Project Title: Real-time Taxi Telemetry Streaming

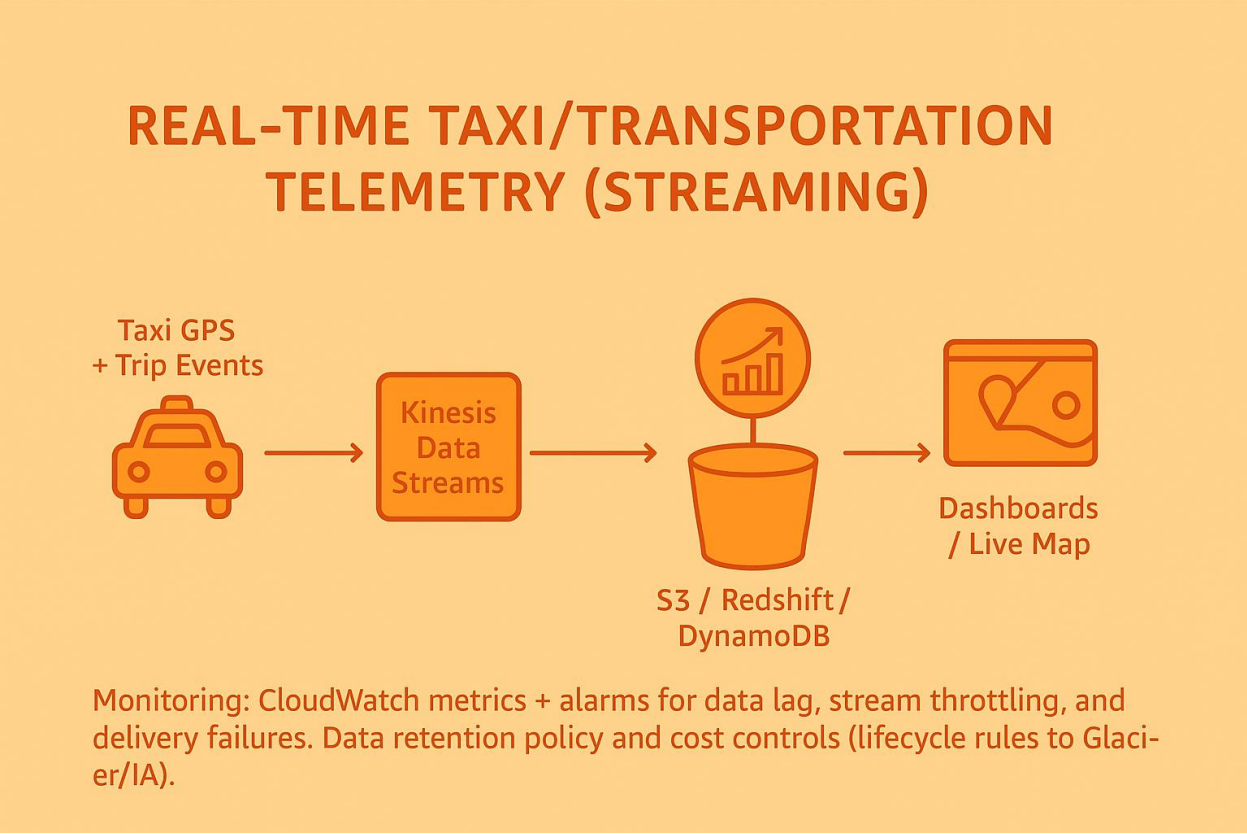
Name: Yashwanth Aravanti

Batch: AWS Data Engineering (2025)

Scenario: An end-to-end pipeline to ingest daily smart electricity meter readings, perform nightly batch ETL processing to aggregate consumption data and detect anomalies, and serve the results for billing insights and analytics. An API provides on-demand access to meter data, and alerts are sent for unusual consumption patterns.

Pipeline Type: Streaming Data Pipeline.

Tools/Services: S3, Kinesis DataStream, Kinesis Data Firehouse , AWS Lambda, Amazon QuickSight, SQS, Redshift, DynamoDB, Amazon Location Service.



Project Goal:

The project aims to build an automated, serverless streaming data pipeline that ingests real-time taxi trip data, enriches it with location and billing information, and serves it through two primary endpoints: a low-latency operational database for micro-billing and a data warehouse for interactive BI dashboards.

Project Overview

Data Ingestion:

A Python script (producer.py) simulates a fleet of taxis, sending real-time trip data as JSON events to an Amazon Kinesis Data Stream. This stream acts as a durable and scalable entry point for all incoming data. A parallel Kinesis Data Firehose stream is also subscribed to the data stream to archive all raw, unaltered events into a raw data lake in an S3 bucket.

Real-time Processing & Enrichment:

An AWS Lambda function (EnrichTripDataFunction) is triggered for each batch of records arriving in the Kinesis Data Stream. It performs real-time data enrichment by calling AWS Location Service to convert the trip's GPS coordinates into a human-readable zone name. It then asynchronously invokes a second Lambda function, passing along the newly enriched data.

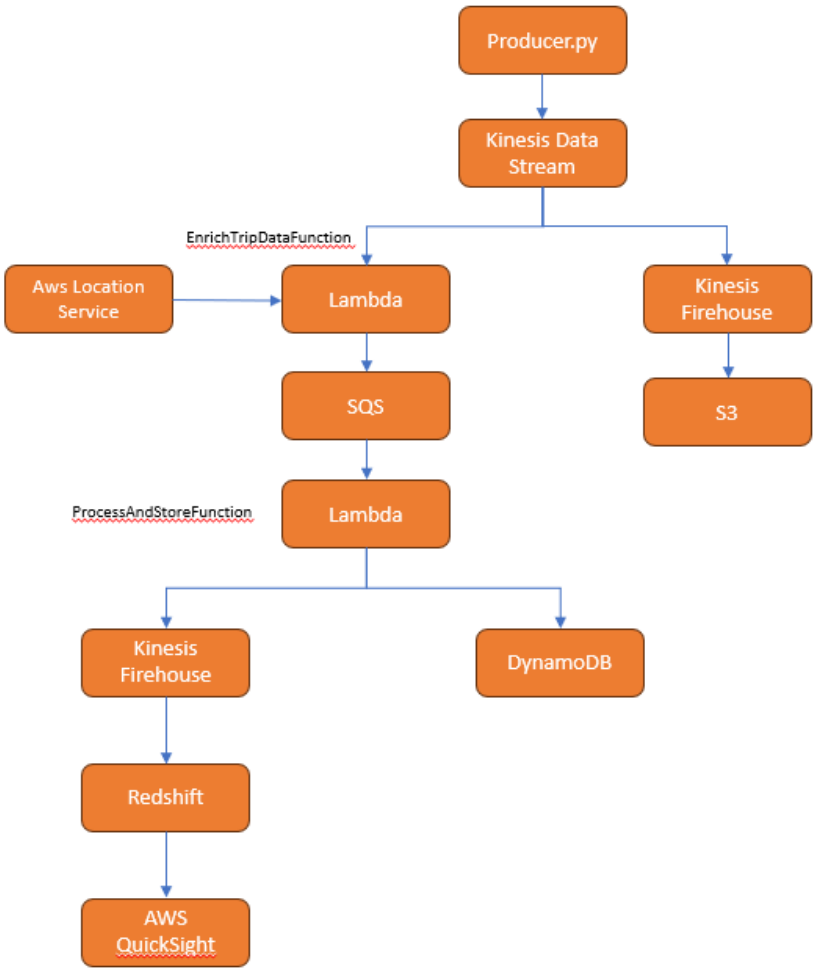
Interactive Analytics & Dashboards:

The second Lambda (ProcessAndStoreFunction) sends the final, fully processed data (including zone and fare information) to a dedicated Kinesis Data Firehose stream. This Firehose stream automatically batches, compresses, and loads the data into an Amazon Redshift data warehouse. Amazon QuickSight then connects to Redshift, providing rich, interactive dashboards for business users to analyze trip trends, fare profitability, and popular zones.

Real-time Data Access

To provide low-latency reads for micro-billing, the ProcessAndStoreFunction writes the final, complete trip record—including the detailed billing breakdown (base fare, distance charge, etc.)—to an Amazon DynamoDB table. This provides a fast, key-value datastore for applications to programmatically access the billing details of any individual trip in single-digit milliseconds.

Architectural Diagram:



Data Flow Explained

The pipeline processes data in a series of logical steps, from ingestion to visualization:

1. **Real-time Data Ingestion**

   A Python script, Producer.py, simulates a fleet of taxis sending trip data. Each event is pushed into an Amazon Kinesis Data Stream, which acts as the scalable and durable "front door" for all incoming raw data.

2. **Raw Data Archiving (Data Lake)**

   A Kinesis Firehose delivery stream is subscribed directly to the Kinesis Data Stream. Its sole purpose is to automatically batch all raw, incoming events and archive them into an S3 bucket. This creates a cost-effective, long-term backup and a raw data lake for future use.

3. **Live Enrichment & Processing**

   An AWS Lambda function, EnrichTripDataFunction, is triggered by the Kinesis Data Stream. It performs real-time enrichment by calling AWS Location Service to convert each trip's GPS coordinates into a human-readable zone name. It then invokes a second Lambda function to continue the processing.

4. **Billing Calculation & Data Fan-Out**

   The second Lambda function, ProcessAndStoreFunction, receives the enriched data. It calculates the fare and micro-billing details for the trip. It then "fans out" the final, processed record to two separate destinations: an operational database and an analytical warehouse.

5. **Operational Datastore**

   For fast, real-time lookups, the ProcessAndStoreFunction writes the complete trip record, including the detailed billing breakdown, to an Amazon DynamoDB table. This serves as a low-latency datastore for applications needing to access the details of a single trip quickly.

6. **Analytics Warehouse & BI**

   Simultaneously, the ProcessAndStoreFunction sends the processed record to a second Kinesis Firehose stream. This Firehose batches the data and efficiently loads it into Amazon Redshift. From there, AWS QuickSight connects to Redshift to power interactive dashboards for business intelligence, allowing for analysis of trends, fare profitability, and popular routes.

Key Architectural Benefits:

- **Serverless & Scalable:** The architecture primarily uses managed services, eliminating the need to manage servers and allowing each component to scale independently based on data volume.
- **Decoupled:** SQS and Kinesis acts as a buffer, decoupling the data producers from the consumers, which increases the system's resilience.
- **Separation of Concerns:** The pipeline effectively separates the operational path (low-latency writes and reads via DynamoDB) from the analytical path (high-throughput batch loads into Redshift for complex queries).

Real-Time Taxi Telemetry Dataset — Overview

This dataset represents real-time telemetry and trip-level data for a fleet of taxis operating across multiple cities. Each record corresponds to a unique taxi trip event, capturing essential attributes for analyzing movement patterns, trip efficiency, and geographical trends.

Dataset Description

Column Name	Type	Description
trip_id	String (Partition Key)	Unique identifier for each trip
taxi_id	String	Unique vehicle identifier within the fleet
pickup_datetime	String	ISO 8601 timestamp capturing the pickup time
latitude	Number	Current GPS coordinates - latitude during trip event
longitude	Number	Current GPS coordinates - longitude during trip event

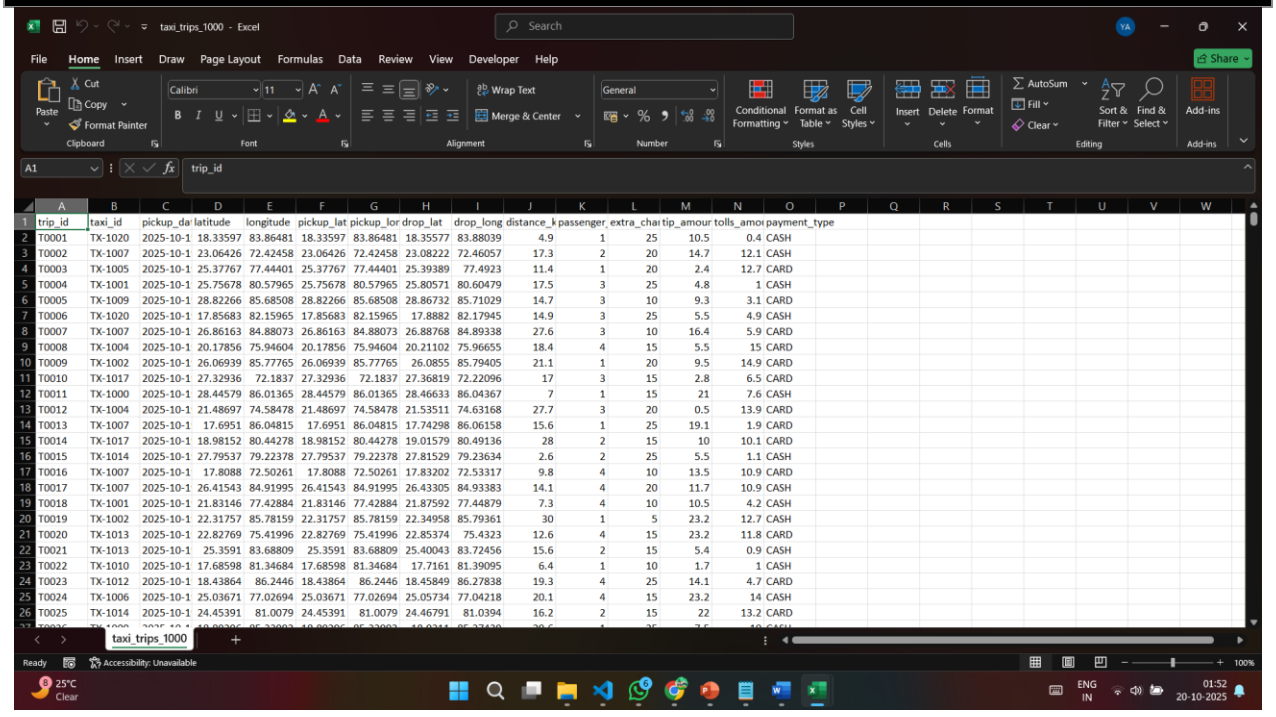
Column Name	Type	Description
pickup_lat	Number	Latitude of the pickup location
pickup_long	Number	Longitude of the pickup location
drop_lat	Number	Latitude of the drop-off location
drop_long	Number	Longitude of the drop-off location
distance_km	Number	Total trip distance in kilometers between pickup and drop points
passenger_count	Number	Number of passengers in the taxi
extra_charges	Number	Additional charges (night surcharge, luggage, etc.)
tip_amount	Number	Tip amount given by the passenger
tolls_amount	Number	Toll charges incurred during the trip
payment_type	String	Payment method used (CASH/CARD)
zone_name	String	Zone name from AWS Location Service reverse geocoding
fare_amount	Number	Calculated fare amount based on distance and zones
total_amount	Number	Total amount (fare + extra_charges + tip_amount + tolls_amount)

Key Highlights

- Simulates live GPS telemetry at 1 event per minute per taxi.
- Covers multiple Indian metro cities (e.g., Delhi, Pune, Kolkata, Ahmedabad, Bangalore).
- Ideal for AWS data pipeline demonstrations, IoT analytics, and QuickSight dashboards.

Sample Dataset:

```
trip_id,taxi_id,pickup_datetime,latitude,longitude,pickup_lat,pickup_long,drop_lat,drop_long,distanc
e_km,passenger_count,extra_charges,tip_amount,tolls_amount,payment_type
T0001,TX-1014,2025-10-
19T14:01:00,28.551823,77.174423,28.551823,77.174423,28.559356,77.183819,17.6,3,25.0,15.0,10.0,CARD
T0002,TX-1001,2025-10-
19T14:02:00,28.596394,77.169191,28.596394,77.169191,28.597551,77.147886,4.3,1,10.0,5.0,0.0,CASH
T0003,TX-1016,2025-10-
19T14:03:00,18.45584,73.855462,18.45584,73.855462,18.473326,73.878359,23.3,2,15.0,20.0,15.0,CARD
T0004,TX-1002,2025-10-
19T14:04:00,22.626806,88.351184,22.626806,88.351184,22.667979,88.329066,23.4,4,20.0,25.0,5.0,CARD
T0005,TX-1009,2025-10-
19T14:05:00,22.610451,88.368129,22.610451,88.368129,22.626527,88.414473,23.7,1,10.0,10.0,0.0,CASH
T0006,TX-1006,2025-10-
19T14:06:00,18.542128,73.89092,18.542128,73.89092,18.50109,73.925077,8.3,2,15.0,8.0,12.0,CARD
T0007,TX-1006,2025-10-
19T14:07:00,17.39741,78.498347,17.39741,78.498347,17.395826,78.524843,7,1,5.0,5.0,0.0,CASH
T0008,TX-1020,2025-10-
19T14:08:00,23.008976,72.563344,23.008976,72.563344,23.00332,72.516269,10.3,3,20.0,12.0,8.0,CARD
```



Step-by-Step Implementation:

Pipeline 1: Real-time Processing and Analytics Pipeline

Step 1: Create the Data Producer (producer.py)

Purpose:

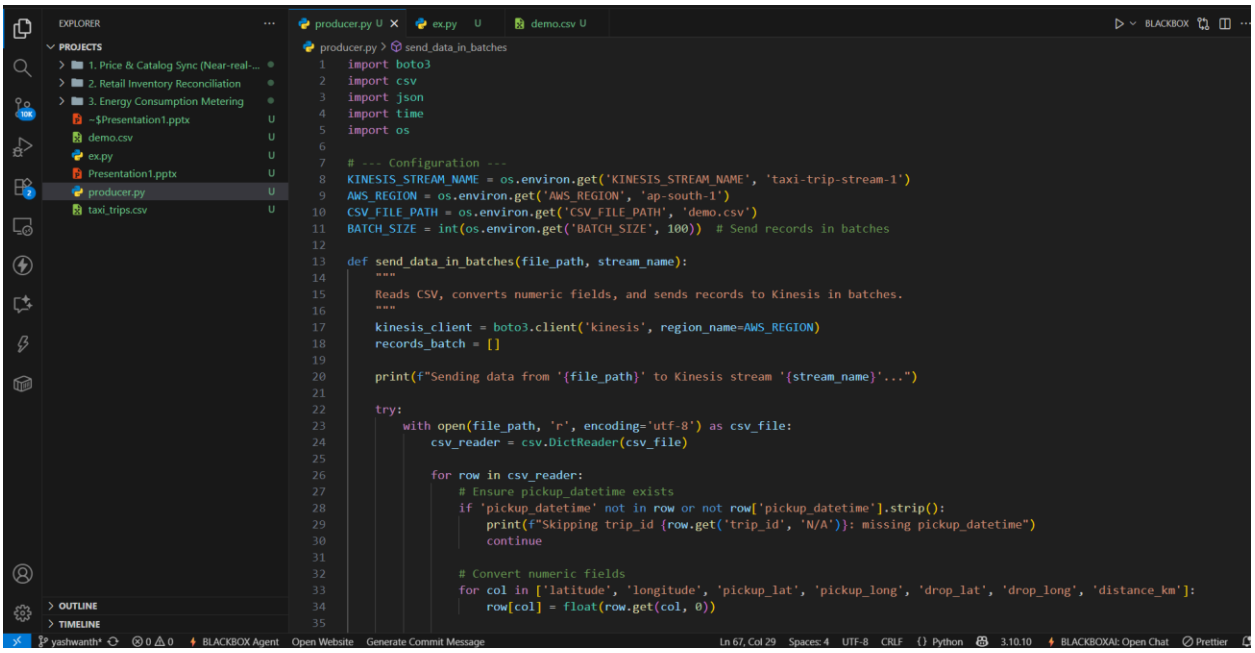
This script simulates a taxi sending real-time trip telemetry data (like trip ID, pickup/drop-off time, GPS coordinates, fare, etc.) to a Kinesis Data Stream.

Steps:

1. Create a Python file named producer.py.
2. Use the **boto3** SDK to connect to Kinesis and send JSON trip events.
3. Example template:

```
import boto3
import csv
import json
import time
import os

# --- Configuration ---
KINESIS_STREAM_NAME = os.environ.get('KINESIS_STREAM_NAME', 'taxi-trip-stream-1')
AWS_REGION = os.environ.get('AWS_REGION', 'ap-south-1')
CSV_FILE_PATH = os.environ.get('CSV_FILE_PATH', 'demo.csv')
BATCH_SIZE = int(os.environ.get('BATCH_SIZE', 100))  # Send records in batches

def send_data_in_batches(file_path, stream_name):
    """
    Reads CSV, converts numeric fields, and sends records to Kinesis in batches.
    """
    kinesis_client = boto3.client('kinesis', region_name=AWS_REGION)
    records_batch = []

    print(f"Sending data from '{file_path}' to Kinesis stream '{stream_name}'...")

    try:
        with open(file_path, 'r', encoding='utf-8') as csv_file:
            csv_reader = csv.DictReader(csv_file)

            for row in csv_reader:
                # Ensure pickup_datetime exists
                if 'pickup_datetime' not in row or not row['pickup_datetime'].strip():
                    print(f"Skipping trip_id {row.get('trip_id', 'N/A')}: missing pickup_datetime")
                    continue

                # Convert numeric fields
                for col in ['latitude', 'longitude', 'pickup_lat', 'pickup_long', 'drop_lat',
'drop_long', 'distance_km']:
                    row[col] = float(row.get(col, 0))

                # Optional fields
                row['passenger_count'] = int(row.get('passenger_count', 0))
                row['fare_amount'] = float(row.get('fare_amount', 0))
                row['extra_charges'] = float(row.get('extra_charges', row.get('extra', 0)))
```

```
                row['tip_amount'] = float(row.get('tip_amount', 0))
                row['tolls_amount'] = float(row.get('tolls_amount', 0))
                row['total_amount'] = float(row.get('total_amount', 0))
                row['zone_name'] = row.get('zone_name', '').strip('"')

                # Prepare Kinesis record
                record = {
                    'Data': json.dumps(row).encode('utf-8'),
                    'PartitionKey': row['trip_id']
                }
                records_batch.append(record)

                # Send batch if full
                if len(records_batch) >= BATCH_SIZE:
                    _send_batch(kinesis_client, stream_name, records_batch)
                    records_batch = []
                    time.sleep(0.1)  # optional throttle

            # Send remaining records
            if records_batch:
                _send_batch(kinesis_client, stream_name, records_batch)

    except FileNotFoundError:
        print(f"Error: File '{file_path}' not found.")
    except Exception as e:
        print(f"Unexpected error: {e}")

    print("Finished sending all data.")

def _send_batch(client, stream_name, batch):
    """Send a batch of records to Kinesis and handle errors."""
    try:
        response = client.put_records(StreamName=stream_name, Records=batch)
        print(f"Sent batch of {len(batch)} records.")
        if response.get('FailedRecordCount', 0) > 0:
            print(f"WARNING: {response['FailedRecordCount']} records failed.")
    except Exception as e:
        print(f"Error sending batch: {e}")


if __name__ == "__main__":
    send_data_in_batches(CSV_FILE_PATH, KINESIS_STREAM_NAME)
```



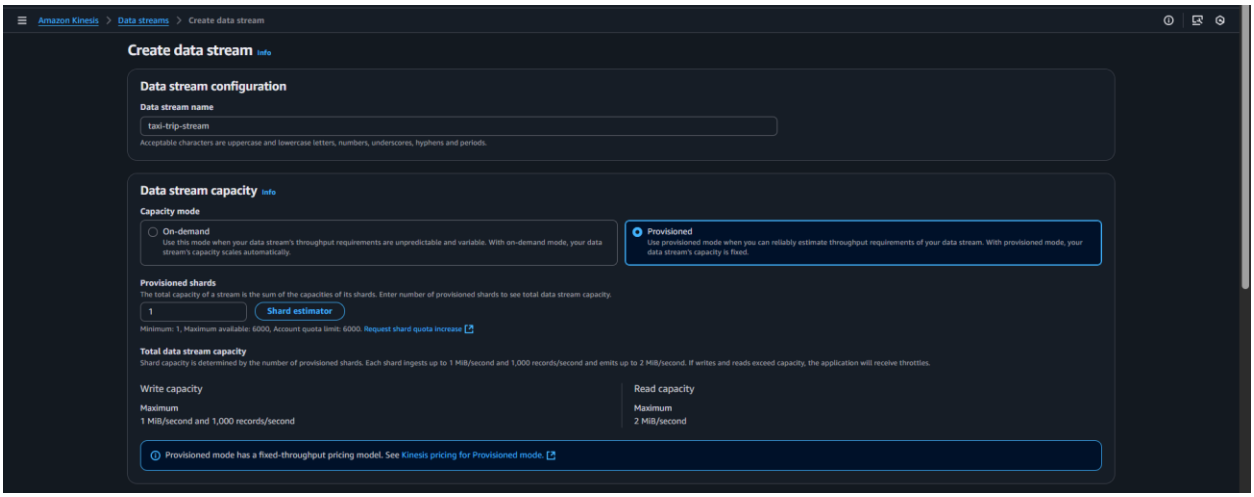
Step 2: Create Kinesis Data Stream (Real-Time Ingestion Layer)

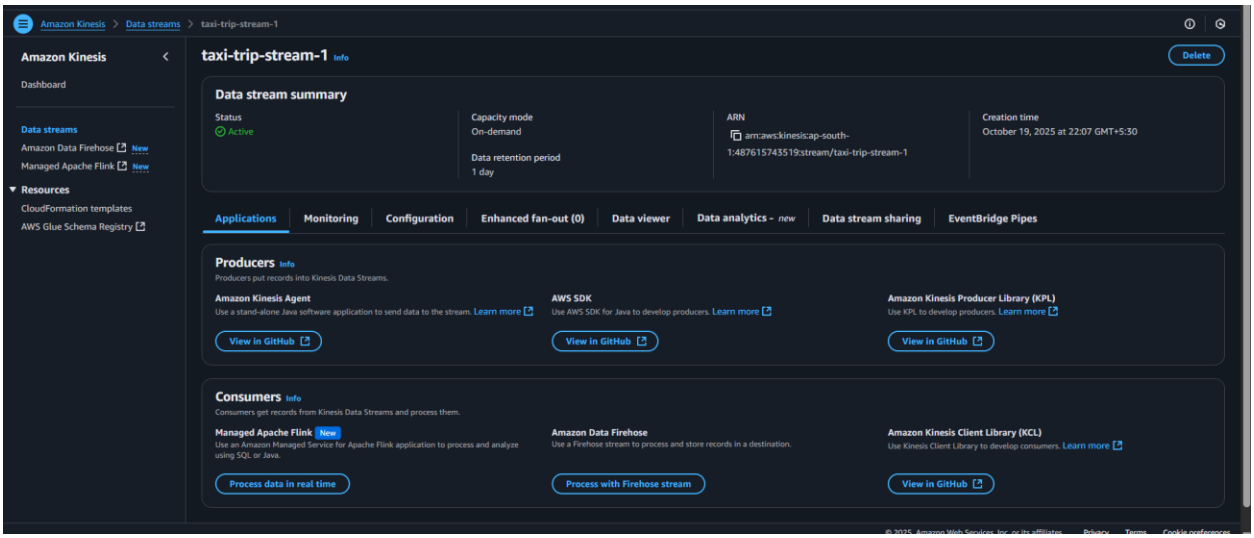
Steps:

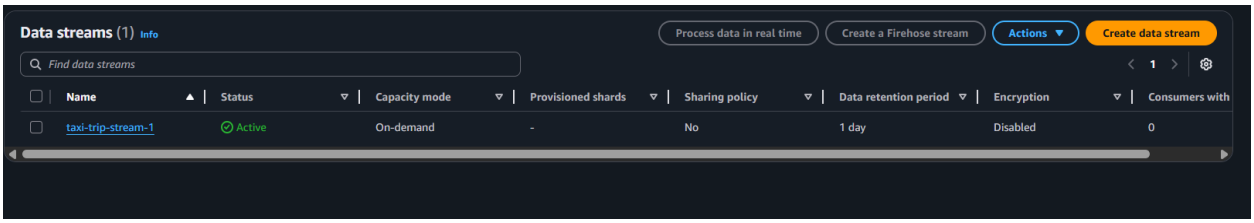
1. Go to AWS Console → Kinesis → Data Streams → Create data stream.
2. Stream name: taxi-trip-stream
3. Capacity mode: Provisioned
4. Number of shards: 1
5. Click Create data stream.

