DATE
02/24/2023

REAL-TIME GPS TRACKING FOR TRANSIT SERVICE

PRESENTED BY
Ameya Ranade, Mayank Bumb,
Riya Adsul, Saransh Bachawat

TABLE OF CONTENTS

PRESENTED BY

Ameya Ranade, Mayank Bumb,
Riya Adsul, Saransh Bachawat

PROJECT OBJECTIVES

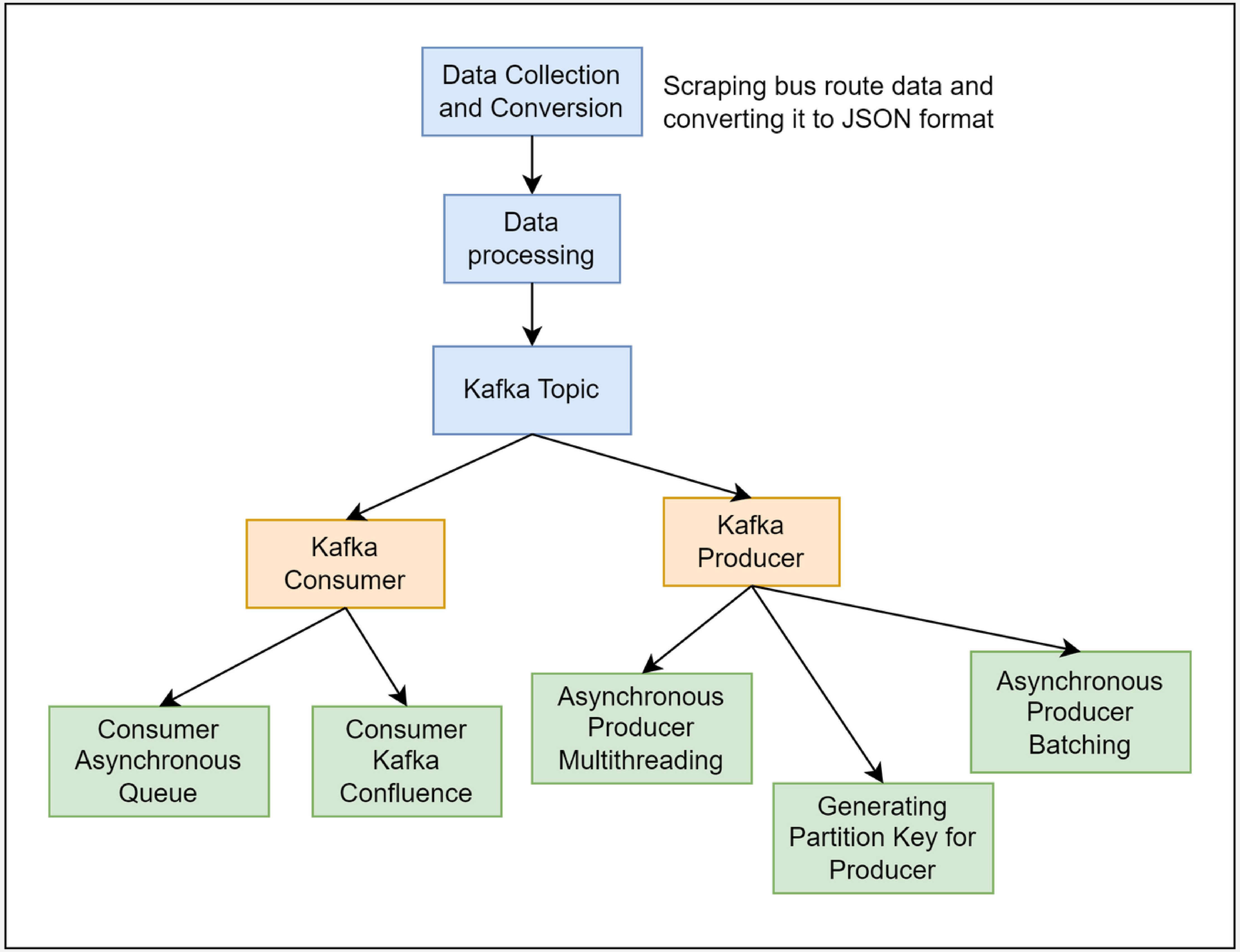
01 The integration of a real-time GPS tracking system for a bus transit service, employing cutting-edge technologies such as **Python, Apache Kafka, ZooKeeper, Apache Spark and Flask**, represents a significant advancement in the transportation sector. This project aims to provide a comprehensive view of real-time bus activities, including the precise location marker movement of various buses on a map GUI (of Amherst).

02 The system's key components involve the collection of live data through **Apache Kafka** and real-time processing via **Apache Spark**. Additionally, we intend to evaluate the system's performance by measuring **Data ingestion rate, Kafka message throughput, and Processing time**, varying various system parameters for the incoming data. These performance metrics will aid in assessing the real-time GPS tracking system's efficiency.

