

Project Title: Real-time Taxi Telemetry Streaming

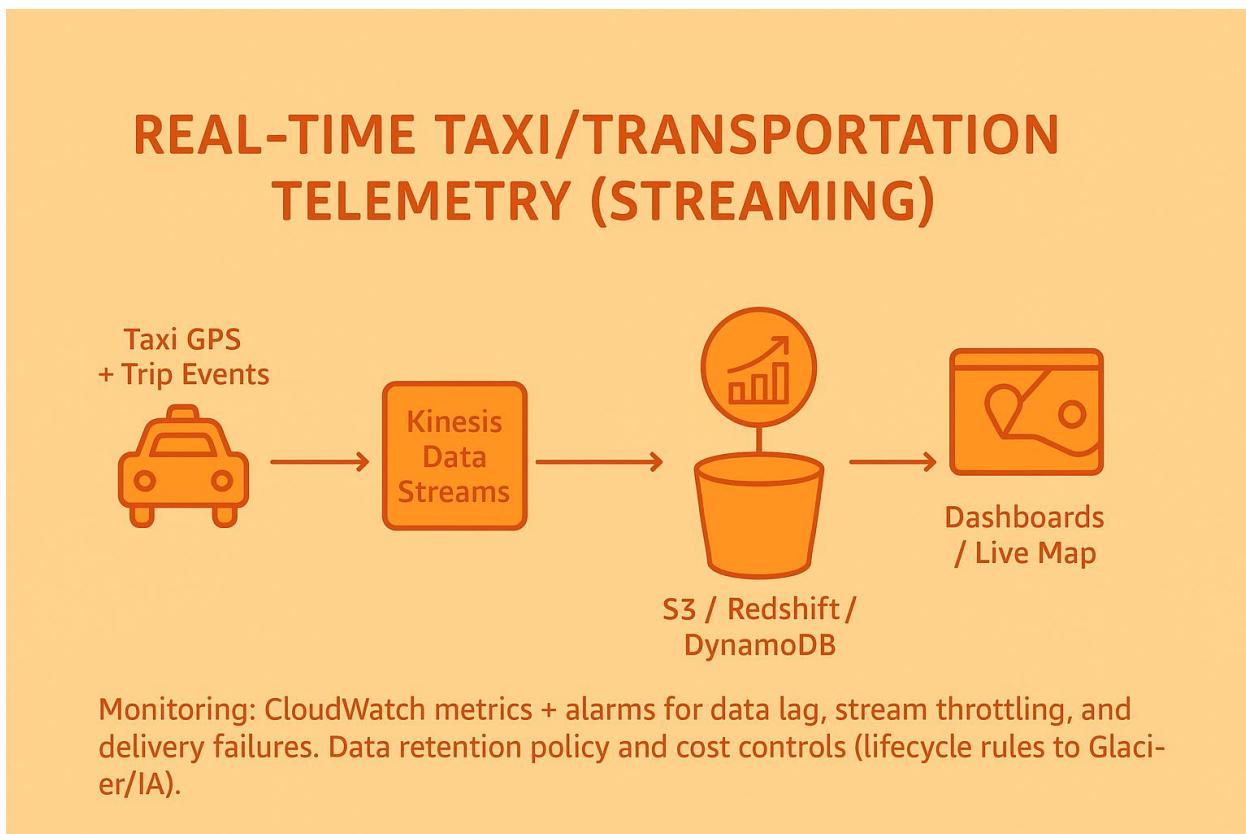
Name: Yashwanth Aravanti

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:

