Streaming Data Processing
Lab 2 Report: Kafka Web Logs

OUEDRAOGO Kiswendsida Abraham

October 16, 2025

1 Introduction

This report presents the work done in Lab 2 about streaming data processing using Apache Kafka. The goal was to create a system that mimics how a real website logs user activities.

In this lab, we did two main tasks, generate fake log files that look real and send these logs through Kafka and read them back.

The tools we used were:

- Python 3 for programming (Notebook)
- Confluent Cloud for Kafka
- Python libraries: `confluent-kafka`, `uuid`, `random`, `datetime`

2 Exercise 1: Generate Fake Log Files

2.1 What We Need to Create

A website log file has 4 main parts:

- **Timestamp**: When something happened (example: 2025-10-16 21:16:47.779100)
- **User ID**: A unique ID for each visitor (example: a0aa0dd0)
- **Action**: What the user did (example: click, view, close)
- **Page**: Which page they visited (example: home, about, contact)

2.2 The Challenge: Making It Look Real

The hardest part was making the logs resemble genuine user behavior. In reality, users don't all arrive at the same time, and many can be active on the website simultaneously. Each user performs several actions before leaving, and while some exit the site normally, others simply vanish without completing their session.

2.3 Our Solution

We created a function called `generate_log_line()` that makes realistic logs. Here is how it works:

Step 1: Users arrive slowly over time
Instead of creating all users at once, we make them arrive one by one. We use a probability system:

- At the beginning: 30% chance a new user arrives
- Later: 10% chance (fewer new users come)

```
arrival_probability = 0.3 if len(users_arrived) < num_users * 0.7
    else 0.1

if random.random() < arrival_probability:
    new_user = random.choice(available)
    users_arrived.append(new_user)
```

Listing 1: Code for user arrival

Step 2: Track active users
We keep a list of users who are currently on the website. For each user, we remember:

- How many actions they did (we limit it randomly beetween (3,8))
- When they started
- If they will close properly (85%) or just disappear (15%)

```
active_users[new_user] = {
    'actions_done': 1,
    'start_time': t,
    'will_close': random.random() > 0.15  # 85% will close
}
```

