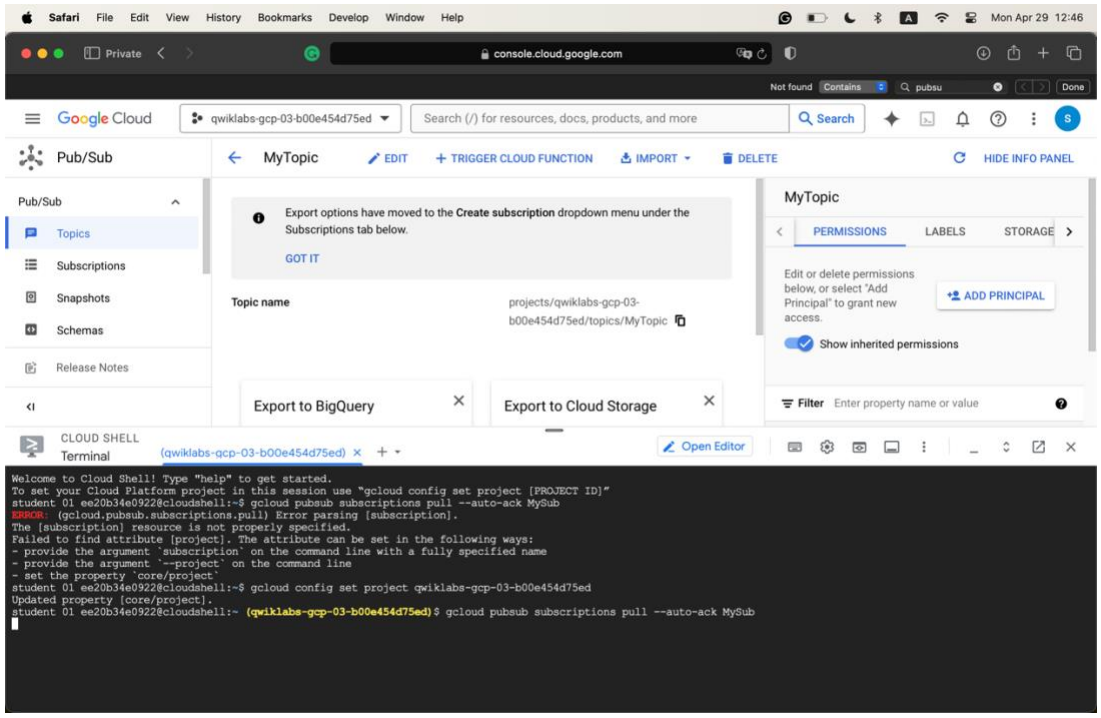
Tony Nguyen

Dr. Shawn Bowers

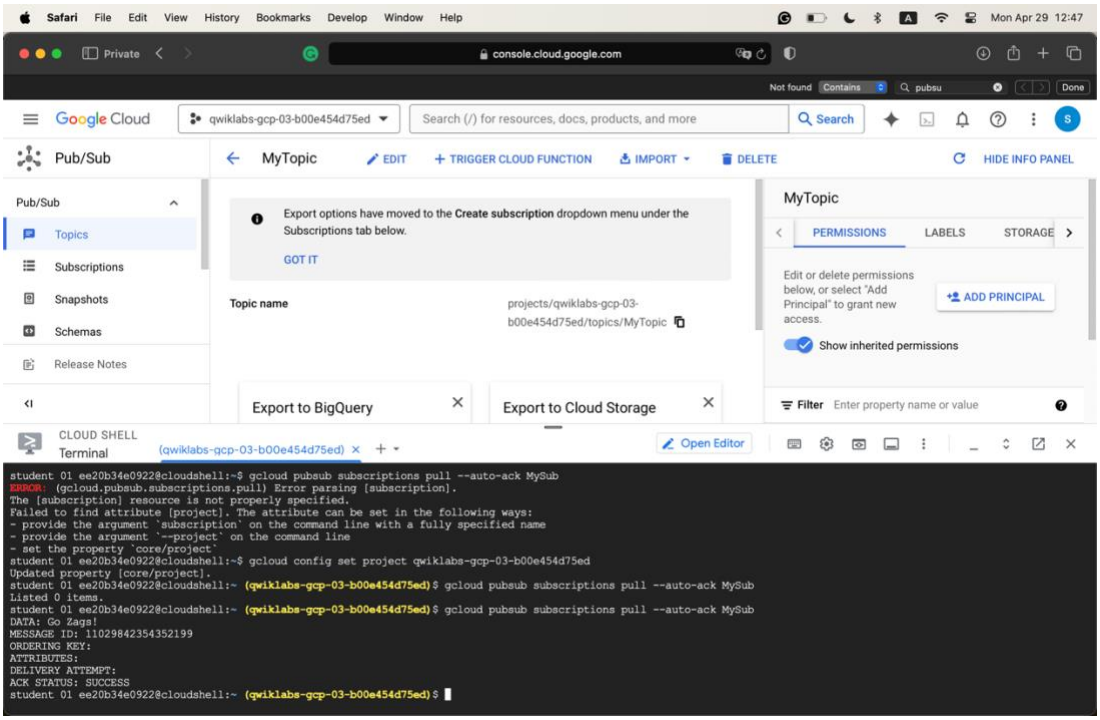
CPSC 324 01

27 April 2024

Homework 6

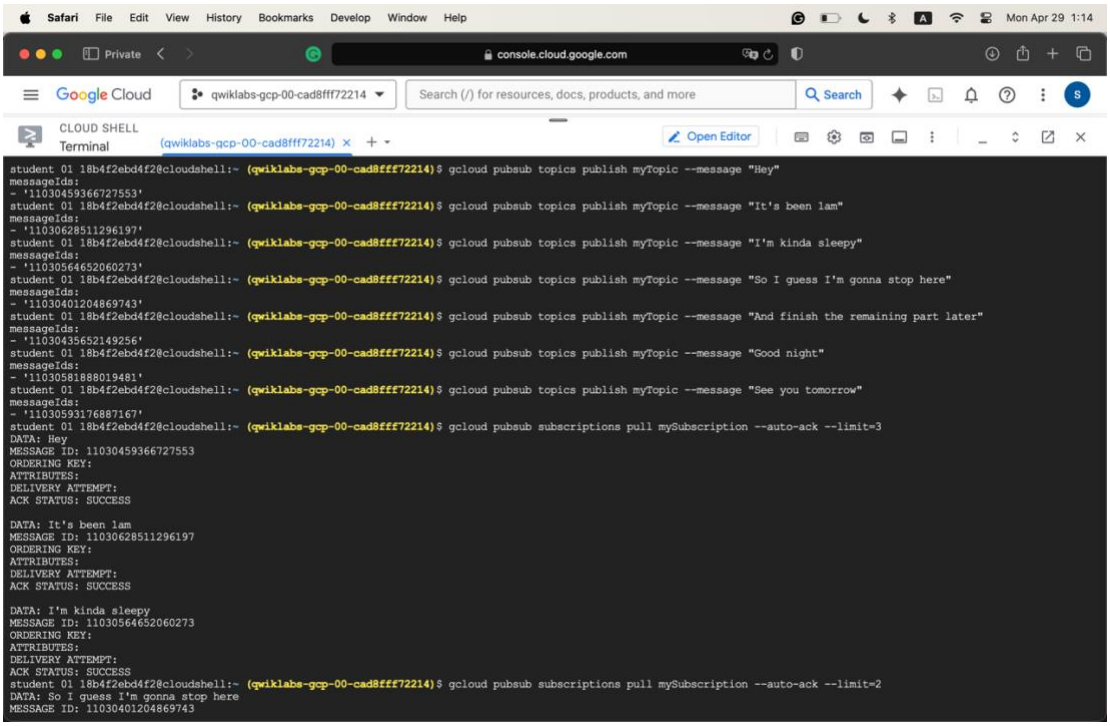
1. Question 1
    - Hello world
        - 

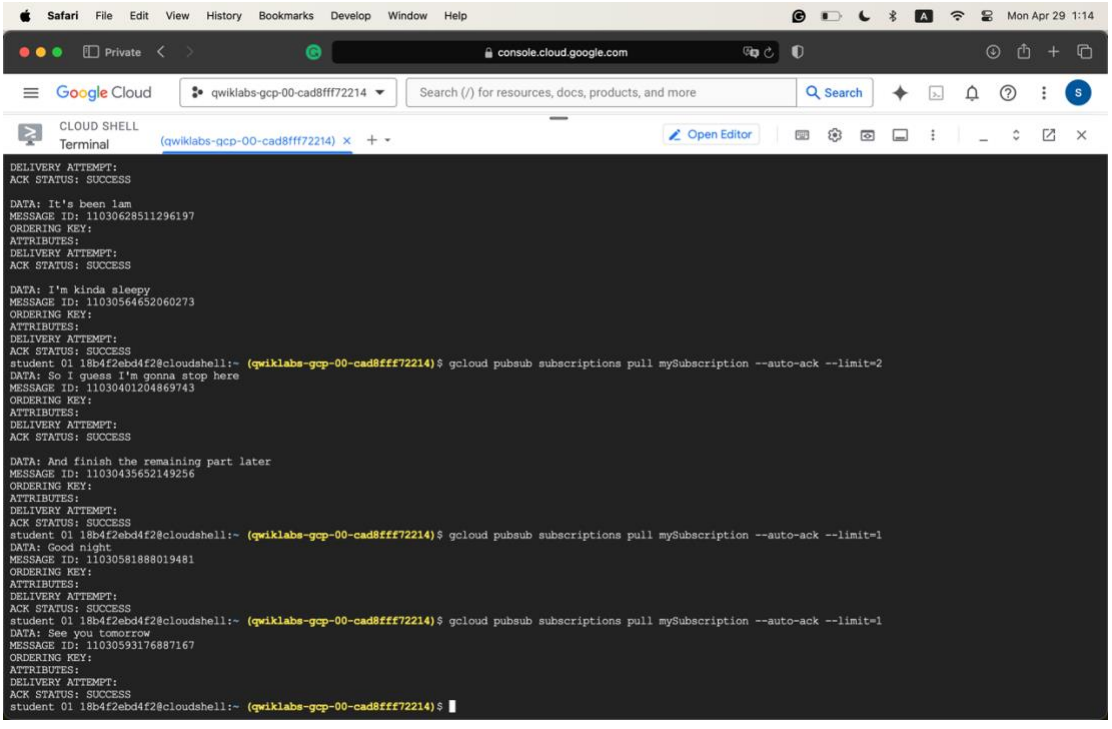
    - Go Zags

- 

2. Question 2
   - Write down CLI commands:
     - Create a topic
       - gcloud pubsub topics create myTopic
     - To see three topics
       - gcloud pubsub topics list
     - Clean up
       - gcloud pubsub topics delete Test1
     - Create a subcription
       - gcloud pubsub subscriptions create --topic myTopic mySubscription
     - To see the subscriptions
       - gcloud pubsub topics list-subscriptions myTopic
     - To delete the two tests

