Real-Time Fraud Detection in Transactions.

---

1. DATASET: THE PAYSIM DATASET

1.1 Background and Origin

The **PaySim dataset** is a **synthetic dataset** that simulates **mobile money transactions**. Originally, it was published on **Kaggle**, where it can be downloaded either manually or via the **Kaggle API**. The dataset is designed to **mimic real financial transactions** in terms of scale, distribution, and types of operations, while not revealing any sensitive private customer data (since it's artificially generated).

Why Synthetic?

Real financial transaction data is typically **strictly confidential** (due to privacy regulations and to avoid revealing vulnerabilities). Hence, PaySim was created to provide **researchers** and **developers** a near-realistic scenario to test **fraud detection** algorithms.

Kaggle Link:

We can find the dataset on Kaggle by searching for "PaySim dataset." Typically we can do something like:

```
kaggle datasets download -d YOURUSERNAME/paysim
```

to retrieve it with the Kaggle CLI, or we can download it directly from the Kaggle website.

1.2 Size and Format

- **Rows**: ~6.36 million transactions (some versions might slightly differ).
- **File Size**: ~500 MB in CSV form.

- **Format**: Plain CSV with a header row.

This large volume makes it ideal for **big data** or **streaming** pipeline demonstrations.

1.3 Columns in the PaySim Dataset

The dataset typically includes the following columns (naming might differ slightly depending on the version used):

1. **step**
   - *Integer*: Each "step" corresponds to a discrete time unit in which transactions occurred.
   - Think of it as an approximation of "hours" or "minutes," but the dataset's authors just named it "step."
2. **type**
   - *Categorical*: Transaction type. Common values include:
     - **PAYMENT**
     - **TRANSFER**
     - **CASH_OUT**
     - **DEBIT**
     - **CASH_IN** (in some versions)
   - This is an important feature because **fraud** often correlates with **TRANSFER** or **CASH_OUT**.
3. **amount**
   - *Float*: The monetary value of the transaction. Ranges from small amounts to very large ones (up to 10^7 or so).
4. **nameOrig**
   - *String*: The unique identifier of the sender (origin) account. Often in the format `Cxxxxxxx`.
5. **oldbalanceOrg**

- *Float*: The sender’s account balance before the transaction.

6. **newbalanceOrig**
    - *Float*: The sender’s account balance after the transaction.
7. **nameDest**
    - *String*: The unique identifier of the recipient (destination) account. Often `Cxxxxxxx` for customer, or `Mxxxxxxx` for merchant.
8. **oldbalanceDest**
    - *Float*: The recipient’s account balance before the transaction.
9. **newbalanceDest**
    - *Float*: The recipient’s account balance after the transaction.
10. **isFraud**

- *Integer (0 or 1)*: **Primary label** indicating whether the transaction is fraudulent (`1`) or legitimate (`0`).
- Highly imbalanced: only a tiny fraction are `1`.

1. **isFlaggedFraud**

- *Integer (0 or 1)*: Whether the transaction was flagged as fraud by some older heuristic approach.
- Typically near zero in most rows.

```
Dataset loaded. Shape: (6362620, 11)
```

	step	type	amount	nameOrig	oldbalanceOrg	newbalanceOrig	nameDest	oldbalanceDest	newbalanceDest	isFraud	isFlaggedFraud
0	1	PAYMENT	9839.64	C1231006815	170136.00	160296.36	M1979787155	0.0	0.00	0	0
1	1	PAYMENT	1864.28	C1666544295	21249.00	19384.72	M2044282225	0.0	0.00	0	0
2	1	TRANSFER	181.00	C1305486145	181.00	0.00	C553264065	0.0	0.00	1	0
3	1	CASH_OUT	181.00	C840083671	181.00	0.00	C38997010	21182.0	0.00	1	0
4	1	PAYMENT	11668.14	C2048537720	41554.00	29885.86	M1230701703	0.0	0.00	0	0
5	1	PAYMENT	7817.71	C90045638	53860.00	46042.29	M573487274	0.0	0.00	0	0
6	1	PAYMENT	7107.77	C154988899	183195.00	176087.23	M408069119	0.0	0.00	0	0
7	1	PAYMENT	7861.64	C1912850431	176087.23	168225.59	M633326333	0.0	0.00	0	0
8	1	PAYMENT	4024.36	C1265012928	2671.00	0.00	M1176932104	0.0	0.00	0	0
9	1	DEBIT	5337.77	C712410124	41720.00	36382.23	C195600860	41898.0	40348.79	0	0

1.4 Key Dataset Characteristics

- **Synthetic** but distributionally realistic.
- **Unbalanced**: ~0.13% or fewer transactions labeled as fraud.
- **Transaction diversity**: multiple transaction "types," with typical money-laundering or fraud patterns lying mostly in `TRANSFER` and `CASH_OUT`.

---

2. EXPLANATION OF EACH COLUMN

Although summarized above, let's detail each column thoroughly:

1. **step**
   - Each unit in this feature could be considered an incremental time step. Some interpreters assume it's an hour. If the highest step is ~743, that might represent a month's worth of hours.
   - For time-series or streaming analysis, `step` can be used to see how transactions evolve over time.
2. **type**
   - Common categories:
     - **TRANSFER**: Money moves from one account to another.
     - **CASH_OUT**: The account holder withdraws money, presumably as cash.
     - **PAYMENT**: Payment to a merchant.
     - **DEBIT**: Possibly a direct debit from an account by a third party.
     - (Sometimes **CASH_IN** if the dataset variant includes it.)
   - This is an important predictor because historically, fraudulent transactions often cluster in certain types.
3. **amount**
   - The transaction's monetary value. Ranges widely from negligible amounts to multi-million.
   - Fraud can be high or low amounts, so we must carefully consider distribution.

4. **nameOrig / nameDest**
    - Strings: `Cxxxxxxx` for customer, `Mxxxxxxx` for merchant.
    - Not always needed for direct numeric modeling but can be used for feature engineering (like counting repeated patterns to the same "destination," etc.).
5. **oldbalanceOrg / newbalanceOrg**
    - The balance before and after the transaction for the origin's account.
    - Potentially important: a fraudulent user might have suspicious patterns like "oldbalanceOrg = amount, newbalanceOrg = 0" repeatedly.
6. **oldbalanceDest / newbalanceDest**
    - Similarly for the recipient's account.
    - If a destination is frequently receiving large amounts, that might be a red flag or normal merchant activity.
7. **isFraud**
    - The main label used for **supervised** or **evaluation** tasks.
    - In our anomaly detection approach, we often used this label **only** at the evaluation stage, because we do an **unsupervised** or **semi-supervised** approach.
8. **isFlaggedFraud**
    - Often zero in most rows.
    - The original creators said it flags transactions that are abnormally high (some threshold). Not extremely useful in practice, but can be a baseline.

