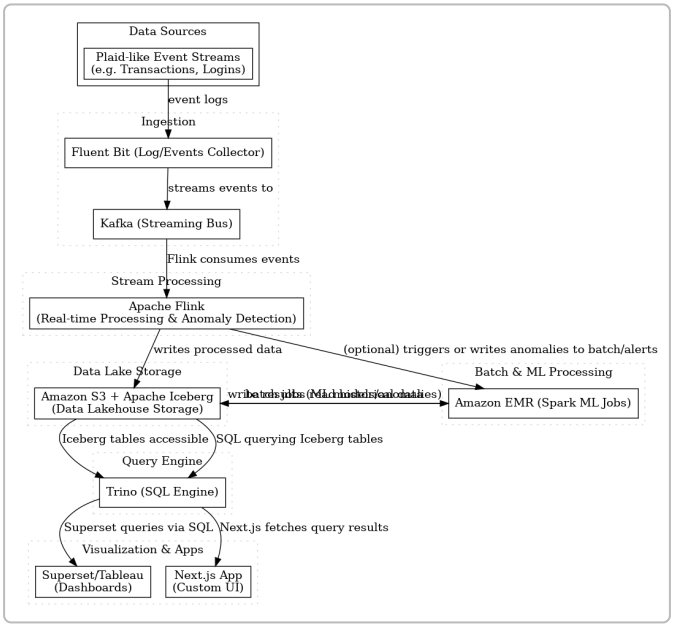
ChatGPT

Fraud Detection Data Pipeline – End-to-End Prototype Plan

Overview: This project outlines a full-stack data pipeline for real-time fraud detection on Plaid-like financial event data. It replicates a modern data architecture (similar to Tableau's data stack) from ingestion to dashboard. We combine streaming analytics for instant anomaly detection with batch processing for deeper ML insights, all feeding into a unified data lake and visualization layer. The stack includes **Fluent Bit + Kafka** for event ingestion, **Apache Flink** for stream processing, **Amazon S3 + Apache Iceberg** for a data lakehouse, **Amazon EMR (Spark)** for batch/ML jobs, **Trino** for SQL querying, and **Superset/Tableau** (with an optional **Next.js** web app) for visualization.

Architecture Overview



End-to-end fraud detection pipeline architecture (streaming & batch). The pipeline ingests financial events (e.g. transactions) in real-time, processes them to detect anomalies, stores both raw and processed data in a data lake, and provides interfaces for querying and visualization. Key stages:

- **Ingestion:** Plaid-like event streams (transactions, logins, etc.) are collected by Fluent Bit and fed into a Kafka streaming bus for high-throughput, reliable delivery [1].
- **Stream Processing:** Apache Flink consumes events from Kafka and performs real-time anomaly detection (applying rules or models). Detected fraud alerts are immediately emitted (e.g. to an `alerts` Kafka topic) and also written to the data lake for persistence and analytics [2].

- **Data Lake Storage:** All events (and alerts) land in an S3-based data lake, stored in Apache Iceberg table format. Iceberg provides an **open lakehouse** layer with ACID transactions and schema evolution on object storage [3].
- **Batch/ML Processing:** Amazon EMR (Spark jobs) periodically reads large historical datasets from the Iceberg tables on S3 to train and run ML models for fraud detection. Batch jobs can enrich data, compute features, and detect anomalies that require heavy computation or longer time windows, complementing the streaming detection.
- **Interactive Query:** Trino (PrestoSQL) serves as a SQL query engine on top of the Iceberg tables. It enables fast, distributed querying of the data lake without data movement [4] [5]. Both the BI dashboards and custom applications use Trino to fetch insights.
- **Visualization & Apps:** Business intelligence tools like Apache Superset (or Tableau) connect to Trino to create dashboards, charts, and real-time reports. Additionally, a custom Next.js web frontend (deployed on Vercel) can call APIs or Trino to display fraud metrics and live alerts in a tailored UI.

