Fintech as a Service

Manav Dhelia
University of Massachusetts Amherst
Amherst, MA, USA
mdhelia@umass.edu

Connor McGarry
University of Massachusetts Amherst
Amherst, MA, USA
ccmcgarry@umass.edu

Dominic Uva
University of Massachusetts Amherst
Amherst, MA, USA
duva@umass.edu

Abstract

As the Fintech landscape continues to grow, security challenges are becoming more complex and critical. We present a solution that tracks and analyzes consumer transaction patterns from Plaid in real-time, identifying suspicious activity to prevent fraud and enhance security. The system continuously monitors transaction behaviors using Kafka, flagging any anomalies or deviations from customer preferences . This proactive approach provides users with the tool to detect and respond to potential security threats before they escalate, ensuring a safer and more trustworthy environment for customers.

1 Introduction & Motivation

In the world of FinTech, the consumer often maintains several bank accounts across multiple financial institutions such as checking, savings, credit, etc.. Despite this, traditional fraud alert systems remain to be specific to each institution and are not very flexible. A user may receive a fraud alert from one bank for a large transaction but receive no alert for a similar transaction at a different bank. This is disjointed and insufficient for modern users who expect real-time customization security alerts.

Our research addresses these limitations by introducing a real-time, user-driven alerting system that is flexible enough to work across multiple bank accounts and capable of being used by multiple users. Through the use of Plaid, we can receive transaction data from a wide variety of financial institutions and send them to a unified stream. When using Kafka, we process streams in real time to filter and flag transactions based on customization criteria like specific keywords, merchants, times of day, and/or location of purchase. This system allows the user to have total control and visibility over any financial activity regardless of the institution in order to warn of fraud detection adapting to your personal preferences.

2 System Overview

The system is designed such that multiple users, each with a number of bank accounts connected to Plaid, can get alerts when there is a purchase on their account that does not match their preferences. The user enters their preferences for location, purchase amount, currency, and item category to an application where these values are used to determine whether the system should notify the user for each transaction observed on connected accounts. Webhooks are used to fetch transaction data and send to the Kafka producer, where the system operates on a Kafka Stream, enabling realtime processing. Our Kafka consumer operates in a way to represent a user notification and consumes the processed data from the application.

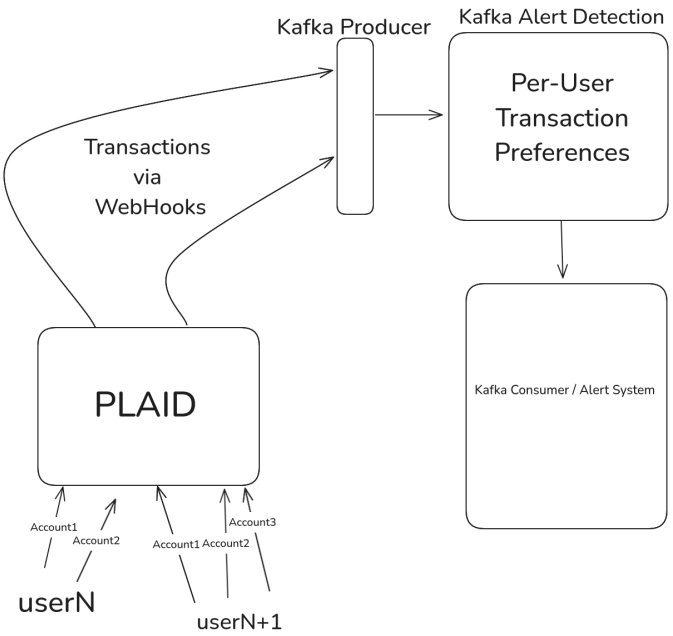

Figure 1: System Overview

We use Plaid due to the detailed financial interface, and we use Kafka due to the distributed storage nature such that we can read and process a massive amount of transaction data. This implementation runs locally; however, we develop such that it can be expanded and added to a cloud environment with ease.

Fig.1 describes the system such that N users can enter an arbitrary number of accounts to the system and be alerted beased on their specified preferences

3 Implementation

3.1 Plaid

3.1.1 Overview

As mentioned previously when creating our fraud alerting system that can work on multiple financial institutions, one of the most significant challenges is being able to get real-time data from diverse banking systems. Each financial institution typically has its own services, API, unique formatting types, and limitations. Plaid is able to solve this problem by acting as one system that can interface with all financial institutions creating its very own ecosystem. Plaid is able to abstract all the differences of the different institutions and

creates a consistent, normalized data set for transactions, accounts, balances, etc..