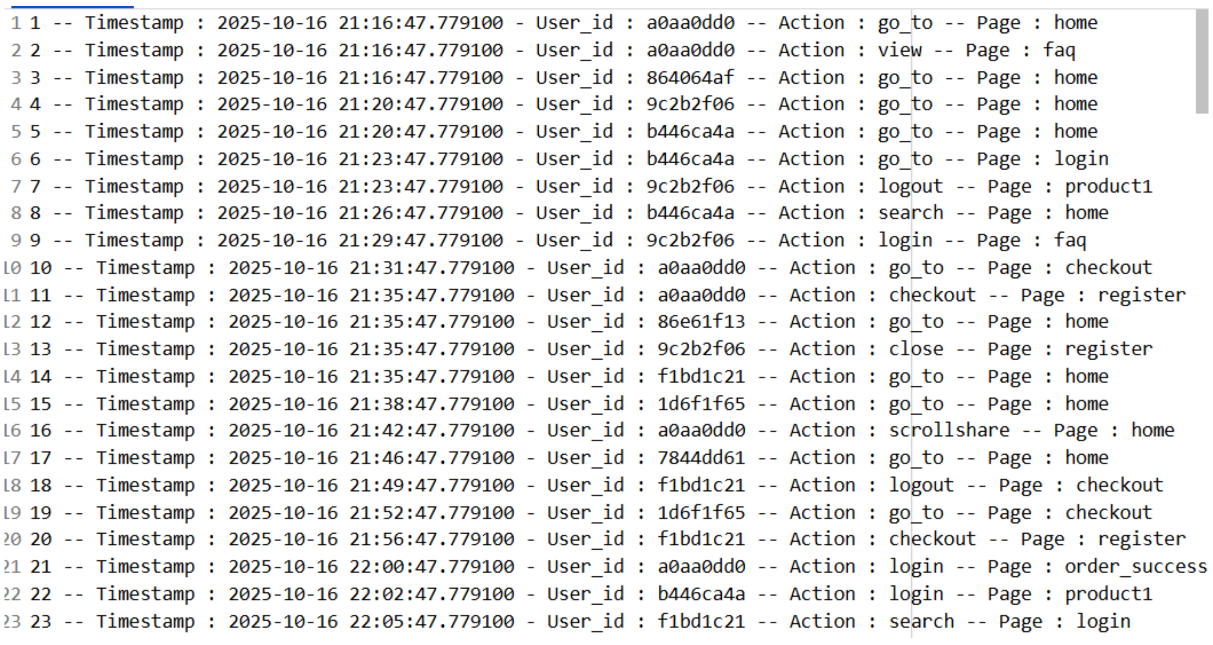
Listing 2: Tracking user sessions

Step 3: Create actions for active users
For users who are already on the website, we randomly pick one and make them do an action. Each user does between 3 and 8 actions before leaving.

Step 4: Sessions overlap naturally
Because we process users one by one over time, their sessions naturally overlap.

2.4 Results

Our function generated 193 log lines for 20 different users. The logs were saved to a file called `logs.txt`.



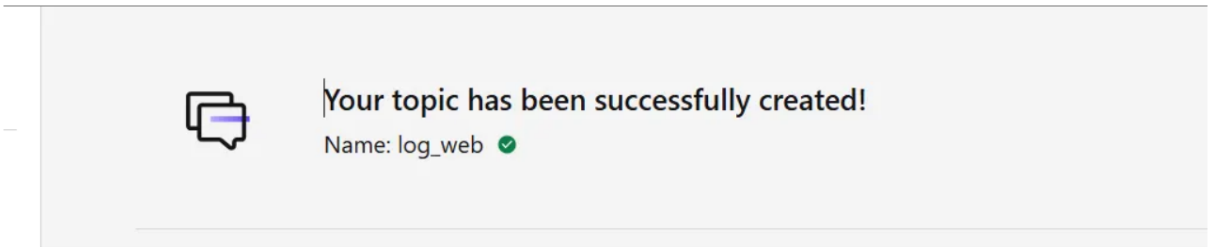
```
1 -- Timestamp : 2025-10-16 21:16:47.779100 - User_id : a0aa0dd0 -- Action : go_to -- Page : home
2 -- Timestamp : 2025-10-16 21:16:47.779100 - User_id : a0aa0dd0 -- Action : view -- Page : faq
3 -- Timestamp : 2025-10-16 21:16:47.779100 - User_id : 864064af -- Action : go_to -- Page : home
4 -- Timestamp : 2025-10-16 21:20:47.779100 - User_id : 9c2b2f06 -- Action : go_to -- Page : home
5 -- Timestamp : 2025-10-16 21:20:47.779100 - User_id : b446ca4a -- Action : go_to -- Page : home
6 -- Timestamp : 2025-10-16 21:23:47.779100 - User_id : b446ca4a -- Action : go_to -- Page : login
7 -- Timestamp : 2025-10-16 21:23:47.779100 - User_id : 9c2b2f06 -- Action : logout -- Page : product1
8 -- Timestamp : 2025-10-16 21:26:47.779100 - User_id : b446ca4a -- Action : search -- Page : home
9 -- Timestamp : 2025-10-16 21:29:47.779100 - User_id : 9c2b2f06 -- Action : login -- Page : faq
10 -- Timestamp : 2025-10-16 21:31:47.779100 - User_id : a0aa0dd0 -- Action : go_to -- Page : checkout
11 -- Timestamp : 2025-10-16 21:35:47.779100 - User_id : a0aa0dd0 -- Action : checkout -- Page : register
12 -- Timestamp : 2025-10-16 21:35:47.779100 - User_id : 86e61f13 -- Action : go_to -- Page : home
13 -- Timestamp : 2025-10-16 21:35:47.779100 - User_id : 9c2b2f06 -- Action : close -- Page : register
14 -- Timestamp : 2025-10-16 21:35:47.779100 - User_id : f1bd1c21 -- Action : go_to -- Page : home
15 -- Timestamp : 2025-10-16 21:38:47.779100 - User_id : 1d6f1f65 -- Action : go_to -- Page : home
16 -- Timestamp : 2025-10-16 21:42:47.779100 - User_id : a0aa0dd0 -- Action : scrollshare -- Page : home
17 -- Timestamp : 2025-10-16 21:46:47.779100 - User_id : 7844dd61 -- Action : go_to -- Page : home
18 -- Timestamp : 2025-10-16 21:49:47.779100 - User_id : f1bd1c21 -- Action : logout -- Page : checkout
19 -- Timestamp : 2025-10-16 21:52:47.779100 - User_id : 1d6f1f65 -- Action : go_to -- Page : checkout
20 -- Timestamp : 2025-10-16 21:56:47.779100 - User_id : f1bd1c21 -- Action : checkout -- Page : register
21 -- Timestamp : 2025-10-16 22:00:47.779100 - User_id : a0aa0dd0 -- Action : login -- Page : order_success
22 -- Timestamp : 2025-10-16 22:02:47.779100 - User_id : b446ca4a -- Action : login -- Page : product1
23 -- Timestamp : 2025-10-16 22:05:47.779100 - User_id : f1bd1c21 -- Action : search -- Page : login
```