This architecture ensures that fraud can be identified *as it happens* and also analyzed over historical data. By processing data in motion instead of waiting for batch ETL, the system can "catch fraud as it happens," providing a game-changing real-time defense [6]. At the same time, the data lakehouse stores comprehensive history for offline analysis and model training.

Data Ingestion with Fluent Bit and Kafka

Sources & Fluent Bit: We simulate financial event sources such as transaction logs or Plaid webhooks. Fluent Bit (a lightweight log/data collector) is deployed on application servers or as a sidecar to capture these event logs in real-time. It streams the events to Kafka using the Fluent Bit Kafka output plugin [1]. Each event might be a JSON message representing a financial transaction or account activity.

Kafka Stream: Apache Kafka serves as the high-throughput ingestion bus buffering and distributing events. We create dedicated topics, for example: a `transactions` topic for incoming events and a `fraud-alerts` topic for downstream alerts. (In a production setup, additional topics can segregate different event types or serve as dead-letter queues.) Kafka's durability and scalability ensure it can handle the volume of Plaid's financial event stream.

Rationale: Decoupling ingestion via Kafka provides backpressure handling and decouples event producers from consumers. This matches modern streaming ingestion practices (Kafka is used by ~80% of Fortune 100 [7]) and reflects Tableau's ability to ingest from various sources into a unified pipeline. Fluent Bit adds minimal overhead and reliably forwards events to Kafka.

Real-Time Stream Processing with Apache Flink

Flink Stream Job: Apache Flink continuously consumes the transactions stream from Kafka and applies real-time fraud detection logic. Flink is chosen for its high throughput, low-latency processing and exactly-once stateful stream handling, which are essential for mission-critical fraud detection [8] [9]. Key functions of the Flink job:

- Parse and enrich raw events (e.g. add geo-info or user metadata if needed).
- Apply **fraud rules and anomaly detection** in real time. For example, the job can implement rules such as: (1) *geographic distance check* ("impossible travel" — same user making transactions far apart

within minutes), (2) *multiple IP addresses* (user appearing from many IPs in short time), (3) *suspicious spending pattern* (small test charges followed by a large amount), (4) *velocity check* (many high-value transactions in a minute), (5) *frequency threshold* (too many transactions in an hour) [10] [11]. These rules can be encoded via Flink's CEP library or sliding windows and keyed state for each user.

- Compute anomaly scores or apply an ML model on the fly. (For instance, if a pre-trained ML model is available, Flink can load it to score each transaction's fraud probability in real-time.)
- **Emit fraud alerts:** When an event triggers a rule or is scored as anomalous, Flink produces a fraud alert event. These alerts are sent to a Kafka `fraud-alerts` topic for immediate consumption by other services (e.g. to block a transaction), and also written into an Iceberg table for record-keeping and analysis.

Flink's streaming job maintains state (e.g. recent transactions per user) and uses event-time processing to accurately detect patterns (handles out-of-order events with watermarks) [12] [13]. It provides fault tolerance via checkpoints, so the pipeline can recover without losing events or creating duplicates (exactly-once end-to-end guarantees). This ensures robust processing even under failures [8] [9].

Dynamic Updates (optional): To mirror a sophisticated system, the Flink job could support dynamic rule updates. For example, a control topic can deliver new fraud rules to Flink at runtime [2], allowing the rules/thresholds to be updated without stopping the job. This uses Flink's broadcast state or similar patterns (as demonstrated in a Flink fraud demo [2]), giving flexibility to adapt detection logic on the fly.

Data Lake Storage with S3 and Apache Iceberg

All ingested events and results are stored in a **data lake** on Amazon S3, using Apache Iceberg as the table format. Iceberg acts as the "single source of truth" storage layer for both streaming and batch data. We define two primary tables in the Iceberg metastore (using AWS Glue or Hive Metastore for catalog):

- `transactions_table` – storing raw (or minimally processed) transaction events. Schema includes fields like `transaction_id`, `user_id`, `account_id`, `amount`, `timestamp`, `location_lat/long`, `ip_address`, etc., reflecting the structure of Plaid events. For example, an incoming transaction event JSON might look like:

```
{
  "userId": "user123",
  "amount": 500.00,
  "timestamp": "2024-01-29T10:00:00Z",
  "latitude": 40.7128,
  "longitude": -74.006,
  "ipAddress": "192.168.1.5",
  "merchant": "ABC Store",
  "type": "purchase"
}
```

(This schema is based on typical fields; e.g. the Yugen.ai example includes userId, amount, timestamp, latitude, longitude, ipAddress [14].) Events are partitioned by date (e.g. transaction date) in the Iceberg table to optimize queries by time range.