3.1.2 Authentication

Plaid's access control uses a key-based system, and an example of this is how our system is authenticated using a unique client-id and secret which is issued by plaid when you create your developer account. The credentials are used to identify and authorize the backend service allowing us to make our own API calls on behalf of the end users in our system. They are application-wide, which means that they are the same for all users of the system and are meant to be kept secure.

The user-specific component of Plaid's authentication begins once a user links their financial institutions. This is handled through Plaid Link which is their client-side mechanism that collects the credentials and bank authorizations. In our implementation, we were able to simulate this through the creation of our temporary public_token through the public token create endpoint. This public_token is a short-living authorization token which is meant to confirm that the user has granted the application access to the specific institution and account.

This public_token is sent over to the backend system, which combines it with the client_id and secret to invoke the public token exchange endpoint meaning that there is an exchange for an access_token. This access_token is the key piece in all of the next API calls meant to retrieve transaction history, account metadata, and real-time updates. Each of these access_tokens are linked to specific users financial accounts.

Overall, this layered authentication model with the application level keys (secret, client_id), and the user-level keys (public_token, access_token) ensures that there is a trust between the individual user and the system. This is what allows for secure and scalable access to the sensitive financial data without compromising anything on a global level.

3.1.3 Transactions & Webhooks

Following the authorization of a user's financial account through Plaid API, we now need to begin to initiate transaction data using the user-specific access tokens. The token is a persistent key that enables the backend to access all of the financial activity across the institutions that the user has linked to their account without the requirement of constant user interaction. In short, this is the system we are using in order to meet the design goals of having a system work in real-time through the use of sending transaction data to a specific server.

Our initial process of getting transaction data from plaid is through the transaction get endpoint which essentially will return a structured response containing a chronological record of financial transactions across all the accounts. By default we are only able to get this system to print out a fixed set of transaction data over any given 30 day window which is a constraint of plaid's sandbox testing infrastructure. We then do some formatting in order to get the data readable by Kafka. Each of these transaction objects includes a plethora of attributes such as the date, time, amount, merchant name, associated categories, location, etc.. This type of information is crucial to our fraud detection system as it provides the simulated data exactly how we would see if someone were to make an actual transaction.

Since this transaction data is fixed and only simulated in the sandbox we are unable to actually create our own real-world simulation cases where fraud is occurring to check our systems functionality. In order to solve this problem we utilized the Plaid API webhooks function which allows us to trigger our own events that demonstrate an example of fraud and check the responsiveness of our system.

Plaid webhooks are generally used to notify the backend when specific events occur in this case, a new transaction. These notifications are delivered as HTTP POST requests to a developer-defined endpoint. In our test environment, the events we have created simulated and follow the same structure of a real transaction returned by Plaid API. Our system exposes a webhook receiver endpoint using the Plaid webhook function through the use of our Express.js server. When receiving the payload, the server would then parse the data to match the same way we did it in the transaction format to be used to run smoothly in Kafka. In summary, we essentially are manually creating different scenarios in JSON format that is sent as a webhook to the main server where kafka is able to retrieve that data and process it.

3.1.4 Kafka Integration

One primary advantage of using Plaid API is for it's data normalization layer which is what makes it possible to format the data from multiple financial institutions in its own formatting schema which is JSON-Based allowing for easy integration with Kafka.

To enable scalability in our project, and be able to process transaction data in real time Plaid needs to send it's transaction data to Kafka. In order to do this we need to use Express.js server which uses the kafkajs library. When the server is starting up, it initializes a Kafka producer and establishes a persistent connection to the broker hosted on port 9092. Once this transaction dataset is sent through the webhook payload, the JSON structured object which is formatted down to one line will run on Kafka to begin the process steps.

3.2 Kafka

3.2.1 Overview

Apache Kafka is a data streaming engine used to collect, process, store, and integrate data at scale, where a stream represents an unbounded, continuously updating data set. We use Kafka to enable a stream processing application that checks transaction data against inputted user preferences in real-time. Kafka allows for information sent to a consumer to be stored for a user-defined amount of time, so data processed can be accessed long after it has been processed in real time. This makes it ideal for deployment over multiple clusters in a cloud environment.

3.2.2 Topics

Topics in Kafka are multi-producer and multi-subscriber, where producers are what write events and consumers are what read events. This means that the same topic can have many streams in and many streams out, making it ideal for our use case. Our implementation initializes two topics: *transactions-input* and *transactions-output*.

