Northeastern University
College of Engineering

IE 6750: DATA WAREHOUSING & INTEGRATION

Milestone #6

Group No. 7

Student 1: Rahul Daruka
(daruka.r@northeastern.edu)

Student 2: Sharvari Deshpande
(deshpande.sha@northeastern.edu)

Percentage of effort contributed by student 1: 50%
Percentage of effort contributed by student 2: 50%
Signature of student 1: Rahul Daruka
Signature of student 2: Sharvari Deshpande
Submission Date: 24th November 2024

Table of Contents

1. Introduction:

In today's rapidly urbanizing world, the efficient management of parking resources has become a critical component of urban infrastructure. With vehicle ownership rates climbing and urban populations expanding, the strain on existing parking facilities has reached unprecedented levels. Traditional parking management systems are increasingly proving inadequate, primarily due to their reliance on outdated static data warehousing methods. These systems are unable to process and analyze data in real time, resulting in several significant issues:

- Congestion in Parking Facilities: Due to the lack of real-time data processing, traditional systems cannot effectively manage the inflow and outflow of vehicles, leading to frequent congestion. This congestion is not just a minor inconvenience; it significantly affects the daily commute of thousands, contributing to overall urban traffic congestion.
- Increased Driver Frustration: Inefficiencies in parking management directly impact user experience, manifesting as increased time spent searching for parking spots. This not only frustrates drivers but also leads to increased fuel consumption and emissions.
- Missed Revenue Opportunities: Inadequate utilization of parking spaces and inefficient pricing strategies, due to the absence of dynamic data, result in lost revenue opportunities for parking operators. This is compounded by the potential loss of customers who may seek alternative parking options due to dissatisfaction with service levels.

2. Problem Statement:

The central challenge that plagues current parking management systems is the significant latency inherent in their data handling and processing capabilities. This latency severely impedes the system's ability to make informed and timely decisions that are critical in managing the dynamic demands of urban parking. Traditional systems, typically reliant on static data warehouses, are not designed to handle the rapid fluctuations and complex variables that characterize modern urban parking scenarios. This inefficiency manifests in several critical areas impacting overall service delivery and system performance:

1. Inability to Adapt to Real-Time Demands:
Traditional parking systems fail to adapt quickly to real-time conditions due to slow data processing speeds. In a typical scenario, data collected from parking sensors, ticketing systems, and user inputs must undergo a lengthy process of extraction, loading, and analysis before actionable insights can be derived. During this delay, the actual parking situation may have altered significantly, rendering the insights outdated and ineffective by the time they are available for use.

2. Suboptimal Space Utilization:
The latency in data processing directly affects the ability of these systems to efficiently manage parking spaces. Without access to real-time occupancy data, parking operators cannot implement dynamic allocation strategies or adjust to sudden increases or decreases in demand. This often leads to scenarios where parking lots are either congested or underutilized, neither of which is optimal for maximizing usage or revenue.

3. Poor Customer Experience:
For customers, the most tangible impact of these outdated systems is the difficulty in finding available parking spaces. The lack of real-time data means that customers are often provided with inaccurate information regarding space availability. This not only leads to increased time spent searching for parking but also contributes to traffic congestion and frustration, degrading the overall user experience.

4. Missed Revenue Opportunities:
Effective revenue management in parking systems depends heavily on the ability to implement flexible pricing models that can adapt to changing demand patterns. However, without real-time data analytics, parking operators miss out on implementing dynamic pricing strategies that could capitalize on peak demand periods. Additionally, the inability to offer targeted promotions or discounts in real-time further diminishes potential revenue.

5. Integration Challenges:
Current systems often operate in silos with minimal interoperability between different data sources such as payment gateways, vehicle detection systems, and customer feedback platforms. This lack of integration complicates the data landscape, making it difficult to synthesize data from multiple sources into a coherent analytical framework. As a result, the full potential of the data remains untapped, limiting the scope and accuracy of predictive analytics.

6. Inadequate Incident and Crisis Management:
In situations where rapid response is crucial, such as managing parking during large events or resolving access disputes, the delayed data processing further exacerbates the problem. Operators lack the necessary situational awareness to manage incidents effectively, leading to operational inefficiencies and potential security risks.

The proposed solution, the ParkEasy data pipeline, aims to revolutionize how parking data is managed by facilitating the seamless integration and real-time processing of diverse data streams. By transforming data handling capabilities, ParkEasy seeks to enhance operational efficiency, improve customer service, maximize revenue potential, and streamline incident management in urban parking environments.

3. Objectives of the Data Pipeline:

The overarching objective of the SmartPark data pipeline is to transform parking management into a highly responsive, customer-centric service. Specific goals include:

- **Real-time Data Flow**: Facilitating the continuous flow of data from various sources to ensure that information regarding parking space availability and usage is always current.
- **Operational Efficiency**: Automating key management functions such as dynamic pricing adjustments, parking slot reservations, and quick incident response.
- **Enhanced Customer Experience**: Providing real-time updates to customers about parking availability through user-friendly mobile interfaces, thereby improving user interaction and satisfaction.

- **Predictive and Strategic Insights**: Employing historical data analysis in conjunction with real-time data to forecast parking demand, identify usage patterns, and guide strategic management decisions.

---

4. Dataset for Analysis:

The data population methodology for ParkEasy entails a systematic approach to creating and maintaining a robust database that can support both initial system functionalities and scalable future needs. This process is divided into two main phases: the initial manual generation of data and subsequent plans for automated data generation. This strategy ensures that while the system starts with a solid foundation of manually populated data, it is also designed to evolve through more sophisticated, automated methods that can handle increased data volume and complexity.

Phase 1: Manual Data Population
Initially, the data required for the ParkEasy system will be generated manually to set up the core structure and to ensure that all system components function correctly with realistic data sets.

1. **Database Creation**:
    - **Tool Used**: PostgreSQL is utilized for database creation due to its robustness and ability to handle complex query operations.
    - **Schema Setup**: The database schema is designed based on the relational model of the ParkEasy system which includes various entities such as parking lots, users, and transactions.
    - **Table Definition**: SQL commands are crafted to define tables with appropriate constraints, data types, and indices to optimize performance and data integrity.
2. **Table Structure Implementation**:
    - **Core Tables**: Tables such as Admin, Member, ParkingLot, ParkingSlot, Booking, Payment, and Incident are created to support essential functionalities.
    - **Data Entry**: Each table is populated using SQL INSERT statements that introduce realistic but fabricated data reflecting typical operational scenarios.
3. **Data Insertion**:
    - **Execution**: Manual insertion of data through SQL scripts ensures that all relationships and constraints among tables are respected.
    - **Validation**: Regular checks are performed to validate data consistency and integrity across the system.

Phase 2: Automated Data Population (Future Strategy)
As the ParkEasy system matures and expands, the data population strategy will shift towards automation to efficiently manage larger datasets and more complex scenarios.

1. **Automated Data Generation**:
    - **Tool**: Use of Mockaroo, a sophisticated online data generator, to create structured data in CSV format that mirrors the database schema.
    - **Scope**: Generation of data for dynamic entities such as Booking, Payment, and Vehicle, which require frequent updates and high volume.
2. **Scripted Data Insertion**:
    - **Implementation**: Development of Python and SQL scripts to automate the data insertion process.

    - **Bulk Operations**: Utilization of PostgreSQL's COPY command for bulk data import from CSV files, enhancing the speed and efficiency of the data population process.
3. **Scalability for Transactional Data**:
    - **High Volume Data Handling**: Focus on generating and managing large volumes of transactional data to support extensive analytics and system testing.
    - **Real-Time Simulation**: Integration of mechanisms to simulate real-time data updates, critical for entities like booking statuses and incident management.

5. Volume and Variety in Data Sources:

The ParkEasy data pipeline is designed to manage and analyze data from multiple sources, accommodating both the volume and variety inherent in the operation of 5-10 parking lots, each containing over 50+ individual parking slots. With more than 100+ active members, the system handles a significant volume of bookings and financial transactions, necessitating a robust infrastructure that can process and analyze data efficiently. Here's a detailed look at how volume and variety are considered in the SmartPark data sources:

Volume Considerations-
The system is engineered to handle large-scale data inputs and interactions generated from day-to-day operations:

1. Granular Time Stamps:
- **Detail**: Every transaction and booking record includes precise time stamps down to the second.
- **Volume**: This results in a massive increase in data granularity, capturing every moment of operation which accumulates vast amounts of data over time.
- **Impact**: The detailed time stamps require the database to handle a higher transaction load and provide the basis for more detailed time-series analysis.

2. **Extensive Logging of User Interactions**:
- **Detail**: Every interaction by the 100+ members with the parking management system, including payment transactions, is logged.
- **Volume**: This leads to a large dataset comprising user actions, preferences, and feedback.
- **Impact**: Requires robust data management strategies to ensure that the system can quickly access and analyze these interactions for customer service and operational improvements.

3. **High-Density Booking Data**:
- **Detail**: Each parking lot supports over 50 slots, and with multiple bookings per slot per day, the dataset grows exponentially.
- **Volume**: Accumulates detailed records of bookings, cancellations, and modifications.
- **Impact**: Necessitates efficient indexing and partitioning in the database to speed up data retrieval and enhance performance during peak load times.

Variety Considerations-
1. Diverse Transaction Types:
- **Detail**: Transactions not only include payments but also fines, refunds, and adjustments.

- **Management**: The database schema includes various transaction types, each requiring specific handling, validation, and processing within the SQL environment.

2. **Multiple User Roles and Access Levels**:
   - **Detail**: Different data views and permissions for admins, regular members, VIP members, and guest users.
   - **Management**: Implementation of role-based access control within the SQL database ensures that data integrity and security are maintained while providing customized data views based on user roles.

The primary data source for the ETL process was an operational OLTP (Online Transaction Processing) database. This OLTP system contains transactional data generated manually using unput statements, including entities such as Admin, ParkingLot, ParkingSlot, Booking, Payment, Incident, and Member. This source provides transactional records that form the basis for the fact tables in the data warehouse.

Prepopulated Time Dimension:

• The time dimension in the data warehouse was prepopulated as a static reference table, providing detailed temporal attributes such as year, quarter, month, day, and time of day. This prepopulated time dimension serves as a reference that is essential for time-based analysis across the data warehouse.

• Using a prepopulated time dimension provides flexibility and efficiency, as it allows for consistent and easy querying of time-related data without needing frequent updates. The time dimension is fundamental for creating time-based reports, tracking trends over periods, and supporting calculations like monthly revenue, booking durations, and incident resolutions over time.

6. Fact Tables:

Fact tables contain measurable, quantitative data and foreign keys that reference dimension tables. They form the core of the data warehouse as they store the events or transactional data.

1. Booking Fact Table

- **Purpose**: Stores information related to parking bookings.
- **Measures**:
  - Booking ID (PK): Unique identifier for each booking.
  - Transaction ID (FK): Links to the payment fact table.
  - Status: The status of the booking (e.g., active, completed, canceled).
  - Total Booking Cost: The total booking cost is calculated as the difference between End_DateTime and Start_DateTime multiplied by the price
  - Total Booking Duration: The total booking duration is calculated as the sum of the difference between End_DateTime and Start_DateTime for all bookings in the FACT_Booking table
- **Cardinality**:
  - 1, N bookings can occur for each Parking Slot and Member.
- **Relationships**:
  - Linked with dimensions: Parking Slot, Vehicle, Time, Member.

- **Example Query**: Total number of bookings per parking lot by month.

2. **Payment Fact Table**

- **Purpose**: Stores payment details for parking bookings.
- **Measures**:
    - Transaction ID (PK): Unique identifier for each payment.
    - Payment Status: Status of the payment (e.g., completed, pending).
    - Amount: The total amount paid.
- **Cardinality**:
    - 1, N payments are linked to each Member and Booking.
- **Relationships**:
    - Linked with dimensions: Time, Member.
- **Example Query**: Total revenue collected by year from all members.

3. **Incident Fact Table**

- **Purpose**: Logs incidents related to parking activities.
- **Measures**:
    - Incident ID (PK): Unique identifier for each incident.
    - Resolution Status: Indicates if the incident is resolved.
    - Incident Count: Tracks the number of incidents per period.
- **Cardinality**:
    - 1, N incidents can occur for each Admin and Parking Slot.
- **Relationships**:
    - Linked with dimensions: Admin, Parking Slot, Incident Details, Time.
- **Example Query**: Monthly count of incidents by parking lot.

7. Dimension Tables: Dimension tables store descriptive information about the business entities involved in the fact events. They provide context to the facts and allow the users to slice, dice, and roll-up data for analysis.

1. **Time Dimension**
- Attributes:
    - Date_Time, Day_of_Week, Time
- Cardinality:
    - 1, N bookings and payments can be associated with each time period.
- Hierarchy:
    - Time → Year → Quarter → Month → Day.
- Example Query: Number of bookings per quarter for each parking lot.

2. **Member Dimension**
- Attributes:
    - Member ID (PK): Unique identifier for each member.
    - Member_Name, Phone No, Email, Address.
    - Membership ID (FK): Links to the Membership dimension.

- Cardinality:
    - 1, N bookings and payments can be made by each member.
- Example Query: Number of bookings made by premium members in 2024.

3. **Parking Slot Dimension**
- Attributes:
    - Slot ID (PK): Unique identifier for each parking slot.
    - Type: Type of parking slot
    - Status: Availability of the slot (free, occupied).
    - Price: Pricing for the parking slot.
- Cardinality:
    - 1, N bookings can be associated with each parking slot.
- Example Query: Parking slot usage analysis by status and price.

4. **Parking Lot Dimension**
- Attributes:
    - Lot ID (PK): Unique identifier for each parking lot.
    - Location ID (FK): Links to the Location dimension.
    - Name: Name of the lot
    - Capacity: Number of parking slots available.
- Cardinality:
    - 1, N parking slots are contained in each parking lot.
- Example Query: Total revenue per parking lot per year.

5. **Vehicle Dimension**
- Attributes:
    - Vehicle Number (PK): Unique identifier for each vehicle.
    - Make ID (FK): Links to the vehicle Make dimension.
    - Color
- Cardinality:
    - 1, N bookings can be associated with each vehicle.
- Example Query: Popular vehicle types used in the downtown parking lot.

6. **Admin Dimension**
- Attributes:
    - Admin ID (PK): Unique identifier for each administrator.
    - Admin_Name, Email, Phone No., Address
- Cardinality:
    - 1, N incidents can be managed by each admin.
- Example Query: Number of incidents resolved by each administrator.

7. **Incident Details Dimension**
- Attributes:
    - Type, Description, Severity.
- Cardinality:
    - 1, N incidents can be associated with each incident type.

- Example Query: Severity analysis of incidents reported in 2023.

8. **Membership Dimension**
   - Attributes:
     - Membership ID (PK): Unique identifier for each membership.
     - Membership_Name, Status, Membership Level.
   - Cardinality:
     - 1, N members can hold a membership.
   - Example Query: Booking analysis based on membership level (e.g., premium vs. basic).

9. **Discount Dimension**
   - Attributes:
     - Discount ID (PK): Unique identifier for each discount.
     - Percentage, Description, Status
   - Cardinality:
     - 1, N memberships may have a discount applied.
   - Example Query: Total discount provided to premium members in 2024.

10. **Location and Region Dimension**
    - Attributes:
      - Location ID (PK): Unique identifier for each location.
      - Location Name, Region ID (FK): Links to the Region dimension.
    - Cardinality:
      - 1, N parking lots belong to a location.
      - 1, N locations belong to a region.
    - Hierarchy:
      - Region → Location → Parking Lot → Parking Slot.
    - Example Query: Total bookings by region for each year.

11. **Vehicle Make and Type Dimension**
    - Attributes:
      - Make ID (PK): Unique identifier for each vehicle make.
      - Make_Name, Type ID (FK): Links to the Type dimension.
    - Cardinality:
      - 1, N vehicle makes are of a specific type.
    - Example Query: Analyze vehicle types used for parking in different locations.

Hierarchies:
- **Time Hierarchy**: Year → Quarter → Month → Day.
- **Location Hierarchy**: Region → Location → Parking Lot → Parking Slot.

Vehicle Hierarchy: Type → Make → Vehicle.

8. Strategic and Operational Insights for the Data Pipeline:

The ParkEasy data pipeline is designed to deliver a broad spectrum of strategic and operational insights that will enhance the efficiency and effectiveness of parking management. Leveraging

the data collected and processed through this advanced system, stakeholders can make informed decisions that not only improve daily operations but also shape long-term strategic initiatives. Below are the detailed insights that ParkEasy aims to achieve:

Operational Analytics
Operational insights focus on leveraging real-time and transactional data to streamline daily parking operations, ensuring optimal resource utilization and customer satisfaction:

1. Real-Time Availability Analysis:
    - Objective: Employ real-time data analytics to provide immediate updates on parking space availability.
2. Dynamic Pricing Optimization:
    - Objective: Implement data-driven pricing models that respond to changes in demand in real-time.
3. Incident Response Analytics:
    - Objective: Enhance the speed and effectiveness of incident management using automated systems.
4. Operational Efficiency Metrics:
    - Objective: Continuously monitor and improve the layout and operations of parking lots.

Strategic Analytics
Strategic insights utilize deep analytics to support long-term business goals, infrastructure planning, and policy development:

1. Customer Behavioral Analytics:
    - Objective: Deepen understanding of customer preferences and behaviors to enhance service personalization.
2. Capacity and Demand Forecasting:
    - Objective: Predict future parking capacity needs and plan infrastructure developments accordingly.
3. Sustainability Impact Studies:
    - Objective: Assess and improve the environmental sustainability of parking operations.
4. Market and Competitive Analytics:
    - Objective: Maintain a competitive edge by analyzing market trends and competitor strategies.
5. Regulatory Compliance Monitoring:
    - Objective: Ensure adherence to evolving regulatory requirements related to urban mobility and parking management.

9. Pipeline Tools:

The Parkeasy Data Pipeline employs a comprehensive set of tools to extract, transform, and load (ETL) data efficiently while maintaining scalability and robustness. These tools are integral in ensuring the seamless integration of data sources, accurate data transformation, and effective data loading into the target warehouse for analysis. Below is a detailed breakdown of each tool used in the pipeline, its functionality, and its specific role within the architecture.

1. AWS Glue:
AWS Glue is the central ETL tool orchestrating data extraction, transformation, and loading processes. It automates schema discovery, manages metadata, and performs custom Python-based data transformations.

Components used:

- **Data Crawlers:** Glue crawlers are set up to automatically discover schemas from the raw data in the source systems. These schemas are stored in the Glue Data Catalog, a centralized metadata repository. Crawlers are particularly useful for handling:
  - ➔ Data from the PostgreSQL database (hosted on Amazon RDS).
  - ➔ CSV files stored in an Amazon S3 bucket.
- **ETL Jobs**: Glue ETL jobs are configured with Python scripts to perform transformations such as:
  - ➔ Renaming columns to standardize naming conventions.
  - ➔ Adding surrogate keys to ensure data consistency.
  - ➔ Typecasting columns (e.g., casting percentages to float).
  - ➔ Complex operations like parsing dates or handling hierarchical data.

Integration: Glue integrates seamlessly with Amazon S3, Redshift, and the Glue Data Catalog, acting as a bridge between raw data sources and the processed data warehouse.

Role in the Pipeline:

1. Extracts data from Amazon RDS and S3.
2. Automates data transformation and schema discovery.
3. Writes processed data into Amazon Redshift for further analysis.

2. Amazon S3:
Amazon S3 acts as a storage layer for the pipeline, providing scalable and durable storage for semi-structured data. It is particularly used for storing flat files, such as CSV files.

Role in the Pipeline:

1. Hosts raw data files, including pre-populated dimension tables like the time dimension**.**
2. AWS Glue crawlers extract data from S3 to populate the Glue Data Catalog.
3. Acts as a temporary storage location for intermediate files during the ETL process.

Use Case:
Storing the time dimension table as a CSV file, which is pre-processed and integrated with other data sources during the ETL process.

3. Amazon RDS (PostgreSQL):
Amazon RDS is a managed database service hosting the PostgreSQL database, which serves as the transactional data source for the ETL pipeline.

Role in the Pipeline:

1. Acts as the primary source of transactional data, including fact and dimension tables.

2. AWS Glue jobs use JDBC connections to extract data from the RDS-hosted PostgreSQL database for processing.
3. Ensures high availability and reliability for data extraction operations.

Use case:
Source tables such as public_admin, public_membership, and public_vehicle are extracted from PostgreSQL for transformation and loading into Redshift.

4. AWS Glue Data Catalog:
The AWS Glue Data Catalog serves as the centralized metadata repository for the entire pipeline. It maintains information about the structure, schema, and locations of data sources.

Role in the Pipeline:
1. Stores metadata for the transactional PostgreSQL tables and CSV files.
2. Enables querying of transformed data using Amazon Athena.
3. Acts as the backbone for schema discovery and validation during ETL.

5. Amazon Redshift:
Amazon Redshift is the final destination for the transformed data. It is a fully managed data warehouse optimized for analytical workloads.

Role in the Pipeline:
1. Acts as the target system for the processed dimension and fact tables.
2. Data from AWS Glue ETL jobs is loaded into Redshift tables like dim_admin, dim_vehicle, and dim_membership.
3. Supports complex SQL queries for reporting and business intelligence.

Use case:
Serves as the core data warehouse for running analytics on Parkeasy data.

6. Amazon Athena:
Amazon Athena is a serverless query service that enables SQL-based querying of the data cataloged in AWS Glue.

Role in the pipeline:
1. Provides an interface to query transformed data stored in S3 via the Glue Data Catalog.
2. Facilitates ad hoc querying without requiring a dedicated data warehouse or infrastructure.

Use Case:
Enables business users to query processed data directly from S3 for quick insights.

7. AWS Step Functions:
AWS Step Functions orchestrate the workflow of the ETL pipeline, ensuring smooth execution of dependent tasks.

Role in the pipeline:
1. Manages the sequence of operations, including:

- Data extraction from RDS and S3.
- Running Glue ETL jobs.
- Loading data into Redshift.

2. Provides fault tolerance by retrying failed steps or alerting users in case of issues.

A lambda function was created for the same which was used to trigger the crawlers to automated pulling the data

8. Amazon Cloudwatch:
Amazon CloudWatch is the monitoring and logging tool for the pipeline.

Role in the pipeline:

1. Logs and monitors the execution of Glue jobs and Step Functions.
2. Provides visibility into pipeline performance and potential bottlenecks.

10. Programming Choice

The programming language used for the Parkeasy ETL Pipeline is Python. Python scripts play a pivotal role in the pipeline, especially in the ETL jobs executed by AWS Glue. These scripts are tailored to meet the pipeline's transformation requirements and handle diverse data sources, such as PostgreSQL (via Amazon RDS) and flat files (CSV format). Below is an overview of how Python has been used:

1. Data Extraction:
Python scripts leverage AWS Glue's PySpark API to extract data from:

- PostgreSQL tables via a JDBC connection.
- CSV files stored in Amazon S3.

Example –

```
## Read data from Glue Data Catalog (public_admin table)
datasource0 = glueContext.create_dynamic_frame.from_catalog(
    database="parkeasy_data_catalog",  # Replace with your Glue Data Catalog database
    table_name="public_admin",         # Replace with your table name
    transformation_ctx="datasource0"
)
```

2. Data Transformation:
Python handles complex transformations, including:

- **Column Renaming**: Standardizes naming conventions for easier analysis.
- **Data Type Casting**: Converts data to appropriate formats (e.g., strings to floats).
- **Key Generation**: Adds surrogate keys using Spark functions like monotonically_increasing_id.
- **Custom Logic**: Python scripts include custom transformations such as date parsing or conditional logic.

Example –

```
## Transformations
df_transformed = df.withColumnRenamed("adminid", "AdminID") \
                   .withColumnRenamed("name", "Name") \
                   .withColumnRenamed("address", "Address") \
                   .withColumnRenamed("email", "Email") \
                   .withColumnRenamed("phoneno", "PhoneNo") \
                   .withColumnRenamed("username", "Username") \
                   .withColumn("AdminKey", monotonically_increasing_id())
```

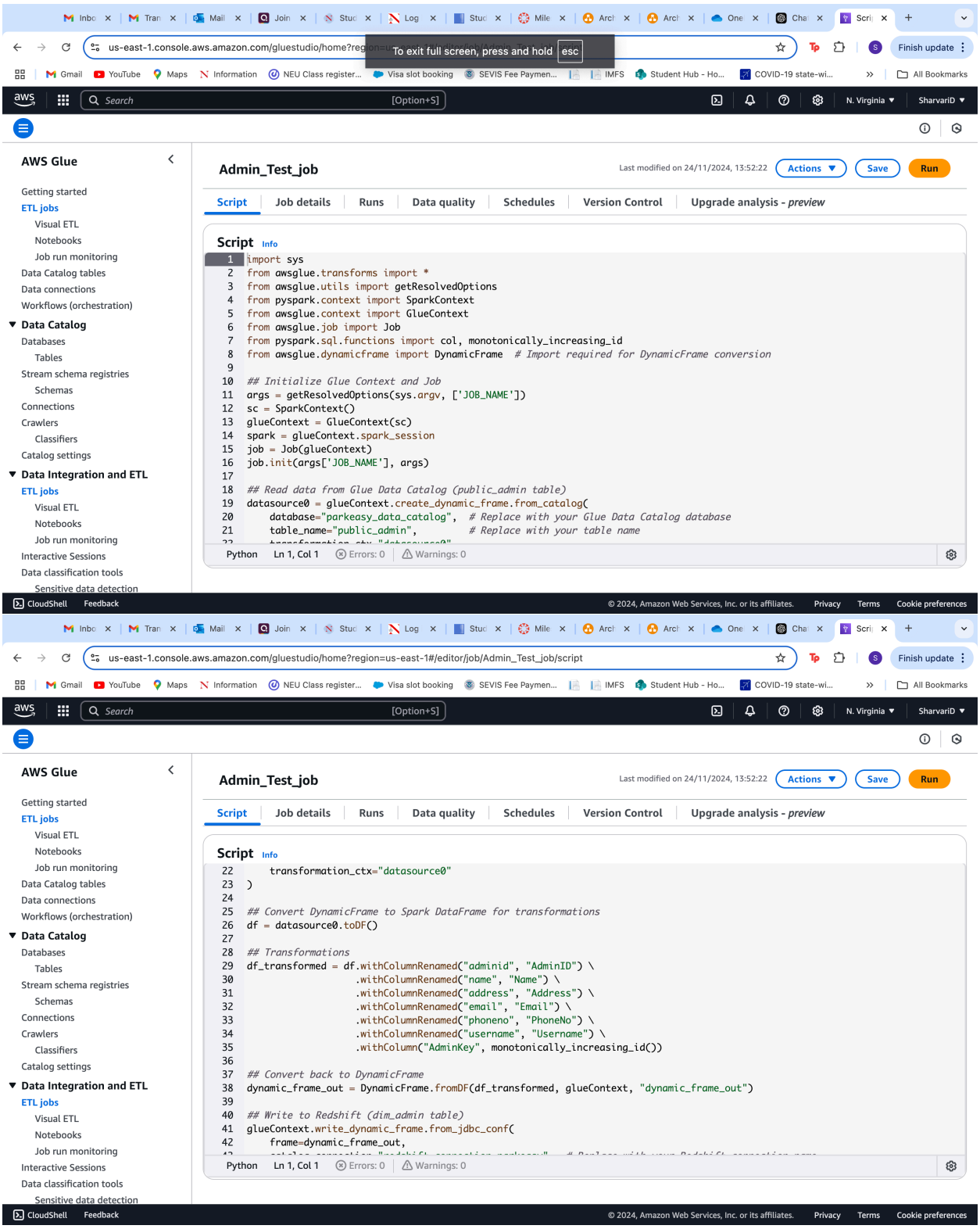
3. Data Loading:
Python scripts write transformed data into Amazon Redshift using Glue's JDBC connection APIs.

Example –

```
## Write to Redshift (dim_admin table)
glueContext.write_dynamic_frame.from_jdbc_conf(
    frame=dynamic_frame_out,
    catalog_connection="redshift-connection-parkeasy",  # Replace with your Redshift connection name
    connection_options={
        "dbtable": "dim_admin",  # Target table in Redshift
        "database": "dev"        # Target database in Redshift
    },
    redshift_tmp_dir="s3://project-parkeasy-data-bucket/Temp/"  # Temporary directory in S3
)
```

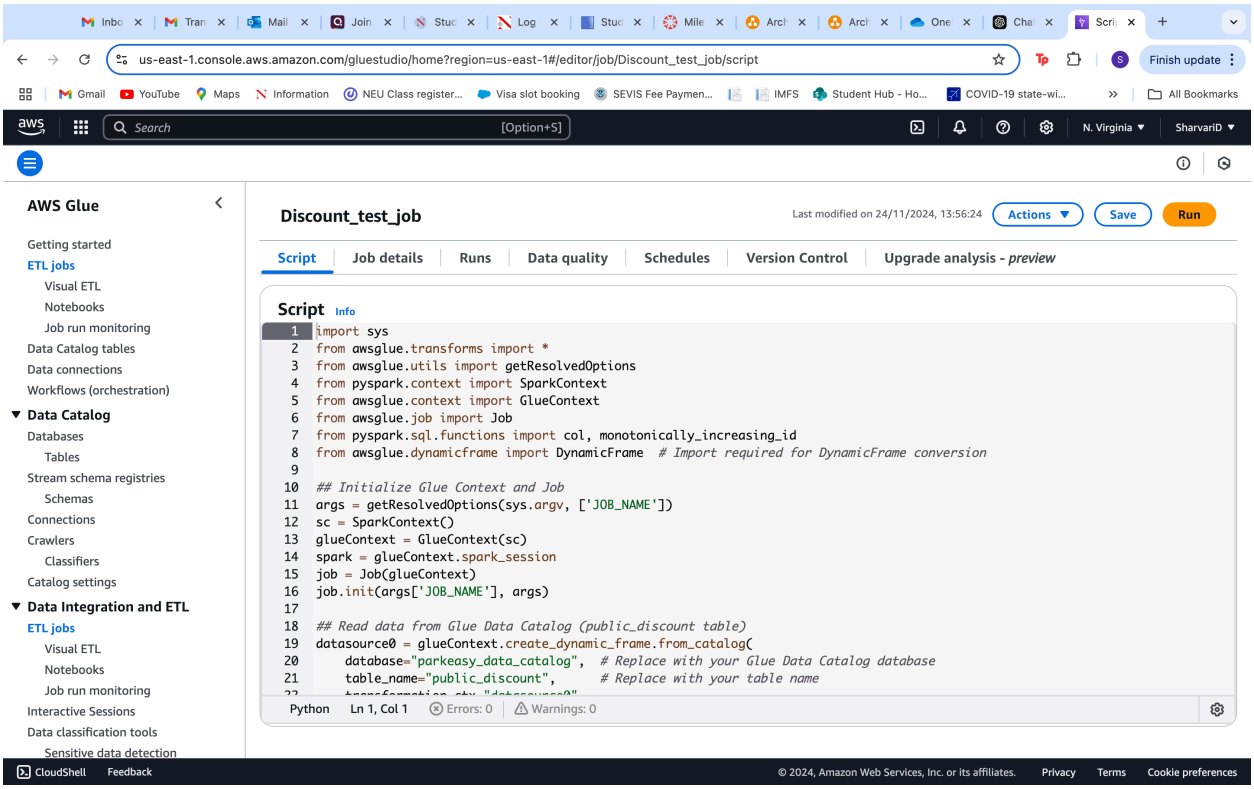
4. Error handling and logging:
Python enables robust error handling to manage job failures, ensuring retries and logging details to CloudWatch Logs for debugging.

