

## Big Data Homework 3

*Group 20 - Amber Gonzalez-Pacheco, Karishma Mehta, Shivani Tayade, Sakshi Kumar*

### Setup and Execution Steps:

To begin, I installed Apache Kafka and ZooKeeper using Homebrew, which is the recommended method for managing packages on macOS. I ran `brew install kafka` and `brew install zookeeper` to install both services locally. After installation, I confirmed the Kafka and ZooKeeper binaries were correctly installed in the `/opt/homebrew/opt` directory, and I updated my shell profile file (`.zshrc`) by adding the bin directories of both Kafka and ZooKeeper to my system PATH environment variable. I mistakenly added duplicate paths and needed to fix this by using the `nano` command in the terminal, but eventually resolved it. This allowed me to run Kafka CLI tools like `kafka-topics` and `kafka-server-start` from any terminal session. Lastly, I reloaded my terminal environment using `source ~/.zshrc`.

Once the environment was configured, I launched ZooKeeper by running `zkServer start`, followed by launching the Kafka broker using `kafka-server-start /opt/homebrew/etc/kafka/server.properties`. These two background services are necessary for Kafka to function in its default setup. After both were running, I created a Kafka topic for my project using the command: `kafka-topics --create --topic waymo-rides --bootstrap-server localhost:9092 --partitions 1 --replication-factor 1`. This topic, `waymo-rides`, acted as the communication channel between my custom Kafka producer and consumer scripts.

To work with Kafka in Python, I installed the `kafka-python` library using `pip install kafka-python`. With the Kafka infrastructure in place, I then moved on to writing and running the producer and consumer logic in Python scripts.

For Step 6, I created a custom Python script (`waymo_producer.py`) that acts as a Kafka producer. The script simulates a real-time journey of a Waymo autonomous vehicle from pickup to destination. I designed the simulation to follow a series of status updates — including stages like `booked`, `en_route_to_pickup`, `arriving`, `passenger_onboard`, `in_transit`, `approaching_destination`, and `arrived`. These stages are sent as JSON messages to the Kafka topic `waymo-rides`, with each message including the ride ID, user name, pickup and drop-off locations, estimated time of arrival (`eta_minutes`), proximity in meters, and a timestamp.

I used `uuid` to generate unique ride IDs and Python's `random` and `time` libraries to simulate variability and timing between events. The producer sends an update every 1.5 to 3 seconds to mimic real-world telemetry frequency. The use of JSON serialization via `KafkaProducer`'s `value_serializer` allowed structured data to be transmitted cleanly and decoded later by the consumer.

For Step 7, I wrote a Kafka consumer script (`waymo_consumer.py`) that subscribes to the `waymo-rides` topic. It listens continuously for new ride updates and prints them in a readable, real-time log format. To make the output user-friendly, I included a summary line that appears the first time a new ride ID is encountered — showing the rider, pickup, and destination. This was implemented by keeping track of previously seen ride IDs using a Python set.