- `fraud_alerts_table` – storing anomalies/alerts detected. Each alert entry includes the `user_id` (or account), details of the triggering transaction (or aggregate pattern), a `reason` or rule that was violated, an anomaly score or severity, and timestamps. For example, an alert record might be:

```
{
  "userId": "user123",
  "fraudulent": true,
  "transaction": {
    "userId": "user123",
    "amount": 50.0,
    "timestamp": "2025-01-29T19:00:57Z",
    "latitude": 40.7128,
    "longitude": -74.006,
    "ipAddress": "192.168.1.5"
  },
  "reason": "Multiple IPs: 5 distinct IPs in 30 minutes",
  "fraudScore": 20,
  "alertTime": "2025-01-29T19:01:00Z"
}
```

This example shows a user flagged for using 5 different IP addresses in 30 minutes [14]. The alerts table is partitioned by alert date or hour.

Why Iceberg? Apache Iceberg is an open table format that brings **database-like functionality to cloud storage**, enabling a true lakehouse architecture [3]. Using Iceberg on S3 allows us to have **ACID transactions** (so streaming Flink jobs can incrementally write to tables safely), time-travel queries, and schema evolution on our data lake. This means as fraud detection logic changes and new fields (or tables) are added, we can evolve the schema without breaking queries. Iceberg also maintains a transaction log and snapshot metadata for efficient querying (only new data files are read on incremental queries) [15]. In essence, Iceberg gives us a **reliable, consistent data warehouse layer on S3** – analogous to Tableau's fast data engine, but built on open source and cloud storage.

Iceberg Integration: Flink has native connectors to write to Iceberg tables. The Flink streaming job will use an Iceberg sink to continuously append events to the `transactions_table` (and possibly to `fraud_alerts_table` for each alert) in **near real-time** [16]. Thanks to Iceberg's design, these writes are atomic and immediately visible to downstream consumers. We configure the Iceberg tables in AWS Glue Catalog so that other engines (Spark, Trino, etc.) can discover the schema and data.

Batch Analytics and ML on Amazon EMR

While streaming covers immediate detection, batch processing on accumulated data allows deeper analysis and machine learning: this is the **"fraud analytics"** layer. We use Amazon EMR to run Apache Spark jobs (or Flink batch jobs) on the data lake. EMR provides a scalable cluster to perform heavy computations like joining large tables, training ML models, and recomputing aggregates.

Batch Use Cases:

- **Feature Engineering & Aggregates:** A nightly Spark job could compute features per user/account (e.g. average transaction amount last 24h, count of distinct IPs, total spend per merchant category, etc.) and store these as a derived feature table in Iceberg. These features can feed ML models or be used in reports.
- **ML Model Training:** Using historical labeled data (if available, e.g. known fraud cases), we can train a fraud detection model offline. For example, train an unsupervised anomaly detection model (like Isolation Forest or clustering) on all transactions to identify outliers, or a supervised model (like a gradient boosted tree) if we have fraud labels. The training runs on EMR (leveraging Spark MLlib or integrations with scikit-learn via Spark). The resulting model artifact can be saved to S3.
- **Batch Anomaly Scoring:** With a trained model, a batch job can score the latest data in bulk. For instance, every hour a Spark job could score all transactions from that hour using the ML model to generate a fraud risk score. Transactions above a certain score threshold are written to the `fraud_alerts_table` as flagged anomalies (with reason "ML anomaly score above threshold"). This complements the rule-based streaming alerts with data-driven insights.