PRESENTED BY

Ameya Ranade, Mayank Bumb,
Riya Adsul, Saransh Bachawat

PROJECT DESIGN

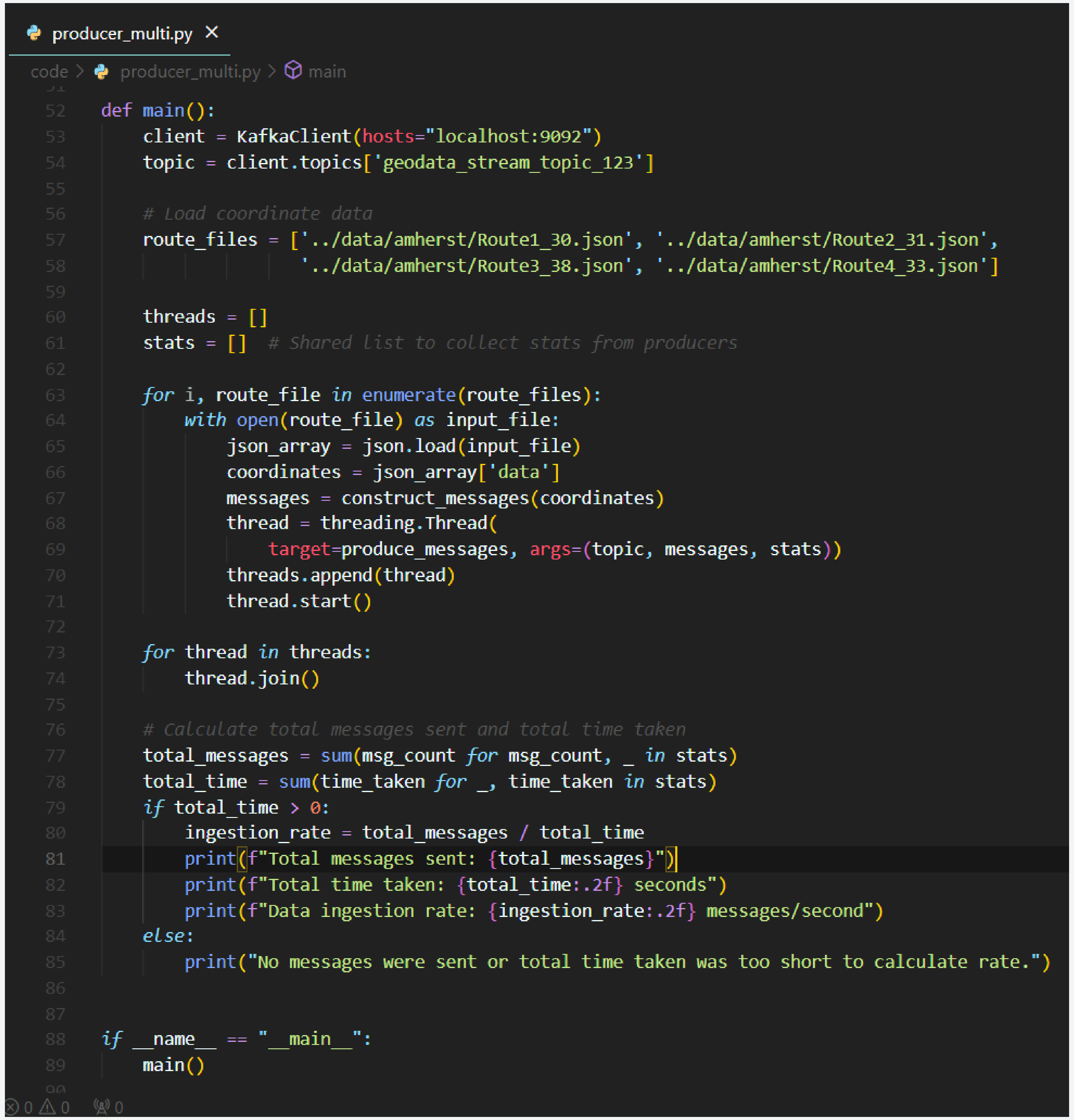

Data Collection and Conversion
Scraping bus route data and converting it to JSON format
Data processing
Kafka Topic
Kafka Consumer
Kafka Producer
Consumer Asynchronous Queue
Consumer Kafka Confluence
Asynchronous Producer Multithreading
Generating Partition Key for Producer
Asynchronous Producer Batching