Purpose:

This Kinesis Data Stream is the real-time ingestion point for all incoming taxi trip events. It decouples data producers and consumers for scalable real-time processing.







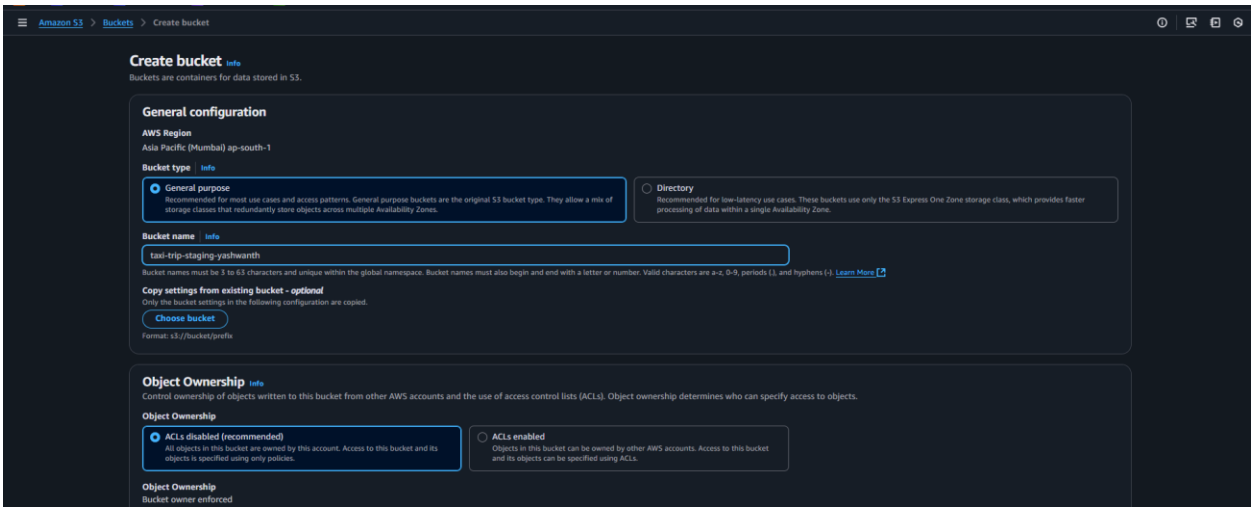
Step 3: Create an S3 Staging Bucket for Firehose

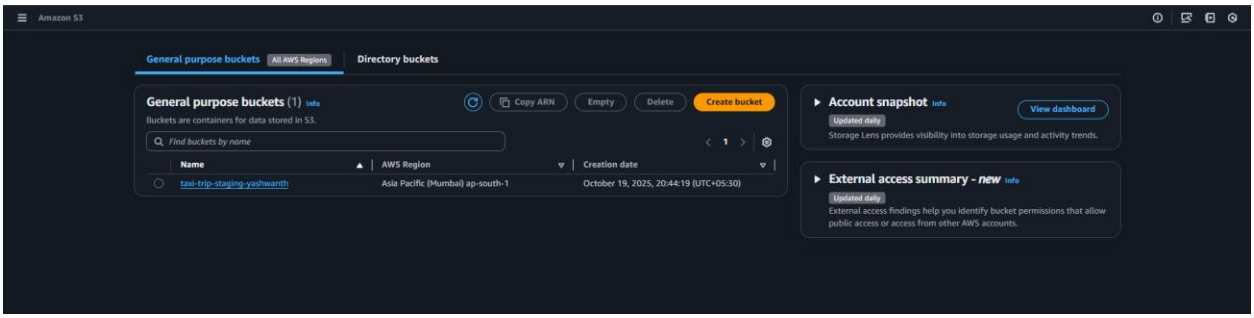
Steps:

1. Go to S3 → Create bucket
2. Bucket name: taxi-trip-staging-yashwanth
3. Region: ap-south-1
4. Click Create bucket.

Purpose:

This bucket acts as temporary staging storage for raw data delivered from Kinesis Firehose before it is loaded into Redshift.





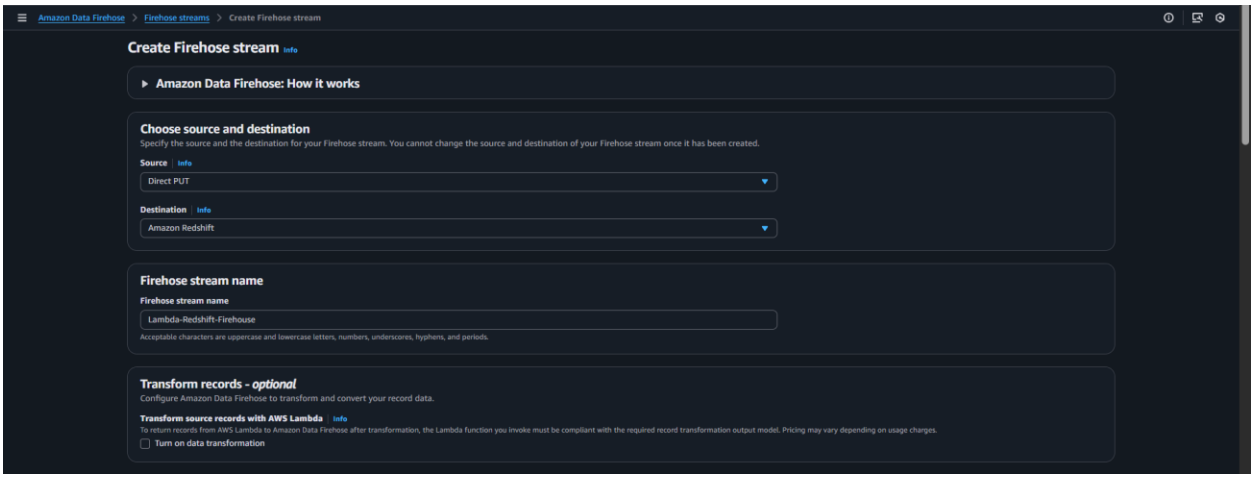
Step 4: Create Amazon Kinesis Firehose Delivery Stream

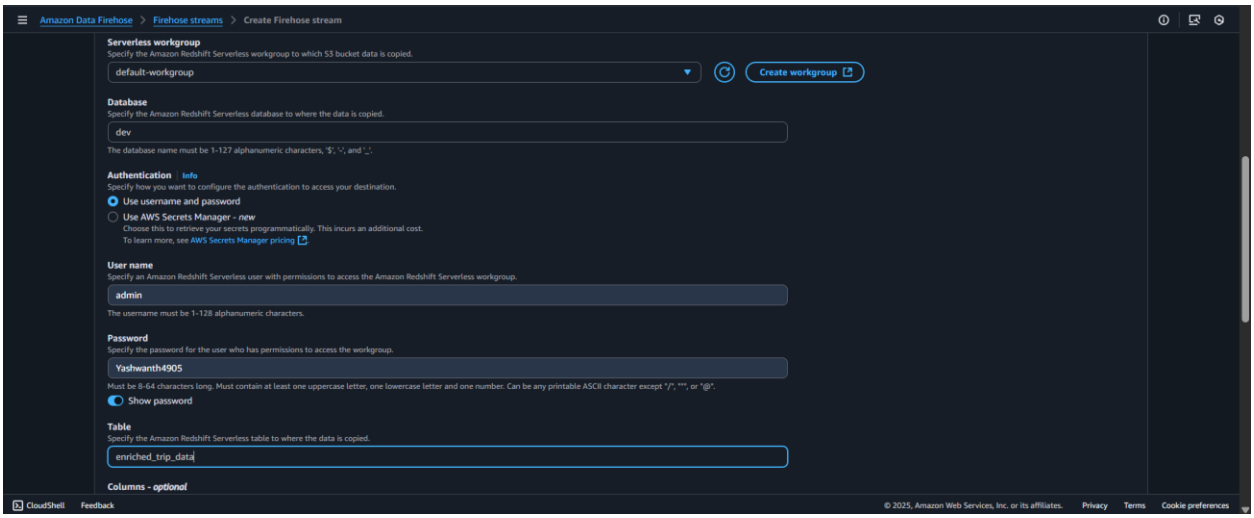
Steps:

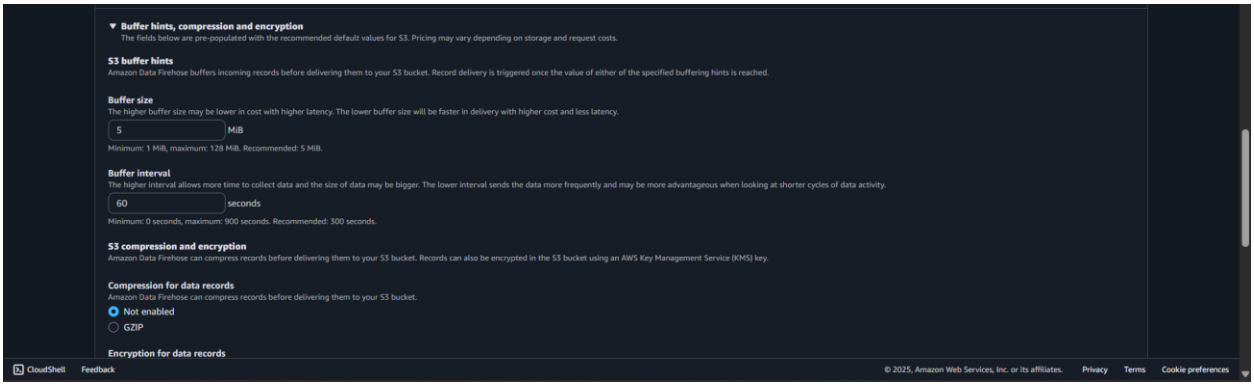
1. Go to **Kinesis → Firehose Delivery Stream → Create delivery stream**.
2. Source: **Direct PUT**
3. Destination: **Redshift** (choose **taxi-trip-staging-yashwanth**).
4. Enter Username and password
5. Firehose stream name: Lambda-Redshift-Firehouse
6. Create the stream with defaults.

Purpose:

Firehose continuously triggers from lambda and delivers the data into Redshift for analytical processing.







COPY enriched_trip_data FROM 's3://taxi-trip-staging-yashwanth/<manifest>' CREDENTIALS 'aws_iam_role=arn:aws:iam::<aws-account-id>:role/<role-name>' MANIFEST FORMAT AS JSON 'auto' TIMEFORMAT 'YYYY-MM-DDTHH:MI:SS';

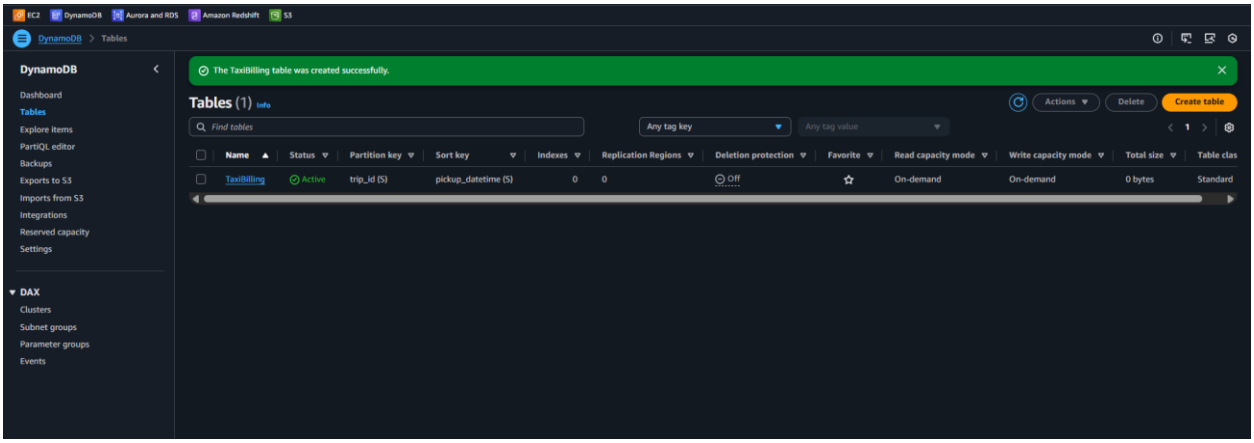
Step 5 : Create Amazon DynamoDB Table (Operational Data Store)

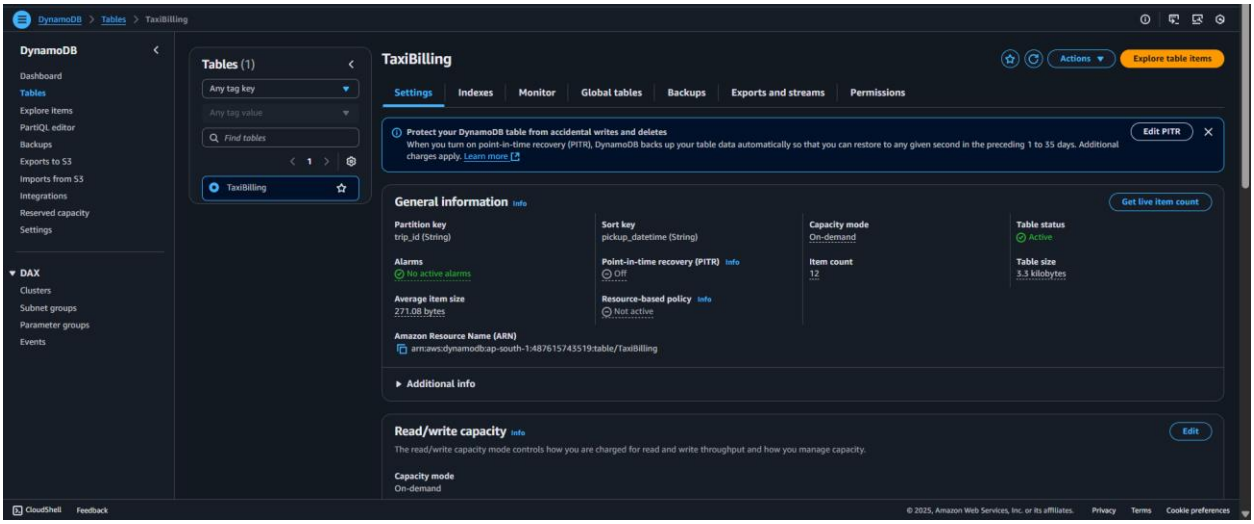
Steps:

1. Go to DynamoDB → Create table.
2. Table name: TaxiBilling
3. Partition key: trip_id (String)
4. Sort key: pickup_datetime (String)
5. Click Create table.

Purpose:

Stores processed and enriched trip data for low-latency lookup and micro-billing applications.





Step 6: Create Amazon Location Service Place Index

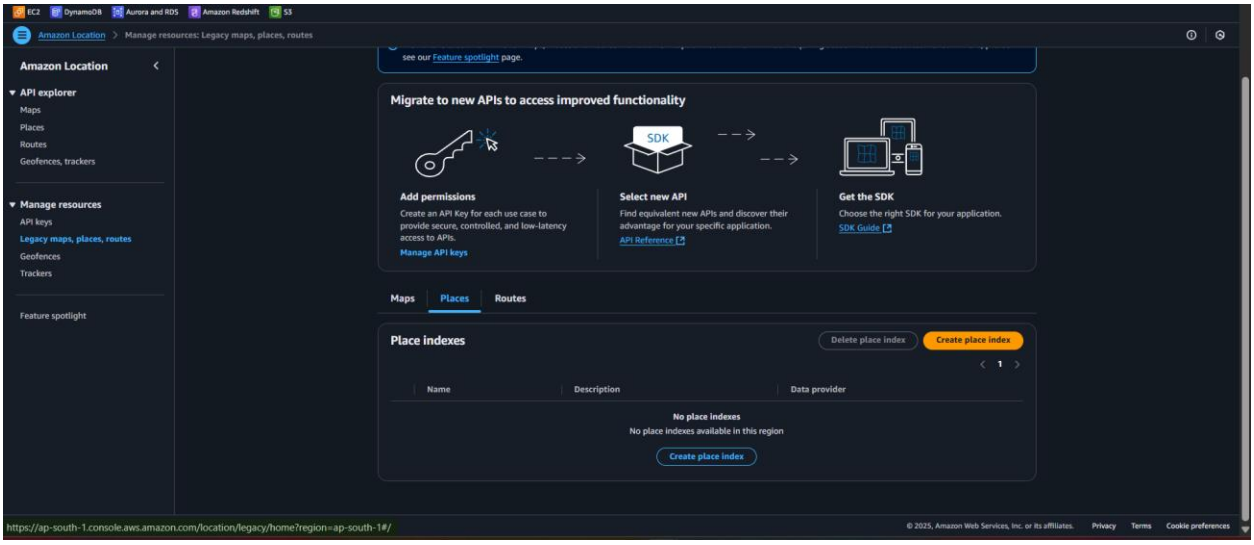
Steps:

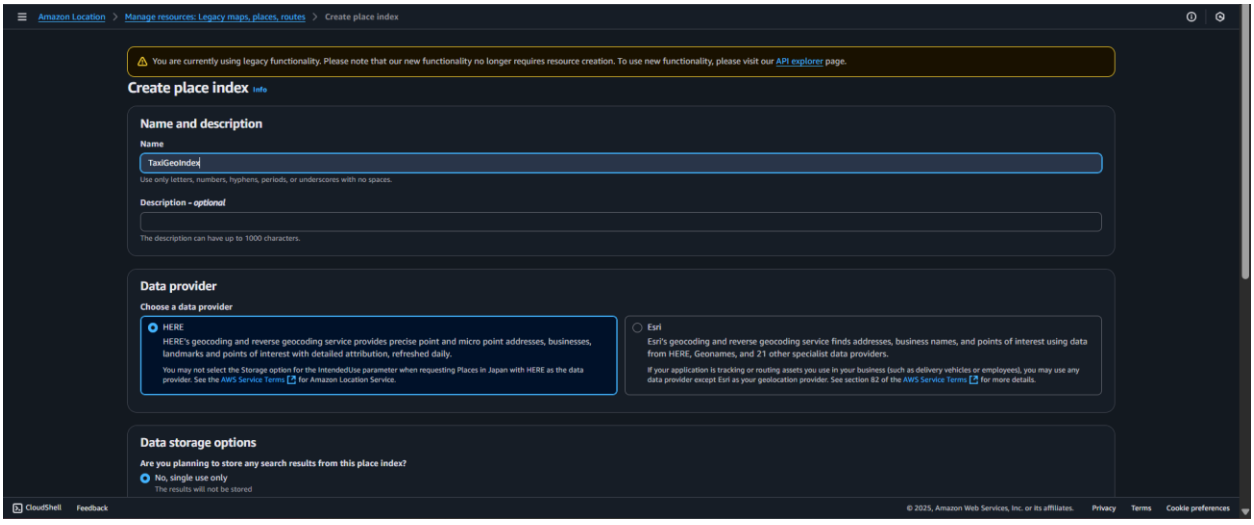
1. Go to Amazon Location Service → Place indexes → Create place index.
2. Name: TaxiGeoIndex
3. Data provider: Esri
4. Click Create place index.

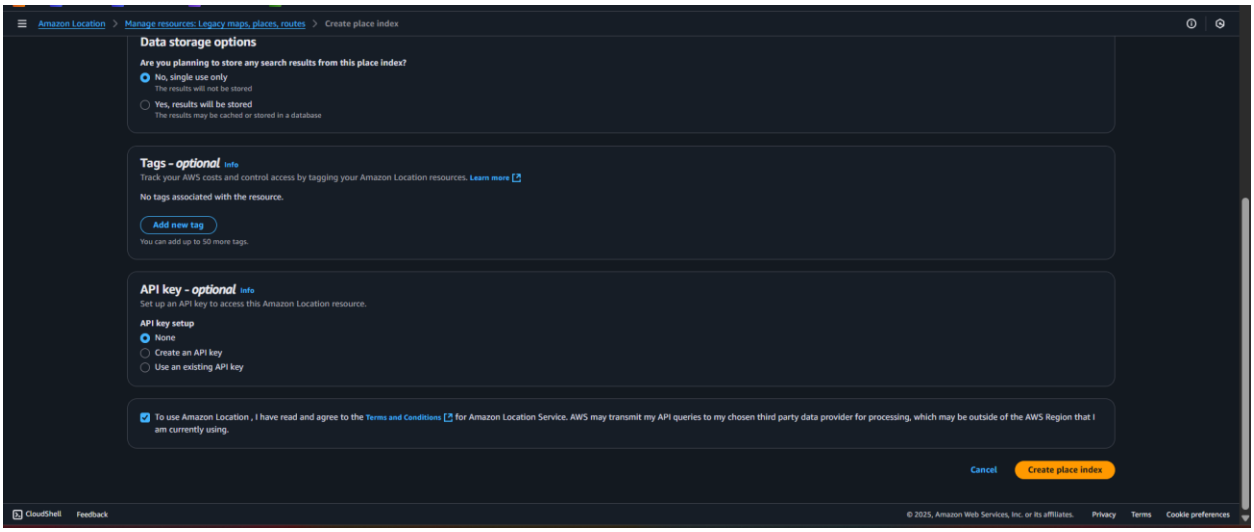
Purpose:

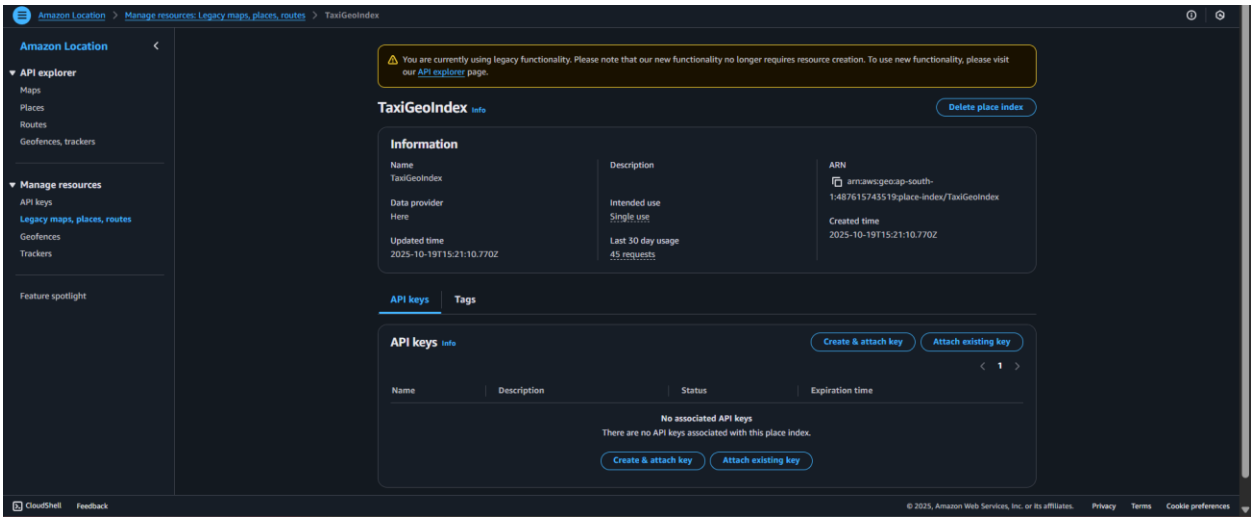
Used for reverse geocoding coordinates into readable zone or area names to enrich trip data during processing.