---

3. OBJECTIVE OF THE PROJECT

Goal: **Real-Time Anomaly Detection in Financial Transactions** using the PaySim dataset as a stand-in for real mobile money data. Specifically:

1. **Detect** suspicious or fraudulent transactions **on the fly** (near real time).

2. **Leverage** an **unsupervised** or **semi-supervised** approach (e.g., IsolationForest, LocalOutlierFactor, Autoencoders) to handle the extreme class imbalance.
3. **Demonstrate** a big data streaming pipeline with **Kafka** + **Spark Structured Streaming** for real-time ingestion.
4. **Provide** a user-friendly dashboard with **Streamlit** that can visualize the flagged anomalies in near-real time, offering **insights** to risk analysts or external stakeholders.

---

4. INTRODUCTION TO THE PROJECT

4.1 Overview

This project aims to simulate a **full end-to-end** pipeline for **fraud detection**:

1. **Data Ingestion**: Instead of a real transaction feed, we have the PaySim CSV. We mimic streaming by publishing rows to **Kafka**.
2. **Stream Processing**: **Spark Structured Streaming** consumes from Kafka, runs a **pre-trained** anomaly detection model (like IsolationForest), and flags suspicious transactions.
3. **Storage**: The flagged anomalies are written out to **CSV** files in near-real time.
4. **Visualization**: A **Streamlit** dashboard loads those flagged anomalies, draws charts, and provides a user-friendly monitoring tool.

4.2 Contents of the Project

1. **EDA & Feature Engineering Notebook** (a local or Colab notebook)
2. **Model Training** with unsupervised ML (IsolationForest, LOF, Autoencoder).
3. **Kafka Producer Script** (`kafka_producer.py`): Streams rows from the CSV into Kafka topic.
4. **Spark Streaming Script** (`spark_streaming.py`): Subscribes to Kafka, loads pre-trained model, flags anomalies, writes them to CSV.

5. **Streamlit Dashboard** (`dashboard.py`): Displays flagged anomalies in real time.
6. **Deployment**: GCP VM (Compute Engine) used to host Kafka + Spark + the entire pipeline.

4.3 Why PaySim?

- The PaySim dataset is large (~6 million rows), capturing the complexity of real transactions.
- The presence of an **isFraud** label allows for **evaluation** of anomaly detection.
- It's free, synthetic, and widely known in academic/industry examples.

4.4 Why Real-Time?

Fraud detection is most valuable when **immediate**. The faster we detect suspicious activity, the faster we can block it or investigate. Traditional offline batch detection might be too slow.

---

5. EXPLORATORY DATA ANALYSIS (EDA), FEATURE ENGINEERING, AND MODEL TRAINING

5.1 EDA

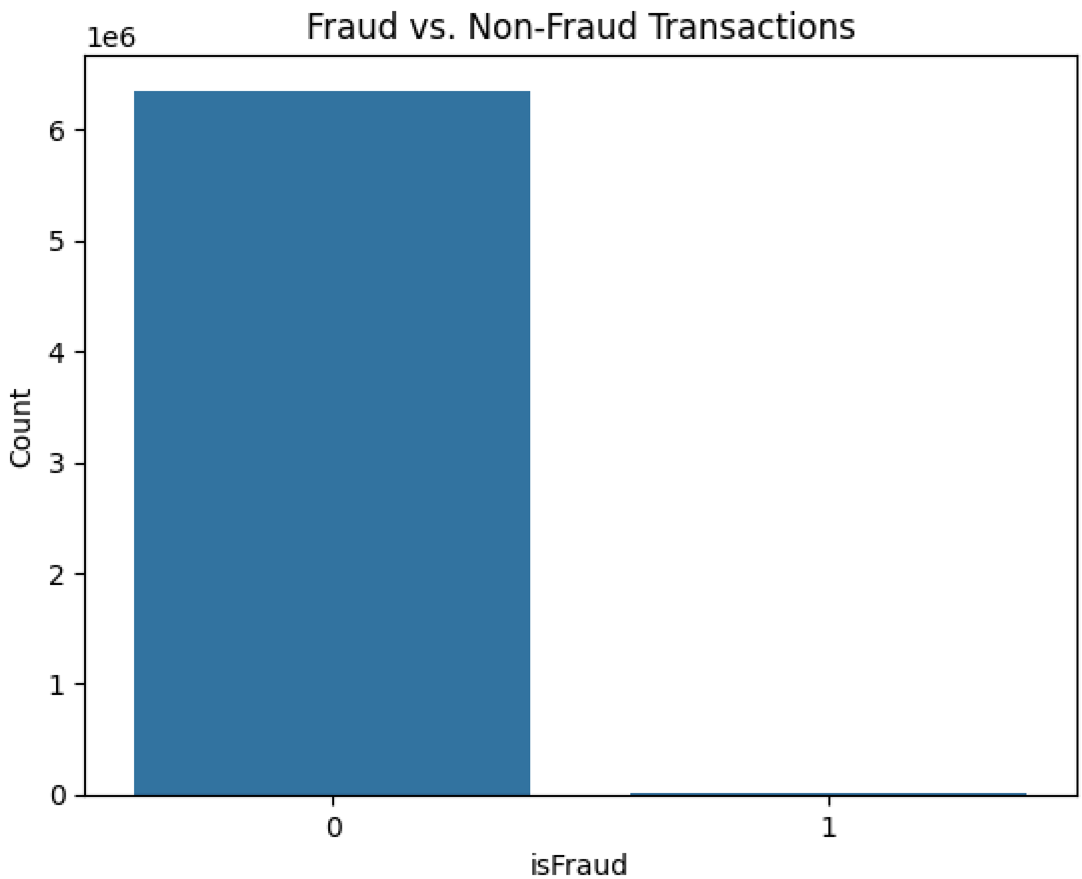
5.1.1 Basic Statistics

- **Total rows**: ~6,362,620
- **Fraud**: ~8,213 (only ~0.13%).
- **Transaction Types**: Payment, Transfer, Cash Out, Debit, etc.
- **Distribution of `amount`**: Skewed; many small transactions, few extremely large.