Figure 1: logs generated

3 Exercise 2: Kafka Streaming

3.1 Step 1: Create Kafka Topic

We created a topic called `log_web` on Confluent Cloud.



Your topic has been successfully created!

Name: log_web

Figure 2: Topic creation on Confluent Cloud

3.2 Step 2: Kafka Producer (Sending Messages)

The producer reads our log file and sends each line to Kafka.

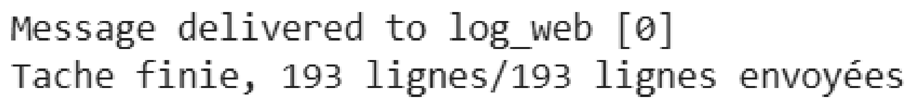
How we proceed:

1. Open the `logs.txt` file
2. Read each line
3. Parse the line to extract timestamp, user_id, action, and page
4. Convert to JSON format
5. Send to Kafka topic `log_web`

```python
def parse_line(line):
    parts = line.strip().split(' -- ')
    return {
        'timestamp': parts[1].split(' : ', 1)[1].split(' - ')[0].strip(),
        'user_id': parts[1].split('User_id : ')[1].split(' --')[0].strip(),
        'action': parts[2].split(' : ')[1].strip(),
        'page': parts[3].split(' : ')[1].strip()
    }
```

Listing 3: Parsing and sending logs

We sent 193 messages to Kafka successfully.



```
Message delivered to log_web [0]
Tache finie, 193 lignes/193 lignes envoyées
```

Figure 3: 193 messages sent to Kafka topic

3.3 Step 3: Kafka Consumer (Reading Messages)

The consumer reads all messages from Kafka and calculates statistics.

How we proceed:

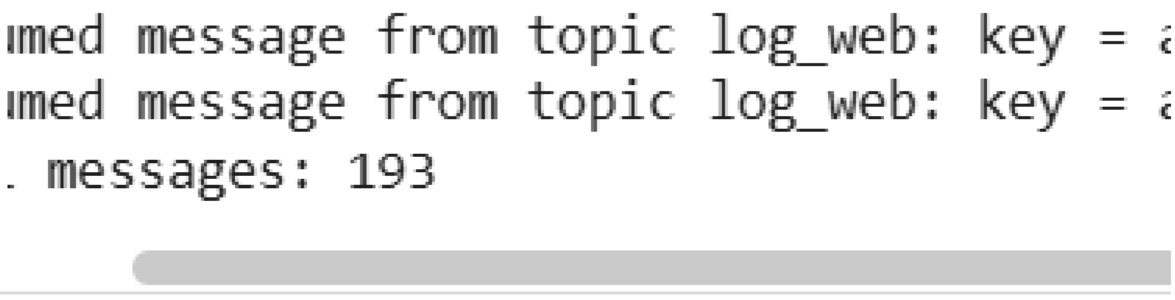
1. Connect to Kafka
2. Subscribe to topic `log_web`
3. Read all messages one by one
4. Store messages in a list
5. Calculate statistics

