary of the platform's transactional health, user activity, and overall risk profile. These Key Performance Indicators (KPIs) are designed to offer a real-time pulse of the business. All KPIs in this section are

interactive and will dynamically update based on the date range selected in the global filter, allowing for flexible period-over-period analysis.



12.1.1 Total Transaction Amount

What it Represents: This KPI displays the total monetary value of all transactions processed through the platform within the selected time frame. It is the primary indicator of the overall financial volume being managed by our services.

Business Value: Tracking this top-line metric is fundamental to understanding business growth, market penetration, and seasonal revenue patterns. A steady increase over time is a key indicator of a healthy, growing business.

12.1. 2 Average Transaction Amount

What it Represents: This is the average value of a single transaction, calculated by dividing the Total Transaction Amount by the Number of Transactions.

Business Value: This metric provides crucial insight into the typical purchasing behavior of our user base. Monitoring for significant shifts is important; a sudden decrease might indicate low-value "card testing" fraud, while a sharp increase could signal new, high-value fraud attacks or a successful entry into a higher-value market segment.

12.1. 3. Number of Transactions

What it Represents: This is the absolute count of all transaction events that have occurred on the platform during the selected period.

Business Value: This KPI is a direct measure of platform activity and user engagement. It helps us understand system load, capacity planning needs, and the overall frequency with which our services are used by customers.

12.1.4. Number of Distinct Users (Active Users)

What it Represents: This KPI shows the total number of unique users who have performed at least one transaction within the specified date range.

Business Value: This is a vital metric for gauging the health and reach of our user base. It helps us measure the effectiveness of user acquisition and retention campaigns. A growing number of active users is a primary sign of a thriving platform.

12.1.5. Fraud Rate

What it Represents: This is the percentage of total transactions that were classified as fraudulent, based on their predictive score exceeding our established risk threshold.

Business Value: This is arguably the most critical risk indicator on the dashboard. It provides a direct measure of the effectiveness of our fraud detection models and rules against the volume of incoming threats. Our goal is to keep this rate as low as possible while minimizing friction for legitimate users.

12.1.6. Number of Distinct Merchants

What it Represents: This is the count of unique merchants who have received at least one transaction during the selected period.

Business Value: This metric indicates the breadth of our merchant network and the diversity of our business ecosystem. Growth in this area signifies successful business development and wider adoption of our payment platform in the market.

12.1.7. Number of Fraudulent Users

What it Represents: This KPI shows the number of unique users who have had at least one transaction classified as high-risk or fraudulent.

Business Value: Crucially distinct from the Fraud Rate, this metric measures the reach of fraud across our customer base.

12.1.8. Chart: Daily Transaction Volume vs. Fraud Volume Over Time

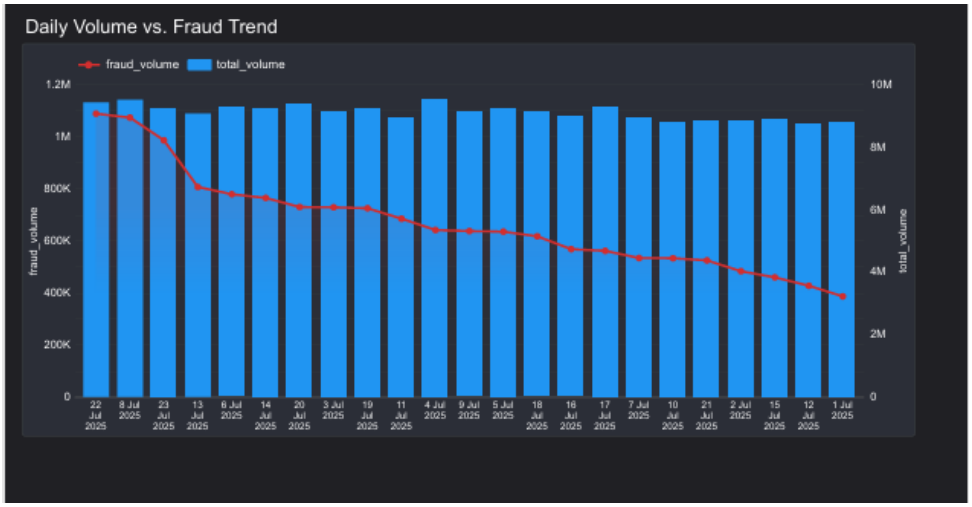
What it Represents: This is a combo chart that visualizes the relationship between legitimate activity and fraud on a daily basis. The bars represent the total number of transactions each day, while the superimposed line represents the number of transactions on that same day that were classified as fraudulent.