PRESENTED BY

Ameya Ranade, Mayank Bumb, Riya Adsul, Saransh Bachawat

OUR TEAM - CONTRIBUTIONS

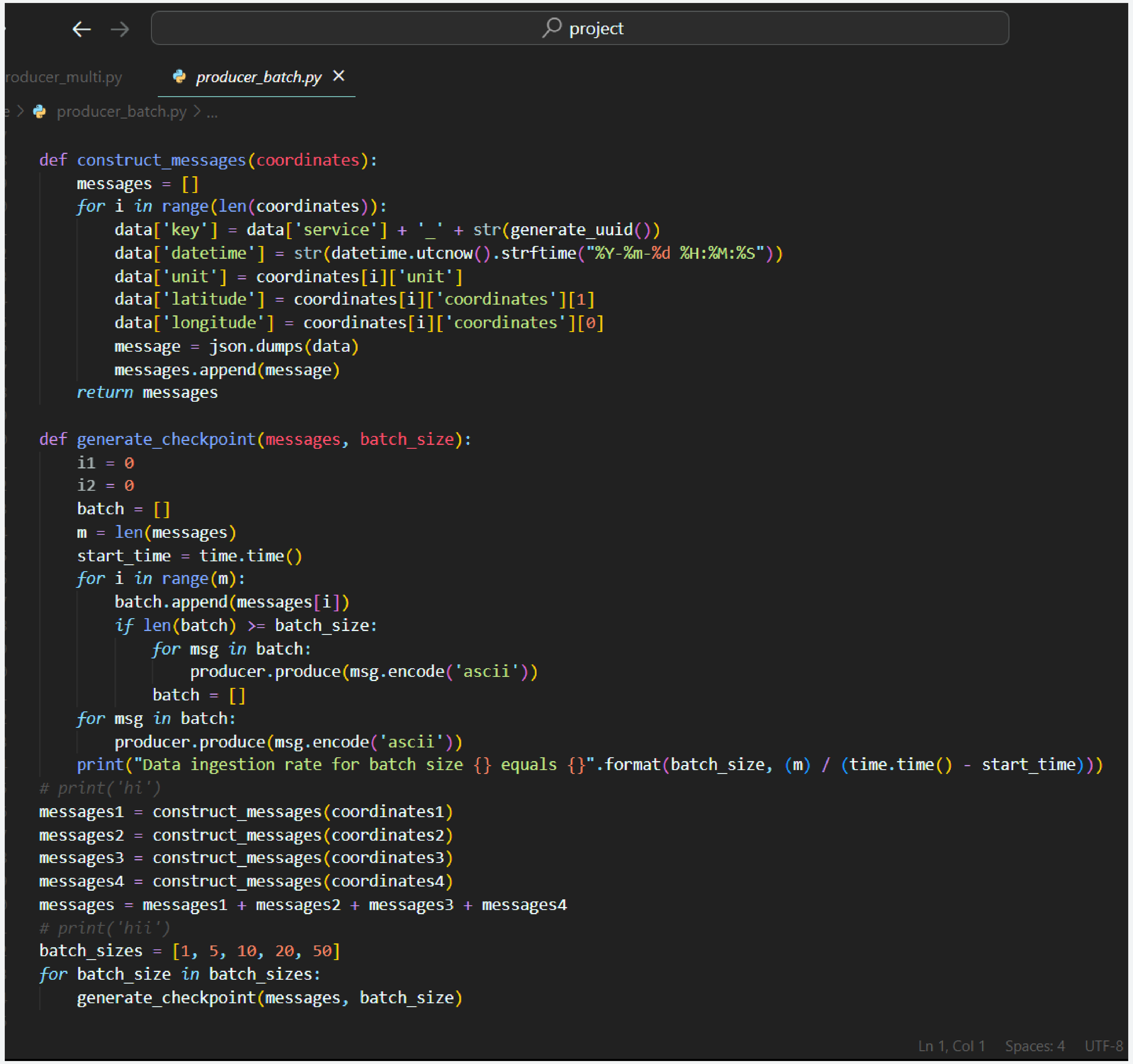
Ameya Ranade	producer_rate, producer_view and consumer_view
Mayank Bumb	Consumer_async, consumer_confluence, consumer_rate and producer_partition_key
Riya Adsul	web scarpping , converting to required json and producer asyn multithreading
Saransh Bhachawat	producer_batch, producer_partition_key and producer_view



```python
def main():
    client = KafkaClient(hosts="localhost:9092")
    topic = client.topics['geodata_stream_topic_123']

    # Load coordinate data
    route_files = ['../data/amherst/Route1_30.json', '../data/amherst/Route2_31.json',
                   '../data/amherst/Route3_38.json', '../data/amherst/Route4_33.json']

    threads = []
    stats = []  # Shared list to collect stats from producers

    for i, route_file in enumerate(route_files):
        with open(route_file) as input_file:
            json_array = json.load(input_file)
            coordinates = json_array['data']
            messages = construct_messages(coordinates)
            thread = threading.Thread(
                target=produce_messages, args=(topic, messages, stats))
            threads.append(thread)
            thread.start()

    for thread in threads:
        thread.join()

    # Calculate total messages sent and total time taken
    total_messages = sum(msg_count for msg_count, _ in stats)
    total_time = sum(time_taken for _, time_taken in stats)
    if total_time > 0:
        ingestion_rate = total_messages / total_time
        print(f"Total messages sent: {total_messages}")
        print(f"Total time taken: {total_time:.2f} seconds")
        print(f"Data ingestion rate: {ingestion_rate:.2f} messages/second")
    else:
        print("No messages were sent or total time taken was too short to calculate rate.")


if __name__ == "__main__":
    main()
```

