Super Beep Jeep: The Data Architecture of a Jeepney Transportation Service

Godfrey Bryan Satiada, Nico Rafael Ting, Gregory Uy

Introduction

Background of the Study

In the year 2169, a startup company called BGN Inc. invented a public vehicle called Beep Jeep. This Beep Jeep is operated by the same company, following the traditional jeepney operations. The motivation behind the Beep Jeep is that BGN Inc. believes that there is a better way to run public transportation around Metro Manila, with part of that improvement being achieved through seamless data collection and insight extraction.

BGN Inc. hired students from a certain institute with knowledge about Data Engineering and Data Science/Analytics to create and implement an architecture that best fit their company. This architecture is a complete end-to-end pipeline that starts from data collection from the different jeepneys in operation and other collections, data storing, and finally extraction of insight.

Problem Statement

Jeepney has been the traditional transportation mode in the Philippines, we could even say that all Filipinos have tried riding one. Despite being the long-standing Public Utility Vehicle (PUV) there has been little to no improvement in its operation. Through experience and intuition, drivers know when there will be a traffic in a certain location or where most passengers are picked up or dropped off. Hence, BGN Inc. wants to revolutionize the jeepney operation in Metro Manila by fully utilizing the supposedly collected data to create an efficient fleet management system and passenger demand prediction.

Therefore, the main problem for BGN Inc. lies in creating an end-to-end pipeline for data collection to insight extraction.

Significance of the Study

Modernization of PUVs has always been a hot topic in the Philippines. Drivers, operators, passengers, and government officials have different ideas about what the Philippines will be like once PUV modernization takes place. But modernization of PUVs did not address the inefficiency of its operation.

For BGC Inc., this report's significance was to provide data-driven insights and solutions in fleet management operation and predicting passenger demand. On a broader scale, with this kind of system, BGN Inc. could likely improve the operation of the

modernized jeepneys by using this as a benchmark for an optimized and sustainable PUV solution. Moreover, once data-driven processes and solutions were involved for PUV operation BGN Inc. may influence the policymaking by offering a data-driven perspective on how these solutions could benefit passengers, operators, and drivers.

Scope and Limitation

The project is all about creating and implementing an end-to-end architecture where it would start from raw data extraction (from the jeepneys), data processing, and finally up to dashboard making. The expected databases could be used for different forms of analytics that mainly focus on fleet management and passenger demand analysis.

The project is limited to available resources found in Amazon Web Services (AWS) and to some Apache framework. Finally, given that the project could be implemented in real life, there are some assumptions made for the purpose of this study.

Design Considerations

For the whole implementation to work, the biggest assumption that we have is that each operating Beep Jeep would have its own scanner. This scanner will provide the raw data for all passenger that is being picked up and dropped off by the Beep Jeep. The assumed process of how the scanner works is that, for every passenger who rides the Beep Jeep, the passenger is expected to tap their “Beep Card” once upon boarding and tap again when getting off. In this way, the scanner may be able to collect the necessary information on the passenger and as well as the needed fare amount. Finally, when a Beep Jeep completes a route, the system will collect the data from the scanner.

In addition, the DynamoDB and RDS in this case are what we considered as a backend database. This means that employees of BGN Inc. will input the data through a proper User Interface (UI) to avoid unnecessary dirty input. The purpose of DynamoDB in this case to store driver-related data only, while the Relational Database (RDS) is a prepopulated transactional database about the Beep Jeep operations for a given day, including its assigned route. Essentially, the purpose of RDS is to manage the distribution of the Beep Jeep and as well as making sure that there is no overlapping between the Beep Jeep, routes, and the drivers.

Data Architecture

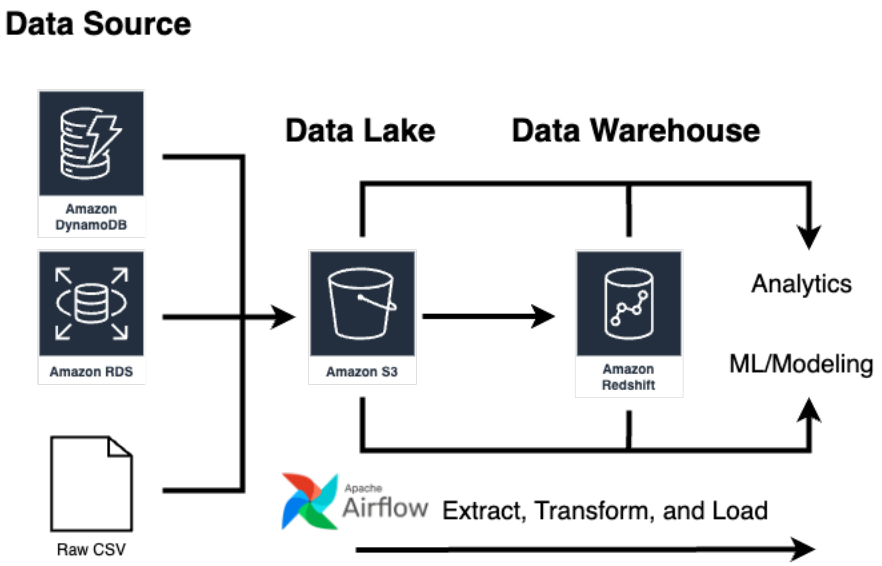


Figure 1. Data Architecture Overview.

The Data Architecture that is shown above follows a simple process. First, we have our Data Sources which are the Amazon DynamoDB, Amazon RDS, and the Raw Data (CSV). The reason why DynamoDB and RDS are considered as Data Sources is because of how they were used for this project. What the BGN Inc. envision is that if there is a driver that wants to work with BGN Inc., they first must register with the company first and since each driver has different informational background, for instance some might have email addresses, while the others do not, then we define the driver's information to be semi- structured. Also, we assume that the driver's informational background is sensitive data , which is why it is stored separately. Therefore, we decided to store Driver's information on DynamoDB.

As stated from the design consideration, the purpose of Amazon RDS is for fleet management. Where it is already pre-populated of when and where should a Beep Jeep be operated (a week ahead). This strategy ensures optimized and efficient allocation of resources and operation. In addition, having an ahead schedule may give enough time for the drivers to familiarize themselves with the schedule and operation that they need to cover, allowing a smoother day-to-day operation. This approach for RDS minimizes last-minute changes which may cause disruption to the service of BGN Inc. Lastly, the raw data that we can obtain from the Beep Jeep.

The Data from the sources would then be extracted, loaded, and transformed into the data lake, which is an S3 Bucket from AWS. S3 Bucket was used simply because of its advantage as a storage. S3 Bucket can accommodate all kinds of data be it structured, semi-structured, and/or unstructured data. Since we are collecting our data from different sources with different structures, the S3 Bucket is just appropriate for the job. Moreover, the S3 Bucket is cheap and durable, which means not only are we paying less, but BGN Inc. is also guaranteed to have strong security with their data.

Once the data is transformed into a data lake, the curated data will then be loaded to the data warehouse which is the Amazon Redshift. Since our data lake is S3 Bucket having Redshift as the Data Warehouse comes with advantage. The integration between S3 Bucket and Redshift is seamless since they are both products of AWS and because of that we expect a smooth loading of data from the data lake to data warehouse. In addition to that we want to utilize the cost-effectiveness of Redshift of having pay-as-you-go pricing model. Since BGN Inc., is just starting with their data-driven insight, solution, and decision making, we want to have everything to be cost-effective strategy since there is a chance that this might be a failure project hence it will be easier for BGN Inc., to pull out.

For a smoother and automated process for Extracting, Loading, and Transforming data from the data source up to data warehouse, Apache Airflow was used to manage the data pipeline.

Finally, Once the data lake and data warehouse were filled by data, Data Scientists/Analysts or any downstream data users may have access (constraint access) to both data lake and data warehouse. By doing this BGN Inc., were able to align its goal on doing passenger demand prediction, moreover it was designed in this way so that the users can have the freedom to explore and experiment with the data which might lead them to find something interesting that could be beneficial for BGN Inc.

Overall, the data architecture of BGN Inc., is a simple end-to-end pipeline. As BGN Inc., is just starting in their journey in data ecosystem having a simple pipeline that is scalable could be an advantage for them as sooner or later they could change some part of it depending on the future challenges and needs that they are seeing. The inflow and outflow of the data were streamlined, the transformation and loading of the data were sufficient for them to generate insights for the earlier part of the operation.

Schemas

Online Transaction Processing (OLTP)

The OLTP system is built using Amazon RDS (Relational Database Service) and designed to manage jeepney-level transactional data. Each transactional record represents a start-to-end jeepney trip.

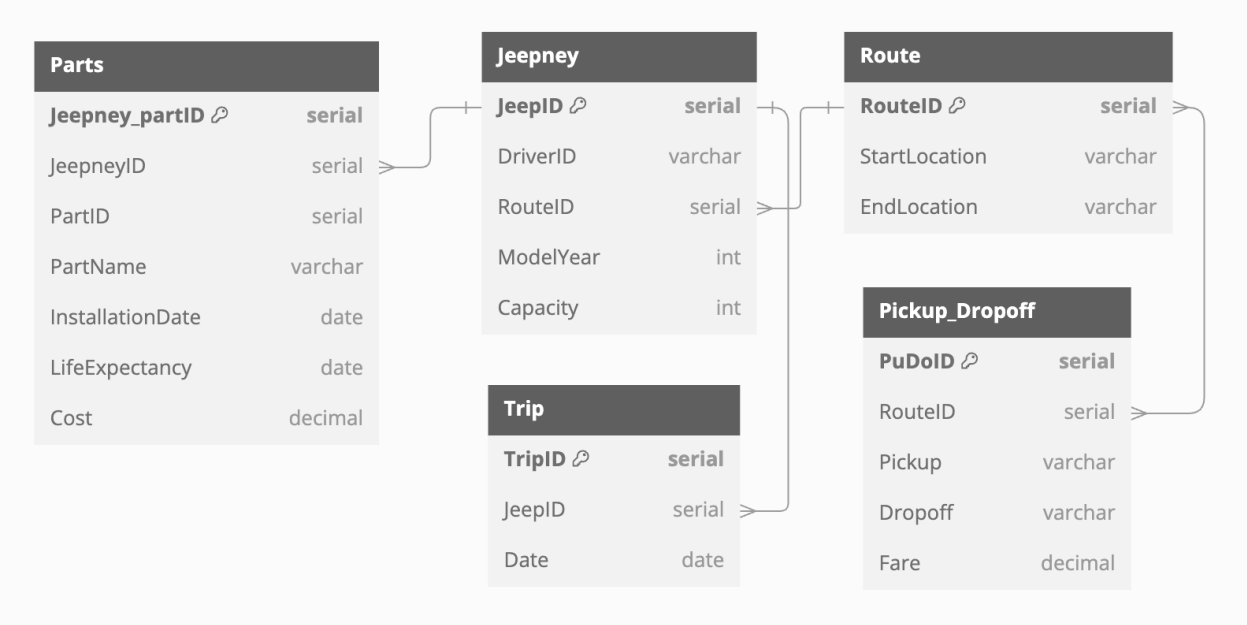


Figure 2. OLTP diagram for Jeepney Operations.