Business Value: This is the most powerful operational chart in the KPI section. It allows for the immediate identification of trends and anomalies:

Correlation: We can see if fraud volumes rise in tandem with overall business volume (e.g., during holidays or marketing campaigns).

Divergence: We can spot days where overall volume is normal, but the fraud line spikes. This is a clear signal of a targeted fraud attack.

Temporal Patterns: It helps us answer critical questions, such as whether fraudsters are more active at night or on weekends, allowing us to adjust monitoring and staffing accordingly. By observing the interplay between these two metrics over time, we gain an unparalleled understanding of our platform's risk dynamics.



12.2. Deep Fraud Analysis

While the KPI section provides the high-level "what," this section is designed to answer the more critical questions of "how" and "where" fraud is occurring. The analyses here dissect fraudulent activity across key dimensions of our business, moving from broad statistics to specific, actionable patterns. This section is the primary workspace for fraud analysts, risk managers, and operational teams to investigate threats and formulate defensive strategies.

12.2.1. Chart: Fraud Incidents by Card Type

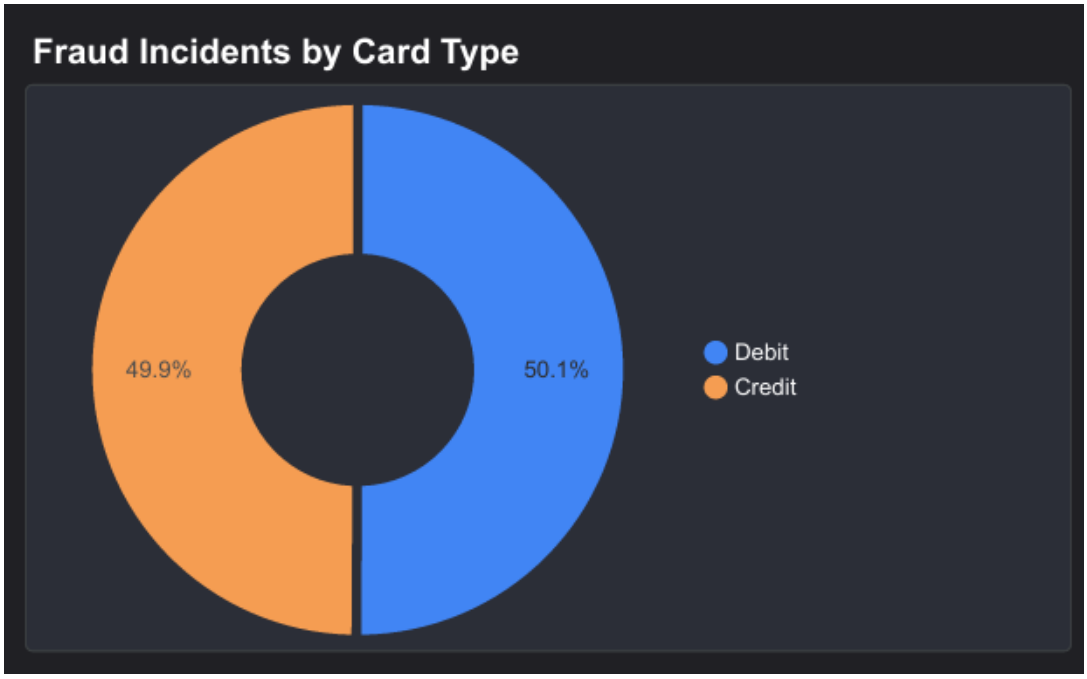
What it Represents: This chart provides a clear breakdown of the total number of fraudulent transactions attributed to each primary payment card type (e.g., Credit, Debit). It visualizes which payment methods are most frequently involved in fraudulent incidents.