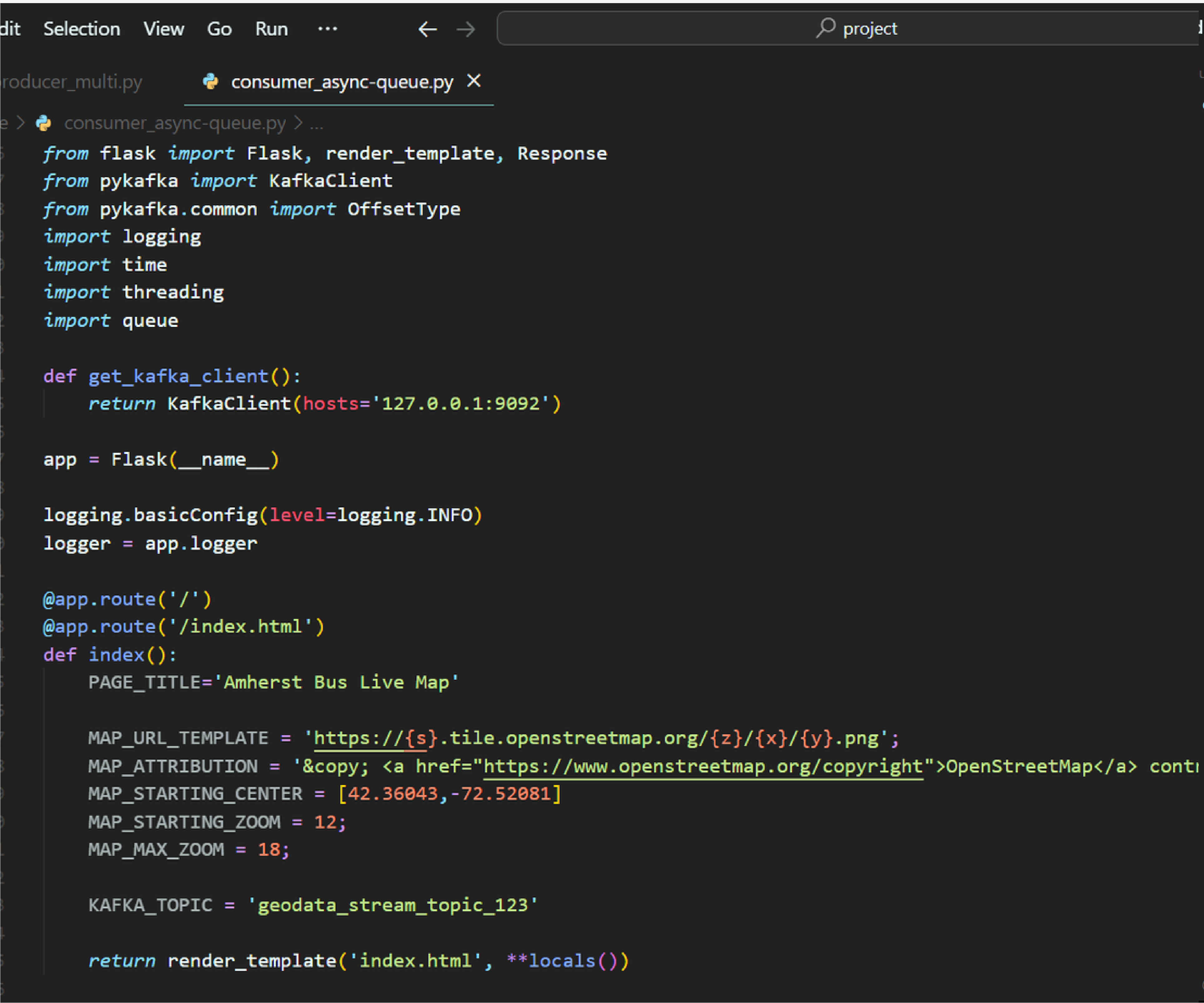
Multi- threading in kafka producer

Creating Batches in kafka producer

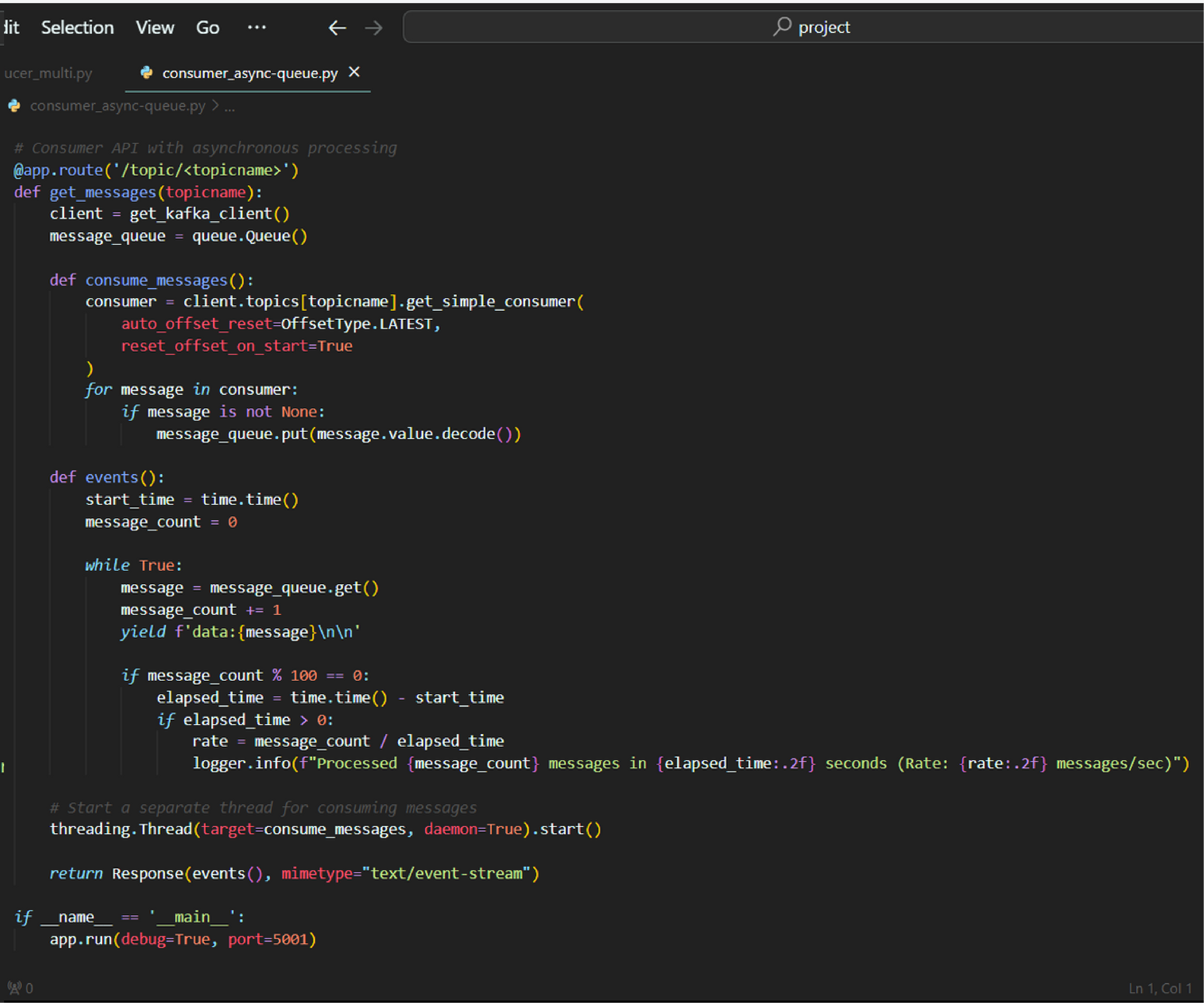


```python
def construct_messages(coordinates):
    messages = []
    for i in range(len(coordinates)):
        data['key'] = data['service'] + '_' + str(generate_uuid())
        data['datetime'] = str(datetime.utcnow().strftime("%Y-%m-%d %H:%M:%S"))
        data['unit'] = coordinates[i]['unit']
        data['latitude'] = coordinates[i]['coordinates'][1]
        data['longitude'] = coordinates[i]['coordinates'][0]
        message = json.dumps(data)
        messages.append(message)
    return messages

def generate_checkpoint(messages, batch_size):
    i1 = 0
    i2 = 0
    batch = []
    m = len(messages)
    start_time = time.time()
    for i in range(m):
        batch.append(messages[i])
        if len(batch) >= batch_size:
            for msg in batch:
                producer.produce(msg.encode('ascii'))
            batch = []
    for msg in batch:
        producer.produce(msg.encode('ascii'))
    print("Data ingestion rate for batch size {} equals {}".format(batch_size, (m) / (time.time() - start_time)))
# print('hi')
messages1 = construct_messages(coordinates1)
messages2 = construct_messages(coordinates2)
messages3 = construct_messages(coordinates3)
messages4 = construct_messages(coordinates4)
messages = messages1 + messages2 + messages3 + messages4
# print('hii')
batch_sizes = [1, 5, 10, 20, 50]
for batch_size in batch_sizes:
    generate_checkpoint(messages, batch_size)
```