5.1.2 Visualizations

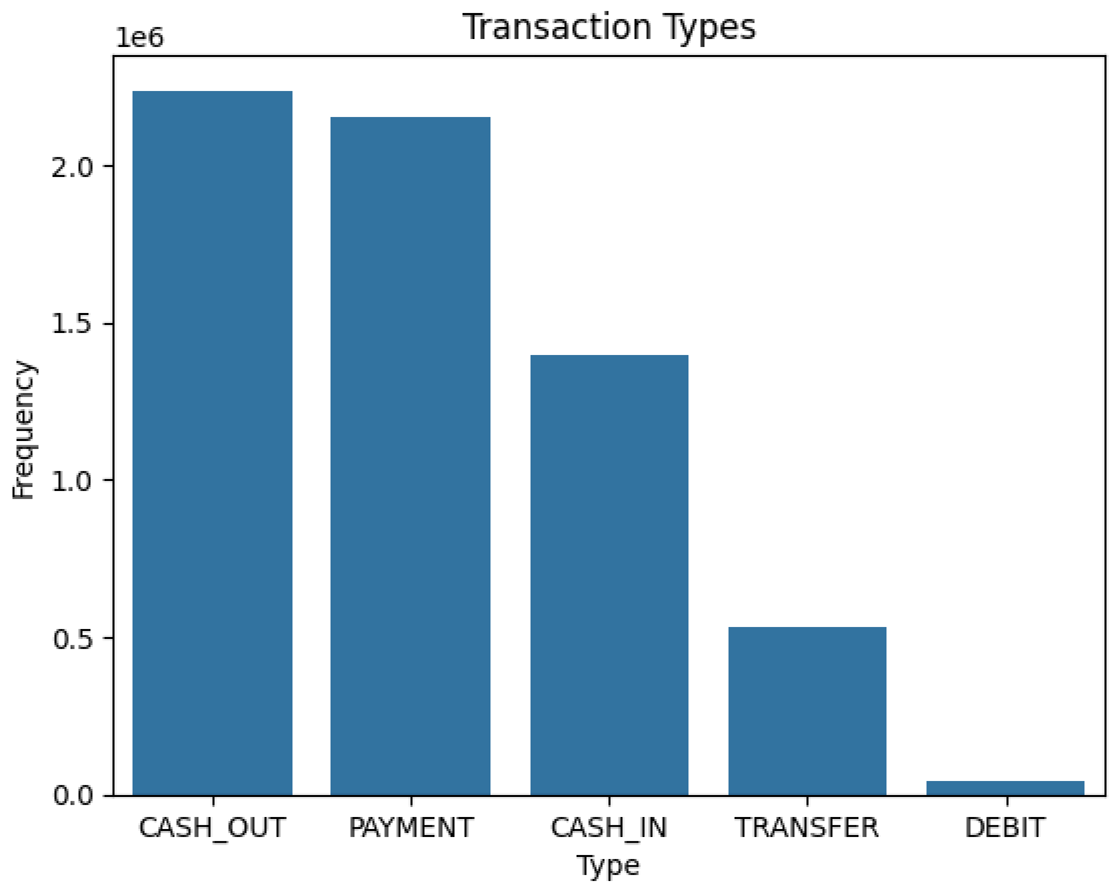
1. **Fraud vs. Non-Fraud Count**

- A bar chart showing an enormous imbalance: ~6.35 million legitimate vs. ~8k fraud.

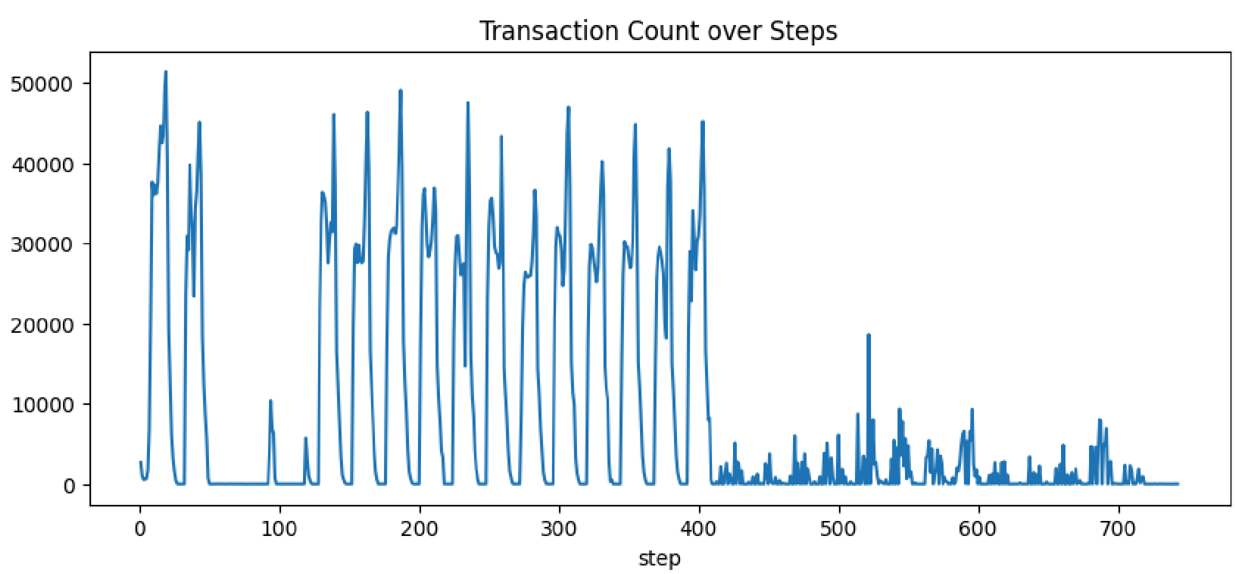


2. **Transaction Types**

   - Payment is the most common type, followed by Transfer and Cash Out.



3. **Time Step Analysis**
    - Plotted transaction volume over `step`. Typically remains fairly stable but some peaks.



