Thrive Data Engineering Project

Introduction

This data engineering project seeks to evaluate the ELT process of a full ELT pipeline. In the project, some libraries and frameworks will be used to implement this in both simple and structural ways.

The result of this is to be able to fully understand, implement, and document the end-to-end process of a functioning ELT process at Thrive Career Wellness.

Approach

There are ways to address the problem statement and the project, but in this documentation, there are going to be two major approaches elaborated here.

Method 1: Simplified ELT/ETL

This method makes use of a simple process to execute the task. The details of this approach will be elaborated further. This section is a summary of the method or process. In this approach, we use the Jupyter Notebook to execute the entire ELT/ETL and schemas for the various tables. Technologies used in this method are Jupyter Notebook as the infrastructure, Terminal, Python, pandas, SQL, and SQLite.

Method 2: Advanced ELT/ETL

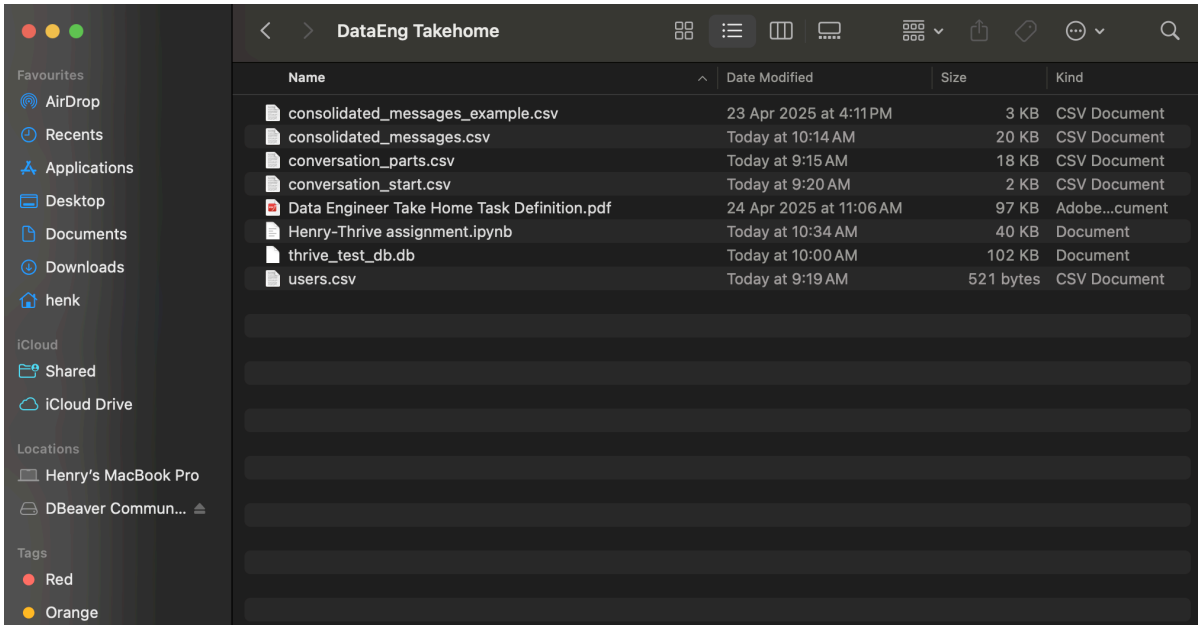
In this method, we implemented the real work scenario of a fully functional ELT structure with the data given and systems. This uses cloud serverless frameworks such as AWS VPC, S3, AWS Glue, Subnet, and Redshift. In this structure, we will also use DBT(DBT Core) for the data transformation aspect based on the Medallion architecture concept for data decentralization and accessibility. This also helps with shaping the ownership of data in the organization.
Technologies to be used in this method as Diagramsnet, AWS VPC, AWS S3, AWS Glue, Subnet, Redshift, DBeaver, Tableau, and DBT.

Method 1: Simplified ELT/ETL

This involves using the terminal. For this documentation, the device used to execute this is a MacBook Pro. This is going to be elaborated as steps in the documentation and explained further for better understanding.

Step 1: File path and folder location

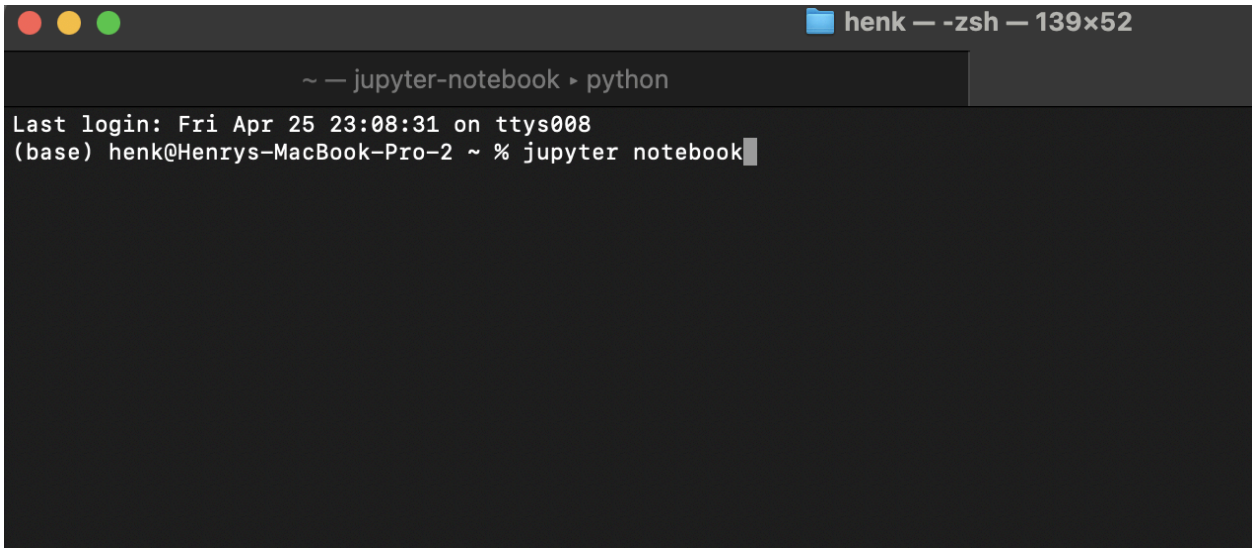
In this step, we need to have all the documents in one folder to enable the Jupyter notebook to have easy access and reduce the challenge with file path location. In my execution, I created a folder named ***DataEng Takehome***. This folder will contain the SQLite database, the CSV files, and the Jupyter Notebook file.



Step 2: Launching the Jupyter Notebook

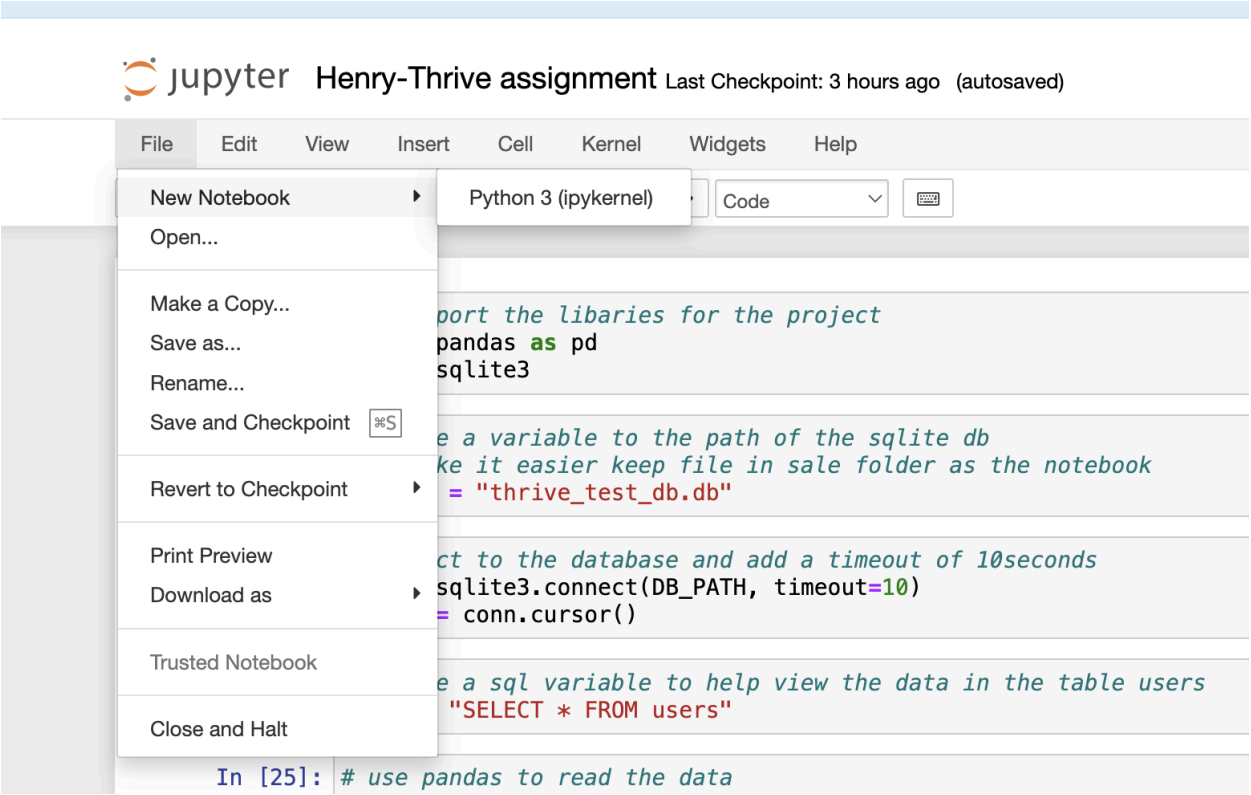
In this step, we need to install the notebook to be able to execute the project smoothly. To install the notebook if not exist,