Asynchronous Kafka Consumer & Queue-Based Messaging with Flask Server-Sent Events for Real-time Bus Location Tracking



```python
from flask import Flask, render_template, Response
from pykafka import KafkaClient
from pykafka.common import OffsetType
import logging
import time
import threading
import queue

def get_kafka_client():
    return KafkaClient(hosts='127.0.0.1:9092')

app = Flask(__name__)

logging.basicConfig(level=logging.INFO)
logger = app.logger

@app.route('/')
@app.route('/index.html')
def index():
    PAGE_TITLE='Amherst Bus Live Map'

    MAP_URL_TEMPLATE = 'https://{s}.tile.openstreetmap.org/{z}/{x}/{y}.png';
    MAP_ATTRIBUTION = '&copy; <a href="https://www.openstreetmap.org/copyright">OpenStreetMap</a> conti
    MAP_STARTING_CENTER = [42.36043,-72.52081]
    MAP_STARTING_ZOOM = 12;
    MAP_MAX_ZOOM = 18;

    KAFKA_TOPIC = 'geodata_stream_topic_123'

    return render_template('index.html', **locals())
```



```python
# Consumer API with asynchronous processing
@app.route('/topic/<topicname>')
def get_messages(topicname):
    client = get_kafka_client()
    message_queue = queue.Queue()

    def consume_messages():
        consumer = client.topics[topicname].get_simple_consumer(
            auto_offset_reset=OffsetType.LATEST,
            reset_offset_on_start=True
        )
        for message in consumer:
            if message is not None:
                message_queue.put(message.value.decode())

    def events():
        start_time = time.time()
        message_count = 0

        while True:
            message = message_queue.get()
            message_count += 1
            yield f'data:{message}\n\n'

            if message_count % 100 == 0:
                elapsed_time = time.time() - start_time
                if elapsed_time > 0:
                    rate = message_count / elapsed_time
                    logger.info(f"Processed {message_count} messages in {elapsed_time:.2f} seconds (Rate: {rate:.2f} messages/sec)")

    # Start a separate thread for consuming messages
    threading.Thread(target=consume_messages, daemon=True).start()

    return Response(events(), mimetype="text/event-stream")

if __name__ == '__main__':
    app.run(debug=True, port=5001)
```

POSSIBLE IMPROVEMENTS

WHILE OUR PROJECT PROVIDES COMPREHENSIVE INSIGHTS INTO REAL-TIME BUS TRACKING AND DATA HANDLING USING KAFKA, THERE IS POTENTIAL FOR FURTHER ENHANCEMENTS IN FUTURE ITERATIONS.

01 INTEGRATION OF ADVANCED GPS AND ERROR CORRECTION

- Adopt high-precision GPS units and sophisticated error correction algorithms for improved location accuracy.
- Impact: Elevated data quality, crucial for reliable analytics and predictions.

02 SCALABLE DATA ARCHITECTURE

- Adoption of Kafka Stream Processing: Leverage Kafka streams for handling large-scale, real-time data efficiently.
- Impact: Enhanced processing capability, vital for real-time analytics and scalability.

03 DATA-DRIVEN OPTIMIZATION

- Route and Schedule Optimization: Utilize data analytics to optimize bus routes and schedules based on traffic patterns and passenger demand.
- Impact: Improved operational efficiency and passenger experience.

POSSIBLE IMPROVEMENTS

WHILE OUR PROJECT PROVIDES COMPREHENSIVE INSIGHTS INTO REAL-TIME BUS TRACKING AND DATA HANDLING USING KAFKA, THERE IS POTENTIAL FOR FURTHER ENHANCEMENTS IN FUTURE ITERATIONS.

04 DYNAMIC PARTITIONING

- Explore dynamic partitioning strategies that adapt to changes in data distribution.
- Impact: potentially avoid hotspots in Kafka partitions.

05 DATA SERIALIZATION OPTIMIZATION

- Evaluate the efficiency of the data serialization/deserialization process.
- Impact: Depending on the chosen serialization format (e.g., JSON, Avro), consider alternatives that may offer better performance. Avro, for instance, is known for its compact binary serialization.

TEST - MONITORING DATA INGESTION RATE

Test Scenario:

01 Utilized dynamic parameter fluctuations to assess the system's ability to adapt and maintain optimal data ingestion rates.

02 Introduced varying loads(2 busses and 4 busses) and parameters to observe the system's responsiveness, ensuring that monitoring mechanisms accurately reflected changes in data ingestion rates under different system conditions.

TEST - MONITORING DATA INGESTION RATE

01 ASYNCHRONOUS VS SYNCHRONOUS IMPLEMENTATION

- Test Scenario: Conducted a comparative analysis between asynchronous and synchronous implementations of data ingestion for real-time bus data.

02 ASYNCHRONOUS PRODUCER MULTITHREADING

- Test Scenario: Evaluated the impact of asynchronous producer multithreading on data ingestion rates.

03 ASYNCHRONOUS PRODUCER BATCHING

- Test Scenario: Examined the efficiency gains achieved by asynchronous producer batching.

TEST - MONITORING DATA INGESTION RATE

04 GENERATING PARTITION KEY FOR PRODUCER

- Test Scenario: Investigated the influence of the partition key generation strategy on Kafka producer performance.

05 CONSUMER ASYNCHRONOUS QUEUE

- Test Scenario: Validated the effectiveness of the asynchronous queue for message handling in the Kafka consumer.

06 CONSUMER KAFKA CONFLUENCE

- Test Scenario: Assessed the performance of Kafka consumer confluence in handling and processing incoming data streams.

EXPERIMENTAL RESULTS

DATA INGESTION	BUSES: 2	BUSES: 4
Normal Producer Rate	6538.76441	7543.6605927
Partition Key for Producer	6857.26076	8226.88199
Asynchronous producer batching for batch size 5	667080.431399	703400.9101
Asynchronous producer batching for batch size 10	660617.056331	666497.3547
Asynchronous producer batching for batch size 20	665868.384283	703513.1669
Asynchronous producer multithreading	676001	705427.03

OBSERVATION

01 NORMAL PRODUCER RATE

- The baseline data ingestion rate without any special configuration
- The increase in the number of buses correlates with a higher ingestion rate.

