# Streaming Data Pipeline with Kafka: Detailed README

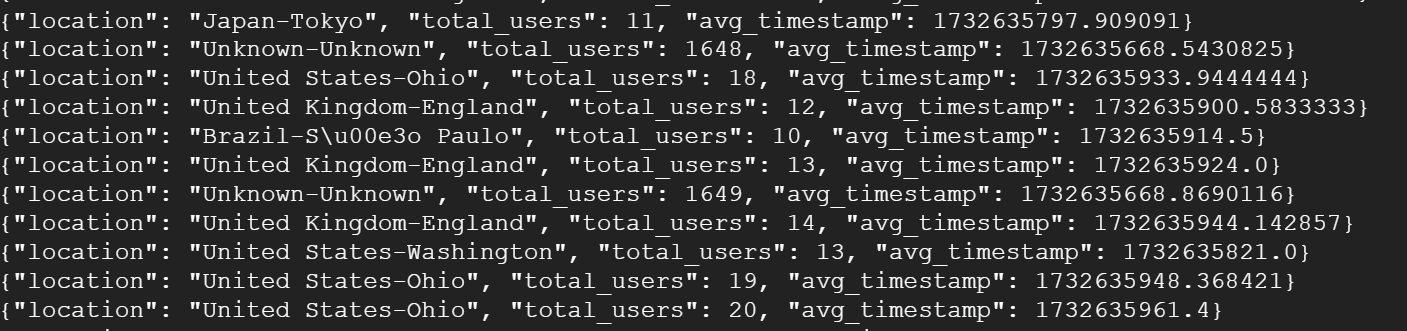
**Yanjie Xing**

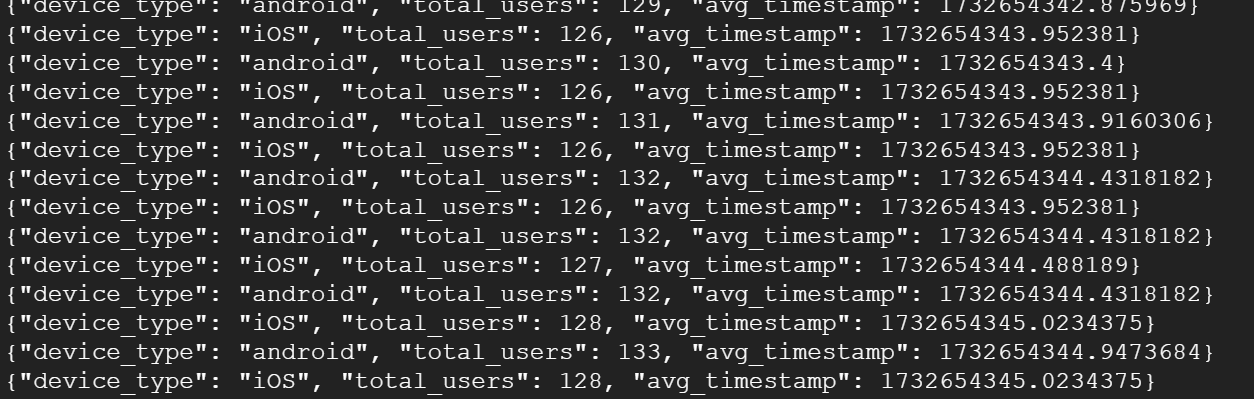
## Introduction

This project implements a real-time streaming data pipeline using Kafka on GCP.   
The pipeline processes and transforms incoming user-login data, performs aggregations, applies filtering based on specific conditions,   
and stores the processed results in separate Kafka topics for further analysis.

## Metrics Calculated

1. Geographical User Aggregation:  
    Total number of users grouped by country and state, it can give a brief overview of user activity on country-state basis.  
    Average timestamp of user logins for each geographical location which can help us analyze most active time period of users in different regions.  
    Filtered out locations with total user count below a threshold of 10.

  
  
2. Device Type Aggregation:  
 Total number of users grouped by device type (e.g., Android, iOS).  
 Average timestamp of user logins for each device type. These two metrics can help us understand which type of device our customers most frequently use.  
 Filtered out device types with total user count below 100.



## Transforming, Aggregating, and Filtering Operations

1. Transforming:   
 Parsed the `ip\_address` field from the user-login data to derive `country` and `state` using an external IP geolocation API.  
 Added a `processed\_at` timestamp field to the data for tracking when it was processed.  
  
2. Aggregating:   
 Calculated total user count and average timestamp for each country-state combination.  
 Performed similar aggregations for device types.  
  
3. Filtering:   
 Excluded records from the output if the user count for a location or device type was below a defined threshold.

## Handling Streaming Data and Ensuring Efficiency

1. Error Handling:  
 Managed cases where the IP geolocation API returned errors or incomplete data by assigning `Unknown-Unknown` to such records.  
 Skipped records missing critical fields such as `ip\_address` or `device\_type`.  
  
2. Continuous Streaming:  
 The pipeline continuously consumes data from the Kafka input topic (`user-login`), processes it in real time, and produces results to output topic (`processed-login`).  
  
3. Efficiency:  
 Used Python's `requests` module with a retry mechanism to handle temporary API failures.  
 Cached resolved IP addresses using `cachetools` to minimize redundant API calls.  
  
4. Scalability:  
 Kafka's distributed architecture allows for horizontal scaling of both producers and consumers to handle higher data throughput.  
 The consumer application can be scaled by increasing the number of consumer instances in the same consumer group.