Because the data lake is on S3/Iceberg, the Spark jobs on EMR can directly read the `transactions_table` and write results to the `fraud_alerts_table` or other output tables. Using Iceberg's snapshot isolation, batch jobs can run without interfering with the streaming writes [15]. EMR can be scheduled (or triggered by new data arrival) to run these jobs at off-peak hours or periodically.

Closing the Loop: The batch layer can also improve the streaming layer. For example, an ML model trained on EMR can be exported (perhaps as PMML or TensorFlow SavedModel) and then loaded into the Flink job to score events in real-time. This way, the more advanced batch-learned patterns are applied immediately on incoming events. This iterative improvement demonstrates full-stack integration – something a technical program manager would plan for to continuously enhance fraud detection efficacy.

Interactive SQL Querying with Trino

To enable flexible analytics and ad-hoc queries on the accumulated data, we use **Trino** (formerly PrestoSQL) as our interactive query layer. Trino connects to the Amazon S3 data lake through the Iceberg catalog, allowing SQL queries over the Iceberg tables as if they were regular database tables [17].

Trino Cluster: We deploy Trino on EMR or as a separate service (with a coordinator and worker nodes). Trino's compute-storage separation means it does not store data itself, but queries the data where it lives (in S3) using distributed execution [4]. This avoids moving data into a separate warehouse, reducing latency and cost [4] [18]. Analysts or dashboard tools can connect to Trino via JDBC/ODBC using standard SQL.

With Trino, one can for example query: *"What is the count of fraud alerts by reason in the last 7 days?"*, *"List the top 10 merchants by total fraud amount"*, or join transaction data with external reference data (e.g. a list of blocked accounts) if needed. Trino's ANSI SQL support and ability to query heterogeneous sources (could also join an Iceberg table with, say, a MySQL table if configured) make it a powerful unified query engine. In our stack, it essentially stands in for Tableau's internal query processing or any data warehouse that Tableau might connect to, but here we keep it open-source and lake-centric.

Performance Considerations: Iceberg tables can be partitioned and indexed (through metadata) to optimize Trino queries. We would partition data by date and possibly by other keys (e.g. by user region for the transactions table) to prune data scans. Trino also benefits from caching and Hive/Glue metastore optimizations. If needed, we could integrate Amazon Athena or Presto DB in place of Trino, but Trino gives more flexibility in a self-managed environment.

Dashboards with Superset / Tableau

Finally, the insights are presented through dashboards and visualizations, showcasing patterns of fraud and system performance. We propose using **Apache Superset** – an open-source BI platform – as a stand-in for Tableau (since Tableau is proprietary but the end result should be similar). Superset allows connecting to our Trino SQL endpoint and building charts and dashboards on the data [19]. Key views in the fraud dashboard could include:

- **Real-Time Alerts View:** A live table or feed of the latest fraud alerts (from the `fraud_alerts_table` via Trino). This lets analysts or support teams see incidents as they occur, including the reason flag (e.g. "High velocity: $X in 1 min") and details of the transaction/user.
- **Trends and Aggregates:** Charts showing the volume of transactions and number of alerts over time (hourly/daily). For example, a time-series line chart of fraud alerts per day, or a stacked bar chart of alert reasons by week. This helps spot spikes of fraud attempts or measure the effectiveness of detection.
- **Geographical Map:** Using latitude/longitude data, a map visualization can plot transaction locations, highlighting those flagged as fraud. This could reveal clusters of fraud origination or anomalous geo patterns.
- **User Behavior Profiles:** Drill-down dashboards that, for a given user or account, show their recent transactions and any alerts. This helps in investigating and confirming fraud cases.
- **Model Performance Metrics:** If ML is used, dashboards can show model precision/recall over time, or distribution of anomaly scores, to track false positives/negatives.

Superset provides a rich set of visualization types and a no-code dashboard building interface [20] [19]. Users can also write SQL in Superset's SQL Lab for ad-hoc analysis. Alternatively, **Tableau** could be used here to connect via an ODBC driver for Trino or by querying the data through a live connection. The choice of Superset vs Tableau doesn't affect the upstream pipeline; it mainly changes how the final charts are created. The goal is to demonstrate that the data pipeline can feed any modern BI tool.