The schema shown in Figure 2 has several entities which represents a key aspect of a jeepney trip. The *Jeepney* table stores information that tracks which jeepney are in use and their details such as *JeepID,* associated driver (*DriverID*), route travelled (*RouteID*), model year, and capacity. The *Trip* table records each individual trip a jeepney makes, linking to a specific jeep through the *JeepID*, and records the date. The *Parts* table provide information about the parts installed in each jeepney such as part name, installation date, life expectancy, and cost. The *Route* table gives details about the various routes in the entire jeepney transportation operation, including the start and end locations. This is linked to the *Pickup_Dropoff* table, which stores data for specific pickup and drop-off points along the routes, as well as the associated fare. Overall, the OLTP schema ensures day-to-day jeepney operations are supported by maintaining accurate record of trips and routes usage.

Online Analytical Processing (OLAP)

We implemented the OLAP system for this project using Amazon Redshift. The OLAP system is designed to analyze passenger-level transactions in a jeepney transportation service. In this case, 1 transaction is a one-way trip of a passenger and the raw data is sourced from a scanner system. The data loaded into the OLAP schema can be used for analysis of operational performance, revenue generation, and passenger behavior.

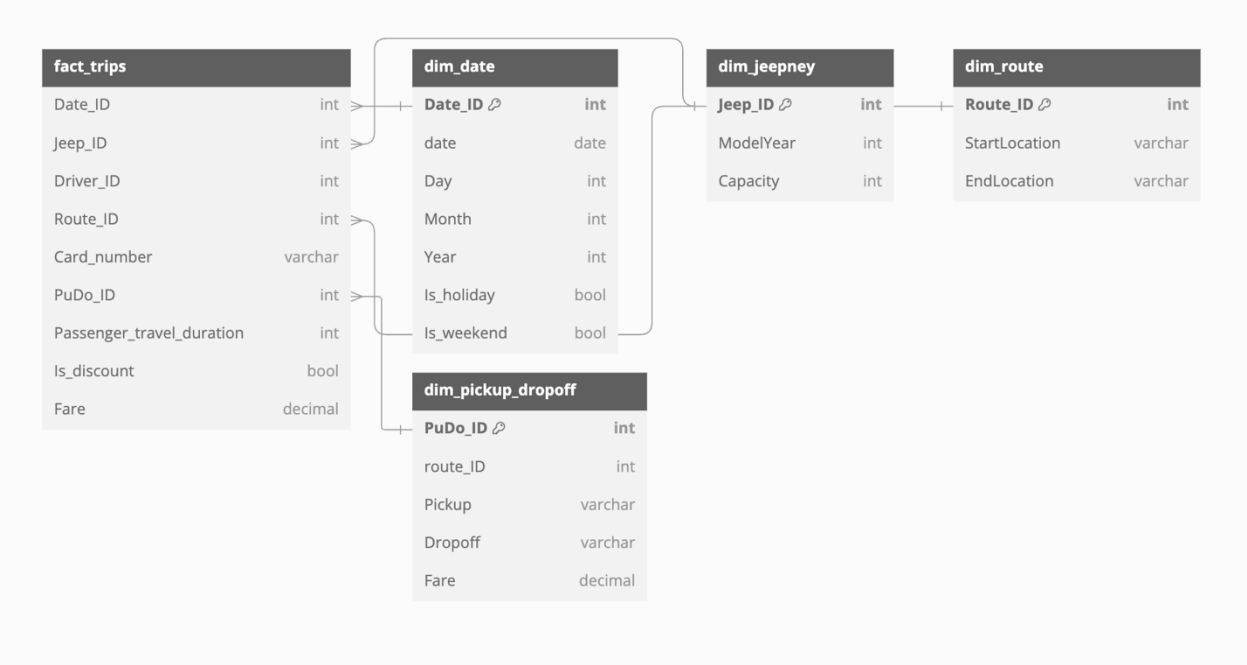


Figure 3. OLAP diagram for Passenger Transactions.

Figure 3 shows the OLAP schema which is centered around the fact table, *fact_trips*, which captures key transactional information such as date of the trip (*date_id)*, jeepney used (*jeep_id*), and route taken (*route_id*). It also provides passenger-specific metrics including travel duration, discount information, and fare. Surrounding the fact table are dimension tables which gives additional context to the trip. The *dim_date* entity is a prepopulated table that gives temporal information, with attributes including day, month, year, and indicators for holidays and weekends. This allows easy time-based analysis to explore trends and seasonality. The *dim_jeepney* table includes information of the jeepney used in the trips like model year and capacity. In addition, the *dim_route* table provides data about the start and end point of the routes. Lastly, *dim_pickup_dropoff* table adds granularity by recording the pickup and drop-off points within each route, along with the fare associated. Together, this schema provides a structure that will be beneficial in analyzing passenger transactions and enable data-driven decisions in areas like route optimization and operational efficiency.

NoSQL

We used a NoSQL database, specifically Amazon DynamoDB, to manage the driver-related information efficiently.

Figure 4. NoSQL key-value store for Drivers database

This schema shown in Figure 4, demonstrates how each driver information is stored. This schema follows a key-value store approach where *driver_id* serves as the key for each unique entry. The value associated with this key includes *birthday, contact_number, first_name, last_name, and license_number*.

Table 1. Key-value description and data type from the driver NoSQL database.

Column Name	**Description**	**Data Type**
driver_id	Unique identifier for each driver.	Number
birthday	Driver's date of birth.	String
contact_number	Driver's contact phone number.	String
first_name	Driver's first name.	String
last_name	Driver's last name.	String
license_number	Driver's license number.	String

Table 1 provide the detailed overview of the schema components, including the description and data type used. The *driver_id* is a number that uniquely identifies each driver, while the remaining attributes are information associated for the driver and stored as a string.

Data Lake

Data Lake Zones

For this project, we chose Amazon S3 as our data lake due to its ability to handle large volume of data in different formats. This allows us to store structured and unstructured data from different sources such as PostgreSQL, DynamoDB, and the csv data from the scanner system. This flexibility is crucial to accommodate future changes in data sources or formats, without needing significant infrastructure modifications. Also, the data lake will serve as a central repository for all data, giving access for different users with their own needs while allowing robust security and access controls. Figure 5 shows the overall structure of the data lake and the flow of data through its zones.

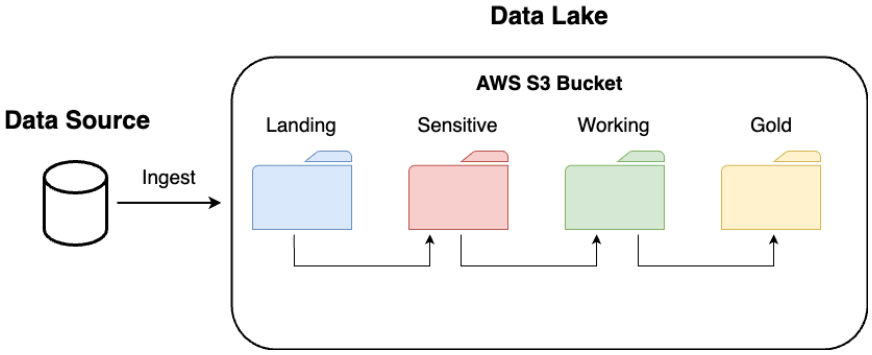


Figure 5. Data Lake Zones Overview.

The data lake consists of four zones – Landing, Sensitive, Working, Gold – each operating a role in the data processing pipeline. The Landing zone is the initial repository for raw data ingested from PostgreSQL database (containing the jeepney operations data), DynamoDB (containing the driver-related information), and the csv data (containing the passenger trips data). The raw data in the Landing zone is unprocessed and this zone serves as a staging area for further validation and transformation. Once the data is validated, they are moved into the Sensitive zone, where the process of merging and removing of sensitive information occur. Only the merged data with sensitive information such as driver details are stored in this zone. This separation helps maintain privacy and ensures compliance to regulations. Meanwhile, the merged data without the sensitive information is moved and stored in the Working zone for further cleaning and processing in preparation for analysis or other use case. The final zone, Gold zone, contains fully processed, clean, and deduplicated data that is ready for consumption by the analysts or data scientists.

Access Control

Access to the data lake zones are strictly regulated to ensure security, privacy, and data governance. Each types of users has their distinct access to zones in the data lake based on their role and data needs. Table 1 below summarizes the access granted to each user type across the four zones.

Table 2. User Access to Data Lake Zones.

Role	Landing	Sensitive	Working	Gold
Data Engineers	Full	Full	Full	Full
Data Scientists	Full	No Access	No Access	Full
Data Stewards	No Access	Full	No Access	No Access
Business Analysts	No Access	No Access	No Access	Full

As shown in the table, Data Engineers have full access to all zones, allowing them to manage the data flow from start to finish. They are responsible to oversee the ingestion, transformation, and storage of data, as well as troubleshooting any issue that arise in the data pipeline. Data Scientists are granted access to both Landing and Gold zones. The access in the Landing zone allows them to have flexibility of applying custom data preprocessing or conducting exploratory data analysis on the raw data. Meanwhile, access on the Gold zone allows them with fully processed and cleaned datasets. Data Stewards have only access to Sensitive zone due to their primary responsibility of maintaining data privacy and compliance with regulations. Lastly, Business Analysts are only allowed to access the Gold zone, where they interact with fully processed and non-sensitive data that is ready for reporting and business intelligence tasks.

Extract-Transform-Load (ETL) Jobs

The ETL pipeline for this project is managed using Apache Airflow, which allowed automation of several tasks. The pipeline is set to run a daily schedule, ensuring the extraction, transformation, and loading occur at a regular interval to keep the data updated. The tasks within the pipeline are organized as a Directed Acyclic Graph (DAG), with each task is dependent on the completion of the previous task. The figure below illustrates the DAG of the ETL pipeline.

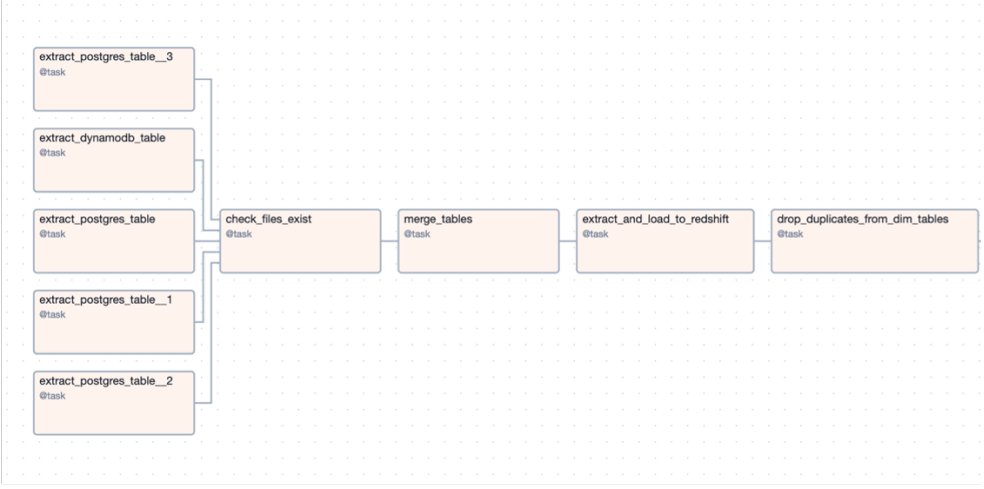


Figure 6. DAG of the ETL pipeline.