02 PARTITION KEY FOR PRODUCER

- Introducing a partition key for the producer can affect how data is distributed across Kafka partitions.
- A more effective partition key distribution might be positively impacting ingestion rates, especially evident with four buses.

03 ASYNCHRONOUS PRODUCER BATCHING

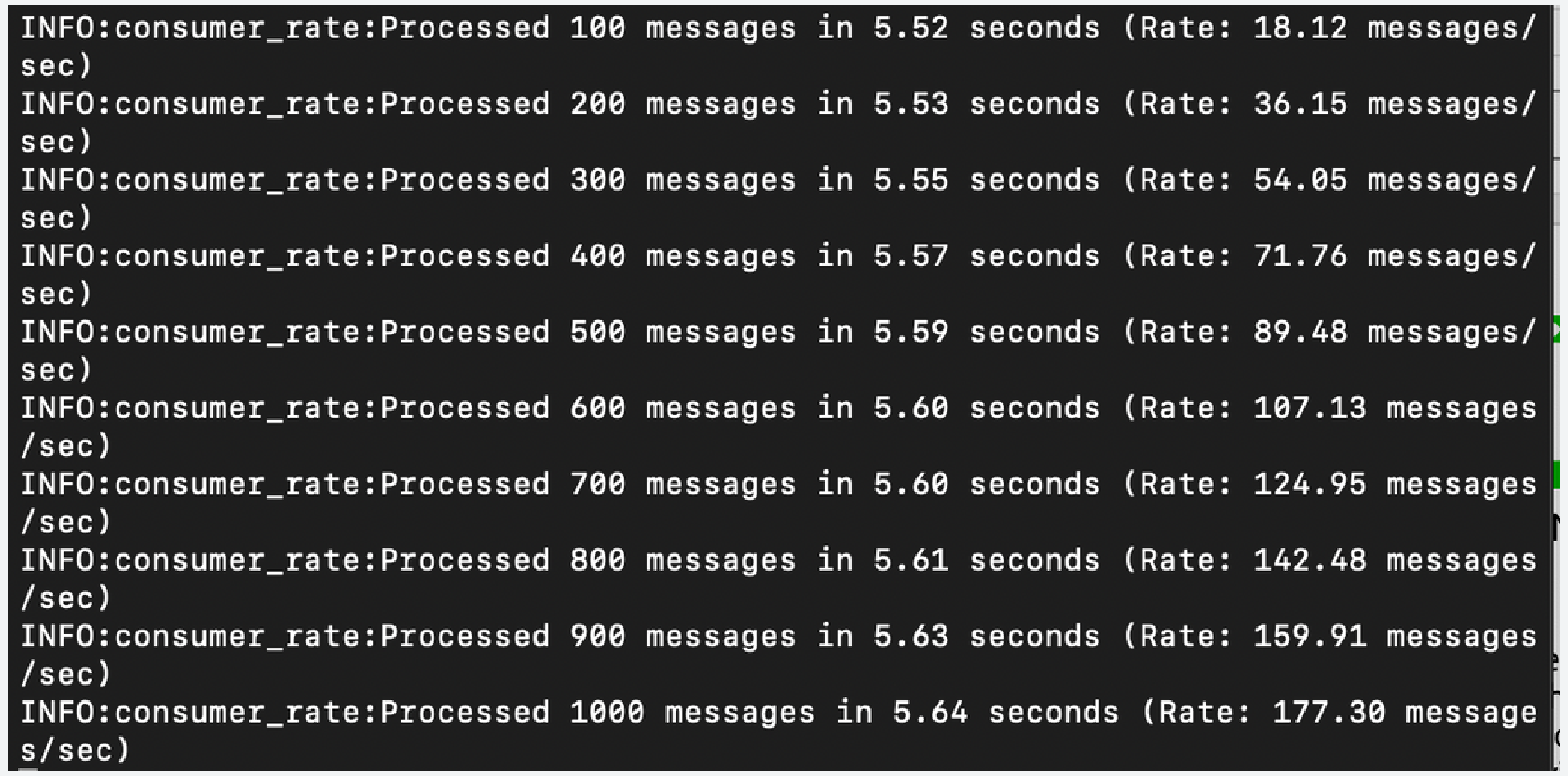
- Batching messages asynchronously before sending them to Kafka.
- Larger batch sizes can lead to more efficient data transmission, but there seems to be some variation in performance based on the specific batch size and the number of buses.

04 ASYNCHRONOUS PRODUCER MULTITHREADING

- Employing multithreading to enhance the producer's efficiency.
- Multithreading is likely improving the producer's ability to handle concurrent tasks, resulting in increased ingestion rates.