- Open your terminal
- Use pip to install the notebook by typing ***pip install notebook***
- After installation, to run the notebook in your terminal, type ***jupyter notebook***



Step 3: Create a new notebook in the same folder

After the notebook is launched and running, create a new project and name it. In this project, my project is named Henry-Thrive assignment and saved in the same folder(***DataEng Takehome)*** created in step 1.



Step 4: Connect to the SQLite DB file

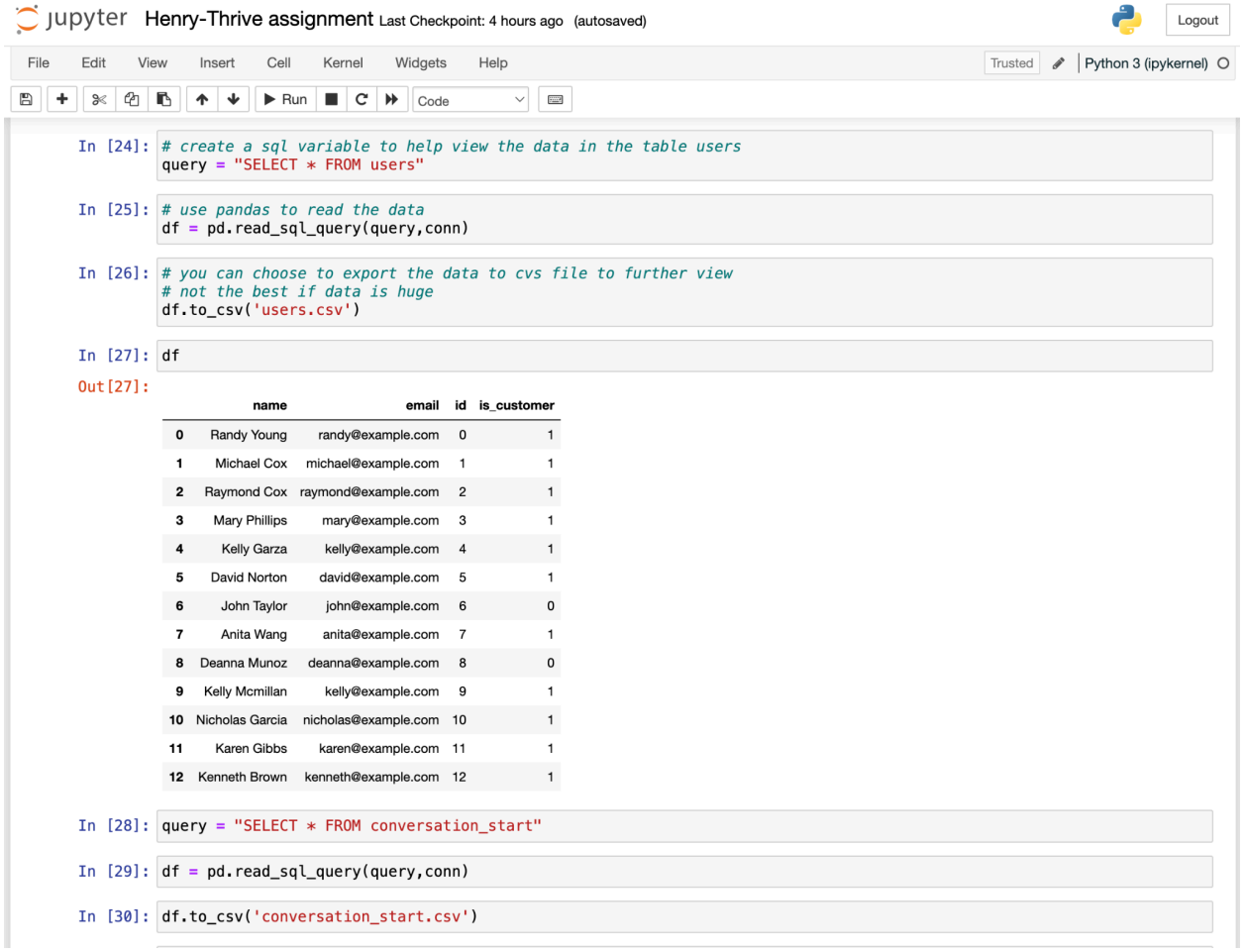
In this stage, load the Python libraries for SQLite and pandas. The SQLite library is used to connect to the SQLite database file and read the database tables.
Pandas to load the database tables and read the data in the database file. This would help to know the data in the tables and if the queries worked appropriately and accurately.

```
In [21]: # 1. import the libaries for the project
         import pandas as pd
         import sqlite3

In [22]: # create a variable to the path of the sqlite db
         # to make it easier keep file in sale folder as the notebook
         DB_PATH = "thrive_test_db.db"

In [23]: # connect to the database and add a timeout of 10seconds
         conn = sqlite3.connect(DB_PATH, timeout=10)
         cursor = conn.cursor()
```

The above image shows the loaded I import the Python library for SQLite and pandas, and connect to the database. This uses variables to connect to the database and set a database timeout to help manage access.

Step 5: Read the various tables using Pandas

Read the various tables in the database to have a better view and understanding of the tables and how they can relate. This will load and read the three tables namely user, conversation_start, and conversation_parts.



	name	email	id	is_customer
0	Randy Young	randy@example.com	0	1
1	Michael Cox	michael@example.com	1	1
2	Raymond Cox	raymond@example.com	2	1
3	Mary Phillips	mary@example.com	3	1
4	Kelly Garza	kelly@example.com	4	1
5	David Norton	david@example.com	5	1
6	John Taylor	john@example.com	6	0
7	Anita Wang	anita@example.com	7	1
8	Deanna Munoz	deanna@example.com	8	0
9	Kelly Mcmillan	kelly@example.com	9	1
10	Nicholas Garcia	nicholas@example.com	10	1
11	Karen Gibbs	karen@example.com	11	1
12	Kenneth Brown	kenneth@example.com	12	1

In the Jupyter notebook above, we use the pandas library to first convert the table into a dataframe and read the table to view the actual data and have a better insight into the content.

Creating Schemas

With reference to the project requirements, we are to create two dimensional schemas which will be used to create the fact table. The tables that will be created will have some foreign keys that will be referenced from the dimension tables. This will make use of SQL DDL (Data Definition Language) to create the tables with their column data types.

The dimensional tables to be created from the raw data are ***dim_user*** and ***dim_conversation_part***.
The fact table to be created is ***consolidated_messages,*** which would combine data from the dimension tables.