As shown in Figure 6, the process starts with extracting the data from the three primary sources: PostgreSQL, DynamoDB, and user-uploaded csv file. The PostgreSQL table provide the data related to the jeepney operations, routes, and trips, while the DynamoDB table gives the driver-related information. The third data is from the scanner , which produces the csv file containing the passenger trips data. It is assumed that the csv file is uploaded directly into the landing zone in the S3 bucket.

After extracting the data, a task verifies the presence of all the required files. This ensures that the necessary files from the three sources are successfully stored in the S3 bucket. This task will raise an error if any file is missing, preventing further execution to maintain data integrity.

Next, the data from the different sources is merged. This process creates a unified dataset that contains non-sensitive information and sensitive information. The merged dataset with sensitive information is stored to the sensitive zone in the S3 bucket. Then, after removing any personal identifiable information, the merged dataset is stored to the working zone for further transformation as needed for the analysis.

With the merged dataset, the next task loads the cleaned data into Amazon Redshift, inserting records into fact and dimension tables. Before the final data load, a deduplication task ensures that only unique records are stored in the database.

Conclusion

In summary, the project successfully designed and implemented an end-to-end data architecture for BGN Inc.'s Beep Jeep operations. This includes data collection, storage, and analysis through OLTP, OLAP, NoSQL database and data lake. Leveraging the capabilities of the Amazon products and services, specifically Amazon RDS for transactional data, Amazon Redshift for analytical processing, Amazon DynamoDB for key-value storage, and Amazon S3 for the data lake, the system was able to efficiently manage structured and unstructured data. The use of Apache Airflow to automate the ETL pipeline ensured a smooth and reliable process for data extraction, transformation, and loading, which helps the data be updated and consistent. This data architecture allows BGN Inc. to gain valuable insights into the jeepney operations and passenger behavior, which can be used to optimize routes, improve jeepney fleet efficiency, and make data-driven decisions.

Recommendation

Data Architecture was created to support the current needs of a company. Currently BGN Inc. is just starting with their data ecosystem so having a simple end-to-end pipeline would be enough. However, as more data is being ingested, the simple architecture would

not be able to hold support it off. Therefore, we recommend that as early as now, BGN Inc. should set a benchmark on how much data their pipeline can handle without trading time complexity. So that when the benchmark or maximum amount of data is met, then BGN Inc. could scale their data architecture.

We also recommend exploring different databases and workflows. What we implemented in this project are the well-known components for such pipeline. There could be some other components that could align with the business values and goals.

Lastly, we recommend that BGN Inc. establish a robust data governance and monitoring framework. This could help define data ownership and data quality assurance. By doing so, this ensures that all the data that is being processed can be consistent, accurate, and secure,

Reference

Andalecio, A. B. P., Aquino, K. E. C., Cruz, C. F. A., De Guzman, A., & Kiong, N. T. C. (2020). Implementation, challenges and stakeholders perception of modernized Jeepneys in Metro Manila. *Journal of Physics Conference Series*, *1529*(3), 032067. https://doi.org/10.1088/1742-6596/1529/3/032067

Appendix

Appendix A. Source Data Sample

Passenger Trips Data

	date	jeep_id	driver_id	route_id	pudo_id	card_id	passenger_travel_duration	is_discount	fare
0	2171-04-05	55	138	1	25	RE8205851737659	4	False	1500.0
1	2171-04-05	55	138	1	10	RE3439173643229	1	False	1900.0
2	2171-04-05	55	138	1	42	RE6168121069988	1	False	1100.0
3	2171-04-05	55	138	1	21	RE3435294006362	1	False	1100.0
4	2171-04-05	55	138	1	48	DS4491613279068	3	True	960.0

Jeepney Trips Data

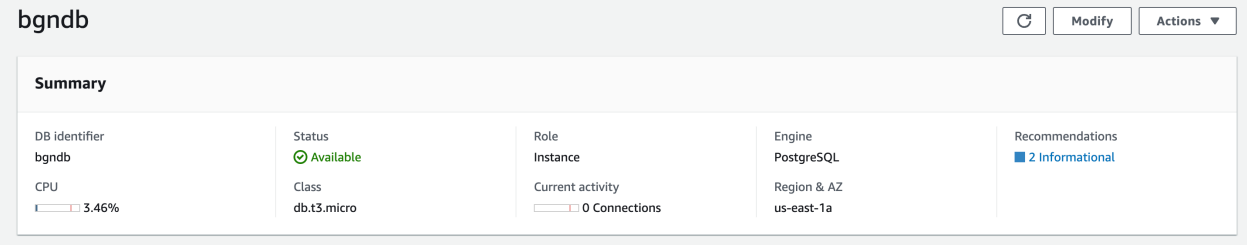
	trip_id	jeep_id	date	model_year	capacity	route_id	jeep_part_id	part_id	part_name	installation_date	life_expectancy	cost	start_location	end_location	pudo_id	pickup	dropoff	fare
0	1	1	05-01-2170	2171	20	1	1	1	Engine	05-01-2171	05-01-2181	2500	startloc1	endloc1	1.0	1.0	1.0	1000.0
1	1	1	05-01-2170	2171	20	1	1	1	Engine	05-01-2171	05-01-2181	2500	startloc1	endloc1	2.0	1.0	2.0	1100.0
2	1	1	05-01-2170	2171	20	1	1	1	Engine	05-01-2171	05-01-2181	2500	startloc1	endloc1	3.0	1.0	3.0	1200.0
3	1	1	05-01-2170	2171	20	1	1	1	Engine	05-01-2171	05-01-2181	2500	startloc1	endloc1	4.0	1.0	4.0	1300.0
4	1	1	05-01-2170	2171	20	1	1	1	Engine	05-01-2171	05-01-2181	2500	startloc1	endloc1	5.0	1.0	5.0	1400.0

Driver Data

	driver_id	first_name	last_name	birthday	contact_number	license_number
0	1	Anthony	Mora	2139-07-19	0110625682593	LN2932440061784
1	2	Evelyn	Tyler	2139-08-19	7092285207826	LN7355721182256
2	3	Jacob	Lawson	2132-10-06	7480859977526	LN5058711195683
3	4	Samuel	Zamora	2138-01-20	8958518364996	LN8842179042003
4	5	Nicole	Williams	2133-03-31	6678756406662	LN8197318112597

Appendix B. OLTP

Appendix B-1. RDS Details



Appendix B-2. SQL Script

```
%load_ext sql
```

```
with open('connection_string.txt') as f:
    connection_string = f.read().strip()
```

```
from sqlalchemy import create_engine
engine = create_engine(connection_string)
```

```
%sql engine
```

Create tables

```
%%sql

CREATE TABLE IF NOT EXISTS jeepney (
    jeep_id SERIAL PRIMARY KEY,
    route_id INT,
    model_year INT,
    capacity INT
);
```

Running query in 'postgresql://postgres:***@bgndb.cjau20mowvcs.us-east-1.rds.amazonaws.com/postgres'

```
%%sql

CREATE TABLE IF NOT EXISTS route (
    route_id SERIAL PRIMARY KEY,
    start_location VARCHAR,
    end_location VARCHAR
);
```

Running query in 'postgresql://postgres:***@bgndb.cjau20mowvcs.us-east-1.rds.amazonaws.com/postgres'

```
%%sql

CREATE TABLE IF NOT EXISTS pickup_dropoff (
    pudo_id SERIAL PRIMARY KEY,
    route_id INT REFERENCES route(route_id),
    pickup VARCHAR,
    dropoff VARCHAR,
    fare DECIMAL
);
```

Running query in 'postgresql://postgres:***@bgndb.cjau20mowvcs.us-east-1.rds.amazonaws.com/postgres'

```
%%sql

CREATE TABLE IF NOT EXISTS trips (
    trip_id SERIAL PRIMARY KEY,
    jeep_id INT REFERENCES jeepney(jeep_id),
    date DATE
);
```

Running query in 'postgresql://postgres:***@bgndb.cjau20mowvcs.us-east-1.rds.amazonaws.com/postgres'

```
%%sql

CREATE TABLE IF NOT EXISTS parts (
    jeep_part_id SERIAL PRIMARY KEY,
    jeep_id INT REFERENCES jeepney(jeep_id),
    part_id INT,
    part_name VARCHAR,
    installation_date DATE,
    life_expectancy DATE,
    cost DECIMAL
);
```

Running query in 'postgresql://postgres:***@bgndb.cjau20mowvcs.us-east-1.rds.amazonaws.com/postgres'

Appendix B-3. List of Tables

```
%sql \dt
```

Running query in 'postgresql://postgres:***@bgndb.cjau20mowvcs.us-east-1.rds.amazonaws.com/postgres'

Schema	Name	Type	Owner
public	jeepney	table	postgres
public	parts	table	postgres
public	pickup_dropoff	table	postgres
public	route	table	postgres
public	trips	table	postgres

Appendix B-4. Schema of Each Table

```
%sqlcmd columns -t jeepney
```

name	type	nullable	default	autoincrement	comment
jeep_id	INTEGER	False	nextval('jeepney_jeep_id_seq'::regclass)	True	None
route_id	INTEGER	True	None	False	None
model_year	INTEGER	True	None	False	None
capacity	INTEGER	True	None	False	None

```
%sqlcmd columns -t parts
```

name	type	nullable	default	autoincrement	comment
jeep_part_id	INTEGER	False	nextval('parts_jeep_part_id_seq'::regclass)	True	None
jeep_id	INTEGER	True	None	False	None
part_id	INTEGER	True	None	False	None
part_name	VARCHAR	True	None	False	None
installation_date	DATE	True	None	False	None
life_expectancy	DATE	True	None	False	None
cost	NUMERIC	True	None	False	None

```
%sqlcmd columns -t pickup_dropoff
```

name	type	nullable	default	autoincrement	comment
pudo_id	INTEGER	False	nextval('pickup_dropoff_pudo_id_seq'::regclass)	True	None
route_id	INTEGER	True	None	False	None
pickup	VARCHAR	True	None	False	None
dropoff	VARCHAR	True	None	False	None
fare	NUMERIC	True	None	False	None

```
%sqlcmd columns -t route
```

name	type	nullable	default	autoincrement	comment
route_id	INTEGER	False	nextval('route_route_id_seq'::regclass)	True	None
start_location	VARCHAR	True	None	False	None
end_location	VARCHAR	True	None	False	None

```
%sqlcmd columns -t trips
```

name	type	nullable	default	autoincrement	comment
trip_id	INTEGER	False	nextval('trips_trip_id_seq'::regclass)	True	None
jeep_id	INTEGER	True	None	False	None
date	DATE	True	None	False	None

Appendix B-5. Overview of Each Table

```
%sql SELECT * FROM jeepney LIMIT 5;
```