ARN: arn:aws:geo:ap-south-1:487615743519:place-index/TaxiGeoIndex









Step 8: Create SQS

The Problem: Risks of Direct Lambda Invocation

Connecting two Lambda functions directly using an asynchronous invoke call is functional but not resilient for production workloads. It introduces several key risks:

- **Data Loss:** If the second Lambda fails, AWS retries the invocation only twice. If it fails a third time, the data (the event payload) is permanently discarded.
- **No Visibility into Failures:** There is no built-in way to easily see which events failed or to re-process them. Once lost, they are gone for good.
- **Tight Coupling:** The first Lambda function is directly aware of and dependent on the second. This makes the system brittle and harder to maintain or modify in the future.

The Solution: Decoupling with Amazon SQS

The best-practice solution is to place a durable service between the two functions. **Amazon SQS (Simple Queue Service)** acts as a reliable buffer, ensuring that messages are stored safely until they can be successfully processed.

This creates a robust, decoupled architecture:

Kinesis Stream → Enrichment Lambda → SQS Queue → Processing Lambda → DynamoDB/Firehose

This design guarantees that even if the processing Lambda is down or fails repeatedly, the enriched data remains safe in the SQS queue, ready to be retried.

How to Create an SQS Queue with a DLQ

A **Dead-Letter Queue (DLQ)** is a critical component that captures messages that have failed processing multiple times, allowing you to debug issues without losing data.

Step 1: Navigate to SQS & Create Queue

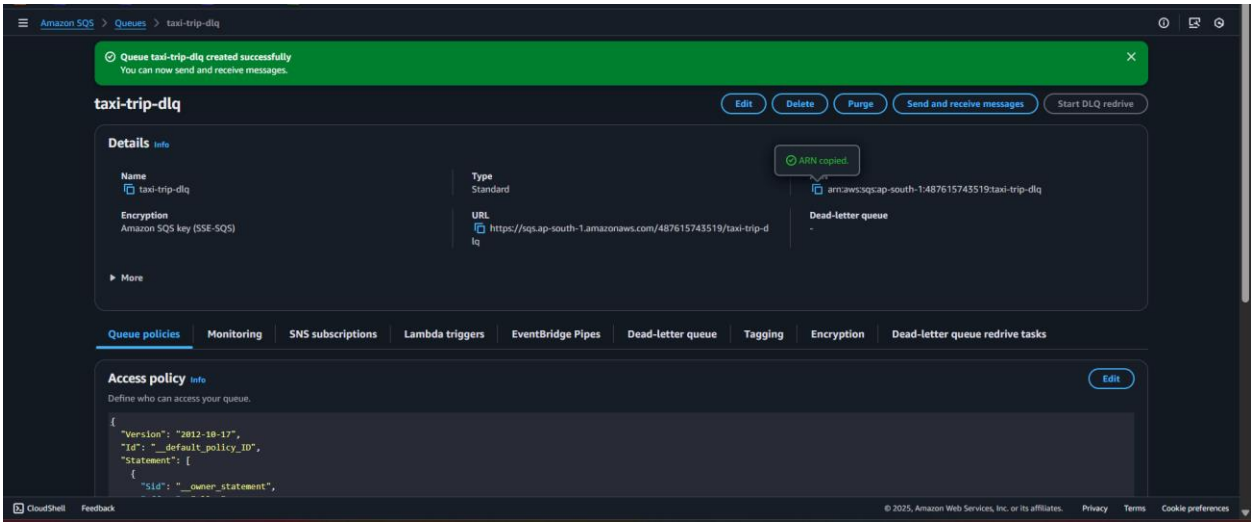
1. In the AWS Console, search for and navigate to **SQS (Simple Queue Service)**.
2. Click the **Create queue** button.
3. Keep the **Type** as **Standard** and give your queue a descriptive **Name** (e.g., taxi-trip-processing-queue).

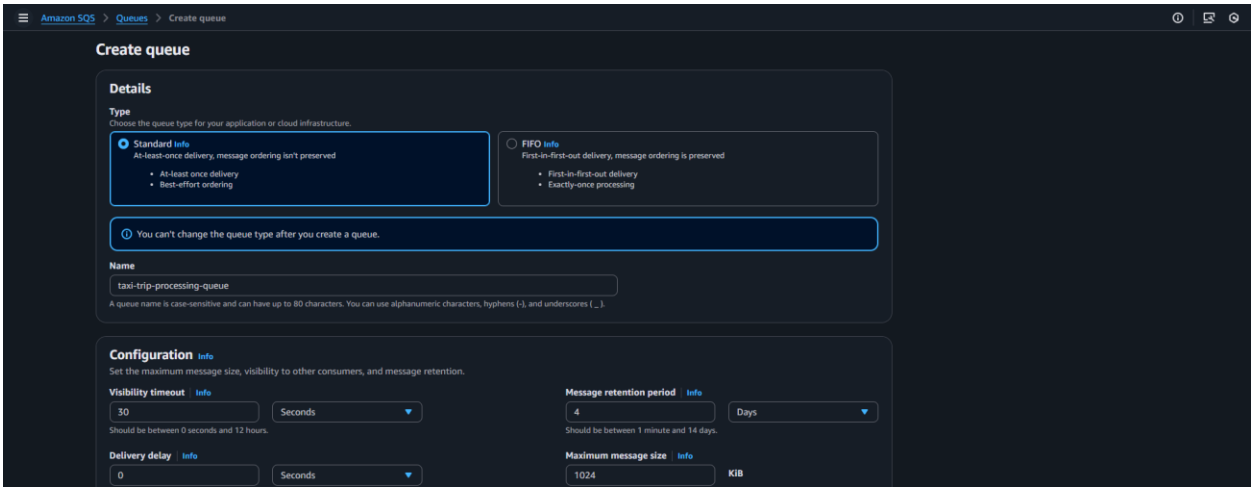
Step 2: Configure the Dead-Letter Queue (DLQ)

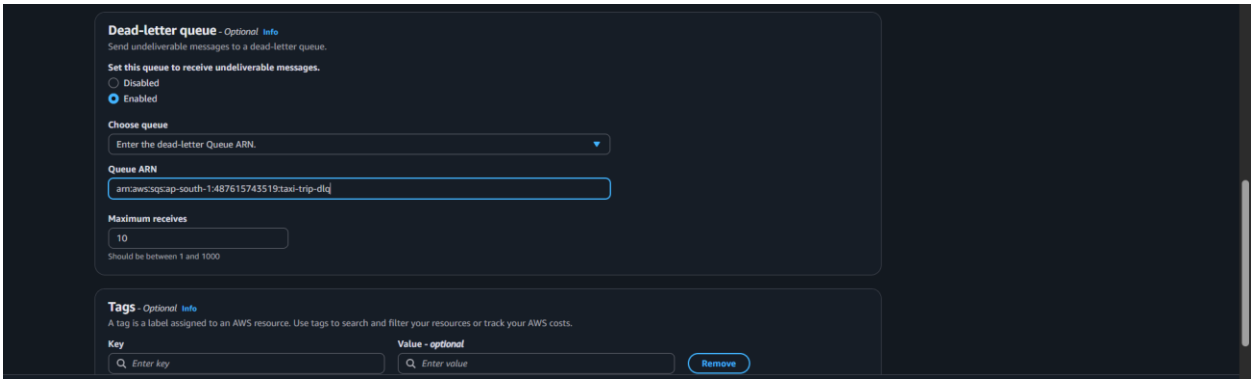
1. Scroll down to the **Dead-letter queue (DLQ)** section and click **Enable**.
2. An "Enter ARN for SQS queue" option will appear. Select **Create new queue**.
3. In the pop-up, name your DLQ (e.g., taxi-trip-dlq) and click **Create**.
4. Set **Maximum receives**. This is the number of processing attempts before a message is sent to the DLQ. A value of **3** or **5** is a good starting point.

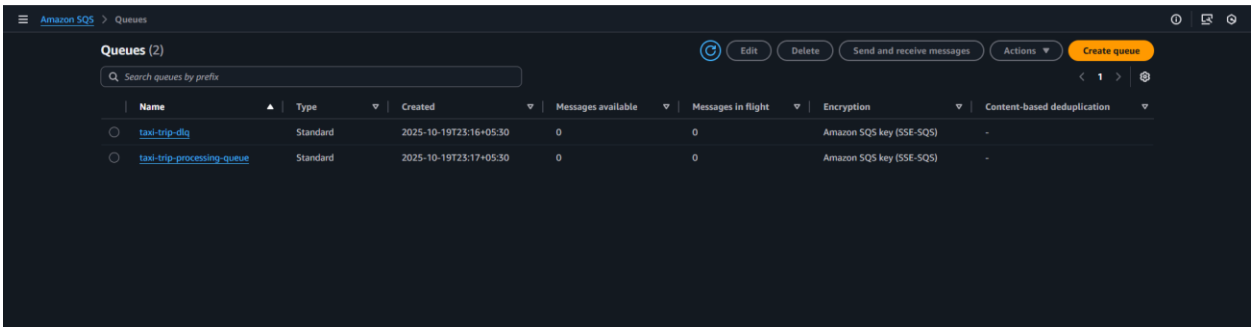
Step 3: Create the Queue & Copy the URL

1. Scroll to the bottom and click **Create queue**.
2. From the SQS dashboard, click on your newly created main queue (taxi-trip-processing-queue).
3. On the details page, **copy the Queue URL**. You will need this URL to set as an environment variable in your enrichment Lambda function.


Amazon SQS > Queues > taxi-trip-dlq
Queue taxi-trip-dlq created successfully
You can now send and receive messages.
taxi-trip-dlq
Edit
Delete
Purge
Send and receive messages
Start DLQ redrive
Details Info
Name
taxi-trip-dlq
Type
Standard
ARN copied.
arn:aws:sqs:ap-south-1:487615743519:taxi-trip-dlq
Encryption
Amazon SQS key (SSE-SQS)
URL
https://sqs.ap-south-1.amazonaws.com/487615743519/taxi-trip-dlq
Dead-letter queue
More
Queue policies
Monitoring
SNS subscriptions
Lambda triggers
EventBridge Pipes
Dead-letter queue
Tagging
Encryption
Dead-letter queue redrive tasks
Access policy Info
Edit
Define who can access your queue.
{
"Version": "2012-10-17",
"Id": "__default_policy_ID",
"Statement": [
{
"Sid": "__owner_statement",
CloudShell
Feedback
© 2025, Amazon Web Services, Inc. or its affiliates.
Privacy
Terms
Cookie preferences


Amazon SQS > Queues > Create queue
Create queue
Details
Type
Choose the queue type for your application or cloud infrastructure.
Standard Info
At-least-once delivery, message ordering isn't preserved
At-least once delivery
Best-effort ordering
FIFO Info
First-in-first-out delivery, message ordering is preserved
First-in-first-out delivery
Exactly-once processing
You can't change the queue type after you create a queue.
Name
taxi-trip-processing-queue
A queue name is case-sensitive and can have up to 80 characters. You can use alphanumeric characters, hyphens (-), and underscores (_).
Configuration Info
Set the maximum message size, visibility to other consumers, and message retention.
Visibility timeout Info
30
Seconds
Should be between 0 seconds and 12 hours.
Message retention period Info
4
Days
Should be between 1 minute and 14 days.
Delivery delay Info
0
Seconds
Maximum message size Info
1024
KiB


Dead-letter queue - Optional Info
Send undeliverable messages to a dead-letter queue.
Set this queue to receive undeliverable messages.
Disabled
Enabled
Choose queue
Enter the dead-letter Queue ARN.
Queue ARN
arn:aws:sqs:ap-south-1:487615743519:taxi-trip-dlq
Maximum receives
10
Should be between 1 and 1000
Tags - Optional Info
A tag is a label assigned to an AWS resource. Use tags to search and filter your resources or track your AWS costs.
Key
Enter key
Value - optional
Enter value
Remove



Step 8: Create AWS Lambda Function 1 (Data Enrichment Lambda)

Function Name: EnrichTripDataFunction

Purpose:

This Lambda function is triggered by a Kinesis Data Stream.
It decodes each taxi trip record, enriches it using AWS Location Service (reverse geocoding), and then asynchronously invokes the next Lambda function (ProcessAndStoreFunction) for further processing and storage.

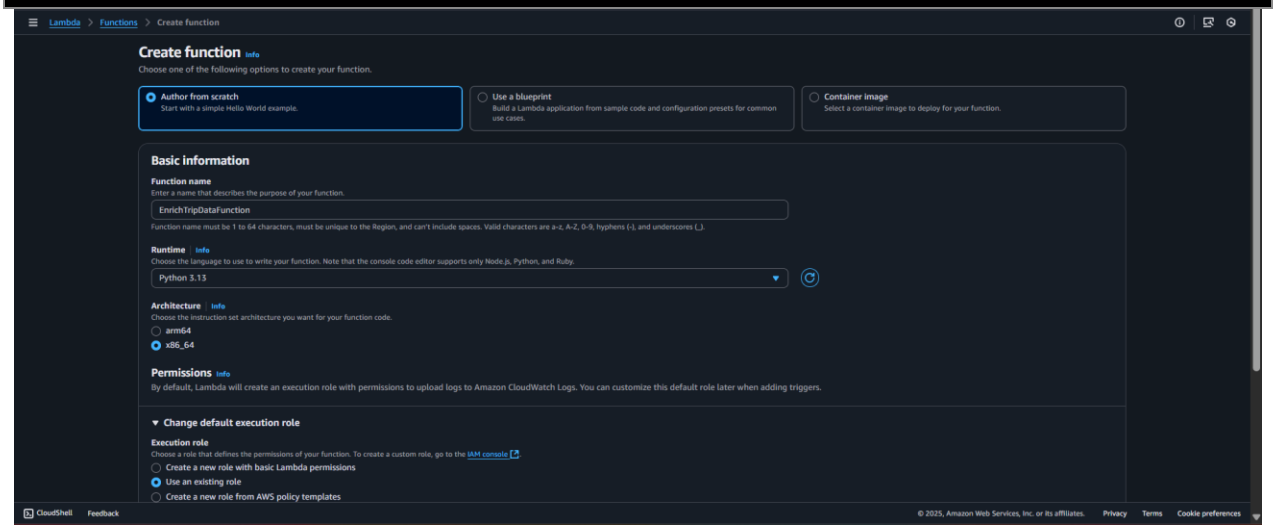
Configuration Steps:

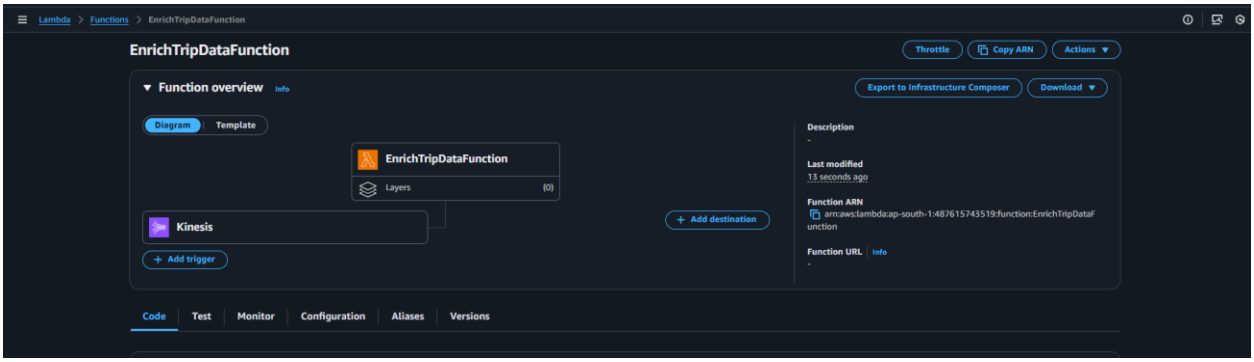
1. Go to AWS Console → Lambda → Create function.
2. Function name: EnrichTripDataFunction
3. Runtime: Python 3.12
4. Permissions (Execution Role):
   Create or attach an IAM role with the following permissions:
   - kinesis:DescribeStream
   - kinesis:GetRecords
   - kinesis:GetShardIterator
   - location:SearchPlaceIndexForPosition
   - lambda:InvokeFunction
   - logs:CreateLogGroup
   - logs:CreateLogStream
   - logs:PutLogEvents

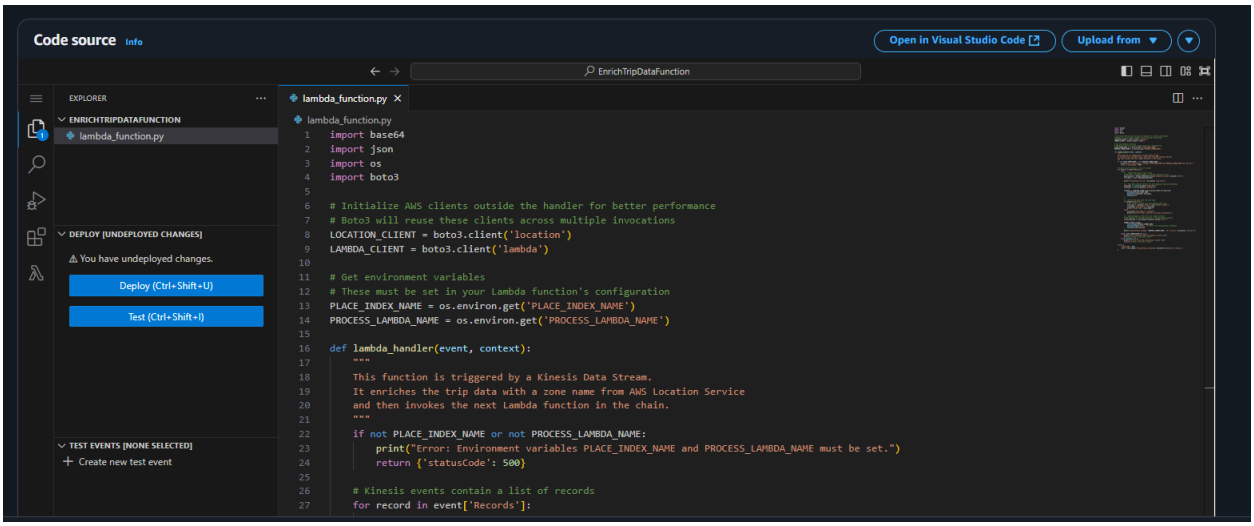
5. Add Environment Variables:
    - PLACE_INDEX_NAME = TaxiGeoIndex
    - PROCESS_LAMBDA_NAME = ProcessAndStoreFunction
6. Trigger:
   Add Kinesis Data Stream trigger from taxi-trip-stream.

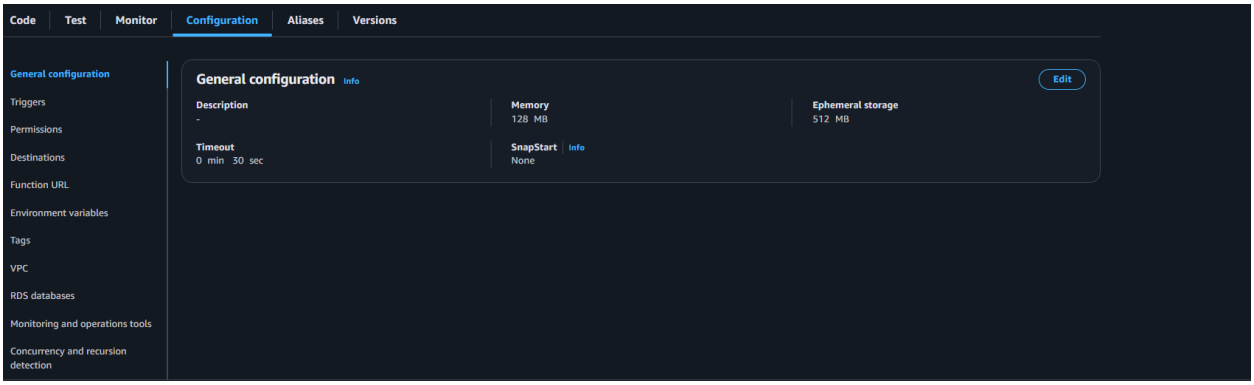
```
import base64
import json
import os
import boto3

# Initialize AWS clients outside the handler for better performance
# Boto3 will reuse these clients across multiple invocations
LOCATION_CLIENT = boto3.client('location')
SQS_CLIENT = boto3.client('sqs')

# Get environment variables
# These must be set in your Lambda function's configuration
PLACE_INDEX_NAME = os.environ.get('PLACE_INDEX_NAME')
SQS_QUEUE_URL = os.environ.get('SQS_QUEUE_URL')

def lambda_handler(event, context):
    """
    This function is triggered by a Kinesis Data Stream.
    It enriches the trip data with a zone name from AWS Location Service
    and then sends the enriched data to an SQS queue.
    """
    if not PLACE_INDEX_NAME or not SQS_QUEUE_URL:
        print("Error: Environment variables PLACE_INDEX_NAME and SQS_QUEUE_URL must be set.")
        return {'statusCode': 500}

    # Kinesis events contain a list of records
    for record in event['Records']:
        try:
            # 1. Decode the Kinesis data record
            # The data is Base64 encoded, so we must decode it first
            payload_data = base64.b64decode(record['kinesis']['data']).decode('utf-8')
            trip_data = json.loads(payload_data)

            print(f"Processing Trip ID: {trip_data.get('trip_id', 'N/A')}")

            # 2. Call AWS Location Service for zone mapping (Reverse Geocoding)
            longitude = float(trip_data['longitude'])
            latitude = float(trip_data['latitude'])

            response = LOCATION_CLIENT.search_place_index_for_position(
                IndexName=PLACE_INDEX_NAME,
                Position=[longitude, latitude],
                MaxResults=1
            )

            # 3. Enrich the data with the zone name
            if response.get('Results'):
                # The 'Label' contains the full address/zone info
                zone_name = response['Results'][0]['Place']['Label']
                trip_data['zone_name'] = zone_name
                print(f"Found Zone: {zone_name}")
            else:
                trip_data['zone_name'] = 'Unknown'
```

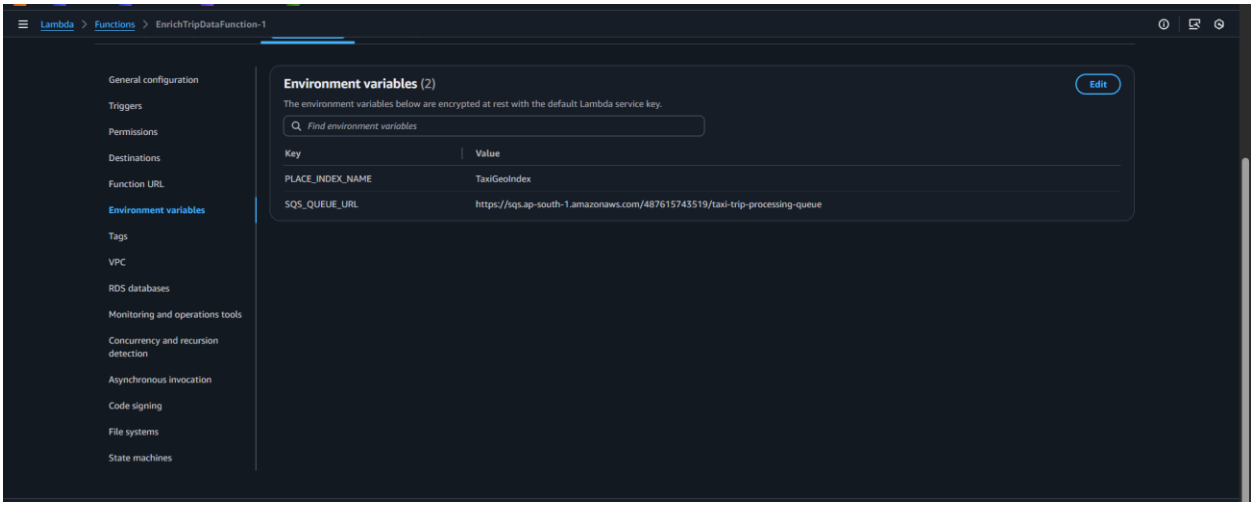
```
                print("Could not find a zone for the given coordinates.")

            # 4. Send the enriched data to the SQS queue
            # The message body must be a string
            message_body = json.dumps(trip_data)

            SQS_CLIENT.send_message(
                QueueUrl=SQS_QUEUE_URL,
                MessageBody=message_body
            )
            print(f"Successfully sent message to SQS for Trip ID: {trip_data.get('trip_id',
'N/A')}")

        except json.JSONDecodeError as e:
            print(f"Error decoding JSON from Kinesis record: {e}")
            continue # Move to the next record
        except Exception as e:
            print(f"An error occurred processing a record: {e}")
            continue # Move to the next record

    return {
        'statusCode': 200,
        'body': json.dumps(f"Successfully processed {len(event['Records'])} records and sent to
SQS.")
    }
```






Code source Info
Open in Visual Studio Code
Upload from
EnrichTripDataFunction
EXPLORER
ENRICHTRIPDATAFUNCTION
lambda_function.py
DEPLOY [UNDEPLOYED CHANGES]
You have undeployed changes.
Deploy (Ctrl+Shift+U)
Test (Ctrl+Shift+I)
TEST EVENTS [NONE SELECTED]
Create new test event
lambda_function.py
import base64
import json
import os
import boto3
# Initialize AWS clients outside the handler for better performance
# Boto3 will reuse these clients across multiple invocations
LOCATION_CLIENT = boto3.client('location')
LAMBDA_CLIENT = boto3.client('lambda')
# Get environment variables
# These must be set in your Lambda function's configuration
PLACE_INDEX_NAME = os.environ.get('PLACE_INDEX_NAME')
PROCESS_LAMBDA_NAME = os.environ.get('PROCESS_LAMBDA_NAME')
def lambda_handler(event, context):
"""
This function is triggered by a Kinesis Data Stream.
It enriches the trip data with a zone name from AWS Location Service
and then invokes the next Lambda function in the chain.
"""
if not PLACE_INDEX_NAME or not PROCESS_LAMBDA_NAME:
print("Error: Environment variables PLACE_INDEX_NAME and PROCESS_LAMBDA_NAME must be set.")
return {'statusCode': 500}
# Kinesis events contain a list of records
for record in event['Records']:


Code
Test
Monitor
Configuration
Aliases
Versions
General configuration
Triggers
Permissions
Destinations
Function URL
Environment variables
Tags
VPC
RDS databases
Monitoring and operations tools
Concurrency and recursion detection
General configuration Info
Edit
Description
-
Memory
128 MB
Ephemeral storage
512 MB
Timeout
0 min 30 sec
SnapStart Info
None


Lambda
Functions
EnrichTripDataFunction-1
General configuration
Triggers
Permissions
Destinations
Function URL
Environment variables
Tags
VPC
RDS databases
Monitoring and operations tools
Concurrency and recursion detection
Asynchronous invocation
Code signing
File systems
State machines
Environment variables (2)
Edit
The environment variables below are encrypted at rest with the default Lambda service key.
Find environment variables
Key
Value
PLACE_INDEX_NAME
TaxiGeoIndex
SQS_QUEUE_URL
https://sqs.ap-south-1.amazonaws.com/487615743519/taxi-trip-processing-queue

Step 8: Create AWS Lambda Function 2 (Process and Store Lambda)

Function Name: ProcessAndStoreFunction

Purpose:

This function receives the enriched trip data from EnrichTripDataFunction.
It calculates the fare amount, stores the record into DynamoDB, and forwards the enriched record to Kinesis Firehose for delivery into S3 and Redshift.