Step 1: Create the dim_user dimension table

Using the notebook, create a ***dim_user*** table with the SQL DDL command in the database file.

```
In [16]: # 2. Create Schemas
         # 2a. This create User Dimension table named dim_user
         cursor.execute("""
         CREATE TABLE IF NOT EXISTS dim_user (
             user_id      INTEGER PRIMARY KEY,
             email        VARCHAR,
             name         TEXT,
             is_customer  BOOLEAN
         )
         """)
         conn.commit()
```

The above SQL code will create the dimension table dim_user if it does not already exist in the ***"thrive_test_db.db"*** database. The columns of the table are based on the requirements and their data types. The below code will insert data from the users data into the dim_user dimension table.

```
In [20]: cursor.execute(
             "INSERT INTO dim_user (user_id, email, name, is_customer) "
             "SELECT id, email, name, is_customer FROM users;"
         )
         conn.commit()
```

```
In [21]: query = "SELECT * FROM dim_user"
```

```
In [22]: df = pd.read_sql_query(query,conn)
```

```
In [23]: df
```

Out[23]:

	user_id	email	name	is_customer
0	0	randy@example.com	Randy Young	1
1	1	michael@example.com	Michael Cox	1
2	2	raymond@example.com	Raymond Cox	1
3	3	mary@example.com	Mary Phillips	1
4	4	kelly@example.com	Kelly Garza	1
5	5	david@example.com	David Norton	1
6	6	john@example.com	John Taylor	0
7	7	anita@example.com	Anita Wang	1
8	8	deanna@example.com	Deanna Munoz	0
9	9	kelly@example.com	Kelly Mcmillan	1
10	10	nicholas@example.com	Nicholas Garcia	1
11	11	karen@example.com	Karen Gibbs	1
12	12	kenneth@example.com	Kenneth Brown	1

Step 2: Create the dim_conversation_part dimension table

Using the notebook, create the ***dim_conversation_part*** table in the database using the SQL command as shown below.

```
In [17]: # 2b. This create Conversation Parts Dimension table named dim_conversation_part
         cursor.execute("""
         CREATE TABLE IF NOT EXISTS dim_conversation_part (
             part_id            INTEGER PRIMARY KEY,
             conversation_id    INTEGER,
             conv_dataset_email VARCHAR,
             part_type          TEXT,
             message            TEXT,
             created_at         DATETIME
         )
         """)
         conn.commit()
```

The table data types help to accept the appropriate data into the columns in the specific table. The below code will insert data from the conversation_parts into the dim_conversation_part.

In [25]:
```
cursor.execute(
    "INSERT INTO dim_conversation_part (part_id, conversation_id, conv_dataset_email, part_type, message, created_at
    "SELECT id, conversation_id, conv_dataset_email, part_type, message, created_at FROM conversation_parts;"
)
conn.commit()
```

Step 3: Create the consolidated_messages fact table

Using the notebook, create the ***consolidated_messages*** fact table in the database using the SQL command as shown below.

In [18]:
```
# 2c. Create the Fact Table named Consolidated_messages table named consolidated_messages
cursor.execute("""
CREATE TABLE IF NOT EXISTS consolidated_messages (
    id              INTEGER,
    user_id         INTEGER,
    email           VARCHAR,
    conversation_id INTEGER,
    message         TEXT,
    message_type    TEXT,
    created_at      DATETIME,
    FOREIGN KEY(user_id) REFERENCES dim_user(user_id)
)
""")
conn.commit()
```

In this fact table, there is a foreign key reference to the source to ensure it gets the right data always.
The code below will insert data into the fact table using the SQL integration code written.

In [27]:
```
# 4b. Insert consolidated rows
cursor.execute("""
INSERT INTO consolidated_messages (id, user_id, email, conversation_id, message, message_type, created_at)
WITH
  start_msgs AS (
    SELECT
      id           AS msg_id,
      conv_dataset_email AS email,
      id           AS conversation_id,
      message,
      'open'       AS message_type,
      created_at
    FROM conversation_start
  ),
  part_msgs AS (
    SELECT
      part_id           AS msg_id,
      conv_dataset_email AS email,
      conversation_id,
      message,
      part_type    AS message_type,
      created_at
    FROM dim_conversation_part
  ),
  all_msgs AS (
    SELECT * FROM start_msgs
    UNION ALL
    SELECT * FROM part_msgs
  ),
  conv_customer AS (
    SELECT
      am.conversation_id,
      u.user_id AS user_id
    FROM all_msgs am
    JOIN dim_user u
      ON am.email = u.email
    WHERE u.is_customer = 1
    GROUP BY am.conversation_id, u.user_id
  )
SELECT
  am.msg_id      AS id,
  cc.user_id     AS user_id,
  am.email       AS email,
  am.conversation_id,
  am.message,
  am.message_type,
  am.created_at
FROM all_msgs am
JOIN conv_customer cc
  ON am.conversation_id = cc.conversation_id
ORDER BY
  am.conversation_id,
  am.created_at;
""")
conn.commit()
```

Step 4: Explanation of the integration of SQL code of the fact table

The code used to create the fact table will be explained from one CTE to another.

```
start_msgs AS (
   SELECT
        id          AS msg_id,
        conv_dataset_email AS email,
        id          AS conversation_id,
         message,
        'open'      AS message_type,
        created_at
   FROM conversation_start
 ),
```

The above CTE(common table expression) creates a table named ***start_msgs,*** which gets all the data about when a conversation started. Which gets details about the conversation, such as

- ***id,*** which is renamed as msg_id for the message id.
- ***conv_dataset_email*** as the email that started the conversation, and renamed as email.
- ***id*** this is repeated to represent the id of the entire conversation named as conversation_id.
- ***message:*** this gets the message of the initial conversation.
- ***open:*** this column is added based on the fact that this is where all conversations are assigned as opened and renamed as message_type.
- ***created_at*** this column gets the datetime when the conversation started from the conversation_start table.

```
part_msgs AS (
   SELECT
        part_id          AS msg_id,
        conv_dataset_email AS email,
        conversation_id,
        message,
        part_type    AS message_type,
        created_at
   FROM dim_conversation_part
 ),
```

The above CTE creates a table named ***part_msgs,*** which gets extra data about a conversation from the dim_conversation_part table. Which gets details about the conversation, such as

- ***part_id,*** which is renamed as msg_id for the message id.
- ***conv_dataset_email*** as the email that started the conversation, and renamed as email.
- ***conversation_id:*** this gets the conversation id.
- ***message*** this gets the message of the table.
- ***part_typ:e*** this gets message typ,e either comment, open, close, etc, renamed as message_type.
- ***created_at*** this column gets the datetime when the conversation started from the dimension table created *dim_conversation_part* table.

```
all_msgs AS (
   SELECT * FROM start_msgs
```

```
    UNION ALL
    SELECT * FROM part_msgs
  )
```

The above CTE creates a table named ***all_msgs,*** which gets all data by combining data from ***start_msgs*** *and* ***part_msgs***. To get all data, such as

- ***msg_id,*** which is renamed as msg_id for the message id.
- ***email*** as the email either author of the original conversation ot part.
- ***conversation_id:*** this gets the conversation id.
- ***message*** this gets the message of the table.
- ***message_type:*** this gets message type, either comment, open, close, etc, renamed as message_type.
- ***created_at*** this column gets.

```
conv_customer AS (
    SELECT
        am.conversation_id,
        u.user_id AS user_id
    FROM all_msgs am
    JOIN dim_user u
        ON am.email = u.email
    WHERE u.is_customer = 1
    GROUP BY am.conversation_id,u.user_id
  )
```

Based on the requirement of the project, we need to get some key data, which includes user_id, and filter out the data for only customers of Thrive Wellness. The above CTE creates a table named ***conv_customer,*** which gets all data from ***all_msgs*** *and* ***dim_user***. To get all data, such as

- ***conversation_id,*** which is renamed as msg_id for the message id.
- ***user_id*** as the email of either the author of the original conversation or part.

To get the above information, the only obvious way to join the all_msgs table to the dim_user table was using the email.