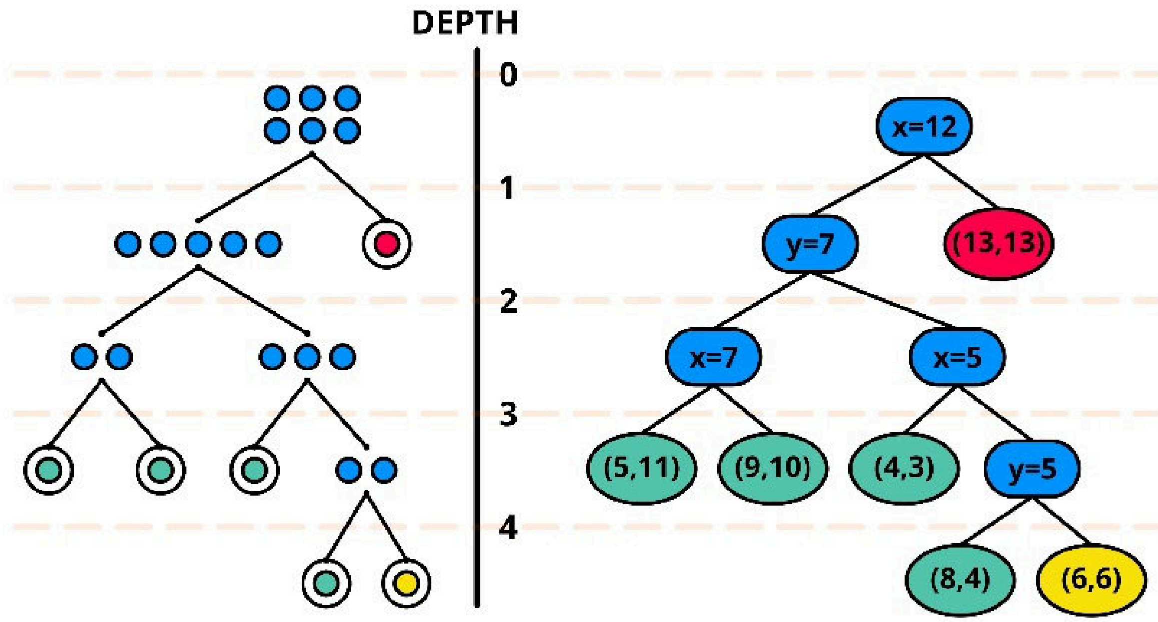
5.2 Feature Engineering

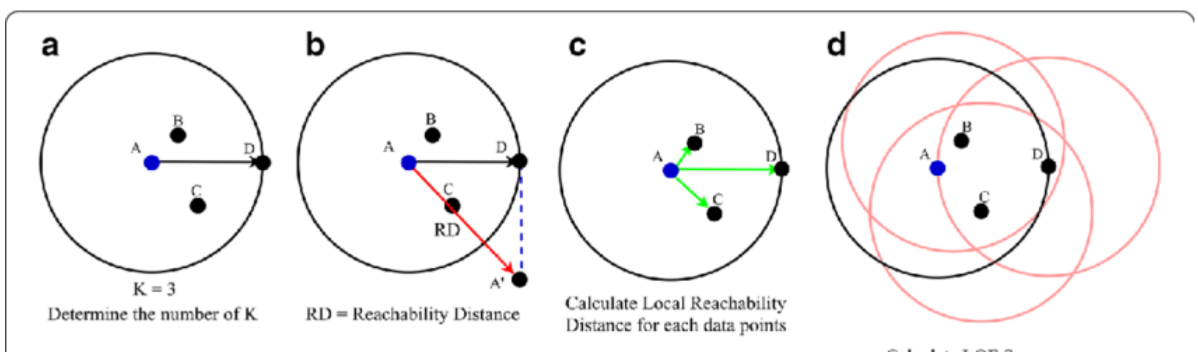
1. **One-Hot Encode** `type`:
    - `type_TRANSFER` = 1 if the transaction type is "TRANSFER," else 0
    - `type_CASH_OUT`, `type_DEBIT`, etc.
2. **Delta Balances**:
    - `delta_balanceOrig = newbalanceOrig - oldbalanceOrg`
    - `delta_balanceDest = newbalanceDest - oldbalanceDest`
3. **Ratio**:
    - `amount_div_oldbalanceOrg` = `amount / oldbalanceOrg` (when `oldbalanceOrg` > 0)
    - Helps detect if someone is draining their entire account in one go.

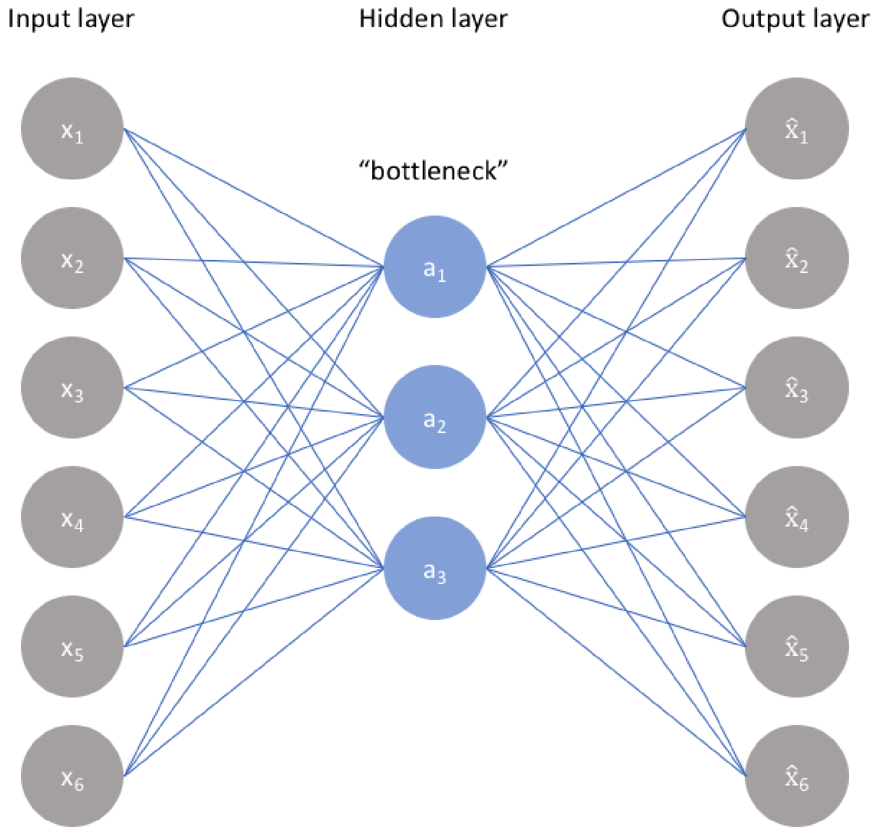
5.3 Model Choices and Why

We tested three unsupervised (or novelty) approaches:

1. **IsolationForest**
    - Excels at isolating anomalies by randomly partitioning data.
    - Very popular for high-dimensional data.
    - In scikit-learn, we do `IsolationForest(...).fit(...)`.
2. **Local Outlier Factor (LOF)**
    - Density-based approach, compares local density of each sample to neighbors.
    - Good for small or moderate datasets, but can be expensive for large data (LOF is O(n^2) with naive approach).
3. **Autoencoder (Neural Network)**
    - A deep learning approach.
    - Trained on normal (non-fraud) data to learn a compressed representation.
    - High reconstruction error on anomalies.