3.2.3 Plaid Integration / Stream Processing Application

We use Plaid to send transaction data to port 9092 on our sever for real-time transaction processing. This works alongside Kafka streams as we are able to process data as soon as it is sent from the Kafka server upon a new transaction. We selected this method over alternatives such as writing from Plaid to a file and then reading from that file with Kafka Connect. This is to eliminate local storage on devices and rely on a direct handoff from Plaid to Kafka to store data in Kafka brokers in a distributed manner. Kafka is designed to store large amounts of data, and this is something to take advantage of when working with multi-user transaction data.

We make use of the Kafka API for Java, specifically libraries within *org.apache.kafka.streams* which enable interfacing with stream I/O. Within our application, we define the input and output topics of where the data needs to be read and sent as seen in *4.2.2*. Based on information found in Plaid transactions, we allow for a user to select categories that create a flag if found in their transaction data. These categories relate to stores, currencies, cost, and more as seen in Fig. 2.

```
TRANSFER_OUT_SAVINGS
TRANSFER_OUT_WITHDRAWAL
TRANSFER_OUT_ACCOUNT_TRANSFER
TRANSFER_OUT_OTHER_TRANSFER_OUT
LOAN_PAYMENTS_CAR_PAYMENT
LOAN_PAYMENTS_CREDIT_CARD_PAYMENT
LOAN_PAYMENTS_PERSONAL_LOAN_PAYMENT
LOAN_PAYMENTS_MORTGAGE_PAYMENT
LOAN_PAYMENTS_STUDENT_LOAN_PAYMENT
LOAN_PAYMENTS_OTHER_PAYMENT
BANK_FEES_ATM_FEES
BANK_FEES_FOREIGN_TRANSACTION_FEES
BANK_FEES_INSUFFICIENT_FUNDS
BANK_FEES_INTEREST_CHARGE
BANK_FEES_OVERDRAFT_FEES
BANK_FEES_OTHER_BANK_FEES
ENTERTAINMENT_CASINOS_AND_GAMBLING
ENTERTAINMENT_MUSIC_AND_AUDIO
```

Figure 2: Snippet of Plaid Transaction Categories

Each user, and each account for each user, has a mapping to a unique ID generated by Plaid. Due to this, it is possible to create a database mapping user to userID, and therefore allowing for per-user and per-account category selection. However, due to the sandboxed environment of Plaid that we are working with, this feature is unintuitive to implement due to the dynamic nature of the IDs each iteration of the system. In an ideal environment, there are constant IDs that can be permanently mapped to their preferences.

As such, the current state of the application relies on global preferences, meaning that each user and account have the same categories that generate flags.

The application operates on the JSON output from Plaid and searches through the file for Plaid-defined keywords. We then find the item mapped to these keywords and cross-check against the user preferences for alert-generating categories in order to determine if we should send an alert to the customer or not.

These alerts are sent to an output topic which formats them such that the user knows what account the alert was generated for and why the alert was generated, an example of which is as follows: *SUSPICIOUS PURCHASE ALERT ON ACCOUNT X (At store Gucci Online for the amount of 1500.00 EUR. Reason: Unapproved Currency; Exceeds max purchase amount).* This represents an alert sent to a user regarding an alert on their account for an expensive purchase at a store. This user had selected USD as the only valid currency for one of their alert requirements, as seen by the *Unapproved Currency* flag in the alert.

In production, this system would be more robust with an actual user UI rather than hard-coded constants, however this represents a functional alert system using real data from Plaid being processed by Kafka.

4 Testing and Results

The goal of our tests is to prove that we can get data from a user transaction from Plaid and flag the transaction if there is a keyword found that the user has blacklisted. First we run the Kafka server and application for data processing. This is active by default at port 9092 locally. Next, we initialize the topics and enable a consumer so we can see the processed data from the application. Our testing process is to use webhooks to fetch real Plaid transaction data. These webhooks output data in JSON form to our Kafka server port. At this port the data is processed through our stream application. Using a variety of transaction types, were are able to demonstrate successful Kafka I/O that represents user notification upon flagging of a bad transaction. The following user-defined preferences are used in our testing:

```
private static final Set<String> BAD_CATEGORIES = Set.of(
    "GENERAL_MERCHANDISE_PET_SUPPLIES",
    "MEDICAL_VETERINARY_SERVICES",
    "GENERAL_MERCHANDISE_OTHER_GENERAL_MERCHANDISE"
);

private static final Set<String> ACCEPTABLE_CURRENCIES = Set.of("USD");
private static final Set<String> BAD_STORES = Set.of("Starbucks", "McDonald's", "Uber");
private static final int PURCHASE_MAXIMUM = 1000;
```

Figure 3: User Requirements

See project video demo for a visual representation of this process where we initialize the server for Kafka, create the topics and run the application. Next, we open a consumer to wait for the data to come in from the Plaid server.