To replicate a Tableau-like experience, we ensure the data served to the BI layer is optimized for fast querying (hence the pre-aggregation in batch jobs and the use of a query engine like Trino). The dashboard layer is where a technical program manager can illustrate how end-users derive value from the data: catching fraud patterns and summarizing them for business insights.

Optional Front-End Application (Next.js)

In addition to traditional BI dashboards, we include a custom web frontend to showcase the data in a tailored product-like interface. Using **Next.js** (a React framework) deployed on Vercel (or similar), we can build a simple web application for fraud analysts or even customers:

- The frontend could display a **real-time alerts dashboard** with push updates (for example, using WebSockets or polling an API that reads from the `fraud_alerts` Kafka topic or Trino). This might show incoming fraud alerts in a ticker or table as they happen, with sound/visual alerts for critical ones.
- It could also provide interactive investigation tools. For instance, clicking on an alert could fetch additional details via Trino (joining transactions and perhaps user data) and show a detailed view with all related transactions for that user.
- Next.js can easily call backend APIs or directly query data. One approach is to create an API endpoint (Node.js serverless function on Vercel) that runs a Trino query (Trino has a REST interface) to retrieve data on demand. This API can return JSON, which the React front-end visualizes (e.g. using D3 or Recharts for custom graphics).
- The frontend might also integrate with authentication and allow internal users to annotate alerts (mark false positives, etc.), writing back to the system (those annotations could be a new Kafka event and stored for model retraining).

While this Next.js app is optional, including it demonstrates end-to-end integration beyond BI – showing how the data pipeline can feed into a user-facing product. It's analogous to building a custom interface on top of Tableau's data for specific needs. In an interview scenario, this portion highlights the ability to interface data engineering with front-end user experience.

Implementation Steps

To deliver this project, the following phased implementation is suggested:

1. **Environment Setup:** Provision necessary infrastructure:
2. Deploy a Kafka cluster (e.g. AWS MSK or local Kafka for dev). Create required topics (`transactions`, `fraud-alerts`, etc.) [21] [22].
3. Set up an EMR cluster (or alternatively Kubernetes clusters) for running Flink and Spark jobs. Install Apache Flink on EMR and ensure connectivity to Kafka.
4. Configure AWS S3 buckets for the data lake. Set up AWS Glue Data Catalog (or Hive Metastore) and define an Iceberg catalog pointing to S3 data locations.
5. Launch a Trino service (this can be on EMR, a separate EC2 cluster, or a containerized deployment) and configure it to use the same Glue/Hive metastore so it can read the Iceberg tables.
6. **Data Ingestion Pipeline:** Install and configure Fluent Bit on data source emitters. For simulation, one can build a **transaction generator** script that publishes events to Kafka (mimicking Plaid's transaction webhooks). Fluent Bit can be tested by tailing a log file of generated events, or we directly produce to Kafka using the generator. Verify events are landing in the Kafka `transactions` topic.

7. **Streaming Job Development:** Implement the Flink streaming application:

8. Use the Kafka source connector to consume from `transactions` topic [23].
9. Define the event schema (e.g. a `Transaction` Java/Scala case class or use JSON parsing). Include fields like userId, amount, timestamp, etc.
10. Implement detection logic: e.g. using Flink CEP for pattern rules (like detecting an event in NY followed by one in LA within 30 min), or simple keyed windows for count-based rules. (Leverage the rule set described earlier as initial scope.)
11. For each anomaly found, create an alert object. Produce it to a Kafka sink (to `fraud-alerts` topic) for immediate consumption. Simultaneously, use the Flink Iceberg sink to append the alert to the `fraud_alerts_table` in S3 (this can be done by configuring Flink's table API or DataStream API with Iceberg connector).
12. Also set up a sink for all transactions to go into the `transactions_table` on S3 (we might choose to have Flink write these in mini-batches or use a separate ingestion process like Kafka Connect – but Flink can do it for simplicity).