Configuration Steps:

1. **Go to AWS Console → Lambda → Create function.**
2. **Function name:** ProcessAndStoreFunction
3. **Runtime:** Python 3.12
4. **Permissions (Execution Role):**
    - dynamodb:PutItem
    - firehose:PutRecord
    - logs:CreateLogGroup
    - logs:CreateLogStream
    - logs:PutLogEvents
5. **Add Environment Variables:**
    - DYNAMODB_TABLE_NAME = TaxiBilling
    - FIREHOSE_STREAM_NAME = taxi-trip-firehose
6. **No Trigger needed** (this function is **invoked by Lambda 1**).

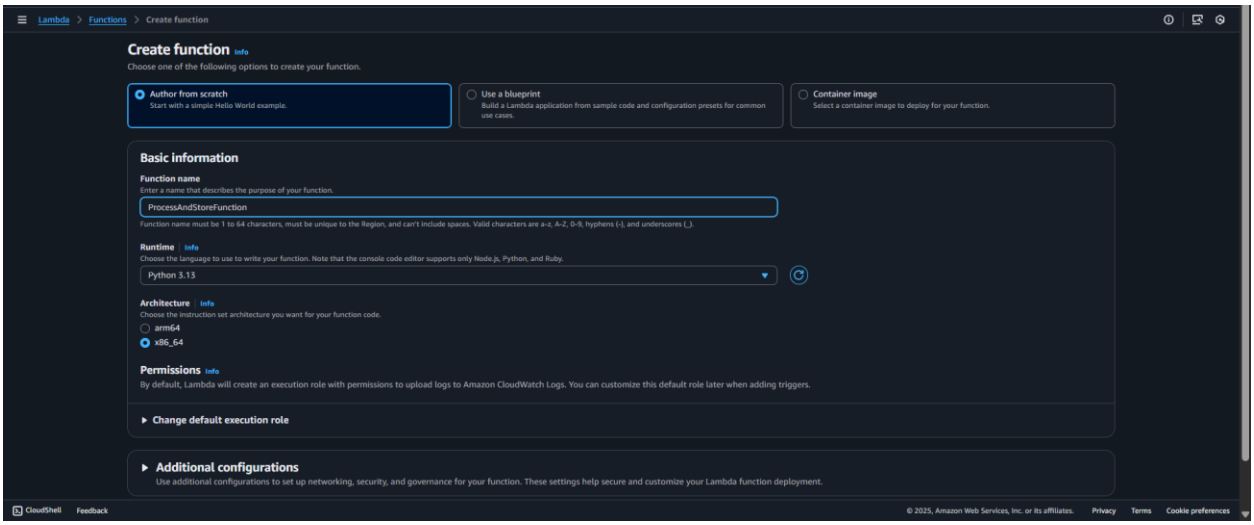
Code (Deploy this inside Lambda console or via ZIP upload):

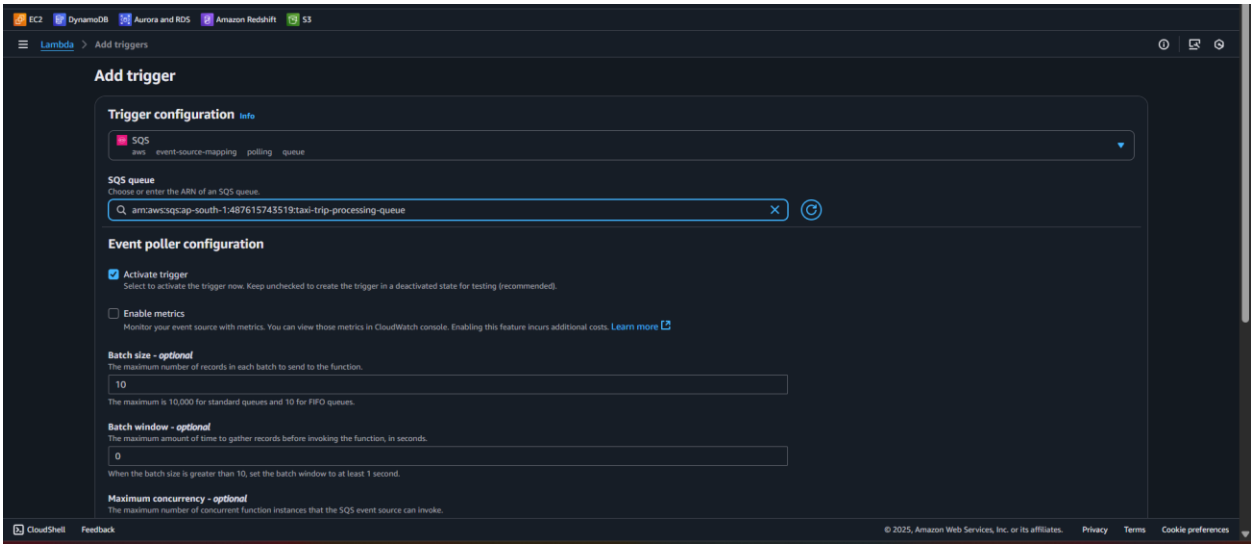
```
import json
import os
import boto3
from decimal import Decimal

# Initialize AWS clients
DYNAMODB = boto3.resource('dynamodb')
FIREHOSE_CLIENT = boto3.client('firehose')

# Get environment variables
DYNAMODB_TABLE_NAME = os.environ.get('DYNAMODB_TABLE_NAME')
FIREHOSE_STREAM_NAME = os.environ.get('FIREHOSE_STREAM_NAME')
```

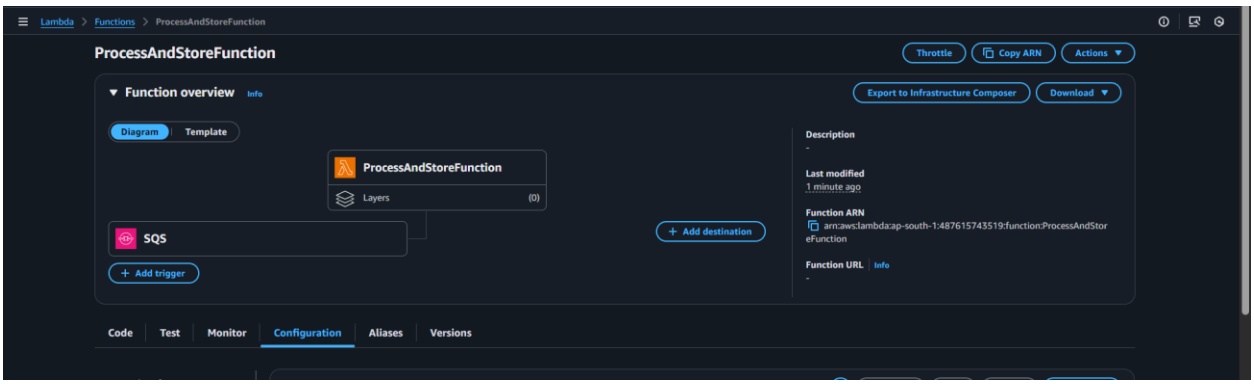
```
def calculate_fare(distance_km, zone_name):
    """
    Calculates the fare based on distance and applies a surcharge for specific zones.
    """
    try:
        distance = float(distance_km)
    except (ValueError, TypeError):
        distance = 0.0

    base_fare = 50.0
    rate_per_km = 18.5
    fare = base_fare + (distance * rate_per_km)

    # Apply a surcharge if the trip is associated with an airport
    if zone_name and 'Airport' in zone_name:
        fare += 100.0

    return round(fare, 2)

def floats_to_decimal(data_dict):
    """
    Recursively converts all float values in a dictionary to the Decimal type,
    which is required for DynamoDB.
    """
    for key, value in data_dict.items():
        if isinstance(value, float):
            data_dict[key] = Decimal(str(value))
        elif isinstance(value, dict):
            floats_to_decimal(value)
    return data_dict

def validate_trip_data(trip_data):
    """
    Validate that all required fields are present and in correct format.
    """
    required_fields = ['trip_id', 'taxi_id', 'pickup_datetime', 'pickup_lat',
                       'pickup_long', 'drop_lat', 'drop_long', 'distance_km']

    missing_fields = [field for field in required_fields if field not in trip_data or
trip_data[field] is None]

    if missing_fields:
        return False, f"Missing required fields: {missing_fields}"

    # Validate numeric fields
    numeric_fields = ['pickup_lat', 'pickup_long', 'drop_lat', 'drop_long', 'distance_km']
    for field in numeric_fields:
        try:
            float(trip_data[field])
        except (ValueError, TypeError):
            return False, f"Invalid numeric value for {field}: {trip_data[field]}"

    return True, "Valid"

def lambda_handler(event, context):
    """
    This function is triggered by an SQS queue. For each message, it calculates
    the trip fare, stores the result in DynamoDB, and sends the data to Kinesis Firehose.
    """
    if not DYNAMODB_TABLE_NAME or not FIREHOSE_STREAM_NAME:
        error_msg = "Error: Environment variables DYNAMODB_TABLE_NAME and FIREHOSE_STREAM_NAME must
be set."
        print(error_msg)
        return {
```

```
            'statusCode': 500,
            'body': json.dumps(error_msg)
        }

    processed_count = 0
    failed_count = 0
    table = DYNAMODB.Table(DYNAMODB_TABLE_NAME)

    print(f"Processing {len(event['Records'])} records from SQS")

    # SQS triggers send a batch of records in the 'Records' key
    for record in event['Records']:
        trip_id = 'Unknown'

        try:
            # The actual message from the previous Lambda is a JSON string in the 'body'
            message_body = record['body']
            trip_data = json.loads(message_body)

            trip_id = trip_data.get('trip_id', 'N/A')
            print(f"🔧 Processing Trip ID: {trip_id}")

            # 1. Validate the incoming data
            is_valid, validation_msg = validate_trip_data(trip_data)
            if not is_valid:
                print(f"❌ Validation failed for Trip ID {trip_id}: {validation_msg}")
                failed_count += 1
                continue

            # 2. Calculate the fare
            fare = calculate_fare(trip_data.get('distance_km'), trip_data.get('zone_name'))
            trip_data['fare_amount'] = fare
            print(f"💰 Calculated Fare: {fare} for distance: {trip_data.get('distance_km')}km")

            # 3. Set default values for optional fields
            trip_data.setdefault('passenger_count', 0)
            trip_data.setdefault('extra_charges', 0.0)
            trip_data.setdefault('tip_amount', 0.0)
            trip_data.setdefault('tolls_amount', 0.0)
            trip_data.setdefault('payment_type', 'CASH')

            # Calculate total amount (fare + extra charges + tip + tolls)
            total_amount = fare + float(trip_data.get('extra_charges', 0.0)) +
float(trip_data.get('tip_amount', 0.0)) + float(trip_data.get('tolls_amount', 0.0))
            trip_data['total_amount'] = round(total_amount, 2)

            # 4. Prepare DynamoDB item with explicit field mapping
            dynamodb_item = {
                'trip_id': trip_data['trip_id'],
                'taxi_id': trip_data.get('taxi_id', ''),
                'pickup_datetime': trip_data.get('pickup_datetime', ''),
                'pickup_lat': Decimal(str(trip_data.get('pickup_lat', 0.0))),
                'pickup_long': Decimal(str(trip_data.get('pickup_long', 0.0))),
                'drop_lat': Decimal(str(trip_data.get('drop_lat', 0.0))),
                'drop_long': Decimal(str(trip_data.get('drop_long', 0.0))),
                'distance_km': Decimal(str(trip_data.get('distance_km', 0.0))),
                'zone_name': trip_data.get('zone_name', ''),
                'fare_amount': Decimal(str(fare)),
                'passenger_count': int(trip_data.get('passenger_count', 0)),
                'extra_charges': Decimal(str(trip_data.get('extra_charges', 0.0))),
                'tip_amount': Decimal(str(trip_data.get('tip_amount', 0.0))),
                'tolls_amount': Decimal(str(trip_data.get('tolls_amount', 0.0))),
                'total_amount': Decimal(str(trip_data.get('total_amount', 0.0))),
                'payment_type': trip_data.get('payment_type', 'CASH')
```

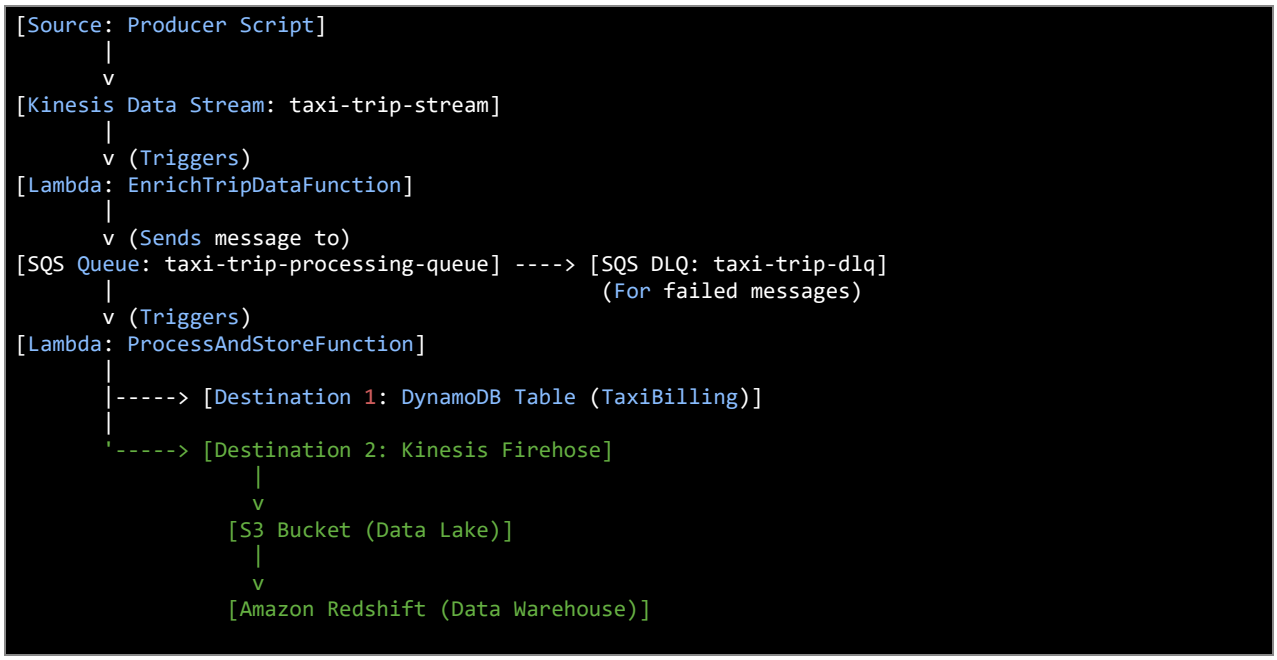
```
            }

            # 5. Write the complete record to DynamoDB
            table.put_item(Item=dynamodb_item)
            print(f"✅ Successfully wrote to DynamoDB table: {DYNAMODB_TABLE_NAME}")

            # 6. Prepare data for Firehose (ensure all fields for Redshift are present)
            firehose_data = trip_data.copy()

            # Add fields expected by Redshift schema
            firehose_data.setdefault('dropoff_datetime', None)
            firehose_data.setdefault('pickup_latitude', trip_data.get('pickup_lat'))
            firehose_data.setdefault('pickup_longitude', trip_data.get('pickup_long'))
            firehose_data.setdefault('dropoff_latitude', trip_data.get('drop_lat'))
            firehose_data.setdefault('dropoff_longitude', trip_data.get('drop_long'))

            # 7. Send the complete record to Kinesis Firehose for analytics/S3 storage
            # Firehose expects a newline character for batching into files
            firehose_payload = (json.dumps(firehose_data) + '\n').encode('utf-8')

            FIREHOSE_CLIENT.put_record(
                DeliveryStreamName=FIREHOSE_STREAM_NAME,
                Record={'Data': firehose_payload}
            )
            print(f"📤 Successfully sent to Firehose stream: {FIREHOSE_STREAM_NAME}")

            processed_count += 1
            print(f"✅ Completed processing Trip ID: {trip_id}")

        except json.JSONDecodeError as e:
            print(f"❌ JSON decode error for record {trip_id}: {e}")
            failed_count += 1
        except Exception as e:
            print(f"❌ Error processing record {trip_id}: {e}")
            failed_count += 1

    # Final summary
    print(f"\n📊 Processing Summary:")
    print(f"✅ Successfully processed: {processed_count} records")
    print(f"❌ Failed: {failed_count} records")
    print(f"📥 Total received from SQS: {len(event['Records'])} records")

    return {
        'statusCode': 200,
        'body': json.dumps({
            'message': f'Successfully processed {processed_count} records from SQS',
            'processed': processed_count,
            'failed': failed_count,
            'total': len(event['Records'])
        })
    }
```


Lambda > Functions > Create function
Create function Info
Choose one of the following options to create your function.
Author from scratch
Start with a simple Hello World example.
Use a blueprint
Build a Lambda application from sample code and configuration presets for common use cases.
Container image
Select a container image to deploy for your function.
Basic information
Function name
Enter a name that describes the purpose of your function.
ProcessAndStoreFunction
Function name must be 1 to 64 characters, must be unique to the Region, and can't include spaces. Valid characters are a-z, A-Z, 0-9, hyphens (-), and underscores (_).
Runtime Info
Choose the language to use to write your function. Note that the console code editor supports only Node.js, Python, and Ruby.
Python 3.13
Architecture Info
Choose the instruction set architecture you want for your function code.
arm64
x86_64
Permissions Info
By default, Lambda will create an execution role with permissions to upload logs to Amazon CloudWatch Logs. You can customize this default role later when adding triggers.
Change default execution role
Additional configurations
Use additional configurations to set up networking, security, and governance for your function. These settings help secure and customize your Lambda function deployment.
CloudShell
Feedback
© 2025, Amazon Web Services, Inc. or its affiliates.
Privacy
Terms
Cookie preferences


EC2
DynamoDB
Aurora and RDS
Amazon Redshift
S3
Lambda > Add triggers
Add trigger
Trigger configuration Info
SQS
aws event-source-mapping polling queue
SQS queue
Choose or enter the ARN of an SQS queue.
arn:aws:sqs:ap-south-1:487615743519:taxi-trip-processing-queue
Event poller configuration
Activate trigger
Select to activate the trigger now. Keep unchecked to create the trigger in a deactivated state for testing (recommended).
Enable metrics
Monitor your event source with metrics. You can view those metrics in CloudWatch console. Enabling this feature incurs additional costs. Learn more
Batch size - optional
The maximum number of records in each batch to send to the function.
10
The maximum is 10,000 for standard queues and 10 for FIFO queues.
Batch window - optional
The maximum amount of time to gather records before invoking the function, in seconds.
0
When the batch size is greater than 10, set the batch window to at least 1 second.
Maximum concurrency - optional
The maximum number of concurrent function instances that the SQS event source can invoke.
CloudShell
Feedback
© 2025, Amazon Web Services, Inc. or its affiliates.
Privacy
Terms
Cookie preferences


Lambda > Functions > ProcessAndStoreFunction
ProcessAndStoreFunction
Throttle
Copy ARN
Actions
Function overview Info
Export to Infrastructure Composer
Download
Diagram
Template
ProcessAndStoreFunction
Layers (0)
SQS
Add trigger
Add destination
Description
-
Last modified
1 minute ago
Function ARN
arn:aws:lambda:ap-south-1:487615743519:function:ProcessAndStoreFunction
Function URL Info
-
Code
Test
Monitor
Configuration
Aliases
Versions

Lambda Chaining Summary:



Step 9: Create Amazon Redshift Serverless (Analytics Warehouse)

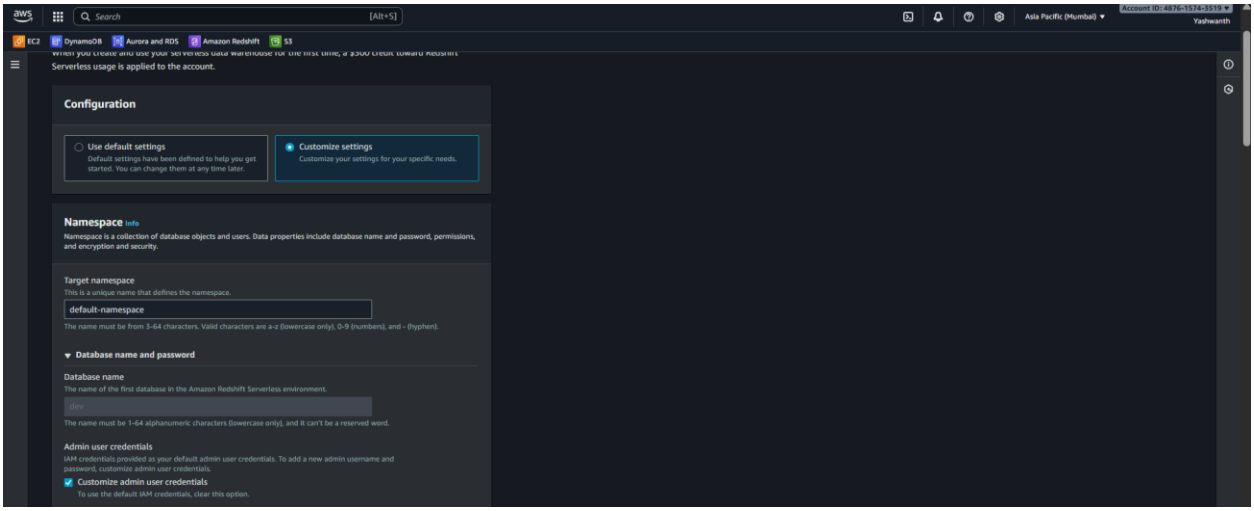
Steps:

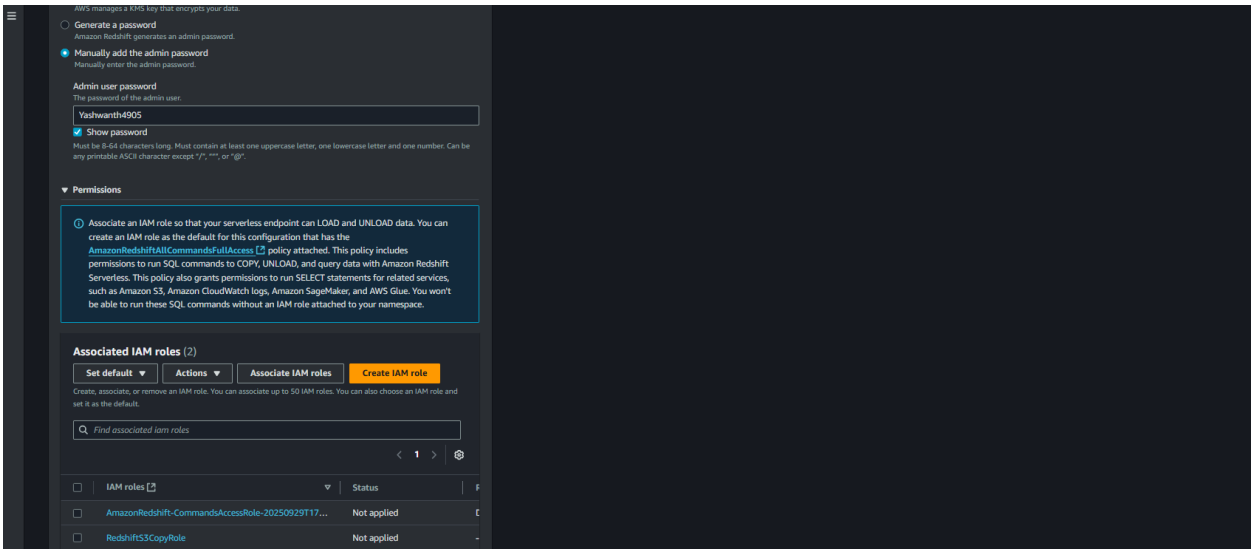
1. Go to Amazon Redshift → Serverless → Create Workgroup.
2. Workgroup name: taxi-trip-analytics
3. Database name: dev
4. Create a new namespace.
5. Once ready, open Query Editor v2 → connect to database.