To achieve multi-user, we can access the Kafka stream from multiple sources, meaning multiple Plaid servers. This is unintuitive in practice, but representative of reality given the sandbox environment that we are working in.

The webhooks are sent to the Kafka server via shell scripts that can be found in the included github with this project. The resulting JSON data are compacted to one line to feed into Kafka easier. This is because Kafka defaults to operating on a line-by-line basis, making it difficult to process events. Fortunately, JSON data are designed to be formatted in a variety of modes, so this does not prove to be an issue.

Once the data has been received by Kafka, it is operated on by the Java stream processing application. This application uses the Kafka APIs to read data from a topic and output data to a different topic.

The output topic - transactions-output - is used to display the flagged transaction information. Of course, there are ways to display each transaction regardless of flag, however in this case a user would only be notified in a bad situation based on their preferences. As in the video, we run a consumer that reads from the output topic from the application. This consumer can be run at any time using the –from-beginning flag to see a full history of all data written to that topic.

The example user ID, which is represented by '*ojBrKa8q...*' changes each time a new user account is added to Plaid, and this remains true for the sandboxed mode. This data represents a financial institution for a single user, of which there can be many.

Again, in a real deployment there would be a larger mapping of user to account with secusity features such that there are unique preferences for each user. In the video we can see that three transactions were sent, and two of them were flagged for various reasons such as unacceptable currency.

4.1 Accuracy

The accuracy of our system should be perfect as we are relying directly on categories defined by Plaid to send our alerts. This means that as long as Plaid has a consistent formatting for their transaction data, which they do, then the stream processing application just needs to find the correct text data.

To test this, we compared the number of alerts that were sent by Kafka with the number of 'bad' transactions that we created via Plaid. This means that we should get the same number of alerts as our known number of bad transactions.

This testing resulted in 100% accuracy as expected due to the reasons mentioned above.

4.2 End-to-End Latency

To test end-to-end latency, we use Plaid to generate a timestamp in the transaction data. Using this timestamp in the JSON file we fetch, we can find the latency by comparing it to the time that Kafka outputs the data to the alert topic.

The round-trip latency for only one server is 2.1 milliseconds for 10 transactions and 100 transactions all on one server. After increasing the amount of transactions to 1000 it stays roughly the same at 2.2 milliseconds. When going up to 10000 and 20000 transactions, the latency goes up to 2.5 milliseconds and 6.4 milliseconds. We have also tested by splitting the load onto more servers

All of the following testing was recorded using two servers with the load split evenly run three different times. In the low-concurrency setting which was 5 users per server simultaneously, the system consistently achieved around 1.2 millisecond response time with barely any variation. After increasing the total users to 100 concurrently the average time ranged around 2 milliseconds. Notably, we can see that when increasing from 1000 concurrently to 10000 concurrently there is little real degradation in the time with the average latency being around 1.7 milliseconds and 1.5 milliseconds for 1000 and 10000 respectively. Compared to the single server the split load of transactions actually stays close to standard at 1.9 milliseconds meaning that this load balancing reduces total latency. Of course there are some outliers with some running at 1 millisecond and others running at 20 milliseconds. The results are shown to demonstrate the consistent low-latency between receiving plaid data to being able send an alert on Kafka.

This latency would be expected to decrease when running on the cloud in different instances and with more server to spread out the load properly.

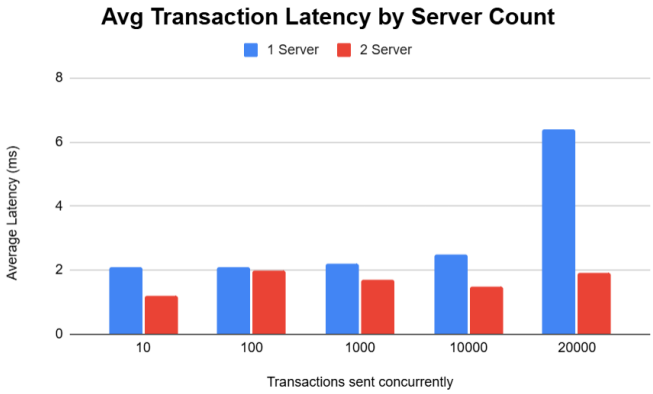

Figure 4: System Latency

4.3 Scalability

We test scalability such that our system should be able to handle many users at once. This means that if a large number of users are purchasing an item at once, nobody will experience unreasonable delays.