DEPTH
0
1
2
3
4
x=12
(13,13)
y=7
x=7
x=5
(5,11)
(9,10)
(4,3)
y=5
(8,4)
(6,6)


a
b
c
d
A
B
C
D
K = 3
Determine the number of K
RD
A'
RD = Reachability Distance
Calculate Local Reachability
Distance for each data points



5.4 Performance of Each Model

Based on our final classification reports (since we tested them on a labeled portion), we found:

5.4.1 IsolationForest

- **Precision (Fraud)**: ~0.01
- **Recall (Fraud)**: ~0.60
- **F1**: ~0.02

This means it catches ~60% of fraud but with a very small precision (tons of false positives).

5.4.2 LOF

- **Precision (Fraud)**: ~0.00
- **Recall (Fraud)**: ~0.03 or 0.05

- **F1**: ~0.00 or 0.01

LOF typically had the worst performance here—lots of difficulties with large-scale data.

5.4.3 Autoencoder

- **Precision (Fraud)**: ~0.06 or 0.07
- **Recall (Fraud)**: ~0.50 or 0.54
- **F1**: ~0.11 or 0.12

This is better than the other two, giving a more balanced approach. Still plenty of false positives, but decent coverage of fraud.

```
--- Model Comparison ---
```

	Model	Precision (Fraud)	Recall (Fraud)	F1-score (Fraud)
0	Isolation Forest	0.01	0.61	0.02
1	LOF	0.00	0.03	0.00
2	Autoencoder	0.07	0.54	0.12

5.5 Selecting the Final Model

We eventually picked **IsolationForest** in the real-time pipeline because:

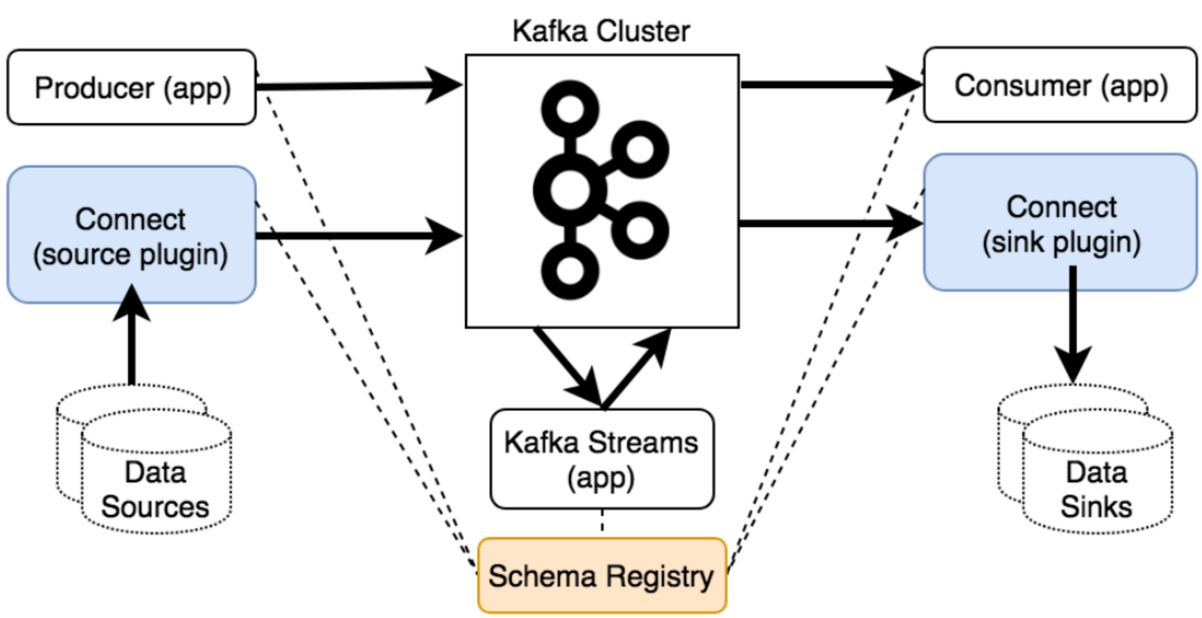
- It's relatively quick to predict.
- Doesn't require a GPU or large overhead.
- Recall was relatively strong (~60%).

Though the autoencoder had decent performance, it can be more complicated to integrate or scale. Some people might prefer the autoencoder if they can handle the overhead or want a better F1.

6. BUILDING THE REAL-TIME PIPELINE (KAFKA + SPARK)

6.1 Pipeline Architecture

1. **Kafka Producer**: A Python script (`kafka_producer.py`) that reads the PaySim CSV row by row, sending each transaction as a JSON message to a **Kafka topic**.
2. **Kafka Topic**: e.g. “paysim_transactions.”
3. **Spark Streaming**: A script (`spark_streaming.py`) that subscribes to that Kafka topic, loads the IsolationForest model (`iso_forest.pkl`), and runs a **UDF** to predict anomalies.
4. **Flagged Anomalies**: Spark writes them as CSV to a specified output folder.



6.2 Why Kafka?

- **Kafka** is a distributed messaging system, widely used for real-time streaming ingestion.
- Simulates a “live feed” of transactions.

- Allows us to scale if we had multiple producers or more consumers.

6.3 Why Spark Structured Streaming?

- It's a popular big-data engine for **stream** processing.
- Can do "micro-batches" in near-real time.
- Integration with Kafka is first-class (`spark-sql-kafka-0-10` connector).

6.4 The `kafka_producer.py` Script

1. **Uses** `csv.DictReader` to read each row from "Transactions.csv."
2. **Transforms** row fields (like `step`, `amount`, etc.) into a JSON message.
3. **Sends** to the "paysim_transactions" topic with `producer.send(...)`.
4. **Sleeps** ~0.01 seconds after each row to simulate ~100 transactions/sec.