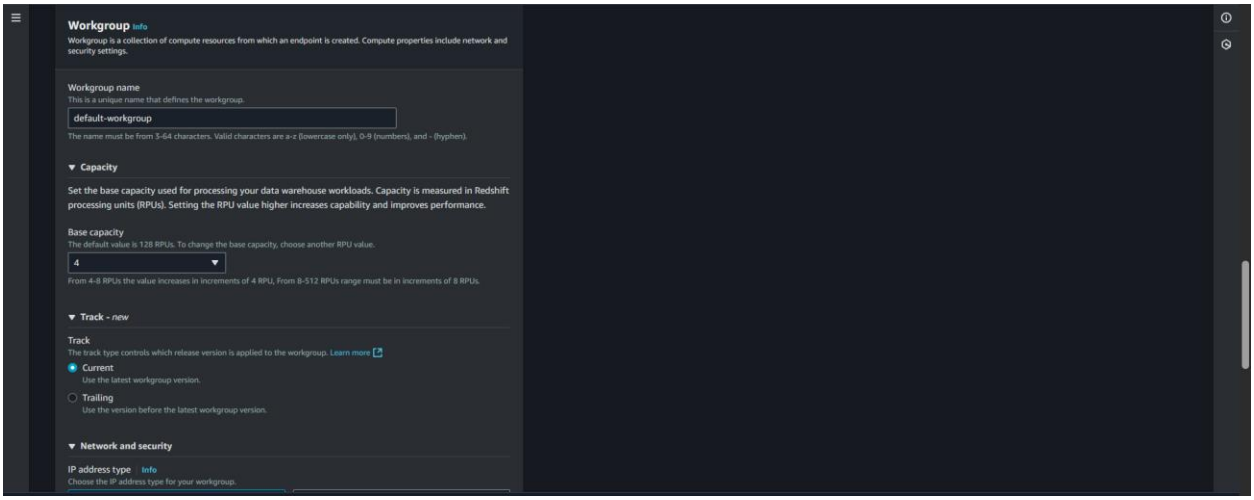
```
CREATE TABLE IF NOT EXISTS enriched_trip_data (
    trip_id              VARCHAR(50)       NOT NULL,
    taxi_id              VARCHAR(50),
    pickup_datetime      TIMESTAMP,
    dropoff_datetime     TIMESTAMP,
    passenger_count      INT DEFAULT 0,
    distance_km          DOUBLE PRECISION,
    pickup_longitude     DOUBLE PRECISION,
    pickup_latitude      DOUBLE PRECISION,
    dropoff_longitude    DOUBLE PRECISION,
    dropoff_latitude     DOUBLE PRECISION,
    zone_name            VARCHAR(255),
    payment_type         VARCHAR(20),
    fare_amount          DOUBLE PRECISION,
    extra_charges        DOUBLE PRECISION DEFAULT 0,
    tip_amount           DOUBLE PRECISION DEFAULT 0,
    tolls_amount         DOUBLE PRECISION DEFAULT 0,
    total_amount         DOUBLE PRECISION DEFAULT 0,
    created_at           TIMESTAMP DEFAULT CURRENT_TIMESTAMP
)
DISTSTYLE AUTO
SORTKEY(pickup_datetime);
```

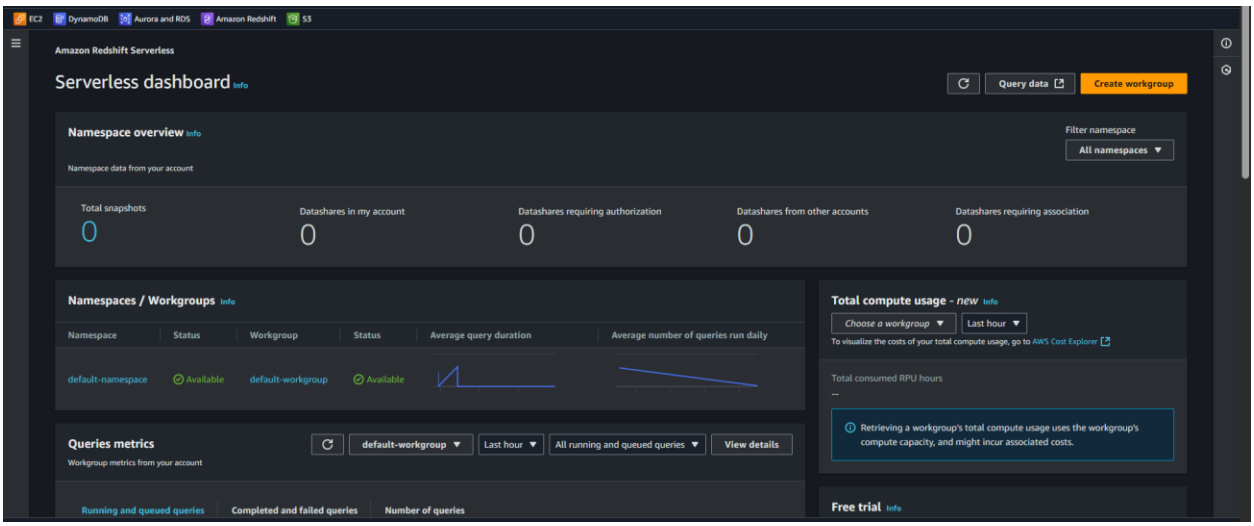
Purpose:

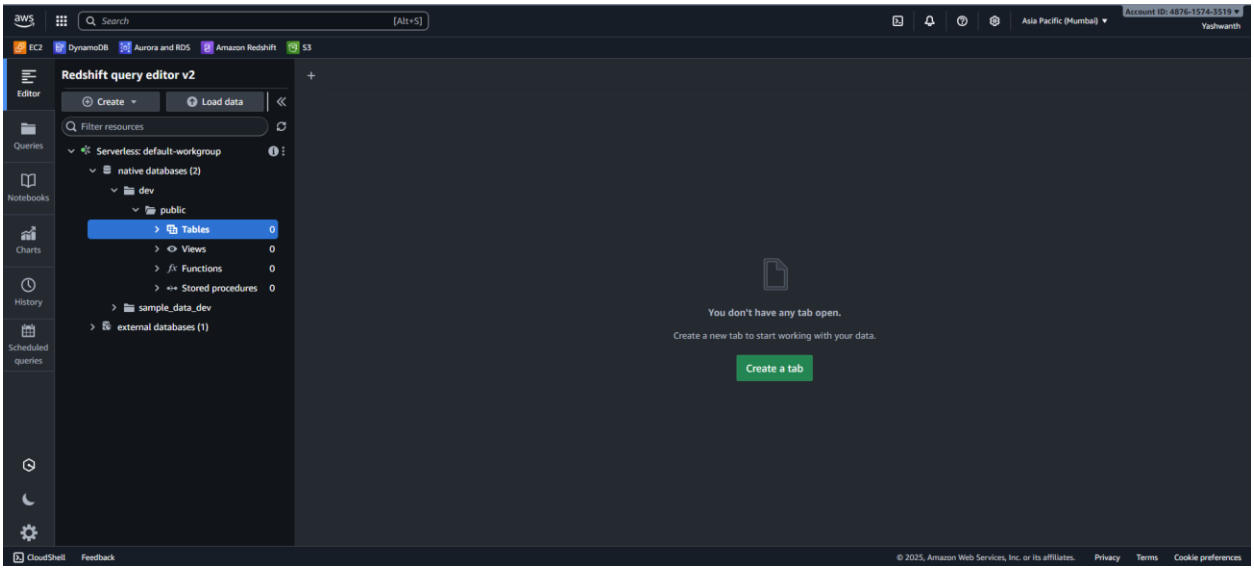
Stores cleaned and enriched trip data for BI and analytics dashboards.

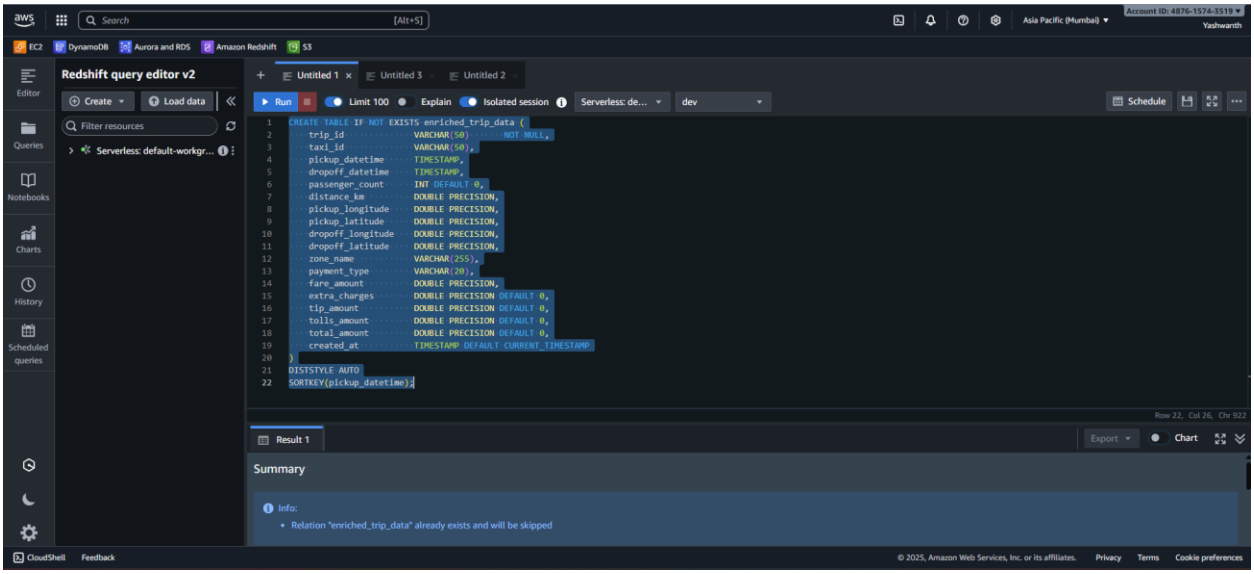



AWS manages a KMS key that encrypts your data.
Generate a password
Amazon Redshift generates an admin password.
Manually add the admin password
Manually enter the admin password.
Admin user password
The password of the admin user.
Yashwanth4905
Show password
Must be 8-64 characters long. Must contain at least one uppercase letter, one lowercase letter and one number. Can be any printable ASCII character except "/", "", or "@".
Permissions
Associate an IAM role so that your serverless endpoint can LOAD and UNLOAD data. You can create an IAM role as the default for this configuration that has the AmazonRedshiftAllCommandsFullAccess policy attached. This policy includes permissions to run SQL commands to COPY, UNLOAD, and query data with Amazon Redshift Serverless. This policy also grants permissions to run SELECT statements for related services, such as Amazon S3, Amazon CloudWatch logs, Amazon SageMaker, and AWS Glue. You won't be able to run these SQL commands without an IAM role attached to your namespace.
Associated IAM roles (2)
Set default
Actions
Associate IAM roles
Create IAM role
Create, associate, or remove an IAM role. You can associate up to 50 IAM roles. You can also choose an IAM role and set it as the default.
Find associated iam roles
IAM roles
Status
AmazonRedshift-CommandsAccessRole-20250929T17...
Not applied
RedshiftS3CopyRole
Not applied

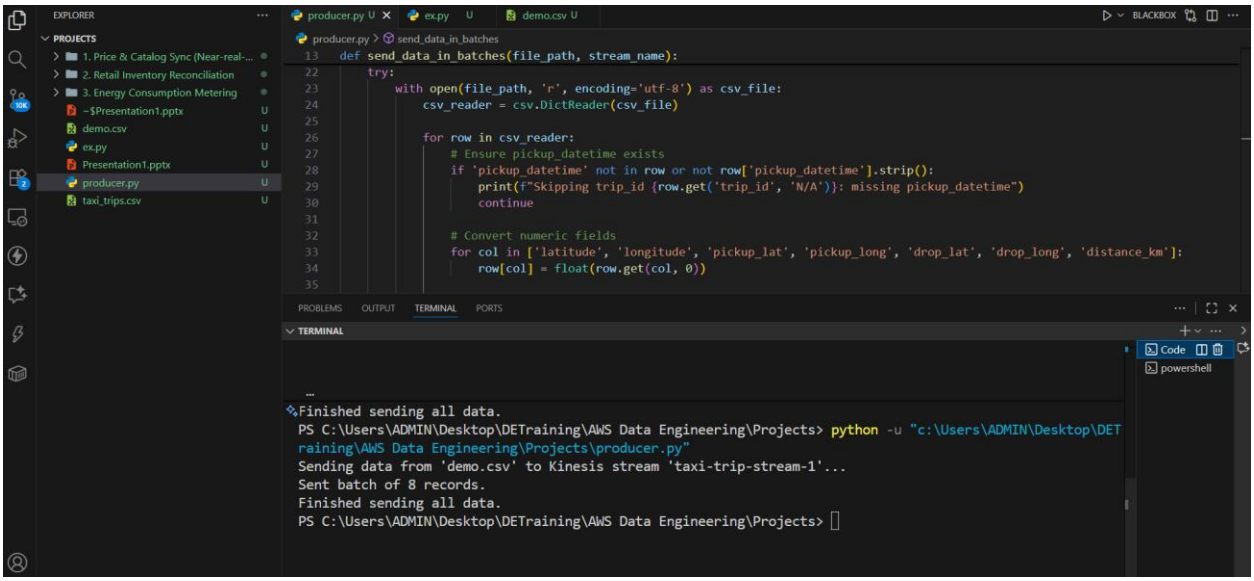

Workgroup Info
Workgroup is a collection of compute resources from which an endpoint is created. Compute properties include network and security settings.
Workgroup name
This is a unique name that defines the workgroup.
default-workgroup
The name must be from 3-64 characters. Valid characters are a-z (lowercase only), 0-9 (numbers), and - (hyphen).
Capacity
Set the base capacity used for processing your data warehouse workloads. Capacity is measured in Redshift processing units (RPUs). Setting the RPU value higher increases capability and improves performance.
Base capacity
The default value is 128 RPUs. To change the base capacity, choose another RPU value.
4
From 4-8 RPUs the value increases in increments of 4 RPU, From 8-512 RPUs range must be in increments of 8 RPUs.
Track - new
Track
The track type controls which release version is applied to the workgroup. Learn more
Current
Use the latest workgroup version.
Trailing
Use the version before the latest workgroup version.
Network and security
IP address type Info
Choose the IP address type for your workgroup.


EC2
DynamoDB
Aurora and RDS
Amazon Redshift
S3
Amazon Redshift Serverless
Serverless dashboard Info
Query data
Create workgroup
Namespace overview Info
Namespace data from your account
Filter namespace
All namespaces
Total snapshots
0
Datashares in my account
0
Datashares requiring authorization
0
Datashares from other accounts
0
Datashares requiring association
0
Namespaces / Workgroups Info
Namespace
Status
Workgroup
Status
Average query duration
Average number of queries run daily
default-namespace
Available
default-workgroup
Available
Total compute usage - new Info
Choose a workgroup
Last hour
To visualize the costs of your total compute usage, go to AWS Cost Explorer
Total consumed RPU hours
--
Retrieving a workgroup's total compute usage uses the workgroup's compute capacity, and might incur associated costs.
Queries metrics
Workgroup metrics from your account
default-workgroup
Last hour
All running and queued queries
View details
Running and queued queries
Completed and failed queries
Number of queries
Free trial Info

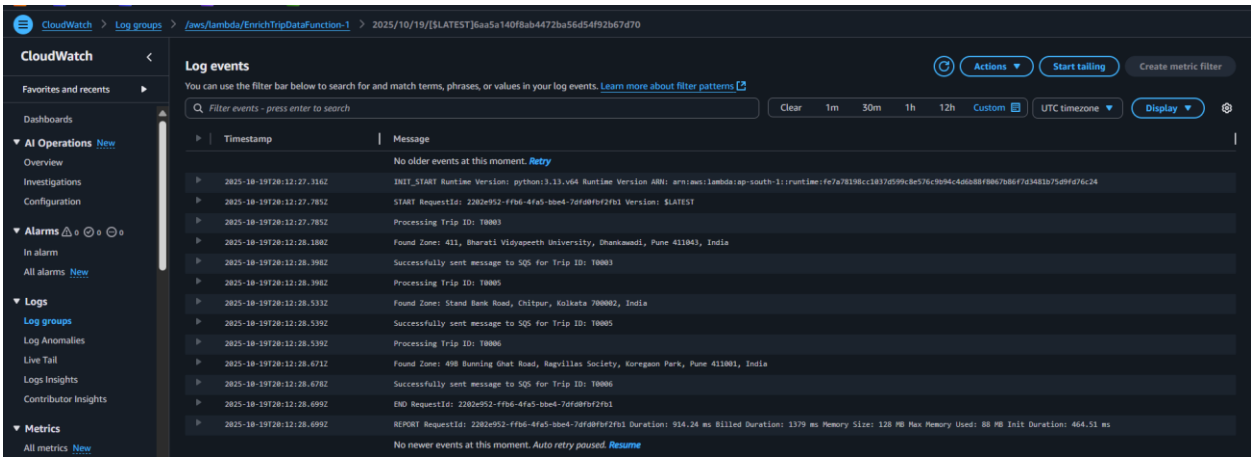

Redshift query editor v2
Create
Load data
Filter resources
Serverless: default-workgroup
native databases (2)
dev
public
Tables 0
Views 0
Functions 0
Stored procedures 0
sample_data_dev
external databases (1)
You don't have any tab open.
Create a new tab to start working with your data.
Create a tab


Redshift query editor v2
Untitled 1
Untitled 3
Untitled 2
Run
Limit 100
Explain
Isolated session
Schedule
CREATE TABLE IF NOT EXISTS enriched_trip_data (
trip_id VARCHAR(50) NOT NULL,
taxi_id VARCHAR(50),
pickup_datetime TIMESTAMP,
dropoff_datetime TIMESTAMP,
passenger_count INT DEFAULT 0,
distance_km DOUBLE PRECISION,
pickup_longitude DOUBLE PRECISION,
pickup_latitude DOUBLE PRECISION,
dropoff_longitude DOUBLE PRECISION,
dropoff_latitude DOUBLE PRECISION,
zone_name VARCHAR(255),
payment_type VARCHAR(20),
fare_amount DOUBLE PRECISION,
extra_charges DOUBLE PRECISION DEFAULT 0,
tip_amount DOUBLE PRECISION DEFAULT 0,
tolls_amount DOUBLE PRECISION DEFAULT 0,
total_amount DOUBLE PRECISION DEFAULT 0,
created_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP
)
DISTSTYLE AUTO
SORTKEY(pickup_datetime);
Result 1
Export
Chart
Summary
Info:
Relation "enriched_trip_data" already exists and will be skipped

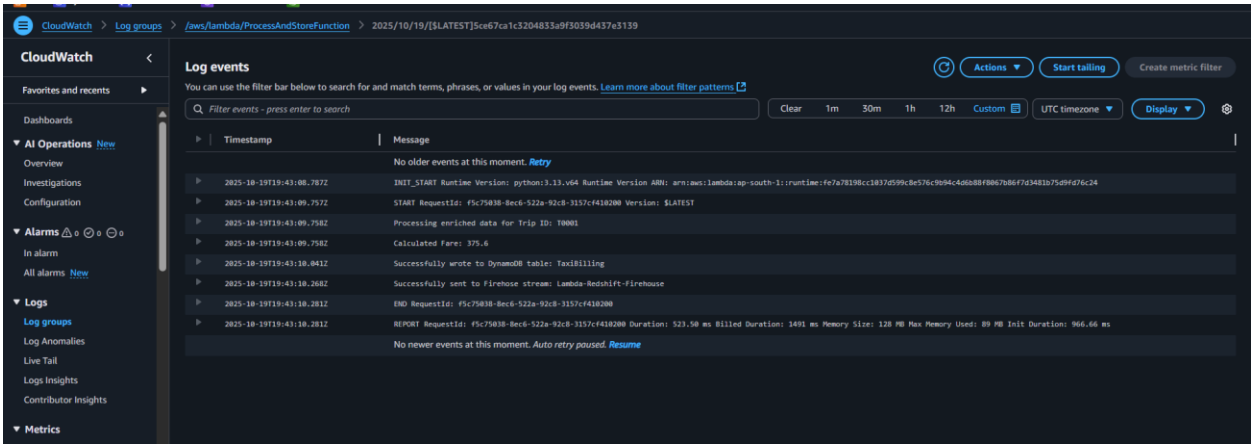
Run Producer.py



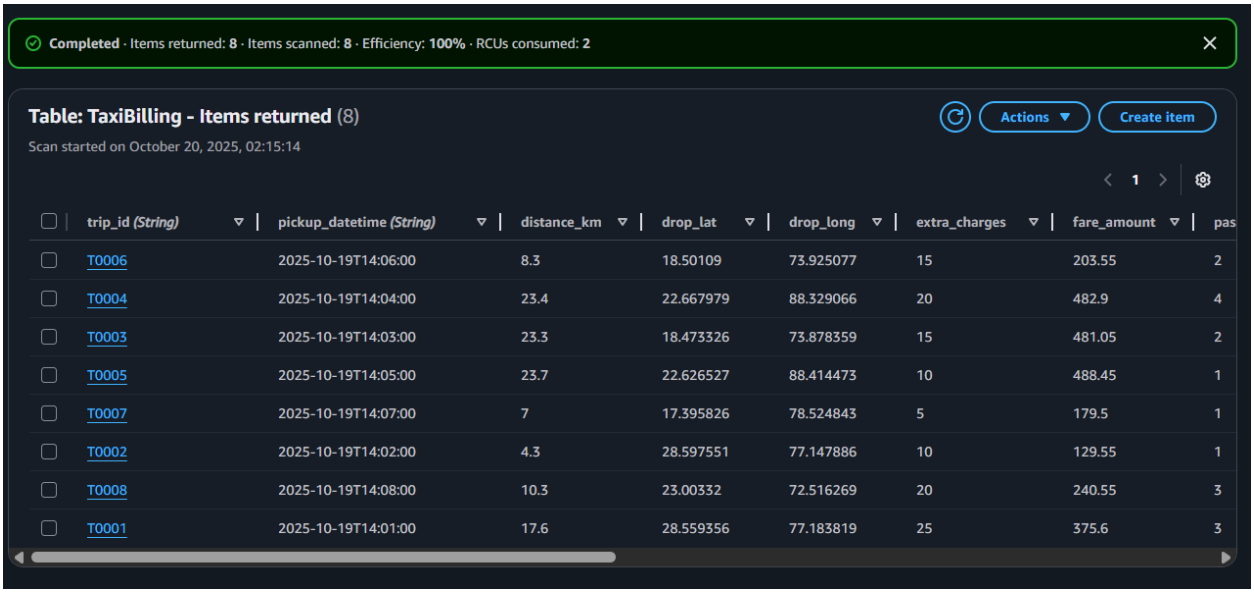
EnrichTripDataFunction Lambda Logs:



ProcessAndStoreFunction Lambda Logs:

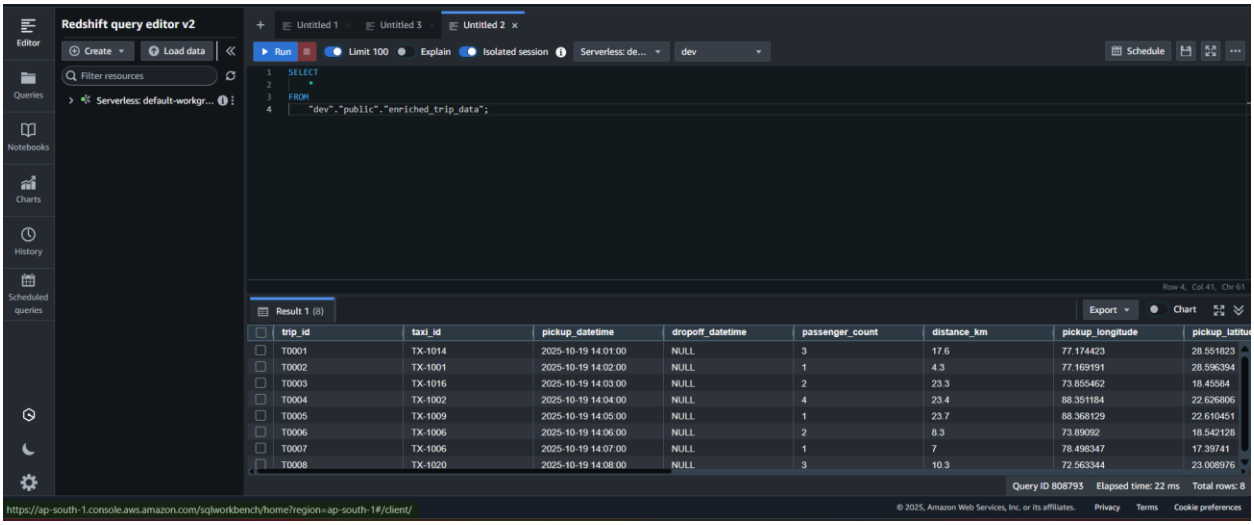


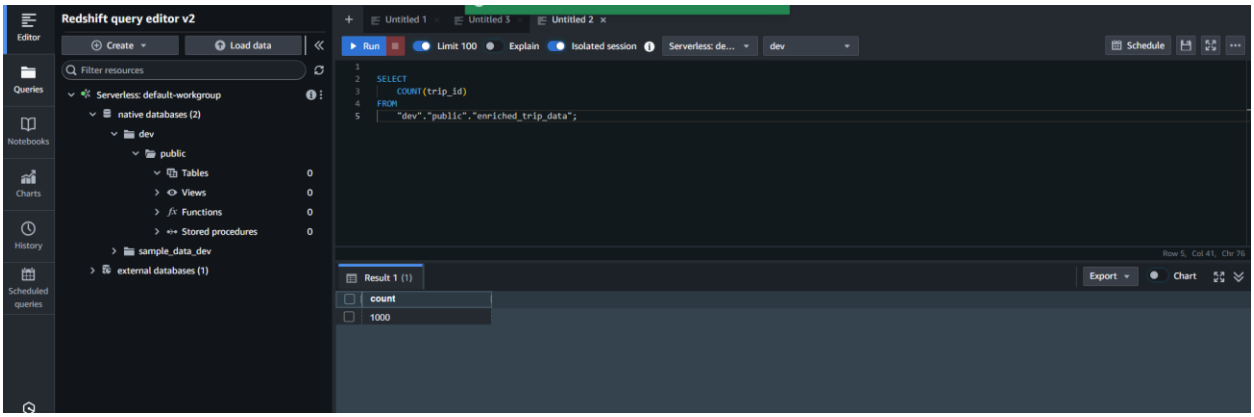
DynamoDB Table:



trip_id (String)	pickup_datetime (String)	distance_km	drop_lat	drop_long	extra_charges	fare_amount	pas
T0006	2025-10-19T14:06:00	8.3	18.50109	73.925077	15	203.55	2
T0004	2025-10-19T14:04:00	23.4	22.667979	88.329066	20	482.9	4
T0003	2025-10-19T14:03:00	23.3	18.473326	73.878359	15	481.05	2
T0005	2025-10-19T14:05:00	23.7	22.626527	88.414473	10	488.45	1
T0007	2025-10-19T14:07:00	7	17.395826	78.524843	5	179.5	1
T0002	2025-10-19T14:02:00	4.3	28.597551	77.147886	10	129.55	1
T0008	2025-10-19T14:08:00	10.3	23.00332	72.516269	20	240.55	3
T0001	2025-10-19T14:01:00	17.6	28.559356	77.183819	25	375.6	3

Redshift:





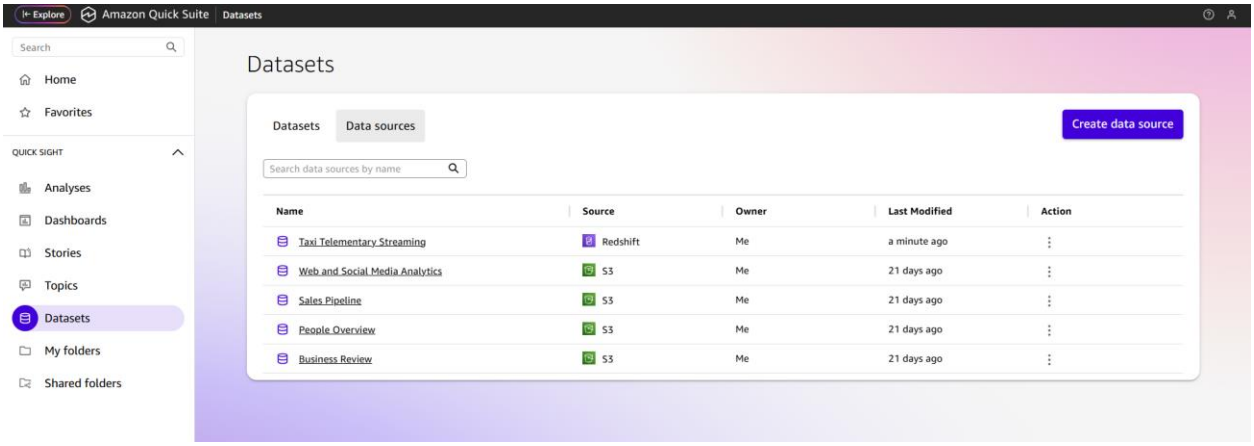
Step 10: Create Amazon QuickSight Dashboard (Visualization Layer)

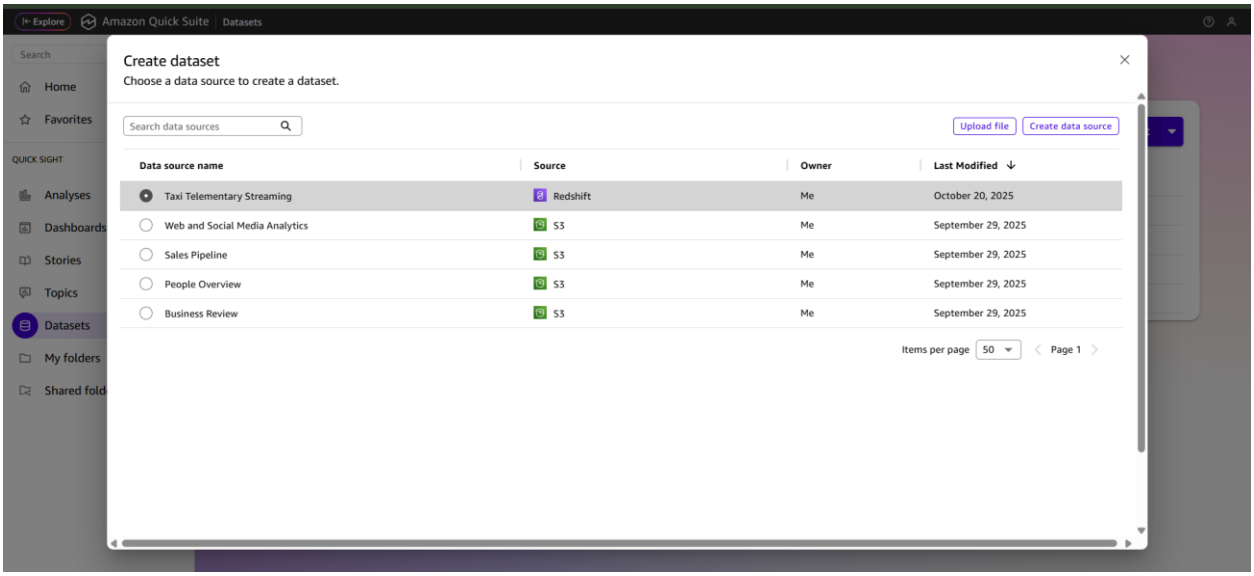
Steps:

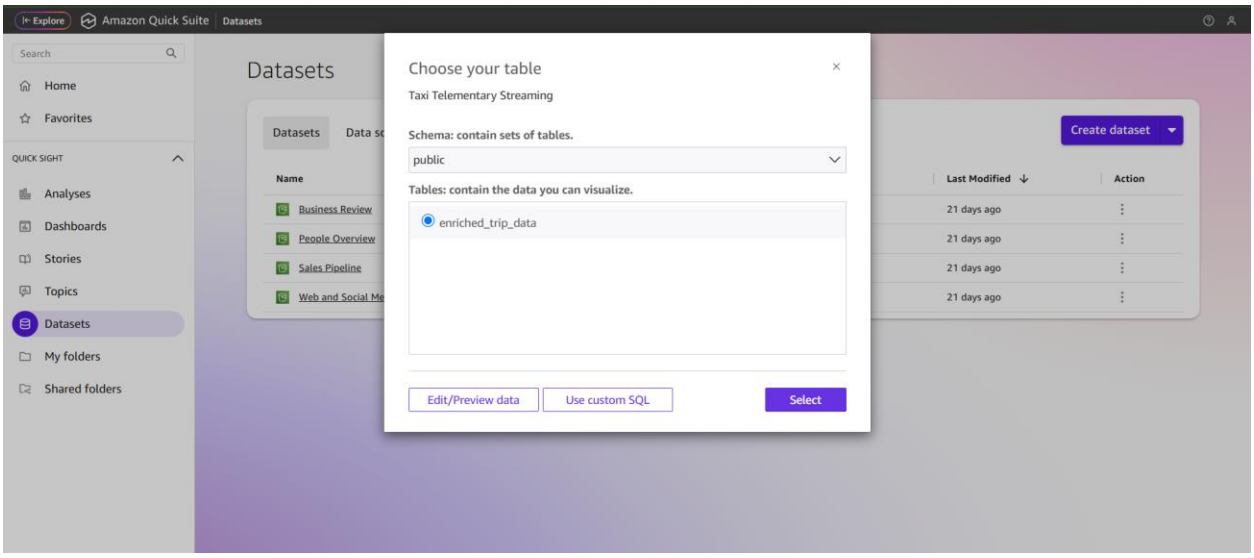
1. Go to **Amazon QuickSight → Datasets → Create new dataset**.
2. Choose **Redshift** as data source.
3. Connect to **taxi_db** in **taxi-trip-analytics** workgroup.
4. Import table taxi_trips.
5. Build dashboards for:
    - Trip volume by zone
    - Fare distribution
    - Real-time revenue by time of day

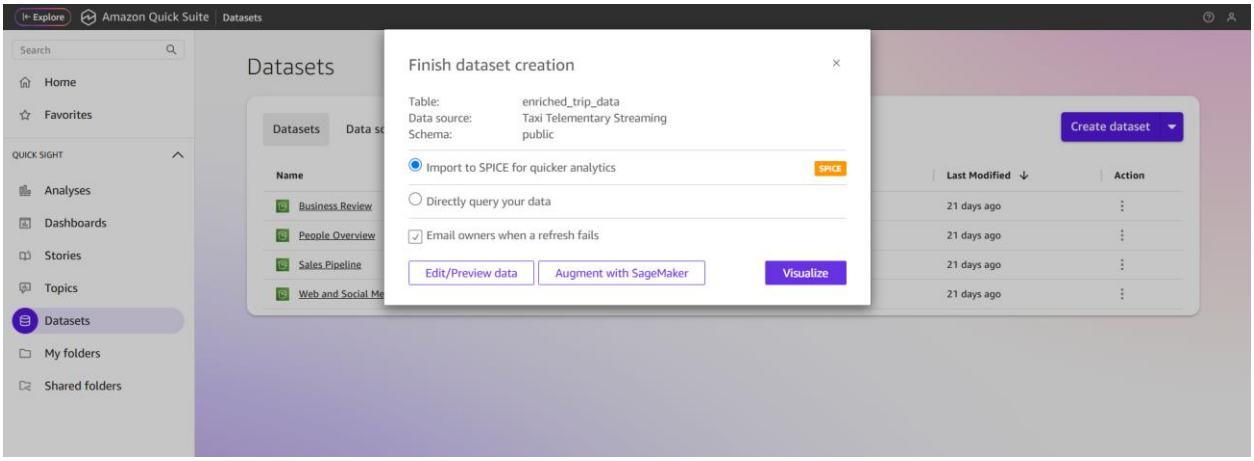
Purpose:

Provides interactive, real-time visualization and insights into taxi operations and billing trends.





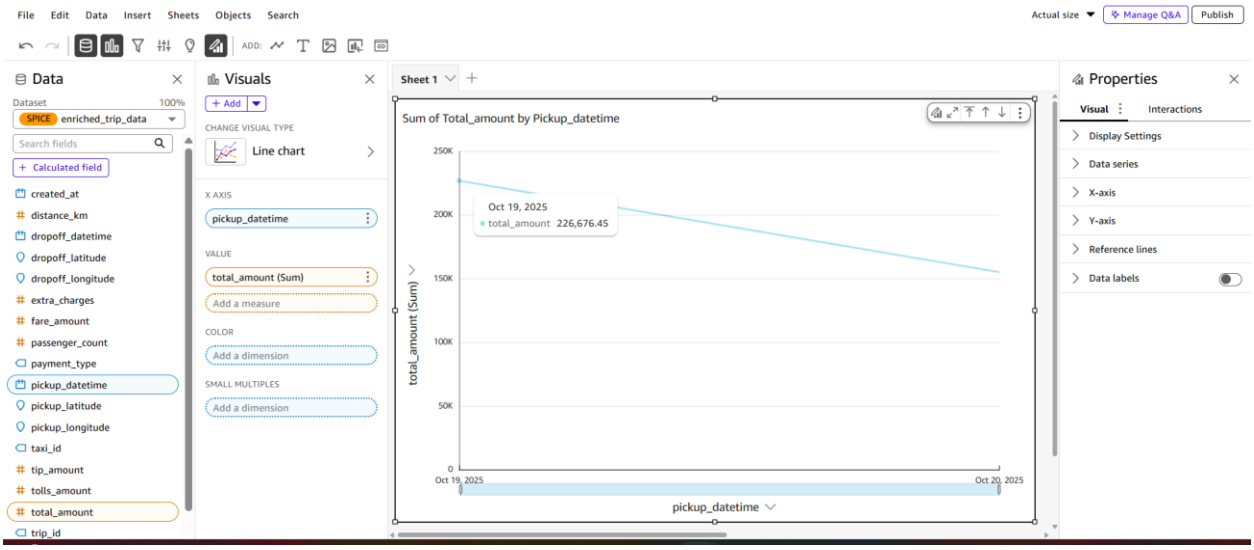

Explore
Amazon Quick Suite
Datasets
Search
Home
Favorites
QUICK SIGHT
Analyses
Dashboards
Stories
Topics
Datasets
My folders
Shared folders
Datasets
Create dataset
Name
Last Modified
Action
Business Review
People Overview
Sales Pipeline
21 days ago
Choose your table
Taxi Telementary Streaming
Schema: contain sets of tables.
public
Tables: contain the data you can visualize.
enriched_trip_data
Edit/Preview data
Use custom SQL
Select

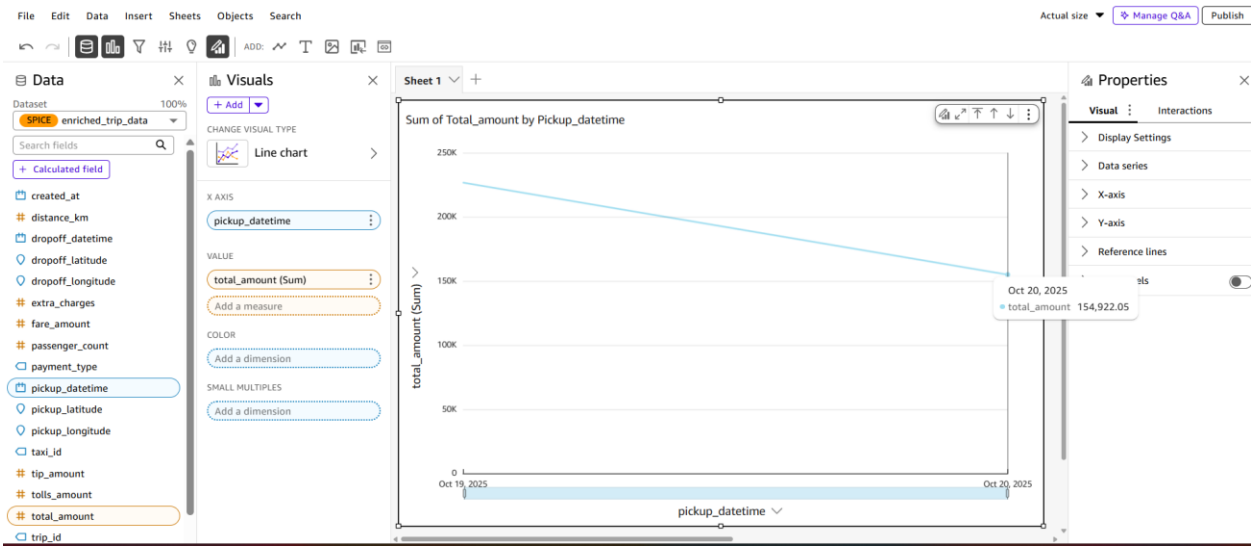

Explore
Amazon Quick Suite
Datasets
Search
Home
Favorites
QUICK SIGHT
Analyses
Dashboards
Stories
Topics
Datasets
My folders
Shared folders
Datasets
Create dataset
Name
Last Modified
Action
Business Review
People Overview
Sales Pipeline
21 days ago
Finish dataset creation
Table: enriched_trip_data
Data source: Taxi Telementary Streaming
Schema: public
Import to SPICE for quicker analytics
SPICE
Directly query your data
Email owners when a refresh fails
Edit/Preview data
Augment with SageMaker
Visualize

Charts & Visuals:

1. Revenue & Trip Trends

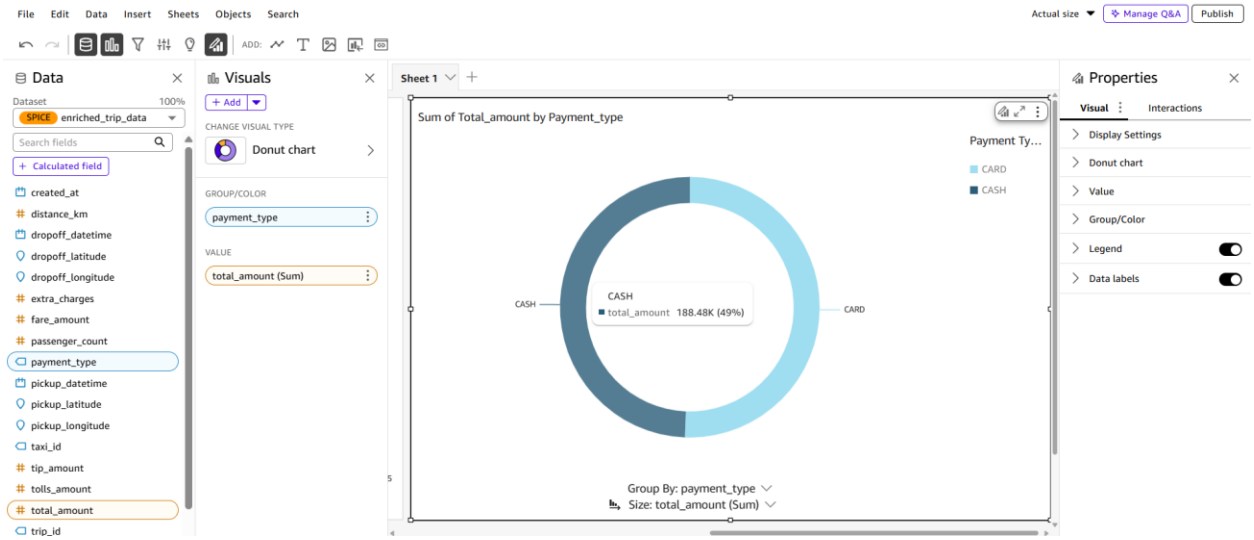
Chart Type: Line or Area Chart
X-axis: pickup_datetime (by hour/day/week)
Y-axis: SUM(total_amount)
Insight: See peak hours and revenue trends across days.



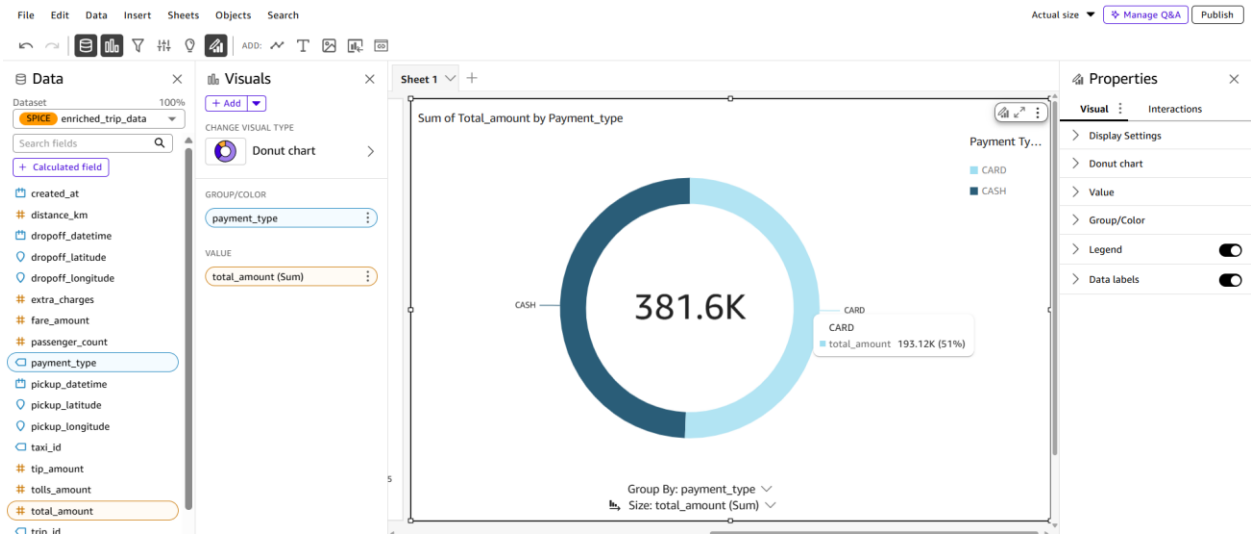


2. Payment Type Breakdown

Chart Type: Donut / Pie Chart
Data: payment_type vs SUM(total_amount)
Insight: Cash vs Card usage ratio.

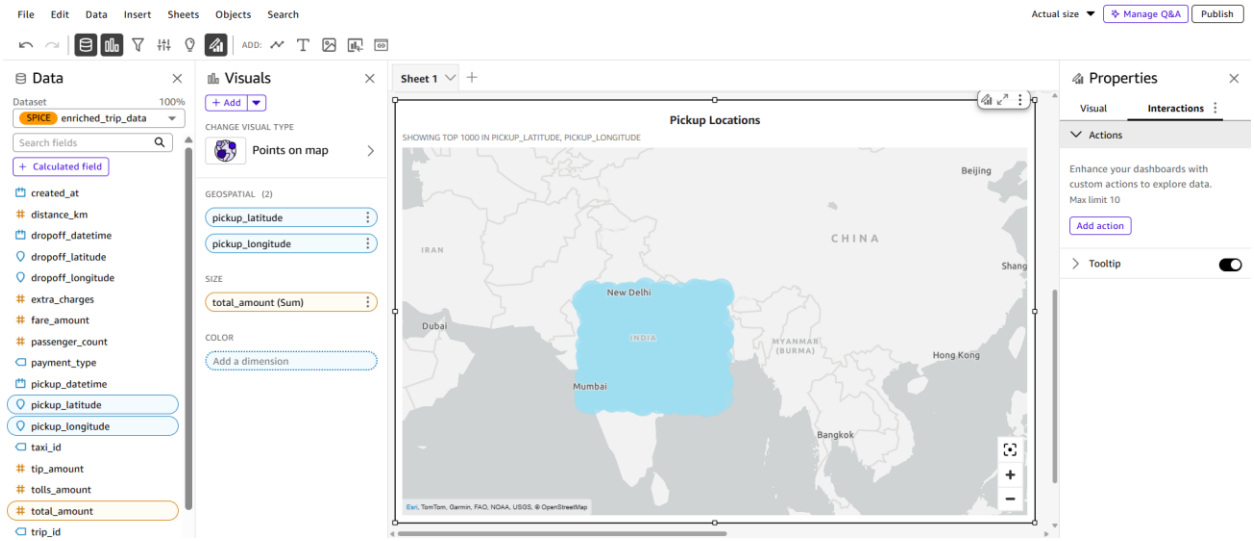


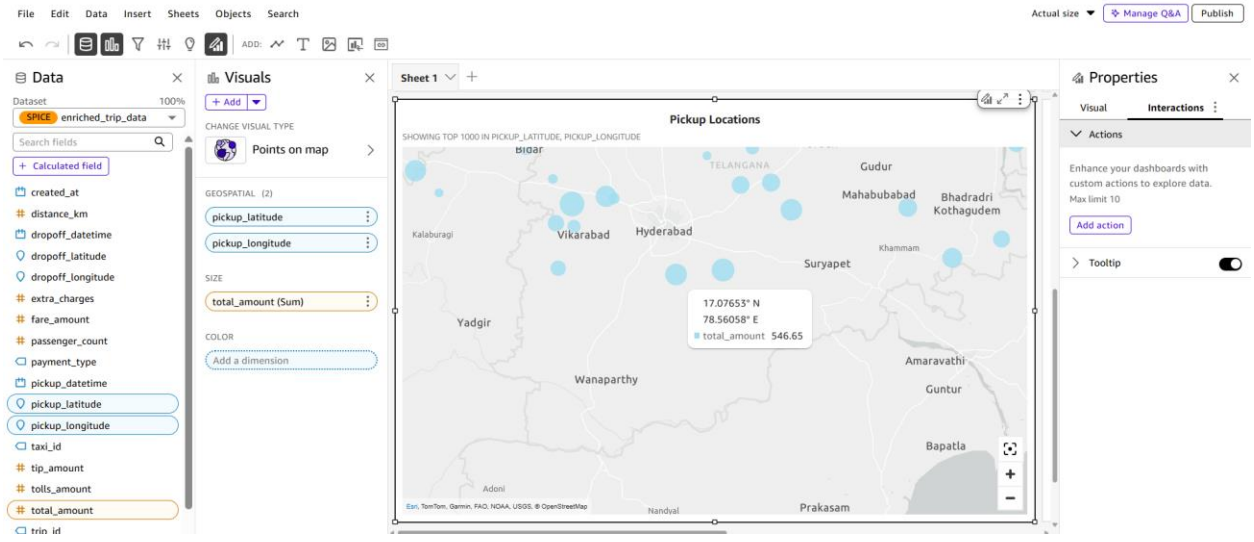
\



3. Points on Map (Pickup Locations)

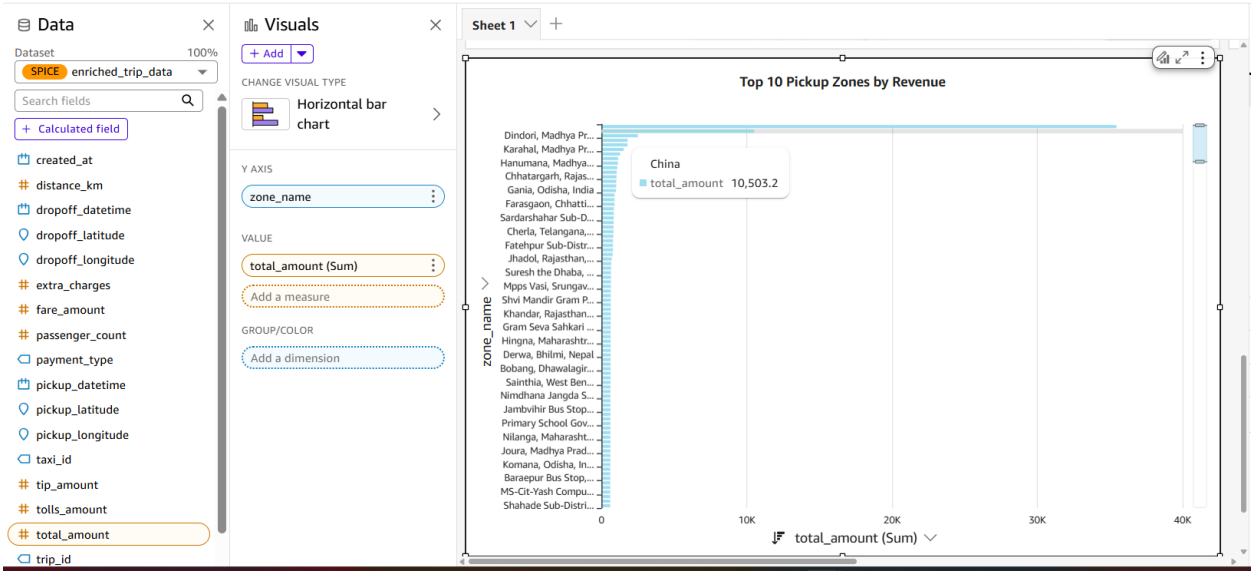
Chart Type: Map (Points on Map)
Data: pickup_latitude, pickup_longitude
Insight: Identify high-demand pickup zones.
(*You can use "zone_name" for labeling major pickup areas.*)