Important fix: We had a bug where we counted messages multiple times. The problem was in the code structure:

```python
# WRONG (bug):
if msg is not None and msg.error() is None:
    value = msg.value().decode("utf-8")

all_msg.append(value)  # This runs even when msg is None!

# CORRECT (fixed):
if msg is None:
    continue  # Skip if no message

if msg.error() is None:
    value = msg.value().decode("utf-8")
    all_msg.append(json.loads(value))  # Only append here!
```

Listing 4: Bug fix - correct indentation

Result: We read exactly 193 messages from Kafka.



```
umed message from topic log_web: key = a
umed message from topic log_web: key = a
. messages: 193
```

Figure 4: 193 messages consumed from Kafka topic

3.4 Step 4: Calculate Statistics

We calculated three statistics:

1. **Number of messages:**

```
num_logs = len(all_msg)
# Result: 193 messages
```

2. **Number of unique visitors:**

```
unique_users = len(set(m['user_id'] for m in all_msg))
# Result: 20 unique users
```

3. **Average visit duration:**
This was the most complex calculation. We had to:

- Find when each user arrived (action = 'go_to' and page = 'home')
- Find when each user left (action = 'close')
- Calculate the time difference
- Take the average

```
sessions = defaultdict(dict)

for m in all_msg:
    ts = datetime.strptime(m['timestamp'], '%Y-%m-%d %H:%M:%S.%f'
    )

    if m['action'] == 'go_to' and m['page'] == 'home':
        sessions[m['user_id']]['start'] = ts
    elif m['action'] == 'close':
        sessions[m['user_id']]['end'] = ts

# Calculate durations only for complete sessions
durations = [(s['end'] - s['start']).total_seconds()
             for s in sessions.values()
             if 'start' in s and 'end' in s]

avg_duration = sum(durations) / len(durations)
# Result: 7263.53 seconds (about 2 hours)
```
Listing 5: Calculate average duration

4 Final Results

Metric	Value
Messages sent to Kafka	193
Messages read from Kafka	193
Unique visitors	20
Average visit duration	7263.53 seconds

Table 1: Final statistics

Important observation: The number of messages sent (193) equals the number of messages read (193). This proves our Kafka system works correctly!

5 Challenges and Solutions

5.1 Challenge 1: Making Realistic Logs

Problem: If we create logs randomly, they don't look like real user behavior.

Solution: To address this, we designed a system in which users arrive progressively over time, each with a complete session that includes a start, a series of actions, and an end. Some sessions remain incomplete when users suddenly disappear, while multiple users can be active simultaneously, creating a more realistic simulation of website activity.

5.2 Challenge 2: Counting Messages Twice

Problem: We were counting 308 messages instead of 193.

Reason: The `append()` was outside the `if` statement, so it ran even when there was no message.

Solution: We fixed the indentation to only append when we have a valid message.

5.3 Challenge 3: Parsing JSON

Problem: We tried to parse JSON twice and got an error.

Reason: The consumer already converts JSON to dictionary, so we don't need to do it again.

Solution: Remove the second `json.loads()`.

6 Conclusion

In this lab, we successfully:

- Created realistic fake log files that mimic real user behavior
- Sent 193 log messages through Kafka
- Read all 193 messages back successfully
- Calculated statistics about user visits

The most interesting part was creating realistic logs. We learned that making fake data look real is harder than it seems! We need to think about:

- How users arrive over time
- How their actions overlap
- How some users close properly and others don't

Kafka worked very well for sending and receiving messages. The number of messages sent exactly matched the number received, which shows the system is reliable.

This lab helped us understand how real-time data streaming works in practice.