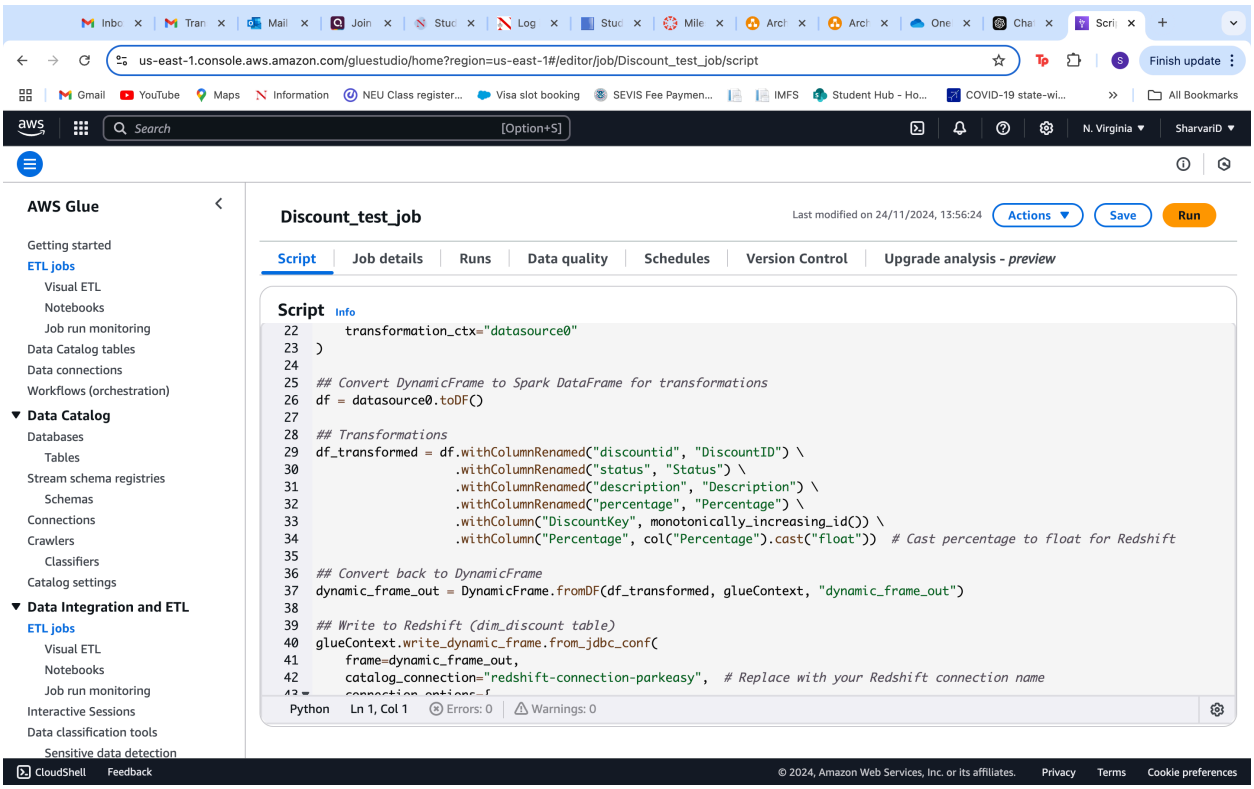
Python scripts for all the jobs –
1. Admin_job


To exit full screen, press and hold esc
us-east-1.console.aws.amazon.com/gluestudio/home?region=us-east-1#/editor/job/Admin_Test_job/script
Finish update
Search
[Option+S]
N. Virginia
SharvariD
AWS Glue
Getting started
ETL jobs
Visual ETL
Notebooks
Job run monitoring
Data Catalog tables
Data connections
Workflows (orchestration)
Data Catalog
Databases
Tables
Stream schema registries
Schemas
Connections
Crawlers
Classifiers
Catalog settings
Data Integration and ETL
ETL jobs
Visual ETL
Notebooks
Job run monitoring
Interactive Sessions
Data classification tools
Sensitive data detection
Admin_Test_job
Last modified on 24/11/2024, 13:52:22
Actions
Save
Run
Script
Job details
Runs
Data quality
Schedules
Version Control
Upgrade analysis - preview
Script Info
1 import sys
2 from awsglue.transforms import *
3 from awsglue.utils import getResolvedOptions
4 from pyspark.context import SparkContext
5 from awsglue.context import GlueContext
6 from awsglue.job import Job
7 from pyspark.sql.functions import col, monotonically_increasing_id
8 from awsglue.dynamicframe import DynamicFrame # Import required for DynamicFrame conversion
9
10 ## Initialize Glue Context and Job
11 args = getResolvedOptions(sys.argv, ['JOB_NAME'])
12 sc = SparkContext()
13 glueContext = GlueContext(sc)
14 spark = glueContext.spark_session
15 job = Job(glueContext)
16 job.init(args['JOB_NAME'], args)
17
18 ## Read data from Glue Data Catalog (public_admin table)
19 datasource0 = glueContext.create_dynamic_frame.from_catalog(
20 database="parkeasy_data_catalog", # Replace with your Glue Data Catalog database
21 table_name="public_admin", # Replace with your table name
Python
Ln 1, Col 1
Errors: 0
Warnings: 0
CloudShell
Feedback
© 2024, Amazon Web Services, Inc. or its affiliates.
Privacy
Terms
Cookie preferences
22 transformation_ctx="datasource0"
23)
24
25 ## Convert DynamicFrame to Spark DataFrame for transformations
26 df = datasource0.toDF()
27
28 ## Transformations
29 df_transformed = df.withColumnRenamed("adminid", "AdminID") \
30 .withColumnRenamed("name", "Name") \
31 .withColumnRenamed("address", "Address") \
32 .withColumnRenamed("email", "Email") \
33 .withColumnRenamed("phoneno", "PhoneNo") \
34 .withColumnRenamed("username", "Username") \
35 .withColumn("AdminKey", monotonically_increasing_id())
36
37 ## Convert back to DynamicFrame
38 dynamic_frame_out = DynamicFrame.fromDF(df_transformed, glueContext, "dynamic_frame_out")
39
40 ## Write to Redshift (dim_admin table)
41 glueContext.write_dynamic_frame.from_jdbc_conf(
42 frame=dynamic_frame_out,

```
    catalog_connection="redshift-connection-parkeasy",  # Replace with your Redshift connection name
    connection_options={
        "dbtable": "dim_admin",  # Target table in Redshift
        "database": "dev"        # Target database in Redshift
    },
    redshift_tmp_dir="s3://project-parkeasy-data-bucket/Temp/"  # Temporary directory in S3
)

job.commit()
```

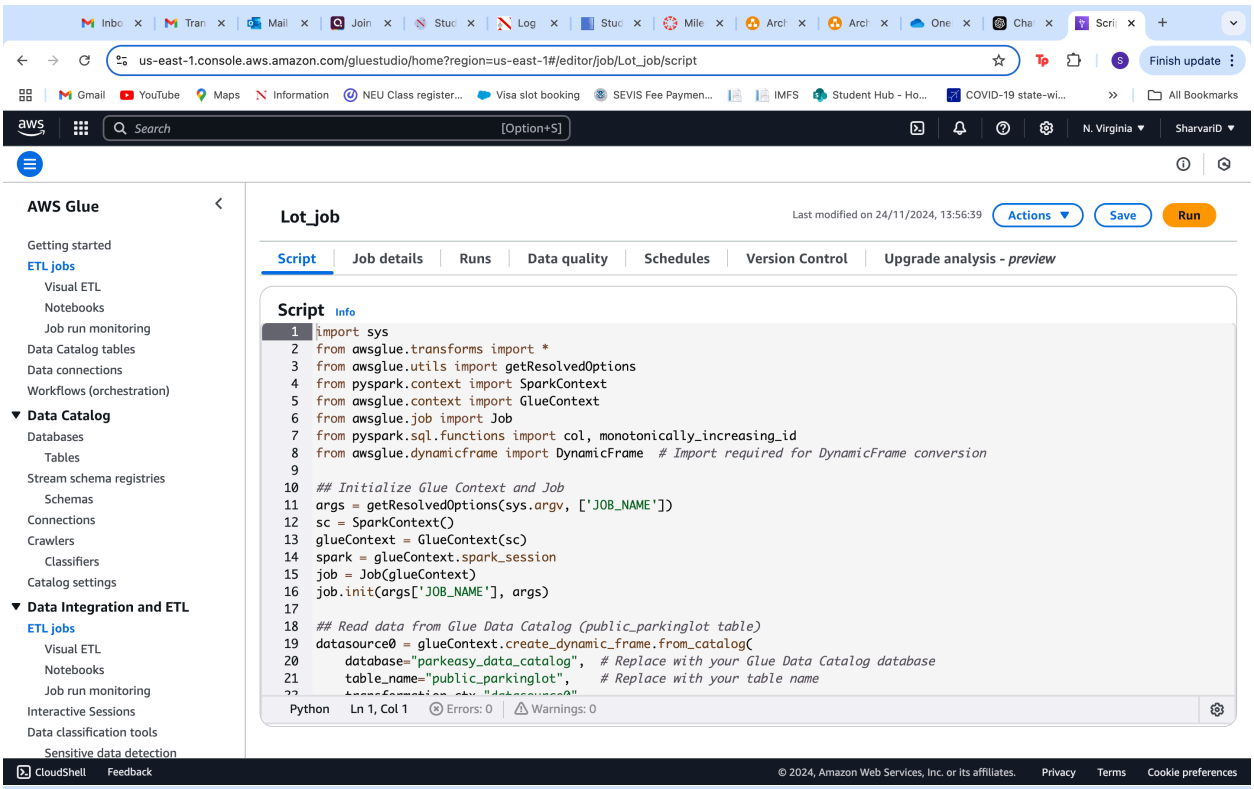
2.Discount_job

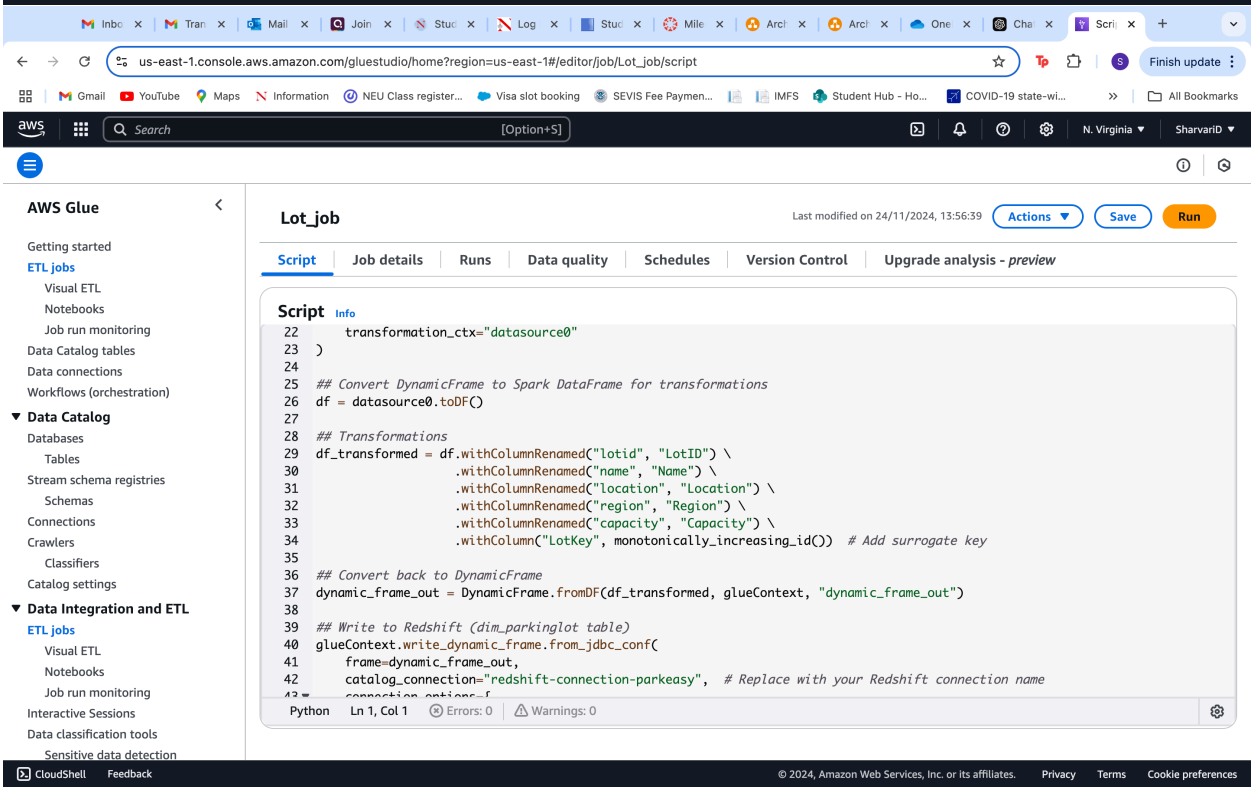





```
    connection_options={
        "dbtable": "dim_discount",  # Target table in Redshift
        "database": "dev"           # Target database in Redshift
    },
    redshift_tmp_dir="s3://project-parkeasy-data-bucket/Temp/"  # Temporary directory in S3
)

job.commit()

```

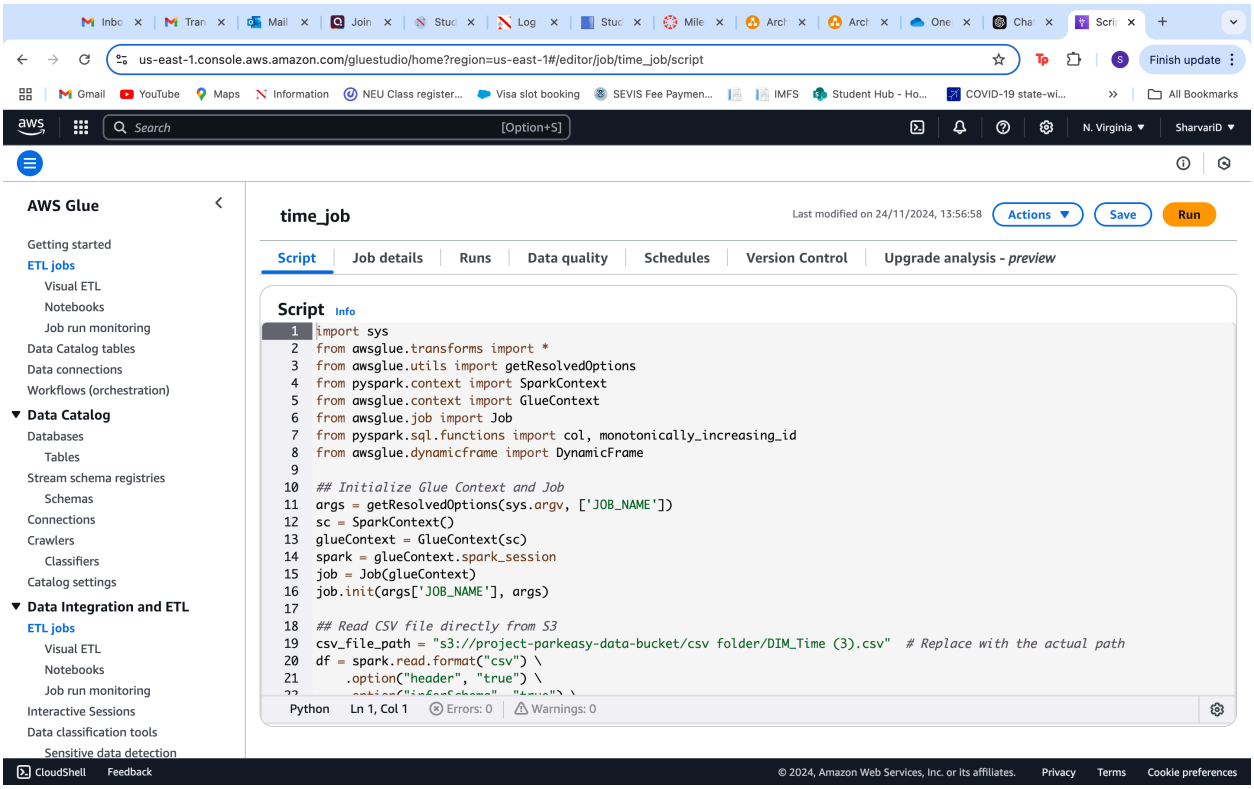
3.ParkingLot_job

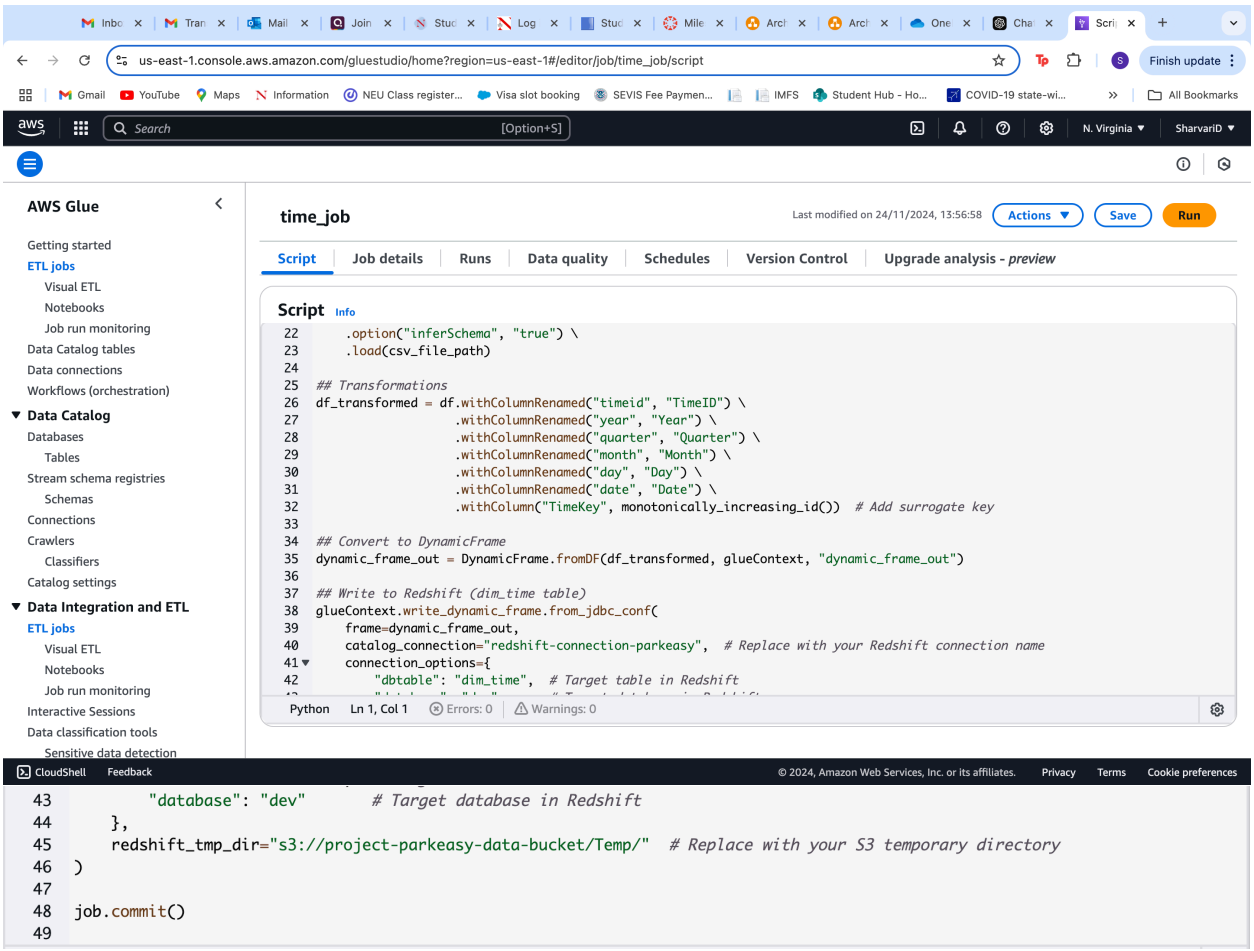

us-east-1.console.aws.amazon.com/gluestudio/home?region=us-east-1#/editor/job/Lot_job/script
Finish update
Gmail YouTube Maps Information NEU Class register... Visa slot booking SEVIS Fee Paymen... IMFS Student Hub - Ho... COVID-19 state-wi... All Bookmarks
Search [Option+S]
N. Virginia SharvariD
AWS Glue
Getting started
ETL jobs
Visual ETL
Notebooks
Job run monitoring
Data Catalog tables
Data connections
Workflows (orchestration)
Data Catalog
Databases
Tables
Stream schema registries
Schemas
Connections
Crawlers
Classifiers
Catalog settings
Data Integration and ETL
ETL jobs
Visual ETL
Notebooks
Job run monitoring
Interactive Sessions
Data classification tools
Sensitive data detection
Lot_job
Last modified on 24/11/2024, 13:56:39
Actions
Save
Run
Script
Job details
Runs
Data quality
Schedules
Version Control
Upgrade analysis - preview
Script Info
1 import sys
2 from awsglue.transforms import *
3 from awsglue.utils import getResolvedOptions
4 from pyspark.context import SparkContext
5 from awsglue.context import GlueContext
6 from awsglue.job import Job
7 from pyspark.sql.functions import col, monotonically_increasing_id
8 from awsglue.dynamicframe import DynamicFrame # Import required for DynamicFrame conversion
9
10 ## Initialize Glue Context and Job
11 args = getResolvedOptions(sys.argv, ['JOB_NAME'])
12 sc = SparkContext()
13 glueContext = GlueContext(sc)
14 spark = glueContext.spark_session
15 job = Job(glueContext)
16 job.init(args['JOB_NAME'], args)
17
18 ## Read data from Glue Data Catalog (public_parkinglot table)
19 datasource0 = glueContext.create_dynamic_frame.from_catalog(
20 database="parkeasy_data_catalog", # Replace with your Glue Data Catalog database
21 table_name="public_parkinglot", # Replace with your table name
Python Ln 1, Col 1 Errors: 0 Warnings: 0
CloudShell Feedback
© 2024, Amazon Web Services, Inc. or its affiliates. Privacy Terms Cookie preferences


us-east-1.console.aws.amazon.com/gluestudio/home?region=us-east-1#/editor/job/Lot_job/script
Finish update
Gmail YouTube Maps Information NEU Class register... Visa slot booking SEVIS Fee Paymen... IMFS Student Hub - Ho... COVID-19 state-wi... All Bookmarks
Search [Option+S]
N. Virginia SharvariD
AWS Glue
Getting started
ETL jobs
Visual ETL
Notebooks
Job run monitoring
Data Catalog tables
Data connections
Workflows (orchestration)
Data Catalog
Databases
Tables
Stream schema registries
Schemas
Connections
Crawlers
Classifiers
Catalog settings
Data Integration and ETL
ETL jobs
Visual ETL
Notebooks
Job run monitoring
Interactive Sessions
Data classification tools
Sensitive data detection
Lot_job
Last modified on 24/11/2024, 13:56:39
Actions
Save
Run
Script
Job details
Runs
Data quality
Schedules
Version Control
Upgrade analysis - preview
Script Info
22 transformation_ctx="datasource0"
23)
24
25 ## Convert DynamicFrame to Spark DataFrame for transformations
26 df = datasource0.toDF()
27
28 ## Transformations
29 df_transformed = df.withColumnRenamed("lotid", "LotID") \
30 .withColumnRenamed("name", "Name") \
31 .withColumnRenamed("location", "Location") \
32 .withColumnRenamed("region", "Region") \
33 .withColumnRenamed("capacity", "Capacity") \
34 .withColumn("LotKey", monotonically_increasing_id()) # Add surrogate key
35
36 ## Convert back to DynamicFrame
37 dynamic_frame_out = DynamicFrame.fromDF(df_transformed, glueContext, "dynamic_frame_out")
38
39 ## Write to Redshift (dim_parkinglot table)
40 glueContext.write_dynamic_frame.from_jdbc_conf(
41 frame=dynamic_frame_out,
42 catalog_connection="redshift-connection-parkeasy", # Replace with your Redshift connection name
Python Ln 1, Col 1 Errors: 0 Warnings: 0
CloudShell Feedback
© 2024, Amazon Web Services, Inc. or its affiliates. Privacy Terms Cookie preferences

```
    connection_options={
        "dbtable": "dim_parkinglot",  # Target table in Redshift
        "database": "dev"             # Target database in Redshift
    },
    redshift_tmp_dir="s3://project-parkeasy-data-bucket/Temp/"  # Temporary directory in S3
)

job.commit()

```

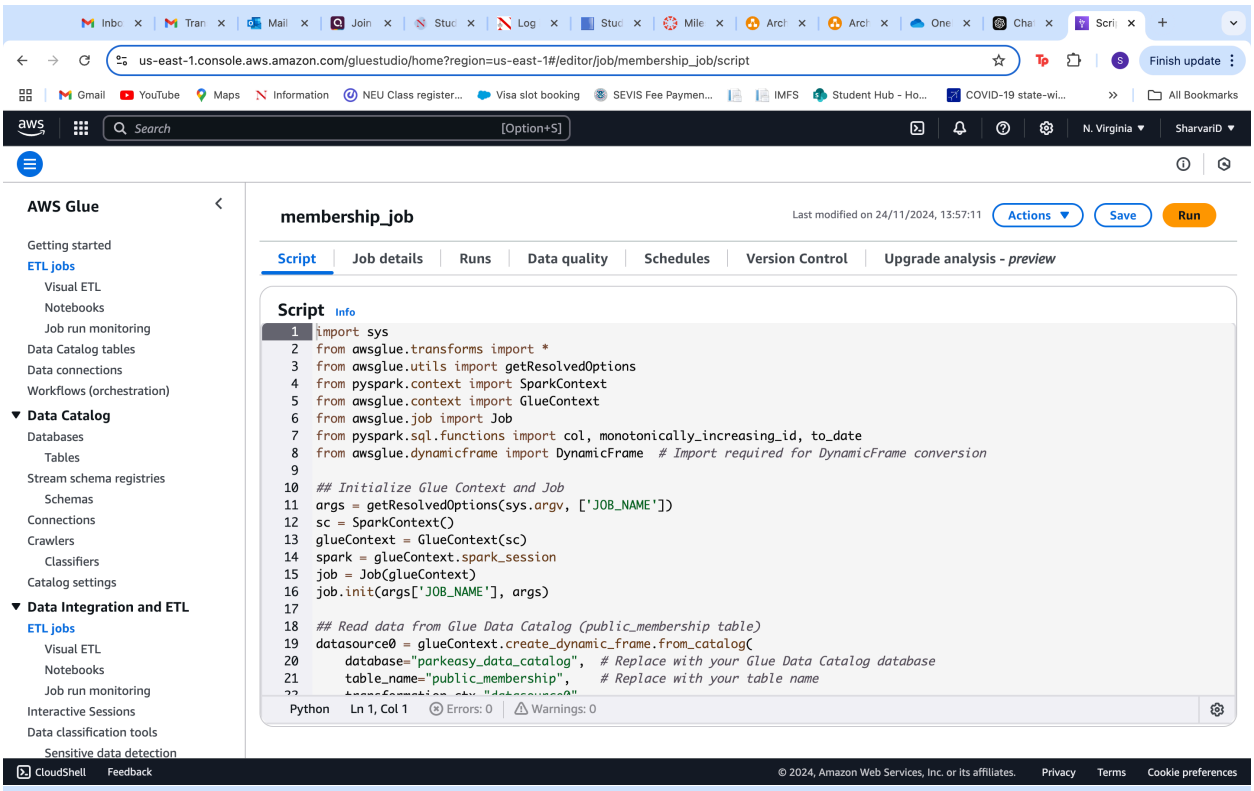
4.Time_job

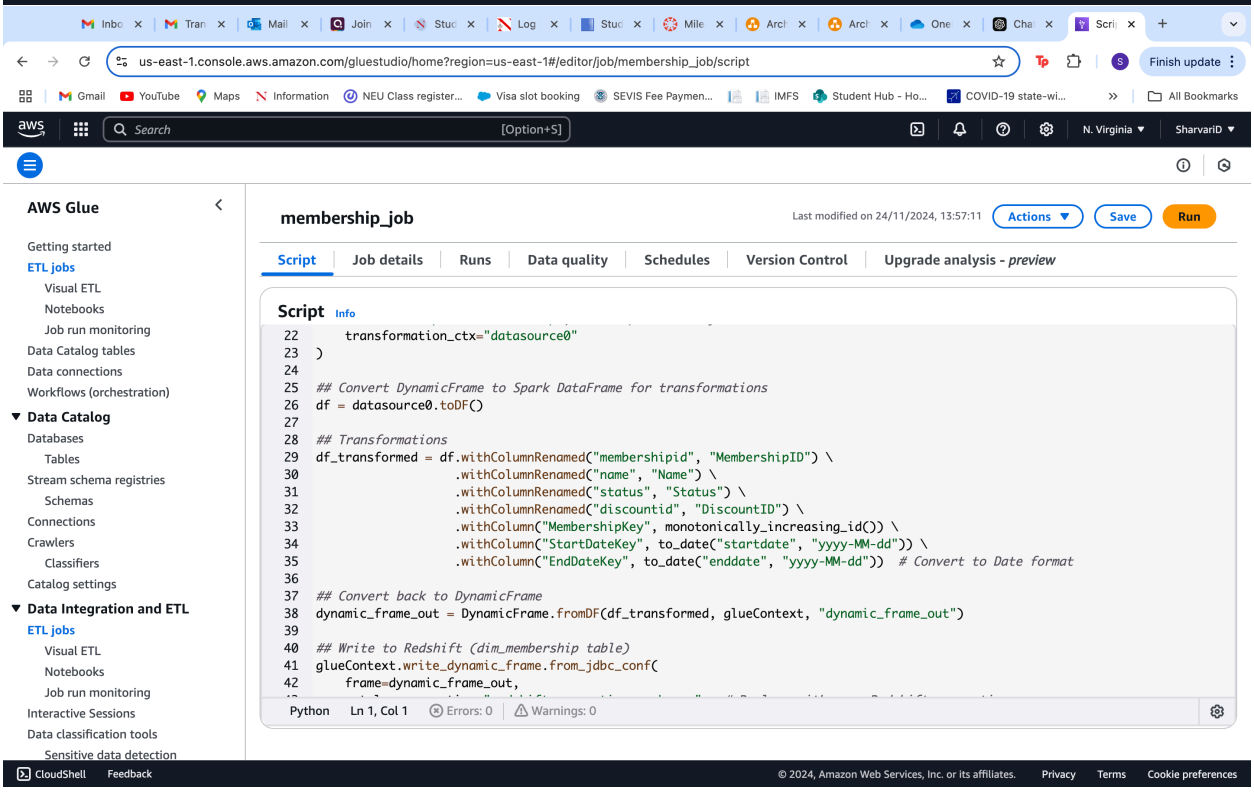





```
        "database": "dev"       # Target database in Redshift
    },
    redshift_tmp_dir="s3://project-parkeasy-data-bucket/Temp/"  # Replace with your S3 temporary directory
)

job.commit()
```