6.5 The `spark_streaming.py` Script

6.5.1 Steps

1. **Create SparkSession** with Kafka support:

```
spark = SparkSession.builder \
    .appName("PaySimFraudDetection") \
    .master("local[*]") \
    .getOrCreate()
```

2. **Load Model**:

```
iso_forest = joblib.load("/home/.../iso_forest.pkl")
```

3. **Define Schema** for incoming JSON (step, amount, oldbalanceOrg, etc.).
4. **Read from Kafka**:

```
df_kafka = spark.readStream \
    .format("kafka") \
    .option("kafka.bootstrap.servers", "localhost:9092") \
    .option("subscribe", "paysim_transactions") \
    .load()
```

5. **Parse** the JSON into columns.
6. **UDF** for `predict_iso(...)`: uses `iso_forest.predict(features)`.
7. **Filter** or mark anomalies: `IF_Anomaly = 1 if pred == -1 else 0`.
8. **Write** flagged rows to CSV via:

```
df_fraud.writeStream \
    .outputMode("append") \
    .format("csv") \
    .option("path", "/home/monishaapatro/flagged_anomalies") \
    .option("checkpointLocation", "/home/monishaapatro/checkpoints") \
    .start() \
    .awaitTermination()
```

6.6 Google Cloud Platform

We used a **Compute Engine** VM for:

- **Installing** Kafka, Spark.
- **Running** the producer script, the Spark streaming job.
- Because it's often cheaper or straightforward than other clouds, but the same approach can be done on AWS, Azure, or local servers.

---

7. INSTALLING DEPENDENCIES AND SETTING UP

7.1 Dependencies

- **Java** (needed by Kafka & Spark)
- **Python 3.11**
- **Pip packages**: `pandas`, `numpy`, `scikit-learn`, `streamlit`, `joblib`, etc.
- **Spark**: downloaded from apache.org (like spark-3.5.4-bin-hadoop3).
- **Kafka**: downloaded from apache.org or used `kafka_2.13-3.5.1.tgz`.

7.1.1 Installing Kafka

1. Download the `.tgz`.
2. Extract to `~/kafka_folder`.
3. Start Zookeeper, then start Kafka server:

```
bin/zookeeper-server-start.sh config/zookeeper.properties
bin/kafka-server-start.sh config/server.properties
```

7.1.2 Installing Spark

1. Download the Spark binary.
2. Extract to `~/spark`.
3. Add environment variables or just call `~/spark/bin/spark-submit`.

7.2 Uploading Models and Dataset

- Transferred `iso_forest.pkl`, `Transactions.csv`, etc., to the VM using either `scp` or the GCP browser-based SSH's "Upload file."
- Placed them in our home directory or specific folders.
- Confirmed paths in scripts: e.g., `CSV_PATH = "/home/monishaapatro/Transactions.csv"`.

---

8. FLAGGED ANOMALIES AND HOW THEY ARE DETERMINED

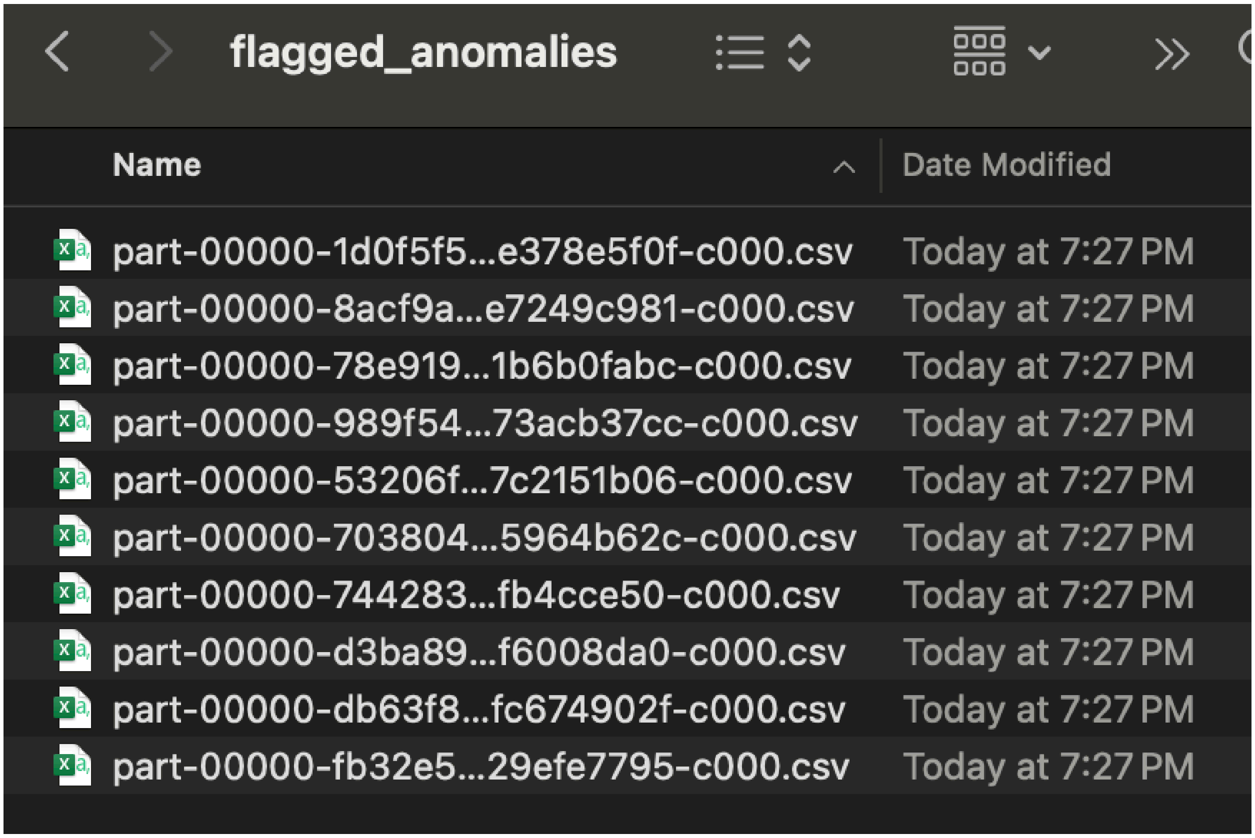
1. **IsolationForest** returns `predict(...)` = `1` for anomaly, `+1` for normal.
2. We convert `1 → 1 (IF_Anomaly=1)` meaning flagged as suspicious.
3. The script effectively says: “If anomaly, print or filter to a separate DataFrame.”
4. Spark’s streaming job writes those anomaly rows to **CSV** (the “flagged_anomalies” folder).

