DataEng S24: PubSub

[this lab activity references tutorials at cloud.google.com]

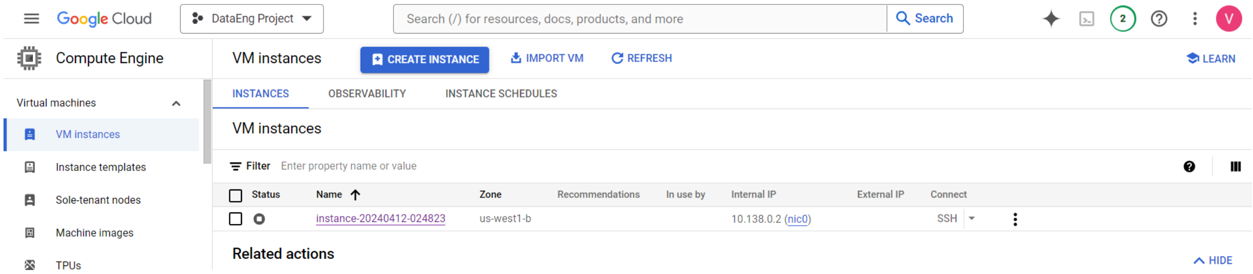
Make a copy of this document and use it to record your results. Store a PDF copy of the document in your git repository along with your code before submitting for this week. For your code, you create several publisher/receiver programs or you might make various features within one program. There is no one single correct way to do it. Regardless, store your code in your repository.

The goal for this week is to gain experience and knowledge of using an asynchronous data transport system (Google PubSub). Complete as many of the following exercises as you can. Proceed at a pace that allows you to learn and understand the use of PubSub with python.

Submit: use the in-class activity submission form which is linked from the Materials page on the class website. Submit by 10pm PT this Friday.

A. [MUST] PubSub Tutorial

1. Get your cloud.google.com account up and running
    a. Redeem your GCP coupon
    b. Login to your GCP console
    c. Create a new, separate VM instance



VM instances

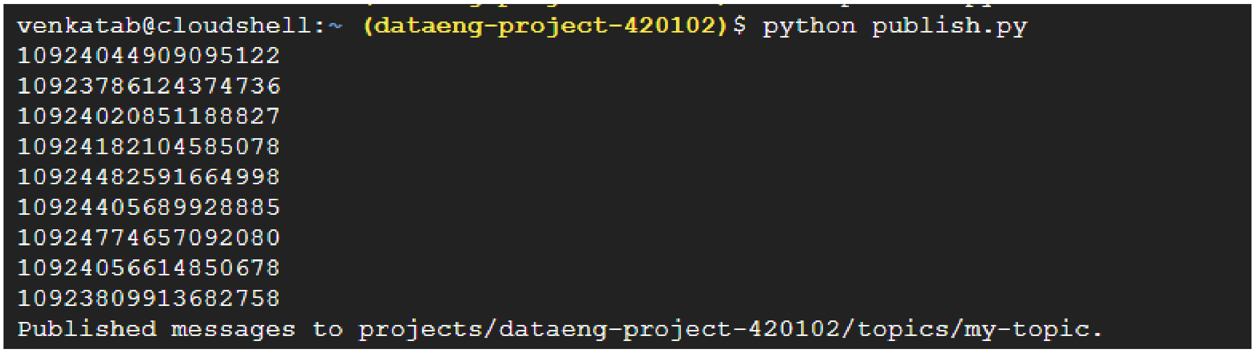
□	Status	Name ↑	Zone	Recommendations	In use by	Internal IP	External IP	Connect
□	○	instance-20240412-024823	us-west1-b			10.138.0.2 (nic0)		SSH

2. Complete this PubSub tutorial: [link]() Note that the tutorial instructs you to destroy your PubSub topic, but you should not destroy your topic just yet. Destroy the topic after you finish the following parts of this in-class assignment.