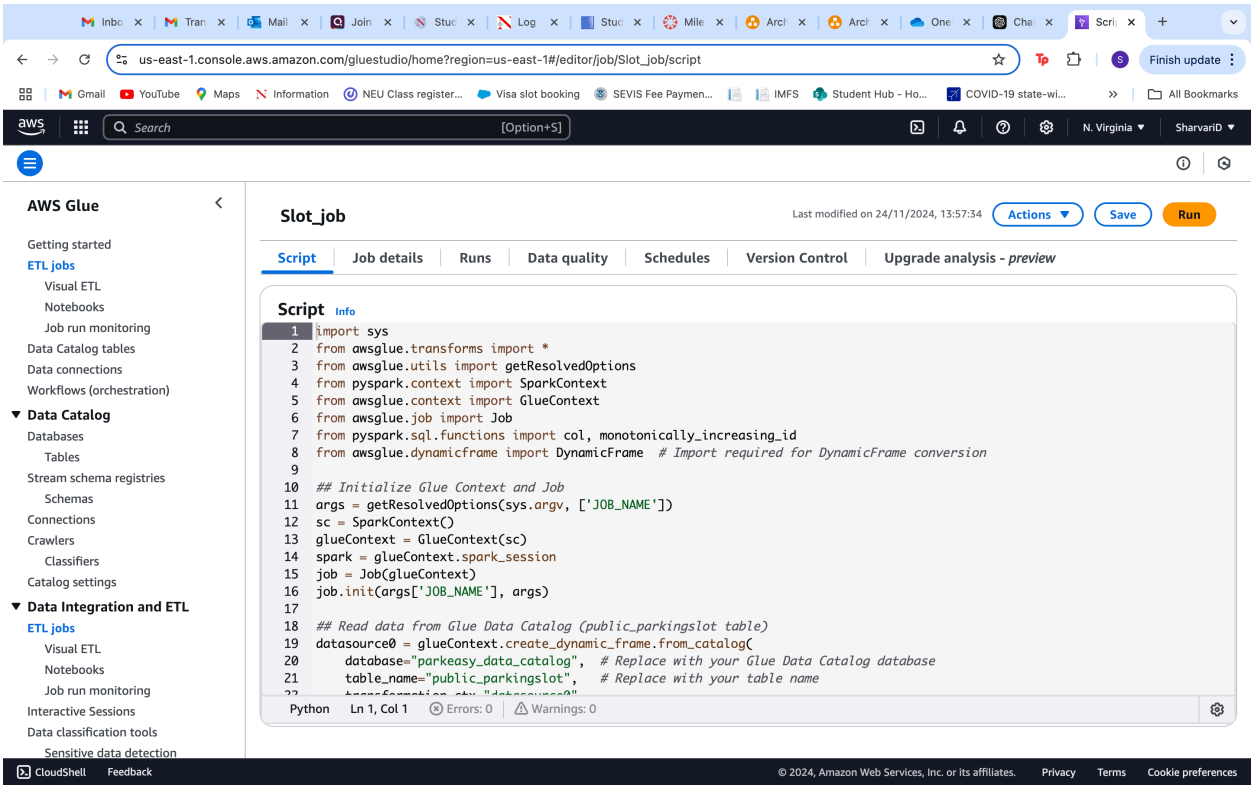
5. Membership_job

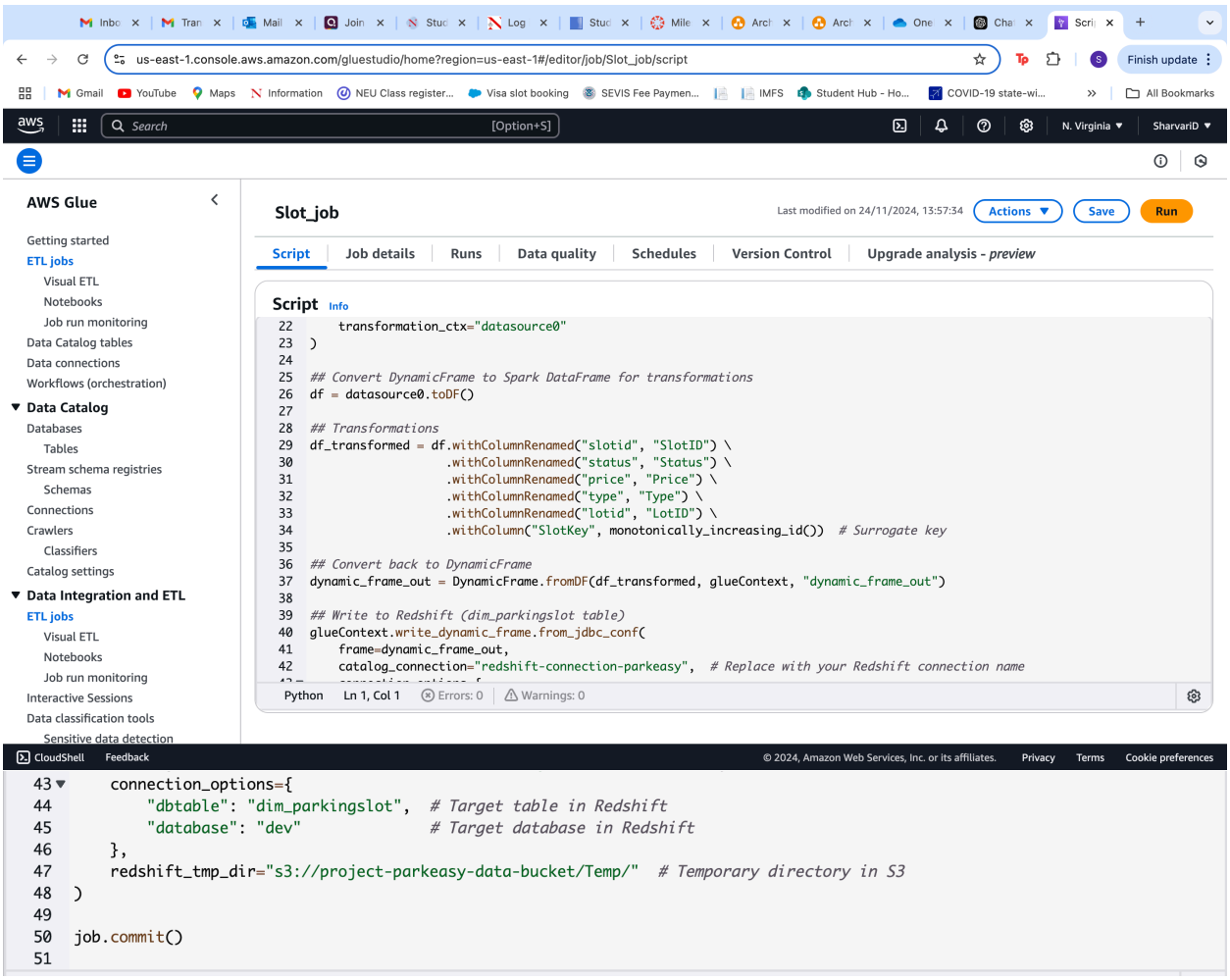

us-east-1.console.aws.amazon.com/gluestudio/home?region=us-east-1#/editor/job/membership_job/script
Finish update
Gmail
YouTube
Maps
Information
NEU Class register...
Visa slot booking
SEVIS Fee Paymen...
IMFS
Student Hub - Ho...
COVID-19 state-wi...
All Bookmarks
Search
[Option+S]
N. Virginia
SharvariD
AWS Glue
Getting started
ETL jobs
Visual ETL
Notebooks
Job run monitoring
Data Catalog tables
Data connections
Workflows (orchestration)
Data Catalog
Databases
Tables
Stream schema registries
Schemas
Connections
Crawlers
Classifiers
Catalog settings
Data Integration and ETL
ETL jobs
Visual ETL
Notebooks
Job run monitoring
Interactive Sessions
Data classification tools
Sensitive data detection
membership_job
Last modified on 24/11/2024, 13:57:11
Actions
Save
Run
Script
Job details
Runs
Data quality
Schedules
Version Control
Upgrade analysis - preview
Script Info
1 import sys
2 from awsglue.transforms import *
3 from awsglue.utils import getResolvedOptions
4 from pyspark.context import SparkContext
5 from awsglue.context import GlueContext
6 from awsglue.job import Job
7 from pyspark.sql.functions import col, monotonically_increasing_id, to_date
8 from awsglue.dynamicframe import DynamicFrame # Import required for DynamicFrame conversion
9
10 ## Initialize Glue Context and Job
11 args = getResolvedOptions(sys.argv, ['JOB_NAME'])
12 sc = SparkContext()
13 glueContext = GlueContext(sc)
14 spark = glueContext.spark_session
15 job = Job(glueContext)
16 job.init(args['JOB_NAME'], args)
17
18 ## Read data from Glue Data Catalog (public_membership table)
19 datasource0 = glueContext.create_dynamic_frame.from_catalog(
20 database="parkeasy_data_catalog", # Replace with your Glue Data Catalog database
21 table_name="public_membership", # Replace with your table name
Python
Ln 1, Col 1
Errors: 0
Warnings: 0
CloudShell
Feedback
© 2024, Amazon Web Services, Inc. or its affiliates.
Privacy
Terms
Cookie preferences


us-east-1.console.aws.amazon.com/gluestudio/home?region=us-east-1#/editor/job/membership_job/script
Finish update
Gmail
YouTube
Maps
Information
NEU Class register...
Visa slot booking
SEVIS Fee Paymen...
IMFS
Student Hub - Ho...
COVID-19 state-wi...
All Bookmarks
Search
[Option+S]
N. Virginia
SharvariD
AWS Glue
Getting started
ETL jobs
Visual ETL
Notebooks
Job run monitoring
Data Catalog tables
Data connections
Workflows (orchestration)
Data Catalog
Databases
Tables
Stream schema registries
Schemas
Connections
Crawlers
Classifiers
Catalog settings
Data Integration and ETL
ETL jobs
Visual ETL
Notebooks
Job run monitoring
Interactive Sessions
Data classification tools
Sensitive data detection
membership_job
Last modified on 24/11/2024, 13:57:11
Actions
Save
Run
Script
Job details
Runs
Data quality
Schedules
Version Control
Upgrade analysis - preview
Script Info
transformation_ctx="datasource0"
)
Convert DynamicFrame to Spark DataFrame for transformations
df = datasource0.toDF()
Transformations
df_transformed = df.withColumnRenamed("membershipid", "MembershipID") \
.withColumnRenamed("name", "Name") \
.withColumnRenamed("status", "Status") \
.withColumnRenamed("discountid", "DiscountID") \
.withColumn("MembershipKey", monotonically_increasing_id()) \
.withColumn("StartDateKey", to_date("startdate", "yyyy-MM-dd")) \
.withColumn("EndDateKey", to_date("enddate", "yyyy-MM-dd")) # Convert to Date format
Convert back to DynamicFrame
dynamic_frame_out = DynamicFrame.fromDF(df_transformed, glueContext, "dynamic_frame_out")
Write to Redshift (dim_membership table)
glueContext.write_dynamic_frame.from_jdbc_conf(
frame=dynamic_frame_out,
Python
Ln 1, Col 1
Errors: 0
Warnings: 0
CloudShell
Feedback
© 2024, Amazon Web Services, Inc. or its affiliates.
Privacy
Terms
Cookie preferences

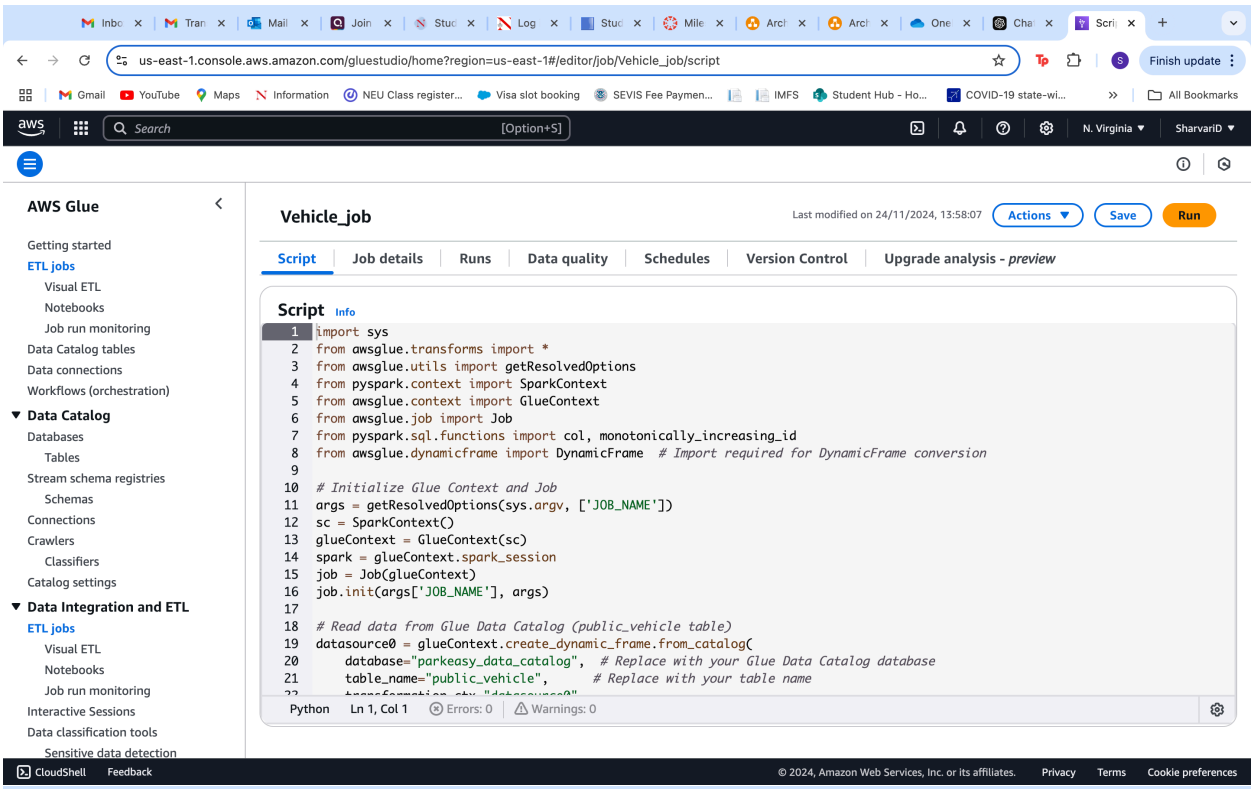
```
    catalog_connection="redshift-connection-parkeasy",  # Replace with your Redshift connection name
    connection_options={
        "dbtable": "dim_membership",  # Target table in Redshift
        "database": "dev"             # Target database in Redshift
    },
    redshift_tmp_dir="s3://project-parkeasy-data-bucket/Temp/"  # Temporary directory in S3
)

job.commit()

```

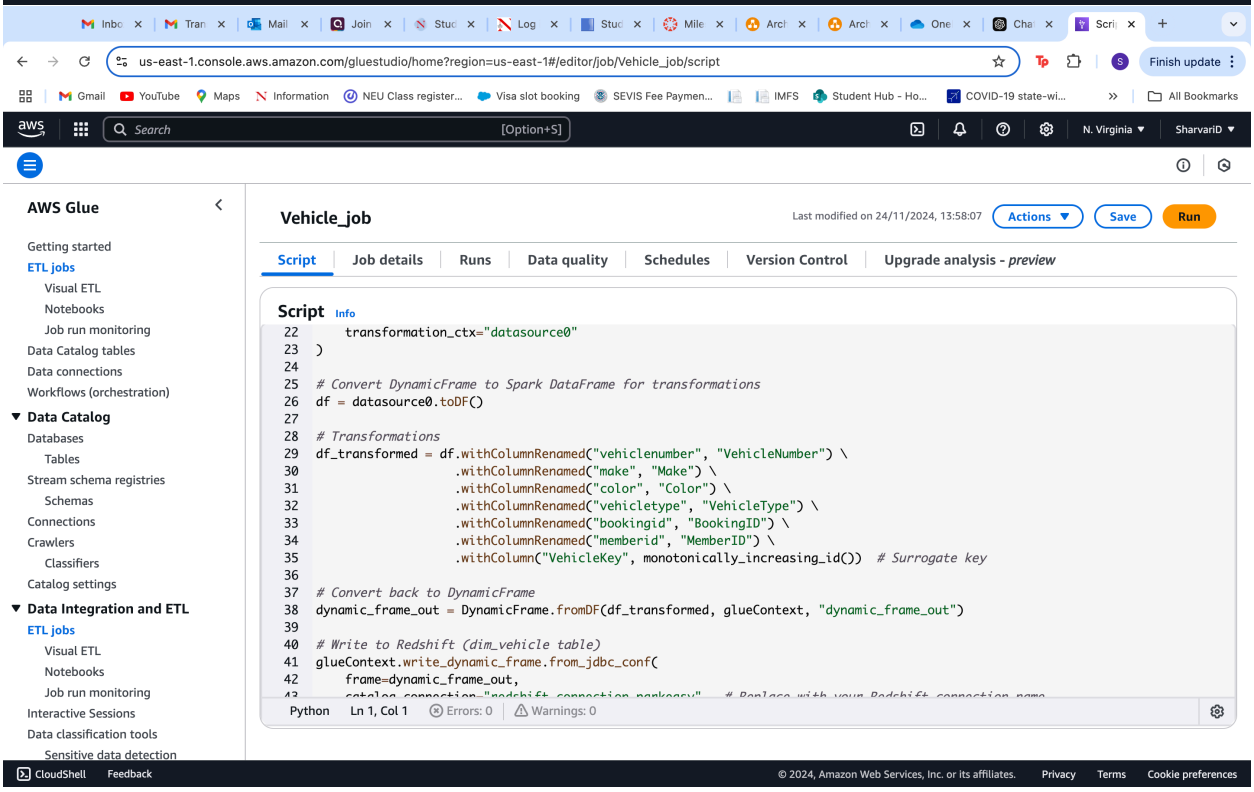
6. ParkingSlot_job





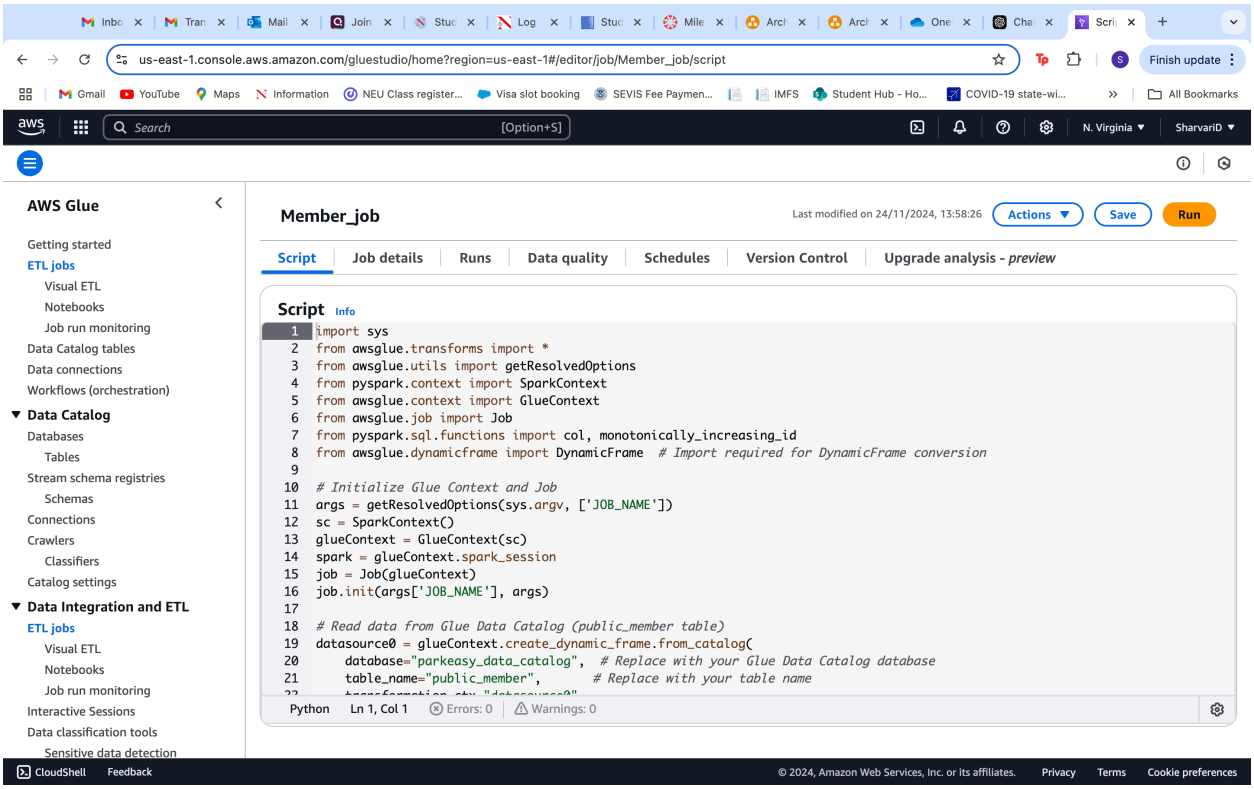
7. Vehicle_job

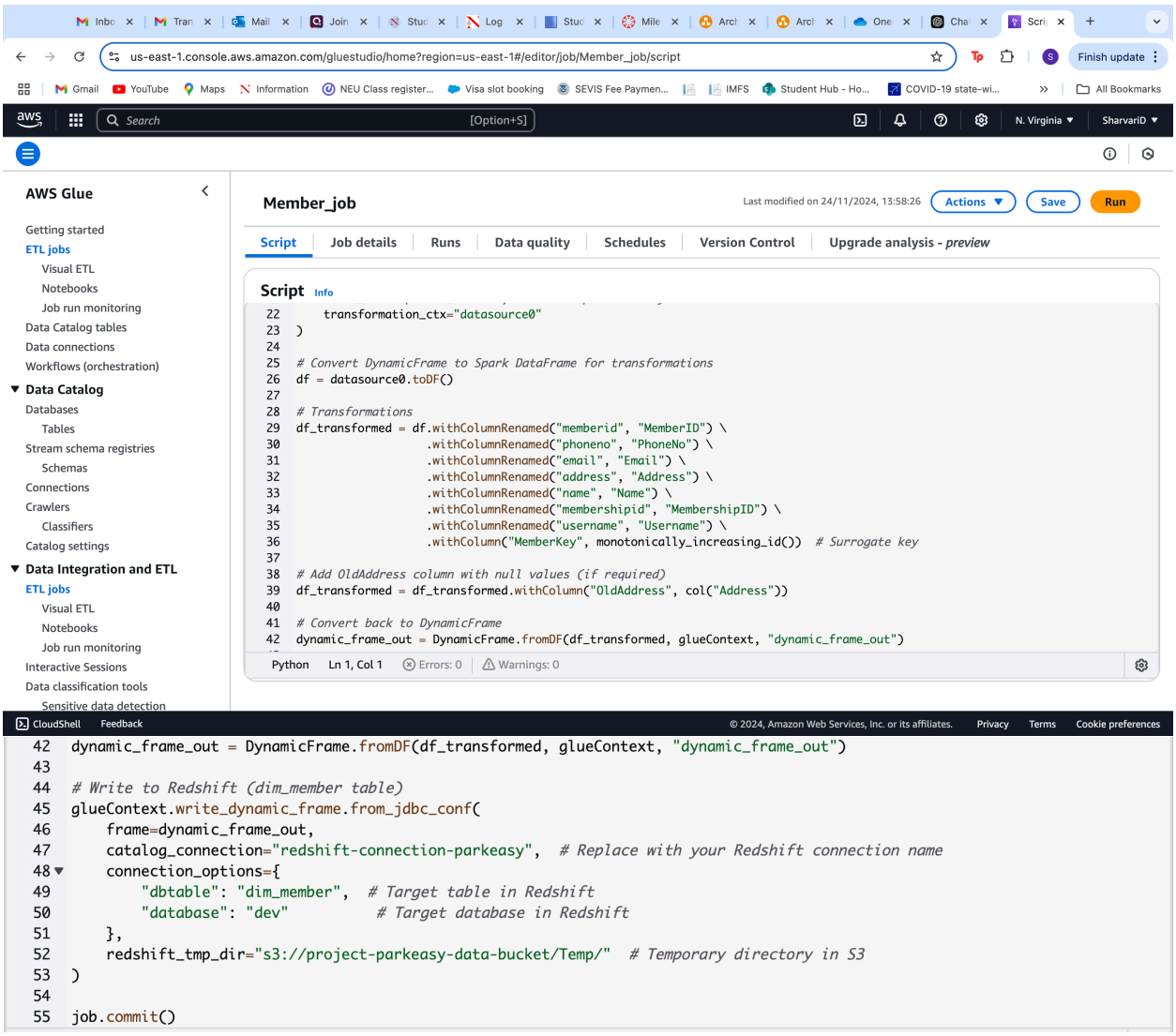

us-east-1.console.aws.amazon.com/gluestudio/home?region=us-east-1#/editor/job/Vehicle_job/script
Finish update
Gmail YouTube Maps Information NEU Class register... Visa slot booking SEVIS Fee Paymen... IMFS Student Hub - Ho... COVID-19 state-wi... All Bookmarks
aws Search [Option+S] N. Virginia SharvariD
AWS Glue
Getting started
ETL jobs
Visual ETL
Notebooks
Job run monitoring
Data Catalog tables
Data connections
Workflows (orchestration)
Data Catalog
Databases
Tables
Stream schema registries
Schemas
Connections
Crawlers
Classifiers
Catalog settings
Data Integration and ETL
ETL jobs
Visual ETL
Notebooks
Job run monitoring
Interactive Sessions
Data classification tools
Sensitive data detection
Vehicle_job
Last modified on 24/11/2024, 13:58:07
Actions
Save
Run
Script
Job details
Runs
Data quality
Schedules
Version Control
Upgrade analysis - preview
Script Info
1 import sys
2 from awsglue.transforms import *
3 from awsglue.utils import getResolvedOptions
4 from pyspark.context import SparkContext
5 from awsglue.context import GlueContext
6 from awsglue.job import Job
7 from pyspark.sql.functions import col, monotonically_increasing_id
8 from awsglue.dynamicframe import DynamicFrame # Import required for DynamicFrame conversion
9
10 # Initialize Glue Context and Job
11 args = getResolvedOptions(sys.argv, ['JOB_NAME'])
12 sc = SparkContext()
13 glueContext = GlueContext(sc)
14 spark = glueContext.spark_session
15 job = Job(glueContext)
16 job.init(args['JOB_NAME'], args)
17
18 # Read data from Glue Data Catalog (public_vehicle table)
19 datasource0 = glueContext.create_dynamic_frame.from_catalog(
20 database="parkeasy_data_catalog", # Replace with your Glue Data Catalog database
21 table_name="public_vehicle", # Replace with your table name
Python Ln 1, Col 1 Errors: 0 Warnings: 0
CloudShell Feedback
© 2024, Amazon Web Services, Inc. or its affiliates. Privacy Terms Cookie preferences


us-east-1.console.aws.amazon.com/gluestudio/home?region=us-east-1#/editor/job/Vehicle_job/script
Finish update
Gmail YouTube Maps Information NEU Class register... Visa slot booking SEVIS Fee Paymen... IMFS Student Hub - Ho... COVID-19 state-wi... All Bookmarks
aws Search [Option+S] N. Virginia SharvariD
AWS Glue
Getting started
ETL jobs
Visual ETL
Notebooks
Job run monitoring
Data Catalog tables
Data connections
Workflows (orchestration)
Data Catalog
Databases
Tables
Stream schema registries
Schemas
Connections
Crawlers
Classifiers
Catalog settings
Data Integration and ETL
ETL jobs
Visual ETL
Notebooks
Job run monitoring
Interactive Sessions
Data classification tools
Sensitive data detection
Vehicle_job
Last modified on 24/11/2024, 13:58:07
Actions
Save
Run
Script
Job details
Runs
Data quality
Schedules
Version Control
Upgrade analysis - preview
Script Info
22 transformation_ctx="datasource0"
23)
24
25 # Convert DynamicFrame to Spark DataFrame for transformations
26 df = datasource0.toDF()
27
28 # Transformations
29 df_transformed = df.withColumnRenamed("vehiclenumber", "VehicleNumber") \
30 .withColumnRenamed("make", "Make") \
31 .withColumnRenamed("color", "Color") \
32 .withColumnRenamed("vehicletype", "VehicleType") \
33 .withColumnRenamed("bookingid", "BookingID") \
34 .withColumnRenamed("memberid", "MemberID") \
35 .withColumn("VehicleKey", monotonically_increasing_id()) # Surrogate key
36
37 # Convert back to DynamicFrame
38 dynamic_frame_out = DynamicFrame.fromDF(df_transformed, glueContext, "dynamic_frame_out")
39
40 # Write to Redshift (dim_vehicle table)
41 glueContext.write_dynamic_frame.from_jdbc_conf(
42 frame=dynamic_frame_out,
Python Ln 1, Col 1 Errors: 0 Warnings: 0
CloudShell Feedback
© 2024, Amazon Web Services, Inc. or its affiliates. Privacy Terms Cookie preferences

```
    catalog_connection="redshift-connection-parkeasy",  # Replace with your Redshift connection name
    connection_options={
        "dbtable": "dim_vehicle",  # Target table in Redshift
        "database": "dev"          # Target database in Redshift
    },
    redshift_tmp_dir="s3://project-parkeasy-data-bucket/Temp/"  # Temporary directory in S3
)

job.commit()
```