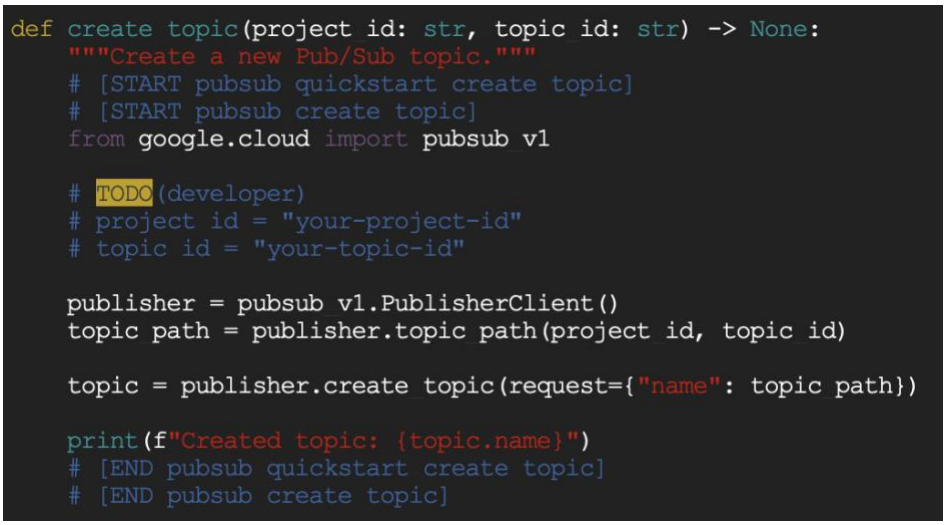
        - gcloud pubsub subscriptions delete Test1
    - To publish a message
        - gcloud pubsub topics publish myTopic --message "Hello"
    - To pull a message
        - gcloud pubsub subscriptions pull mySubscription --auto-ack
        - Run several times to see more results
        - gcloud pubsub topics publish myTopic --message "Publisher is starting to get the hang of Pub/Sub"
    - limit flag
        - gcloud pubsub subscriptions pull mySubscription --auto-ack --limit=3
- Four messages order
    - "Hello"
    - "Publisher's name is Tony"
    - "Publisher likes to eat BunBO"
    - "Publisher thinks Pub/Sub is awesome"
- Results

- 

- 

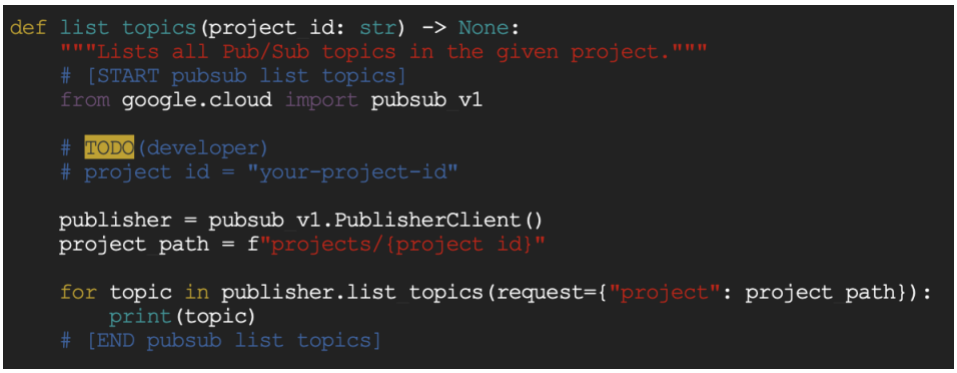
3. Question 3
   - Notes
     - Pub/Sub is an asynchronous global messaging service. Three terms

    - Topics
    - Publishing
    - Subscribing

- create_topic() in Step 4
    - 

```
def create_topic(project_id: str, topic_id: str) -> None:
    """Create a new Pub/Sub topic."""
    # [START pubsub_quickstart_create_topic]
    # [START pubsub_create_topic]
    from google.cloud import pubsub_v1

    # TODO(developer)
    # project_id = "your-project-id"
    # topic_id = "your-topic-id"

    publisher = pubsub_v1.PublisherClient()
    topic_path = publisher.topic_path(project_id, topic_id)

    topic = publisher.create_topic(request={"name": topic_path})

    print(f"Created topic: {topic.name}")
    # [END pubsub_quickstart_create_topic]
    # [END pubsub_create_topic]
```

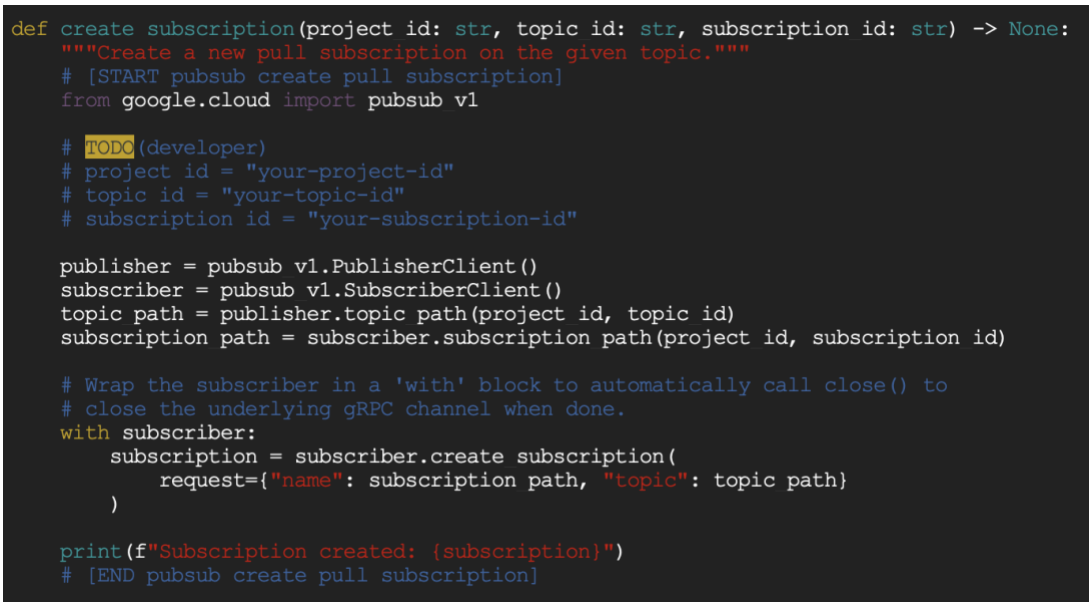
    - This code creates a Publisher object. Then, it creates a topic from that object. It does not seem to pass it into any function.

- list_topic() in Step 4
    - 

```
def list_topics(project_id: str) -> None:
    """Lists all Pub/Sub topics in the given project."""
    # [START pubsub_list_topics]
    from google.cloud import pubsub_v1

    # TODO(developer)
    # project_id = "your-project-id"

    publisher = pubsub_v1.PublisherClient()
    project_path = f"projects/{project_id}"

    for topic in publisher.list_topics(request={"project": project_path}):
        print(topic)
    # [END pubsub_list_topics]
```

    - This code creates a PublisherClient object and then finds the project path that the user passes in. Then, it sends a request to query the data, then print it out.

- To create a topic: python publisher.py $GOOGLE_CLOUD_PROJECT create MyTopic

- Create a subscription
  - Create a subscription: python subscriber.py $GOOGLE_CLOUD_PROJECT create MyTopic MySub
  - 

```
def create_subscription(project_id: str, topic_id: str, subscription_id: str) -> None:
    """Create a new pull subscription on the given topic."""
    # [START pubsub_create_pull_subscription]
    from google.cloud import pubsub_v1

    # TODO(developer)
    # project_id = "your-project-id"
    # topic_id = "your-topic-id"
    # subscription_id = "your-subscription-id"

    publisher = pubsub_v1.PublisherClient()
    subscriber = pubsub_v1.SubscriberClient()
    topic_path = publisher.topic_path(project_id, topic_id)
    subscription_path = subscriber.subscription_path(project_id, subscription_id)

    # Wrap the subscriber in a 'with' block to automatically call close() to
    # close the underlying gRPC channel when done.
    with subscriber:
        subscription = subscriber.create_subscription(
            request={"name": subscription_path, "topic": topic_path}
        )

    print(f"Subscription created: {subscription}")
    # [END pubsub_create_pull_subscription]
```

  - For this function, it creates two Publisher and Subscriber objects, which are then used to obtain a topic path and a subscription path. Finally, those two paths are used to create a subscription using the subscriber object.

- Create pull messages

```python
def receive_messages(
    project_id: str, subscription_id: str, timeout: Optional[float] = None
) -> None:
    """Receives messages from a pull subscription."""
    # [START pubsub_subscriber_async_pull]
    # [START pubsub_quickstart_subscriber]
    from concurrent.futures import TimeoutError
    from google.cloud import pubsub_v1

    # TODO(developer)
    # project_id = "your-project-id"
    # subscription_id = "your-subscription-id"
    # 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.
    # [END pubsub_subscriber_async_pull]
    # [END pubsub_quickstart_subscriber]
```

- This function creates a subscriber object, which is then used to get the subscription_path. This path is later passed to obtain a streaming pull future (I'm not sure what this is used for).