Publish:
Given code in link:
from google.cloud import pubsub_v1

```
# TODO(developer)
project_id = "dataeng-project-420102"
topic_id = "my-topic"

publisher = pubsub_v1.PublisherClient()
# The `topic_path` method creates a fully qualified identifier
# in the form `projects/{project_id}/topics/{topic_id}`
topic_path = publisher.topic_path(project_id, topic_id)

for n in range(1, 10):
    data_str = f"Message number {n}"
    # Data must be a bytestring
    data = data_str.encode("utf-8")
    # When you publish a message, the client returns a future.
    future = publisher.publish(topic_path, data)
    print(future.result())

print(f"Published messages to {topic_path}.")
```

Output:



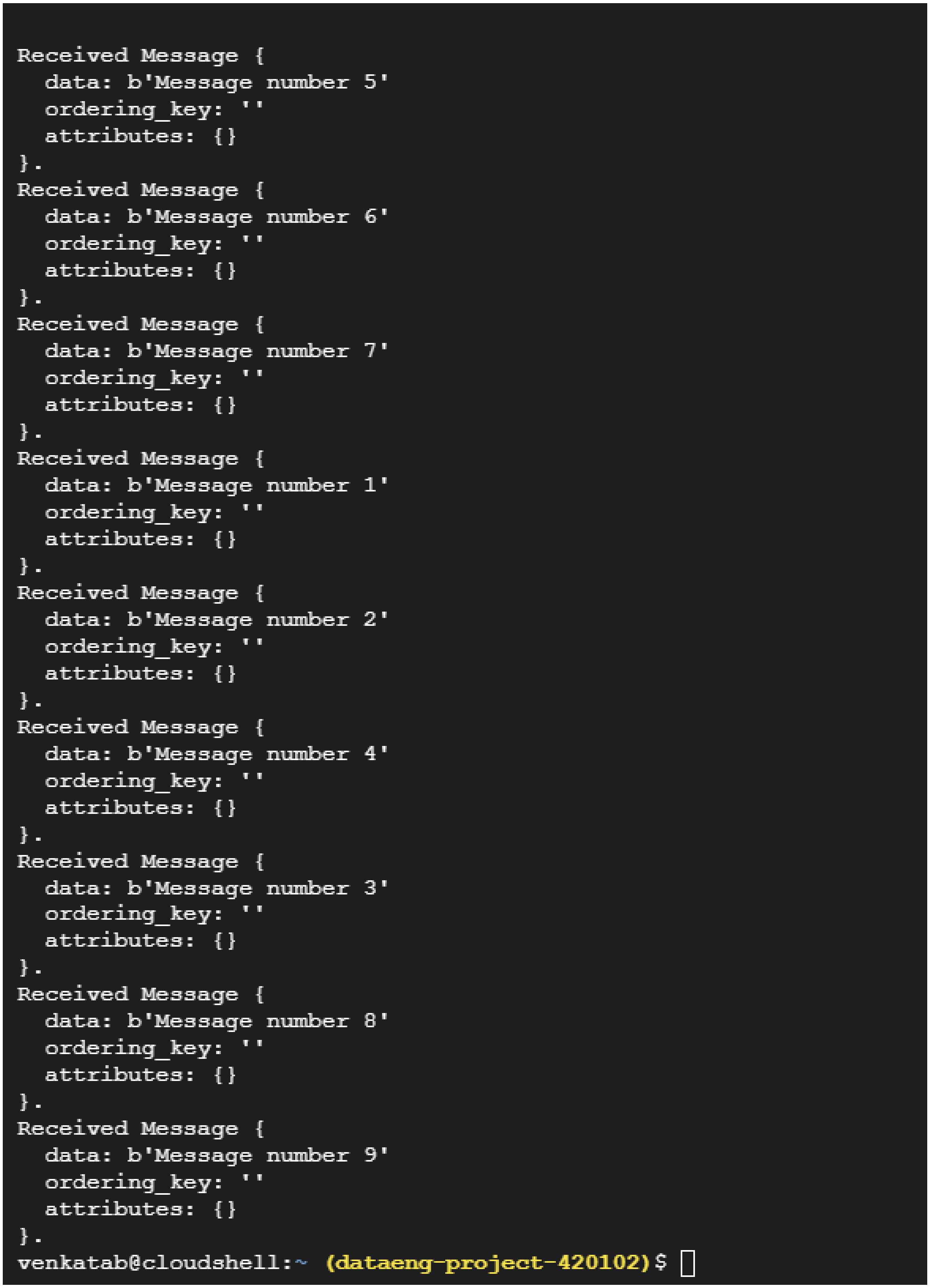
```
venkatab@cloudshell:~ (dataeng-project-420102)$ python publish.py
10924044909095122
10923786124374736
10924020851188827
10924182104585078
10924482591664998
10924405689928885
10924774657092080
10924056614850678
10923809913682758
Published messages to projects/dataeng-project-420102/topics/my-topic.
```