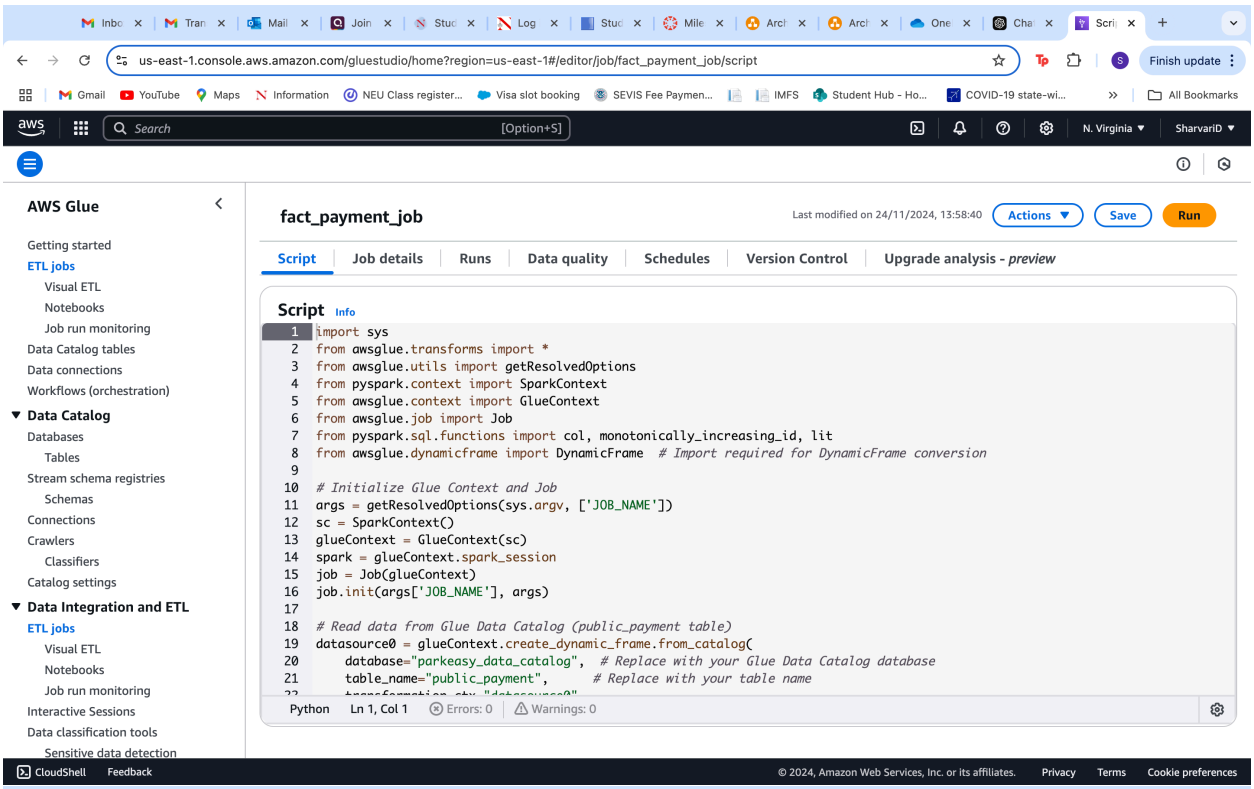
8. Member_job

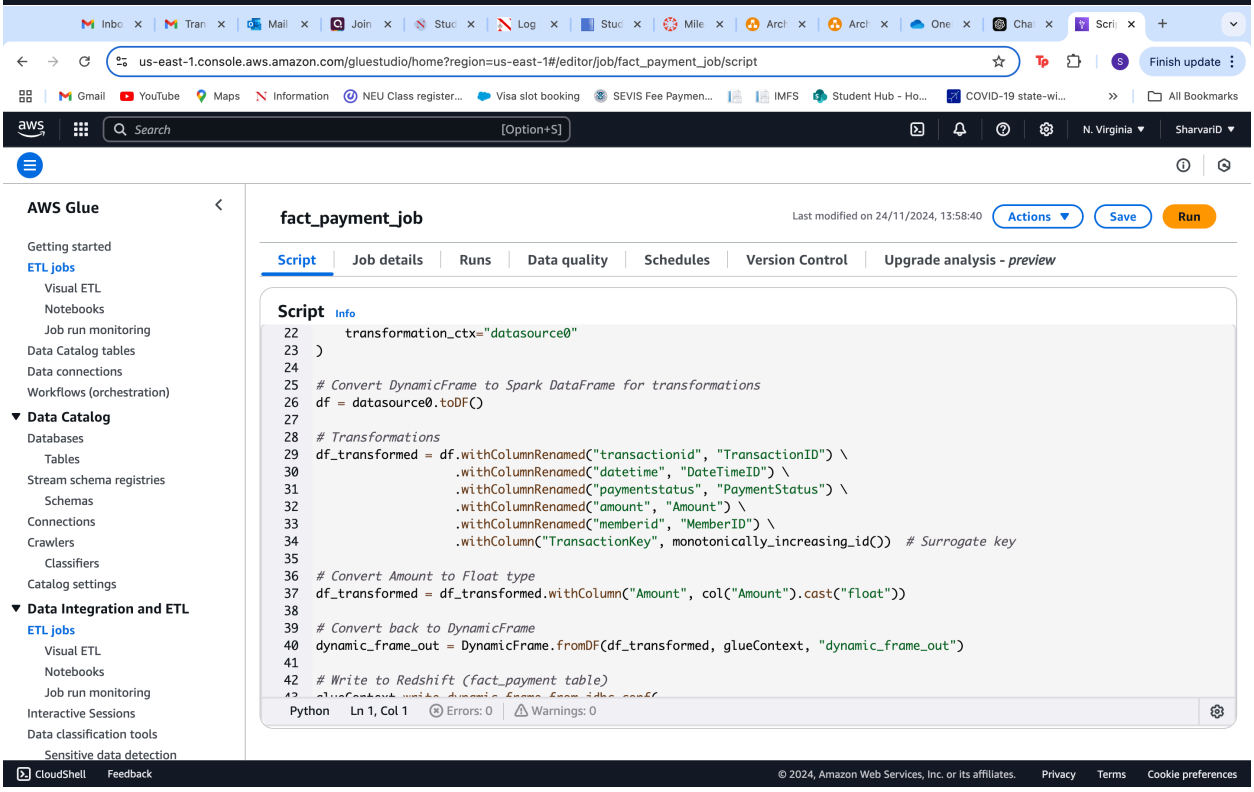





```python
dynamic_frame_out = DynamicFrame.fromDF(df_transformed, glueContext, "dynamic_frame_out")

# Write to Redshift (dim_member table)
glueContext.write_dynamic_frame.from_jdbc_conf(
    frame=dynamic_frame_out,
    catalog_connection="redshift-connection-parkeasy",  # Replace with your Redshift connection name
    connection_options={
        "dbtable": "dim_member",  # Target table in Redshift
        "database": "dev"          # Target database in Redshift
    },
    redshift_tmp_dir="s3://project-parkeasy-data-bucket/Temp/"  # Temporary directory in S3
)

job.commit()
```

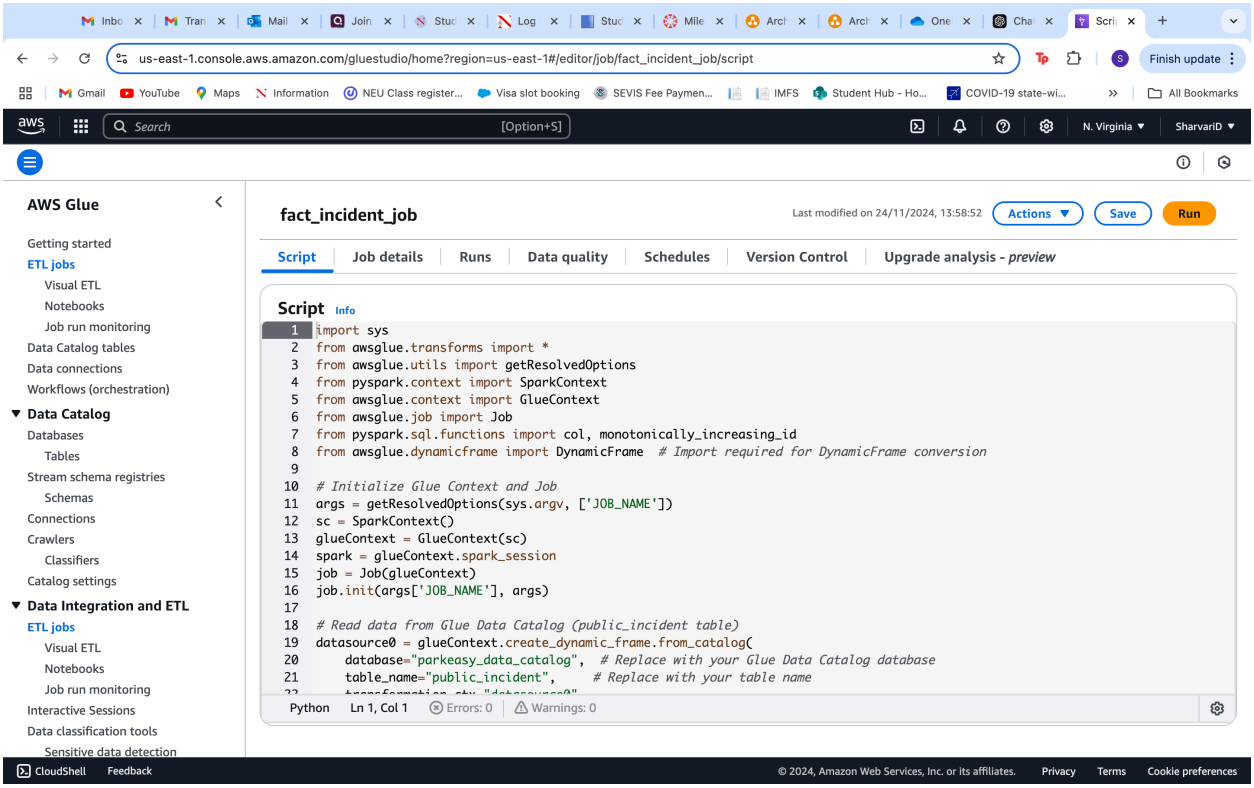
9. Fact_payment_job

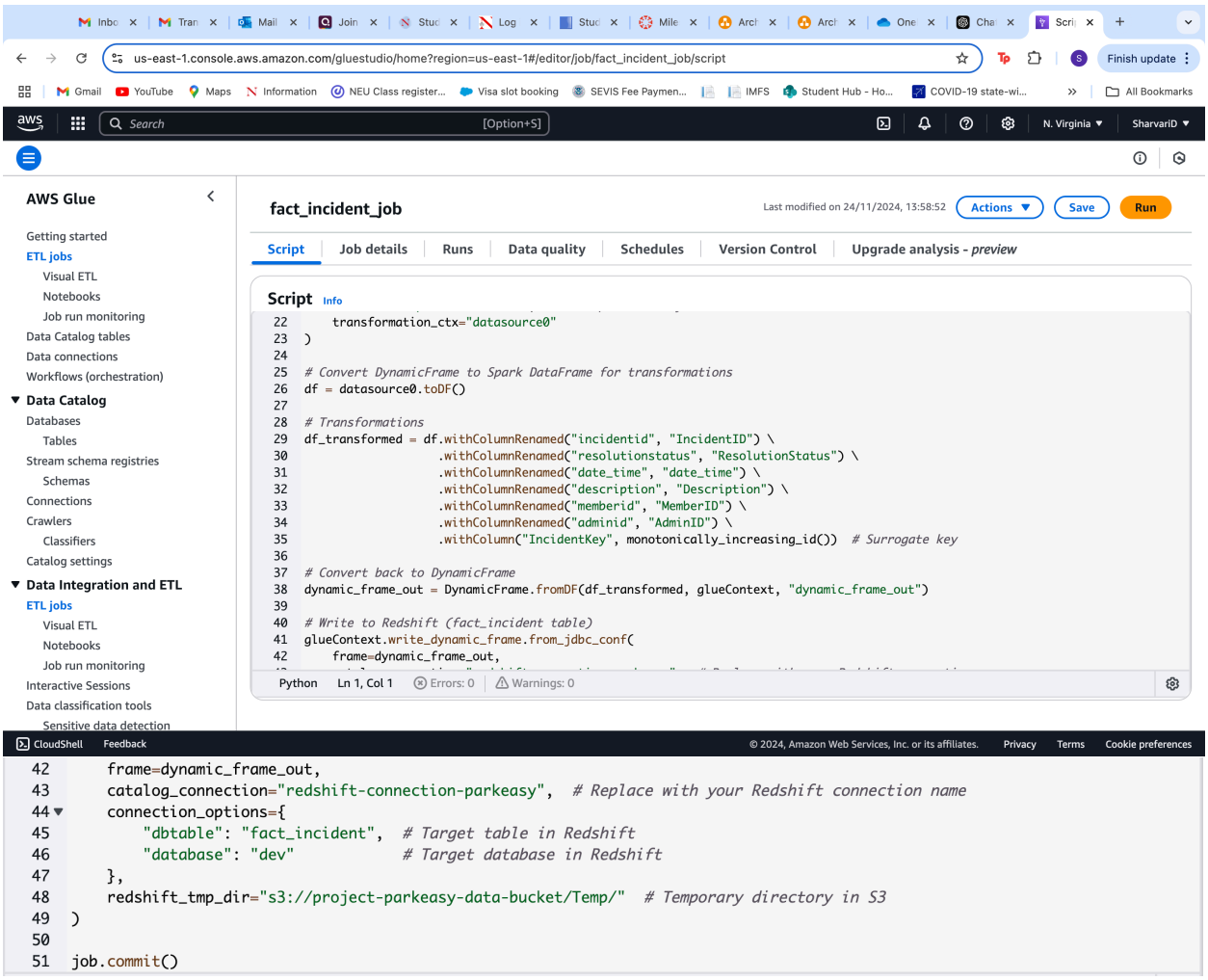

us-east-1.console.aws.amazon.com/gluestudio/home?region=us-east-1#/editor/job/fact_payment_job/script
Finish update
Gmail
YouTube
Maps
Information
NEU Class register...
Visa slot booking
SEVIS Fee Paymen...
IMFS
Student Hub - Ho...
COVID-19 state-wi...
All Bookmarks
Search
[Option+S]
N. Virginia
SharvariD
AWS Glue
Getting started
ETL jobs
Visual ETL
Notebooks
Job run monitoring
Data Catalog tables
Data connections
Workflows (orchestration)
Data Catalog
Databases
Tables
Stream schema registries
Schemas
Connections
Crawlers
Classifiers
Catalog settings
Data Integration and ETL
ETL jobs
Visual ETL
Notebooks
Job run monitoring
Interactive Sessions
Data classification tools
Sensitive data detection
fact_payment_job
Last modified on 24/11/2024, 13:58:40
Actions
Save
Run
Script
Job details
Runs
Data quality
Schedules
Version Control
Upgrade analysis - preview
Script Info
1 import sys
2 from awsglue.transforms import *
3 from awsglue.utils import getResolvedOptions
4 from pyspark.context import SparkContext
5 from awsglue.context import GlueContext
6 from awsglue.job import Job
7 from pyspark.sql.functions import col, monotonically_increasing_id, lit
8 from awsglue.dynamicframe import DynamicFrame # Import required for DynamicFrame conversion
9
10 # Initialize Glue Context and Job
11 args = getResolvedOptions(sys.argv, ['JOB_NAME'])
12 sc = SparkContext()
13 glueContext = GlueContext(sc)
14 spark = glueContext.spark_session
15 job = Job(glueContext)
16 job.init(args['JOB_NAME'], args)
17
18 # Read data from Glue Data Catalog (public_payment table)
19 datasource0 = glueContext.create_dynamic_frame.from_catalog(
20 database="parkeasy_data_catalog", # Replace with your Glue Data Catalog database
21 table_name="public_payment", # Replace with your table name
Python
Ln 1, Col 1
Errors: 0
Warnings: 0
CloudShell
Feedback
© 2024, Amazon Web Services, Inc. or its affiliates.
Privacy
Terms
Cookie preferences


us-east-1.console.aws.amazon.com/gluestudio/home?region=us-east-1#/editor/job/fact_payment_job/script
Finish update
Gmail
YouTube
Maps
Information
NEU Class register...
Visa slot booking
SEVIS Fee Paymen...
IMFS
Student Hub - Ho...
COVID-19 state-wi...
All Bookmarks
Search
[Option+S]
N. Virginia
SharvariD
AWS Glue
Getting started
ETL jobs
Visual ETL
Notebooks
Job run monitoring
Data Catalog tables
Data connections
Workflows (orchestration)
Data Catalog
Databases
Tables
Stream schema registries
Schemas
Connections
Crawlers
Classifiers
Catalog settings
Data Integration and ETL
ETL jobs
Visual ETL
Notebooks
Job run monitoring
Interactive Sessions
Data classification tools
Sensitive data detection
fact_payment_job
Last modified on 24/11/2024, 13:58:40
Actions
Save
Run
Script
Job details
Runs
Data quality
Schedules
Version Control
Upgrade analysis - preview
Script Info
22 transformation_ctx="datasource0"
23)
24
25 # Convert DynamicFrame to Spark DataFrame for transformations
26 df = datasource0.toDF()
27
28 # Transformations
29 df_transformed = df.withColumnRenamed("transactionid", "TransactionID") \
30 .withColumnRenamed("datetime", "DateTimeID") \
31 .withColumnRenamed("paymentstatus", "PaymentStatus") \
32 .withColumnRenamed("amount", "Amount") \
33 .withColumnRenamed("memberid", "MemberID") \
34 .withColumn("TransactionKey", monotonically_increasing_id()) # Surrogate key
35
36 # Convert Amount to Float type
37 df_transformed = df_transformed.withColumn("Amount", col("Amount").cast("float"))
38
39 # Convert back to DynamicFrame
40 dynamic_frame_out = DynamicFrame.fromDF(df_transformed, glueContext, "dynamic_frame_out")
41
42 # Write to Redshift (fact_payment table)
Python
Ln 1, Col 1
Errors: 0
Warnings: 0
CloudShell
Feedback
© 2024, Amazon Web Services, Inc. or its affiliates.
Privacy
Terms
Cookie preferences

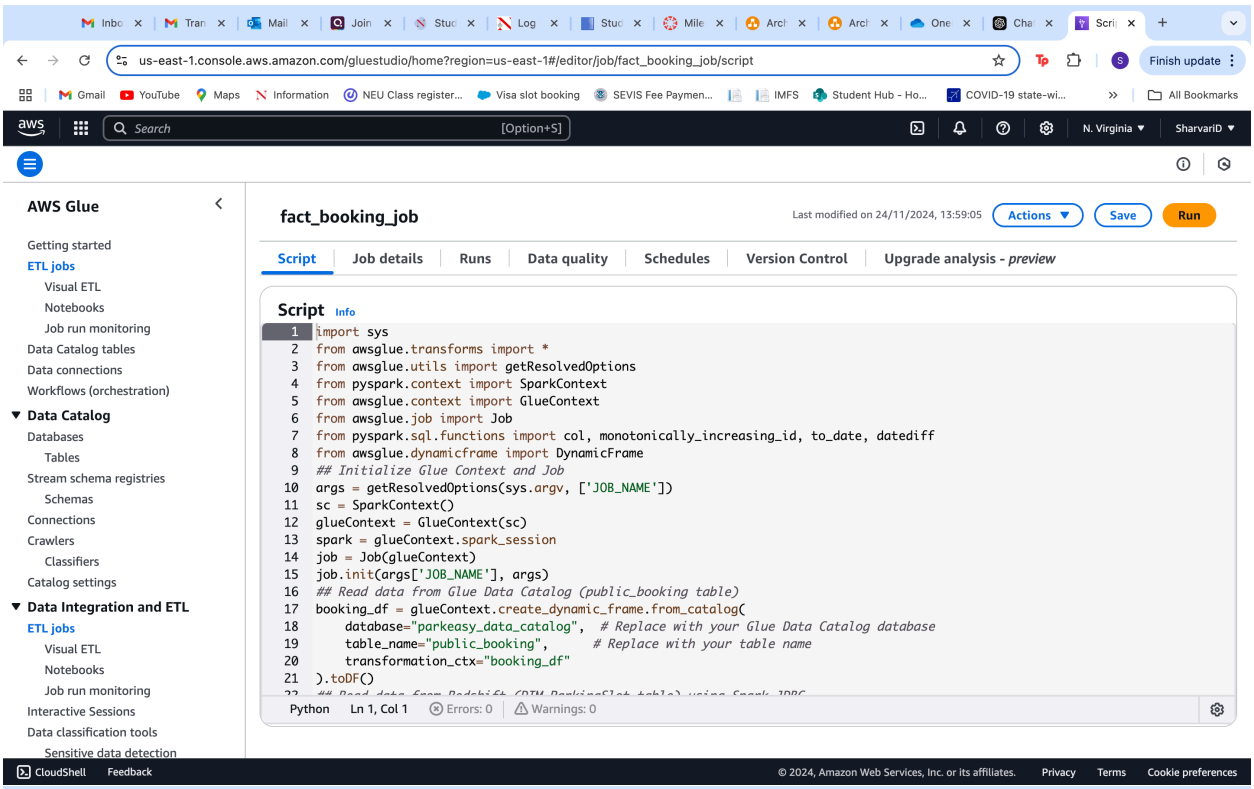
```python
# Write to Redshift (fact_payment table)
glueContext.write_dynamic_frame.from_jdbc_conf(
    frame=dynamic_frame_out,
    catalog_connection="redshift-connection-parkeasy",  # Replace with your Redshift connection name
    connection_options={
        "dbtable": "fact_payment",  # Target table in Redshift
        "database": "dev"           # Target database in Redshift
    },
    redshift_tmp_dir="s3://project-parkeasy-data-bucket/Temp/"  # Temporary directory in S3
)

job.commit()
```

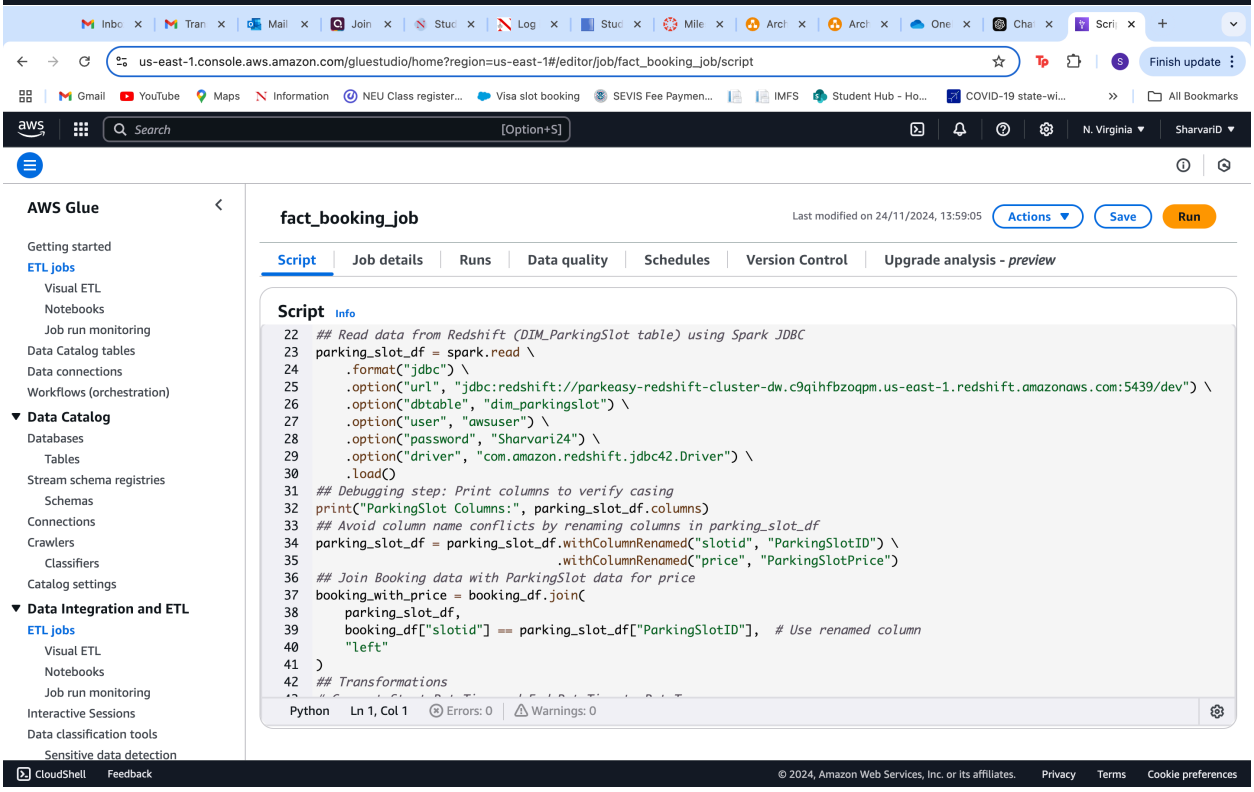
10. Fact_incident_job

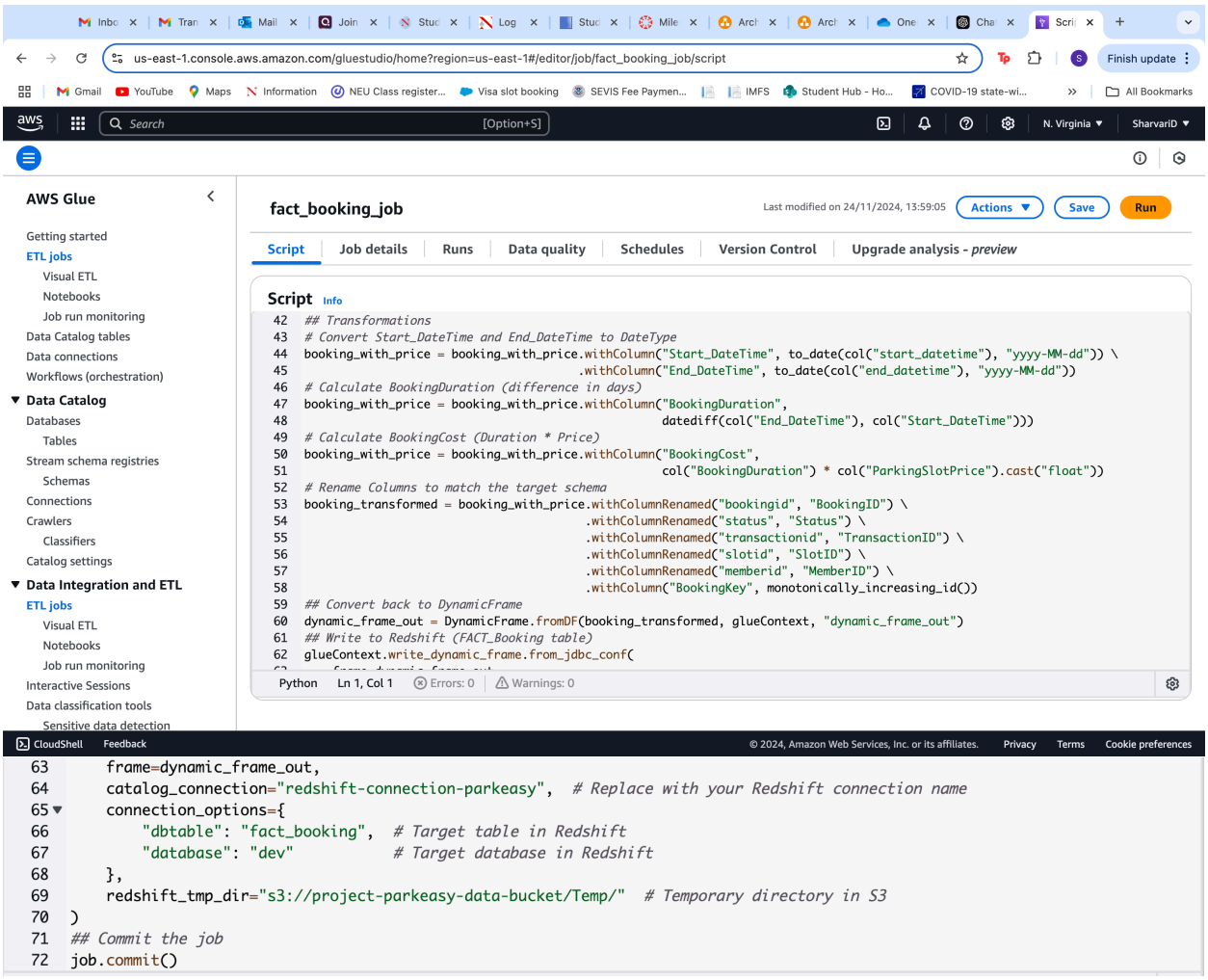




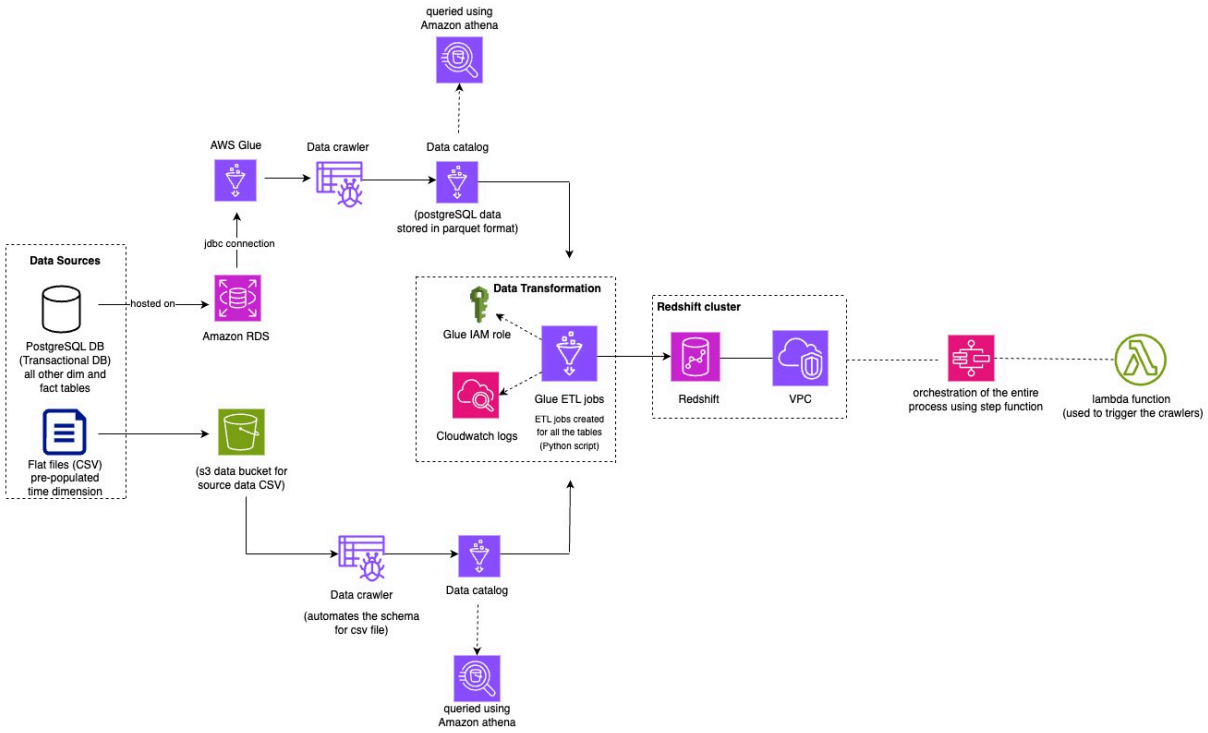
11. Fact_booking_job


us-east-1.console.aws.amazon.com/gluestudio/home?region=us-east-1#/editor/job/fact_booking_job/script
AWS Glue
fact_booking_job
Last modified on 24/11/2024, 13:59:05
Actions
Save
Run
Script
Job details
Runs
Data quality
Schedules
Version Control
Upgrade analysis - preview
Script Info
1 import sys
2 from awsglue.transforms import *
3 from awsglue.utils import getResolvedOptions
4 from pyspark.context import SparkContext
5 from awsglue.context import GlueContext
6 from awsglue.job import Job
7 from pyspark.sql.functions import col, monotonically_increasing_id, to_date, datediff
8 from awsglue.dynamicframe import DynamicFrame
9 ## Initialize Glue Context and Job
10 args = getResolvedOptions(sys.argv, ['JOB_NAME'])
11 sc = SparkContext()
12 glueContext = GlueContext(sc)
13 spark = glueContext.spark_session
14 job = Job(glueContext)
15 job.init(args['JOB_NAME'], args)
16 ## Read data from Glue Data Catalog (public_booking table)
17 booking_df = glueContext.create_dynamic_frame.from_catalog(
18 database="parkeasy_data_catalog", # Replace with your Glue Data Catalog database
19 table_name="public_booking", # Replace with your table name
20 transformation_ctx="booking_df"
21).toDF()
Python
Ln 1, Col 1
Errors: 0
Warnings: 0
CloudShell
Feedback
© 2024, Amazon Web Services, Inc. or its affiliates.
Privacy
Terms
Cookie preferences


us-east-1.console.aws.amazon.com/gluestudio/home?region=us-east-1#/editor/job/fact_booking_job/script
AWS Glue
fact_booking_job
Last modified on 24/11/2024, 13:59:05
Actions
Save
Run
Script
Job details
Runs
Data quality
Schedules
Version Control
Upgrade analysis - preview
Script Info
22 ## Read data from Redshift (DIM_ParkingSlot table) using Spark JDBC
23 parking_slot_df = spark.read \
24 .format("jdbc") \
25 .option("url", "jdbc:redshift://parkeasy-redshift-cluster-dw.c9qihfbzoqpm.us-east-1.redshift.amazonaws.com:5439/dev") \
26 .option("dbtable", "dim_parkingslot") \
27 .option("user", "awsuser") \
28 .option("password", "Sharvari24") \
29 .option("driver", "com.amazon.redshift.jdbc42.Driver") \
30 .load()
31 ## Debugging step: Print columns to verify casing
32 print("ParkingSlot Columns:", parking_slot_df.columns)
33 ## Avoid column name conflicts by renaming columns in parking_slot_df
34 parking_slot_df = parking_slot_df.withColumnRenamed("slotid", "ParkingSlotID") \
35 .withColumnRenamed("price", "ParkingSlotPrice")
36 ## Join Booking data with ParkingSlot data for price
37 booking_with_price = booking_df.join(
38 parking_slot_df,
39 booking_df["slotid"] == parking_slot_df["ParkingSlotID"], # Use renamed column
40 "left"
41)
42 ## Transformations
Python
Ln 1, Col 1
Errors: 0
Warnings: 0
CloudShell
Feedback
© 2024, Amazon Web Services, Inc. or its affiliates.
Privacy
Terms
Cookie preferences



11. Architectural Diagram of the Data Pipeline : Parkeasy Parking Management System



1. Data Sources

The pipeline begins with two primary data sources:

1. PostgreSQL Database:
    - Hosted on Amazon RDS, this transactional database contains the source tables for all dimension and fact data.
    - It serves as the main repository for operational data, including information about users, memberships, vehicles, parking lots, and discounts.
2. Flat Files (CSV Format):
    - Stored in an Amazon S3 bucket, these files include pre-populated dimension tables such as the time dimension.
    - These files complement the transactional data by providing auxiliary information for analysis.