Business Value & Actionable Insights: Understanding which card types are most abused is a fundamental step in risk management. This chart allows us to:

Focus Investigative Efforts: If we observe that debit cards, for example, constitute a disproportionately high percentage of fraud incidents compared to their share of total transactions, our teams know to prioritize the analysis of debit card fraud patterns.

Tailor Risk Rules: Fraud often manifests differently across payment types. Credit card fraud might involve high-value luxury goods, while debit card fraud might be characterized by rapid, low-value "draining" of an account. This insight allows us to build more nuanced and effective rule sets specific to the card type.

Inform Partner Discussions: Persistent issues with a specific card type can be a data point for strategic discussions with our issuing bank partners to collaborate on enhanced security measures.



12.2.1. Chart: Fraud Incidents by Card Type

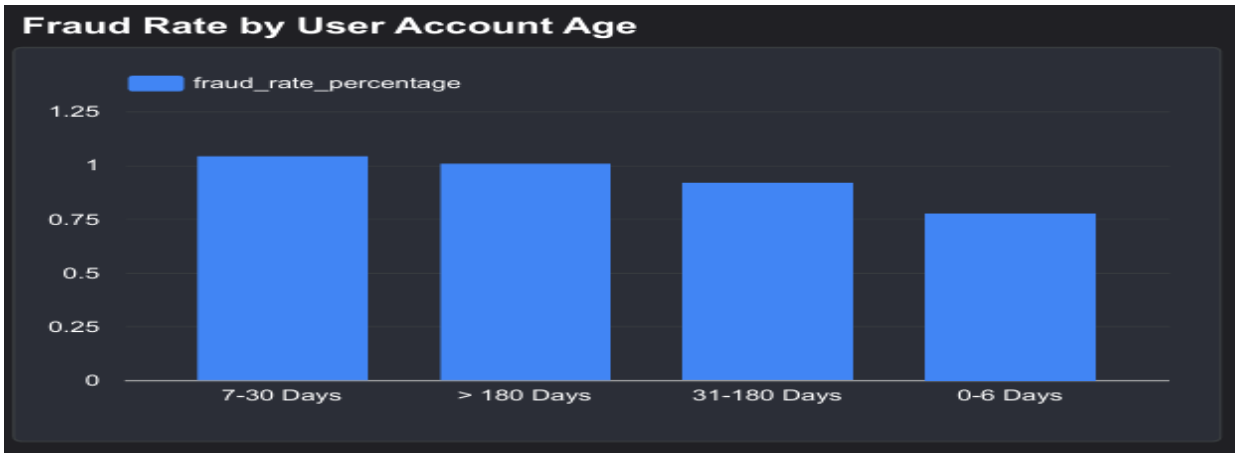
9.2.2. Chart: Fraud Rate by User Account Age

What it Represents: This bar chart segments user accounts into time-based cohorts based on their age (e.g., 0-7 days, 8-30 days, 31-90 days, >90 days) and displays the calculated fraud rate for each group.

Business Value & Actionable Insights: An account's age is one of the most powerful predictors of its risk profile. This analysis is critical for detecting two primary types of fraud:

Onboarding and Synthetic Fraud: A significantly elevated fraud rate in the newest account buckets (e.g., "0-7 days") is a classic indicator that fraudsters are exploiting our sign-up process. They may be creating accounts en masse with stolen or fabricated information (synthetic IDs) to commit fraud before a legitimate behavioral pattern can be established. This insight provides a direct mandate to strengthen our identity verification and security checks during user onboarding.

Account Takeover (ATO): Conversely, a sudden and unexpected spike in the fraud rate for mature, established accounts (e.g., ">90 days") can signal a successful ATO attack, where criminals have gained unauthorized access to legitimate customer accounts. This would trigger an immediate investigation into phishing campaigns or data breaches.