Receive:

Given code in link:

```
from concurrent.futures import TimeoutError
from google.cloud import pubsub_v1

# TODO(developer)
project_id = "dataeng-project-420102"
subscription_id = "my-sub"
# Number of seconds the subscriber should listen for messages
timeout = 5.0
```

```
subscriber = pubsub_v1.SubscriberClient()
# The `subscription_path` method creates a fully qualified identifier
# in the form `projects/{project_id}/subscriptions/{subscription_id}`
subscription_path = subscriber.subscription_path(project_id, subscription_id)

def callback(message: pubsub_v1.subscriber.message.Message) -> None:
    print(f"Received {message}.")
    message.ack()

streaming_pull_future = subscriber.subscribe(subscription_path, callback=callback)
print(f"Listening for messages on {subscription_path}..\n")

# Wrap subscriber in a 'with' block to automatically call close() when done.
with subscriber:
    try:
        # When `timeout` is not set, result() will block indefinitely,
        # unless an exception is encountered first.
        streaming_pull_future.result(timeout=timeout)
    except TimeoutError:
        streaming_pull_future.cancel()  # Trigger the shutdown.
        streaming_pull_future.result()  # Block until the shutdown is complete.
```

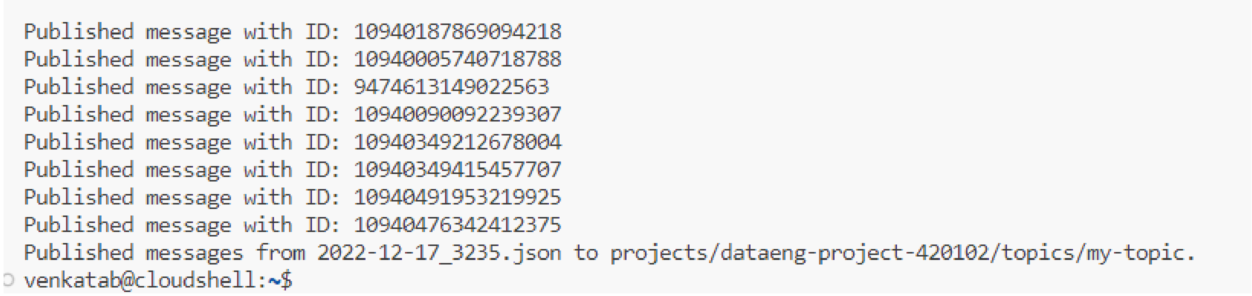


```
Received Message {
  data: b'Message number 5'
  ordering_key: ''
  attributes: {}
}.
Received Message {
  data: b'Message number 6'
  ordering_key: ''
  attributes: {}
}.
Received Message {
  data: b'Message number 7'
  ordering_key: ''
  attributes: {}
}.
Received Message {
  data: b'Message number 1'
  ordering_key: ''
  attributes: {}
}.
Received Message {
  data: b'Message number 2'
  ordering_key: ''
  attributes: {}
}.
Received Message {
  data: b'Message number 4'
  ordering_key: ''
  attributes: {}
}.
Received Message {
  data: b'Message number 3'
  ordering_key: ''
  attributes: {}
}.
Received Message {
  data: b'Message number 8'
  ordering_key: ''
  attributes: {}
}.
Received Message {
  data: b'Message number 9'
  ordering_key: ''
  attributes: {}
}.
venkatab@cloudshell:~ (dataeng-project-420102)$
```

B. [MUST] Create Sample Data

1. Get data from https://busdata.cs.pdx.edu/api/getBreadCrumbs for two Vehicle IDs from among those that have been assigned to you for the class project.
2. Save this data in a sample file (named bcsample.json)
3. Update the publisher python program that you created in the PubSub tutorial to read and parse your bcsample.json file and send its contents, one record at a time, to the my-topic PubSub topic that you created for the tutorial.