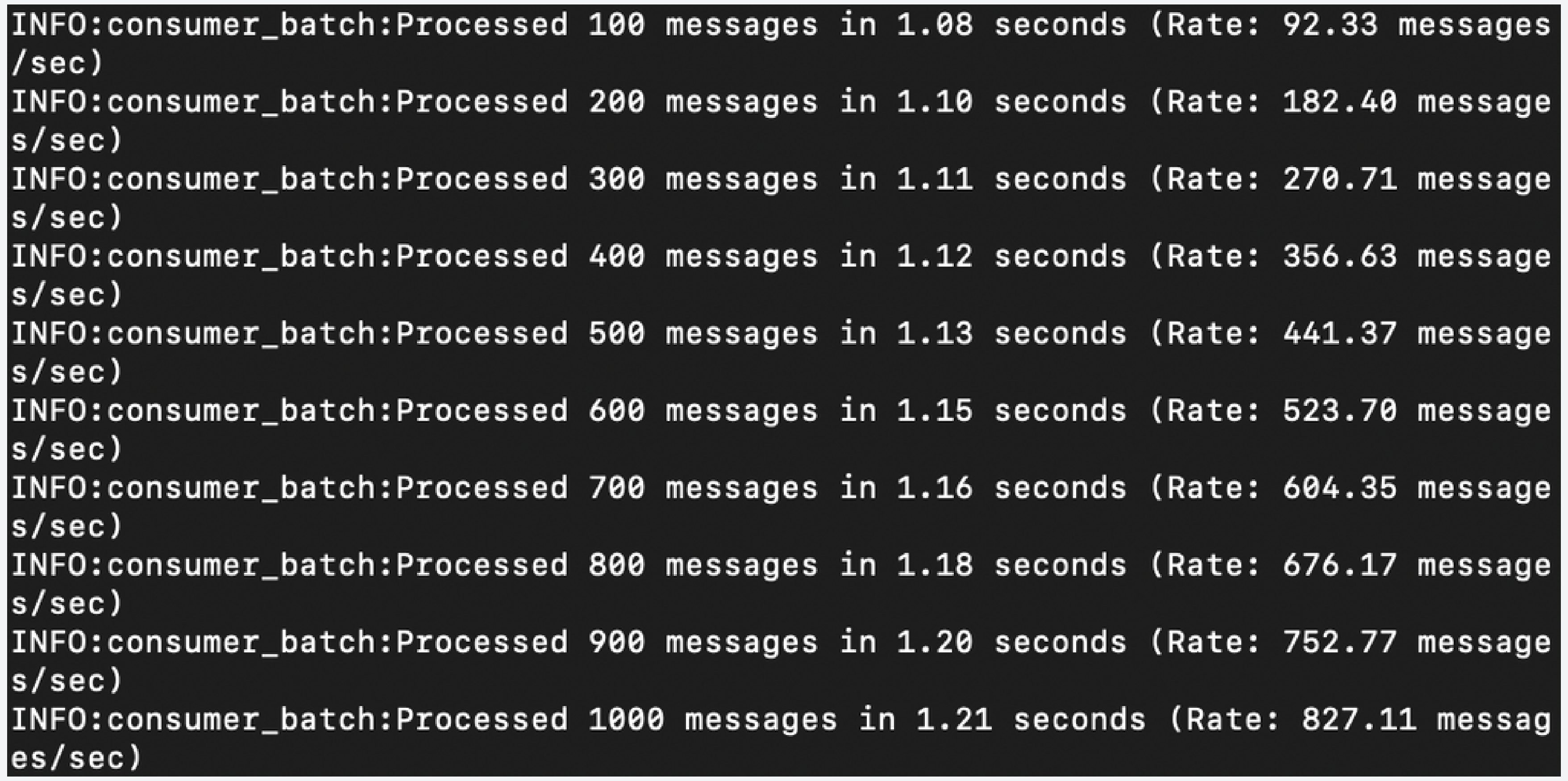
Monitored system responsiveness and message processing rates by implementing asynchronous queue configuaration and adding kafka confluence parameters. Observed that the Asynchrous queue has better ingestion rates than the rest.

CONSUMER DATA INGESTION RATE



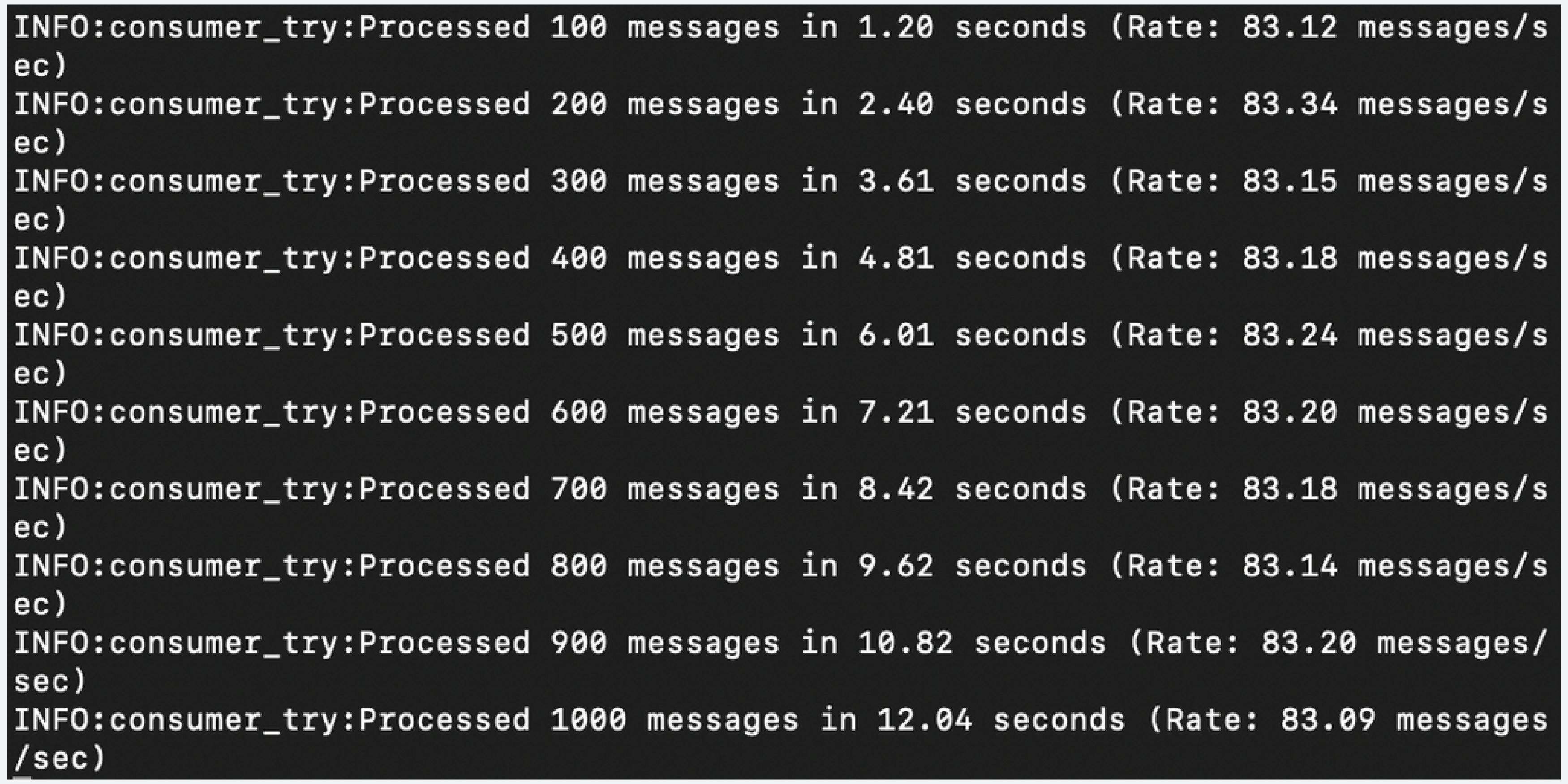
```
INFO:consumer_rate:Processed 100 messages in 5.52 seconds (Rate: 18.12 messages/
sec)
INFO:consumer_rate:Processed 200 messages in 5.53 seconds (Rate: 36.15 messages/
sec)
INFO:consumer_rate:Processed 300 messages in 5.55 seconds (Rate: 54.05 messages/
sec)
INFO:consumer_rate:Processed 400 messages in 5.57 seconds (Rate: 71.76 messages/
sec)
INFO:consumer_rate:Processed 500 messages in 5.59 seconds (Rate: 89.48 messages/
sec)
INFO:consumer_rate:Processed 600 messages in 5.60 seconds (Rate: 107.13 messages
/sec)
INFO:consumer_rate:Processed 700 messages in 5.60 seconds (Rate: 124.95 messages
/sec)
INFO:consumer_rate:Processed 800 messages in 5.61 seconds (Rate: 142.48 messages
/sec)
INFO:consumer_rate:Processed 900 messages in 5.63 seconds (Rate: 159.91 messages
/sec)
INFO:consumer_rate:Processed 1000 messages in 5.64 seconds (Rate: 177.30 message
s/sec)
```