9.2.2. Chart: Fraud Rate by User Account Age

9.2.3. Chart: Fraud Rate by Merchant Category

What it Represents: This chart ranks our merchant categories from highest to lowest based on their fraud rate. It instantly visualizes which types of businesses are most frequently targeted by fraudsters.

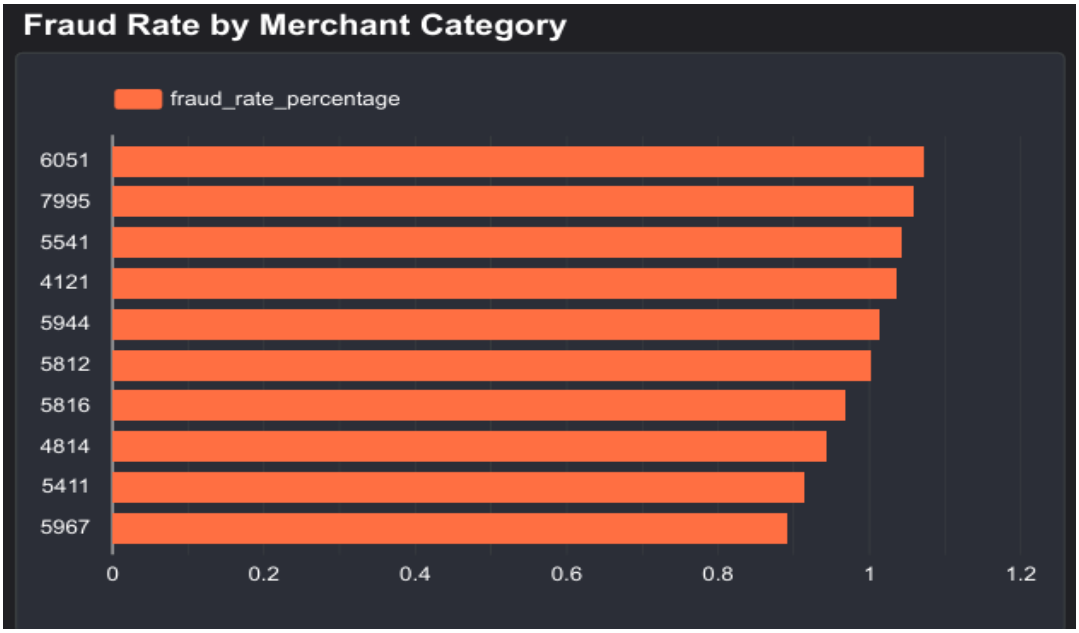
Business Value & Actionable Insights: Risk is not evenly distributed across the commercial landscape. This analysis enables a more intelligent and targeted approach to merchant management:

Identify High-Risk Industries: It confirms that industries selling digital goods, gift cards, or money transfers are inherently riskier than those selling physical goods that require shipping.

Drive Dynamic Risk Policy: This data provides the justification for applying different risk policies to different merchants.

High-risk categories may be subject to lower transaction velocity limits, mandatory address verification, or longer payout settlement times to mitigate potential losses.

Proactive Merchant Support: When a new fraud trend emerges in a specific category, our operations team can use this chart to identify all merchants in that vertical and proactively reach out with warnings and best practices, turning our platform into a protective partner.



9.2.4. Chart: Fraud Rate by Distance From Home

What it Represents: This chart groups transactions into logical buckets based on the physical distance between the transaction's point of origin (derived from its IP address) and the user's registered home address. It then displays the fraud rate for each distance bucket (e.g., Local, Regional, Long Distance, International).

Business Value & Actionable Insights: This analysis is based on the simple but powerful premise that people's spending habits are geographically predictable. A significant deviation from this norm is a strong indicator of risk.

Quantify Geographic Risk: It provides a precise, data-driven measure of how much risk increases when a transaction is geographically anomalous. Knowing that a transaction from >500km away is 8x riskier than a local one is a powerful input for our fraud models.