To get only customers, we need to add a WHERE clause to filter for just customers and then GROUP BY conversation_id in ascending order.

```
SELECT
        am.msg_id      AS id,
        cc.user_id     AS user_id,
        am.email       AS email,
        am.conversation_id,
        am.message,
        am.message_type,
        am.created_at
FROM all_msgs am
JOIN conv_customer cc
        ON am.conversation_id = cc.conversation_id
ORDER BY
        am.conversation_id,
        am.created_at;
```

This is the final CTE to get the complete data for the integration SQL table. The table gets its data from ***all_msgs*** and ***conv_customer*** CTEs. To final integration table contains

- ***msg_id,*** which is renamed as id.
- ***user_id*** as the user_id.
- ***email*** as the email of either the author of the original conversation or the conversation part email.
- ***conversation_id*** this is the id of the entire conversation.
- ***mesaage*** this is the message in the conversation.
- ***message_type:*** this indicates some details if the message is open, closed, etc.
- ***created_at*** when the message was created

To get the above information, the only obvious way to join the all_msgs table to the conv_customer table was using the conversation_id.
Based on the requirement of the project, the data is ordered by conversation_id and created_at.

Finalization and GitHub Repo

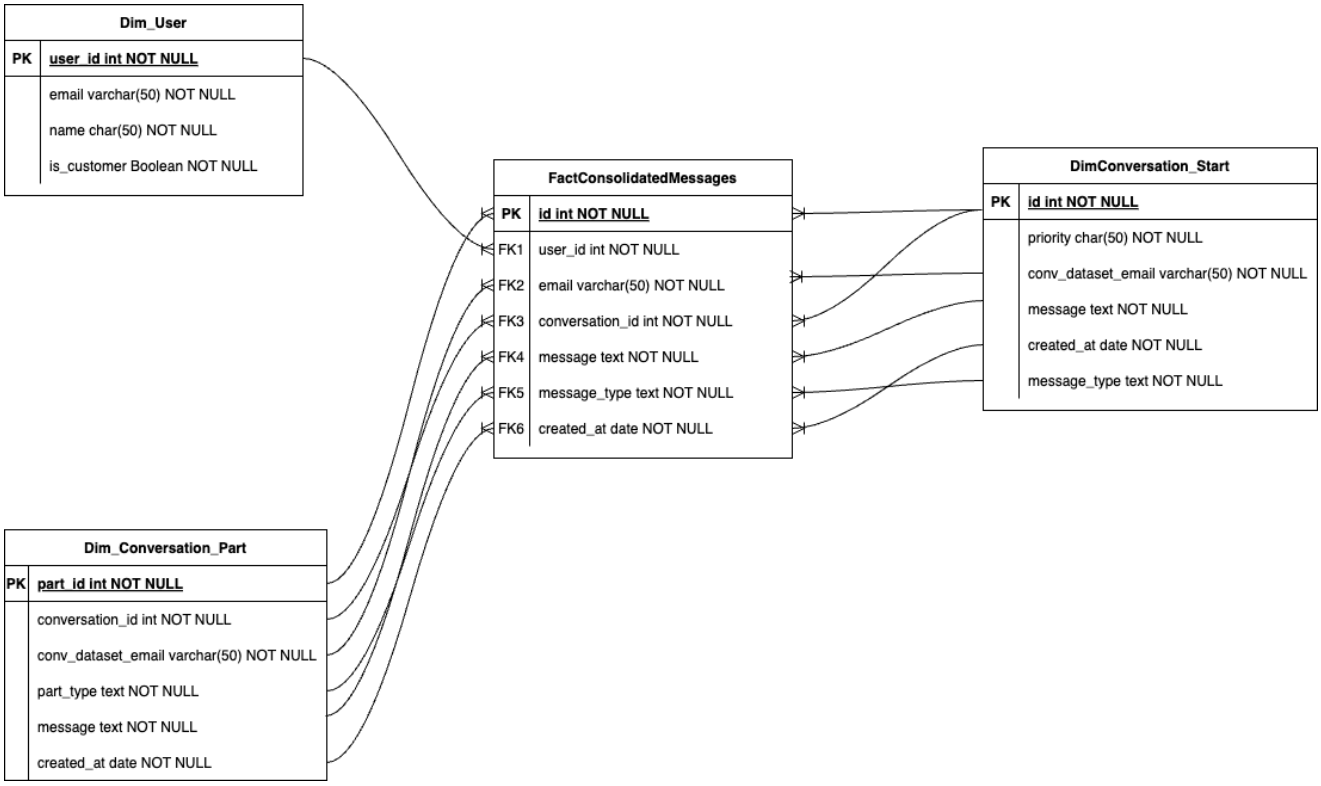
This method used the Jupyter Notebook to execute the task, putting all the considerations into implementation. The notebook and the documentation will help provide insight. It could be realized that the structure of the project is in the star schema method for data storage architecture.
GitHub:

Method 2: Advanced ELT/ETL

In this method, we would implement the ELT process using AWS serverless service and the cloud to process the data following the medallion architecture. This would involve more advanced concepts and processes, which is more of a real job project and a structured way of implementing ELT.

Logical Views Mapping

With reference to the project requirements, we would create the logical view, which would give us a better perspective of the structure of the various tables and how they feed into the fact table. The diagram would show the relationship and how the implementation would work. Below is the diagram of the logical view.

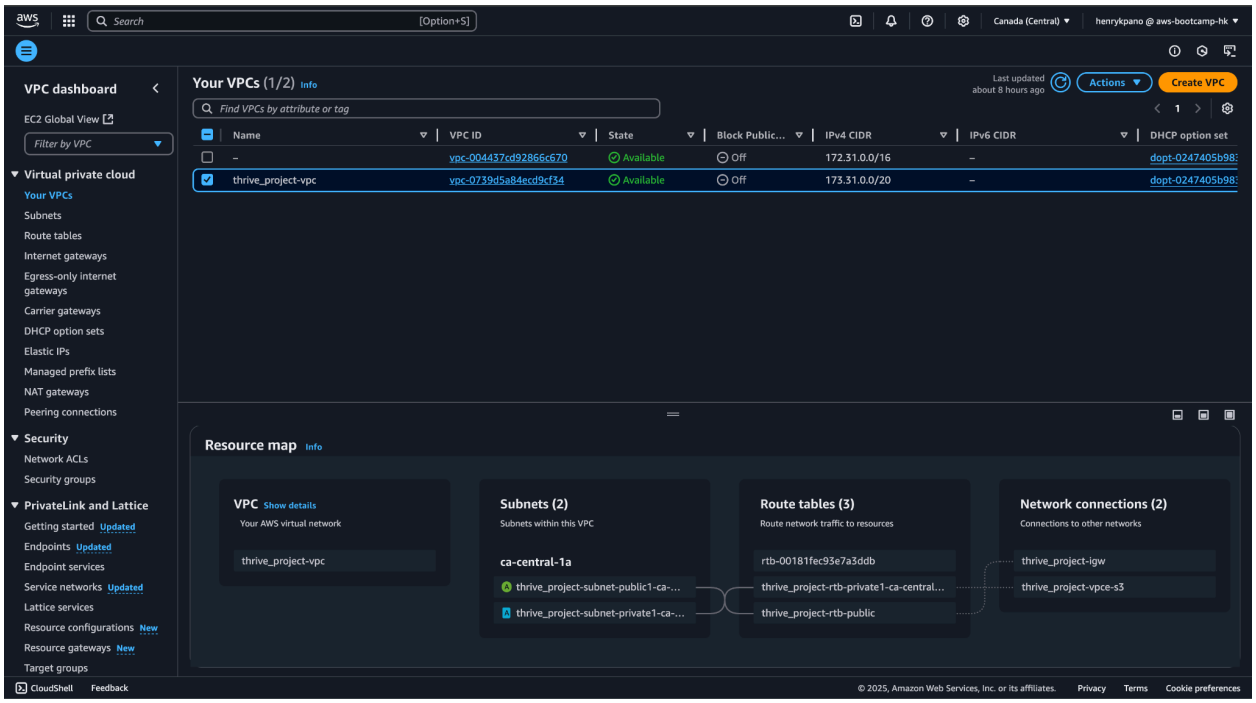


Create AWS Cloud Infrastructure

With reference to the project requirements, we would need to create a VPC (Virtual Private Cloud) on AWS to be able to control and have our cloud for our resources.