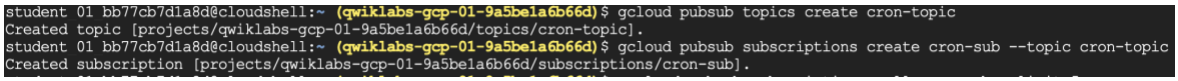
4. Question 4

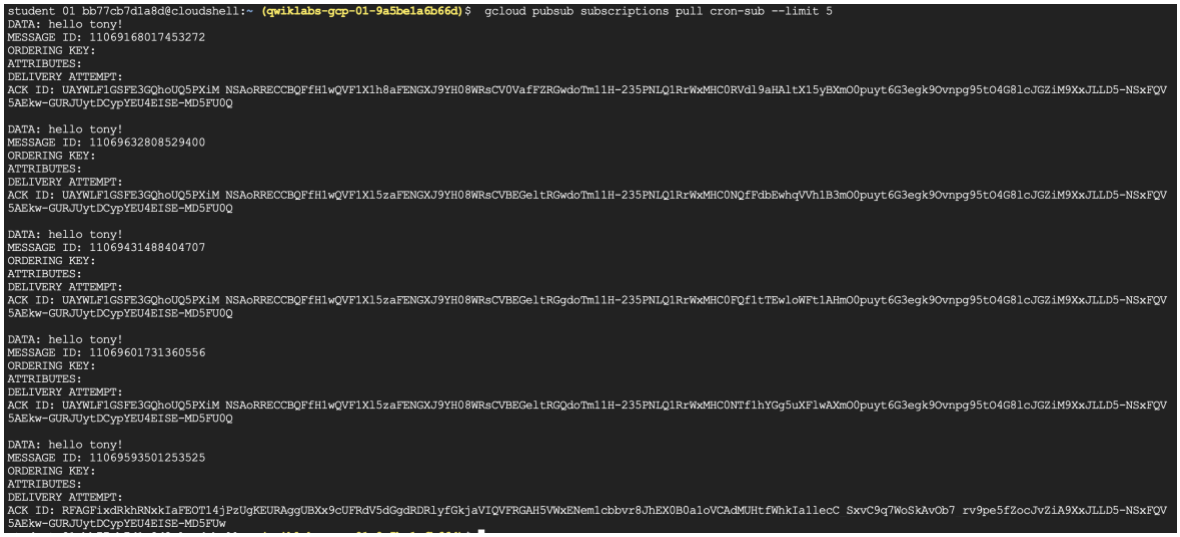
- send_messages.py

```
student_01_1a50d9dfad53@cloudshell:~$ python send_messages.py
Published a message to partition 0 and offset 0.
student_01_1a50d9dfad53@cloudshell:~$ gcloud config set project qwiklabs-gcp-01-1af8635687ae
Updated property [core/project].
student_01_1a50d9dfad53@cloudshell:~ (qwiklabs-gcp-01-1af8635687ae)$ python send_messages.py
Published a message to partition 0 and offset 1.
student_01_1a50d9dfad53@cloudshell:~ (qwiklabs-gcp-01-1af8635687ae)$ python send_messages.py
Published a message to partition 0 and offset 2.
student_01_1a50d9dfad53@cloudshell:~ (qwiklabs-gcp-01-1af8635687ae)$ python send_messages.py
Published a message to partition 0 and offset 3.
student_01_1a50d9dfad53@cloudshell:~ (qwiklabs-gcp-01-1af8635687ae)$ python send_messages.py
Published a message to partition 0 and offset 4.
student_01_1a50d9dfad53@cloudshell:~ (qwiklabs-gcp-01-1af8635687ae)$ python send_messages.py
Published a message to partition 0 and offset 5.
student_01_1a50d9dfad53@cloudshell:~ (qwiklabs-gcp-01-1af8635687ae)$ python send_messages.py
Published a message to partition 0 and offset 6.
```

- receive_message.py
  - 

- The difference between Pub/Sub and Pub/Sub Lite
  - Pub/Sub: an asynchronous and scalable messaging service that decouples services producing messages from services processing those messages.
  - Pub/Sub Lite: A separate but similar messaging service built for lower cost. It offers lower reliability compared to Pub/Sub.

5. Question 5
   - 

   - 

6. Question 6
   - gcloud commands
     - Create a dataset
       - bq mk taxirides

- Create a table
  - bq mk \

    --time_partitioning_field timestamp \

    --schema ride_id:string,point_idx:integer,latitude:float,longitude:float,\

    timestamp:timestamp,meter_reading:float,meter_increment:float,ride_

    status:string,\

    passenger_count:integer -t taxirides.realtime
- Create a cloud storage
  - export BUCKET_NAME=[PROJECT_ID]
  - gsutil mb gs://$BUCKET_NAME/
- Deploy the dataflow template
  - gcloud dataflow jobs run iotflow \

    --gcs-location gs://dataflow-templates-europe-

    west4/latest/PubSub_to_BigQuery \

    --region europe-west4 \

    --worker-machine-type e2-medium \

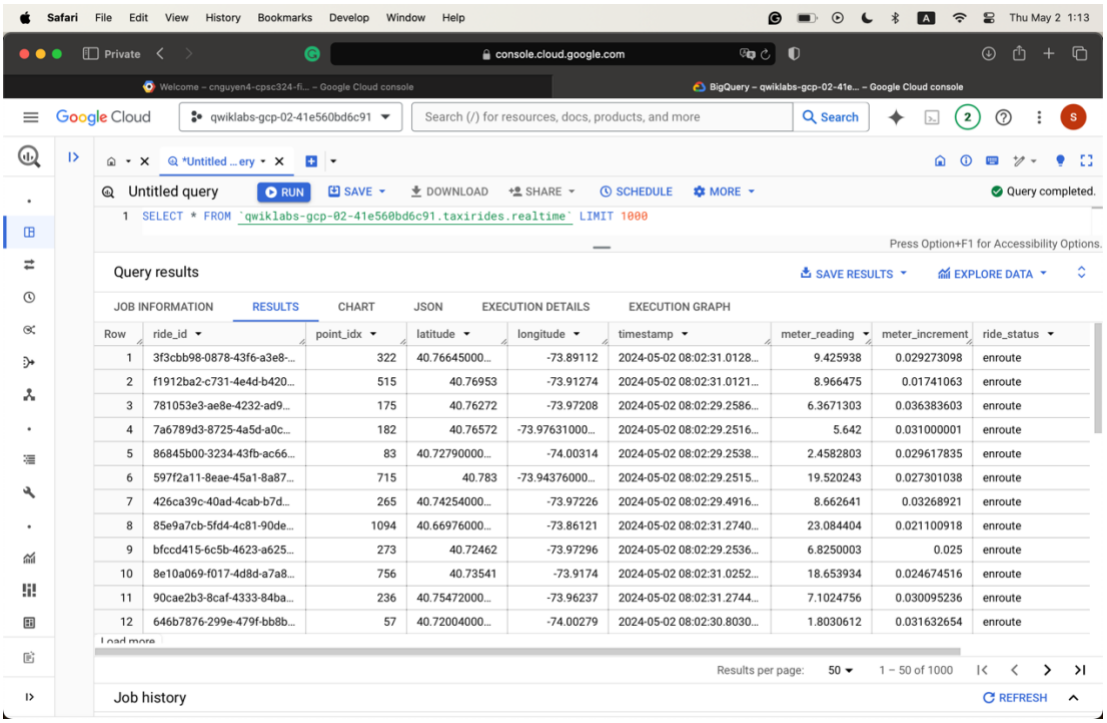
    --staging-location gs://qwiklabs-gcp-02-41e560bd6c91/temp \