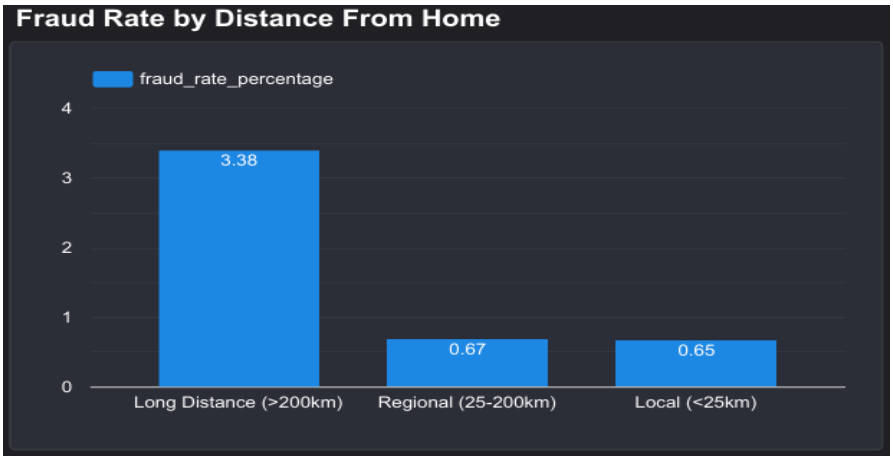
Refine Customer Experience: This analysis helps us balance security with convenience. We can fine-tune our rules to be less suspicious of a user who frequently travels between two specific cities, while still flagging a completely unexpected international transaction. This prevents frustrating legitimate customers with unnecessary declines.

Implement Step-Up Authentication: This chart provides the perfect justification for implementing dynamic security measures. For a transaction that falls into a high-risk distance

bucket, we can automatically trigger a request for "step-up" authentication (such as a one-time password sent via SMS) to verify the user's identity before approving the purchase.

Analysis of Daily Transaction Volume & Risk Patterns

This section of the dashboard transitions from static, category-based analysis to a dynamic view of our transaction ecosystem. The charts herein are designed to uncover patterns based on when transactions occur and how much they are worth. By analyzing the rhythm of our daily business and comparing it to fraudulent activity, we can identify critical vulnerabilities and strategic opportunities.



9.3.1. Chart: Daily Fraud & Transactions Trend

What it Represents: This is the foundational chart for temporal analysis. It is a time-series combo chart that plots two key metrics for each day in the selected period:

1.Total Transactions (Bars): The absolute count of all transactions, representing the daily business volume.

2.Fraudulent Transactions (Line): The absolute count of fraudulent transactions, representing the daily risk volume.

Business Value & Actionable Insights: This chart provides an immediate operational overview of platform activity versus risk. It is the primary tool for identifying macro-level trends and pinpointing specific dates of interest. Its value lies in:

Establishing a Baseline: By observing the chart over a long period, we establish a visual baseline for what "normal" looks like, making any deviation instantly noticeable.

Pinpointing Attack Dates: A day where the "Fraudulent Transactions" line shows a dramatic spike, especially if the "Total Transactions" bars remain normal, is a definitive indicator of a targeted fraud attack. This allows analysts to immediately isolate and investigate the events of that specific day.

Resource and Capacity Planning: The chart visualizes our busiest days (e.g., weekends, holidays, paydays), allowing operations and support teams to plan staffing schedules effectively to handle both customer inquiries and fraud alerts.

9.3.2. Chart: Peak Times for Transactions vs. Fraud

What it Represents: This is a heatmap that visualizes risk concentration by time of day and day of the week. The chart's grid displays each day of the week on one axis and each hour of the day on the other. The color of each cell indicates the fraud rate during that specific time block, with darker or redder colors representing periods of higher risk.

Business Value & Actionable Insights: This analysis reveals the hidden temporal patterns of criminal behavior, which often diverge significantly from legitimate customer activity. It answers the critical question: "When are we most vulnerable?"

Identifying Fraudster "Work Hours": Legitimate customer activity often peaks during daytime and evening hours. Fraudsters, however, frequently operate during off-peak hours (e.g., 1 AM - 5 AM) to take advantage of lower staffing and monitoring. This chart will clearly visualize these high-risk "graveyard" shifts.