13. Test the Flink job locally or on EMR with a small stream of events, and verify that alerts are correctly identified (e.g., trigger some known fraud scenarios and see that alerts appear).

14. **Data Lake and Iceberg Validation:** Once Flink writes data to S3, use the Glue Catalog or Iceberg tools to inspect the tables. Ensure that partitions are created (e.g., a partition for current date if using date partitioning). Run sample queries using Spark or the Iceberg API to count records and validate schema. Iceberg's snapshot can be used to time-travel or ensure exactly-once (no duplicates) from Flink's writes.

15. **Batch Processing Jobs:** Develop Spark jobs on EMR for advanced analytics:

16. A **batch anomaly detection job**: e.g. compute aggregates per user over a longer window (last 7 days) and flag outliers. This job reads from `transactions_table` (possibly using SparkSQL with Iceberg support) and writes anomalies to `fraud_alerts_table` with a tag indicating batch detection.
17. A **model training job**: if labeled data is available, train a model. If not, train an unsupervised model. This could be done in PySpark or Spark ML. Save the model to S3.

18. (Optional) A **batch scoring job**: load the saved model and run it on recent data to output scores. (If we plan to integrate the model into Flink, this may be optional.) Schedule these jobs with AWS Step Functions or cron (e.g. daily at midnight). Monitor their runtime and output.

19. **Trino Integration:** Connect Trino to the data lake. In Trino's catalog configuration, add an Iceberg connector pointing to the Glue catalog and S3 bucket. Then verify by running queries in Trino:

20. e.g. `SELECT COUNT(*) FROM iceberg.default.transactions_table;`
21. e.g. `SELECT userId, reason, fraudScore, alertTime FROM iceberg.default.fraud_alerts_table WHERE alertDate = current_date;`. Ensure these queries return correct results. Fine-tune Trino config (memory, splits, etc.) for performance as needed.

22. **Dashboard Creation:** Deploy Apache Superset (possibly on a small EC2 or as a Docker container) and connect it to Trino via SQLAlchemy URI. In Superset:

23. Register the tables (or define SQL queries) for transactions and alerts.
24. Create charts: e.g. use a table view for latest alerts, a time-series line chart for alerts count by hour, a world map visualization for transaction locations (Superset has a deck.gl geospatial chart).
25. Combine charts into a dashboard titled “Fraud Detection Overview”. Add filters (date range, userId) for interactivity.

26. Alternatively, if using Tableau, set up an ODBC connection to Trino or export a subset of data to Hyper for a demo (though real-time is better achieved via live connection). Design equivalent dashboards in Tableau.

27. **Optional Next.js App:** Initialize a Next.js project. Implement API routes that query Trino or (for simplicity) read from the Kafka `fraud-alerts` topic:

28. For example, a `/api/alerts` route that returns the last N alerts from the Iceberg table (via a Trino query).
29. A React component that uses `SWR` or WebSocket to hit this API and auto-refresh a list of alerts on screen.
30. Implement pages for various views (alerts list, alert details, summary stats). Use a component library or custom D3 charts for visualization.

31. Deploy the app to Vercel and test it with live data. This app can be shown alongside Superset to illustrate custom use cases.

32. **Testing & Demo:** Simulate a variety of fraud scenarios to test the entire pipeline:

33. Feed known-pattern fraudulent transactions (e.g. rapid-fire transactions, geo-distant transactions) and verify Flink catches them (they appear in alerts).
34. Check that those alerts are stored in S3 and visible via Trino/Superset.
35. Run the batch jobs on a snapshot of data that has some anomalies to ensure they flag expected items.
36. Test failure scenarios: e.g. stop a Flink node to ensure checkpoint recovery works (no data loss or double processing).

37. Evaluate latency from ingestion to alert (should be on the order of seconds). Also ensure the dashboard query performance is acceptable (Trino should query recent data quickly especially if partition pruned).