    --parameters inputTopic=projects/pubsub-public-

    data/topics/taxirides-realtime,outputTableSpec=qwiklabs-gcp-02-

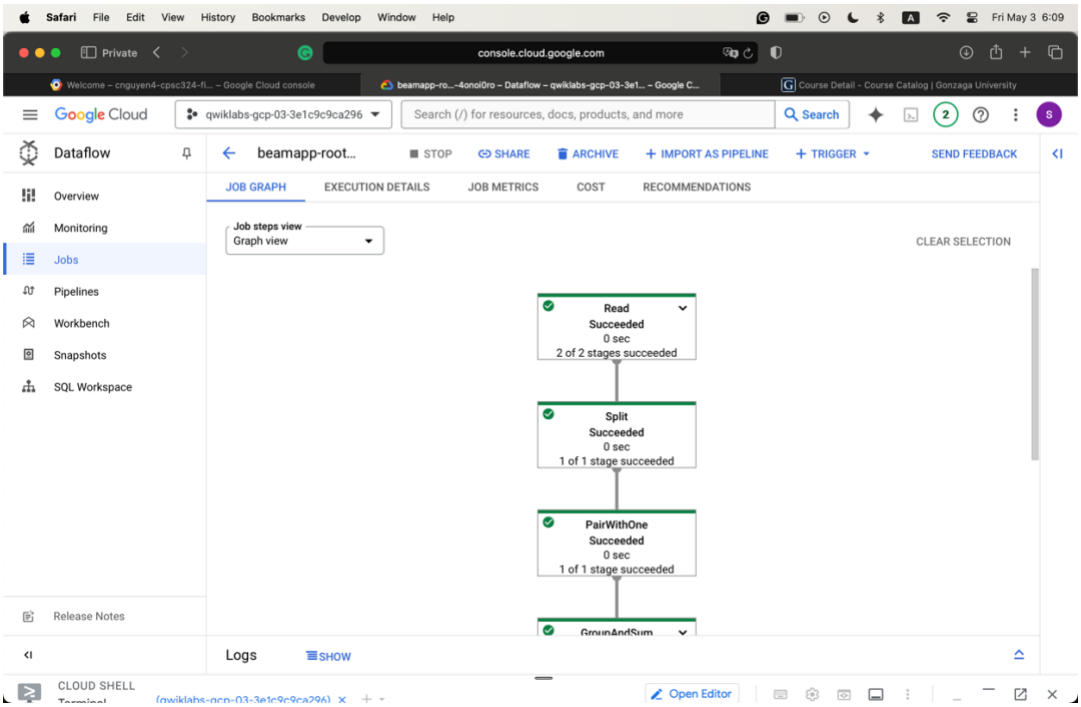
    41e560bd6c91:taxirides.realtime

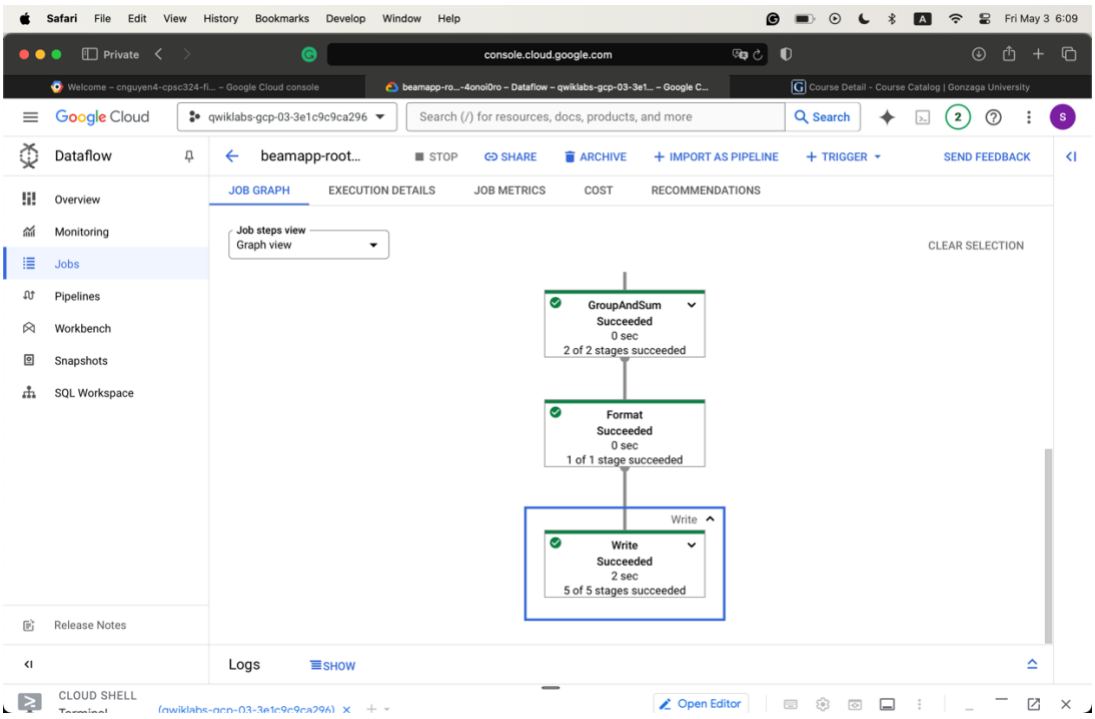
- Screenshot of query result

- 

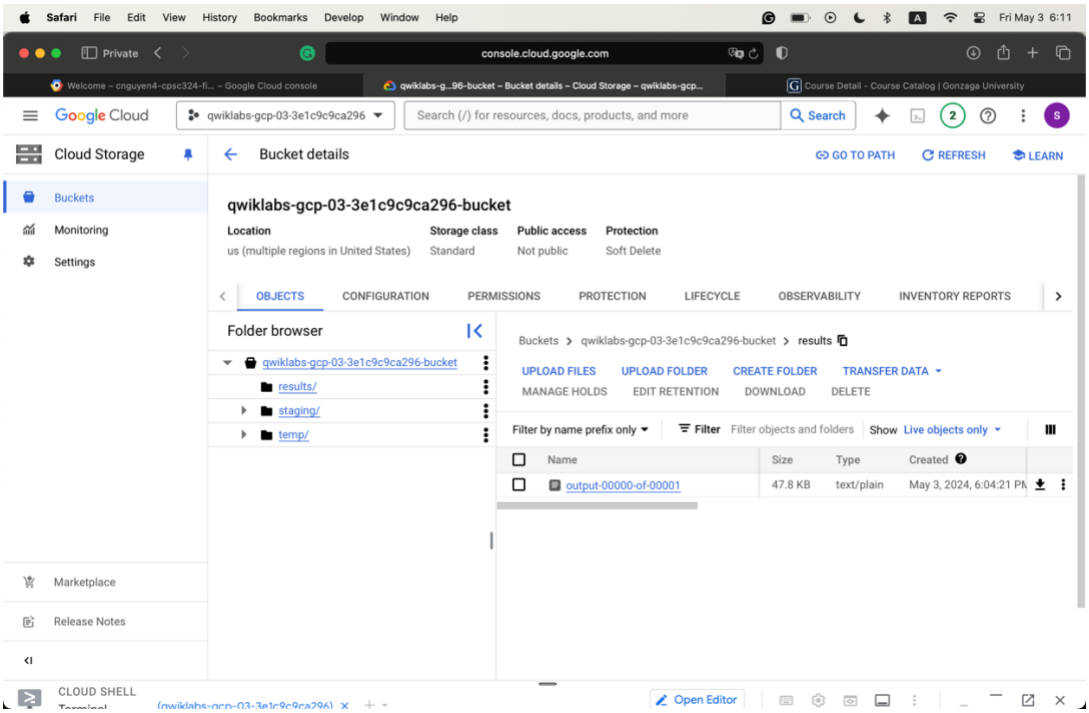
7. Question 7

- JOB_MESSAGE_DETAILED similar items
  - Fusing: Refers to combining multiple operations into a single operation to optimize pipeline execution.
  - Autoscaling: Involves adjusting the number of worker resources automatically based on the workload.
  - Expanding: Pertains to transforming user-defined pipelines into detailed execution graphs for processing.Diagram screenshot
-





- Directory screenshot

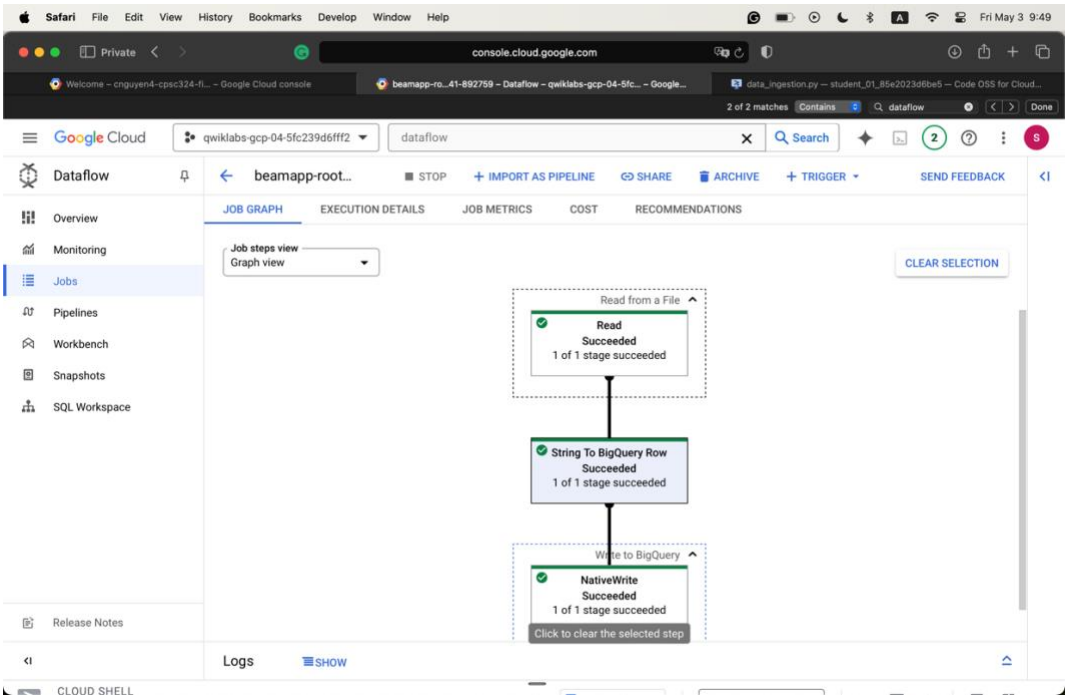


8. Question 8
    - Files copied to the bucket
        - usa_names.csv
        - head_usa_names.csv
        - They both have the same schemas
            - state
            - gender
            - year
            - name
            - number
            - created_date
    - Files 8
        - How is the pipeline p initialized?

        - p = beam.Pipeline(options=PipelineOptions(pipeline_args))
    - What is the first step of the pipeline? What does it do?
        - Read a line as the source of the pipeline. Skip the first line
        - | 'Read from a File' >> beam.io.ReadFromText(known_args.input,
          skip_header_lines=1)
    - What is the second step?
        - Translates from a CSV file single row input as a string to a dictionary object consumable by BigQuery
        - | 'String To BigQuery Row' >>
          beam.Map(lambda s: data_ingestion.parse_method(s))
    - What is the third step?
        - Write the parsed result to Big Query
        - | 'Write to BigQuery' >> beam.io.Write(
          beam.io.BigQuerySink(

          schema='state:STRING,gender:STRING,year:STRING,name:STRING,
          'number:STRING,created_date:STRING',
          create_disposition=beam.io.BigQueryDisposition.CREATE_IF_NEEDED,
          write_disposition=beam.io.BigQueryDisposition.WRITE_TRUNCATE)))

- Job graph in step 9



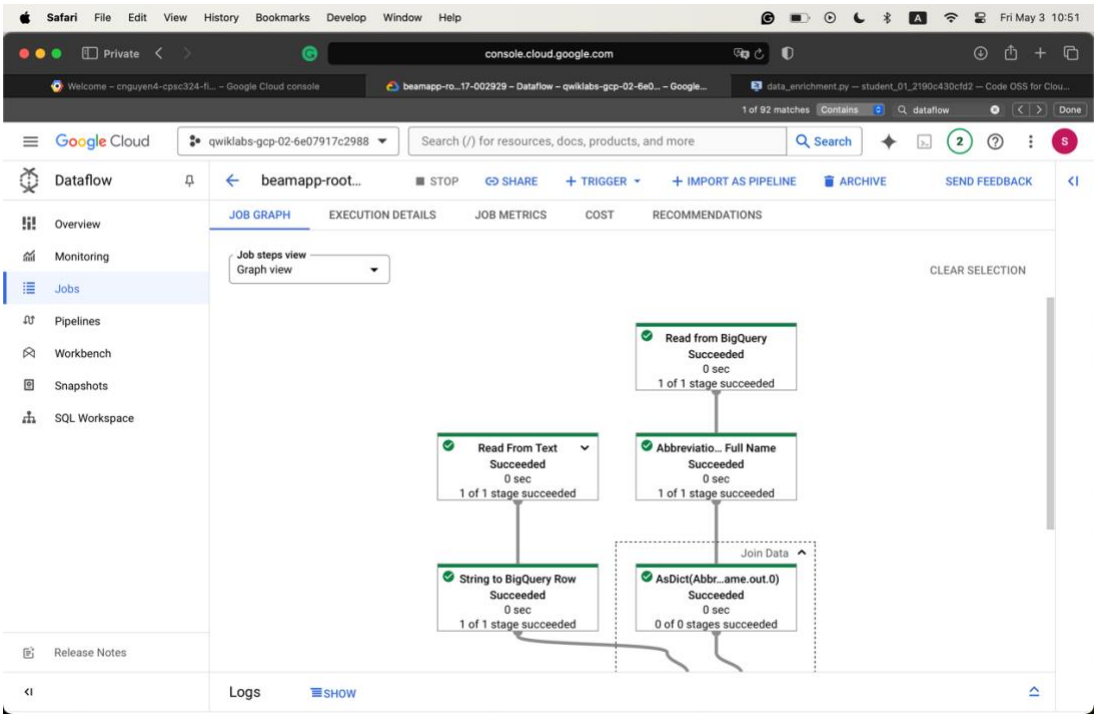
- Data pipeline comparison in Steps 9 and 11
    - In step 9, the pipeline converts the input into dictionary objects after taking input. However, in step 11, after converting into dictionary objects, the pipeline transforms the data that contains the year to a format understandable to Big Query.
- Steps for the pipeline in task 13
    - Initiate the command line
        - p = beam.Pipeline(options=PipelineOptions(pipeline_args))
        - schema = parse_table_schema_from_json(data_ingestion.schema_str)
    - Read the file
        - | 'Read From Text' >> beam.io.ReadFromText(known_args.input, skip_header_lines=1)
    - Translates the CSV input row into a dictionary object
        - | 'String to BigQuery Row' >> beam.Map(lambda s:

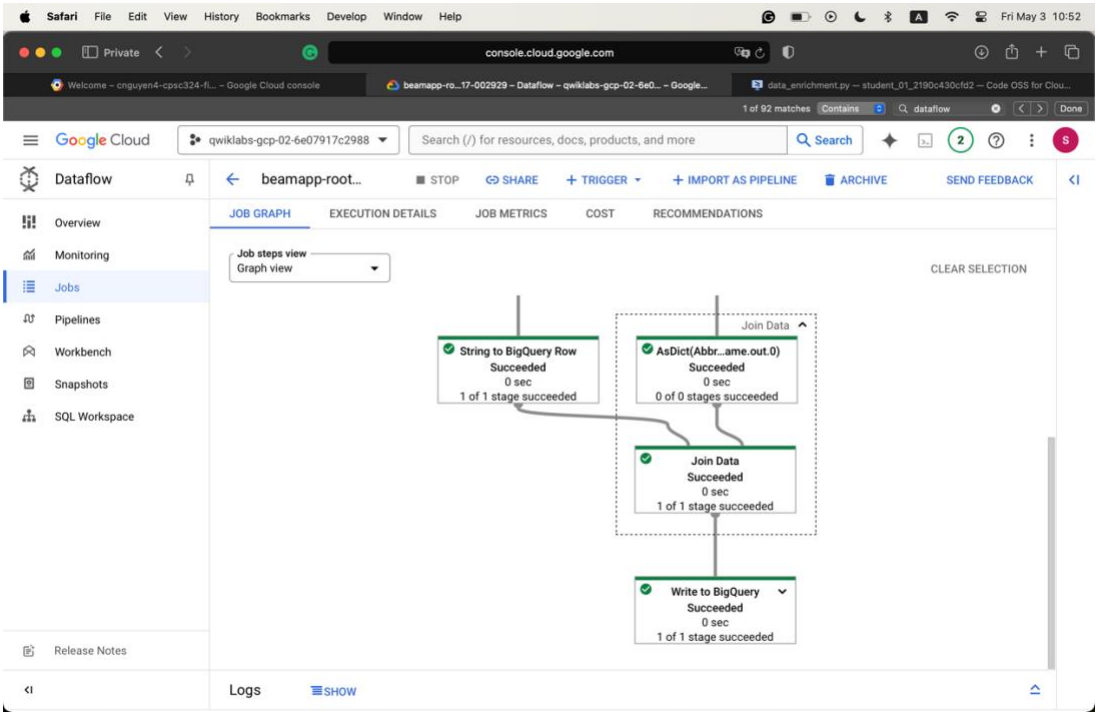
data_ingestion.parse_method(s))