It is important to note that Kafka should be deployed in the cloud for a system like this such that there are many clusters. In this case, the latency for each user would be reduced as there is a different instance of the stream processing application running for a company-defined group of users of size N.

In our testing, as seen in the video demo, we show that we can initiate multiple 'users' by starting multiple servers to fetch transaction data from Plaid. In our case, we demonstrate 2 servers running with multiple users each. These servers are tested with up to 50,000 users each where we initialize 50,000 get requests into an array, and then send these requests in parallel to the Plaid servers. The order in which they are handled is up to the scheduler, however each transaction goes into the input Kafka topic for processing.

This shows that we can not only add more sources for the Plaid data, but also add more transactions per user per source, all running at the same time, without the system breaking. This will only improve in the ideal implementation in What is the average monthly bills for a cloud environment.

5 Challenges & Limitations

One of the primary challenges involved with trying to create a system that utilizes the Plaid API is the fact that we are limited to the free to use sandbox mode. This is exactly why we need

to use webhooks to send our fake transaction data as opposed to using the transaction get endpoint because we did not want to be constrained to a fixed set of static, pre-generated transactions. Although we were able to use this to test the responsiveness of incoming data, we didn't have the ability to use the get functions that are used in a standard system to print out the actual transaction data. In a higher-tier development environment we would've had more access to custom st transaction behavior and overall more supporting functions.

In terms of overall limitations of our system, we have found that some of the vendor information on Plaid do not have all the necessary data information for what we are looking to flag. This is usually found in smaller vendors who will have the minimal amount of categories and metadata due to cost, scale, or technology constraints which would mean that a flag could go under our nose. Another limitation is how robust the streaming application can be given the challenge of sandboxing discussed above. This means that the user preferences are hard-coded and not automated for the addition of multiple users and multiple accounts. This can easily be fixed once using a paid version of Plaid by setting a map between user, user accounts, and Plaid user IDs, such that each user has their own account in our application. Again, this is meaningless to present as part of this system as our focus is Plaid and Kafka integration rather than the UI.

6 Future Work

While our current system demonstrates the proof of concept behind a real-time fraud detection system using Plaid and Kafka, there are many ways we can go about improvements for the future.

The first of which that would increase the overall scalability of our system would be to have more persistent storage of the access tokens. Our current system stores the tokens in memory and are handled manually which would limit the amount of users we can actually use in the system. By putting these tokens into a database to be stored would mean that the system could dynamically store and retrieve the tokens, allowing support for more concurrent processes.

Another example of how we could increase the functionality of our fraud detection system would be to use machine learning on the historical transaction data to identify which transactions are outside of the usual to add more layers of our fraud detection indicators. In theory this would give us more data on how to adapt our filtering system and reduce the likelihood of any false positives.

Our next step would be to improve the overall user interface. Right now we are only able to interact with the system through manual scripts and command line tools. Through creating either a website with a dashboard of all your data or even a mobile able that could send you personalized alerts depending on what flags you had set up to see. Another way to enhance usability would be to actually have an interface that allows you to select the exact type of flags you want to be alerted of making it clean and easy for anyone to log in.

Of course, the goal of this system would to be deployed in the cloud. This was we could use the full power of Kafka and the other resources were are using in the project. This way we could store the transaction history of each user, and provide them with information like a review of fraud each year.

The last thing would be to increase the security of our system through encrypting any of the keys we used in our application and ensuring that this data is not stored locally on the primary device it is running off of.

7 Conclusion

We demonstrate the core functionality of a system that could be built on given the necessary time and money, as discussed in section *6*. Despite the lack of a user-facing system, we were able to create an ecosystem of real-time financial transaction processing. We learned the process of working with massive API libraries, of which takes time to absorb the necessary documents. This was our first time working with Java and webhooks, so the entire system environment represents a firsttime exposure to these tools. The choice to use Kafka was important due to the realtime nature of the project, but also given the context of a distributed application that would be exposed to a massive number of users in a server deployment manner. We acknowledge the limitations of the system, however we are satisfied with the results given the limited access to production-level tools and accounts for testing and development.

References

[1] Kreps, Narkhede, Rao, *Kafka: a Distributed Messaging System for Log Processing*. [Online]. Available: https://notes.stephenholiday.com/Kafka.pdf
[2] Apache Kafka, "Apache Kafka Documentation." [Online]. Available: https://kafka.apache.org/
[3] Plaid, "Plaid API Reference." [Online]. Available: https://plaid.com/docs/api/
[4] Confluent, "Kafka APIs." [Online]. Available: https://docs.confluent.io/kafka/kafka-apis.html