38. **Documentation & Diagram:** Provide architecture diagrams (like the one above) and data model docs. Include sample event definitions and a glossary of fields (so interviewers see you considered data semantics). Document how each component connects (e.g. config files for Fluent Bit, Flink job code structure, etc.). Emphasize how this design meets the requirements of real-time fraud detection (minimal latency, scalability) and is **technically sound** – using proven open-source components in a decoupled, scalable manner, much like a production Tableau/BI data stack but tailored to fraud analytics.

Throughout the implementation, a technical program manager would coordinate between data engineering (for pipeline and storage), data science (for the ML model and detection logic), DevOps (for deploying Kafka, EMR, Trino), and business intelligence teams (for dashboard requirements). The result is a compelling end-to-end system demonstrating skills across data ingestion, stream processing, batch ML, and visualization.

Conclusion

This prototype project showcases a cohesive fraud detection platform inspired by a modern analytics stack. By **replicating elements of Tableau's architecture** – from data connectors and fast queries to rich dashboards – it proves out a full data pipeline on financial events. The use of Fluent Bit, Kafka, Flink, Iceberg, EMR, Trino, and Superset covers the spectrum of data infrastructure. Such a project would allow a candidate to demonstrate technical program management by aligning multiple technologies towards a real-world use case (fraud detection), ensuring they all integrate seamlessly. The outcome is a system where streaming analytics catch fraud in real-time (preventing loss before it happens) [24], and a robust data lake/warehouse foundation supports deeper investigation and continuous model improvements. This blend of real-time and batch processing, tied together with a visualization layer, illustrates the end-to-end ownership a TPm can have in delivering data-driven solutions.

Sources: The design is informed by real-world practices in fraud detection and data lakehouse architecture, including Apache Flink's fraud detection patterns [25], streaming analytics blogs [6] [26], and Iceberg/Trino lakehouse principles [3] [27]. All components chosen are scalable and widely adopted, ensuring the prototype can evolve into a production-grade solution at Plaid.

---

[1] Kafka | Fluent Bit: Official Manual
https://docs.fluentbit.io/manual/pipeline/outputs/kafka

[2] [25] Advanced Flink Application Patterns Vol.1: Case Study of a Fraud Detection System | Apache Flink
https://flink.apache.org/2020/01/15/advanced-flink-application-patterns-vol.1-case-study-of-a-fraud-detection-system/

[3] [15] [17] Apache Iceberg: Architecture, benefits, partitioning | Trino
https://www.starburst.io/blog/introduction-to-apache-iceberg-in-trino/

[4] [5] [18] [27] Exploring an Alternative to Redshift: Trino, Apache Iceberg, and AWS S3 | by Bruno Ferreira da Silva | Medium
https://medium.com/@brunoferreiradasilva/exploring-an-alternative-to-redshift-trino-apache-iceberg-and-aws-s3-328ec09b9d20

[6] [8] [9] [10] [11] [12] [13] [14] [21] [22] [23] Real Time Fraud Detection Using Apache Flink — Part 1 | by Yugen.ai | Yugen.ai Technology Blog | Medium
https://medium.com/yugen-ai-technology-blog/real-time-fraud-detection-using-apache-flink-part-1-f4b1c9d6e952

[7] Apache Kafka, Flink, and Druid: Open Source Essentials for Real-Time Data Products - Developer Center
https://imply.io/developer/articles/apache-kafka-flink-and-druid-open-source-essentials-for-real-time-data-products/

[16] Build a data lake with Apache Flink on Amazon EMR | AWS Big Data Blog
https://aws.amazon.com/blogs/big-data/build-a-unified-data-lake-with-apache-flink-on-amazon-emr/

[19] Apache Superset: The Open-Source Solution for Data Visualization I
https://blog.elest.io/apache-superset-the-open-source-solution-for-data-visualization-i/

20 Welcome | Superset - The Apache Software Foundation
https://superset.apache.org/

24 26 Fraud Detection with Apache Kafka, KSQL and Apache Flink - Kai Waehner
https://www.kai-waehner.de/blog/2022/10/25/fraud-detection-with-apache-kafka-ksql-and-apache-flink/