Hence, “flagged anomalies” are simply transactions that the model deems suspicious, typically because they deviate from normal patterns.

---

9. TRANSPORTING ANOMALIES TO CSV

1. The final `df_anomalies` (or `df_fraud`) is written out by `df_anomalies.writeStream` with `format("csv")`.
2. Each micro-batch creates a `part-00000-xxxx.csv` file in the target folder.
3. `_spark_metadata` is also created for housekeeping.
4. This yields a growing list of CSV files each time Spark processes new data from Kafka.



---

10. USING STREAMLIT TO PRESENT THE DATA

10.1 Approach

We created a **Streamlit** app (`dashboard.py`) that:

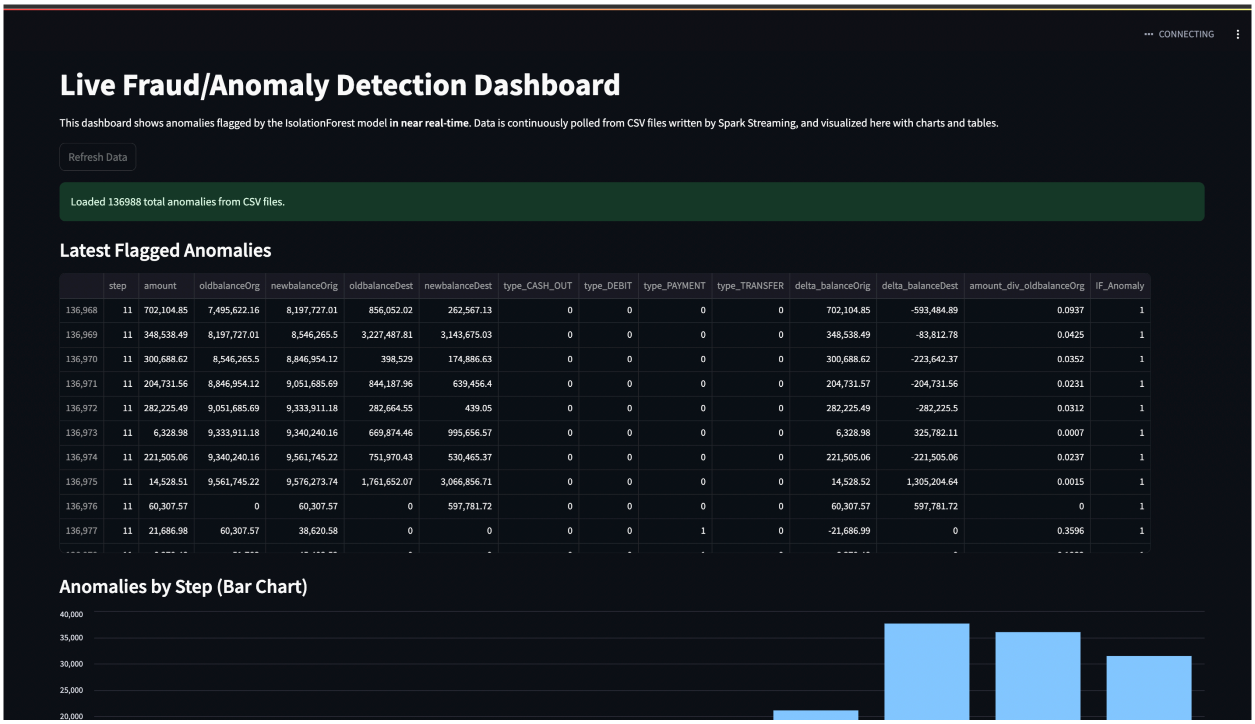
1. **Loads** the CSV files from "flagged_anomalies/" (or whichever directory Spark is writing to).
2. **Concatenates** them to gather all flagged transactions.
3. **Visualizes** them with line charts, bar charts, tables, etc.