   Output:



```
Published message with ID: 10940187869094218
Published message with ID: 10940005740718788
Published message with ID: 9474613149022563
Published message with ID: 10940090092239307
Published message with ID: 10940349212678004
Published message with ID: 10940349415457707
Published message with ID: 10940491953219925
Published message with ID: 10940476342412375
Published messages from 2022-12-17_3235.json to projects/dataeng-project-420102/topics/my-topic.
venkatab@cloudshell:~$
```

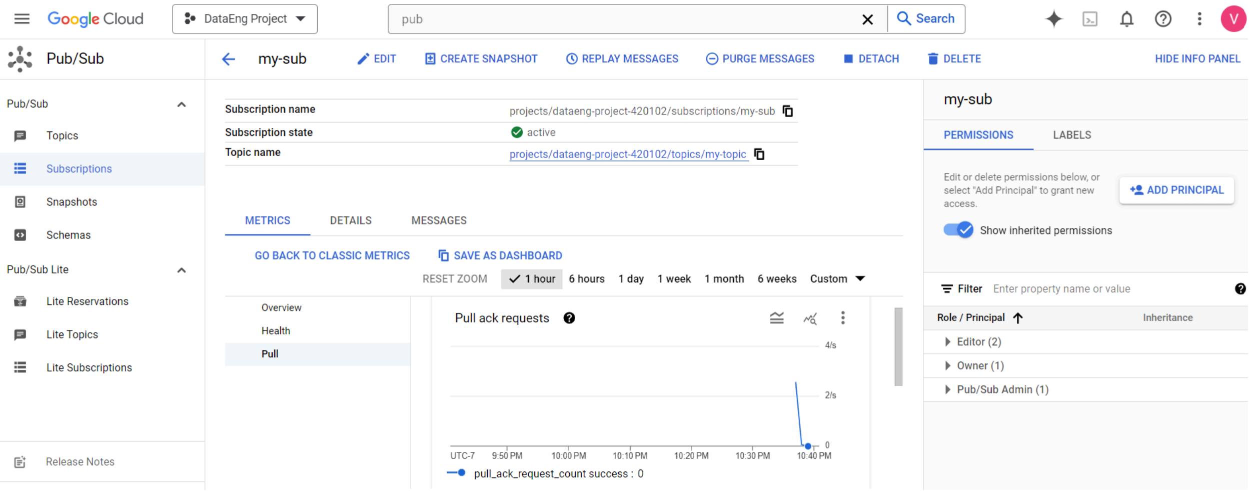
4. Use your receiver python program (from the tutorial) to consume your records.

   Output:

```
Received Message {
  data: b'{"EVENT_NO_TRIP": 221577937, "EVENT_NO_STOP": 2215...'
  ordering_key: ''
  attributes: {}
}.
Received Message {
  data: b'{"EVENT_NO_TRIP": 221577681, "EVENT_NO_STOP": 2215...'
  ordering_key: ''
  attributes: {}
}.
Received Message {
  data: b'{"EVENT_NO_TRIP": 221577681, "EVENT_NO_STOP": 2215...'
  ordering_key: ''
  attributes: {}
}.
Received Message {
  data: b'{"EVENT_NO_TRIP": 221577681, "EVENT_NO_STOP": 2215...'
  ordering_key: ''
  attributes: {}
}.
Received Message {
  data: b'{"EVENT_NO_TRIP": 221577937, "EVENT_NO_STOP": 2215...'
  ordering_key: ''
  attributes: {}
}.
Received Message {
  data: b'{"EVENT_NO_TRIP": 221577937, "EVENT_NO_STOP": 2215...'
  ordering_key: ''
  attributes: {}
}.
Received Message {
  data: b'{"EVENT_NO_TRIP": 221577681, "EVENT_NO_STOP": 2215...'
  ordering_key: ''
  attributes: {}
}.
```

C. [MUST] PubSub Monitoring

1. Review the PubSub Monitoring tutorial: link and work through the steps listed there. You might need to rerun your publisher and receiver programs multiple times to trigger enough activity to monitor your my-topic effectively.