2. Data Ingestion

Data from the above sources is ingested into the pipeline using two distinct methods:

1. JDBC Connection to Amazon RDS:
    - AWS Glue connects to the PostgreSQL database on Amazon RDS using a secure JDBC connection.
    - This ensures seamless access to transactional data for ETL jobs.
2. S3 Bucket Integration:
    - CSV files in the S3 bucket are directly accessed by AWS Glue crawlers to automate schema discovery and load data into the pipeline.

3. AWS Glue Data Crawlers

AWS Glue Data Crawlers automate the schema discovery process for both data sources:

1. PostgreSQL Data:
    - Crawlers identify the structure and schema of tables in the RDS-hosted PostgreSQL database.
    - The data is cataloged in the AWS Glue Data Catalog for subsequent processing.
2. Flat Files in S3:
    - Crawlers analyze the CSV files in the S3 bucket to infer their schema and catalog them in the Glue Data Catalog.

Key Outcome:

- The Glue Data Catalog stores metadata for all tables and files, acting as a centralized repository for schema and data definitions.

4. AWS Glue ETL Jobs

AWS Glue ETL jobs transform the raw data into a format suitable for analysis. These jobs are Python-based and use PySpark for distributed processing. Key transformations include:

1. Schema Standardization:
    - Columns are renamed to adhere to consistent naming conventions across all tables.
2. Data Enrichment:
    - Surrogate keys are added to enhance data integrity.
    - Data types are cast to appropriate formats (e.g., percentages are converted to floats).
3. Business Logic Implementation:
    - Custom transformations such as date parsing and conditional logic are applied to meet analytical requirements.

Output:

- The transformed data is prepared for loading into Amazon Redshift.

5. CloudWatch Logs

AWS Glue jobs are integrated with Amazon CloudWatch Logs to provide real-time monitoring and logging of job executions. Key functionalities include:

- Job Monitoring: Tracks the progress and status of ETL jobs.
- Error Logging: Captures errors and exceptions for debugging purposes.
- Performance Metrics: Records resource utilization and execution times to optimize job performance.

6. Amazon Redshift

The transformed data is loaded into Amazon Redshift, a fully managed data warehouse optimized for analytical queries. Key steps include:

1. Data Loading:
    - AWS Glue writes the processed data into Redshift tables using JDBC connections.
    - Fact and dimension tables such as dim_admin, dim_vehicle, and dim_membership are populated.
2. Data Storage:

    - Redshift organizes data using columnar storage to enhance query performance.
3. Data Analysis:
    - Redshift supports complex SQL queries, enabling analysts to generate reports and insights.

7. Orchestration Using AWS Step Functions

AWS Step Functions orchestrate the pipeline by managing dependencies and automating the sequence of tasks. Key functionalities include:

- Workflow Management:
    - Step Functions define the order of operations, such as running data crawlers, executing Glue ETL jobs, and loading data into Redshift.
- Error Handling:
    - Built-in retry mechanisms handle job failures, ensuring the pipeline executes reliably.
- Integration:
    - Step Functions interact seamlessly with Glue, Redshift, and other AWS services to streamline the entire process.

A lambda function was created for the same which was used to trigger the crawlers to automated pulling the data

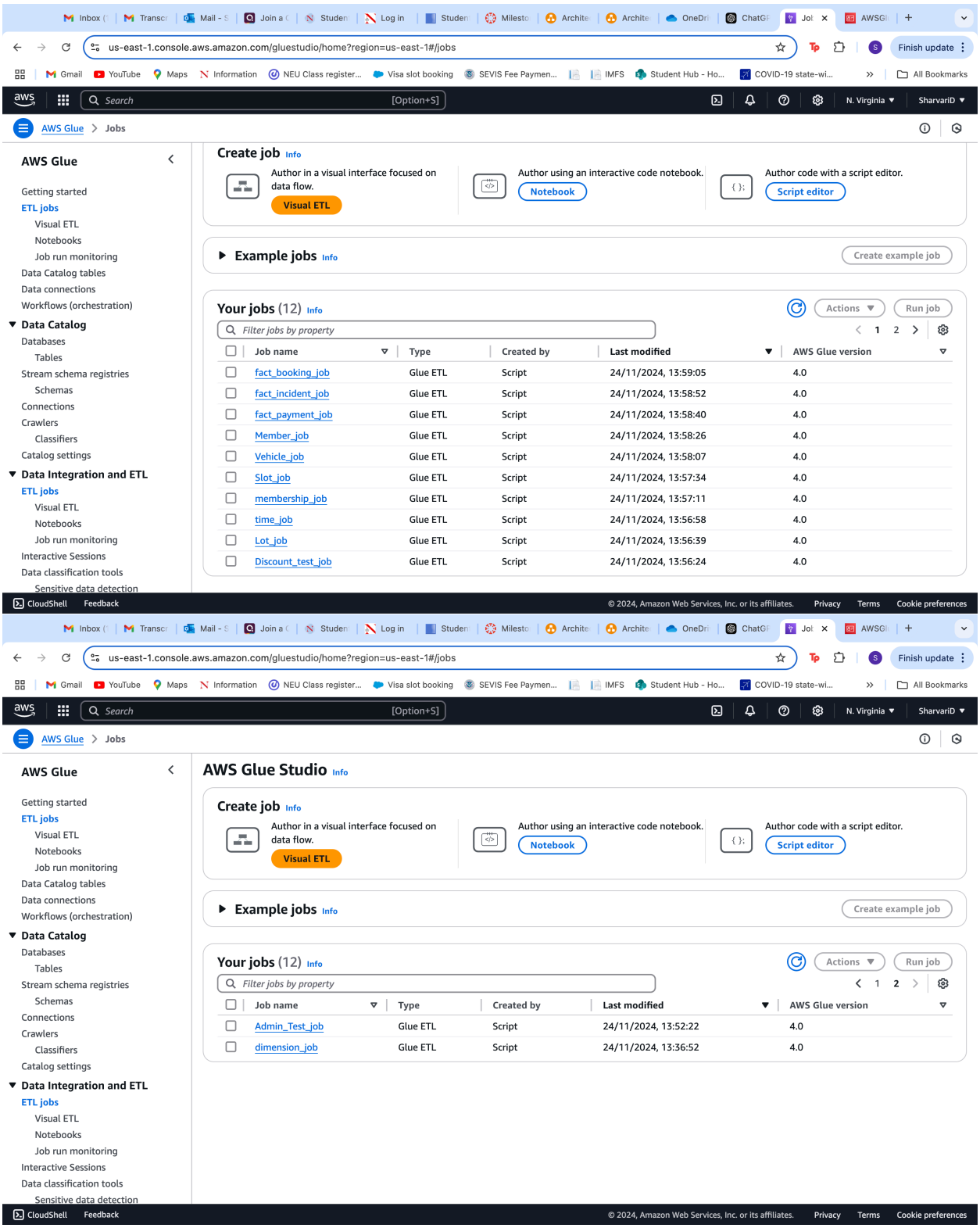
8. Data Querying with Amazon Athena

The processed data in Redshift and the Glue Data Catalog can be queried using Amazon Athena:

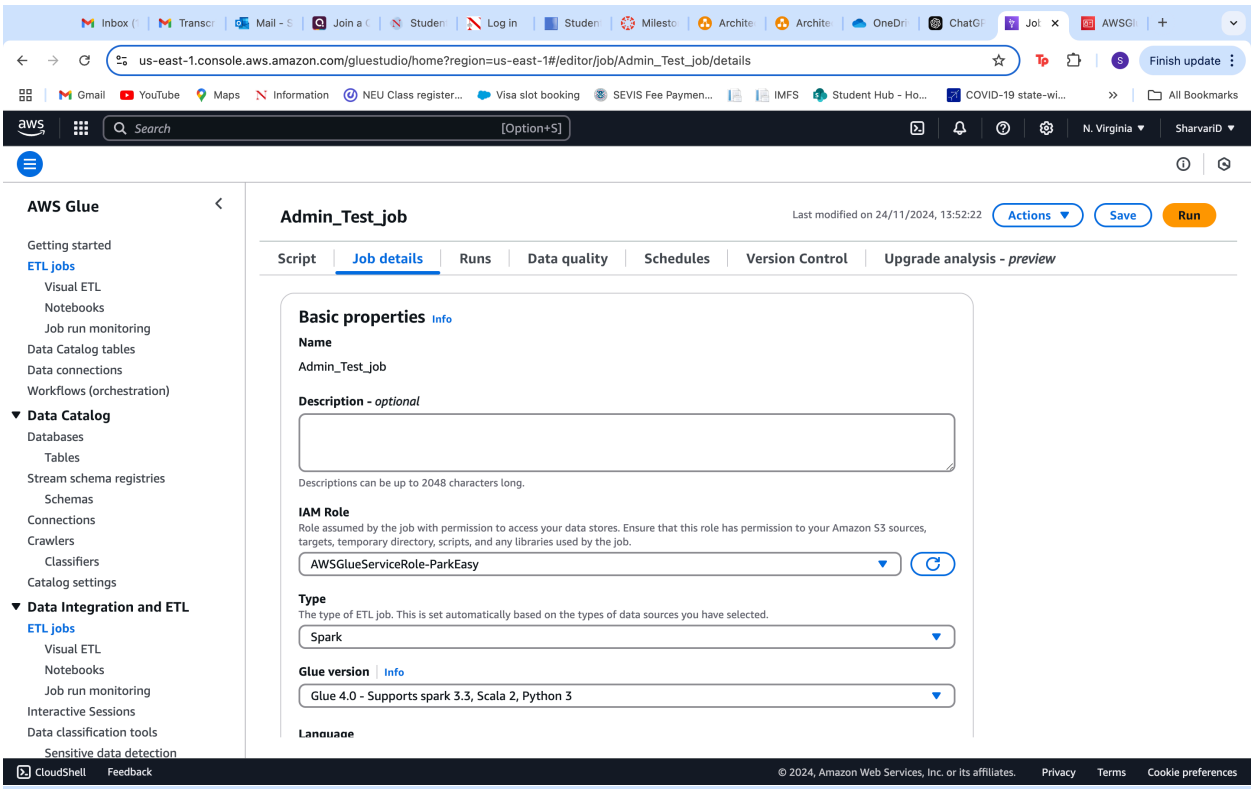
1. Querying Transformed Data:
    - Analysts use Athena to execute ad hoc SQL queries on the cataloged data stored in S3 or Redshift.
2. Integration with Glue Data Catalog:
    - Athena leverages the Glue Data Catalog to retrieve metadata and schema information for efficient query execution.
3. Insights Generation:
    - Results from Athena queries provide actionable insights for decision-making.

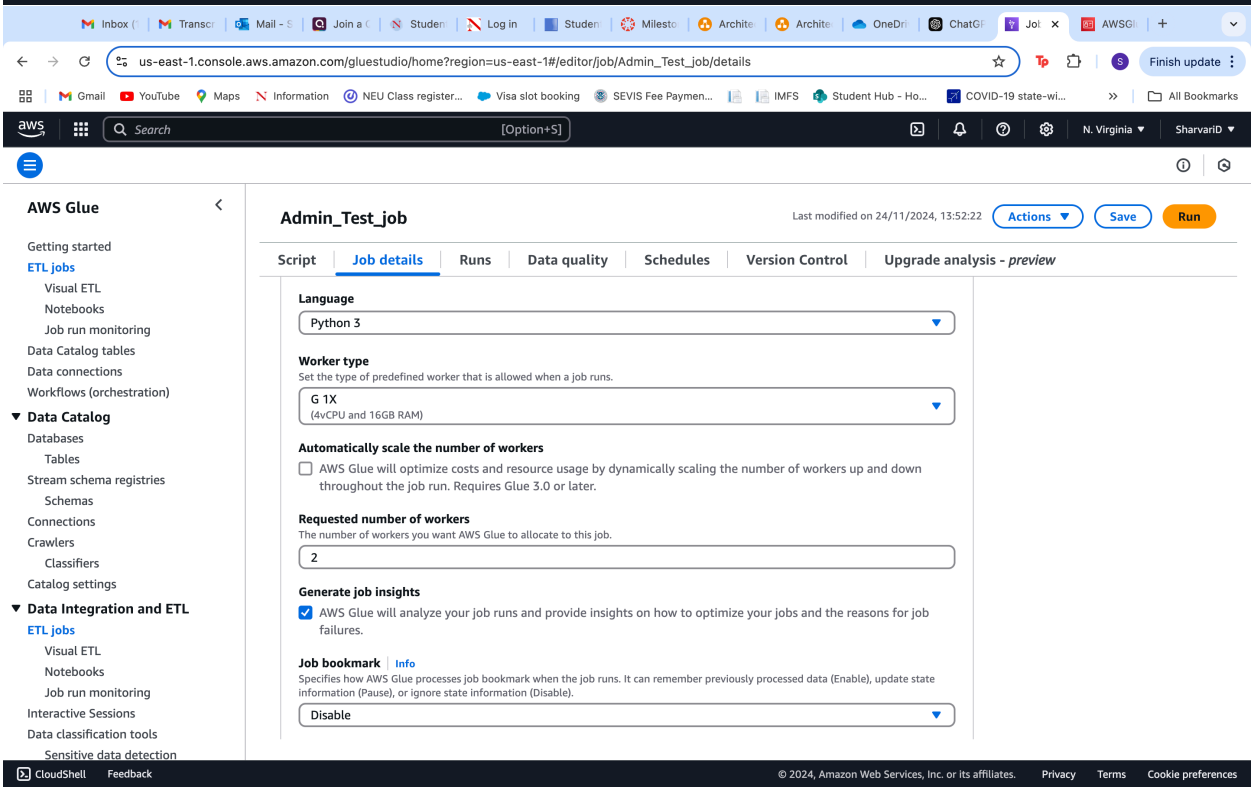
12. Pipeline Execution:

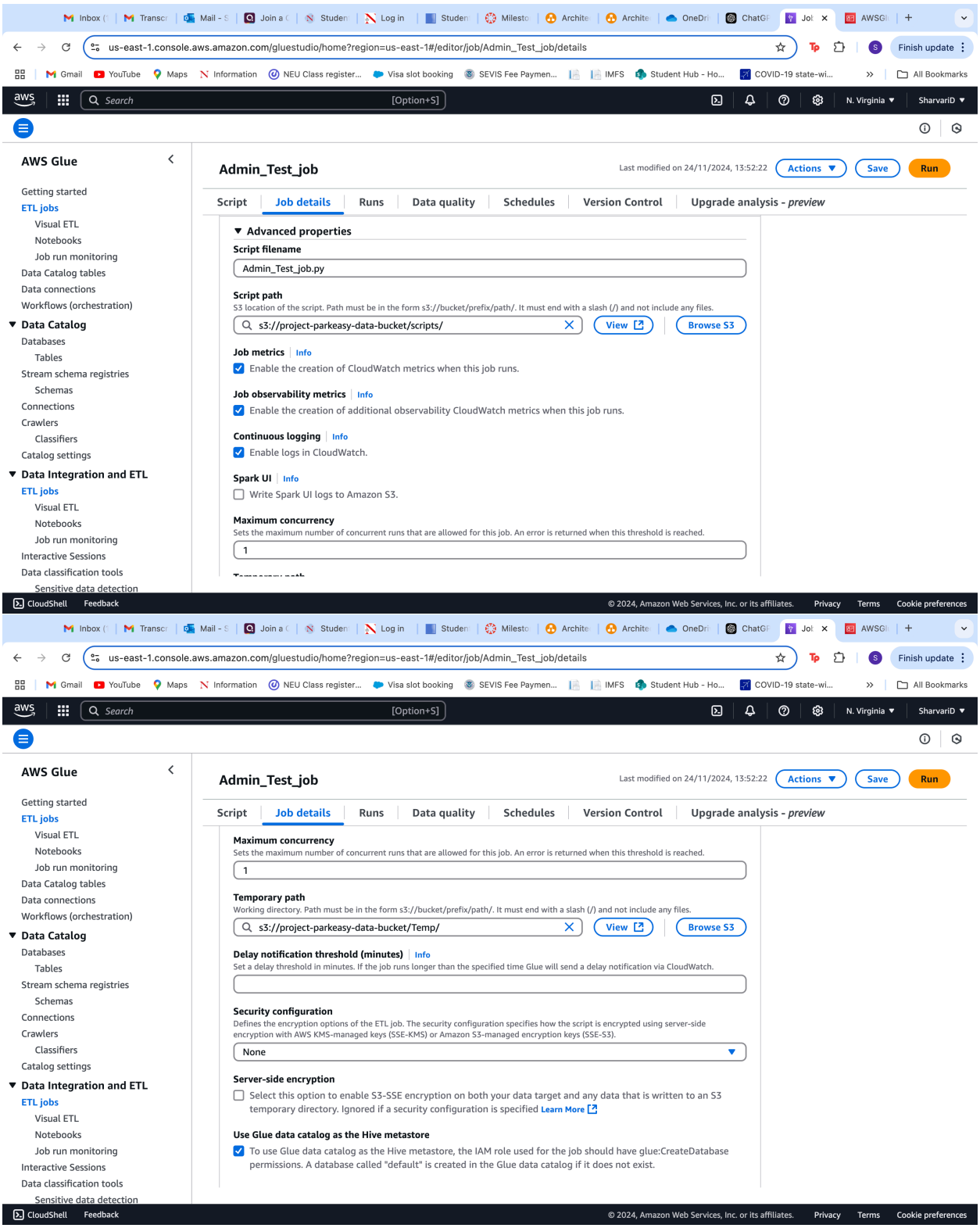
ETL Jobs created in Glue ETL



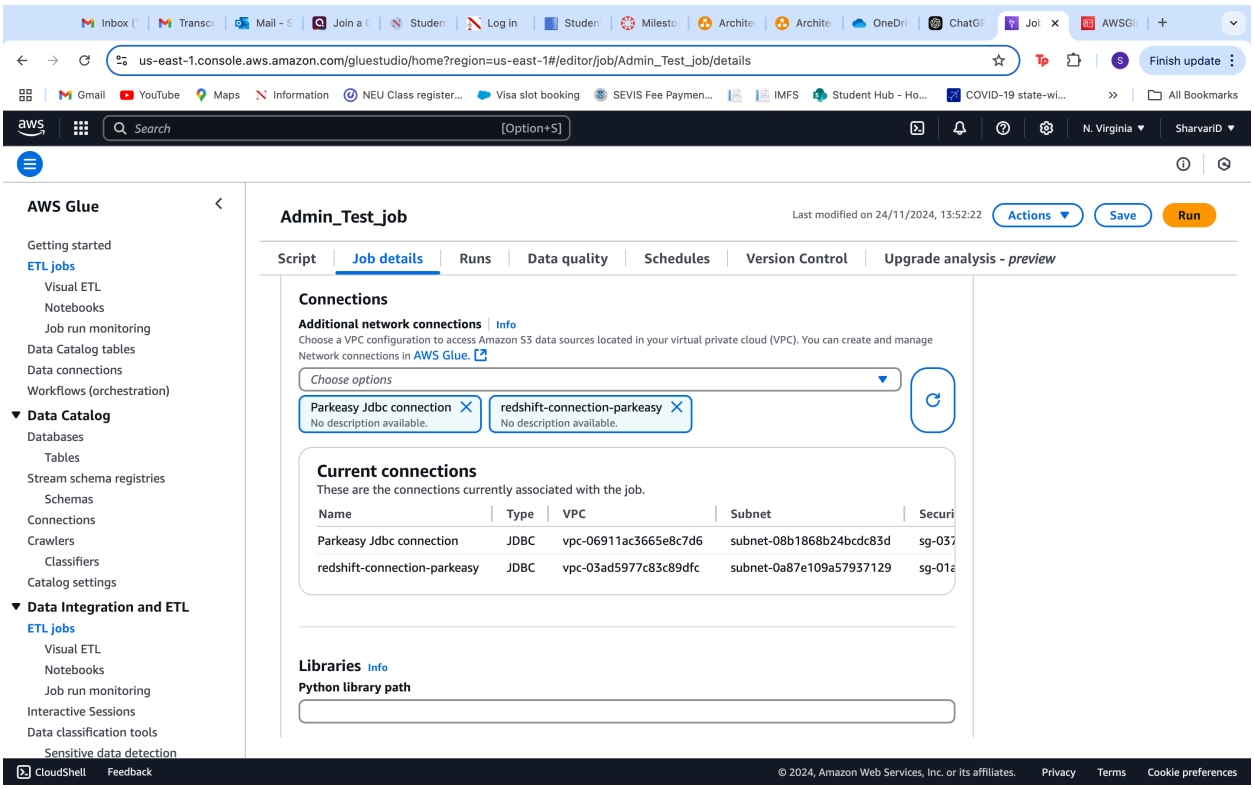
Example – Job configuration details for the Admin_job


us-east-1.console.aws.amazon.com/gluestudio/home?region=us-east-1#/editor/job/Admin_Test_job/details
Finish update
Gmail
YouTube
Maps
Information
NEU Class register...
Visa slot booking
SEVIS Fee Paymen...
IMFS
Student Hub - Ho...
COVID-19 state-wi...
All Bookmarks
Search
[Option+S]
N. Virginia
SharvariD
AWS Glue
Getting started
ETL jobs
Visual ETL
Notebooks
Job run monitoring
Data Catalog tables
Data connections
Workflows (orchestration)
Data Catalog
Databases
Tables
Stream schema registries
Schemas
Connections
Crawlers
Classifiers
Catalog settings
Data Integration and ETL
ETL jobs
Visual ETL
Notebooks
Job run monitoring
Interactive Sessions
Data classification tools
Sensitive data detection
Admin_Test_job
Last modified on 24/11/2024, 13:52:22
Actions
Save
Run
Script
Job details
Runs
Data quality
Schedules
Version Control
Upgrade analysis - preview
Basic properties
Info
Name
Admin_Test_job
Description - optional
Descriptions can be up to 2048 characters long.
IAM Role
Role assumed by the job with permission to access your data stores. Ensure that this role has permission to your Amazon S3 sources, targets, temporary directory, scripts, and any libraries used by the job.
AWSGlueServiceRole-ParkEasy
Type
The type of ETL job. This is set automatically based on the types of data sources you have selected.
Spark
Glue version
Info
Glue 4.0 - Supports spark 3.3, Scala 2, Python 3
Language
CloudShell
Feedback
© 2024, Amazon Web Services, Inc. or its affiliates.
Privacy
Terms
Cookie preferences


us-east-1.console.aws.amazon.com/gluestudio/home?region=us-east-1#/editor/job/Admin_Test_job/details
Finish update
Gmail
YouTube
Maps
Information
NEU Class register...
Visa slot booking
SEVIS Fee Paymen...
IMFS
Student Hub - Ho...
COVID-19 state-wi...
All Bookmarks
Search
[Option+S]
N. Virginia
SharvariD
AWS Glue
Getting started
ETL jobs
Visual ETL
Notebooks
Job run monitoring
Data Catalog tables
Data connections
Workflows (orchestration)
Data Catalog
Databases
Tables
Stream schema registries
Schemas
Connections
Crawlers
Classifiers
Catalog settings
Data Integration and ETL
ETL jobs
Visual ETL
Notebooks
Job run monitoring
Interactive Sessions
Data classification tools
Sensitive data detection
Admin_Test_job
Last modified on 24/11/2024, 13:52:22
Actions
Save
Run
Script
Job details
Runs
Data quality
Schedules
Version Control
Upgrade analysis - preview
Language
Python 3
Worker type
Set the type of predefined worker that is allowed when a job runs.
G 1X
(4vCPU and 16GB RAM)
Automatically scale the number of workers
AWS Glue will optimize costs and resource usage by dynamically scaling the number of workers up and down throughout the job run. Requires Glue 3.0 or later.
Requested number of workers
The number of workers you want AWS Glue to allocate to this job.
2
Generate job insights
AWS Glue will analyze your job runs and provide insights on how to optimize your jobs and the reasons for job failures.
Job bookmark
Info
Specifies how AWS Glue processes job bookmark when the job runs. It can remember previously processed data (Enable), update state information (Pause), or ignore state information (Disable).
Disable
CloudShell
Feedback
© 2024, Amazon Web Services, Inc. or its affiliates.
Privacy
Terms
Cookie preferences

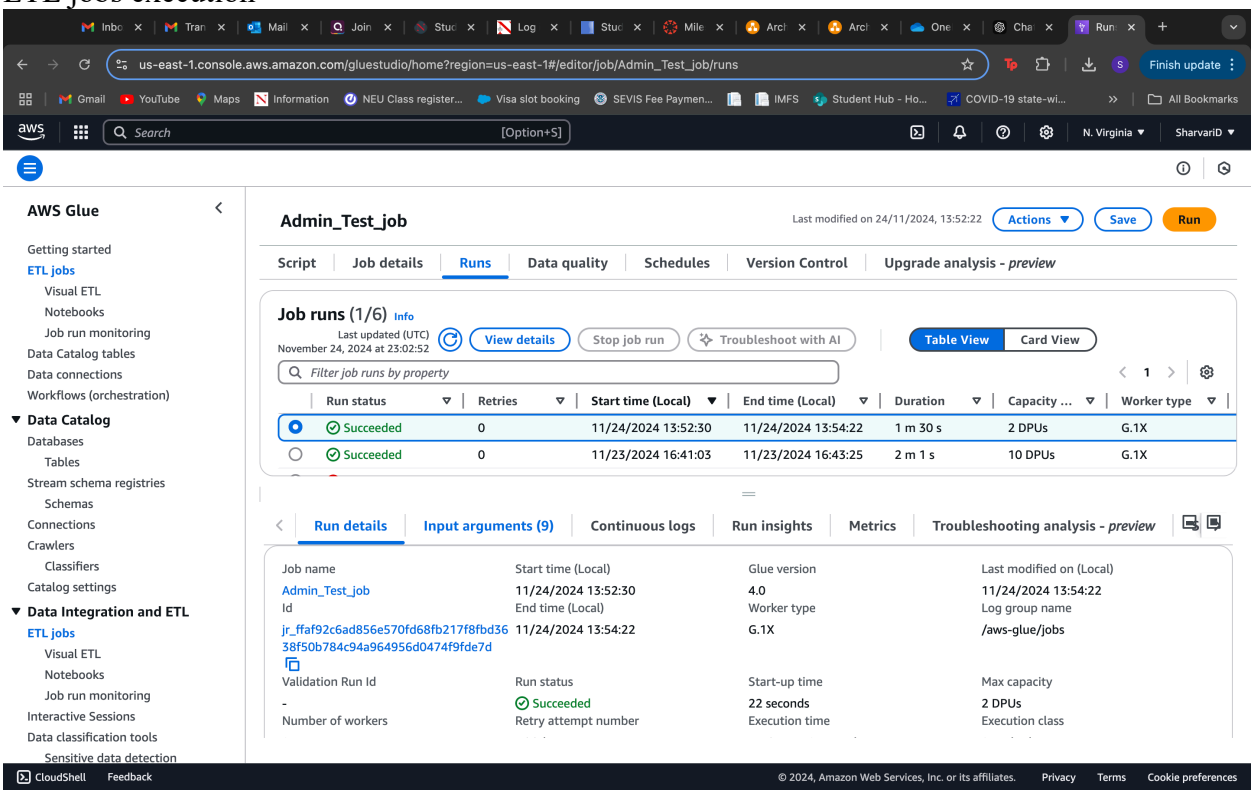

us-east-1.console.aws.amazon.com/gluestudio/home?region=us-east-1#/editor/job/Admin_Test_job/details
Finish update
Gmail
YouTube
Maps
Information
NEU Class register...
Visa slot booking
SEVIS Fee Paymen...
IMFS
Student Hub - Ho...
COVID-19 state-wi...
All Bookmarks
Search
[Option+S]
N. Virginia
SharvariD
AWS Glue
Getting started
ETL jobs
Visual ETL
Notebooks
Job run monitoring
Data Catalog tables
Data connections
Workflows (orchestration)
Data Catalog
Databases
Tables
Stream schema registries
Schemas
Connections
Crawlers
Classifiers
Catalog settings
Data Integration and ETL
ETL jobs
Visual ETL
Notebooks
Job run monitoring
Interactive Sessions
Data classification tools
Sensitive data detection
Admin_Test_job
Last modified on 24/11/2024, 13:52:22
Actions
Save
Run
Script
Job details
Runs
Data quality
Schedules
Version Control
Upgrade analysis - preview
Advanced properties
Script filename
Admin_Test_job.py
Script path
S3 location of the script. Path must be in the form s3://bucket/prefix/path/. It must end with a slash (/) and not include any files.
s3://project-parkeasy-data-bucket/scripts/
View
Browse S3
Job metrics
Info
Enable the creation of CloudWatch metrics when this job runs.
Job observability metrics
Info
Enable the creation of additional observability CloudWatch metrics when this job runs.
Continuous logging
Info
Enable logs in CloudWatch.
Spark UI
Info
Write Spark UI logs to Amazon S3.
Maximum concurrency
Sets the maximum number of concurrent runs that are allowed for this job. An error is returned when this threshold is reached.
1
CloudShell
Feedback
© 2024, Amazon Web Services, Inc. or its affiliates.
Privacy
Terms
Cookie preferences