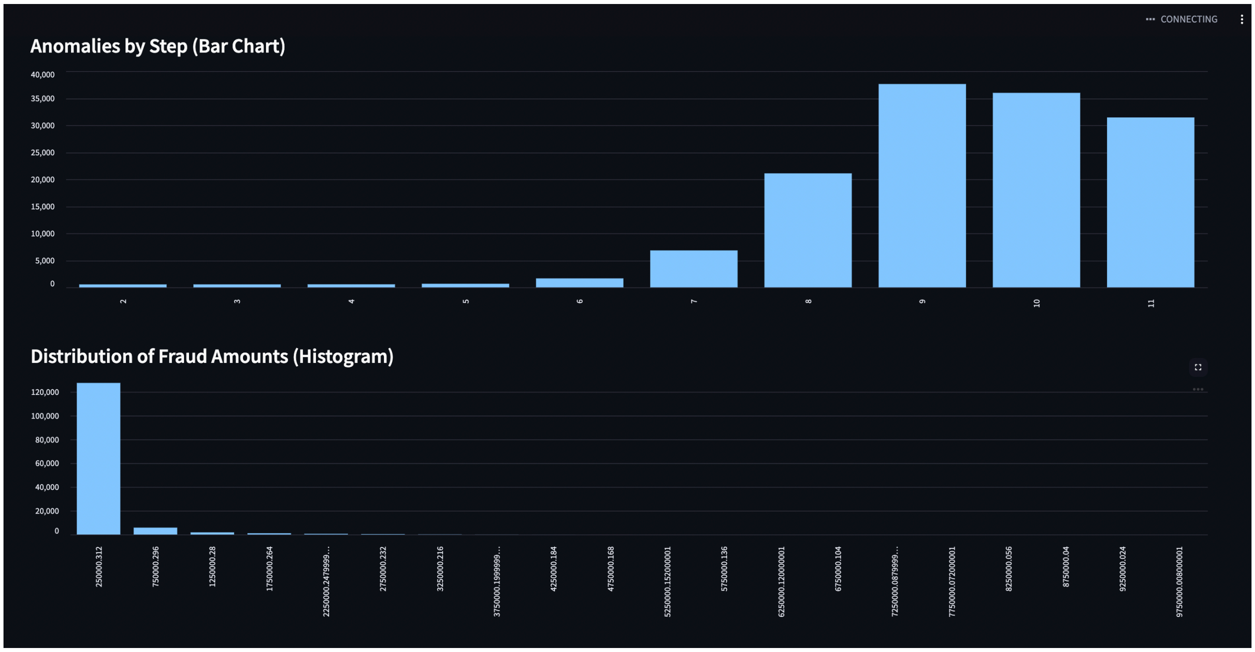
10.2 Real-Time Refresh

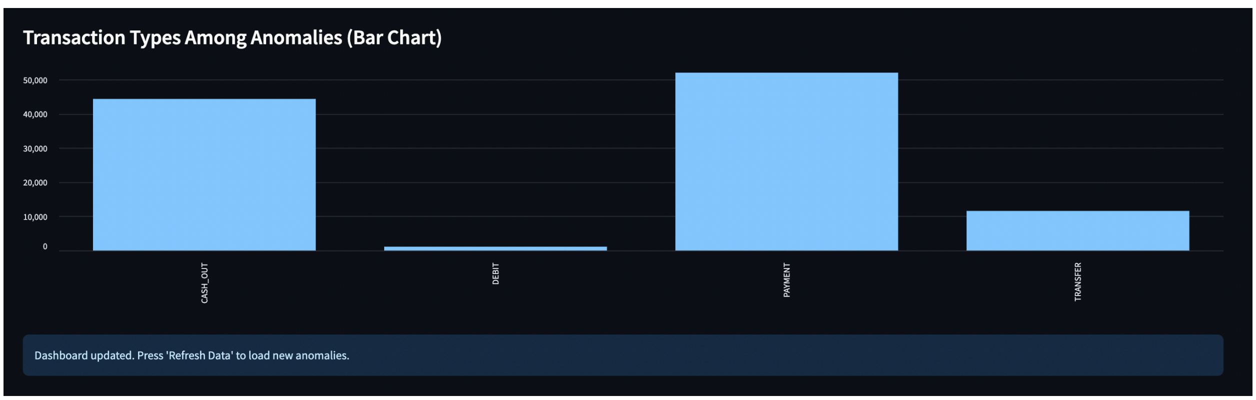
Because Spark is continuously writing new part-*.csv files, the Streamlit script can periodically re-read the folder. For instance:

```
def load_anomalies_data():
    files = glob.glob("/home/monishaapatro/flagged_anomalies/*.csv")
    ...
    # read + concat data
    return df
```

A “Refresh Data” button in Streamlit might call `load_anomalies_data()` again. The user sees updated anomalies in near real time.







10.3 External Access

- If hosting on the **same** GCP VM, we can open the firewall for port 8501 and share the IP.
- We **deploy** to **Streamlit Community Cloud** with static sample data.

---

11. BENEFITS OF THE PROJECT

1. **Rapid Detection**:
    - Instead of batch detection after the fact, suspicious transactions can be flagged within seconds.

2. **Scalability**:
    - Kafka + Spark can handle millions of events per day.
    - The unsupervised ML approach can be extended to more advanced or ensemble methods.
3. **End-to-End Pipeline**:
    - Showcases ingestion, stream processing, ML inference, and final visualization—a full data engineering + data science solution.
4. **Demonstrates** skill in:
    - Big data tools (Kafka, Spark)
    - Cloud environment (GCP)
    - Machine learning (IsolationForest, LOF, Autoencoder)
    - Dashboarding (Streamlit)

---

12. FULL LIST OF SKILLS, TOOLS, FRAMEWORKS

- **Python** (3.11)
- **Kafka** (Apache Kafka 2.13+ distribution)
- **Spark** (Spark 3.5.4 with Structured Streaming + Kafka connector)
- **Scikit-learn** (for isolation forests, LOF)
- **TensorFlow**/Keras
- **pandas**, **numpy**, **seaborn**, **matplotlib** (EDA, data wrangling)
- **Streamlit** (for real-time dashboard)
- **Google Cloud Platform**
    - **Compute Engine** VM
- **Git**, **GitHub** (version control)
- Potentially **tmux** or **systemd** for running scripts on the VM continuously

- If using the Kaggle API, **kaggle** CLI.

---

13. CONCLUSION

This project demonstrates a **complete pipeline** for **Real-Time Fraud Detection**:

1. We **ingest** synthetic transactions from the PaySim dataset in a streaming manner using Kafka.
2. **Spark** processes each transaction, applies an **IsolationForest** anomaly detection model, and flags suspicious ones.
3. These flagged anomalies are **written** to CSV.
4. A **Streamlit dashboard** reads those CSVs, producing interactive charts and tables to help investigators or analysts see suspicious transactions in near real time.
5. Everything is deployed on a GCP VM, showcasing **cloud** and **big data** skills.

Despite being synthetic, the **PaySim** dataset is large enough to **replicate** real production-scale challenges. This blueprint can adapt to a real financial system by hooking into actual transaction streams, training a more robust model on real data, and employing advanced security measures.

---

14. FUTURE WORK

1. **Ensemble Models**: Combine multiple anomaly detection models (e.g., autoencoder + isolation forest) for improved precision.
2. **Incorporate Additional Features**: E.g., geographic location, device info, historical user patterns.
3. **Real Production**: Use a real streaming pipeline with secure authentication, encryption, and robust monitoring.
4. **Active Learning**: Collect feedback from fraud investigators to refine the model.

5. **Advanced Visualizations**: Implement a more dynamic real-time chart that auto-refreshes every few seconds.
6. **Auto-scaling**: Deploy Kafka + Spark on a cluster.
7. **Deployment**: Containerize with Docker + use a platform like Cloud Run or Kubernetes for fully managed scaling.

---

15. BONUS SECTIONS OR ADDITIONS

- **Performance Tuning**:
    - Spark micro-batch interval configuration, partitioning strategies for Kafka.
    - Model hyperparameter tuning for isolation forest.
- **Security**:
    - SSL/TLS for Kafka, firewall rules for the GCP VM.
- **Data Governance**:
    - Real deployment must respect KYC (Know Your Customer), AML (Anti-Money Laundering) regulations.

FINAL REMARKS

Built a **multi-technology** solution that merges **data science** with **data engineering**. Even though PaySim is synthetic, we have gained experience with:

- **Large-scale** data (6 million+ rows)
- **Real-time** streaming using Kafka and Spark
- **Unsupervised** ML approaches for fraud detection
- A **web dashboard** (Streamlit) for easy monitoring
- **Cloud hosting** on GCP

This sets the stage for to handle real production pipelines, adapt to more advanced fraud detection, and integrate with sophisticated data or security frameworks.