Running query in 'postgresql://postgres:***@bgndb.cjau20mowvcs.us-east-1.rds.amazonaws.com/postgres'
5 rows affected.

jeep_id	route_id	model_year	capacity
1	1	2171	20
2	2	2169	16
3	3	2170	18
4	4	2170	18
5	5	2171	20

```
%sql SELECT * FROM parts LIMIT 5;
```

Running query in 'postgresql://postgres:***@bgndb.cjau20mowvcs.us-east-1.rds.amazonaws.com/postgres'
5 rows affected.

jeep_part_id	jeep_id	part_id	part_name	installation_date	life_expectancy	cost
1	1	1	Engine	2171-05-01	2181-05-01	2500
2	1	2	Transmission	2171-05-01	2181-05-01	2500
3	1	3	Brakes	2171-05-01	2181-05-01	2500
4	1	4	Tires	2171-05-01	2181-05-01	2500
5	1	5	Suspension	2171-05-01	2181-05-01	2500

```
%sql SELECT * FROM pickup_dropoff LIMIT 5;
```

Running query in 'postgresql://postgres:***@bgndb.cjau20mowvcs.us-east-1.rds.amazonaws.com/postgres'
5 rows affected.

pudo_id	route_id	pickup	dropoff	fare
1	1	1	1	1000
2	1	1	2	1100
3	1	1	3	1200
4	1	1	4	1300
5	1	1	5	1400

```
%sql SELECT * FROM route LIMIT 5;
```

Running query in 'postgresql://postgres:***@bgndb.cjau20mowvcs.us-east-1.rds.amazonaws.com/postgres'
5 rows affected.

route_id	start_location	end_location
1	startloc1	endloc1
2	startloc2	endloc2
3	startloc3	endloc3
4	startloc4	endloc4
5	startloc5	endloc5

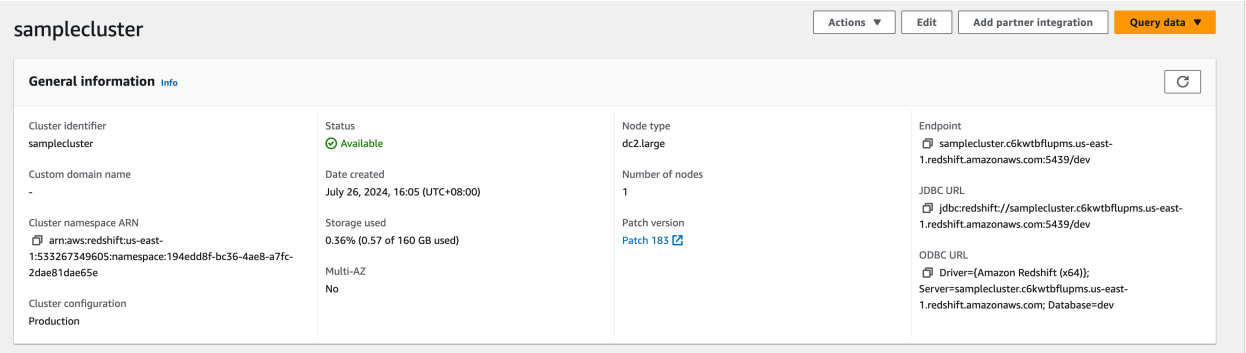
```
%sql SELECT * FROM trips LIMIT 5;
```

Running query in 'postgresql://postgres:***@bgndb.cjau20mowvcs.us-east-1.rds.amazonaws.com/postgres'
5 rows affected.

trip_id	jeep_id	date
1	1	2170-05-01
2	2	2170-05-01
3	3	2170-05-01
4	4	2170-05-01
5	5	2170-05-01

Appendix C. OLAP

Appendix C-1. Redshift Details



Appendix C-2. SQL Script

```
from sqlalchemy import create_engine
```

```
%load_ext sql
```

```
%config SqlMagic.named_parameters = "enabled"
%config SqlMagic.displaylimit = None
```

displaylimit: Value None will be treated as 0 (no limit)

```
from sqlalchemy import create_engine
with open('connection_string_redshift.txt') as f:
    engine2 = create_engine(f.read().strip())
```

```
%sql engine2
```

```
%sql CREATE DATABASE BGNrsdb;
```

Running query in 'redshift://awsuser:***@samplecluster.c6kwtbflupms.us-east-1.redshift.amazonaws.com:5439/dev'

```
with open('connection_string_redshift2.txt') as f:
    engine3 = create_engine(f.read().strip())
```

Create tables

dim_date

```
%%sql
CREATE TABLE IF NOT EXISTS dim_date (
    date_id INTEGER NOT NULL,
    date DATE NOT NULL,
    day INTEGER NOT NULL,
    month INTEGER NOT NULL,
    year INTEGER NOT NULL,
    is_holiday BOOL NOT NULL,
    is_weekend BOOL NOT NULL
)
```

Running query in 'redshift://awsuser:***@samplecluster.c6kwtbflupms.us-east-1.redshift.amazonaws.com:5439/bgnrsdb'

dim_pickup_dropoff

```
%%sql
CREATE TABLE IF NOT EXISTS dim_pickup_dropoff (
    pudo_id INTEGER NOT NULL,
    route_id INTEGER NOT NULL,
    pickup VARCHAR(100) NOT NULL,
    dropoff VARCHAR(100) NOT NULL,
    fare DECIMAL NOT NULL
)
```

Running query in 'redshift://awsuser:***@samplecluster.c6kwtbflupms.us-east-1.redshift.amazonaws.com:5439/bgnrsdb'

dim_jeepney

```
%%sql
CREATE TABLE IF NOT EXISTS dim_jeepney (
    jeepney_id INT NOT NULL,
    model_year INT NOT NULL,
    capacity INT NOT NULL
)
```

Running query in 'redshift://awsuser:***@samplecluster.c6kwtbflupms.us-east-1.redshift.amazonaws.com:5439/bgnrsdb'

dim_route

```
%%sql
CREATE TABLE IF NOT EXISTS dim_route (
    route_id INTEGER NOT NULL,
    start_location VARCHAR(100) NOT NULL,
    end_location VARCHAR(1000) NOT NULL
)
```

Running query in 'redshift://awsuser:***@samplecluster.c6kwtbflupms.us-east-1.redshift.amazonaws.com:5439/bgnrsdb'

fact_trips

```
%%sql
CREATE TABLE IF NOT EXISTS fact_trips (
    date VARCHAR(100) NOT NULL,
    jeep_id INTEGER NOT NULL,
    driver_id INTEGER NOT NULL,
    route_id INTEGER NOT NULL,
    pudo_id INTEGER NOT NULL,
    card_number VARCHAR(100) NOT NULL,
    passenger_travel_duration INTEGER NOT NULL,
    is_discount BOOLEAN NOT NULL,
    fare DECIMAL NOT NULL
)
DISTSTYLE EVEN
COMPOUND SORTKEY (jeep_id, route_id)
```

Running query in 'redshift://awsuser:***@samplecluster.c6kwtbflupms.us-east-1.redshift.amazonaws.com:5439/bgnrsdb'

Appendix C-3. List of Tables

```
%sql \dt
```

Running query in 'redshift://awsuser:***@samplecluster.c6kwtbflupms.us-east-1.redshift.amazonaws.com:5439/bgnrsdb'

schema	name	type	owner
public	dim_date	table	awsuser
public	dim_jeepney	table	awsuser
public	dim_jeepney_part	table	awsuser
public	dim_pickup_dropoff	table	awsuser
public	dim_route	table	awsuser
public	fact_trips	table	awsuser

Appendix C-4. Schema of Each Table

```
%sqlcmd columns -t dim_date
```

name	type	nullable	default	autoincrement	comment	info
date	DATE	True	None	False	None	{'encode': 'az64'}
date_id	BIGINT	True	None	False	None	{'encode': 'az64'}
day	INTEGER	True	None	False	None	{'encode': 'az64'}
month	INTEGER	True	None	False	None	{'encode': 'az64'}
year	INTEGER	True	None	False	None	{'encode': 'az64'}
is_holiday	BOOLEAN	True	None	False	None	{}
is_weekday	BOOLEAN	True	None	False	None	{}

```
%sqlcmd columns -t dim_jeepney
```

name	type	nullable	default	autoincrement	comment	info
jeepney_id	INTEGER	False	None	False	None	{'encode': 'az64'}
model_year	INTEGER	False	None	False	None	{'encode': 'az64'}
capacity	INTEGER	False	None	False	None	{'encode': 'az64'}

```
%sqlcmd columns -t dim_pickup_dropoff
```

name	type	nullable	default	autoincrement	comment	info
pudo_id	INTEGER	False	None	False	None	{'encode': 'az64'}
route_id	INTEGER	False	None	False	None	{'encode': 'az64'}
pickup	VARCHAR(100)	False	None	False	None	{'encode': 'lzo'}
dropoff	VARCHAR(100)	False	None	False	None	{'encode': 'lzo'}
fare	NUMERIC(18, 0)	False	None	False	None	{'encode': 'az64'}

```
%sqlcmd columns -t dim_route
```

name	type	nullable	default	autoincrement	comment	info
route_id	INTEGER	False	None	False	None	{'encode': 'az64'}
start_location	VARCHAR(100)	False	None	False	None	{'encode': 'lzo'}
end_location	VARCHAR(1000)	False	None	False	None	{'encode': 'lzo'}

```
%sqlcmd columns -t fact_trips
```

name	type	nullable	default	autoincrement	comment	info
date	VARCHAR(100)	False	None	False	None	{'encode': 'lzo'}
jeep_id	INTEGER	False	None	False	None	{}
driver_id	INTEGER	False	None	False	None	{'encode': 'az64'}
route_id	INTEGER	False	None	False	None	{}
pudo_id	INTEGER	False	None	False	None	{'encode': 'az64'}
card_number	VARCHAR(100)	False	None	False	None	{'encode': 'lzo'}
passenger_travel_duration	INTEGER	False	None	False	None	{'encode': 'az64'}
is_discount	BOOLEAN	False	None	False	None	{}
fare	NUMERIC(18, 0)	False	None	False	None	{'encode': 'az64'}

Appendix C-5. Overview of Each Table

```
%sql SELECT * FROM dim_date LIMIT 5;
```

Running query in 'redshift://awsuser:***@samplecluster.c6kwtbflupms.us-east-1.redshift.amazonaws.com:5439/bgnrsdb'
5 rows affected.

date	date_id	day	month	year	is_holiday	is_weekday
2169-01-01	21690101	1	1	2169	False	False
2169-01-02	21690102	2	1	2169	False	True
2169-01-03	21690103	3	1	2169	False	True
2169-01-04	21690104	4	1	2169	False	True
2169-01-05	21690105	5	1	2169	False	True

```
%sql SELECT * FROM dim_pickup_dropoff LIMIT 5;
```

Running query in 'redshift://awsuser:***@samplecluster.c6kwtbflupms.us-east-1.redshift.amazonaws.com:5439/bgnrsdb'
5 rows affected.