## Design Choices and Data Flow

1. Design Choices:  
 Used Kafka for its robust, distributed message-brokering capabilities.  
 Leveraged Python's Kafka library for ease of integration and rapid development.  
 Used a modular approach, separating consumer logic for geographical and device-type aggregations into distinct scripts.  
  
2. Data Flow:  
 Input: The `user-login` topic receives raw user login data.  
 Processing: The consumer scripts process the data, performing transformations, aggregations, and filtering.  
 Output: Processed data is published to precreated Kafka topics (`processed-login`).

## Project Setup

### Prerequisites

- Docker and Docker Compose installed on your system.  
- Python 3.8 or later installed.  
- Clone the project repository to your local machine.

## Running the Project

### Start Docker Containers

Ensure you are in the project root directory and run the following command to start the required Docker containers:

docker-compose up -d

### Enter Kafka Container and Create Topics

Enter the Kafka container and navigate to the /usr/bin directory to create necessary Kafka topics:

docker exec -it kafka-streaming-project-kafka-1 bash  
cd /usr/bin  
  
# Create input topic  
kafka-topics --create --bootstrap-server localhost:9092 --replication-factor 1 --partitions 1 --topic user-login  
  
# Create output topics  
kafka-topics --create --bootstrap-server localhost:9092 --replication-factor 1 --partitions 1 --topic processed-login  
  
# Verify topics  
kafka-topics --list --bootstrap-server localhost:9092  
  
# Exit Kafka container  
exit

### Install Python Dependencies

Install Python dependencies on your host machine:

pip install -r app/requirements.txt

### Run Consumer Scripts

Run the consumer scripts on your host machine:

# Run the geo-location consumer  
python3 app/consumer\_geo.py   
  
# Run the device-type aggregation consumer  
python3 app/consumer\_device.py

### Verify Output Results

Enter the Kafka container to consume messages from the output topics and verify the processing results:

docker exec -it kafka-streaming-project-kafka-1 bash  
  
# View results  
kafka-console-consumer --bootstrap-server localhost:9092 --topic processed-login --from-beginning  
  
# Exit Kafka container  
exit

## Questions and Answers

1. **How would you deploy this application in production?**  
   To deploy this application in production, start by publishing the Docker images to a container registry such as Docker Hub, AWS ECR, or GCP Artifact Registry. Use a container orchestration tool like Kubernetes to manage deployment, scaling, and fault tolerance.

Evaluate the expected workload, user activity, and revenue impact of the real-time data pipeline to determine the appropriate Kafka cluster size. For larger-scale production, a Kafka cluster with multiple brokers, such as a 30-broker configuration, can handle high throughput and provide fault tolerance.

Select a stable, high-speed, and cost-effective cloud provider for hosting Kafka services. Managed options like AWS MSK, Confluent Cloud, or GCP Pub/Sub can simplify operations. Set up CI/CD pipelines with tools such as Jenkins, GitHub Actions, or GitLab CI/CD to ensure automated testing and seamless deployment. Implement robust monitoring and alerting using tools like Prometheus and Grafana for real-time metrics and visualization. Configure alerts for issues like high latency, message backlog, or broker failures to facilitate quick response.

These methods will ensure scalability, fault tolerance and reliability in production environment.

2. **What other components would you add to make this production ready?**

**Computation Engine**: Replace the Python-based consumer logic with a more robust and scalable real-time computation engine like Apache Flink or Spark Structured Streaming. These engines provide built-in capabilities for fault tolerance, stateful processing, and exactly-once semantics, making them better suited for handling large-scale, real-time data streams.

**Data Storage**: Persist processed data in durable storage systems such as Parquet files (for efficient analytics), MySQL (for structured queries), or S3 (for scalable and cost-effective cloud storage). These solutions enable long-term data availability and facilitate downstream processing and analysis.

**Data Quality**: Integrate schema validation (e.g., using Confluent Schema Registry) to ensure data consistency across the pipeline. Use dead-letter queues to manage malformed or missing data, preventing processing disruptions while maintaining high-quality data output.

**Data Pipeline Management**: Utilize orchestration tools like Apache Airflow or cloud-based solutions (e.g., AWS Step Functions) to automate pipeline triggers. This ensures processing begins automatically based on predefined events or schedules, improving operational efficiency.

**Monitoring and Alerts**: Implement monitoring tools such as Prometheus and Grafana to track system health, resource utilization, and processing metrics. Set up alerts for anomalies like consumer lag or throughput drops to quickly identify and resolve issues.

**Access Control and Security**: Secure the Kafka communication with SASL/SSL authentication and implement granular authorization to control access to Kafka topics, ensuring data privacy and security.

3. **How can this application scale with a growing dataset?**

Increase the number of Kafka partitions to distribute the load across more brokers and enable higher concurrency. This allows multiple consumer instances to read from the topic in parallel, improving throughput. Optimize partitioning strategies by using relevant keys (e.g., user ID or region) to ensure balanced data distribution and minimize hot partitions.

Add more consumer instances to the same consumer group. Kafka ensures each partition is assigned to only one consumer in a group, enabling horizontal scaling of message processing.

Dynamically adjust the number of consumers based on workload using auto-scaling mechanisms in container orchestration platforms like Kubernetes.

Use managed Kafka services (e.g., Confluent Cloud or AWS MSK) that handle partition scaling and infrastructure optimization, reducing operational overhead.