Dynamic Policy Deployment: This insight is directly actionable. We can configure our fraud detection system to apply stricter rules or lower transaction limits automatically during the time blocks identified as high-risk. For example, a high-value transaction at 3 PM might be normal, but the same transaction at 3 AM should be subject to extra scrutiny.

Optimizing Maintenance Windows: By identifying periods of both low legitimate volume and low fraud volume, we can schedule system maintenance and updates with minimal impact on business and minimal security exposure.

Peak Times for Transactions vs. Fraud

	hour_of_day / fraud_rate_percentage				
day_of_week_name	08	02	20	10	22
Wednesday	1.7	1.25	1.44	1.19	1.3
Sunday	1.27	0.98	0.6	1.45	1.21
Friday	1.04	1.49	1.07	0.88	1.01
Thursday	0.81	0.86	1.15	1.41	1.01
Monday	0.97	0.97	1.58	0.72	1.4
Saturday	1.36	1.02	1.16	1.23	0.98
Tuesday	1.13	1.59	0.94	0.88	0.81

9.3.3. Chart: Spend vs. Risk (Customer Segment Analysis)

What it Represents: This combo chart provides a powerful analysis connecting customer value to risk. It segments our active users into spending tiers (e.g.

"Low Spender," "Medium Spender," "High Spender") based on their average transaction value. For each segment, the chart displays

1. **Total Transactions (Bars):** The volume of activity from that segment.

2. **Fraudulent Transactions (Line):** The volume of fraud originating from that segment.

Business Value & Actionable Insights: This chart helps us understand who the fraudsters are targeting and what attack strategies they are employing. It allows us to move beyond a one-size-fits-all approach to security.

Diagnosing Attack Strategies: A high fraud line in the "Low Spender" segment suggests a "death-by-a-thousand-cuts" strategy, such as large-scale bot networks performing card testing with small amounts. In contrast, a high fraud line in the "High Spender" segment points to more sophisticated attacks aimed at high-value targets, such as account takeovers of our most valuable customers.

Tailoring Customer Security: If "High Spenders" are being targeted, we must prioritize their security with enhanced, yet frictionless, protection to safeguard their trust and our brand's reputation. This could involve offering opt-in biometric security or more personalized monitoring.

Informing Marketing & Product Strategy: Identifying a segment that is both high-spend and low-risk is a significant business intelligence win. This tells our marketing teams exactly which user profile is our most valuable and secure, making them a prime audience for loyalty programs, new product offerings, and targeted growth campaigns.

9.4. Fraud Analysis by Device

This section provides a forensic analysis of the technical footprint left by every transaction. The primary goal is to identify patterns in the device characteristics—specifically the operating system (OS) and web browser—to detect non-human behavior, expose exploited platforms, and uncover the tools used by fraudsters. While previous analyses focus on user and transactional behavior, this view focuses on the technology used to perpetrate fraud, providing a powerful, complementary layer of defense.

9.4.1. Chart: Risk by Operating System

What it Represents: This is a high-level summary table that aggregates all transactions by the operating system of the device used. For each major OS (e.g., Windows, Android, iOS, macOS, Linux), it presents four critical metrics:

1.Total Transactions: The overall transaction volume from that OS, indicating its popularity among our user base.

2. Fraudulent Transactions: The absolute number of high-risk transactions from that OS.

3. Total Fraud Value: The total financial loss attributed to fraud from that OS.

4. Fraud Rate %: The percentage of transactions from an OS that are fraudulent, highlighting the risk concentration.

Business Value & Actionable Insights: This chart serves as the starting point for any device-based investigation, allowing analysts to quickly prioritize their focus.

Immediate Threat Prioritization: It instantly reveals which platform is the source of the most financial damage. If a single OS accounts for a majority of the Total Fraud Value, it becomes the number one priority for the risk team, regardless of its overall popularity.