pudo_id	route_id	pickup	dropoff	fare
25	1	3	8	1500
10	1	1	10	1900
42	1	6	7	1100
21	1	3	4	1100
48	1	7	9	1200

```
%sql SELECT * FROM dim_jeepney LIMIT 5;
```

Running query in 'redshift://awsuser:***@samplecluster.c6kwtbflupms.us-east-1.redshift.amazonaws.com:5439/bgnrsdb'
5 rows affected.

jeepney_id	model_year	capacity
55	2171	20
55	2171	20
55	2171	20
55	2171	20
55	2171	20

```
%sql SELECT * FROM dim_route LIMIT 5;
```

Running query in 'redshift://awsuser:***@samplecluster.c6kwtbflupms.us-east-1.redshift.amazonaws.com:5439/bgnrsdb'
5 rows affected.

route_id	start_location	end_location
1	startloc1	endloc1
1	startloc1	endloc1
1	startloc1	endloc1
1	startloc1	endloc1
1	startloc1	endloc1

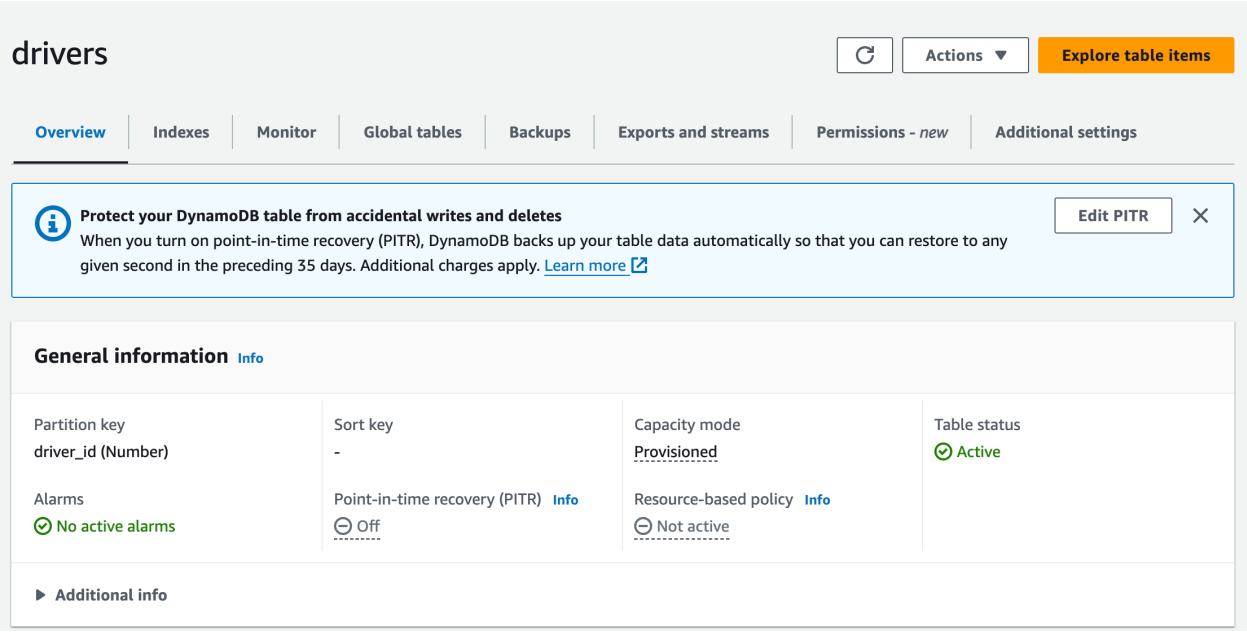
```
%sql SELECT * FROM fact_trips LIMIT 5;
```

Running query in 'redshift://awsuser:***@samplecluster.c6kwtbflupms.us-east-1.redshift.amazonaws.com:5439/bgnrsdb'
5 rows affected.

date	jeep_id	driver_id	route_id	pudo_id	card_number	passenger_travel_duration	is_discount	fare
2171-04-05	55	138	1	25	RE8205851737659	4	False	1500
2171-04-05	55	138	1	10	RE3439173643229	1	False	1900
2171-04-05	55	138	1	4	RE0932180567906	3	False	1300
2171-04-05	55	138	1	30	RE3479128371805	2	False	1200
2171-04-05	55	138	1	25	RE0875637963253	2	False	1500