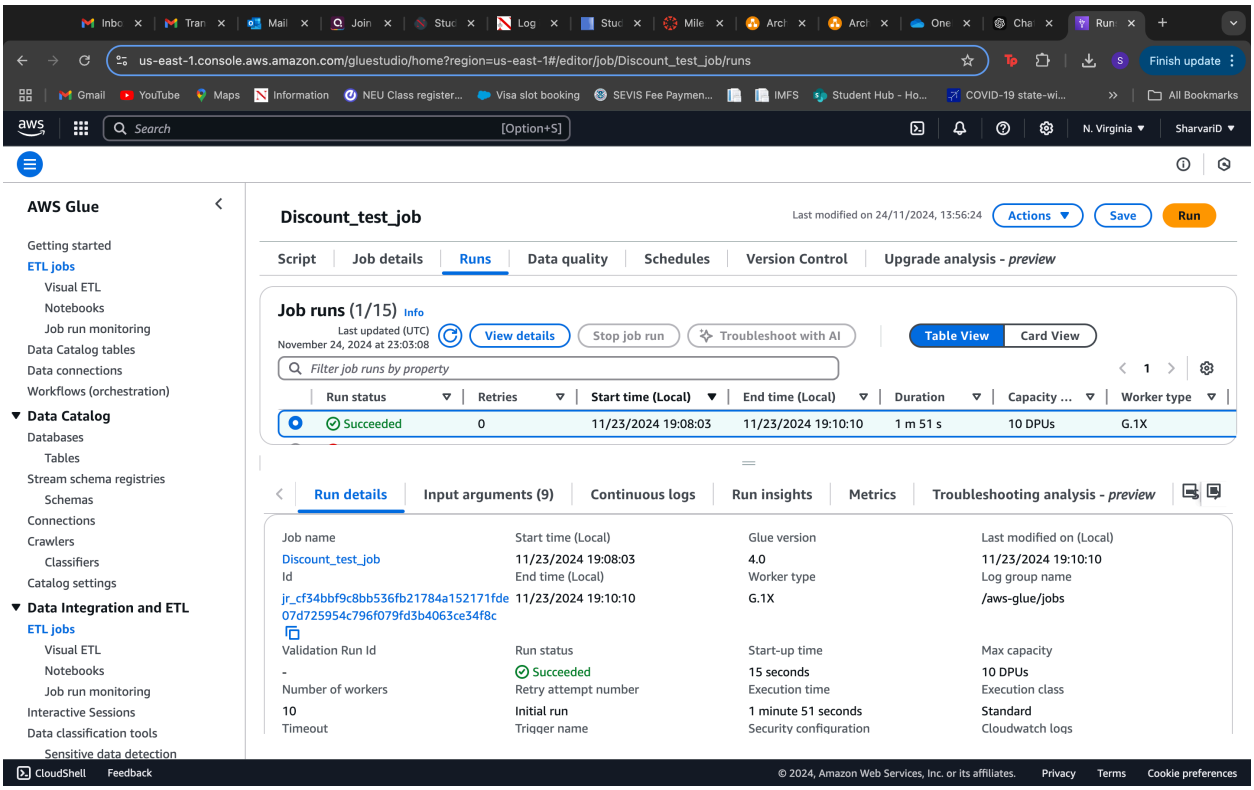
us-east-1.console.aws.amazon.com/gluestudio/home?region=us-east-1#/editor/job/Admin_Test_job/details
Finish update
Gmail
YouTube
Maps
Information
NEU Class register...
Visa slot booking
SEVIS Fee Paymen...
IMFS
Student Hub - Ho...
COVID-19 state-wi...
All Bookmarks
Search
[Option+S]
N. Virginia
SharvariD
AWS Glue
Getting started
ETL jobs
Visual ETL
Notebooks
Job run monitoring
Data Catalog tables
Data connections
Workflows (orchestration)
Data Catalog
Databases
Tables
Stream schema registries
Schemas
Connections
Crawlers
Classifiers
Catalog settings
Data Integration and ETL
ETL jobs
Visual ETL
Notebooks
Job run monitoring
Interactive Sessions
Data classification tools
Sensitive data detection
Admin_Test_job
Last modified on 24/11/2024, 13:52:22
Actions
Save
Run
Script
Job details
Runs
Data quality
Schedules
Version Control
Upgrade analysis - preview
Maximum concurrency
Sets the maximum number of concurrent runs that are allowed for this job. An error is returned when this threshold is reached.
1
Temporary path
Working directory. Path must be in the form s3://bucket/prefix/path/. It must end with a slash (/) and not include any files.
s3://project-parkeasy-data-bucket/Temp/
View
Browse S3
Delay notification threshold (minutes)
Info
Set a delay threshold in minutes. If the job runs longer than the specified time Glue will send a delay notification via CloudWatch.
Security configuration
Defines the encryption options of the ETL job. The security configuration specifies how the script is encrypted using server-side encryption with AWS KMS-managed keys (SSE-KMS) or Amazon S3-managed encryption keys (SSE-S3).
None
Server-side encryption
Select this option to enable S3-SSE encryption on both your data target and any data that is written to an S3 temporary directory. Ignored if a security configuration is specified
Learn More
Use Glue data catalog as the Hive metastore
To use Glue data catalog as the Hive metastore, the IAM role used for the job should have glue:CreateDatabase permissions. A database called "default" is created in the Glue data catalog if it does not exist.
CloudShell
Feedback
© 2024, Amazon Web Services, Inc. or its affiliates.
Privacy
Terms
Cookie preferences

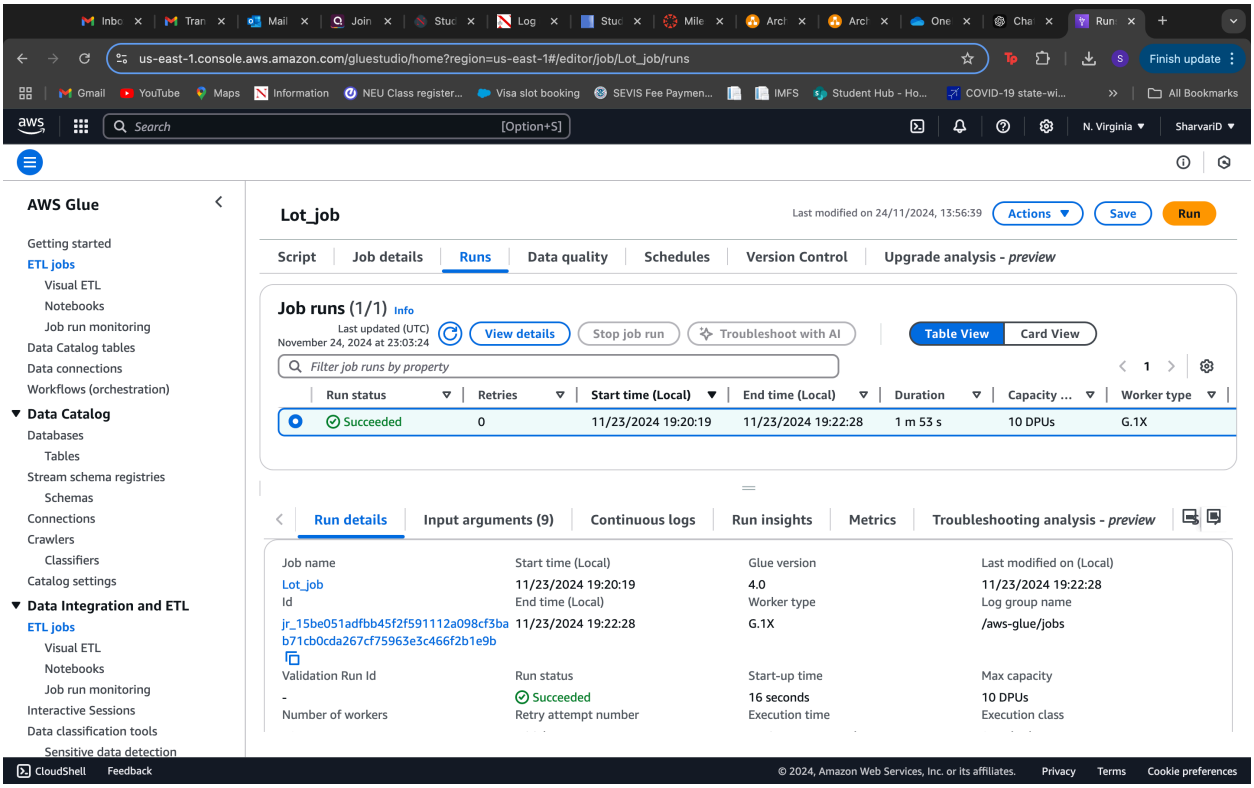


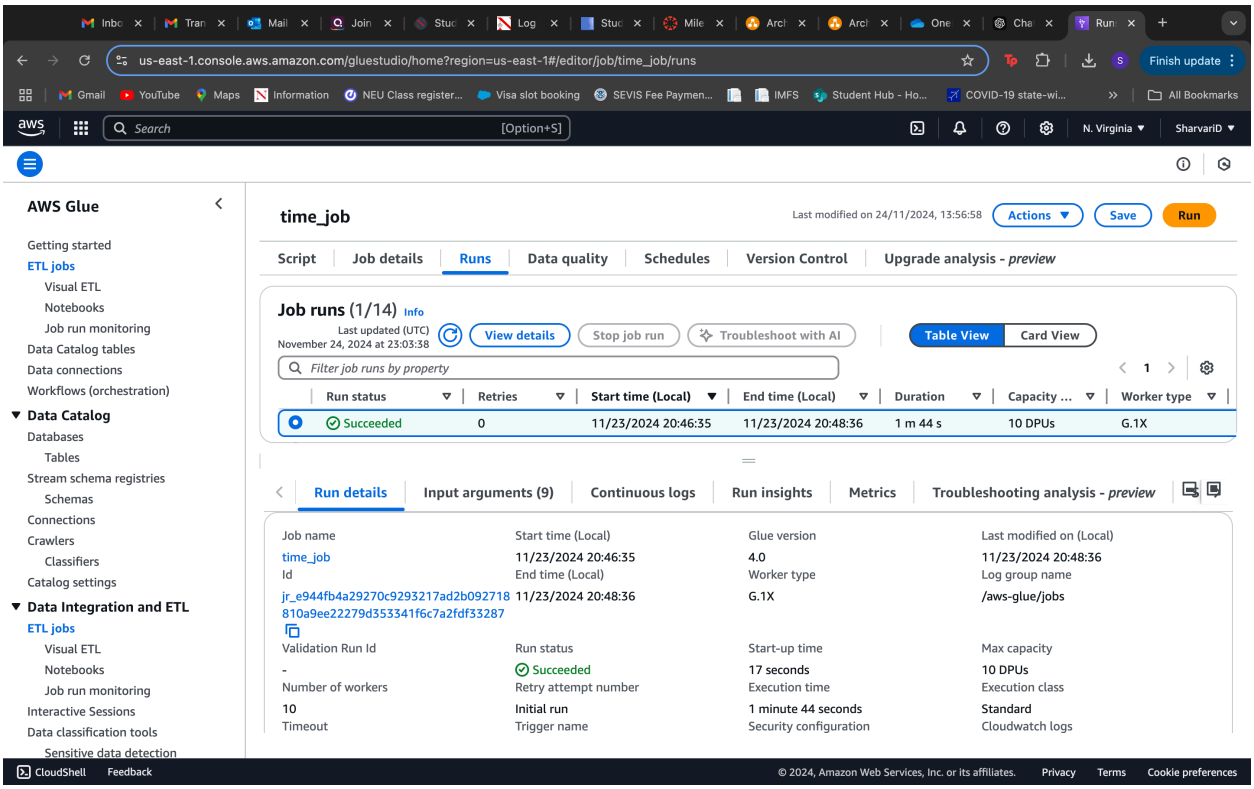
Similarly, all other jobs for the facts and dimension tables were configured in the same way

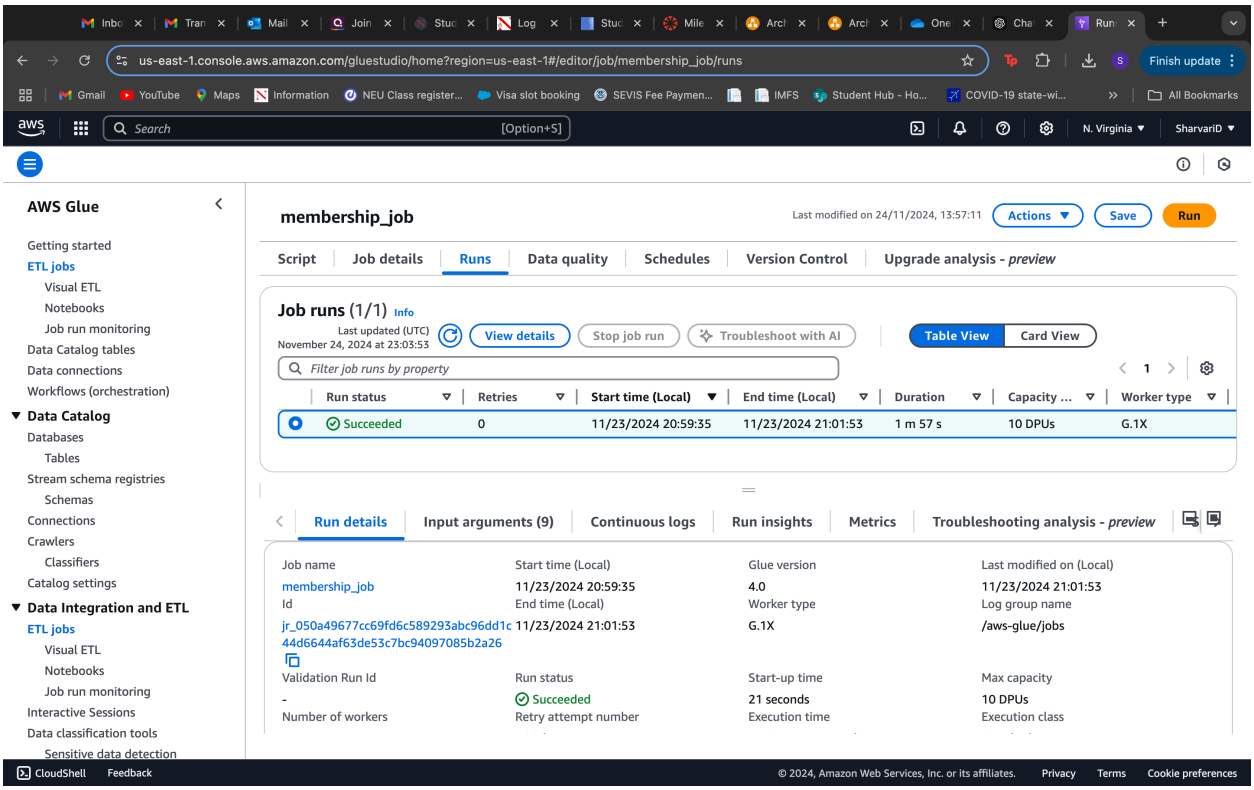
ETL jobs execution

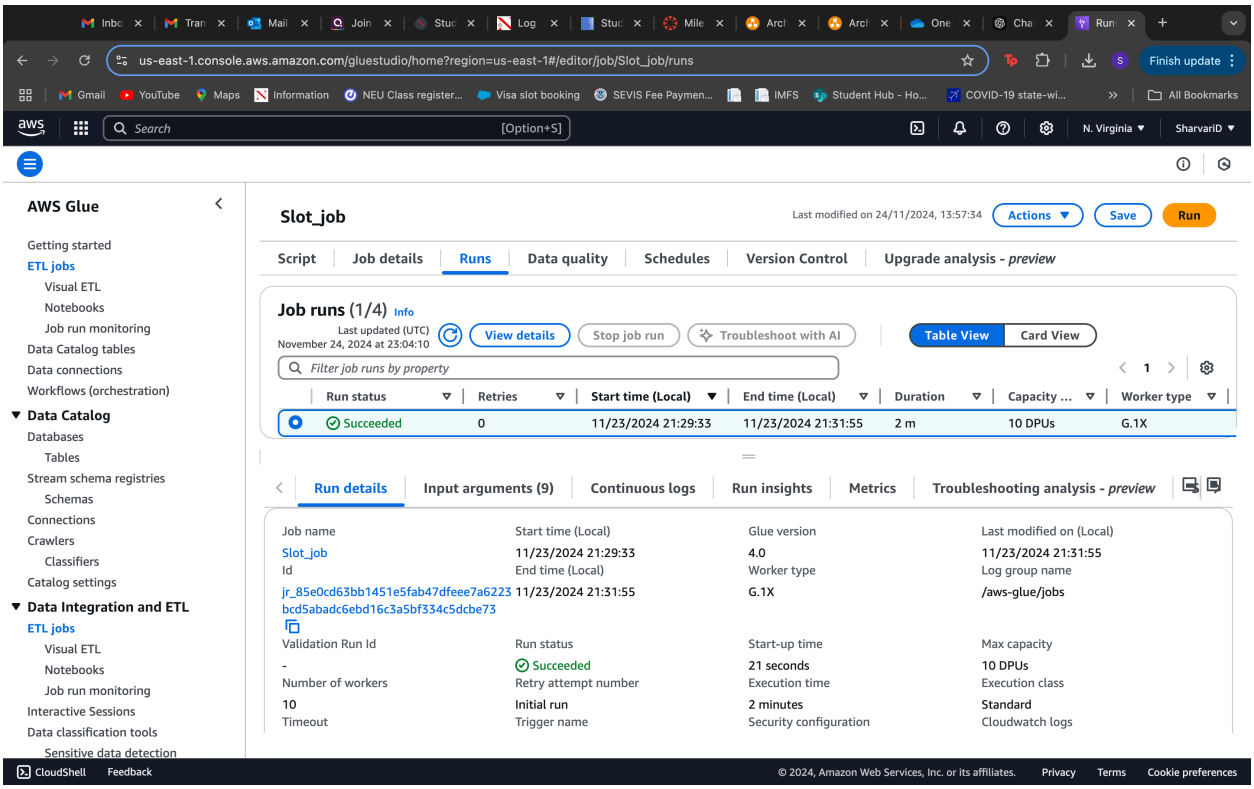


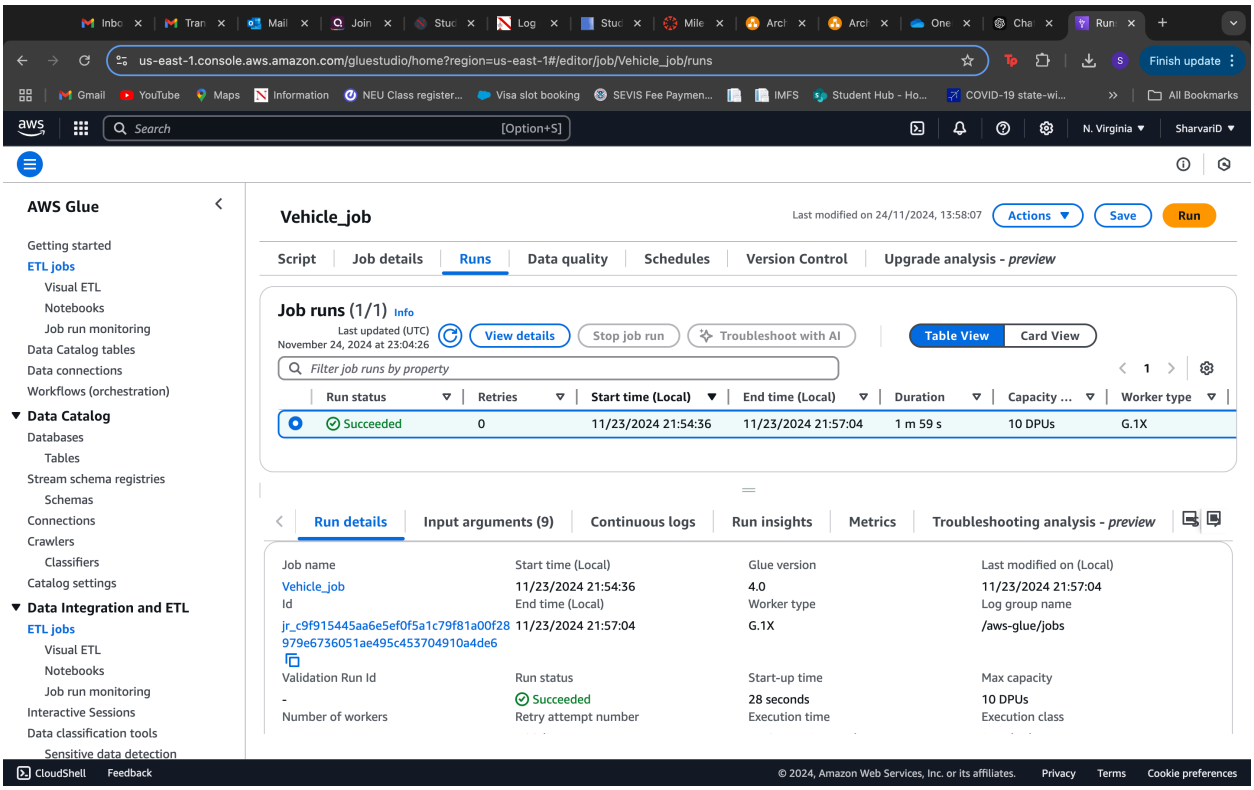

us-east-1.console.aws.amazon.com/gluestudio/home?region=us-east-1#/editor/job/Discount_test_job/runs
AWS Glue
Discount_test_job
Last modified on 24/11/2024, 13:56:24
Actions
Save
Run
Script
Job details
Runs
Data quality
Schedules
Version Control
Upgrade analysis - preview
Job runs (1/15) Info
Last updated (UTC) November 24, 2024 at 23:03:08
View details
Stop job run
Troubleshoot with AI
Table View
Card View
Filter job runs by property
Run status
Retries
Start time (Local)
End time (Local)
Duration
Capacity ...
Worker type
Succeeded
0
11/23/2024 19:08:03
11/23/2024 19:10:10
1 m 51 s
10 DPUs
G.1X
Run details
Input arguments (9)
Continuous logs
Run insights
Metrics
Troubleshooting analysis - preview
Job name
Discount_test_job
Start time (Local)
11/23/2024 19:08:03
Glue version
4.0
Last modified on (Local)
11/23/2024 19:10:10
Id
jr_cf34bbf9c8bb536fb21784a152171fde07d7225954c796f079fd3b4063ce34f8c
End time (Local)
11/23/2024 19:10:10
Worker type
G.1X
Log group name
/aws-glue/jobs
Validation Run Id
-
Run status
Succeeded
Start-up time
15 seconds
Max capacity
10 DPUs
Number of workers
10
Retry attempt number
Initial run
Execution time
1 minute 51 seconds
Execution class
Standard
Timeout
Trigger name
Security configuration
Cloudwatch logs
CloudShell
Feedback
© 2024, Amazon Web Services, Inc. or its affiliates.
Privacy
Terms
Cookie preferences

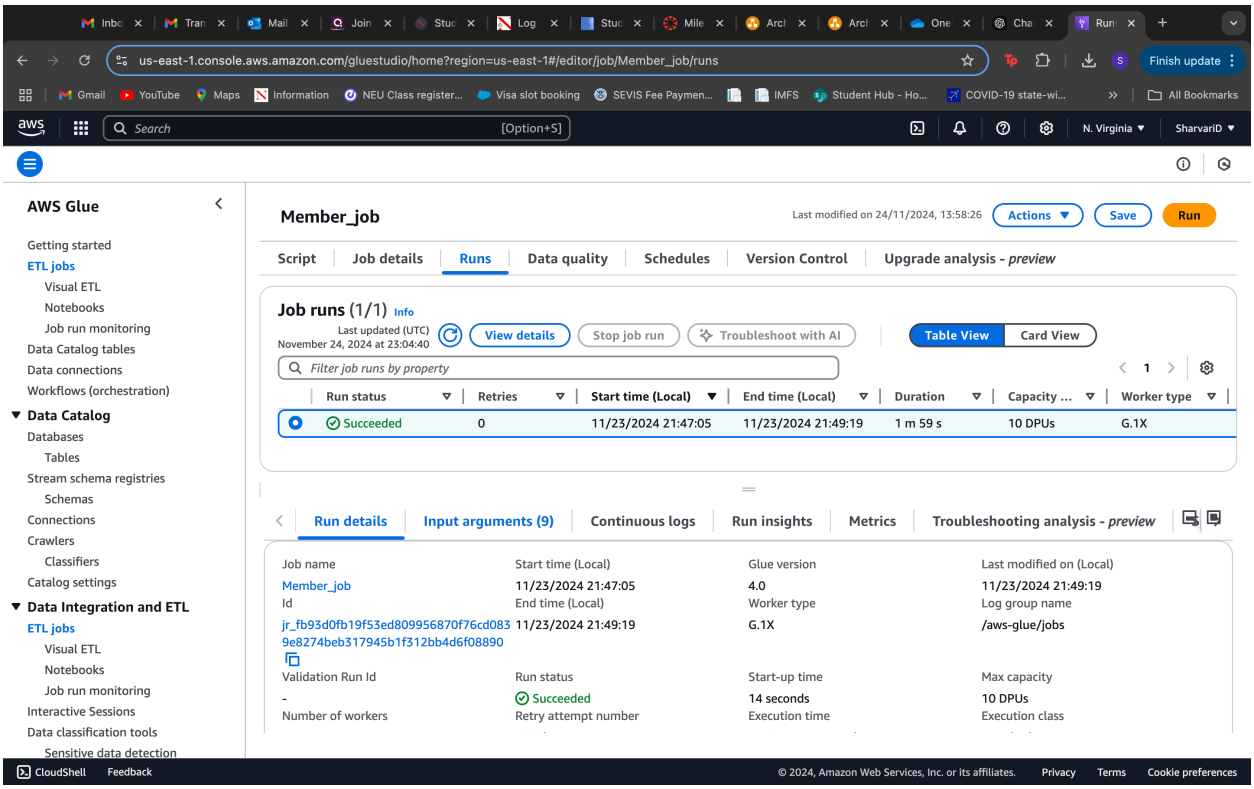

us-east-1.console.aws.amazon.com/gluestudio/home?region=us-east-1#/editor/job/Lot_job/runs
AWS Glue
Lot_job
Last modified on 24/11/2024, 13:56:39
Actions
Save
Run
Script
Job details
Runs
Data quality
Schedules
Version Control
Upgrade analysis - preview
Job runs (1/1) Info
Last updated (UTC) November 24, 2024 at 23:03:24
View details
Stop job run
Troubleshoot with AI
Table View
Card View
Filter job runs by property
Run status
Retries
Start time (Local)
End time (Local)
Duration
Capacity ...
Worker type
Succeeded
0
11/23/2024 19:20:19
11/23/2024 19:22:28
1 m 53 s
10 DPUs
G.1X
Run details
Input arguments (9)
Continuous logs
Run insights
Metrics
Troubleshooting analysis - preview
Job name
Lot_job
Start time (Local)
11/23/2024 19:20:19
Glue version
4.0
Last modified on (Local)
11/23/2024 19:22:28
Id
jr_15be051adfbb45f2f591112a098cf3bab71cb0cda267cf75963e3c466f2b1e9b
End time (Local)
11/23/2024 19:22:28
Worker type
G.1X
Log group name
/aws-glue/jobs
Validation Run Id
-
Run status
Succeeded
Start-up time
16 seconds
Max capacity
10 DPUs
Number of workers
Retry attempt number
Execution time
Execution class
CloudShell
Feedback
© 2024, Amazon Web Services, Inc. or its affiliates.
Privacy
Terms
Cookie preferences

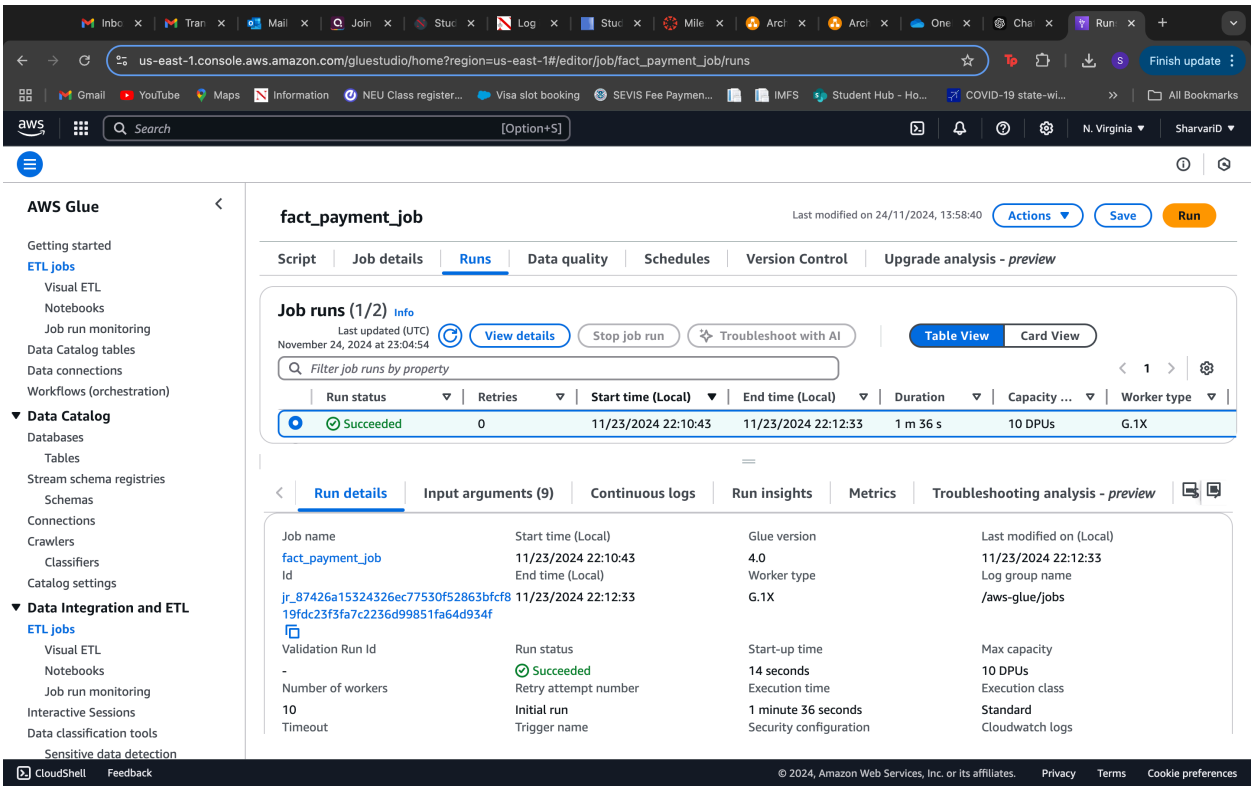

us-east-1.console.aws.amazon.com/gluestudio/home?region=us-east-1#/editor/job/time_job/runs
AWS Glue
time_job
Last modified on 24/11/2024, 13:56:58
Actions
Save
Run
Script
Job details
Runs
Data quality
Schedules
Version Control
Upgrade analysis - preview
Job runs (1/14) Info
Last updated (UTC) November 24, 2024 at 23:03:38
View details
Stop job run
Troubleshoot with AI
Table View
Card View
Filter job runs by property
Run status
Retries
Start time (Local)
End time (Local)
Duration
Capacity ...
Worker type
Succeeded
0
11/23/2024 20:46:35
11/23/2024 20:48:36
1 m 44 s
10 DPUs
G.1X
Run details
Input arguments (9)
Continuous logs
Run insights
Metrics
Troubleshooting analysis - preview
Job name
time_job
Start time (Local)
11/23/2024 20:46:35
Glue version
4.0
Last modified on (Local)
11/23/2024 20:48:36
Id
jr_e944fb4a29270c9293217ad2b092718810a9ee22279d353341f6c7a2fdf33287
End time (Local)
11/23/2024 20:48:36
Worker type
G.1X
Log group name
/aws-glue/jobs
Validation Run Id
-
Run status
Succeeded
Start-up time
17 seconds
Max capacity
10 DPUs
Number of workers
10
Retry attempt number
Initial run
Execution time
1 minute 44 seconds
Execution class
Standard
Timeout
Trigger name
Security configuration
Cloudwatch logs
CloudShell
Feedback
© 2024, Amazon Web Services, Inc. or its affiliates.
Privacy
Terms
Cookie preferences