Step 1: Create an AWS VPC

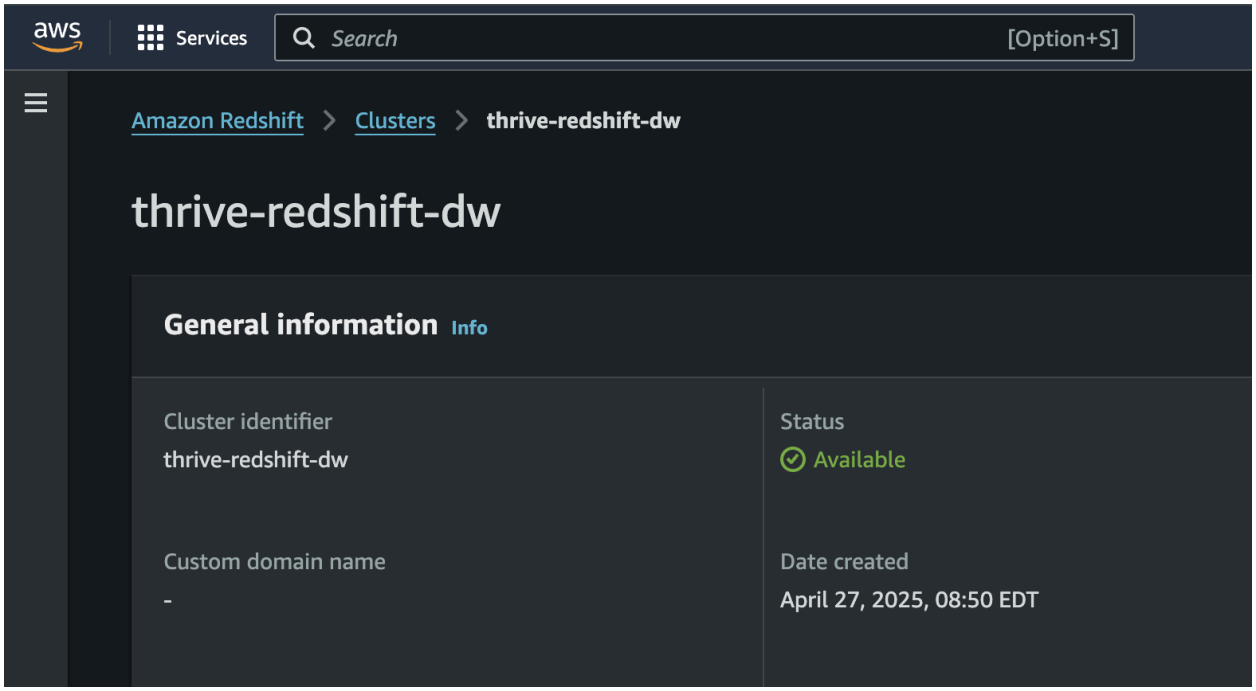
This is where we create our VPC to be able to manage the resources and communicate with the managed services. The VPC for this project is named ***thrive_project-vpc***. The creation automatically adds the various subnets needed for the VPC. In this VPC, there are 2 subnets created. The subnets are created in one AZ (availability zone), which is ***ca-central-1a***.



After the VPC is created, we add a Redshift security group to the VPC to enable us to access Redshift because it's a global service.

Step 2: Create Redshift Cluster

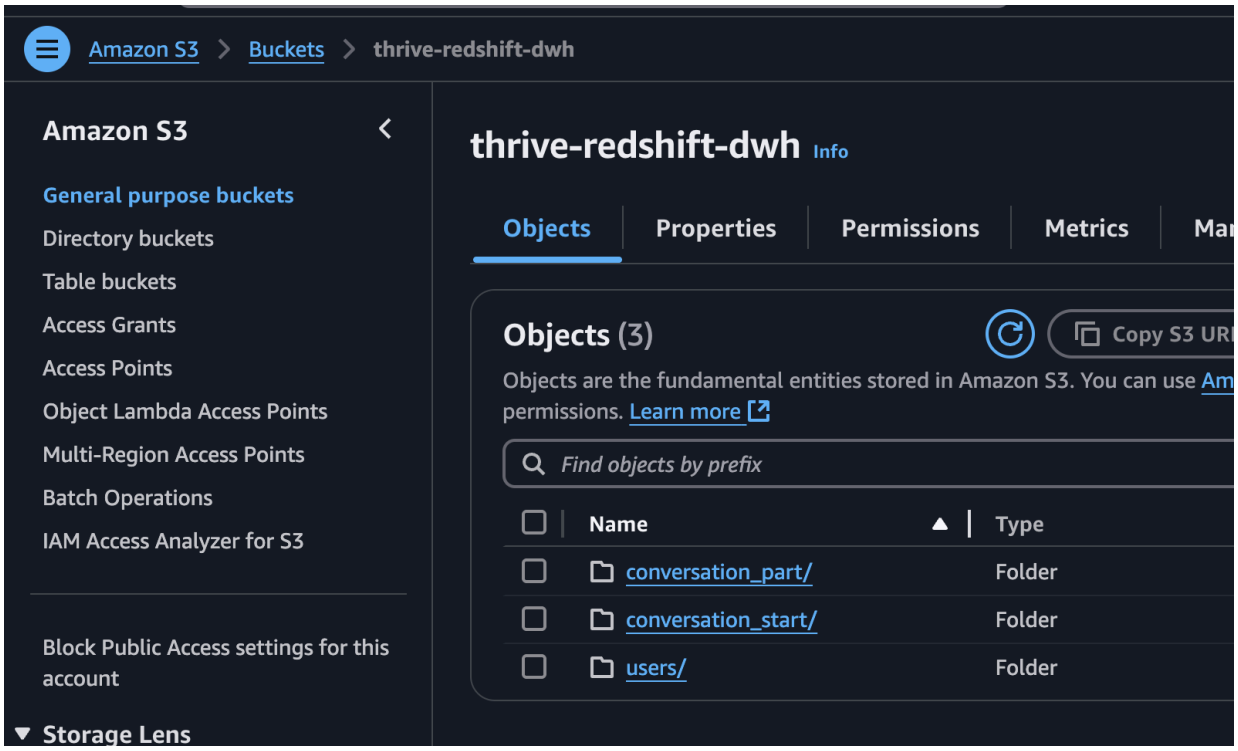
This is the data warehouse for the project. Due to the medallion architecture of the project, we need to create a Redshift cluster to be able to store the data and help with the implementation of the transformations with DBT. In this project, we will name the cluster database as ***thrivedev.***



Attach the appropriate VPC and the security group as needed.

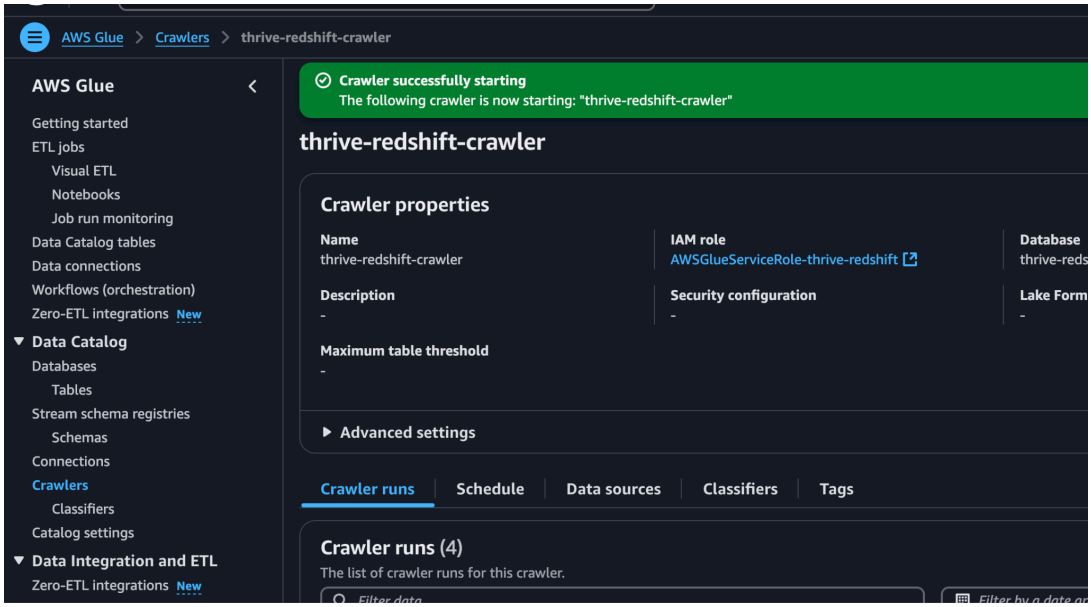
Step 3: Create an S3 Bucket

This is the data lake where the raw data from Thrive will be stored. There will be a Glue crawler to get the data from this bucket and store it in the Redshift database. For this project, we named our bucket as ***thrive-redshift-dwh*** and created folders for each CSV file.