Subscription name projects/dataeng-project-420102/subscriptions/my-sub
Subscription state ✓ active
Topic name projects/dataeng-project-420102/topics/my-topic

D. [MUST] PubSub Storage

1. What happens if you run your receiver multiple times while only running the publisher once?

   Ans: When you run the receiver multiple times while only running the publisher once, we will get messages for the first time. But on the second run, we will not get any messages as we are not running publisher again.

2. Before the consumer runs, where might the data go, where might it be stored?
   Ans:Data Center

3. Is there a way to determine how much data PubSub is storing for your topic? Do the PubSub monitoring tools help with this?

   Ans: Google Cloud Pub/Sub offers monitoring tools and metrics that provide insights into the data storage status of your topics. Although these tools don't directly reveal the exact storage usage, they offer valuable indicators such as subscription activity, message throughput, and latency, aiding in understanding storage trends. By analyzing these metrics, you can estimate the stored data volume indirectly.

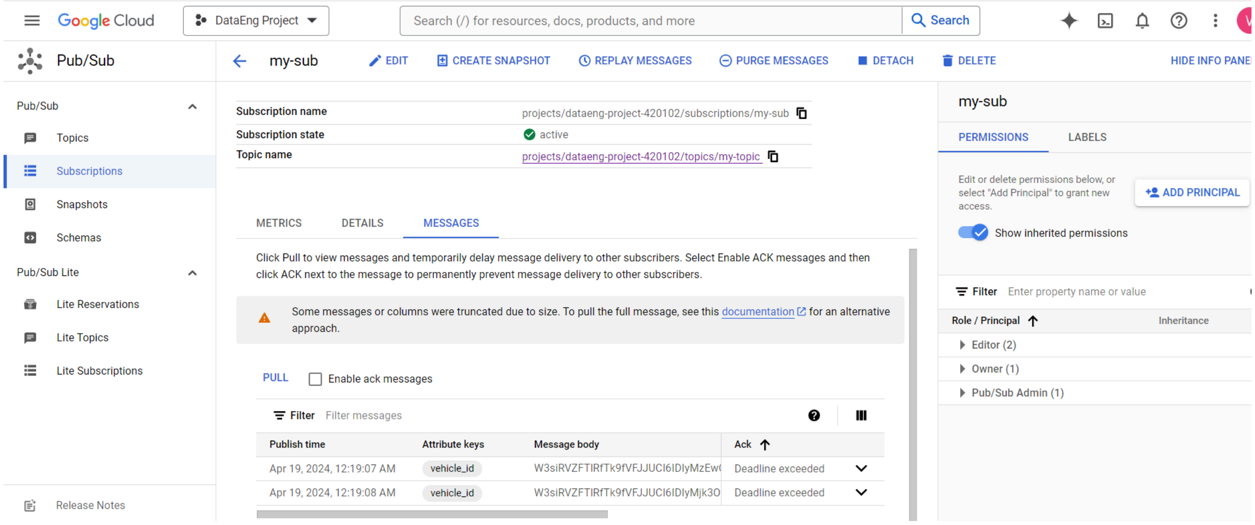
4. Create a “topic_clean.py” receiver program that reads and discards all records for a given topic. This type of program can be very useful for debugging your project code.

E. [SHOULD] Multiple Publishers

1. Clear all data from the topic (run your topic_clean.py program whenever you need to clear your topic)
2. Run two versions of your publisher concurrently, have each of them send all of your sample records. When finished, run your receiver once. Describe the results.

Concurrently running two versions of the publisher, each transmitting all sample records, and subsequently executing a single run of the receiver can result in duplicated message consumption by the receiver. Despite the presence of two publishers, both dispatching identical records, the receiver handles each message autonomously, leading to duplicate processing. Consequently, the receiver may face redundant data processing, potentially affecting system performance and resource utilization.

Implementing message deduplication mechanisms or ensuring coordinated message transmission by publishers can help optimize processing and reduce duplication.