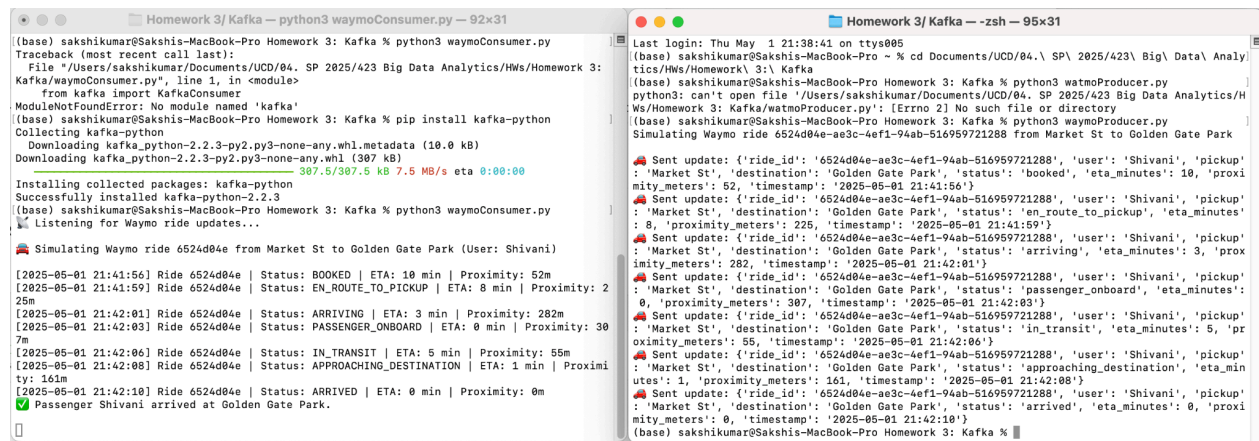
## Big Data Homework 3

*Group 20 - Amber Gonzalez-Pacheco, Karishma Mehta, Shivani Tayade, Sakshi Kumar*

Each time a message is received, the consumer extracts the JSON data and prints a timestamped line showing the current ride status, ETA, and proximity. When a ride reaches the arrived status, the consumer prints a special confirmation message noting that the passenger has successfully arrived. This gives the feeling of a real-time status board or tracking console — similar to what might power the backend of a ride-hailing or autonomous dispatch system.

The final step is running these two scripts in separate terminals, starting with the Consumer script, which waits on the Producer script to finish running.

### Waymo Producer and Consumer Collaboration



The image shows two terminal windows side-by-side. The left window, titled 'Homework 3/ Kafka — python3 waymoConsumer.py — 92x31', shows the execution of the consumer script. It starts with a traceback, then installs kafka-python, and begins listening for updates. It prints a series of status updates for a ride from Market St to Golden Gate Park, including 'BOOKED', 'EN\_ROUTE\_TO\_PICKUP', 'ARRIVING', 'IN\_TRANSIT', 'APPROACHING\_DESTINATION', and finally 'ARRIVED'. The right window, titled 'Homework 3/ Kafka — -zsh — 95x31', shows the execution of the producer script. It starts with a last login message, then runs the producer script. It prints a series of status updates for the same ride, including 'booked', 'en\_route\_to\_pickup', 'arriving', 'in\_transit', and 'arrived'.

The two python scripts, waymoConsumer.py and waymoProducer.py, work in conjunction to simulate a real-time ride-tracking system in Kafka. The latter producer script generates synthetic ride data for a synthetic Waymo ride. It randomly selects a user, pickup location, and destination from a predefined list of values for each variable, then simulates the ride progressing through various stages like “booked,” “en\_route\_to\_pickup,” “in\_transit,” and finally “arrived.” For each stage, the script creates a structured message with metadata such as ETA, proximity, and a timestamp, and sends it to a Kafka topic called *waymo-rides* using a JSON serialiser.

The consumer script listens to this same Kafka topic and prints out updates as they are received. It connects to the broker, deserialises each JSON message, and checks if it's seeing a ride for the first time, if so, it prints a summary of the ride. It then continuously logs each status update with details like ETA and proximity. When the final “arrived” status is received, it prints a completion message. Together, the scripts demonstrate a full publish-subscribe pipeline where real-time messages are produced and consumed in a stream, leveraging Kafka as a high-throughput event broker.

### Real-World Application

Apache Kafka is extensively reliable for handling real-time data streaming due to its ability to modulate fault tolerant communication with reduced latency which is important in dynamic

## Big Data Homework 3

*Group 20 - Amber Gonzalez-Pacheco, Karishma Mehta, Shivani Tayade, Sakshi Kumar*

systems.

Considering a real life scenario, this setup can be used for autonomous car companies like Zoox or even Uber to get real time updates from vehicles to make improvements and adjustments in real time. The consumer script works as a downstream system for example, like for alerts whereas the producer script can be used for pushing dynamic updates.

For safety related applications, Kafka would be a great platform to ensure no information is lost and real time updates are being received in a seamless manner even during unfavorable network conditions.

For example, in the case of hospitals having many ambulances going to and fro across the city during emergency situations. Using Kafka, there can be real time updates that can be sent regarding present location, ambulance status, patient pickup status. The Producer can push updates and the consumer can be at the hospital giving updates to a dashboard and updating the hospital emergency team to help monitor nurses on duty, staff situation and rooms to prepare.