Appendix D. NoSQL

Appendix D-1. DynamoDB Details

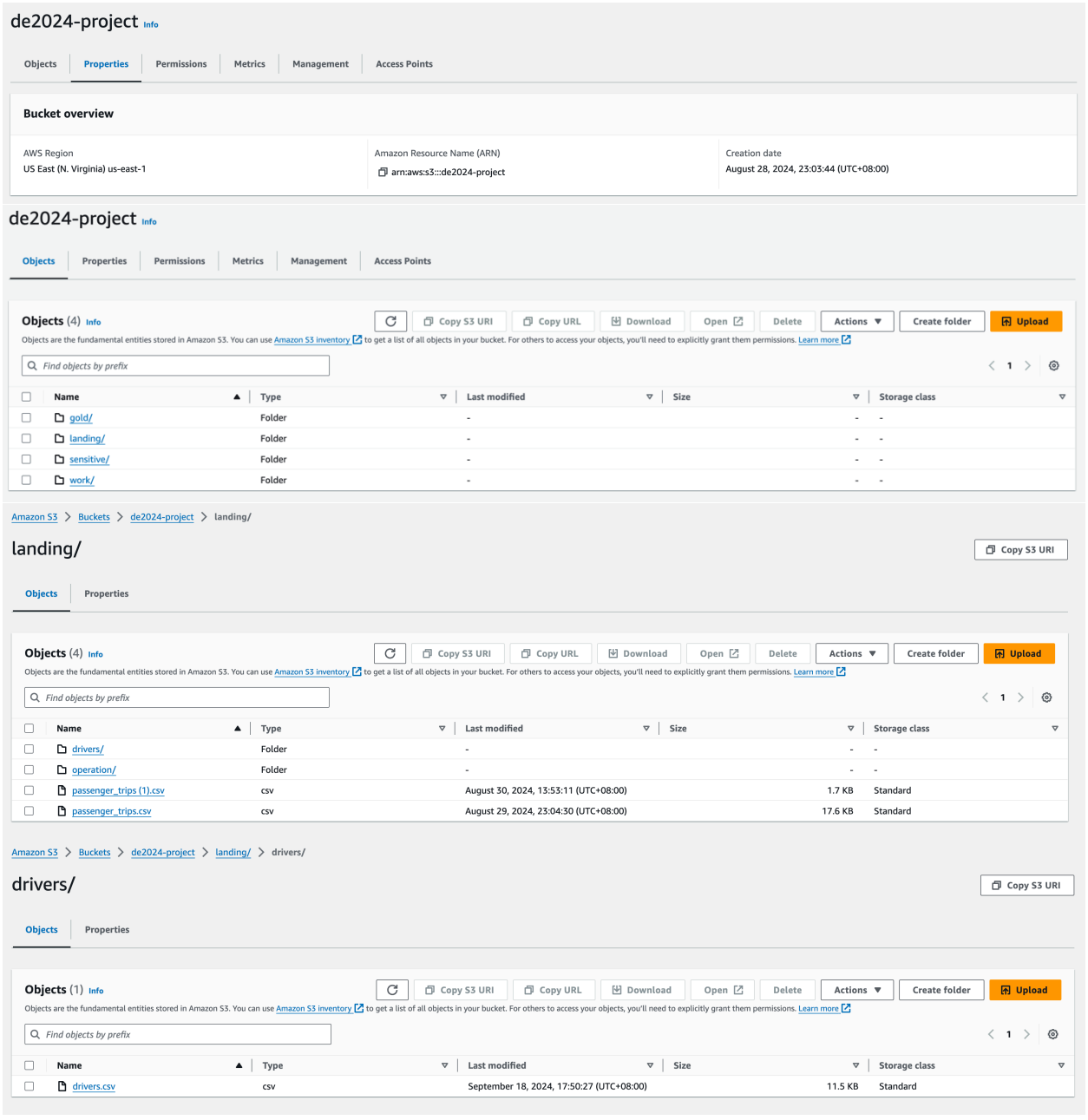


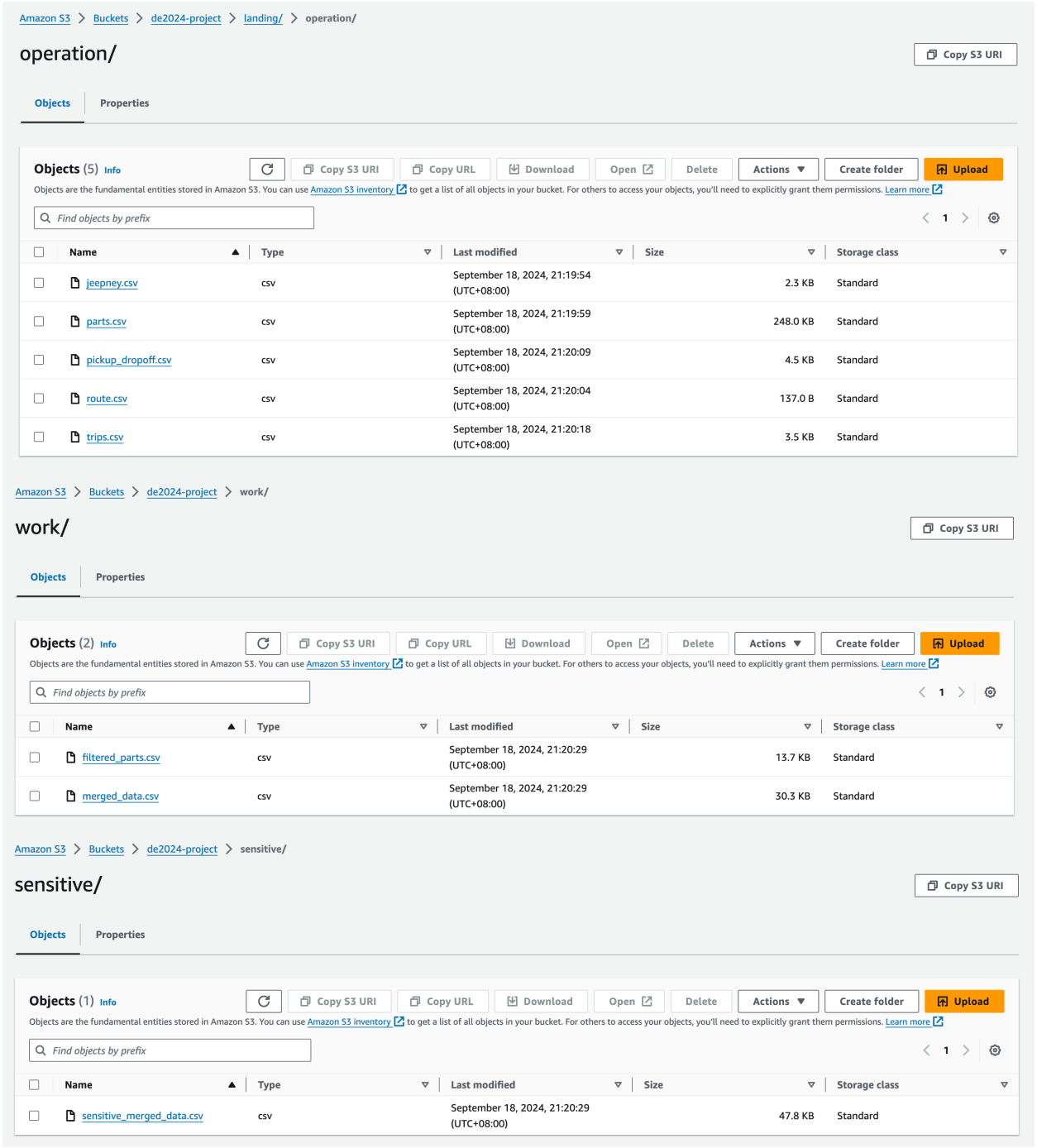
Appendix D-2. DynamoDB Data Sample

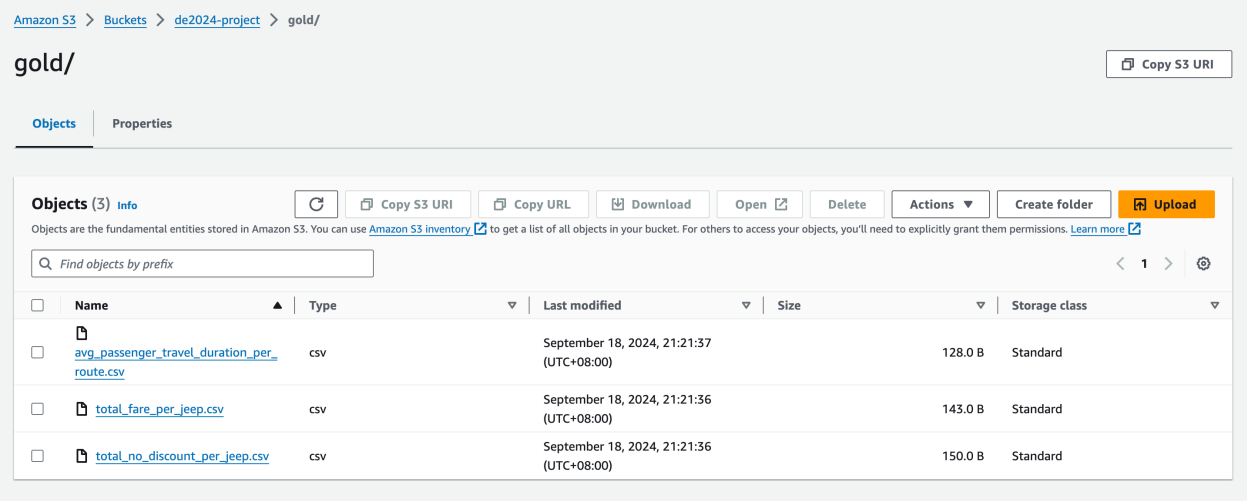
```
: import boto3

: dynamodb = boto3.resource('dynamodb')

: table = dynamodb.Table('drivers')

: table.scan()

: {'Items': [{'driver_id': Decimal('187'),
    'contact_number': '1191454341790',
    'license_number': 'LN7831919178484',
    'last_name': 'Buchanan',
    'birthday': '2137-08-12',
    'first_name': 'Michele'},
   {'driver_id': Decimal('154'),
    'contact_number': '7727782579852',
    'license_number': 'LN3280362692892',
    'last_name': 'Harris',
    'birthday': '2131-04-20',
    'first_name': 'Jamie'},
   {'driver_id': Decimal('7'),
    'contact_number': '8727237702448',
    'license_number': 'LN0013297118818',
    'last_name': 'Walker',
    'birthday': '2131-04-27',
    'first_name': 'Denise'},
   {'driver_id': Decimal('115'),
    'contact_number': '4416775962841',
    'license_number': 'LN7708421958824',
    'last_name': 'Stein',
    'birthday': '2136-05-26',
    'first_name': 'Larry'},
   {'driver_id': Decimal('117'),
    'contact_number': '3554234187550',
    'license_number': 'LN8749430407006',
    'last_name': 'Weber',
    'birthday': '2134-07-20',
    'first_name': 'Nicole'},
   {'driver_id': Decimal('47'),
    'contact_number': '2642012499246',
    'license_number': 'LN6658513723459',
    'last_name': 'Mccormick',
    'birthday': '2132-04-10',
    'first_name': 'Bradley'},
```

Appendix E. Data Lake

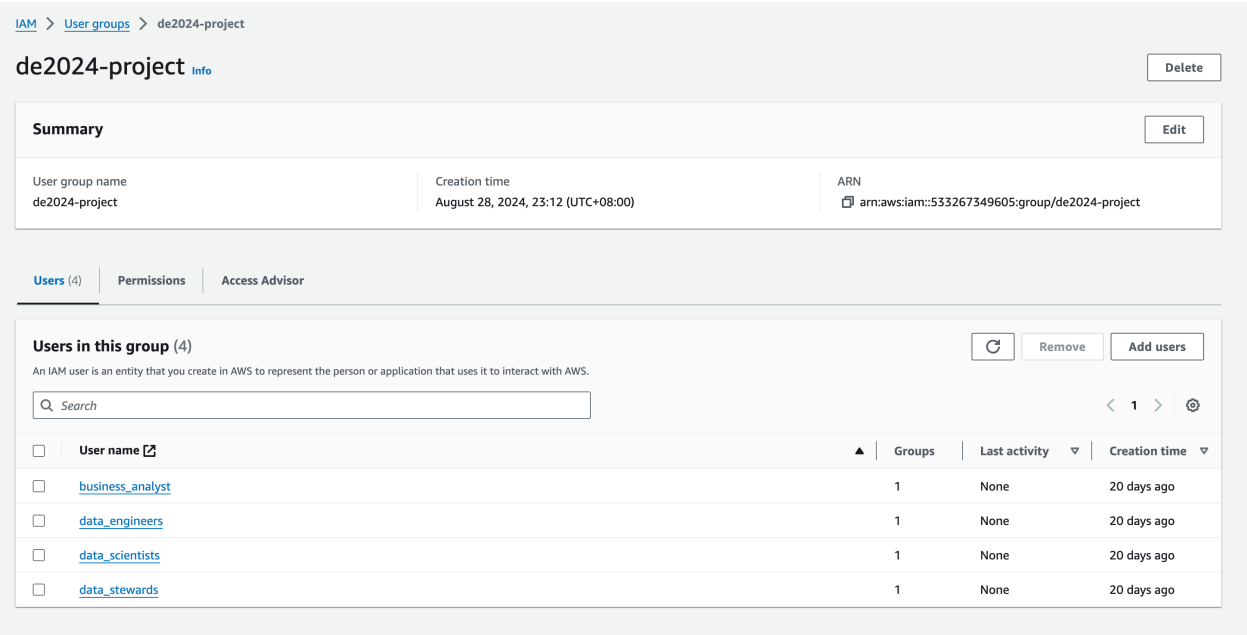
Appendix E-1. S3 Bucket and folders




Amazon S3 > Buckets > de2024-project > landing/ > operation/
operation/
Copy S3 URI
Objects
Properties
Objects (5) Info
Copy S3 URI
Copy URL
Download
Open
Delete
Actions
Create folder
Upload
Objects are the fundamental entities stored in Amazon S3. You can use Amazon S3 inventory to get a list of all objects in your bucket. For others to access your objects, you'll need to explicitly grant them permissions. Learn more
Find objects by prefix
1
Name
Type
Last modified
Size
Storage class
jeepney.csv
csv
September 18, 2024, 21:19:54 (UTC+08:00)
2.3 KB
Standard
parts.csv
csv
September 18, 2024, 21:19:59 (UTC+08:00)
248.0 KB
Standard
pickup_dropoff.csv
csv
September 18, 2024, 21:20:09 (UTC+08:00)
4.5 KB
Standard
route.csv
csv
September 18, 2024, 21:20:04 (UTC+08:00)
137.0 B
Standard
trips.csv
csv
September 18, 2024, 21:20:18 (UTC+08:00)
3.5 KB
Standard
Amazon S3 > Buckets > de2024-project > work/
work/
Copy S3 URI
Objects
Properties
Objects (2) Info
Copy S3 URI
Copy URL
Download
Open
Delete
Actions
Create folder
Upload
Objects are the fundamental entities stored in Amazon S3. You can use Amazon S3 inventory to get a list of all objects in your bucket. For others to access your objects, you'll need to explicitly grant them permissions. Learn more
Find objects by prefix
1
Name
Type
Last modified
Size
Storage class
filtered_parts.csv
csv
September 18, 2024, 21:20:29 (UTC+08:00)
13.7 KB
Standard
merged_data.csv
csv
September 18, 2024, 21:20:29 (UTC+08:00)
30.3 KB
Standard
Amazon S3 > Buckets > de2024-project > sensitive/
sensitive/
Copy S3 URI
Objects
Properties
Objects (1) Info
Copy S3 URI
Copy URL
Download
Open
Delete
Actions
Create folder
Upload
Objects are the fundamental entities stored in Amazon S3. You can use Amazon S3 inventory to get a list of all objects in your bucket. For others to access your objects, you'll need to explicitly grant them permissions. Learn more
Find objects by prefix
1
Name
Type
Last modified
Size
Storage class
sensitive_merged_data.csv
csv
September 18, 2024, 21:20:29 (UTC+08:00)
47.8 KB
Standard



Appendix E-2. IAM User Permissions



IAM > Users > business_analyst

business_analyst Info

Delete

Summary

ARN arn:aws:iam::533267349605:user/business_analyst	Console access Disabled	Access key 1 Create access key
Created August 28, 2024, 23:45 (UTC+08:00)	Last console sign-in -	

Permissions | Groups (1) | Tags | Security credentials | Access Advisor

Permissions policies (2)

Permissions are defined by policies attached to the user directly or through groups.

Remove | Add permissions

Search | Filter by Type: All types | 1

Policy name	Type	Attached via
AllowReadAccessToGold	Customer inline	Inline

IAM > Users > data_engineers

data_engineers Info

Delete

Summary

ARN arn:aws:iam::533267349605:user/data_engineers	Console access Disabled	Access key 1 AKIAXYKJWABS3WBUAZGW - Active Used today. 20 days old.
Created August 28, 2024, 23:38 (UTC+08:00)	Last console sign-in -	Access key 2 Create access key

Permissions | Groups (1) | Tags | Security credentials | Access Advisor

Permissions policies (2)

Permissions are defined by policies attached to the user directly or through groups.

Remove | Add permissions

Search | Filter by Type: All types | 1

Policy name	Type	Attached via
AmazonDynamoDBFullAccess	AWS managed	Group de2024-project
AmazonS3FullAccess	AWS managed	Directly

IAM > Users > data_scientists

data_scientists Info

Delete

Summary

ARN arn:aws:iam::533267349605:user/data_scientists	Console access Disabled	Access key 1 Create access key
Created August 28, 2024, 23:49 (UTC+08:00)	Last console sign-in -	

Permissions | Groups (1) | Tags | Security credentials | Access Advisor

Permissions policies (2)

Permissions are defined by policies attached to the user directly or through groups.