Subscription name	projects/dataeng-project-420102/subscriptions/my-sub
Subscription state	active
Topic name	projects/dataeng-project-420102/topics/my-topic

Click Pull to view messages and temporarily delay message delivery to other subscribers. Select Enable ACK messages and then click ACK next to the message to permanently prevent message delivery to other subscribers.

Some messages or columns were truncated due to size. To pull the full message, see this documentation for an alternative approach.

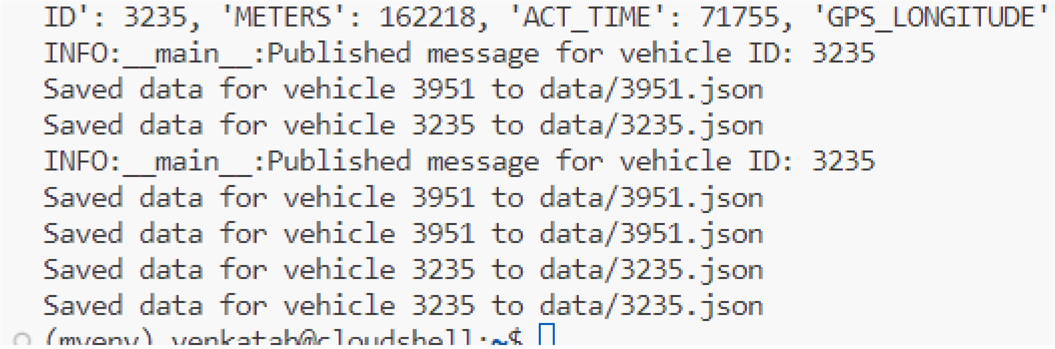
☐ Enable ack messages

Publish time	Attribute keys	Message body	Ack
Apr 19, 2024, 12:19:07 AM	vehicle_id	W3siRVZFTIRfTk9fVFJJUCI6IDlyMzEw(	Deadline exceeded
Apr 19, 2024, 12:19:08 AM	vehicle_id	W3siRVZFTIRfTk9fVFJJUCI6IDlyMjk3O	Deadline exceeded

Edit or delete permissions below, or select "Add Principal" to grant new access.

Show inherited permissions

Role / Principal	Inheritance
Editor (2)	
Owner (1)	
Pub/Sub Admin (1)	

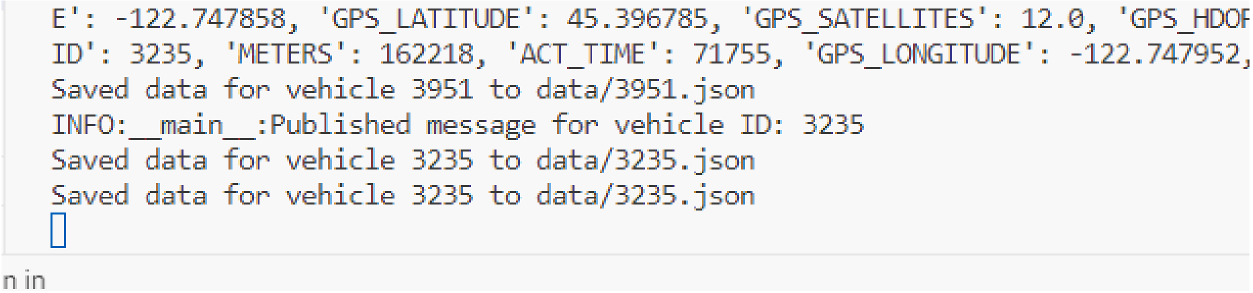


```
  ID': 3235, 'METERS': 162218, 'ACT_TIME': 71755, 'GPS_LONGITUDE'
  INFO:__main__:Published message for vehicle ID: 3235
  Saved data for vehicle 3951 to data/3951.json
  Saved data for vehicle 3235 to data/3235.json
  INFO:__main__:Published message for vehicle ID: 3235
  Saved data for vehicle 3951 to data/3951.json
  Saved data for vehicle 3951 to data/3951.json
  Saved data for vehicle 3235 to data/3235.json
  Saved data for vehicle 3235 to data/3235.json
○ (myenv) venkatab@cloudshell:~$ 
```

F. [SHOULD] Multiple Concurrent Publishers and Receivers

1. Clear all data from the topic
2. Update your publisher code to include a 250 msec sleep after each send of a message to the topic.
3. Run two or three concurrent publishers and two concurrent receivers all at the same time. Have your receivers redirect their output to separate files so that you can sort out the results more easily.
4. Describe the results.