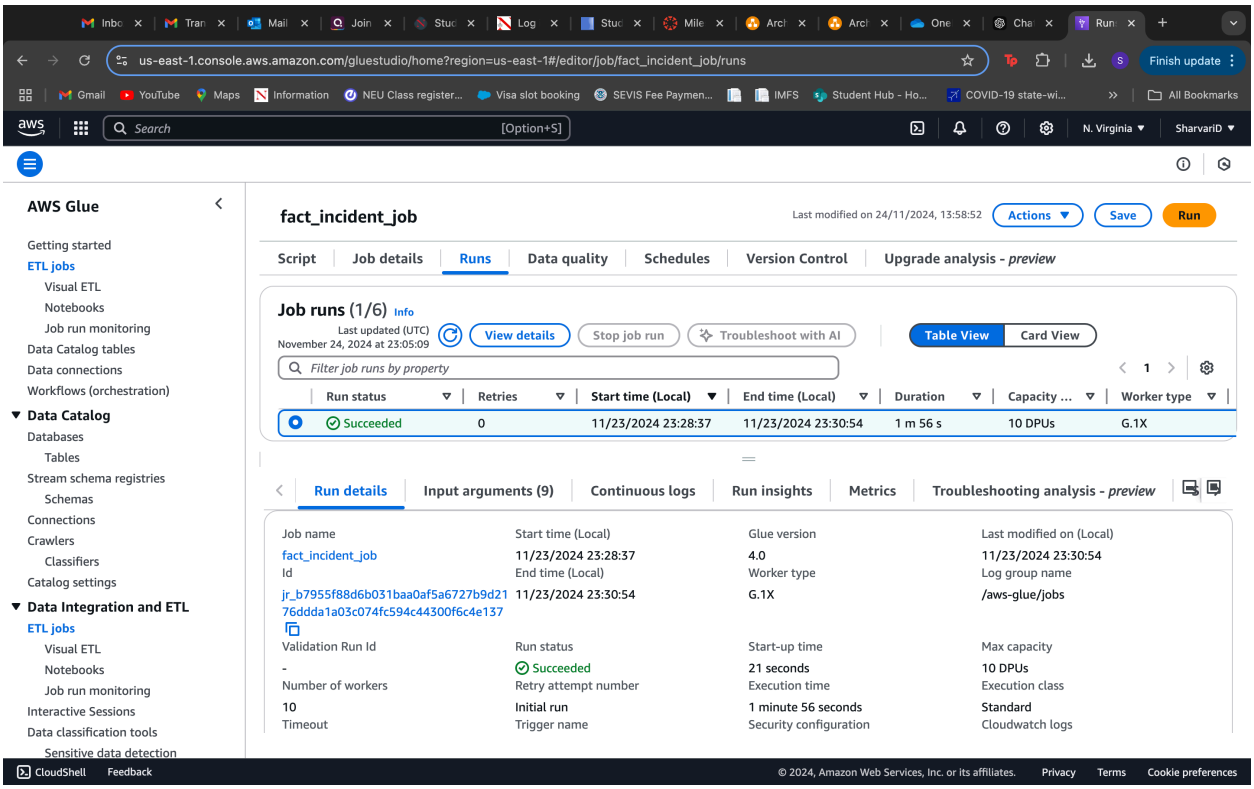

us-east-1.console.aws.amazon.com/gluestudio/home?region=us-east-1#/editor/job/membership_job/runs
AWS Glue
membership_job
Last modified on 24/11/2024, 13:57:11
Actions
Save
Run
Script
Job details
Runs
Data quality
Schedules
Version Control
Upgrade analysis - preview
Job runs (1/1) Info
Last updated (UTC) November 24, 2024 at 23:03:53
View details
Stop job run
Troubleshoot with AI
Table View
Card View
Filter job runs by property
Run status
Retries
Start time (Local)
End time (Local)
Duration
Capacity ...
Worker type
Succeeded
0
11/23/2024 20:59:35
11/23/2024 21:01:53
1 m 57 s
10 DPUs
G.1X
Run details
Input arguments (9)
Continuous logs
Run insights
Metrics
Troubleshooting analysis - preview
Job name
membership_job
Start time (Local)
11/23/2024 20:59:35
Glue version
4.0
Last modified on (Local)
11/23/2024 21:01:53
Id
jr_050a49677cc69fd6c589293abc96dd1c44d6644af63de53c7bc94097085b2a26
End time (Local)
11/23/2024 21:01:53
Worker type
G.1X
Log group name
/aws-glue/jobs
Validation Run Id
-
Run status
Succeeded
Start-up time
21 seconds
Max capacity
10 DPUs
Number of workers
Retry attempt number
Execution time
Execution class
CloudShell
Feedback
© 2024, Amazon Web Services, Inc. or its affiliates.
Privacy
Terms
Cookie preferences

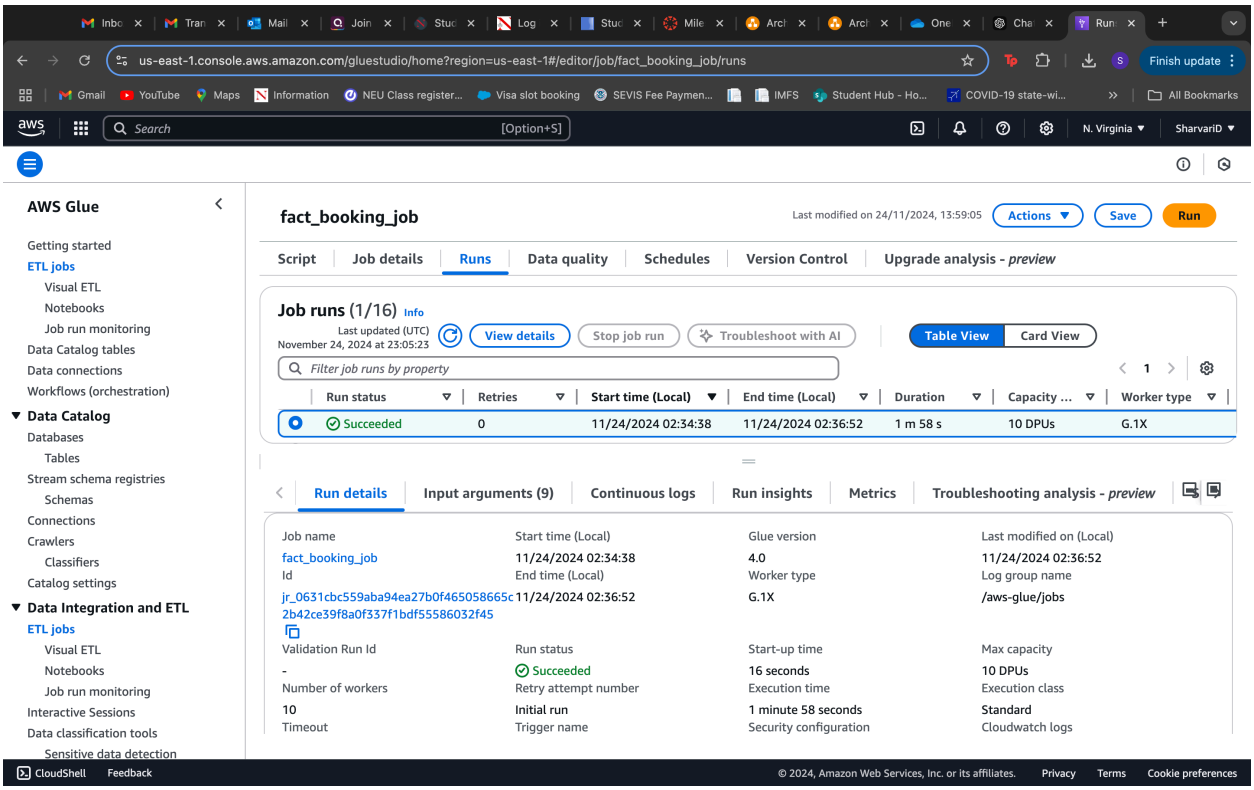

us-east-1.console.aws.amazon.com/gluestudio/home?region=us-east-1#/editor/job/Slot_job/runs
AWS Glue
Slot_job
Last modified on 24/11/2024, 13:57:34
Actions
Save
Run
Script
Job details
Runs
Data quality
Schedules
Version Control
Upgrade analysis - preview
Job runs (1/4) Info
Last updated (UTC) November 24, 2024 at 23:04:10
View details
Stop job run
Troubleshoot with AI
Table View
Card View
Filter job runs by property
Run status
Retries
Start time (Local)
End time (Local)
Duration
Capacity ...
Worker type
Succeeded
0
11/23/2024 21:29:33
11/23/2024 21:31:55
2 m
10 DPUs
G.1X
Run details
Input arguments (9)
Continuous logs
Run insights
Metrics
Troubleshooting analysis - preview
Job name
Slot_job
Start time (Local)
11/23/2024 21:29:33
Glue version
4.0
Last modified on (Local)
11/23/2024 21:31:55
Id
jr_85e0cd63bb1451e5fab47dfeee7a6223bcd5abadc6ebd16c3a5bf334c5dcbe73
End time (Local)
11/23/2024 21:31:55
Worker type
G.1X
Log group name
/aws-glue/jobs
Validation Run Id
-
Run status
Succeeded
Start-up time
21 seconds
Max capacity
10 DPUs
Number of workers
10
Retry attempt number
Initial run
Execution time
2 minutes
Execution class
Standard
Timeout
Trigger name
Security configuration
Cloudwatch logs
CloudShell
Feedback
© 2024, Amazon Web Services, Inc. or its affiliates.
Privacy
Terms
Cookie preferences


us-east-1.console.aws.amazon.com/gluestudio/home?region=us-east-1#/editor/job/Vehicle_job/runs
AWS Glue
Vehicle_job
Last modified on 24/11/2024, 13:58:07
Actions
Save
Run
Script
Job details
Runs
Data quality
Schedules
Version Control
Upgrade analysis - preview
Job runs (1/1) Info
Last updated (UTC) November 24, 2024 at 23:04:26
View details
Stop job run
Troubleshoot with AI
Table View
Card View
Filter job runs by property
Run status
Retries
Start time (Local)
End time (Local)
Duration
Capacity ...
Worker type
Succeeded
0
11/23/2024 21:54:36
11/23/2024 21:57:04
1 m 59 s
10 DPUs
G.1X
Run details
Input arguments (9)
Continuous logs
Run insights
Metrics
Troubleshooting analysis - preview
Job name
Vehicle_job
Start time (Local)
11/23/2024 21:54:36
Glue version
4.0
Last modified on (Local)
11/23/2024 21:57:04
Id
jr_c9f915445aa6e5ef0f5a1c79f81a00f28979e6736051ae495c453704910a4de6
End time (Local)
11/23/2024 21:57:04
Worker type
G.1X
Log group name
/aws-glue/jobs
Validation Run Id
-
Run status
Succeeded
Start-up time
28 seconds
Max capacity
10 DPUs
Number of workers
Retry attempt number
Execution time
Execution class
CloudShell
Feedback
© 2024, Amazon Web Services, Inc. or its affiliates.
Privacy
Terms
Cookie preferences


us-east-1.console.aws.amazon.com/gluestudio/home?region=us-east-1#/editor/job/Member_job/runs
AWS Glue
Member_job
Last modified on 24/11/2024, 13:58:26
Actions
Save
Run
Script
Job details
Runs
Data quality
Schedules
Version Control
Upgrade analysis - preview
Job runs (1/1) Info
Last updated (UTC) November 24, 2024 at 23:04:40
View details
Stop job run
Troubleshoot with AI
Table View
Card View
Filter job runs by property
Run status
Retries
Start time (Local)
End time (Local)
Duration
Capacity ...
Worker type
Succeeded
0
11/23/2024 21:47:05
11/23/2024 21:49:19
1 m 59 s
10 DPUs
G.1X
Run details
Input arguments (9)
Continuous logs
Run insights
Metrics
Troubleshooting analysis - preview
Job name
Member_job
Start time (Local)
11/23/2024 21:47:05
Glue version
4.0
Last modified on (Local)
11/23/2024 21:49:19
Id
jr_fb93d0fb19f53ed809956870f76cd0839e8274beb317945b1f312bb4d6f08890
End time (Local)
11/23/2024 21:49:19
Worker type
G.1X
Log group name
/aws-glue/jobs
Validation Run Id
-
Run status
Succeeded
Start-up time
14 seconds
Max capacity
10 DPUs
Number of workers
Retry attempt number
Execution time
Execution class
CloudShell
Feedback
© 2024, Amazon Web Services, Inc. or its affiliates.
Privacy
Terms
Cookie preferences


us-east-1.console.aws.amazon.com/gluestudio/home?region=us-east-1#/editor/job/fact_payment_job/runs
AWS Glue
fact_payment_job
Last modified on 24/11/2024, 13:58:40
Actions
Save
Run
Script
Job details
Runs
Data quality
Schedules
Version Control
Upgrade analysis - preview
Job runs (1/2) Info
Last updated (UTC) November 24, 2024 at 23:04:54
View details
Stop job run
Troubleshoot with AI
Table View
Card View
Filter job runs by property
Run status
Retries
Start time (Local)
End time (Local)
Duration
Capacity ...
Worker type
Succeeded
0
11/23/2024 22:10:43
11/23/2024 22:12:33
1 m 36 s
10 DPUs
G.1X
Run details
Input arguments (9)
Continuous logs
Run insights
Metrics
Troubleshooting analysis - preview
Job name
fact_payment_job
Start time (Local)
11/23/2024 22:10:43
Glue version
4.0
Last modified on (Local)
11/23/2024 22:12:33
Id
jr_87426a15324326ec77530f52863bfcf819fdc23f3fa7c2236d99851fa64d934f
End time (Local)
11/23/2024 22:12:33
Worker type
G.1X
Log group name
/aws-glue/jobs
Validation Run Id
-
Run status
Succeeded
Start-up time
14 seconds
Max capacity
10 DPUs
Number of workers
10
Retry attempt number
Initial run
Execution time
1 minute 36 seconds
Execution class
Standard
Timeout
Trigger name
Security configuration
Cloudwatch logs
CloudShell
Feedback
© 2024, Amazon Web Services, Inc. or its affiliates.
Privacy
Terms
Cookie preferences





Orchestration of the entire process using step function –

State machine successfully created

GlueWorkflowStateMachine-ParkEasy

Edit | Actions | Start execution

Details

Arn
arn:aws:states:us-east-1:593793039905:stateMachine:GlueWorkflowStateMachine-ParkEasy

IAM role ARN
arn:aws:iam::593793039905:role/GlueStepFunctionsExecutionRole

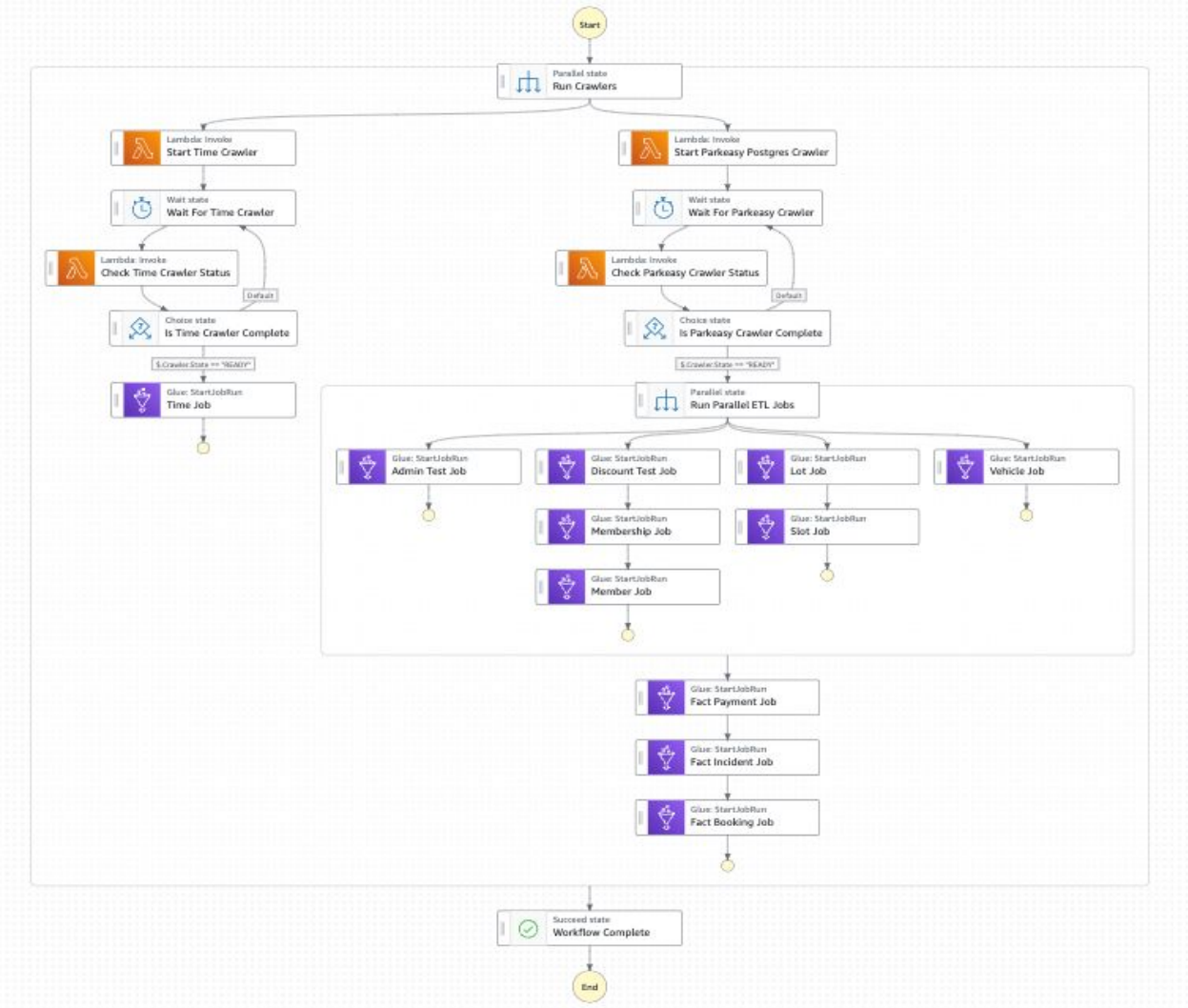
Type
Standard

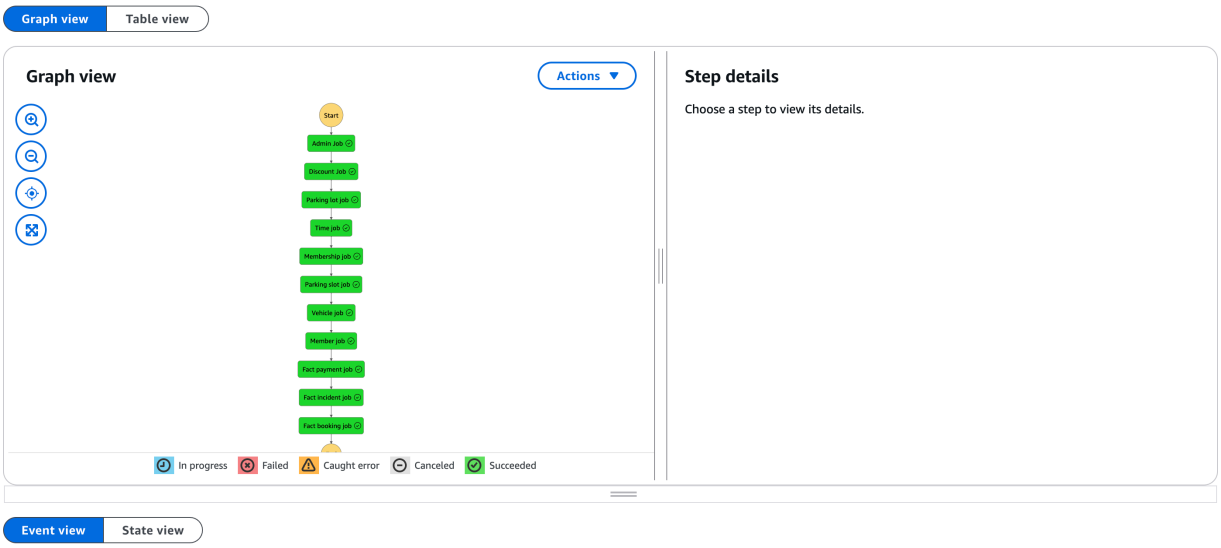
Status
Active

Creation date
Nov 24, 2024, 16:43:51 (UTC-05:00)

X-Ray tracing
Disabled

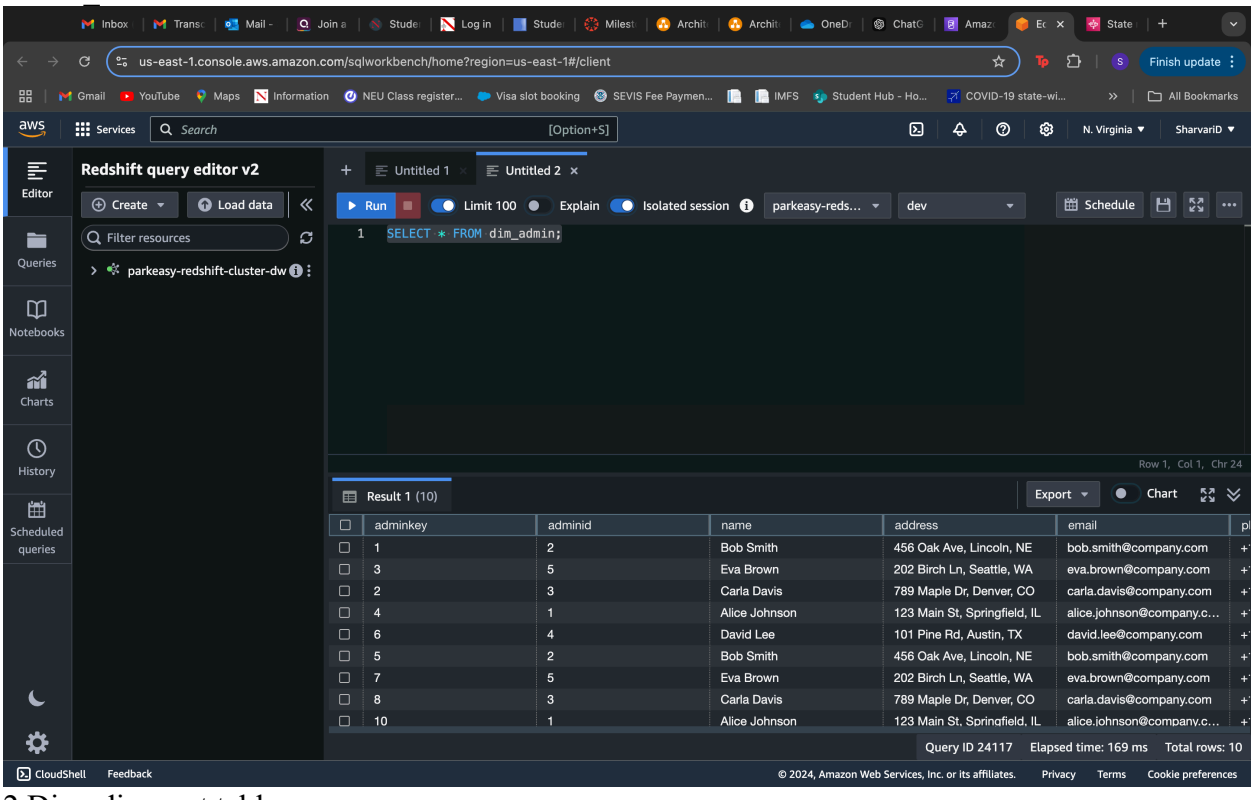
Executions | Monitoring | Logging | Definition | Aliases | Versions | Tags



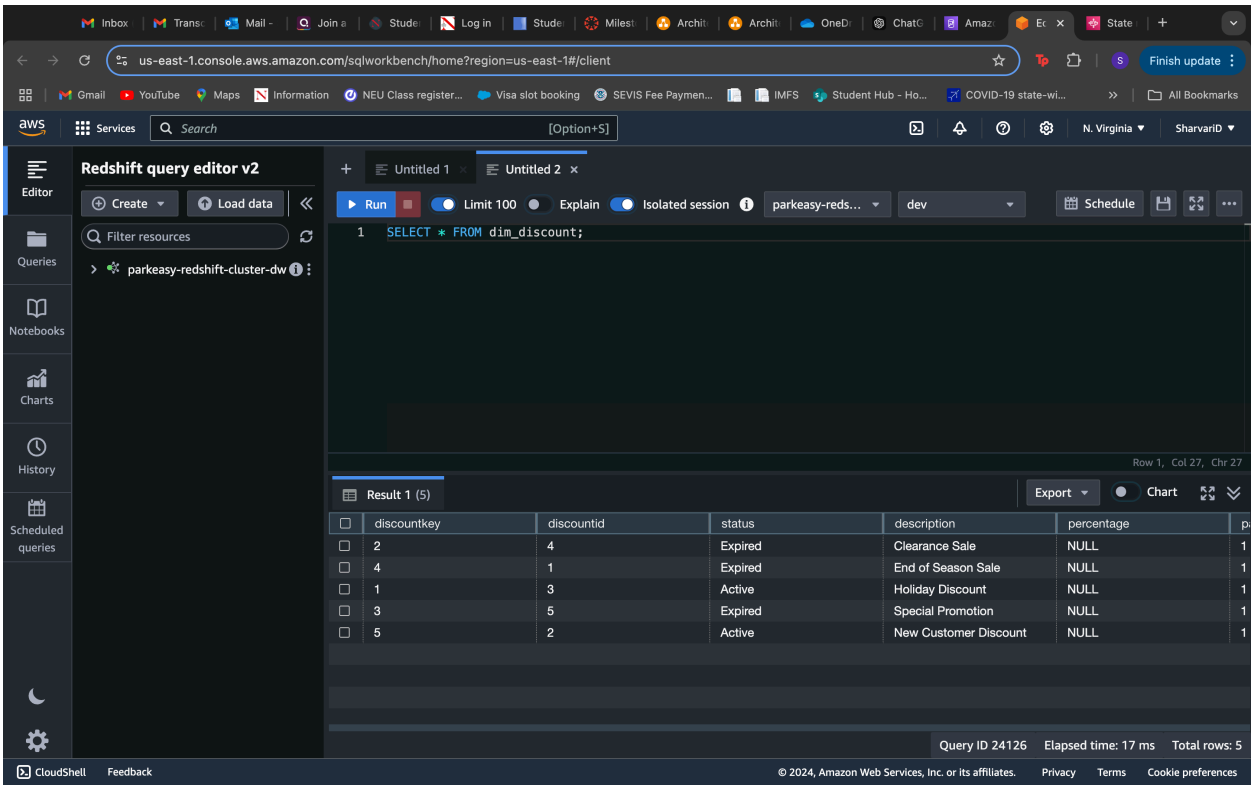


To check if all the Tables were loaded completely, we made use of the Redshift query editor v2