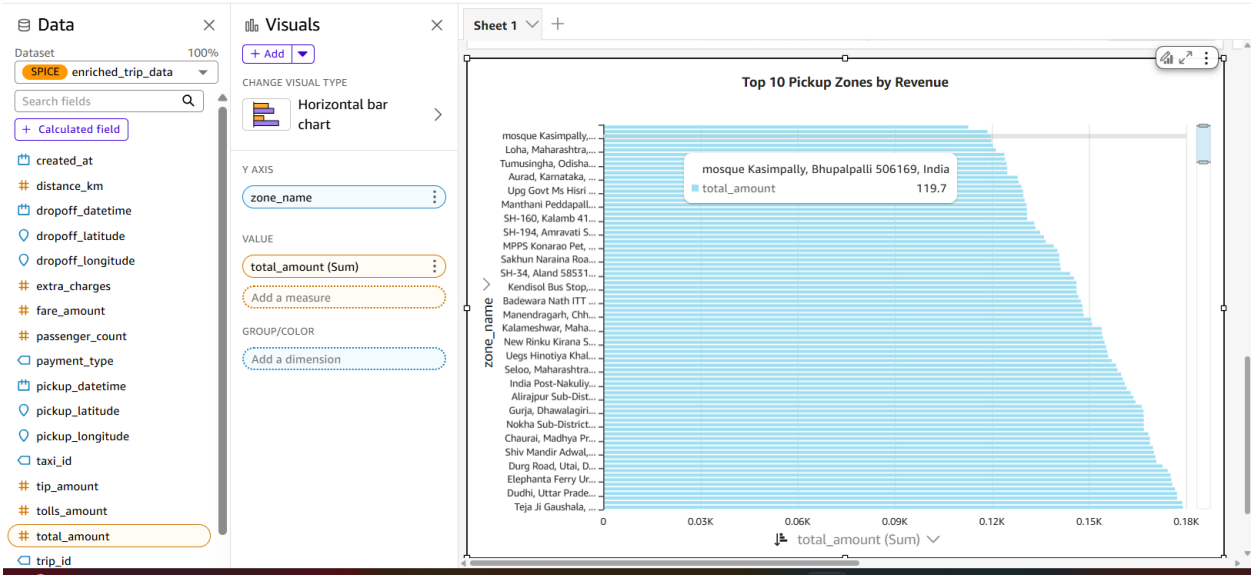




4. Top 10 Pickup Zones by Revenue

Chart Type: Horizontal Bar Chart
X-axis: SUM(total_amount)
Y-axis: zone_name (Top 10)
Insight: Which areas generate the most business.

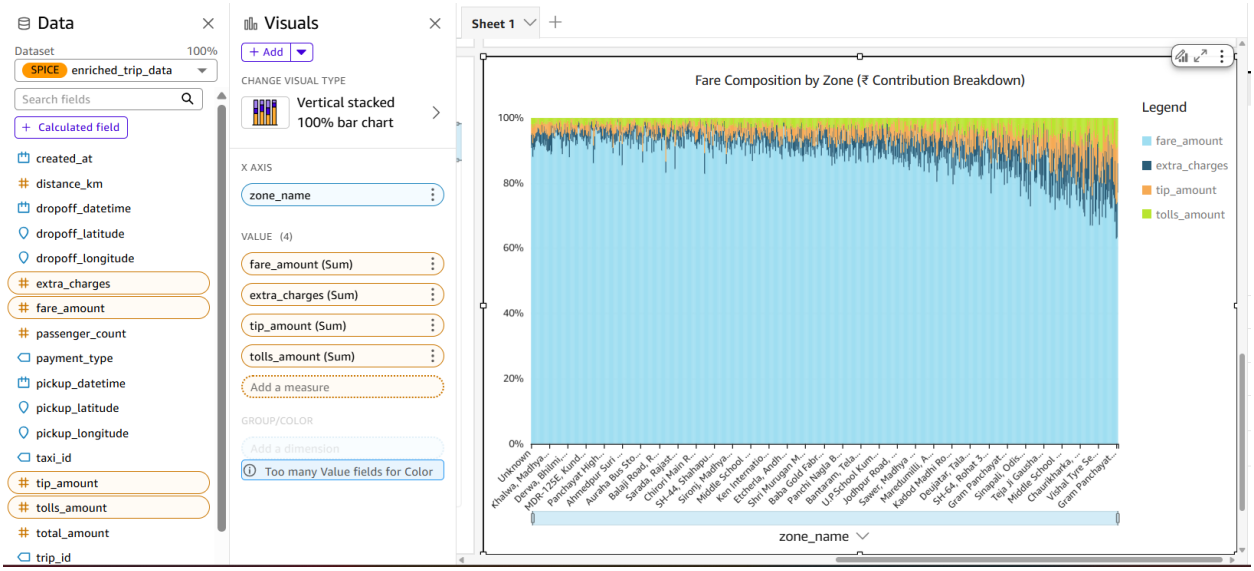




5. Fare Composition Chart (Stacked Bar / Column)

Assign Fields:

Field	Assign To	Example
zone_name *(or pickup_datetime if you want time trend)*	X-Axis	"Place of Worship, Amadalavalasa 532484, India"
fare_amount	Value	Base Fare
extra_charges	Value	Added as another measure
tip_amount	Value	Added as another measure
tolls_amount	Value	Added as another measure

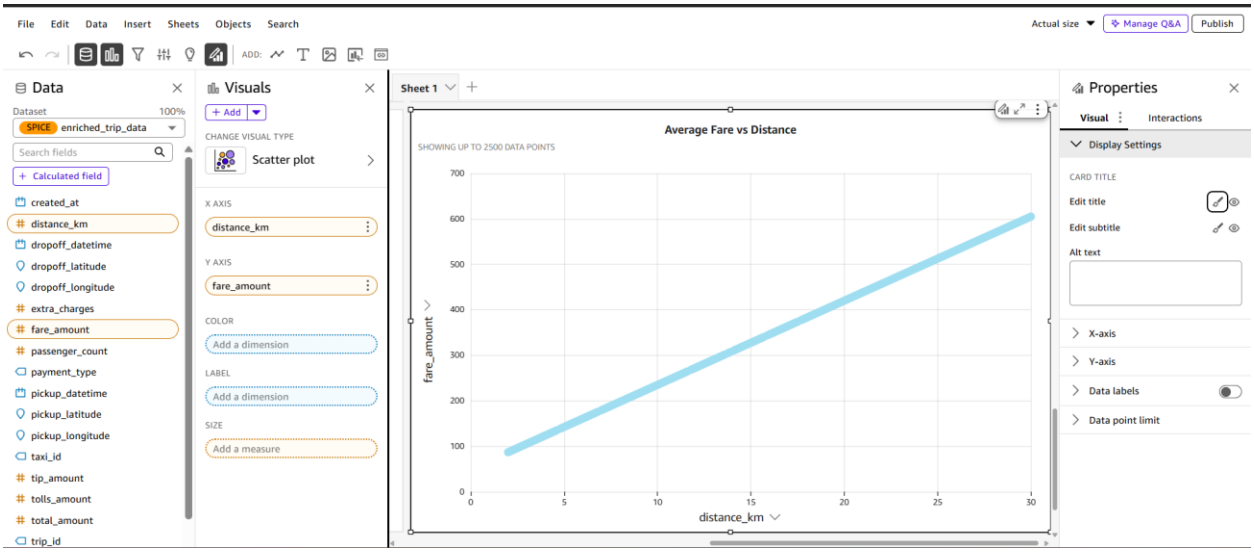


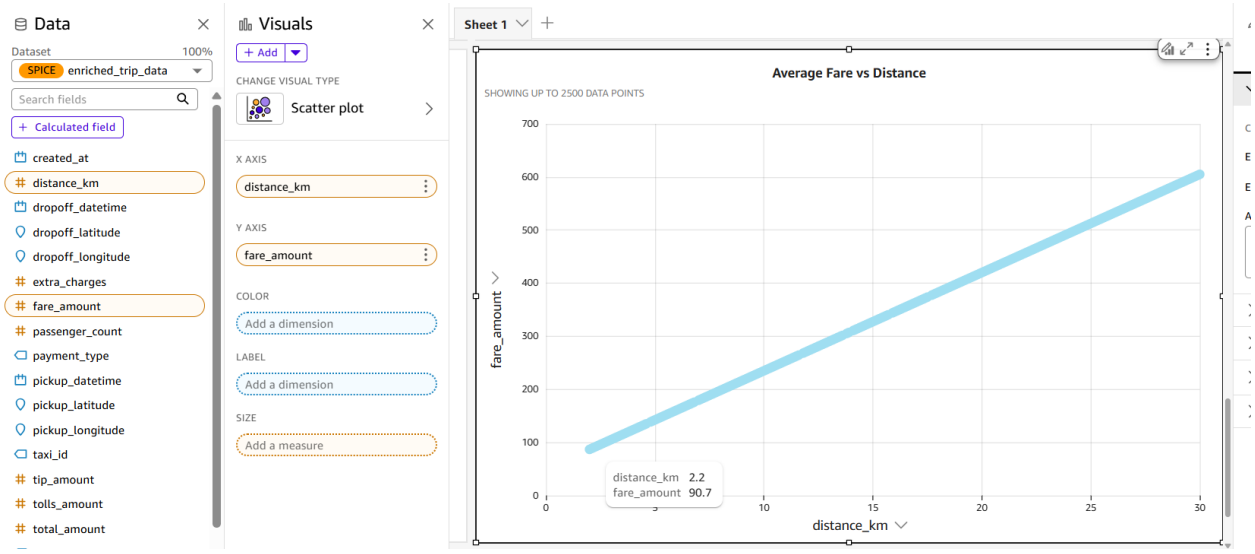
6. Passenger Distribution

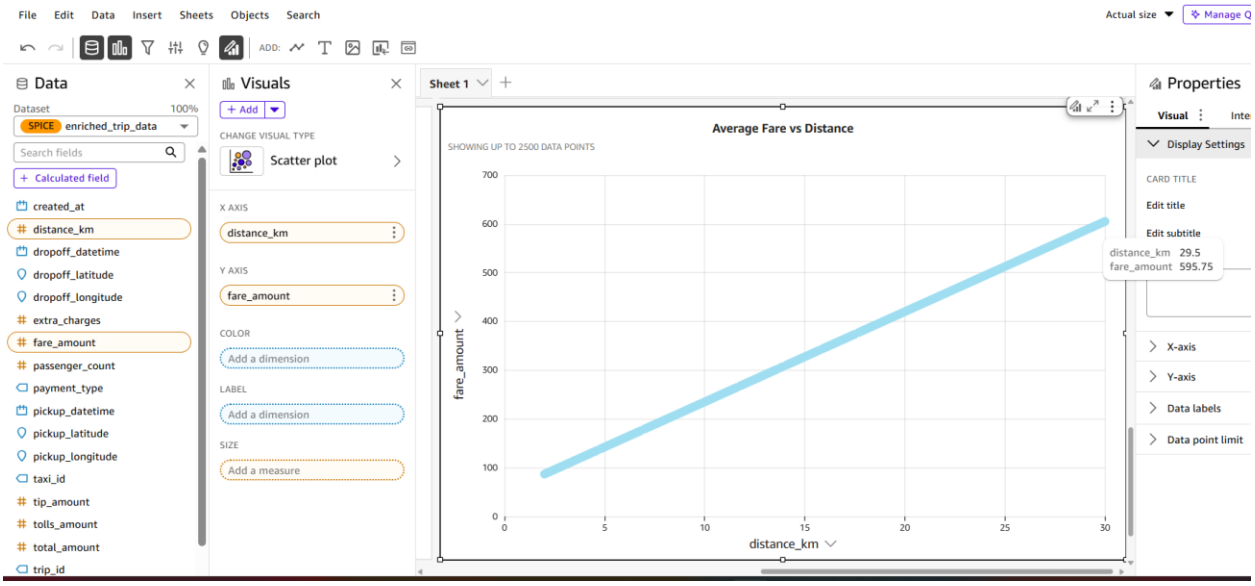
Chart Type: Bar / Histogram
X-axis: passenger_count
Y-axis: COUNT(trip_id)
Insight: Typical passenger loads per trip.

7. Average Fare vs Distance

Chart Type: Scatter Plot
X-axis: distance_km
Y-axis: fare_amount
Insight: Identify any outlier trips (too expensive or too cheap for distance).




Data
Dataset
100%
SPICE enriched_trip_data
Search fields
Calculated field
created_at
distance_km
dropoff_datetime
dropoff_latitude
dropoff_longitude
extra_charges
fare_amount
passenger_count
payment_type
pickup_datetime
pickup_latitude
pickup_longitude
taxi_id
tip_amount
tolls_amount
total_amount
Visuals
Add
CHANGE VISUAL TYPE
Scatter plot
X AXIS
distance_km
Y AXIS
fare_amount
COLOR
Add a dimension
LABEL
Add a dimension
SIZE
Add a measure
Sheet 1
Average Fare vs Distance
SHOWING UP TO 2500 DATA POINTS
fare_amount
distance_km
distance_km 2.2
fare_amount 90.7


File
Edit
Data
Insert
Sheets
Objects
Search
Actual size
Manage Q
Data
Visuals
Sheet 1
Properties
Average Fare vs Distance
SHOWING UP TO 2500 DATA POINTS
distance_km 29.5
fare_amount 595.75
Display Settings
CARD TITLE
Edit title
Edit subtitle
X-axis
Y-axis
Data labels
Data point limit
trip_id

Pipeline 2: Raw Data Archiving Pipeline

Goal:

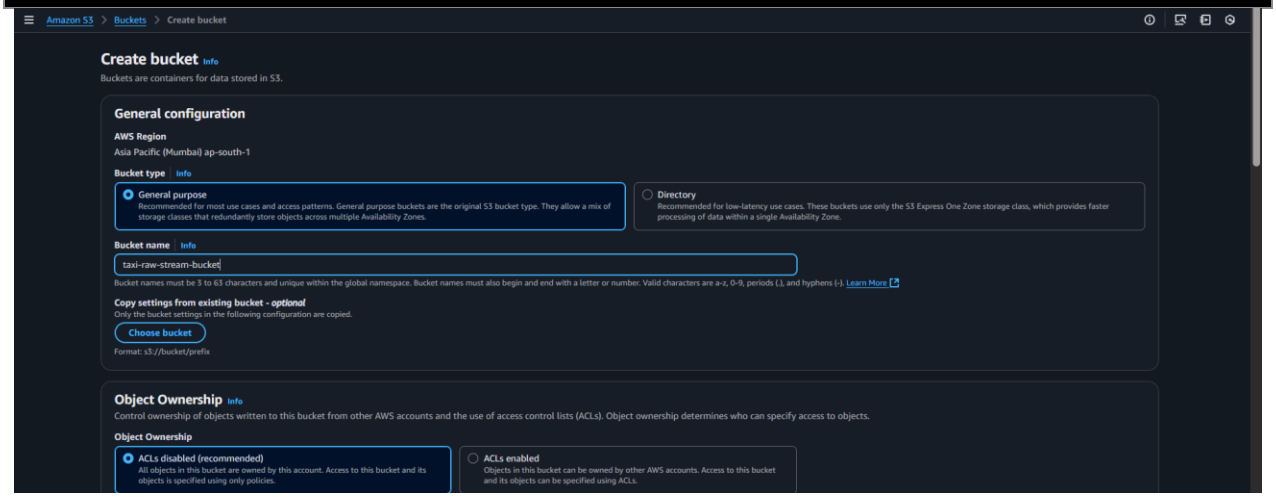
Store all raw taxi/transportation telemetry data reliably and automatically transition older data to low-cost archival tiers (S3 Glacier / Deep Archive) for long-term retention and auditing.

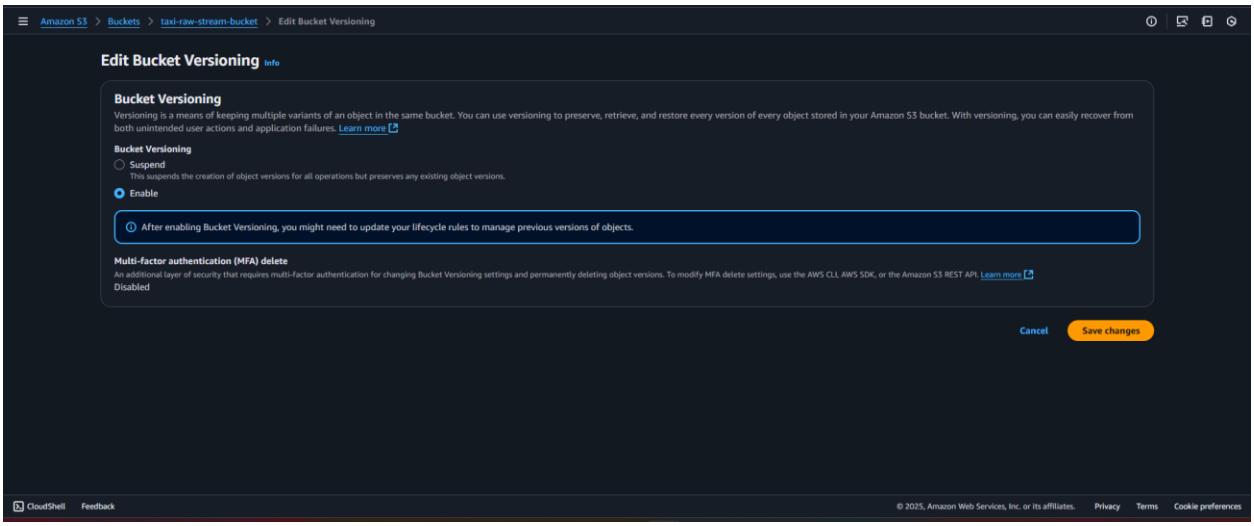
Step-by-Step Implementation:

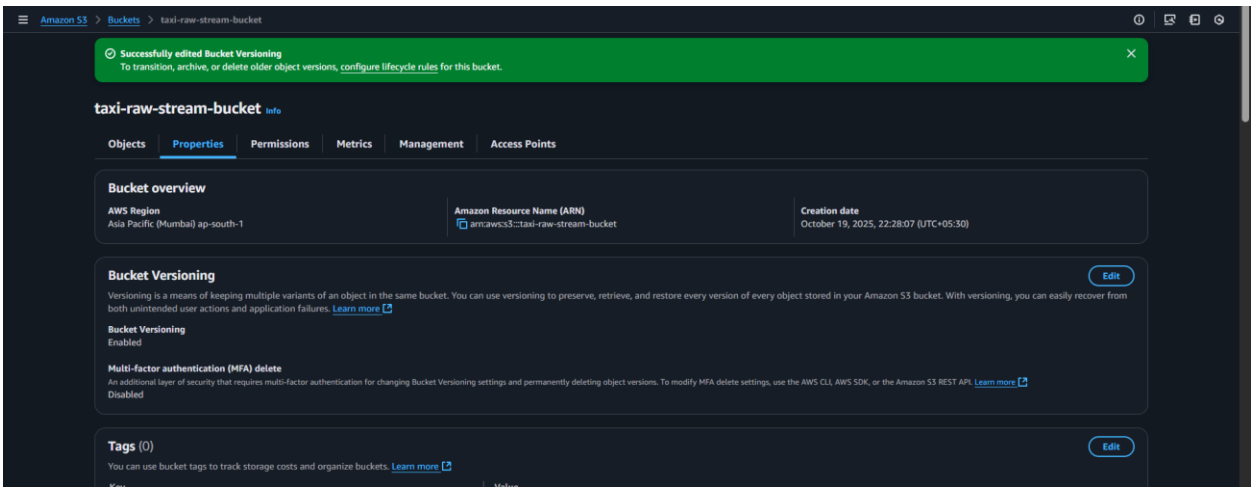
Step 1 – Create S3 Bucket for Raw Data

- Name: taxi-raw-stream-bucket
- Region: same as Kinesis Firehose delivery stream.
- Enable **versioning** for data auditability.
- Recommended structure:

```
s3://taxi-raw-stream-bucket/
    └── year=2025/
        └── month=10/
            └── day=19/
                └── hour=14/
                    └── taxi_events.json
```





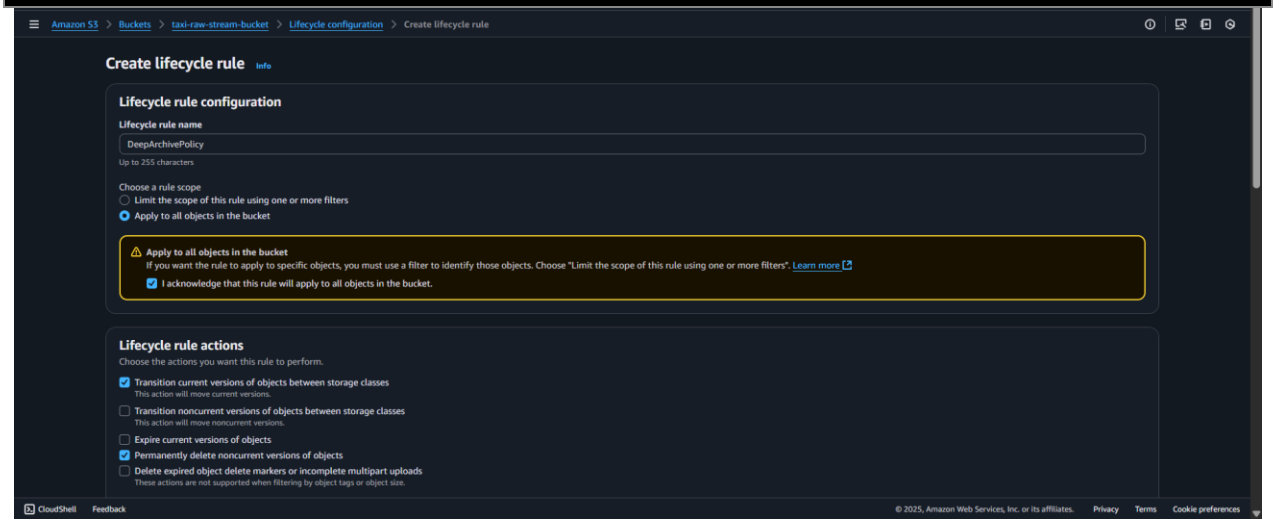


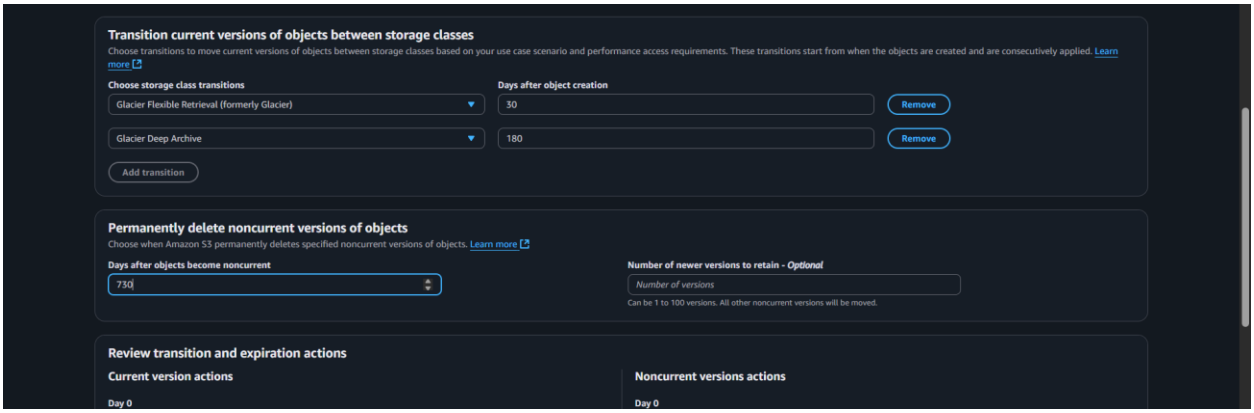
Step 2 – Set Lifecycle Configuration (Deep Archive Policy)