Remove | Add permissions

Search | Filter by Type: All types | 1

Policy name	Type	Attached via
AllowAccessToLandingAndGold	Customer inline	Inline

IAM > Users > data_stewards

data_stewards Info

Delete

Summary

ARN arn:aws:iam::533267349605:user/data_stewards	Console access Disabled	Access key 1 Create access key
Created August 28, 2024, 23:15 (UTC+08:00)	Last console sign-in -	

Permissions | Groups (1) | Tags | Security credentials | Access Advisor

Permissions policies (2)

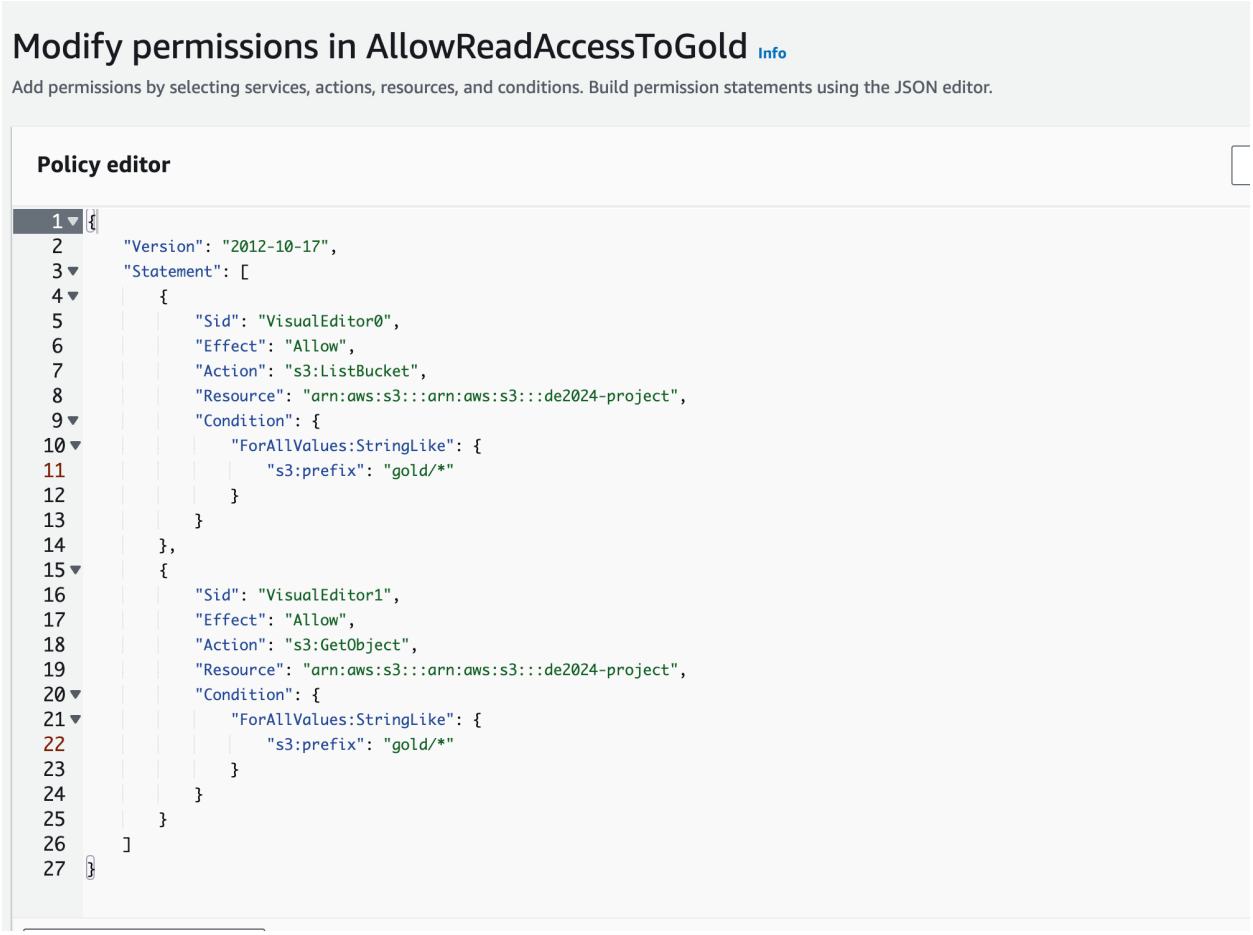
Permissions are defined by policies attached to the user directly or through groups.

Remove | Add permissions

Search | Filter by Type: All types | 1

Policy name	Type	Attached via
AllowAcessToSensitive	Customer inline	Inline

Appendix E-3. Permissions Json Code



Modify permissions in AllowReadAccessToGold Info

Add permissions by selecting services, actions, resources, and conditions. Build permission statements using the JSON editor.

Policy editor

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Sid": "VisualEditor0",
            "Effect": "Allow",
            "Action": "s3:ListBucket",
            "Resource": "arn:aws:s3:::arn:aws:s3:::de2024-project",
            "Condition": {
                "ForAllValues:StringLike": {
                    "s3:prefix": "gold/*"
                }
            }
        },
        {
            "Sid": "VisualEditor1",
            "Effect": "Allow",
            "Action": "s3:GetObject",
            "Resource": "arn:aws:s3:::arn:aws:s3:::de2024-project",
            "Condition": {
                "ForAllValues:StringLike": {
                    "s3:prefix": "gold/*"
                }
            }
        }
    ]
}
```

Modify permissions in AllowAccessToLandingAndGold Info

Add permissions by selecting services, actions, resources, and conditions. Build permission statements using the JSON editor.

Policy editor Visua

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Sid": "VisualEditor0",
            "Effect": "Allow",
            "Action": "s3:ListBucket",
            "Resource": "arn:aws:s3:::arn:aws:s3:::de2024-project",
            "Condition": {
                "ForAllValues:StringLike": {
                    "s3:prefix": [
                        "landing/*",
                        "gold/*"
                    ]
                }
            }
        },
        {
            "Sid": "VisualEditor1",
            "Effect": "Allow",
            "Action": "s3:GetObject",
            "Resource": "arn:aws:s3:::arn:aws:s3:::de2024-project",
            "Condition": {
                "ForAllValues:StringLike": {
                    "s3:prefix": [
                        "landing/*",
                        "gold/*"
                    ]
```

Modify permissions in AllowAccessToLandingAndGold Info

Add permissions by selecting services, actions, resources, and conditions. Build permission statements using the JSON editor.

Policy editor Visual

```
            "Effect": "Allow",
            "Action": "s3:GetObject",
            "Resource": "arn:aws:s3:::arn:aws:s3:::de2024-project",
            "Condition": {
                "ForAllValues:StringLike": {
                    "s3:prefix": [
                        "landing/*",
                        "gold/*"
                    ]
                }
            }
        },
        {
            "Sid": "VisualEditor2",
            "Effect": "Allow",
            "Action": "s3:PutObject",
            "Resource": "arn:aws:s3:::arn:aws:s3:::de2024-project",
            "Condition": {
                "ForAllValues:StringLike": {
                    "s3:prefix": [
                        "landing/*",
                        "gold/*"
                    ]
                }
            }
        }
    ]
}
```

Modify permissions in AllowAccessToLandingAndGold Info

Add permissions by selecting services, actions, resources, and conditions. Build permission statements using the JSON editor.

Policy editor Visual

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Sid": "VisualEditor0",
            "Effect": "Allow",
            "Action": "s3:ListBucket",
            "Resource": "arn:aws:s3:::arn:aws:s3:::de2024-project",
            "Condition": {
                "ForAllValues:StringLike": {
                    "s3:prefix": [
                        "landing/*",
                        "gold/*"
                    ]
                }
            }
        },
        {
            "Sid": "VisualEditor1",
            "Effect": "Allow",
            "Action": "s3:GetObject",
            "Resource": "arn:aws:s3:::arn:aws:s3:::de2024-project",
            "Condition": {
                "ForAllValues:StringLike": {
                    "s3:prefix": [
                        "landing/*",
                        "gold/*"
                    ]
```

Modify permissions in AllowAccessToLandingAndGold Info

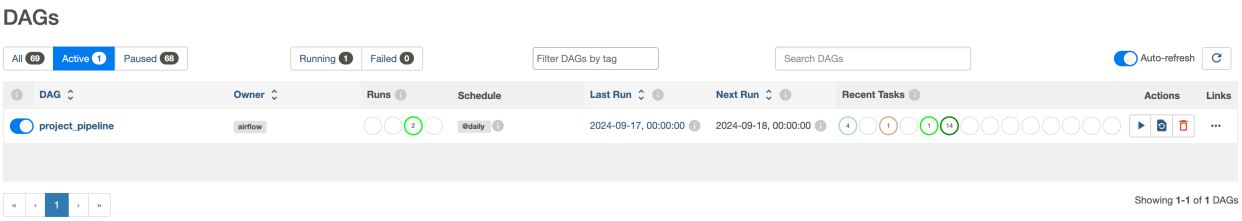
Add permissions by selecting services, actions, resources, and conditions. Build permission statements using the JSON editor.

Policy editor Visu

```
            "Effect": "Allow",
            "Action": "s3:GetObject",
            "Resource": "arn:aws:s3:::arn:aws:s3:::de2024-project",
            "Condition": {
                "ForAllValues:StringLike": {
                    "s3:prefix": [
                        "landing/*",
                        "gold/*"
                    ]
                }
            }
        },
        {
            "Sid": "VisualEditor2",
            "Effect": "Allow",
            "Action": "s3:PutObject",
            "Resource": "arn:aws:s3:::arn:aws:s3:::de2024-project",
            "Condition": {
                "ForAllValues:StringLike": {
                    "s3:prefix": [
                        "landing/*",
                        "gold/*"
                    ]
                }
            }
        }
    ]
}
```

Appendix F. Workflow Manager

Appendix F-1. Airflow DAGs



Appendix F-2. ETL Code

```
from datetime import datetime
from airflow.decorators import dag, task
from airflow.providers.amazon.aws.hooks.dynamodb import DynamoDBHook
from airflow.providers.postgres.hooks.postgres import PostgresHook
from airflow.providers.amazon.aws.hooks.s3 import S3Hook
from airflow.providers.amazon.aws.hooks.redshift_sql import RedshiftSQLHook
import pandas as pd
from io import StringIO

@dag(
    dag_id='project_pipeline',
    start_date=datetime(2024, 9, 16),
    schedule='@daily'
)
def etl_pipeline():

    @task
    def extract_postgres_table(table_name, output_path):
        hook = PostgresHook(postgres_conn_id="bgndb")
        sql = f"SELECT * FROM {table_name}"
        df = hook.get_pandas_df(sql)

        # Save to S3
        s3_hook = S3Hook(aws_conn_id='data_engineers')
        csv_buffer = StringIO()
        df.to_csv(csv_buffer, index=False)
        s3_hook.load_string(
            string_data=csv_buffer.getvalue(),
            key=output_path,
            bucket_name='de2024-project',
            replace=True
        )