Identification of Vulnerable Platforms: This analysis often highlights that older, unsupported operating systems (e.g., Android 8, Windows 7) have a disproportionately high fraud rate. This confirms that fraudsters are actively seeking out and exploiting known security vulnerabilities in these legacy platforms.

Detection of Server-Based Fraud: A high fraud rate originating from server-centric operating systems like Linux, which typically have very low legitimate consumer transaction volume, is a significant red flag. This often indicates large-scale, automated attacks being run from servers, not from genuine user devices.

Informing Engineering Roadmaps: This data provides a clear business case for the engineering team to invest in more advanced Software Development Kits (SDKs) and device fingerprinting technologies for the highest-risk platforms.

Risk by OS

	os	total_fraud_value	fraud_rate_percentage	total_fraud_value	total_transactions	fraudulent_transactions
1.	Windows 11	817,612	0.17	817,612	44,632	76
2.	Android 14	726,496	0.19	726,496	44,116	83
3.	Windows 10	647,267	0.19	647,267	43,937	83
4.	iOS 17.5	624,081	0.18	624,081	43,411	80
5.	macOS Sonoma	594,124	0.19	594,124	44,166	82
6.	Android 13	549,651	0.17	549,651	44,163	75

1 - 6 / 6

9.4.2. Chart: Risk by OS & Browser Combination

What it Represents: This detailed table provides a granular breakdown of risk by looking at the specific combination of operating system and browser used for each transaction. It displays the same four key metrics (Total Transactions, Fraudulent Transactions, Total Fraud Value, and Fraud Rate %) for each unique pair (e.g., "Windows 10 + Chrome," "iOS 17 + Safari," "Android 13 + Chrome Mobile").

Business Value & Actionable Insights: This is the most powerful chart in this section for uncovering specific attack vectors and creating surgical, high-confidence security rules.

Pinpointing Exact Attack Vectors: While the previous chart might identify "Android" as a problem, this chart can pinpoint the issue to a very specific combination, such as "Android 12 using the DuckDuckGo browser." This level of specificity is invaluable for understanding how an attack is being carried out. **Uncovering Bots and Emulators:** Fraudsters often use automation scripts or device emulators that result in strange or rare device combinations. A high fraud rate associated with a combination that has very low legitimate use (e.g., a mobile browser on a desktop OS) is a strong signal of non-human, fraudulent activity.

Creating High-Fidelity Block Rules: This analysis is the foundation for our most effective security rules.

Instead of implementing a broad and disruptive rule like "add extra checks for all Android users," we can create a precise rule that targets only the riskiest combinations. For example, "automatically block any transaction from 'Linux + Firefox' attempting to purchase a gift card." This allows us to neutralize specific threats with virtually no impact on our legitimate customers.

Detecting Spoofing: When fraudsters attempt to disguise their device, they often create illogical combinations. Seeing a significant number of fraudulent transactions from an impossible or highly improbable pairing (like "iOS + Internet Explorer") provides definitive proof of malicious intent and allows for immediate action.

Risk by OS & Browser Combination

	browser	os	total_fraud_value ▾	fraud_rate_percen...	total_fraud_value	total_transactions	fraudulent_transac...
1.	Samsung Internet	Windows 11	270,715	0.14	270,715	9,077	13
2.	Edge	Android 14	258,055	0.27	258,055	8,877	24
3.	Firefox	macOS Sonoma	226,437	0.19	226,437	8,760	17
4.	Edge	Android 13	208,937	0.24	208,937	8,693	21
5.	Edge	iOS 17.5	190,564	0.17	190,564	8,834	15
6.	Samsung Internet	Windows 10	183,721	0.23	183,721	9,024	21
7.	Safari	Android 14	167,551	0.15	167,551	8,954	13
8.	Edge	Windows 11	160,225	0.17	160,225	8,662	15
9.	Firefox	Windows 10	158,969	0.16	158,969	8,583	14
10.	Safari	iOS 17.5	154,033	0.24	154,033	8,651	21
11.	Edge	macOS Sonoma	142,254	0.29	142,254	8,894	26
12.	Chrome	Windows 11	141,876	0.22	141,876	8,954	20

1 - 30 / 30