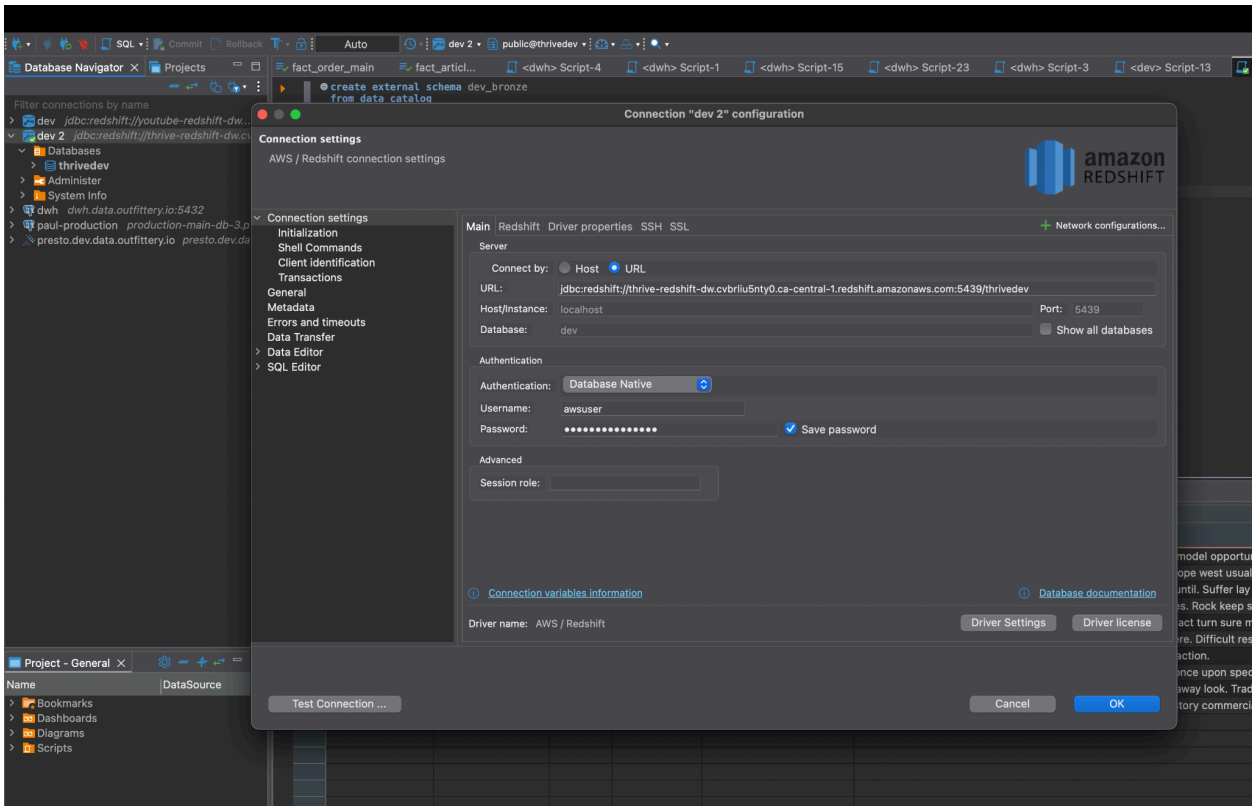
Step 3: Create an AWS Glue crawler

The crawler is created to be able to scan any S3 folder in the specified bucket. Create folders in the folder for each CSV file for Thrive. The delimiter for the crawler is a comma, and it's set to scan on demand. Add the IAM role to access Redshift to be able to send data.

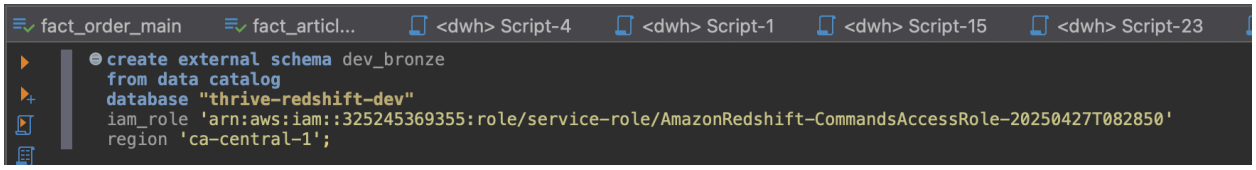


Step 4: Link Redshift on DBeaver

This is where we use a third-party database tool to access our Redshift database to be able to query the Redshift database ***thrivedev***. Use the JDBC access login and the password created to access the database. Ensure to enable SSL access in the DBeaver settings.



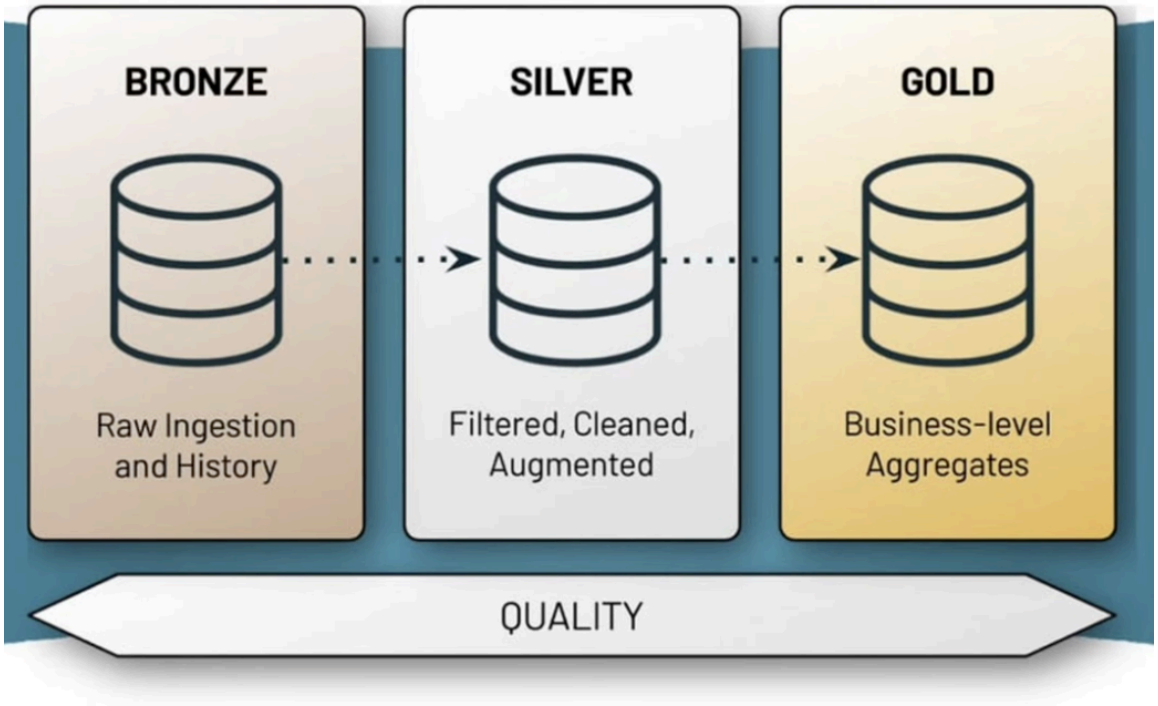
Test the connection to ensure it has successfully been set. This will give access to the Redshift cluster. Set up the bronze medallion as the raw data after initiating the crawler to get the various CSV files.



The above code connects Redshift to the Glue data catalog and stores the data in the schema named **dev_bronze**, which is the first structure in the medallion architecture framework.