- | 'Write to BigQuery' >> beam.io.Write(
  beam.io.BigQuerySink(
  known_args.output,
  schema=schema,
  create_disposition=beam.io.BigQueryDisposition.CREATE_IF_NEEDED,
  write_disposition=beam.io.BigQueryDisposition.WRITE_TRUNCATE)))

- Steps for the pipeline in task 13
  - Initiate the pipeline using the pipeline arguments
    - p = beam.Pipeline(options=PipelineOptions(pipeline_args))
    - schema = parse_table_schema_from_json(data_ingestion.schema_str)
  - Fetch in the second source of data (which is Big Query)
    - p | 'Read from BigQuery' >> beam.io.Read(
      beam.io.BigQuerySource(query=read_query,
      use_standard_sql=True))
      | 'Abbreviation to Full Name' >> beam.Map(
      lambda row: (row['state_abbreviation'], row['state_name'])))
  - Read the file
    - | 'Read From Text' >> beam.io.ReadFromText(known_args.input,
      skip_header_lines=1)
  - Translates from the raw string data in the CSV to a dictionary.

        - | 'String to BigQuery Row' >> beam.Map(lambda s:
          data_ingestion.parse_method(s))
    - Join the data
        - | 'Join Data' >> beam.Map(add_full_state_name, AsDict(
          state_abbreviations))
    - Writing to BigQuery
        - | 'Write to BigQuery' >> beam.io.Write(
          beam.io.BigQuerySink(
          known_args.output,
          schema=schema,
          create_disposition=beam.io.BigQueryDisposition.CREATE_IF_NEEDED,
          write_disposition=beam.io.BigQueryDisposition.WRITE_TRUNCATE)))
- Graph for task 14

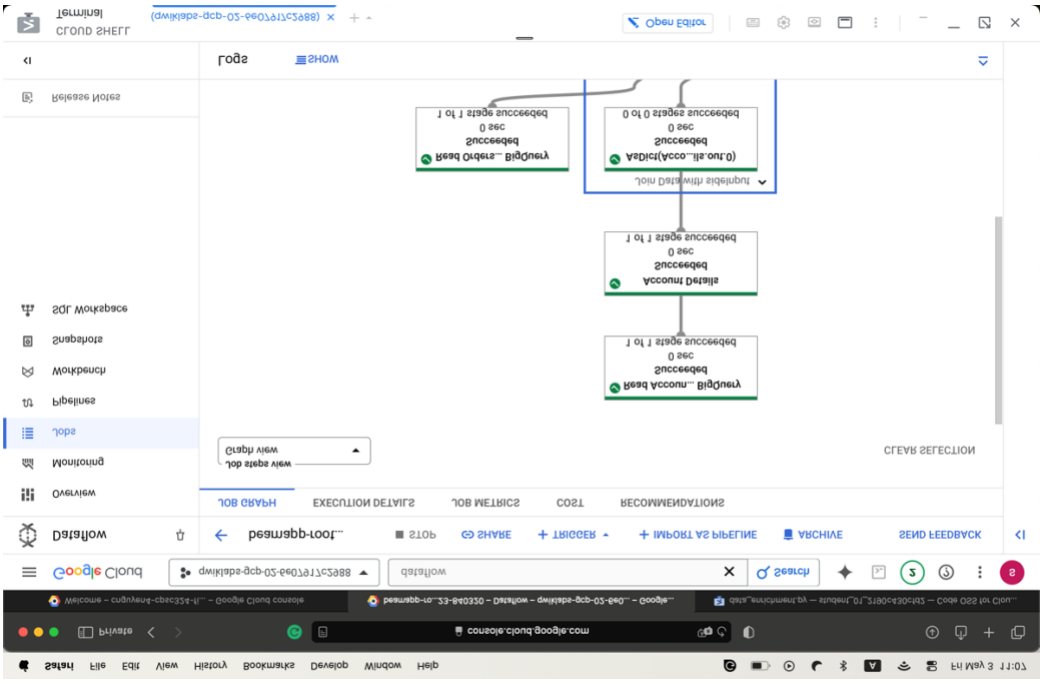


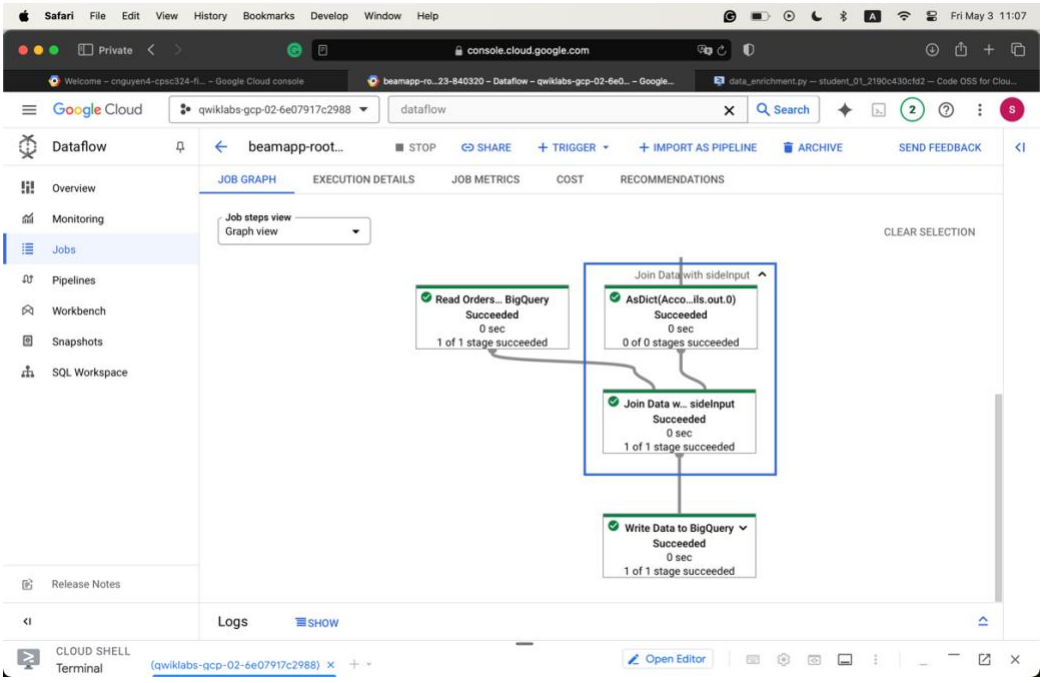


- Steps for the pipeline in task 15
  - Get data from two BigQuery sources
  - Join two data sources

- Filter out the header row
- Convert the lines read to dictionary objects
- Output the rows to BigQuery