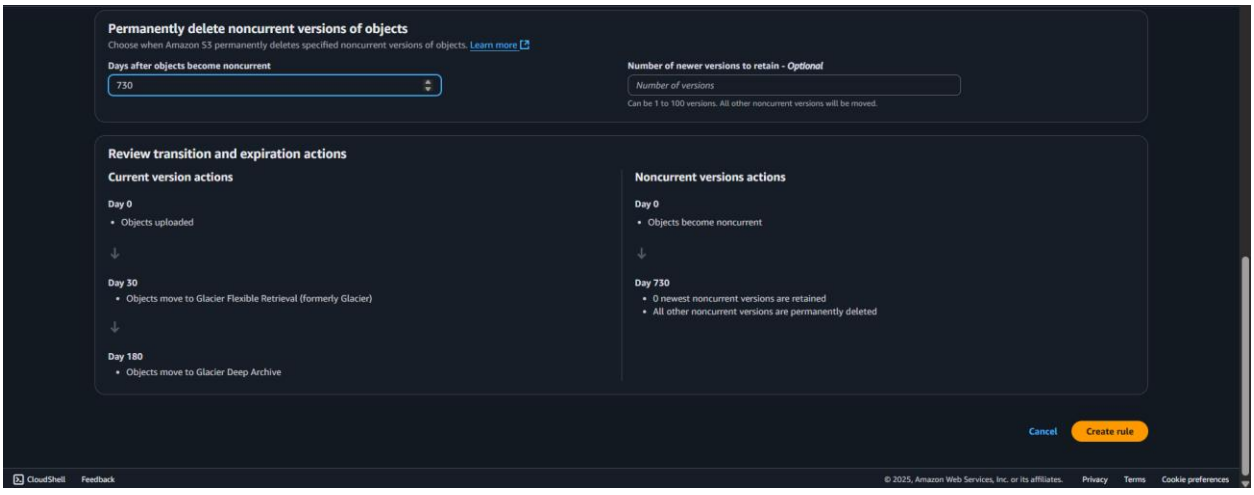
- Open S3 → Management → Lifecycle Rules → Create Rule.
- Example policy:
    - Transition objects to S3 Glacier Instant Retrieval after 30 days.
    - Transition to S3 Deep Archive after 180 days.
    - Expire (delete) objects after 730 days (optional).
- JSON policy sample:

```
{
  "Rules": [
    {
      "ID": "DeepArchivePolicy",
      "Filter": { "Prefix": "" },
      "Status": "Enabled",
      "Transitions": [
```

```
        { "Days": 30, "StorageClass": "GLACIER_IR" },
        { "Days": 180, "StorageClass": "DEEP_ARCHIVE" }
      ],
      "Expiration": { "Days": 730 }
    } ] }
```

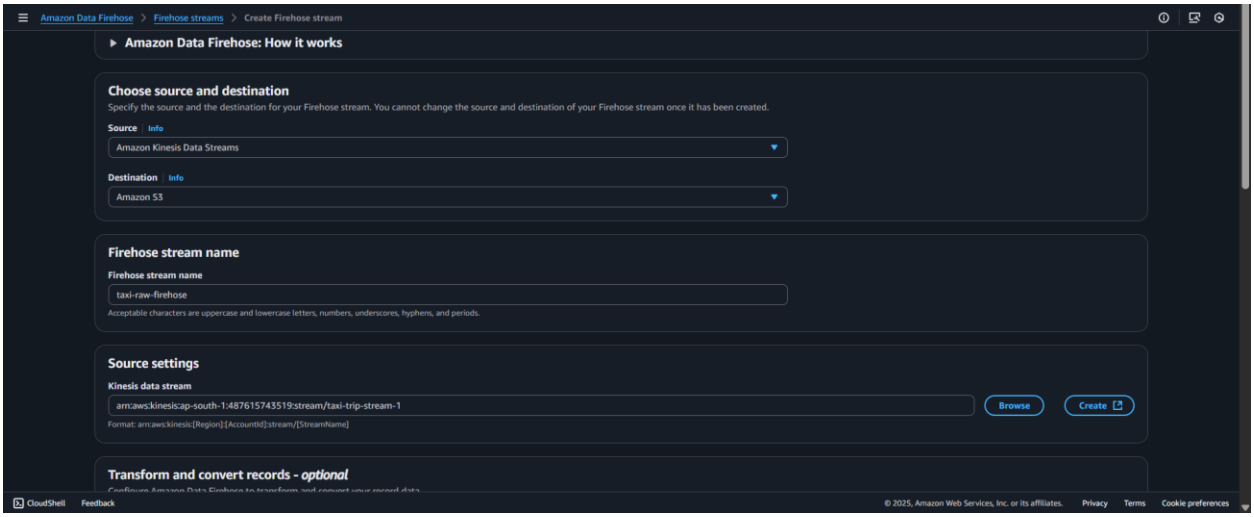


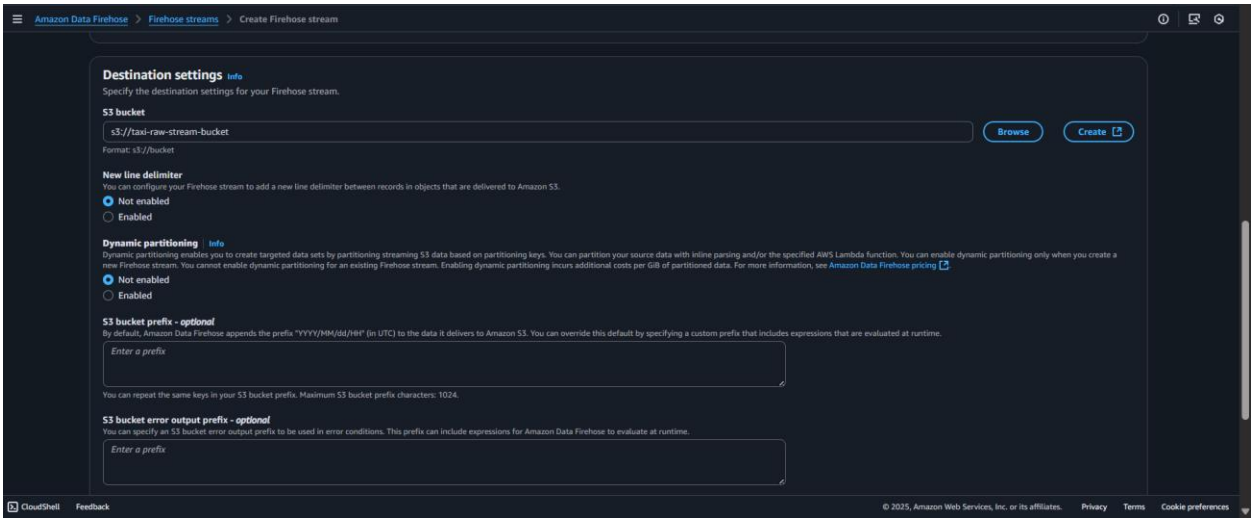


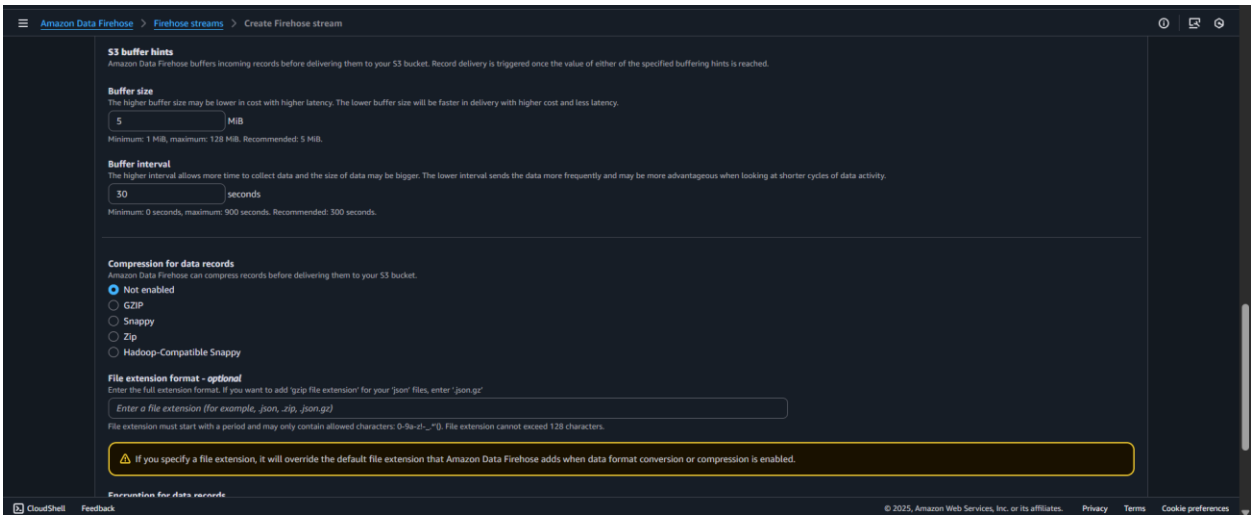


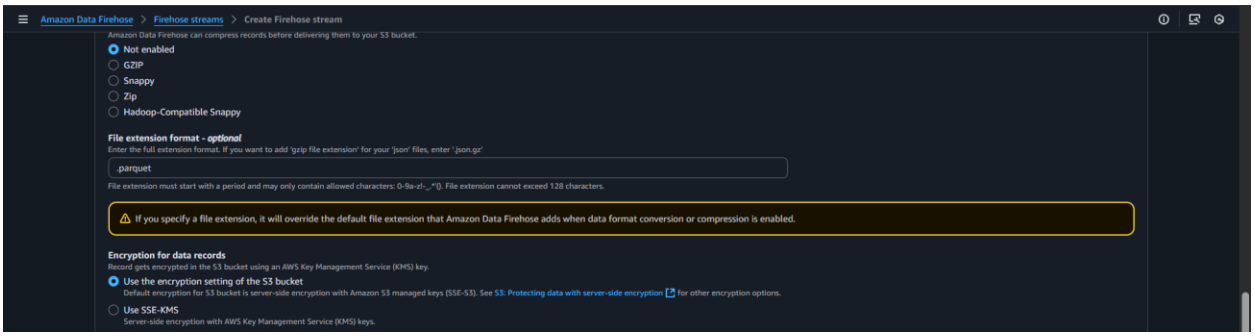
Step 3 – Create Firehose Delivery Stream

- Name: taxi-raw-firehose
- Source: Kinesis Data Stream (from Pipeline 1)
- Destination: S3 (taxi-raw-stream-bucket)
- Buffering:
    - Size: 5 MB
    - Interval: 60 seconds
- Compression: GZIP or Parquet (if you plan Athena queries)
- Enable error output prefix for debugging, e.g. errors/









Step 4 – Configure Permissions

- IAM role FirehoseToS3Role must have:

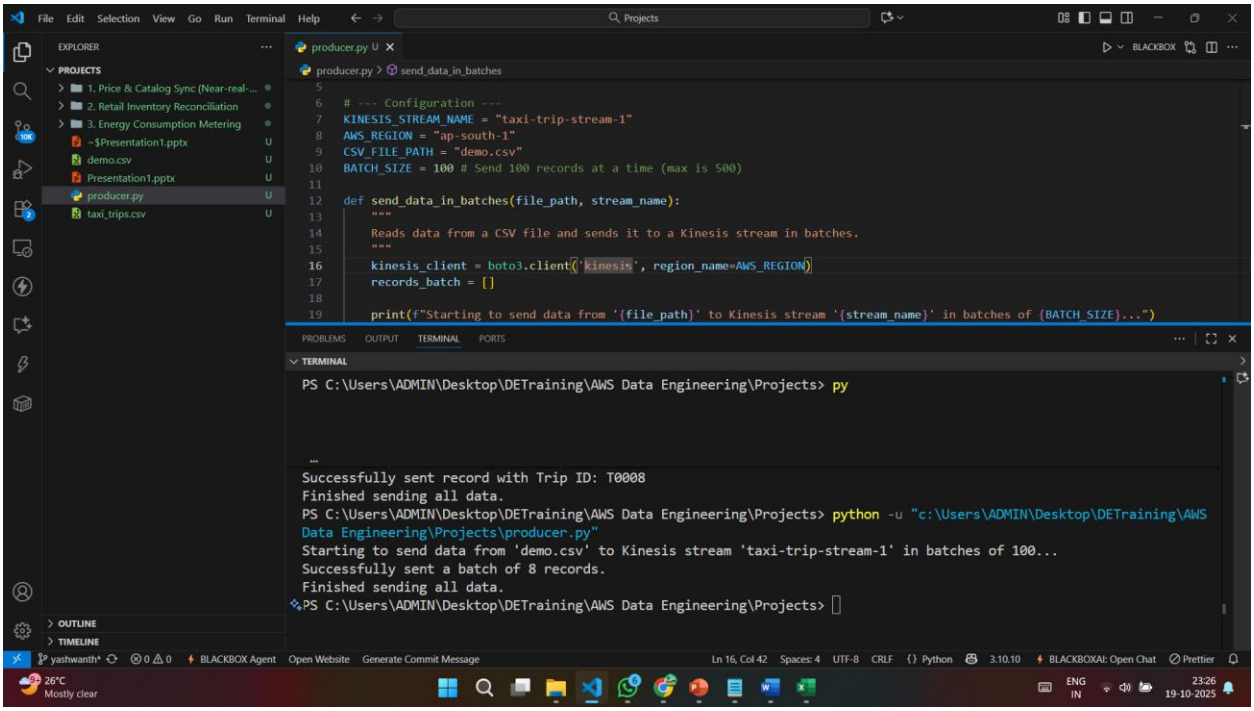
```
{
 "Effect": "Allow",
 "Action": ["s3:PutObject", "s3:PutObjectAcl"],
 "Resource": "arn:aws:s3:::taxi-raw-stream-bucket/*"
}
```

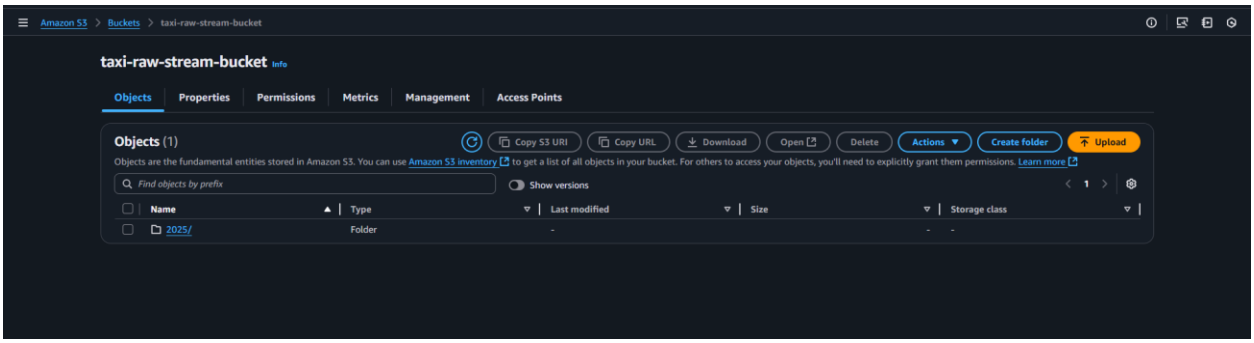
- Add Kinesis Stream read permissions too.

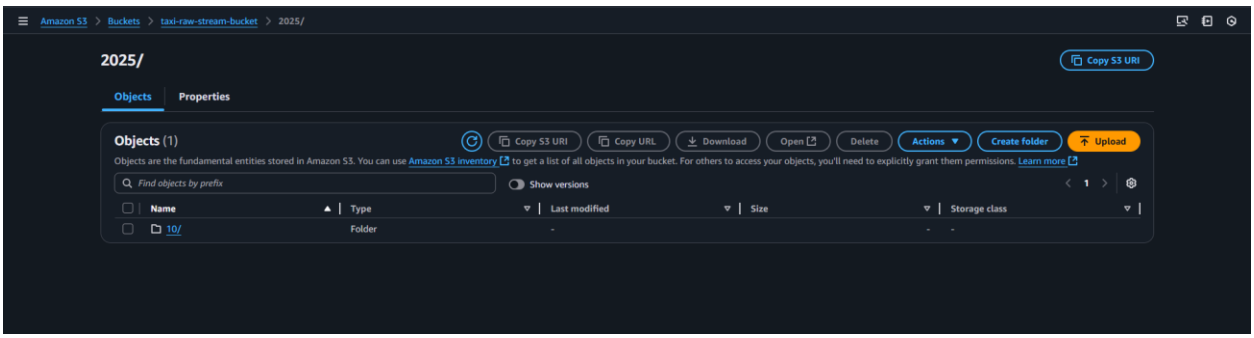
Step 5 – Validate Data Flow

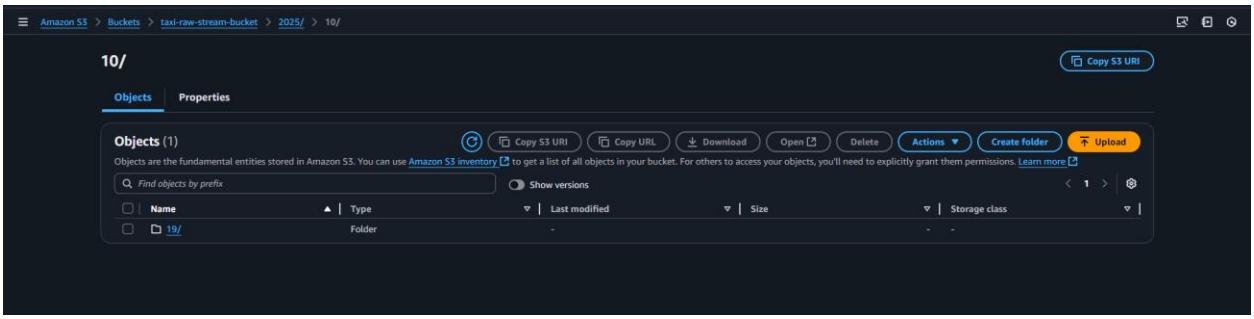
- Start producer.py (data generator).
- Verify:
    - Data flows from Kinesis Stream → Firehose → S3.

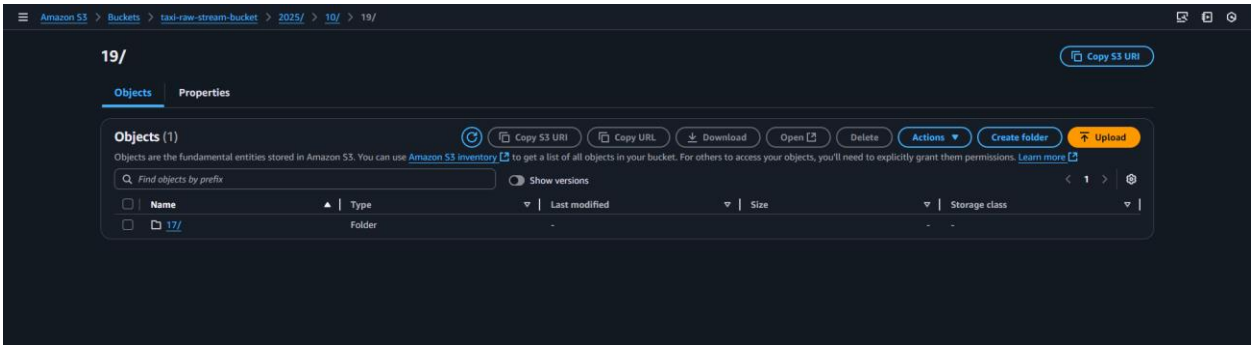
- Objects are created in partitions (date/hour folders).
- Lifecycle rules are automatically applied.

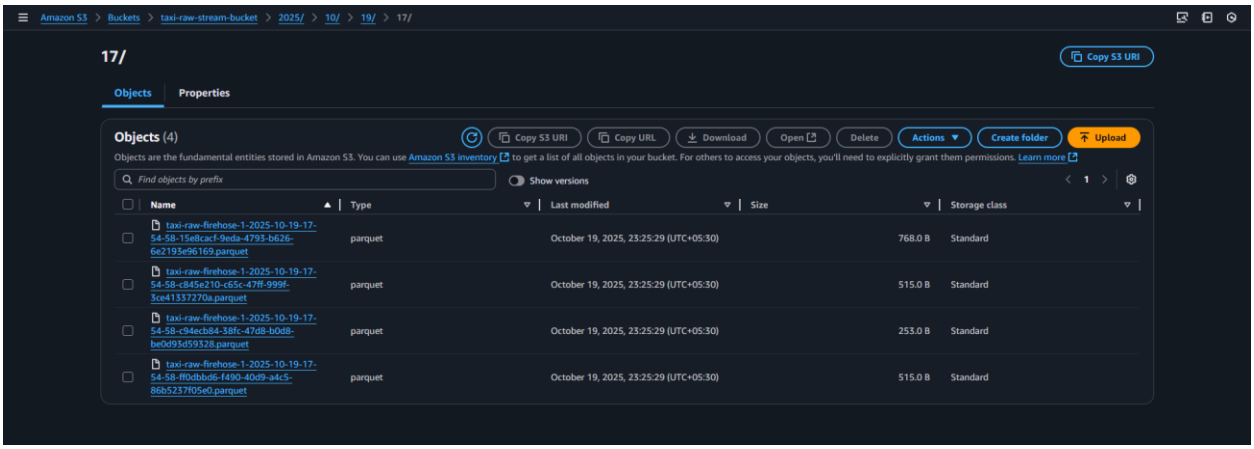












Outcome

You now have a fully automated raw data archiving pipeline that:

- Stores all incoming telemetry in S3.
- Reduces storage cost automatically over time.
- Keeps data retrievable for audits or model retraining.

Project Recap: Real-time Taxi Telemetry Streaming

Successfully built a comprehensive, serverless, real-time data pipeline on AWS to process and analyze taxi trip data. The project was divided into two main pipelines: one for real-time processing and analytics, and another for raw data archiving.

1. Data Ingestion & Simulation:

- You created a Python script (producer.py) to simulate a fleet of taxis sending trip data as JSON events.
- This script pushed data into an **Amazon Kinesis Data Stream** (taxi-trip-stream), which acted as the central, scalable entry point for all incoming real-time data.

2. Real-time Processing & Enrichment:

- An AWS Lambda function, EnrichTripDataFunction, was automatically triggered by the Kinesis stream.
- This function enriched the raw data by using **Amazon Location Service** to convert GPS coordinates into human-readable zone names.
- To create a resilient and decoupled architecture, you placed an **Amazon SQS queue** (taxi-trip-processing-queue) between your Lambda functions. The first Lambda sent the enriched data to this queue.
- You also configured a Dead-Letter Queue (DLQ) to capture any messages that failed processing, preventing data loss.

3. Data Storage & Analytics (Fan-Out Architecture):

- A second AWS Lambda function, ProcessAndStoreFunction, was triggered by the SQS queue.
- This function performed final processing, including calculating the fare for each trip.
- It then fanned out the data to two distinct destinations, separating the operational and analytical data paths:
    - **Operational Store:** The final, complete trip record was written to an **Amazon DynamoDB** table (TaxiBilling) to enable fast, low-latency access for applications like micro-billing.
    - **Analytical Warehouse:** The processed data was also sent to a **Kinesis Data Firehose** stream (Lambda-Redshift-Firehouse), which automatically batched and loaded the data into an **Amazon Redshift Serverless** data warehouse.

4. Raw Data Archiving (Data Lake):

- In parallel to the processing pipeline, you configured a second Kinesis Data Firehose stream (taxi-raw-firehose) subscribed directly to the main Kinesis Data Stream.
- This stream's purpose was to archive all raw, unaltered events into a dedicated S3 bucket (taxi-raw-stream-bucket), creating a raw data lake.
- You implemented an S3 Lifecycle Policy to automatically transition this raw data to cheaper storage tiers like S3 Glacier and Deep Archive over time, optimizing storage costs.

5. Data Visualization:

- Finally, you connected **Amazon QuickSight** to your Redshift data warehouse.
- You built several interactive dashboards to visualize the data, creating charts for:
    - Revenue trends over time
    - Payment type distribution (Cash vs. Card)
    - Geographical heatmaps of pickup locations
    - Top 10 revenue-generating zones
    - Fare composition breakdowns
    - Passenger count distribution
    - The relationship between trip distance and fare amount

In summary, you built an end-to-end, event-driven pipeline using a serverless architecture that ingests, processes, stores, and visualizes real-time data, showcasing a robust separation of operational and analytical concerns.