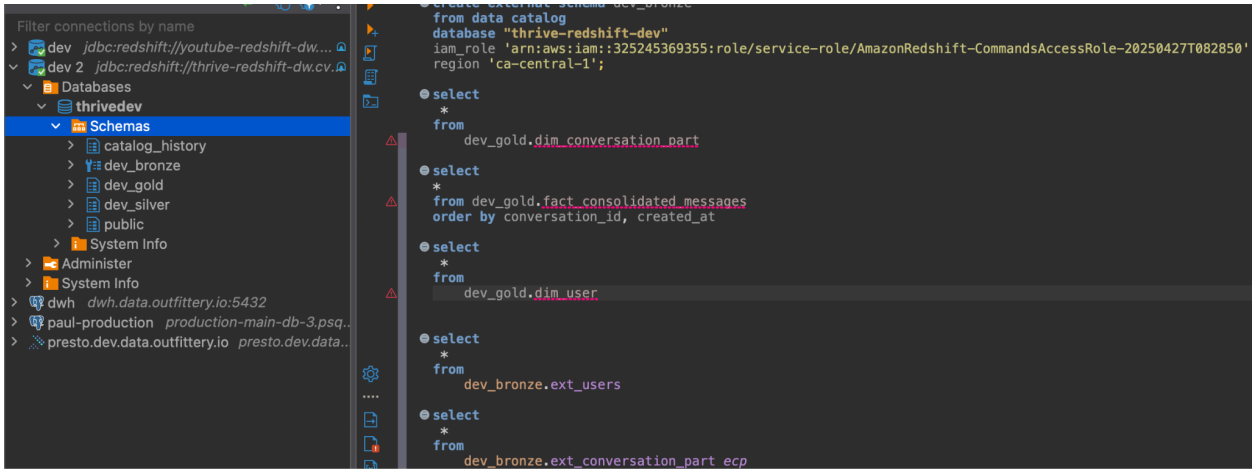
Setup DBT Core on a virtual env on a laptop

To create your virtual environment, this article would be of help to do so. This would help us complete the entire medallion structure to be able to execute the project smoothly.



Build the DBT Bronze Layer

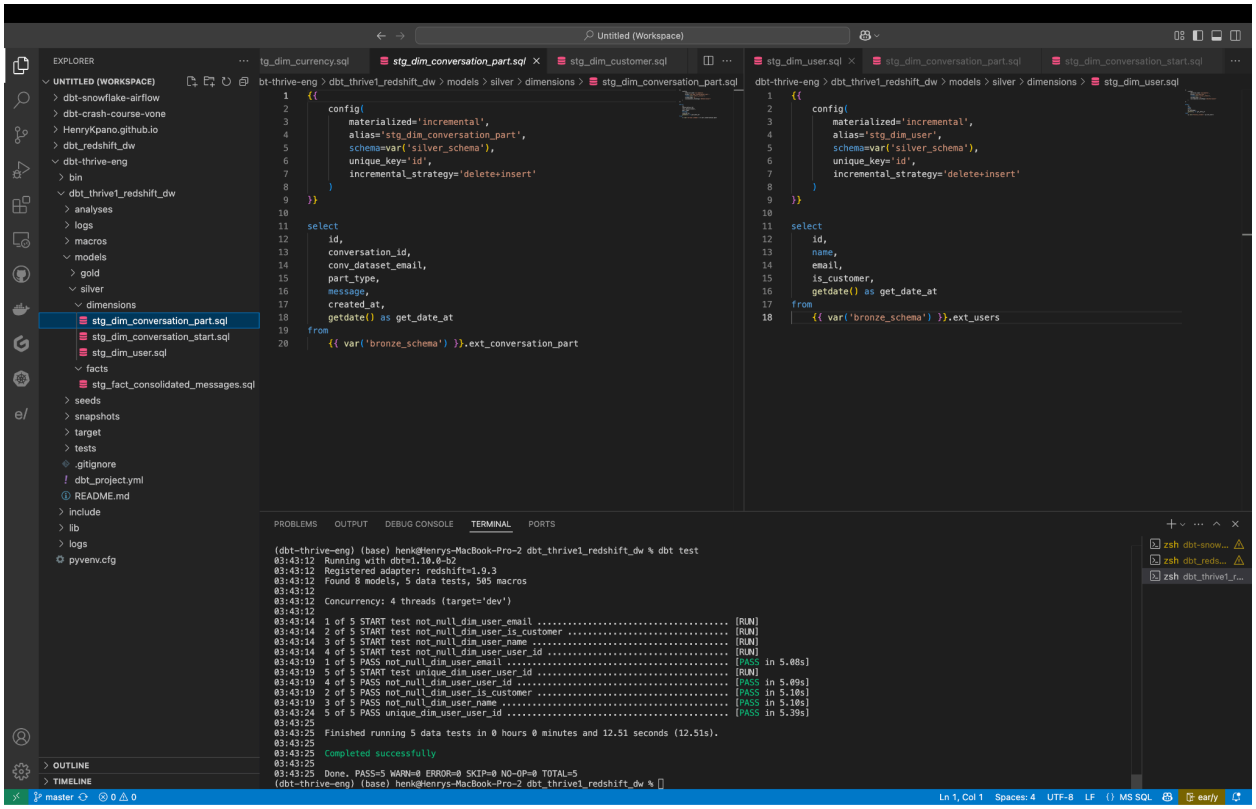
This is where the bronze layer of our architecture is done, which is the raw data of the source data. To prevent loss of data in the entire process.



In this layer, we create the dimensions in their raw state.

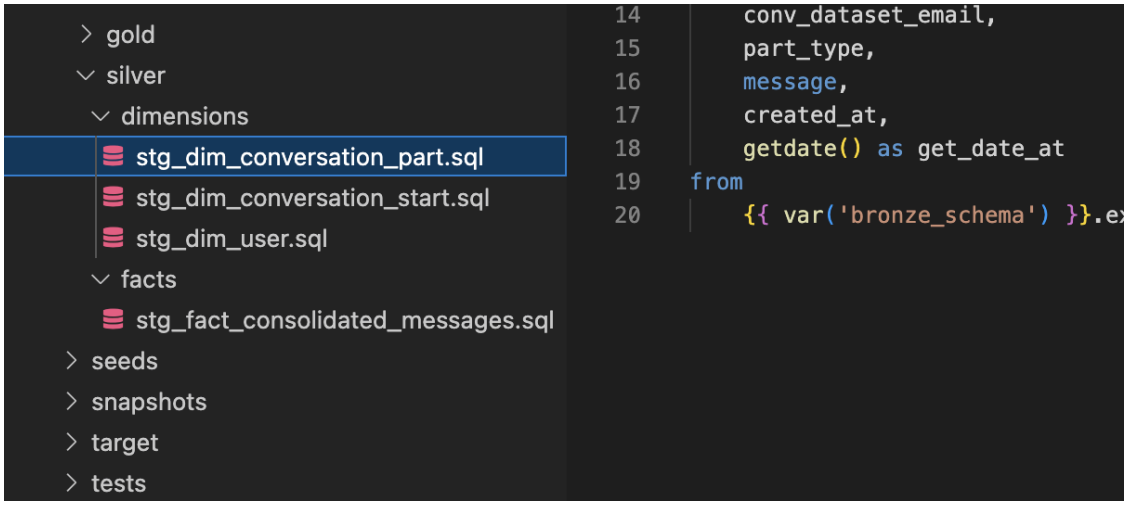
Build the DBT Silver Layer

In this layer, there will be some data cleaning remaining to be performed for a better understanding of the data. This will be done using DBT core. Create a new dbt project in your virtual environment and give it your preferred name. In this project, we named it ***dbt-thrive1-redshift-dw***. Connect to Redshift using the dbt-redshift adaptor.