   Simultaneously running multiple publishers and receivers can significantly influence the message processing dynamics in the Pub/Sub system. The concurrent action of publishers boosts the rate of message ingestion, resulting in a higher volume of messages published to the topics. Concurrent receivers, meanwhile, enhance processing throughput by consuming messages concurrently, leading to faster message consumption and acknowledgment. Redirecting the output of each receiver to separate files streamlines result sorting and analysis, making it easier to comprehend and interpret the received messages. This concurrent operation improves the system's scalability, throughput, and flexibility in managing Pub/Sub message streams, ultimately enhancing its efficiency and responsiveness. The concurrent operation of multiple publishers and receivers can potentially reveal redundancy in the message processing flow within the Pub/Sub system. Redundancy may occur when multiple publishers unintentionally publish duplicate messages or when multiple receivers consume and process the same messages independently.



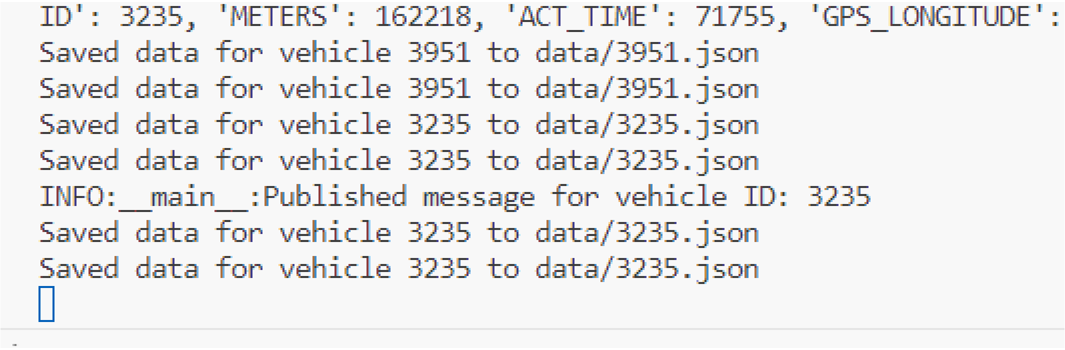
```
E': -122.747858, 'GPS_LATITUDE': 45.396785, 'GPS_SATELLITES': 12.0, 'GPS_HDOP
ID': 3235, 'METERS': 162218, 'ACT_TIME': 71755, 'GPS_LONGITUDE': -122.747952,
Saved data for vehicle 3951 to data/3951.json
INFO:__main__:Published message for vehicle ID: 3235
Saved data for vehicle 3235 to data/3235.json
Saved data for vehicle 3235 to data/3235.json
```

n in

F. [ASPIRE] Multiple Subscriptions

1. So far your receivers have all been competing with each other for data. Next, create a new subscription for each receiver so that each one receives a full copy of the data sent by the publisher. Parameterize your receiver so that you can specify a separate subscription for each receiver.
2. Rerun the multiple concurrent publishers/receivers test from the previous section. Assign each receiver to its own subscription.
3. Describe the results.

   Executing the multiple concurrent publishers and receivers test with each receiver assigned to its own subscription demonstrates improved system performance and resource management in the Pub/Sub environment. By allocating dedicated subscriptions to each receiver, message consumption becomes more distributed, mitigating potential bottlenecks and enhancing parallel processing capabilities.



```
ID': 3235, 'METERS': 162218, 'ACT_TIME': 71755, 'GPS_LONGITUDE':
Saved data for vehicle 3951 to data/3951.json
Saved data for vehicle 3951 to data/3951.json
Saved data for vehicle 3235 to data/3235.json
Saved data for vehicle 3235 to data/3235.json
INFO:__main__:Published message for vehicle ID: 3235
Saved data for vehicle 3235 to data/3235.json
Saved data for vehicle 3235 to data/3235.json
```