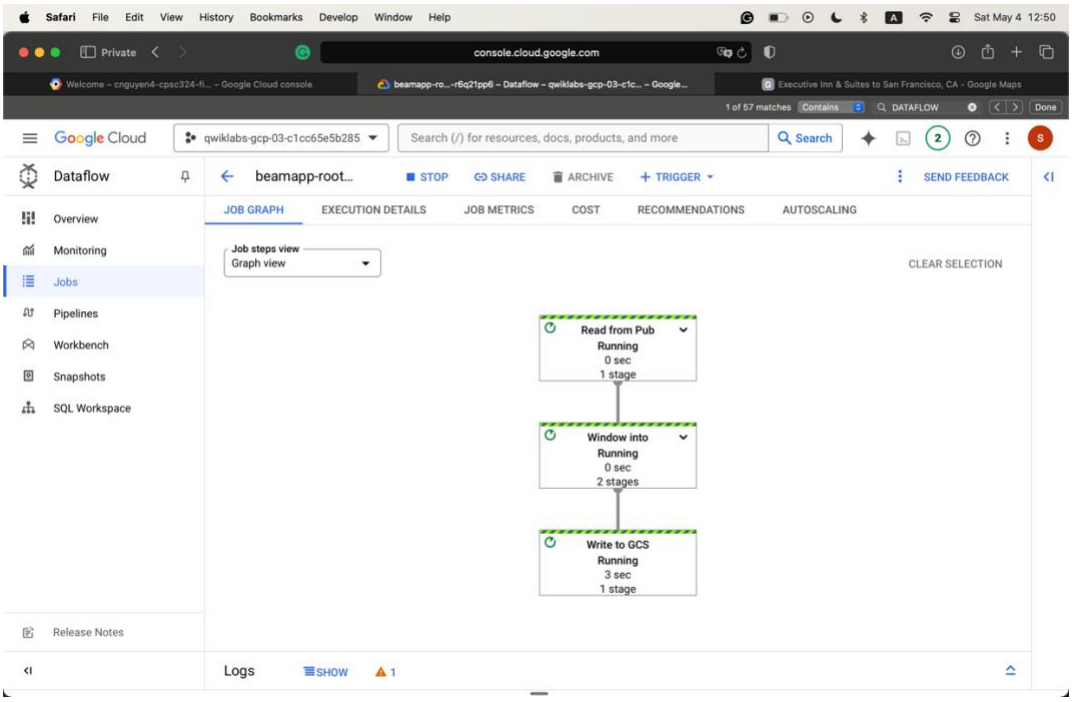
- Graph for task 16



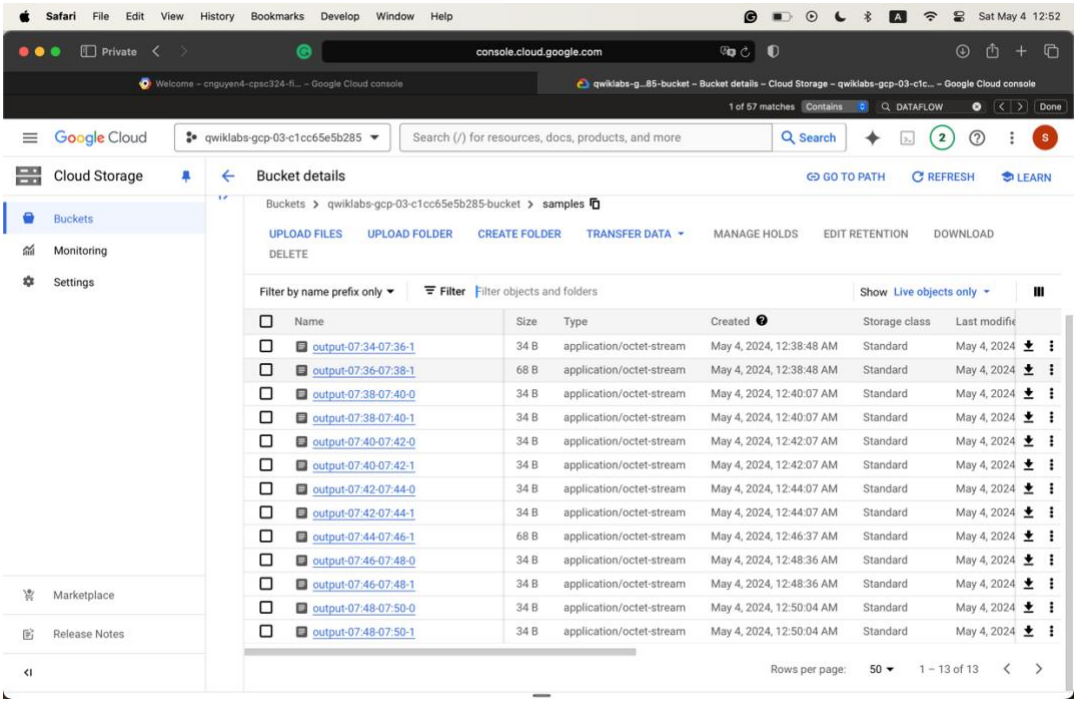


9. Question 9

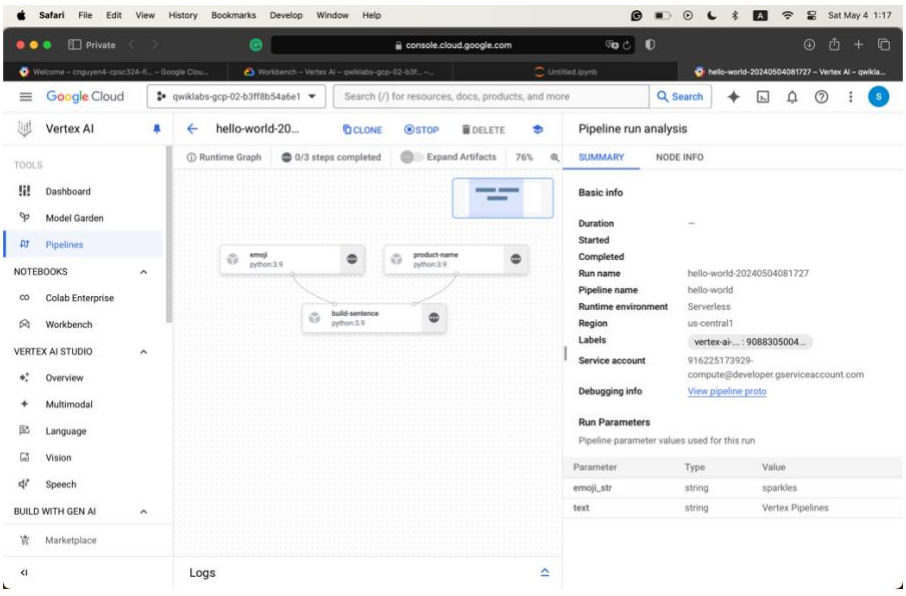
- Graph photo

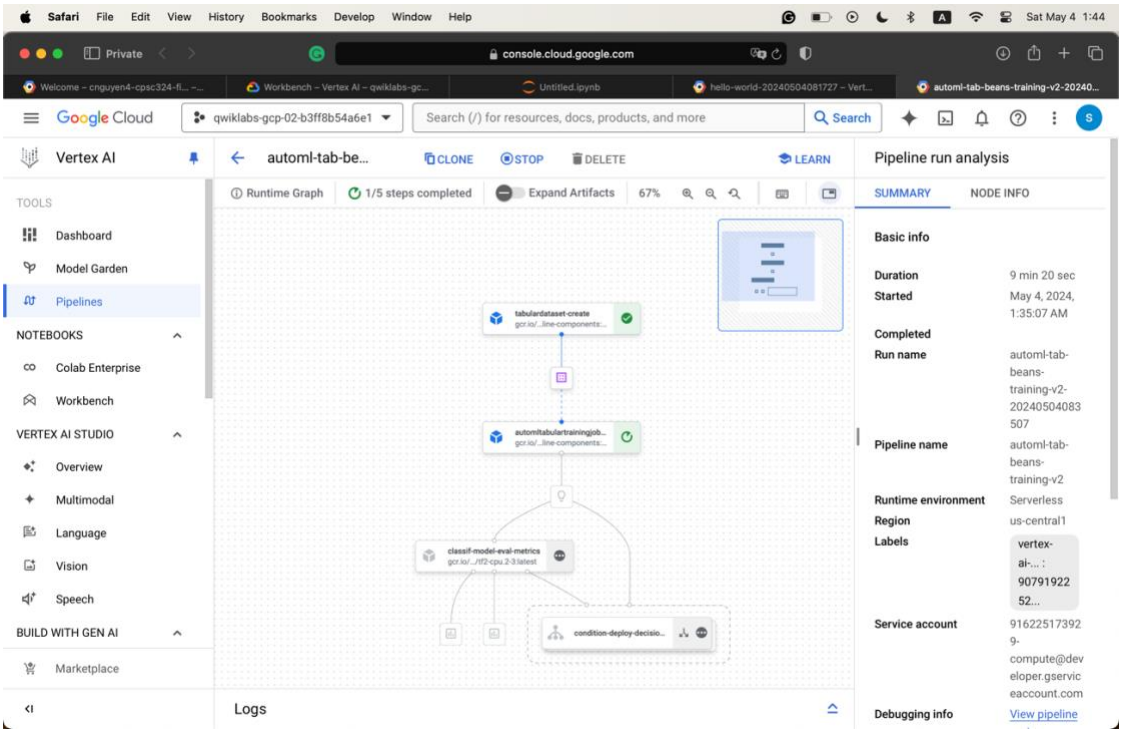


- Files generated in part 4



10. Question 10

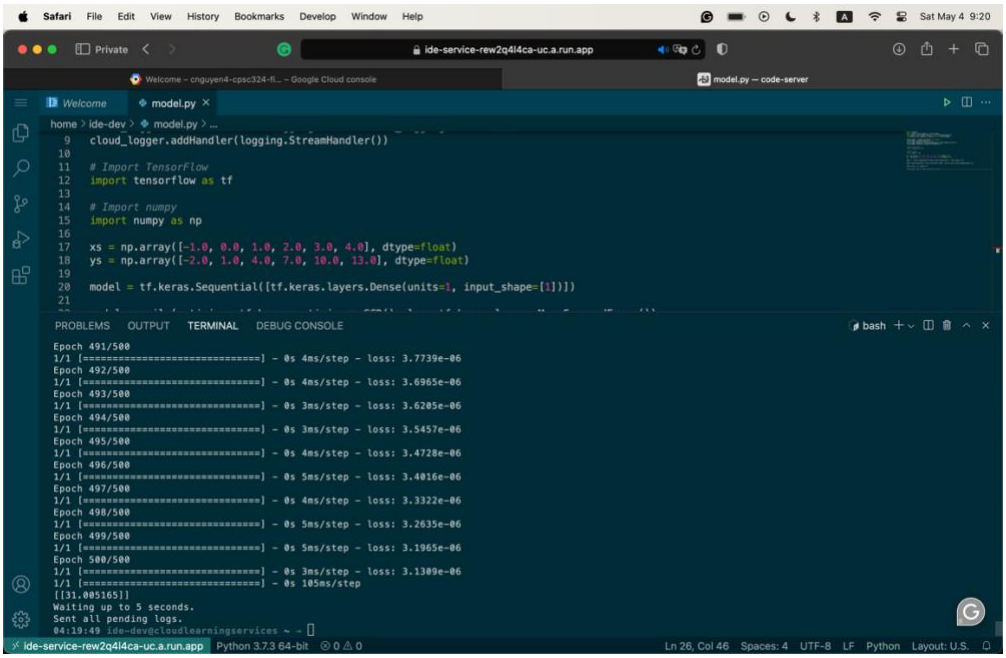
- Three pipeline components
    - product_name: take the product name as input and return that string as output
    - emoji: take the text description of an emoji and convert it to an emoji
    - build_sentence: consume the output of the previous two to build a sentence that uses the emoji
- The pipeline
    - To set up the pipeline, use the @dsl.pipeline decorator
    - Write the intro_pipeline function
        - product_task: takes a product name as input
        - emoji_task: takes the text code for an emoji as input
        - consume_task: to execute the task. Has 3 inputs
            - output of product_task
            - output of emoji_task
            - the output of the previous two
- Screenshot of the pipeline running
    - 

- Screenshot of the pipeline from task 4
  - 

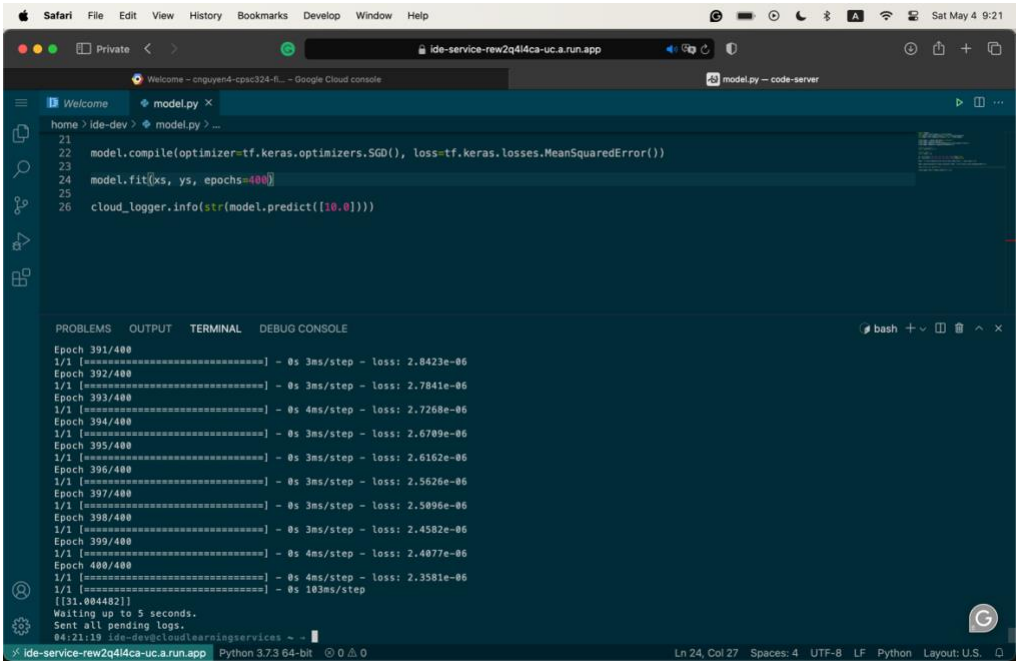
- The purpose of task 4
  - Custom evaluation: Evaluates model performance and decides whether to deploy.
  - Tabular Dataset Creation: Prepares data for training.
  - AutoML Tabular Training Job: Trains the model.
  - Model Deployment: Deploys the trained model.
  - Conditional Logic: Determines deployment based on evaluation metrics