    @task
    def extract_dynamodb_table(table_name, output_path):
        dynamodb = DynamoDBHook(aws_conn_id="data_engineers")
        table = dynamodb.get_conn().Table(table_name)
        response = table.scan()
        data = response['Items']
        df = pd.DataFrame(data)

        # Save to S3
        s3_hook = S3Hook(aws_conn_id='data_engineers')
        csv_buffer = StringIO()
        df.to_csv(csv_buffer, index=False)
        s3_hook.load_string(
            string_data=csv_buffer.getvalue(),
            key=output_path,
            bucket_name='de2024-project',
            replace=True
        )
```

```python
@task
def check_files_exist(file_paths):
    s3_hook = S3Hook(aws_conn_id='data_engineers')
    missing_files = [path for path in file_paths if not s3_hook.check_for_key(key=path, bucket_name='de2024-project')]
    if missing_files:
        raise FileNotFoundError(f"Missing files: {', '.join(missing_files)}")

@task
def merge_tables():
    s3_hook = S3Hook(aws_conn_id='data_engineers')
    passenger_trips = pd.read_csv(s3_hook.download_file('landing/passenger_trips.csv', 'de2024-project'))
    jeepney = pd.read_csv(s3_hook.download_file('landing/operation/jeepney.csv', 'de2024-project'))
    driver = pd.read_csv(s3_hook.download_file('landing/drivers/drivers.csv', 'de2024-project'))
    route = pd.read_csv(s3_hook.download_file('landing/operation/route.csv', 'de2024-project'))
    pudo = pd.read_csv(s3_hook.download_file('landing/operation/pickup_dropoff.csv', 'de2024-project'))
    parts = pd.read_csv(s3_hook.download_file('landing/operation/parts.csv', 'de2024-project'))

    merged_df = (passenger_trips.merge(jeepney, on='jeep_id', how='left')
                                .merge(driver, on='driver_id', how='left')
                                .merge(route, on='route_id', how='left')
                                .merge(pudo, on='pudo_id', how='left'))

    # Save merged data (with sensitive info)
    csv_buffer = StringIO()
    merged_df.to_csv(csv_buffer, index=False)
    s3_hook.load_string(
        string_data=csv_buffer.getvalue(),
        key='sensitive/sensitive_merged_data.csv',
        bucket_name='de2024-project',
        replace=True
    )

    sensitive_columns = ['first_name', 'last_name', 'birthday', 'contact_number', 'license_number']
    non_sensitive_df = merged_df.drop(columns=sensitive_columns)

    csv_buffer = StringIO()
    non_sensitive_df.to_csv(csv_buffer, index=False)
    s3_hook.load_string(
        string_data=csv_buffer.getvalue(),
        key='work/merged_data.csv',
        bucket_name='de2024-project',
        replace=True
    )

    jeep_ids = passenger_trips['jeep_id'].unique()
    filtered_parts = parts[parts['jeep_id'].isin(jeep_ids)]

    csv_buffer = StringIO()
    filtered_parts.to_csv(csv_buffer, index=False)
    s3_hook.load_string(
        string_data=csv_buffer.getvalue(),
```

```python
@task
def extract_and_load_to_redshift():
    s3_hook = S3Hook(aws_conn_id='data_engineers')
    redshift_hook = RedshiftSQLHook(redshift_conn_id='bgnrsdb')

    s3_key_merged = 'work/merged_data.csv'
    s3_key_filtered_parts = 'work/filtered_parts.csv'

    csv_data_merged = s3_hook.read_key(key=s3_key_merged, bucket_name='de2024-project')
    df_merged = pd.read_csv(StringIO(csv_data_merged))

    csv_data_filtered_parts = s3_hook.read_key(key=s3_key_filtered_parts, bucket_name='de2024-project')
    df_filtered_parts = pd.read_csv(StringIO(csv_data_filtered_parts))

    df_merged = df_merged.astype({
        'jeep_id': 'Int32',
        'model_year': 'Int32',
        'capacity': 'Int32',
        'route_id_x': 'Int32',
        'pudo_id': 'Int32',
        'driver_id': 'Int32',
        'passenger_travel_duration': 'Int32',
        'is_discount': 'boolean',
        'fare_x': 'float32',
    })

    df_filtered_parts = df_filtered_parts.astype({
        'jeep_part_id': 'Int32',
        'part_name': 'str',
        'installation_date': 'str',
        'life_expectancy': 'str',
        'cost': 'float32',
    })

    conn = redshift_hook.get_conn()
    cursor = conn.cursor()

    table_insert_sql = {
        'dim_pickup_dropoff': """
            INSERT INTO dim_pickup_dropoff (
                pudo_id, route_id, pickup, dropoff, fare
            ) VALUES (%s, %s, %s, %s, %s);
        """,
        'dim_jeepney_part': """
            INSERT INTO dim_jeepney_part (
                jeepney_part_id, part_name, installation_date, life_expectancy, cost
            ) VALUES (%s, %s, %s, %s, %s);
        """,
        'dim_jeepney': """
            INSERT INTO dim_jeepney (
                jeepney_id, model_year, capacity
            ) VALUES (%s, %s, %s);
```

```
table_insert_sql = {
    'dim_pickup_dropoff': """
        INSERT INTO dim_pickup_dropoff (
            pudo_id, route_id, pickup, dropoff, fare
        ) VALUES (%s, %s, %s, %s, %s);
    """,
    'dim_jeepney_part': """
        INSERT INTO dim_jeepney_part (
            jeepney_part_id, part_name, installation_date, life_expectancy, cost
        ) VALUES (%s, %s, %s, %s, %s);
    """,
    'dim_jeepney': """
        INSERT INTO dim_jeepney (
            jeepney_id, model_year, capacity
        ) VALUES (%s, %s, %s);
    """,
    'dim_route': """
        INSERT INTO dim_route (
            route_id, start_location, end_location
        ) VALUES (%s, %s, %s);
    """,
    'fact_trips': """
        INSERT INTO fact_trips (
            date, jeep_id, driver_id, route_id, pudo_id, card_number,
            passenger_travel_duration, is_discount, fare
        ) VALUES (%s, %s, %s, %s, %s, %s, %s, %s, %s);
    """

}

table_columns = {
    'dim_pickup_dropoff': ['pudo_id', 'route_id_y', 'pickup', 'dropoff', 'fare_y'],
    'dim_jeepney_part': ['jeep_part_id', 'part_name', 'installation_date', 'life_expectancy', 'cost'],
    'dim_jeepney': ['jeep_id', 'model_year', 'capacity'],
    'dim_route': ['route_id_x', 'start_location', 'end_location'],
    'fact_trips': ['date', 'jeep_id', 'driver_id', 'route_id_x', 'pudo_id', 'card_id',
                   'passenger_travel_duration', 'is_discount', 'fare_x']
}
for table, columns in table_columns.items():
    if table == 'dim_jeepney_part':
        df_table = df_filtered_parts[columns]
    else:
        df_table = df_merged[columns]

    data_tuples = [tuple(row) for row in df_table.to_numpy()]

    cursor.executemany(table_insert_sql[table], data_tuples)
    conn.commit()

cursor.close()
conn.close()
```

```python
@task
def drop_duplicates_from_dim_tables():
    redshift_hook = RedshiftSQLHook(redshift_conn_id='bgnrsdb')
    conn = redshift_hook.get_conn()
    cursor = conn.cursor()

    dedup_sql = {
        'dim_pickup_dropoff': """
            DELETE FROM dim_pickup_dropoff
            WHERE (pudo_id, route_id, pickup, dropoff, fare) NOT IN (
                SELECT pudo_id, route_id, pickup, dropoff, fare
                FROM (
                    SELECT pudo_id, route_id, pickup, dropoff, fare,
                           ROW_NUMBER() OVER (PARTITION BY pudo_id, route_id, pickup, dropoff, fare ORDER BY pudo_id) AS row_num
                    FROM dim_pickup_dropoff
                ) tmp
                WHERE row_num = 1
            );
        """,
        'dim_jeepney_part': """
            DELETE FROM dim_jeepney_part
            WHERE (jeepney_part_id, part_name, installation_date, life_expectancy, cost) NOT IN (
                SELECT jeepney_part_id, part_name, installation_date, life_expectancy, cost
                FROM (
                    SELECT jeepney_part_id, part_name, installation_date, life_expectancy, cost,
                           ROW_NUMBER() OVER (PARTITION BY jeepney_part_id, part_name, installation_date, life_expectancy, cost ORDER BY jeepney_part_id) AS row_num
                    FROM dim_jeepney_part
                ) tmp
                WHERE row_num = 1
            );
        """,
        'dim_jeepney': """
            DELETE FROM dim_jeepney
            WHERE (jeepney_id, model_year, capacity) NOT IN (
                SELECT jeepney_id, model_year, capacity
                FROM (
                    SELECT jeepney_id, model_year, capacity,
                           ROW_NUMBER() OVER (PARTITION BY jeepney_id, model_year, capacity ORDER BY jeepney_id) AS row_num
                    FROM dim_jeepney
                ) tmp
                WHERE row_num = 1
            );
        """,
        'dim_route': """
            DELETE FROM dim_route
            WHERE (route_id, start_location, end_location) NOT IN (
                SELECT route_id, start_location, end_location
                FROM (
                    SELECT route_id, start_location, end_location,
                           ROW_NUMBER() OVER (PARTITION BY route_id, start_location, end_location ORDER BY route_id) AS row_num
                    FROM dim_route
```

```
                    SELECT route_id, start_location, end_location,
                           ROW_NUMBER() OVER (PARTITION BY route_id, start_location, end_location ORDER BY route_id) AS row_num
                    FROM dim_route
                ) tmp
                WHERE row_num = 1
            );
        """
    }

    for table, sql in dedup_sql.items():
        try:
            cursor.execute(sql)
            conn.commit()
        except Exception as e:
            conn.rollback()
            raise e

    cursor.close()
    conn.close()

# Define task dependencies
extract_jeepney = extract_postgres_table('jeepney', 'landing/operation/jeepney.csv')
extract_parts = extract_postgres_table('parts', 'landing/operation/parts.csv')
extract_route = extract_postgres_table('route', 'landing/operation/route.csv')
extract_pudo = extract_postgres_table('pickup_dropoff', 'landing/operation/pickup_dropoff.csv')
extract_drivers = extract_dynamodb_table('drivers', 'landing/drivers/drivers.csv')
extract_trips = extract_postgres_table('trips', 'landing/operation/trips.csv')

check_files = check_files_exist([
    'landing/operation/jeepney.csv',
    'landing/operation/parts.csv',
    'landing/operation/route.csv',
    'landing/operation/pickup_dropoff.csv',
    'landing/operation/trips.csv',
    'landing/passenger_trips.csv',
    'landing/drivers/drivers.csv'
])

merge = merge_tables()
load_to_redshift = extract_and_load_to_redshift()
drop_duplicates = drop_duplicates_from_dim_tables()

# Set task dependencies
[extract_jeepney, extract_parts, extract_route, extract_pudo, extract_drivers, extract_trips] >> check_files

check_files >> merge >> load_to_redshift >> drop_duplicates

etl_pipeline()
```