ASYNCHRONOUS QUEUE



```
INFO:consumer_batch:Processed 100 messages in 1.08 seconds (Rate: 92.33 messages
/sec)
INFO:consumer_batch:Processed 200 messages in 1.10 seconds (Rate: 182.40 message
s/sec)
INFO:consumer_batch:Processed 300 messages in 1.11 seconds (Rate: 270.71 message
s/sec)
INFO:consumer_batch:Processed 400 messages in 1.12 seconds (Rate: 356.63 message
s/sec)
INFO:consumer_batch:Processed 500 messages in 1.13 seconds (Rate: 441.37 message
s/sec)
INFO:consumer_batch:Processed 600 messages in 1.15 seconds (Rate: 523.70 message
s/sec)
INFO:consumer_batch:Processed 700 messages in 1.16 seconds (Rate: 604.35 message
s/sec)
INFO:consumer_batch:Processed 800 messages in 1.18 seconds (Rate: 676.17 message
s/sec)
INFO:consumer_batch:Processed 900 messages in 1.20 seconds (Rate: 752.77 message
s/sec)
INFO:consumer_batch:Processed 1000 messages in 1.21 seconds (Rate: 827.11 messag
es/sec)
```

KAFKA CONFLUENCE



```
INFO:consumer_try:Processed 100 messages in 1.20 seconds (Rate: 83.12 messages/s
ec)
INFO:consumer_try:Processed 200 messages in 2.40 seconds (Rate: 83.34 messages/s
ec)
INFO:consumer_try:Processed 300 messages in 3.61 seconds (Rate: 83.15 messages/s
ec)
INFO:consumer_try:Processed 400 messages in 4.81 seconds (Rate: 83.18 messages/s
ec)
INFO:consumer_try:Processed 500 messages in 6.01 seconds (Rate: 83.24 messages/s
ec)
INFO:consumer_try:Processed 600 messages in 7.21 seconds (Rate: 83.20 messages/s
ec)
INFO:consumer_try:Processed 700 messages in 8.42 seconds (Rate: 83.18 messages/s
ec)
INFO:consumer_try:Processed 800 messages in 9.62 seconds (Rate: 83.14 messages/s
ec)
INFO:consumer_try:Processed 900 messages in 10.82 seconds (Rate: 83.20 messages/
sec)
INFO:consumer_try:Processed 1000 messages in 12.04 seconds (Rate: 83.09 messages
/sec)
```

RESULT ANALYSIS

01 PRODUCER RESULT

- The superior data ingestion rate observed with asynchronous producer multithreading for four buses can be attributed to its concurrent and parallel processing capabilities.
- Allows the system to handle multiple tasks simultaneously, enabling efficient parallel execution of data
- Reduces blocking and idle time and optimizes resource utilization.
- As the workload increases with four buses, the concurrent and parallel nature of asynchronous multithreading proves highly effective

02 CONSUMER RESULT

- The Asynchronous Queue exhibits superior ingestion rates over Kafka Confluence and potentially synchronous mechanisms due to its non-blocking behavior, support for concurrency and parallelism, efficient resource utilization, and adaptability to fluctuating workloads. The asynchronous nature of the queue enables the system to efficiently process messages without waiting, handle multiple tasks simultaneously, and dynamically adapt to changes in incoming message rates. This makes it well-suited for optimizing ingestion rates in scenarios with dynamic and high-volume data streams.

GOALS

01 DATA INGESTION RATE ANALYSIS

- Goal: Measure data ingestion speed with varying buses and data update frequencies.
- Achievement: Successfully assessed the impact of these factors on data ingestion.

02 SYSTEM SCALABILITY ASSESSMENT

- Goal: Explore system's scalability by adding more buses and consumers.
- Achievement: Gained insights into how well the system can handle increased loads.

03 KAFKA CONFIGURATION OPTIMIZATION

- Goal: Fine-tune Kafka parameters (partitions keys, batch size, threads).
- Achievement: Optimized data flow and improved system responsiveness.

04 MANAGING HIGH DATA VOLUMES AND PRESENTING ON MAP GUI

- Goal: Address the impact of increased GPS data influx and present the moving bus markers on a GUI using Flask.
- Achievement: Analyzed and managed high data volumes effectively and visualized the marker movement on maps correctly.