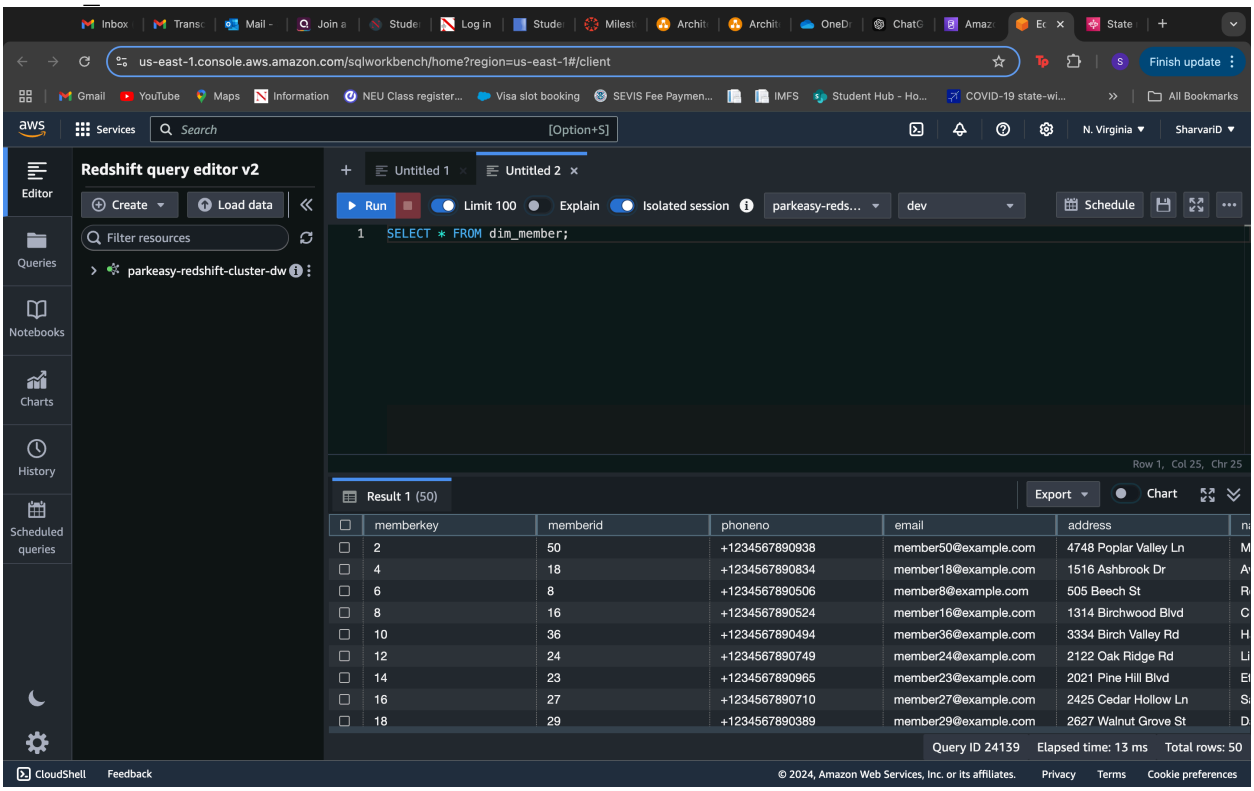
1.Dim_admin table



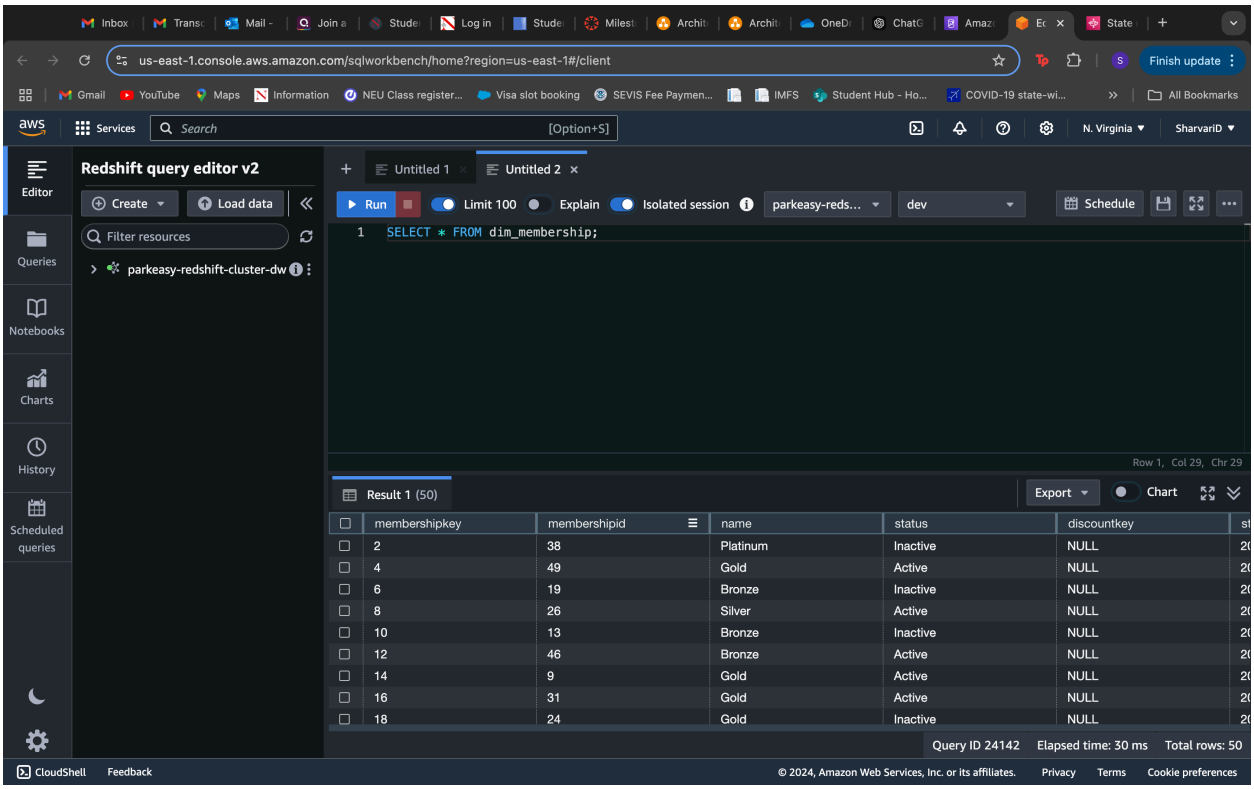
2.Dim_discount table



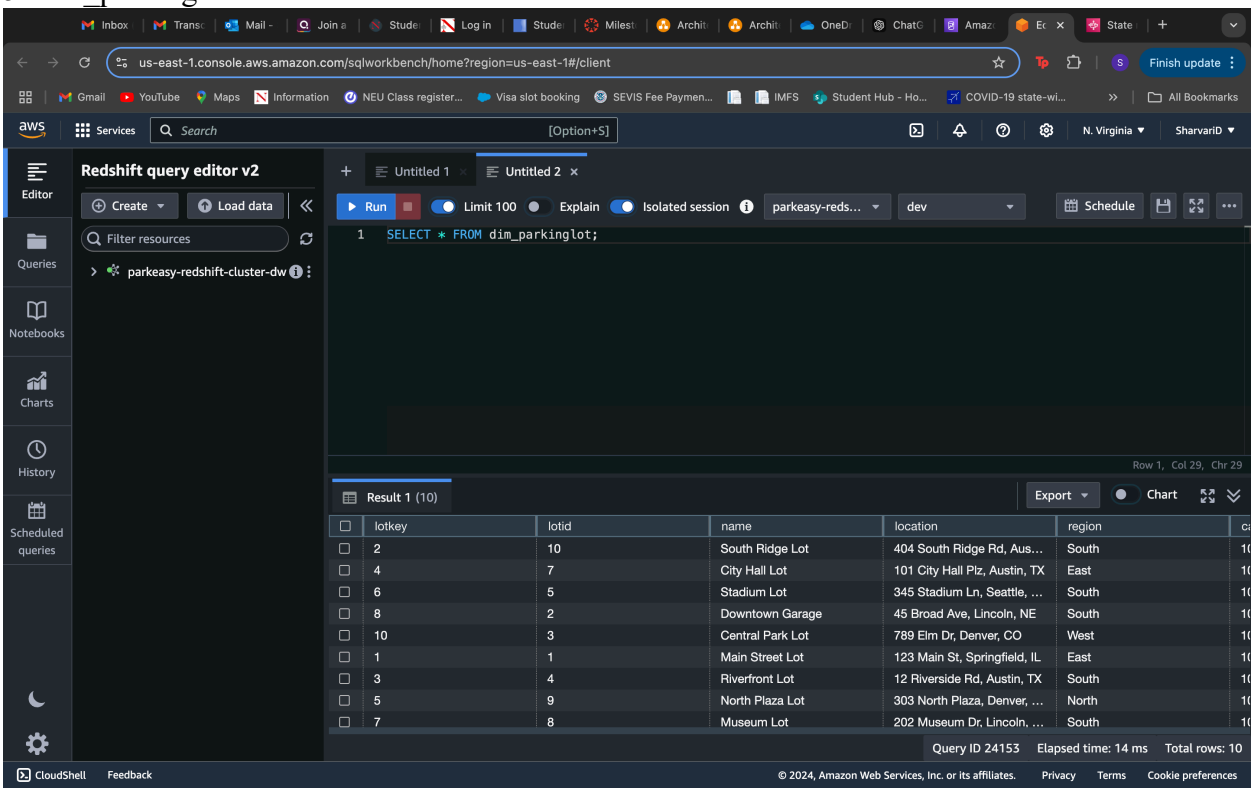
3.Dim_member table



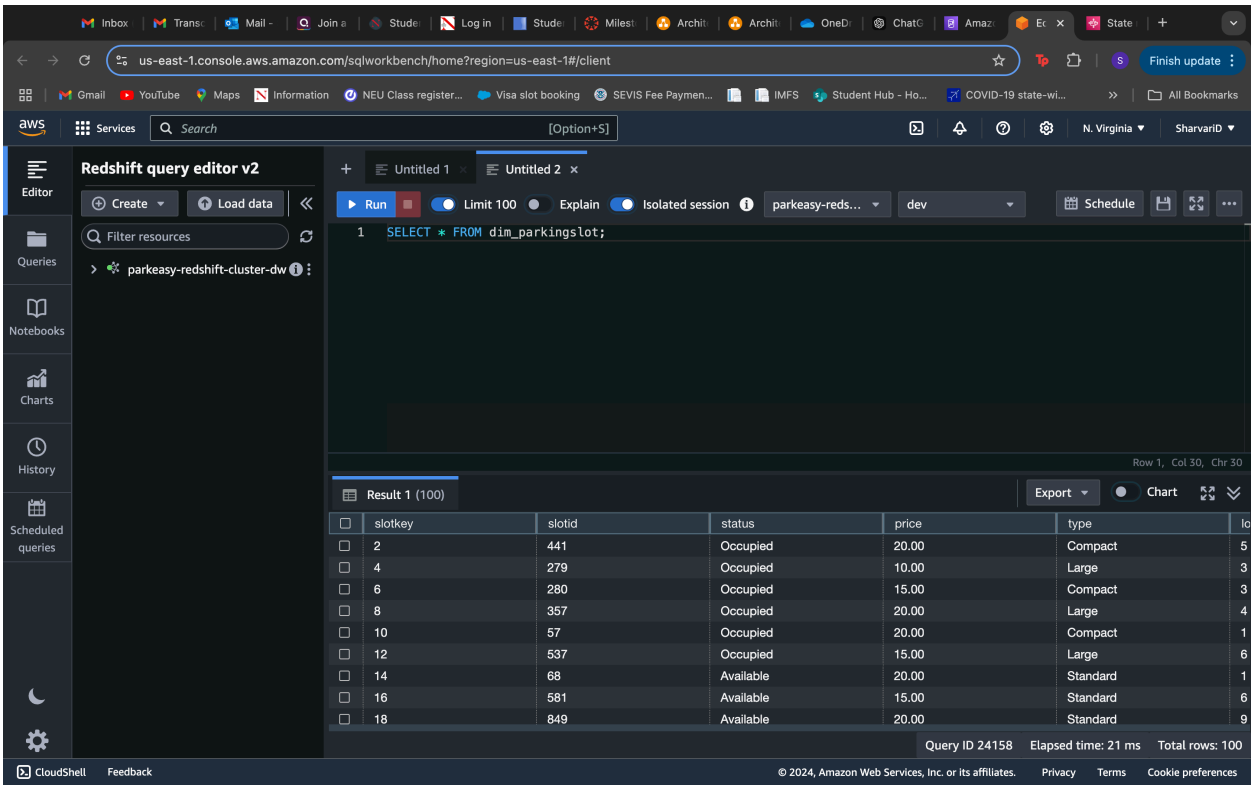
4.Dim_membership table



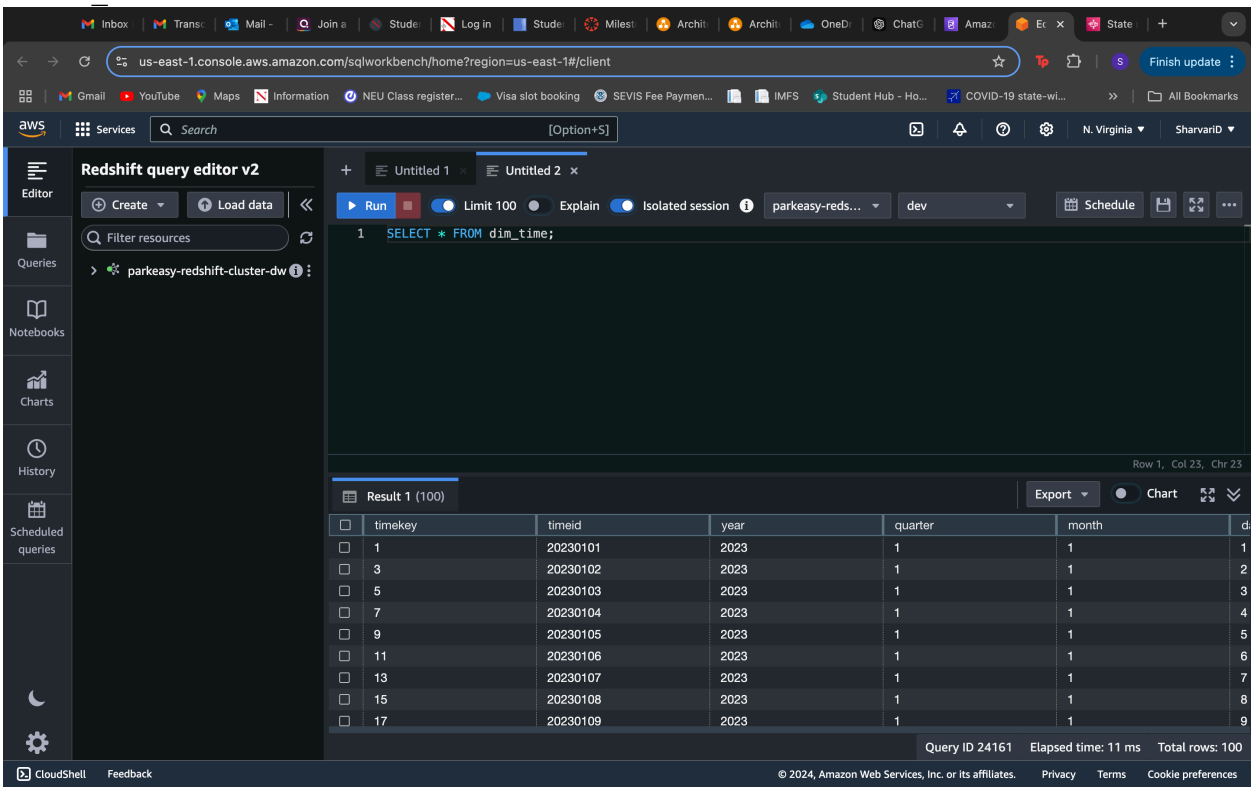
5.Dim_parkinglot table



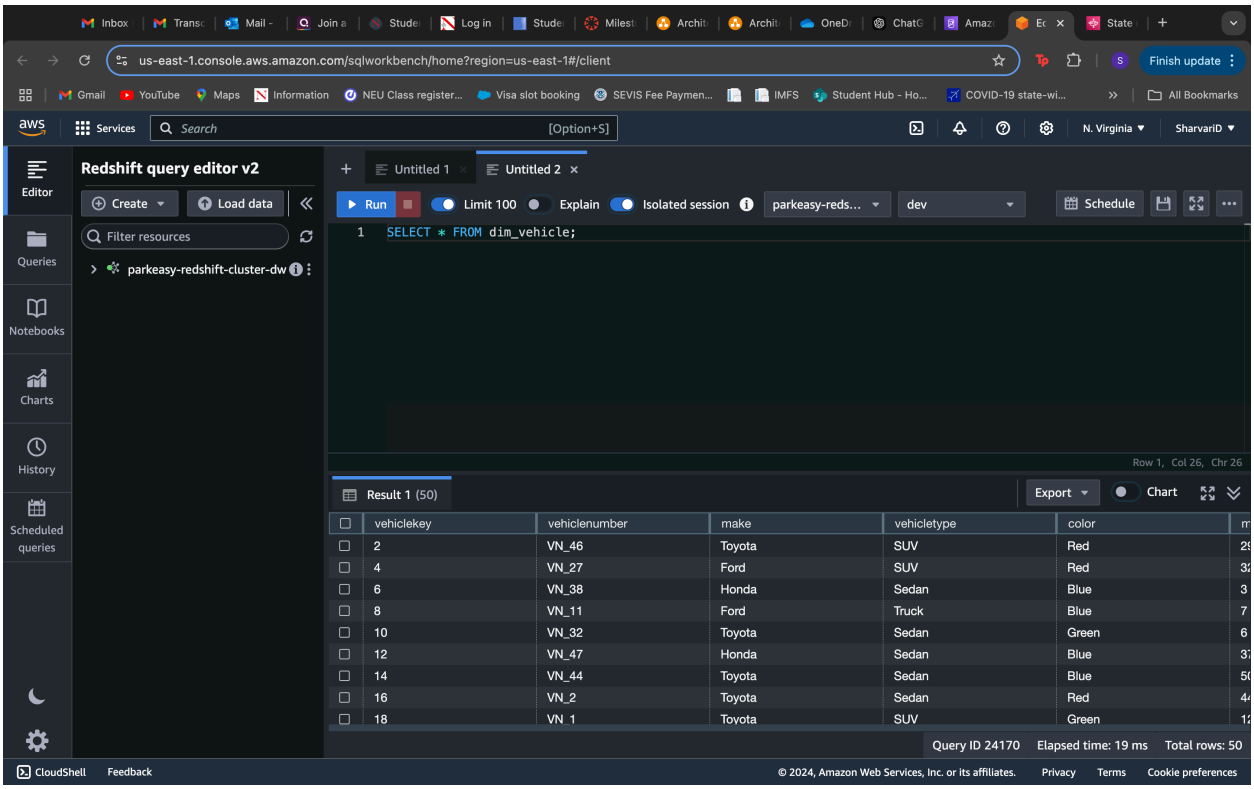
6. Dim_parkingslot table



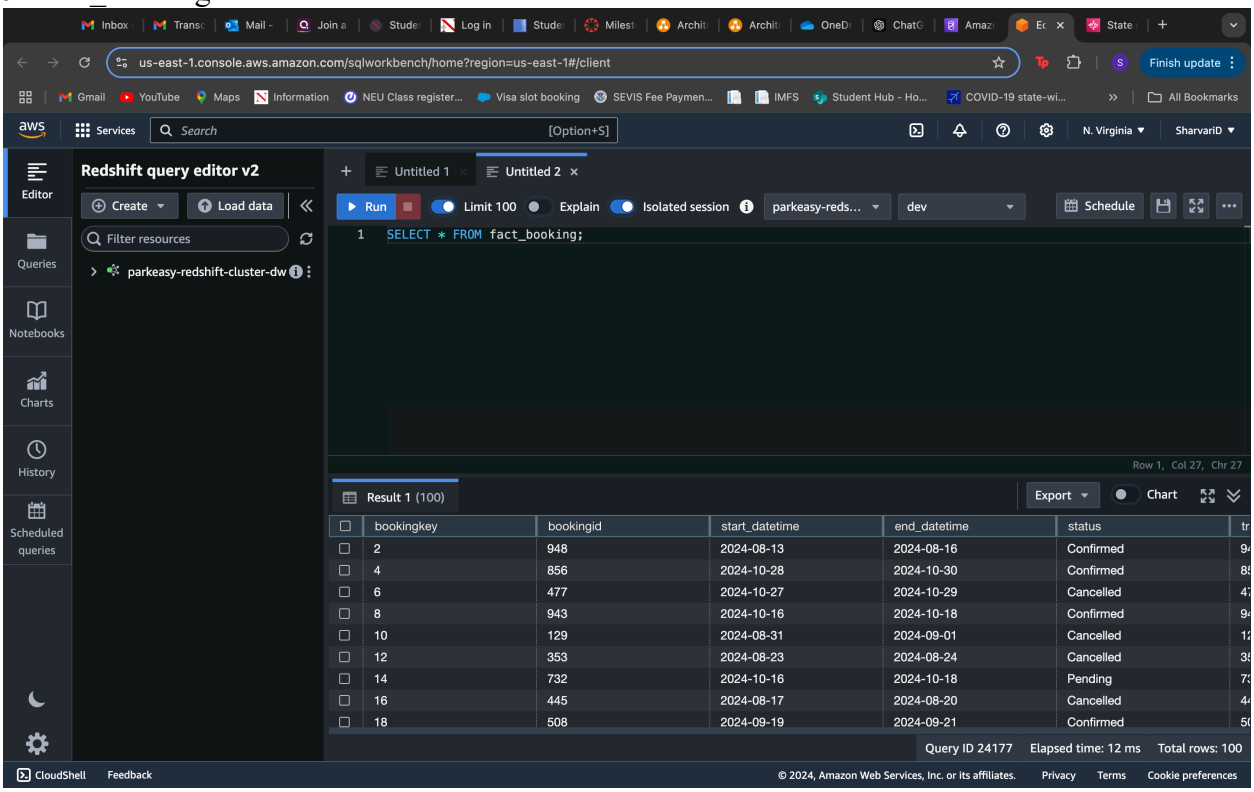
7. Dim_time table



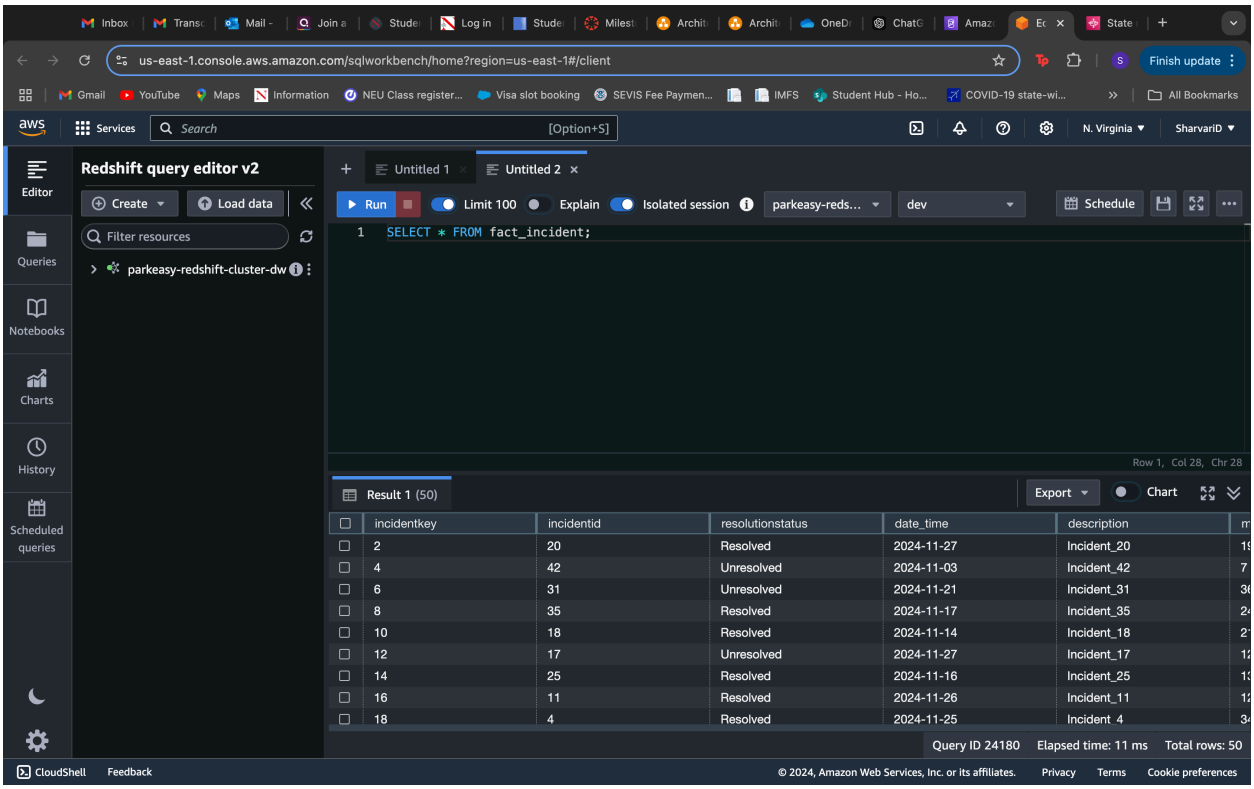
8. Dim_vehicle table



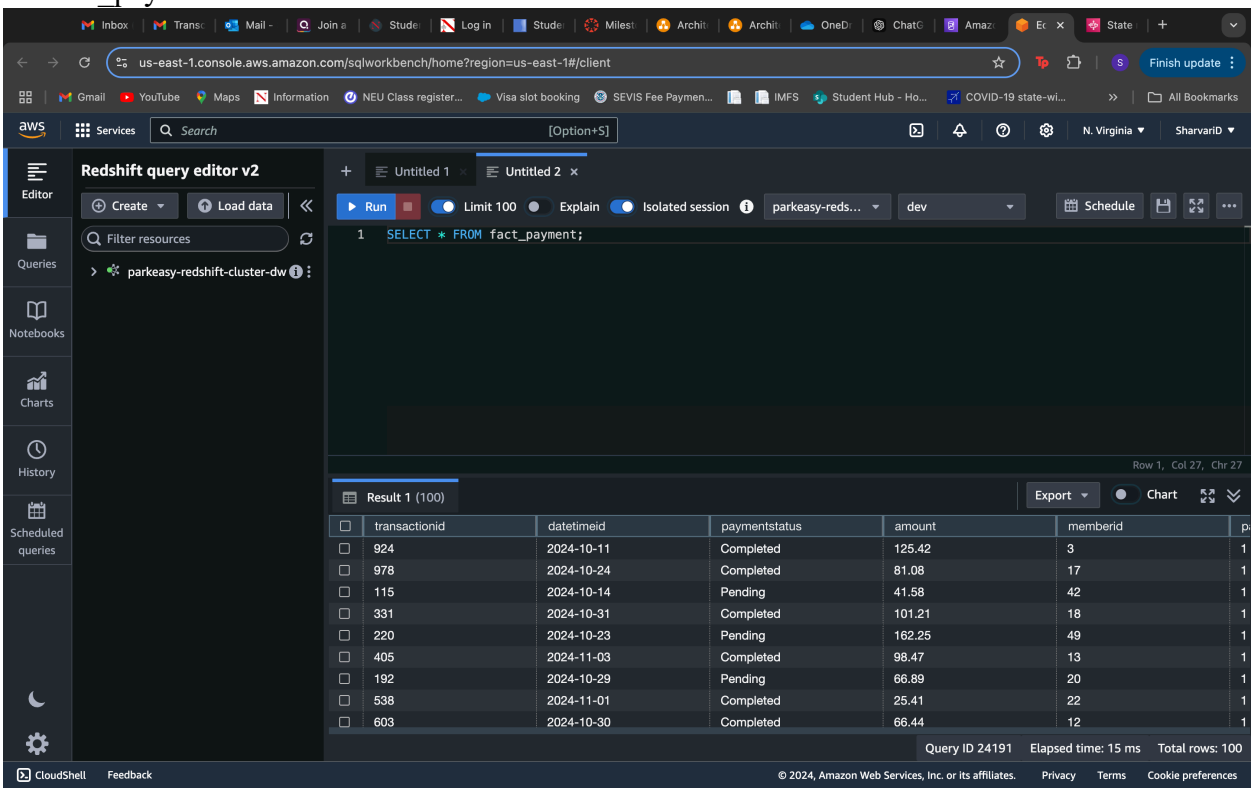
9. Fact_booking table



10. Fact_incident table



11.Fact_payment table



13. Dashboard Creation:

Dashboard 2: Financial Performance & Customer Engagement Report

This dashboard aims to track the financial health of the parking management operations and gauge customer engagement through membership and booking patterns. It focuses on revenue generation, the impact of discount strategies, and customer loyalty, providing a strategic view of business growth and sustainability.

Key Performance Indicators (KPIs)

1. Total Revenue from Bookings (Over Time)
   - Formula: Total Revenue = SUM(Amount) (extracted from FACT_Payment table)
   - Purpose: This indicator provides a longitudinal view of revenue trends, helping to gauge the financial health and growth trajectory of the parking operations.
   - Display: Shown as a cumulative monetary value, this KPI allows for immediate assessment of financial status over selectable time frames.

2. Percentage of Active Memberships
   - Formula: Active Membership % = (COUNT(Active Memberships) / COUNT(Total Memberships)) × 100
   - Purpose: This percentage illuminates the level of engagement and retention of members, serving as a barometer for customer loyalty and the effectiveness of membership programs.
   - Display: Visualized as a percentage, this metric highlights engagement trends and can drive strategies to boost membership renewals and incentives.

3. Average Revenue Per User (ARPU):
   - Formula: ARPU = Total Revenue / Total Active Members.
   - Purpose: Provides a measure of the revenue generated per user, helping to assess the value contributed by each member.
   - Display: As a monetary value, ideally displayed next to total revenue for a quick comparative assessment.

4. Cost of Acquisition vs Lifetime Value:
   - Formula: COA = Total Marketing Spend / Number of New Members; LTV = Average Revenue Per Member × Average Membership Duration.
   - Purpose: These metrics compare the cost of acquiring new members with the revenue they generate over time, indicating the efficiency and sustainability of marketing efforts.
   - Display: Displayed as a dual-line chart for direct comparison.

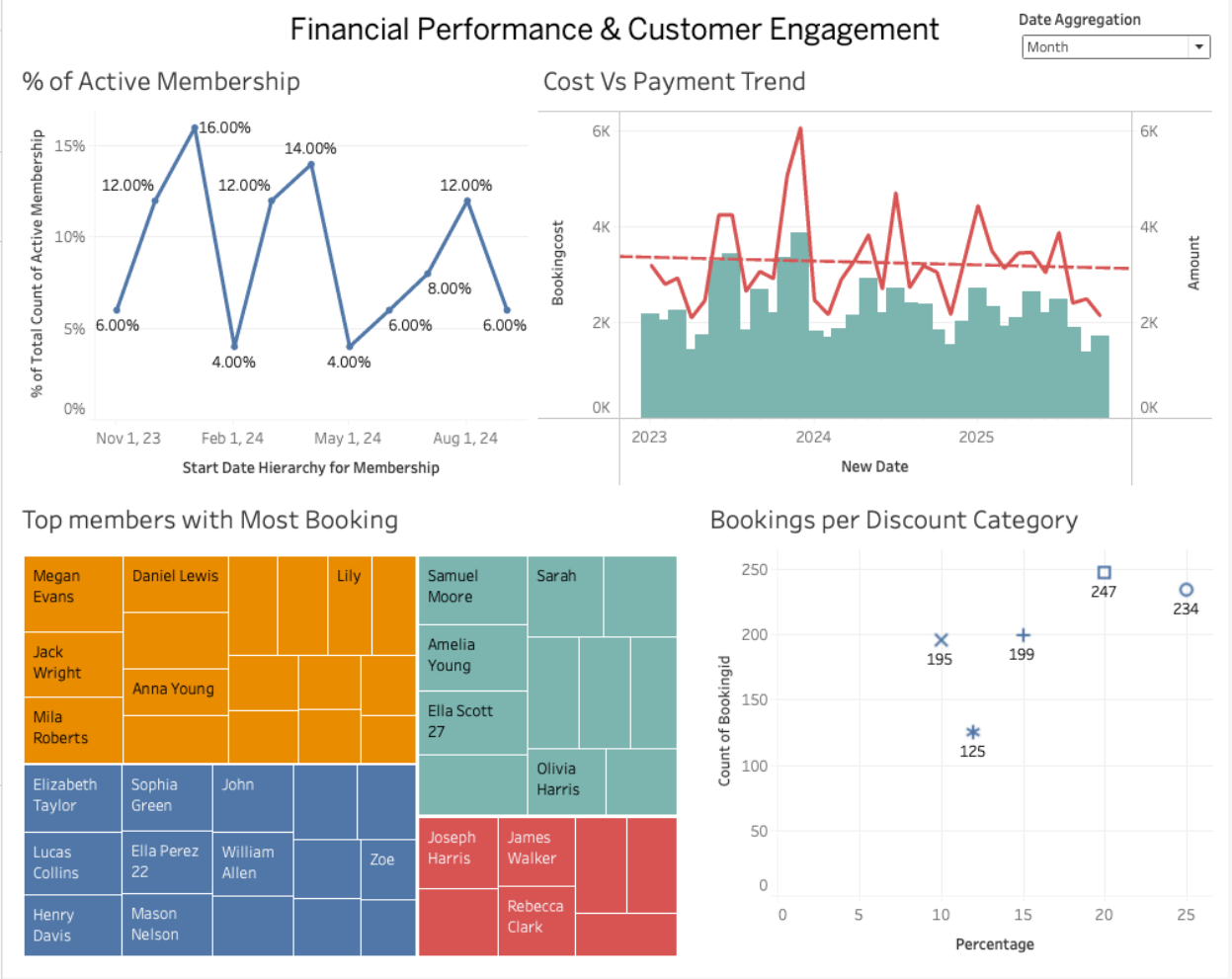
5. Member Churn Rate:
   - Formula: Churn Rate = (Number of Members at Start of Period - Number of Members at End of Period) / Number of Members at Start of Period.
   - Purpose: Measures the rate at which members discontinue their services, crucial for identifying issues in customer satisfaction or competitive challenges.

- Display: Visualized as a simple percentage, highlighting trends over selectable time frames.

6. Utilization Rate by Member Tier:
    - Formula: Utilization Rate = Number of Bookings by Tier / Total Number of Slots Available.
    - Purpose: Assesses how different membership tiers are using the parking facilities, which can help in tailoring services and benefits for different customer segments.
    - Display: Presented as a segmented bar chart for each membership tier.

Visualizations

1. Number of Bookings Per Discount Category (Scatter Plot)
    - Purpose: Analyzes the impact of discount categories on booking behavior, guiding promotional strategies and discount offerings.
    - Key Metric: Bookings Impact from Discounts.
    - Details: The scatter plot categorizes bookings by discount levels, providing a visual correlation between discounts and booking frequency.
2. Members with Most Bookings (Tree Map)
    - Purpose: Identifies top customers by volume of bookings, essential for recognizing and rewarding loyalty and understanding customer value.
    - Key Metric: Top Customers by Bookings.
    - Details: The tree map offers a hierarchical view of members, highlighting those with the highest engagement.
3. Revenue Impact from Discounts (Bar Chart)
    - Purpose: This chart displays the portion of revenue derived from discounted bookings, vital for evaluating the financial impact of discounts on overall earnings.
    - Key Metric: Revenue from Discounted Bookings.
    - Details: By showing revenue from discounted versus full-price bookings, this visualization helps assess the profitability of discount strategies.
4. Repeat Customer Rate (Pie Chart)
    - Purpose: Shows the ratio of repeat to new customers, indicating the success of customer retention efforts and overall satisfaction.
    - Key Metric: Repeat Customer Percentage.
    - Details: The pie chart simplifies the comparison, making it easy to visualize loyalty metrics at a glance.
5. Incident Resolution Rate (Line Chart)
    - Purpose: Monitors the efficiency and timeliness of incident resolutions, critical for maintaining high standards of customer service.
    - Key Metric: Incident Resolution Rate.
    - Details: Trends in incident resolution are tracked over time, highlighting improvements or areas needing attention in customer support processes.



Analysis:

Here's an analysis of the key components in the dashboard:

1. **Percentage of Active Membership**: This line graph shows fluctuations in active membership percentages over time, with significant peaks and troughs. It measures the proportion of members actively using the parking facilities, which is essential for understanding customer retention and loyalty trends.
2. **Cost vs Payment Trend**: This combined line and bar chart tracks both the booking costs and payments over time, with costs generally surpassing payments. This visualization is crucial for assessing the financial sustainability of the operations, identifying periods of revenue shortfall or surplus.
3. **Top Members with Most Bookings**: The tree map highlights the members with the highest number of bookings. This helps in recognizing high-value customers and tailoring marketing strategies to enhance customer loyalty and satisfaction.
4. **Bookings per Discount Category**: This scatter plot shows the relationship between discount levels and the number of bookings, useful for evaluating the effectiveness of different discount strategies in driving bookings.