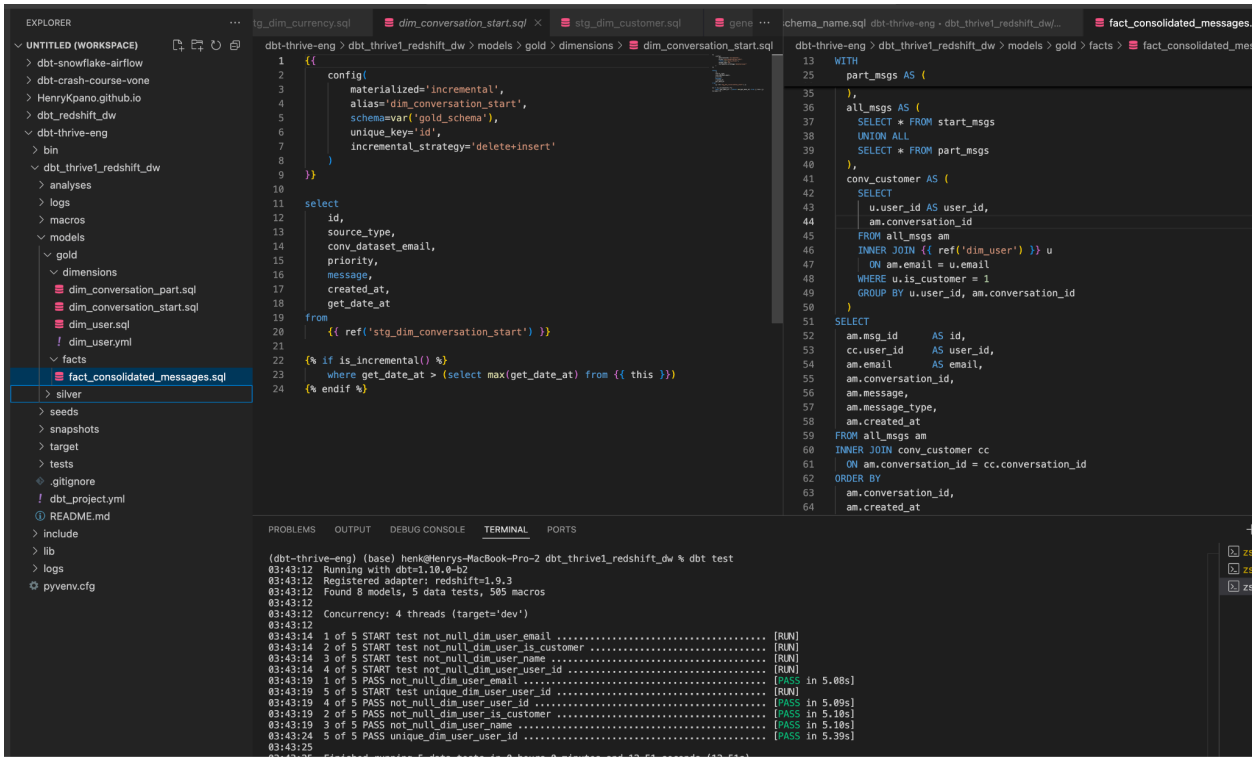
In the view above, we create the various dimensions shown in the logical view. This view will stage the various dimensions for later reference by the fact tables. The dimensions will be named as

1. stg_dim_user
2. stg_dim_conversation_start
3. stg_dim_conversation_part
4. stg_fact_consolidated_messages



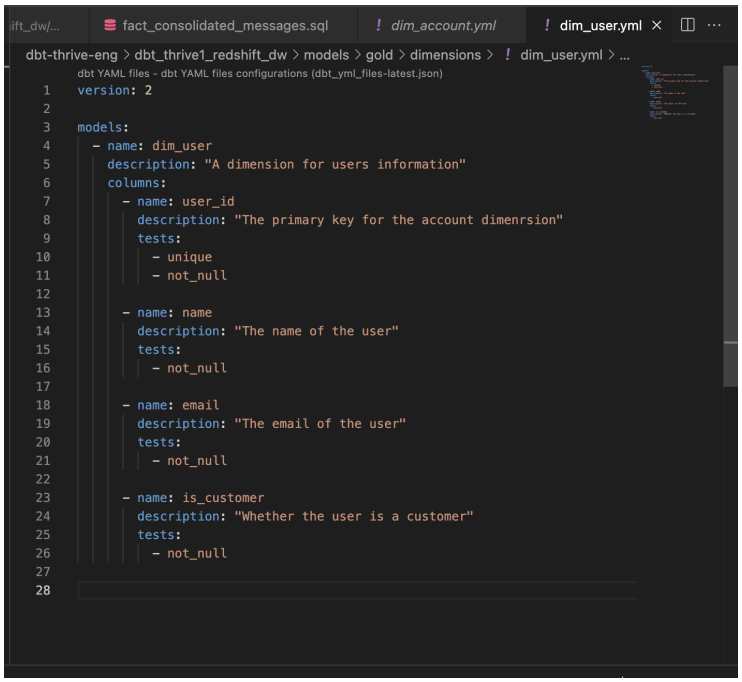
Build the DBT Gold Layer

In this layer, this is the final layer in the medallion architecture, where we create the final dimensional tables, and from them, the fact view is created. The dimensions use incremental materialization to be able to get updates when they happen.



Test Case: User table test

For the user table, we wrote a test case to test if the data meets our desired information and structure. In this case, if all the test cases are passed, then this is great for us. In the event of a failure, the code will need to be reviewed.



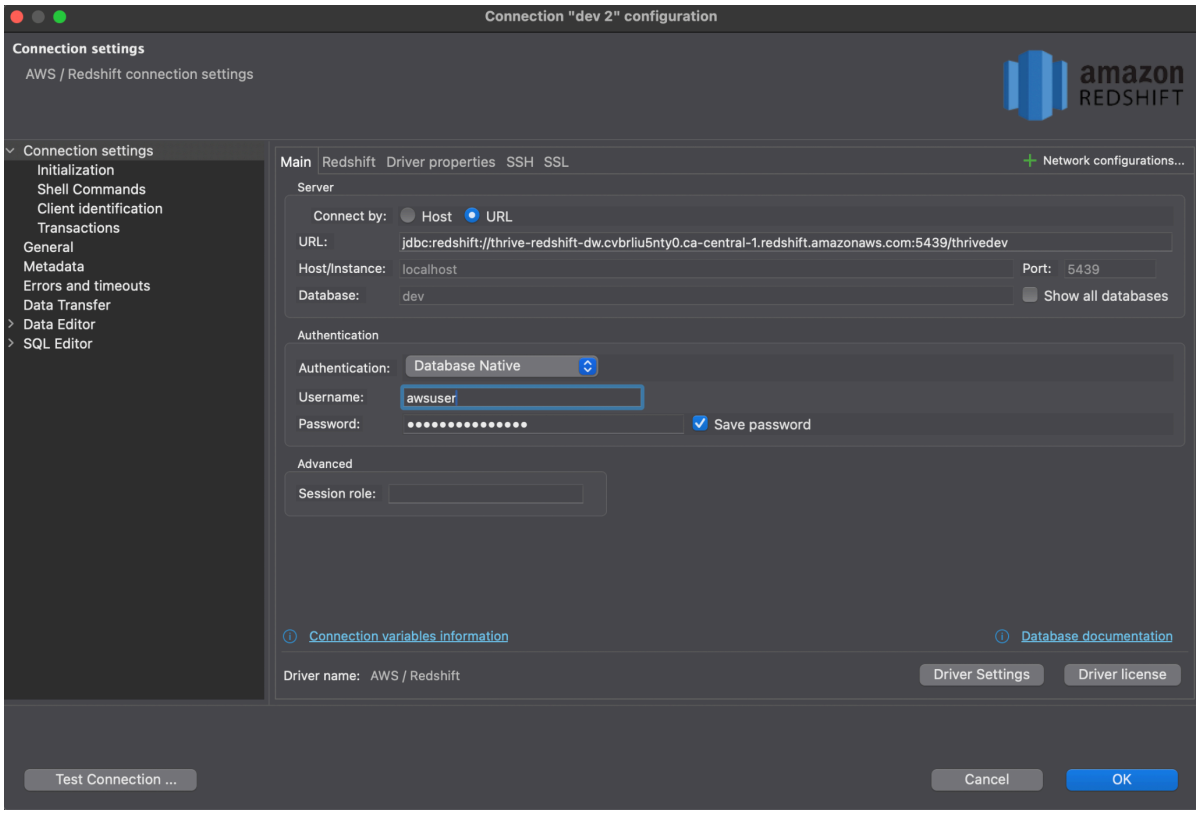
Access 1: Github Repo

The link to this DBT Thrive Career Wellness project can be found below. It is a public repository and can be cloned as well.
Link: https://github.com/HenryKpano/Thrive-AWS-DBT-Enginering-project

Access 2: Access to the Redshift database using DBeaver

The details below will enable anybody to have access to the Redshift database built from their DBeaver tool. This will be active for a week and deactivated after.

1. Connect by URL
2. URL: jdbc:redshift://thrive-redshift-dw.cvbrliu5nty0.ca-central-1.redshift.amazonaws.com:5439/thrivedev
3. Username: awsuser
4. Password: tHriveusereng1#



Conclusion

In this project, we implemented the full structure of the medallion architecture of the ELT process for Thrive Career Wellness. This helps to keep data clean and accurate.