11. Question 11

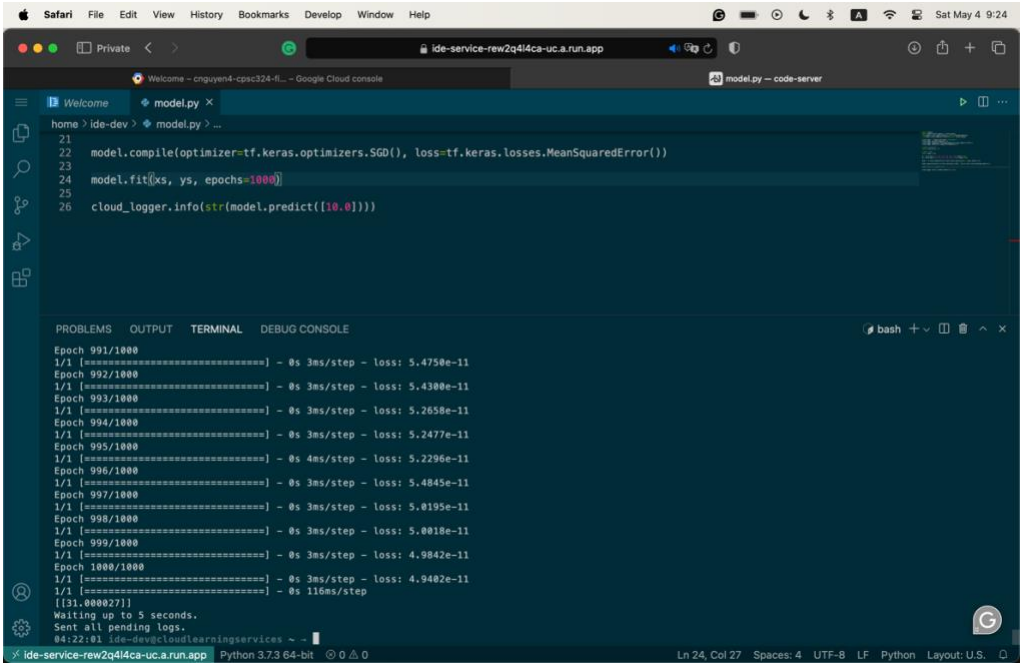
- Example screenshots
  - 500 epochs

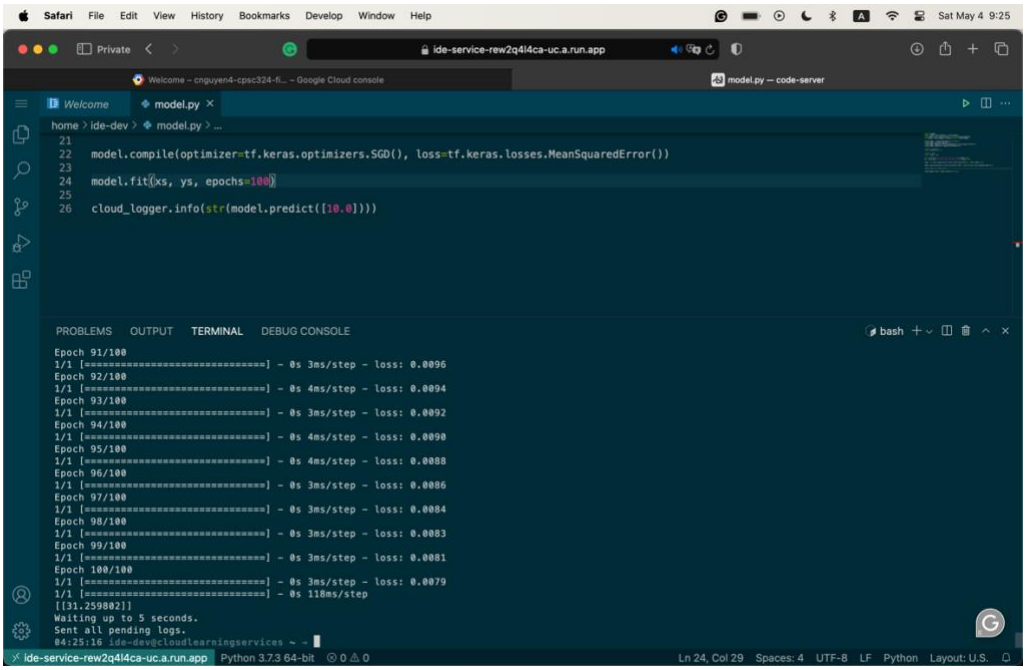
- 

  - 400 epochs

- 

  - 1000 epochs

- 

    - 100 epochs
- 

    - The lower the epoch number, the less accurate the result is

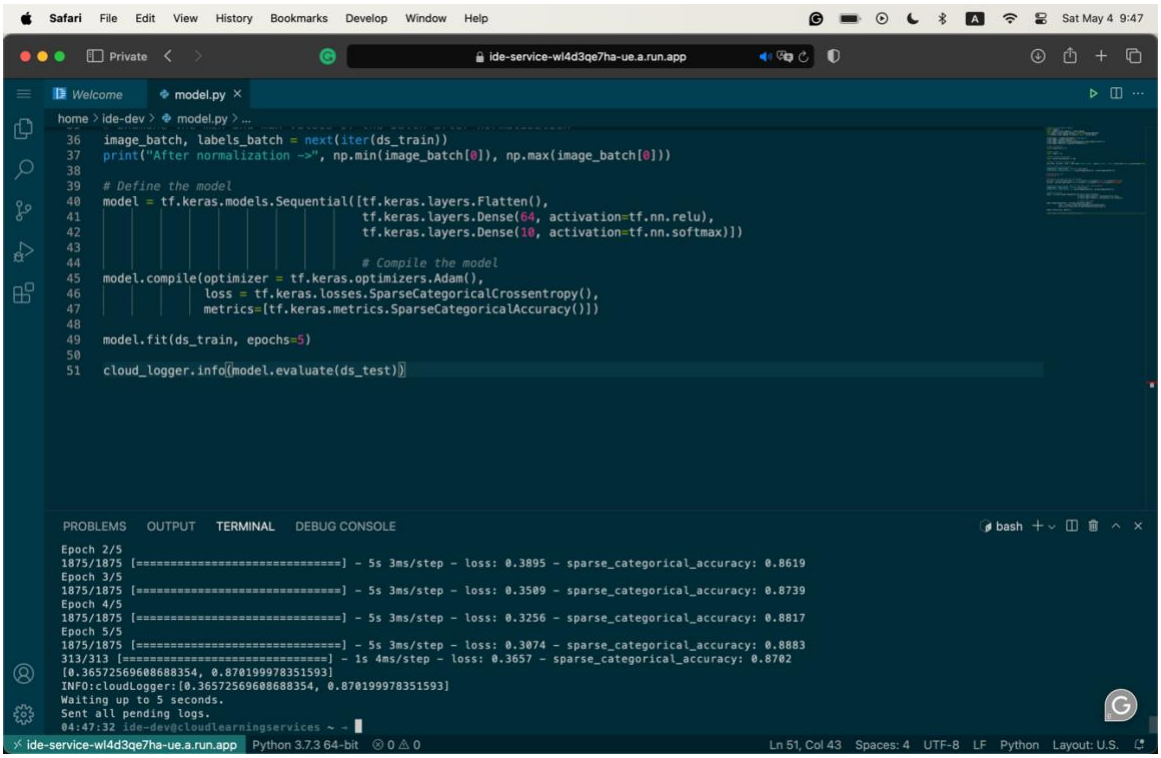
- Script
    - ```
      import logging
      ```
    - ```
      import google.cloud.logging as cloud_logging
      ```

```
from google.cloud.logging.handlers import CloudLoggingHandler
from google.cloud.logging_v2.handlers import setup_logging

cloud_logger = logging.getLogger('cloudLogger')
cloud_logger.setLevel(logging.INFO)
cloud_logger.addHandler(CloudLoggingHandler(cloud_logging.Client()))
cloud_logger.addHandler(logging.StreamHandler())

# Import TensorFlow
import tensorflow as tf

# Import numpy
import numpy as np

xs = np.array([-1.0, 0.0, 1.0, 2.0, 3.0, 4.0], dtype=float)
ys = np.array([-2.0, 1.0, 4.0, 7.0, 10.0, 13.0], dtype=float)

model = tf.keras.Sequential([tf.keras.layers.Dense(units=1, input_shape=[1])])

model.compile(optimizer=tf.keras.optimizers.SGD(), loss=tf.keras.losses.MeanSquaredError())

model.fit(xs, ys, epochs=100)

cloud_logger.info(str(model.predict([10.0])))
```

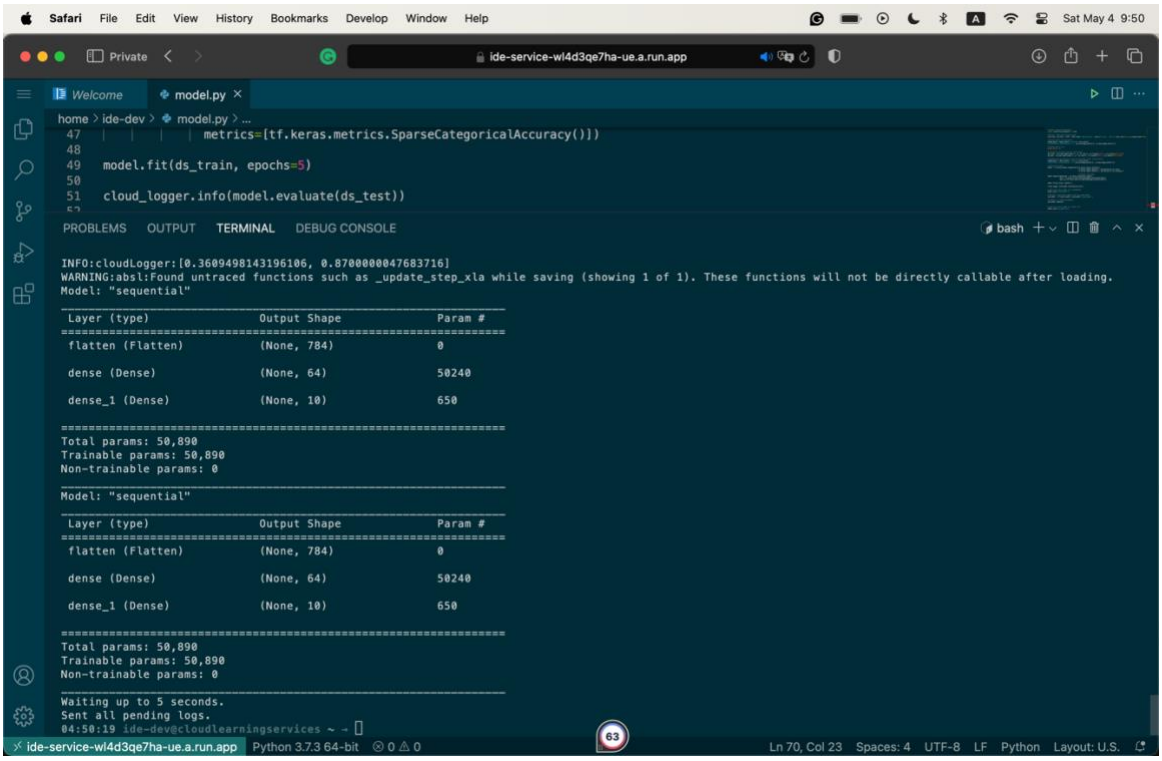
12. Question 12

- Describe the structure of the code
    - A sequence of task
    - Flatten the image from a square matrix shape to a 1-D vector
    - Dense: adds a layer of neurons
        - There are two calls on Dense
            - Relu: if X>0, return X, else 0. It passes 0 or greater to the next layer in the network

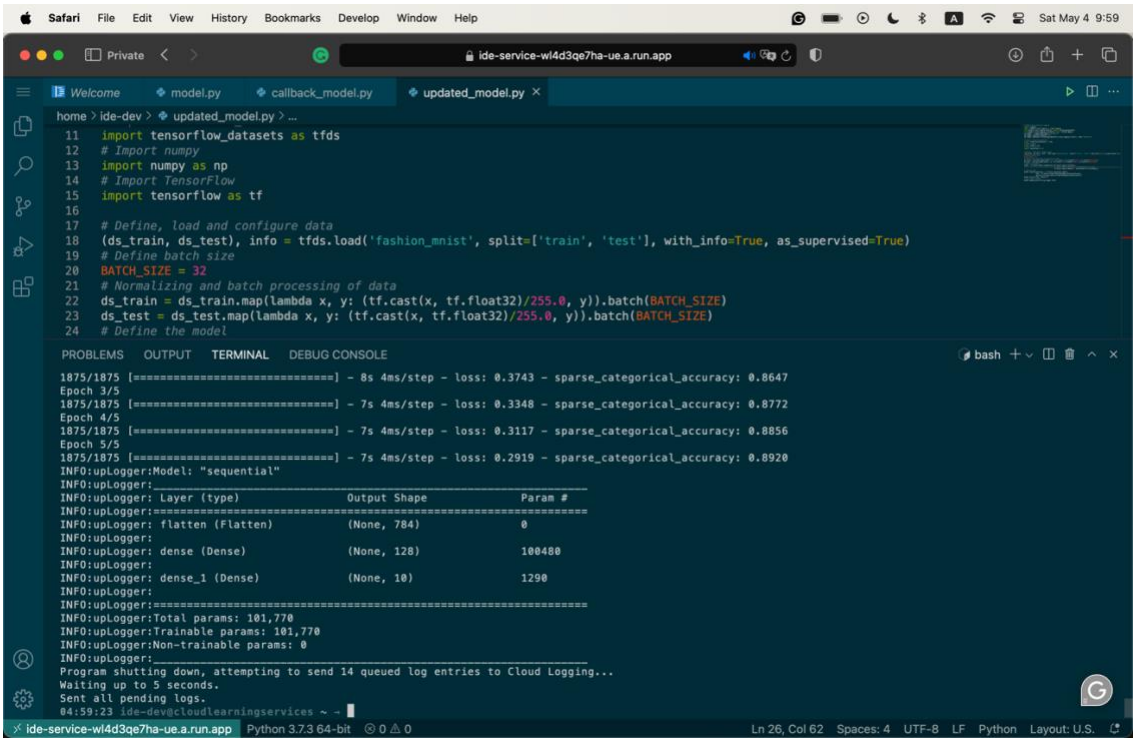
- Softmax: takes a set of values and effectively picks the biggest one so that we won't have to sort

- Evaluation at the end of step 4
  - 

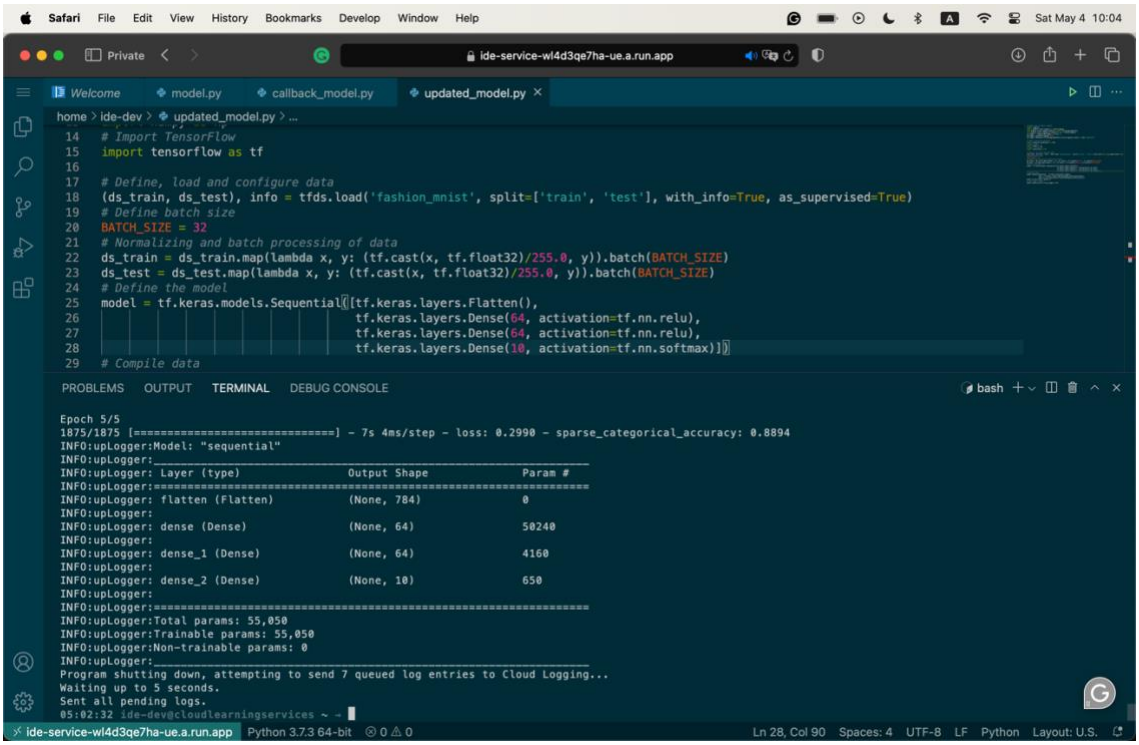
- Evalutation in step 7

-


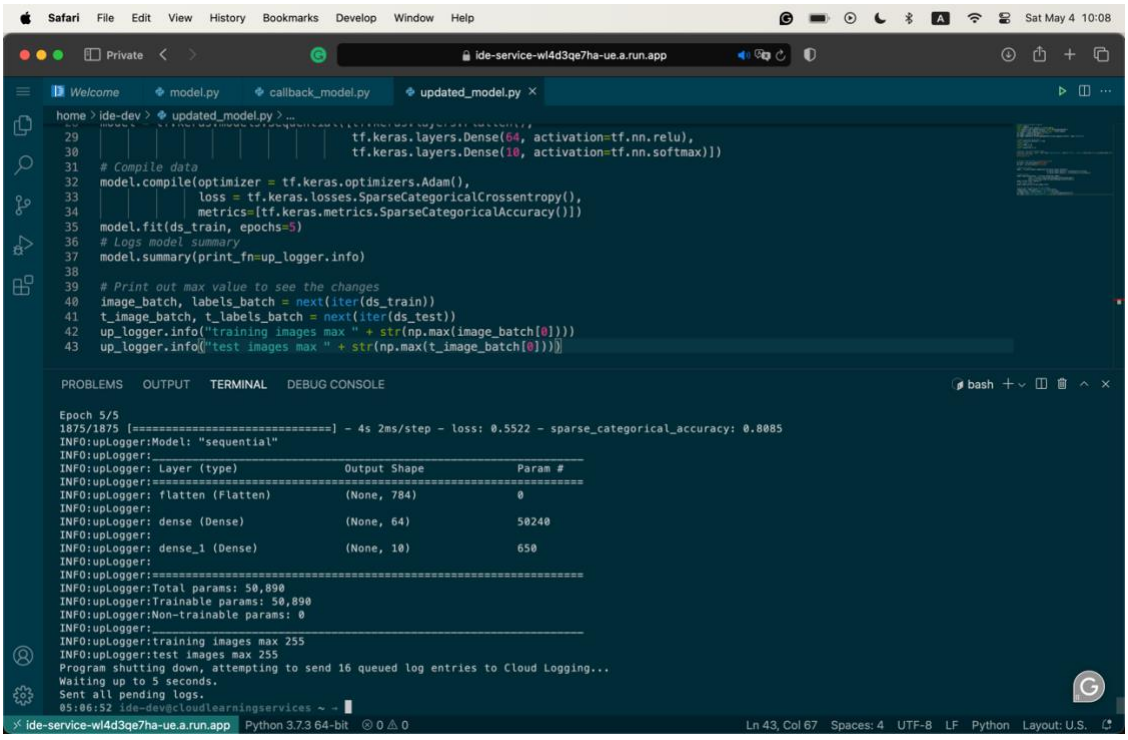
- Screenshot of the evaluation exercise 1

-


- Screenshot of the evaluation exercise 2

- 

- Screenshot of the evaluation exercise 3

- 

13. Question 13

- I choose to work with the Introduction to Convolutions with TensorFlow lab

- Purpose of the lab
    - This lab shows how to train an image classifier using a deep neural network.
    - It also teaches the fundamentals of convolutional neural network
    - It shows how convolutions operate on images to extract features, using filters to identify patterns and pooling to reduce dimensionality while preserving important features.
- Outcome of the lab
    - Understand the basic concept.
    - Understand how these operations affect image processing and utilize these techniques to build an image classifier.
- What did I learn?
    - The basics of convolutional neural network. This is, in fact, my first time working with neural networks despite having to talk about it so many times.

14. Question 14

- I choose to work with the Classify Images with TensorFlow Convolutional Neural Networks lab.
- Purpose of the lab
    - Show how to train and deploy a custom Convolutional Neural Network (CNN) for image classification using TensorFlow on Vertex AI.
- Outcome of the lab
    - Training a custom CNN model using the Vertex AI.
    - Deploying the trained model for online prediction.

    - Understanding how to manipulate TensorFlow datasets, build CNN models, and make predictions.
    - Deploying and making predictions with the model

- What did I learn
    - This lab allows me to learn many important things about this technology, such as how to use it, how to set it up, and more.
    - It also shows me techniques to manipulate and process the data, especially with vision computing.