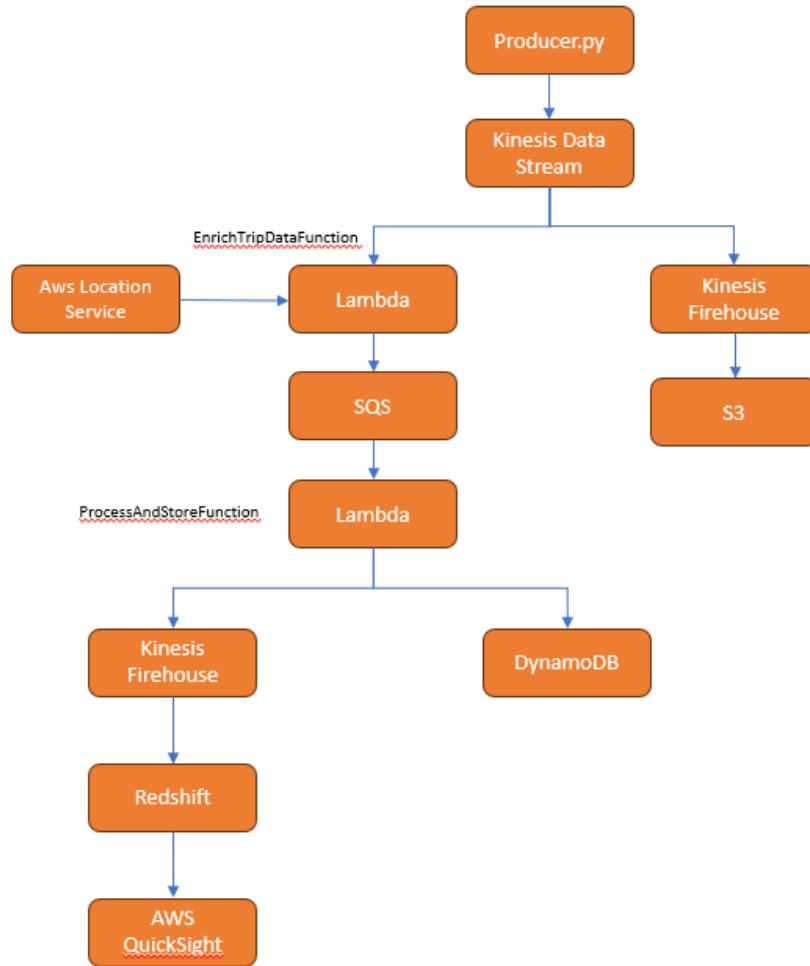
An AWS Lambda function (EnrichTripDataFunction) is triggered for each batch of records arriving in the Kinesis Data Stream. It performs real-time data enrichment by calling AWS Location Service to convert the trip's GPS coordinates into a human-readable zone name. It then asynchronously invokes a second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

The screenshot shows a Microsoft Excel spreadsheet titled "taxi_trips_1000 - Excel". The spreadsheet contains a single sheet with data from row 1 to 200. The columns are labeled A through Z, and the first few rows provide a detailed breakdown of taxi trip data. The data includes columns for trip_id, taxi_id, pickup_datetime, latitude, longitude, pickup_lat, pickup_long, drop_lat, drop_long, distance_km, passenger_count, extra_charges, tip_amount, tolls_amount, and payment_type. The payment_type column shows various modes of transport such as CARD, CASH, and other less common options. The Excel interface includes standard ribbon tabs like File, Home, Insert, Draw, Page Layout, Formulas, Data, Review, View, Developer, and Help. The status bar at the bottom shows the date as 20-10-2025 and the time as 01:52.

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)))

                records_batch.append(json.dumps(row))

                if len(records_batch) == BATCH_SIZE:
                    kinesis_client.put_records(StreamName=stream_name, Records=records_batch)
                    records_batch = []

    except Exception as e:
        print(f"An error occurred while reading the CSV file: {e}")

    finally:
        if records_batch:
            kinesis_client.put_records(StreamName=stream_name, Records=records_batch)
```

```

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)

```

The screenshot shows a Python development environment with the following details:

- EXPLORER**: Shows a project structure with files like `producer.py`, `ex.py`, and `demo.csv`.
- PROJECTS**: Lists several projects including "Price & Catalog Sync (Near-real-time)".
- CODE**: The `producer.py` file is open, containing code to send data in batches to a Kinesis stream. The code uses `boto3` to interact with AWS Kinesis.

```
producer.py > send_data_in_batches

1 import boto3
2 import csv
3 import json
4 import time
5 import os
6
7 # --- Configuration ---
8 KINESIS_STREAM_NAME = os.environ.get('KINESIS_STREAM_NAME', 'taxi-trip-stream-1')
9 AWS_REGION = os.environ.get('AWS_REGION', 'ap-south-1')
10 CSV_FILE_PATH = os.environ.get('CSV_FILE_PATH', 'demo.csv')
11 BATCH_SIZE = int(os.environ.get('BATCH_SIZE', 100)) # Send records in batches
12
13 def send_data_in_batches(file_path, stream_name):
14     """
15     Reads CSV, converts numeric fields, and sends records to Kinesis in batches.
16     """
17     kinesis_client = boto3.client('kinesis', region_name=AWS_REGION)
18     records_batch = []
19
20     print(f"Sending data from '{file_path}' to Kinesis stream '{stream_name}'...")
21
22     try:
23         with open(file_path, 'r', encoding='utf-8') as csv_file:
24             csv_reader = csv.DictReader(csv_file)
25
26             for row in csv_reader:
27                 # Ensure pickup_datetime exists
28                 if 'pickup_datetime' not in row or not row['pickup_datetime'].strip():
29                     print(f"Skipping trip_id {row.get('trip_id', 'N/A')}: missing pickup_datetime")
30                     continue
31
32                 # Convert numeric fields
33                 for col in ['latitude', 'longitude', 'pickup_lat', 'pickup_long', 'drop_lat', 'drop_long', 'distance_km']:
34                     row[col] = float(row.get(col, 0))
35
36     except Exception as e:
37         print(f"An error occurred: {e}")
38
39     kinesis_client.put_records(StreamName=stream_name, Records=records_batch)
```

- STATUS**: Shows the status bar with "Ln 67, Col 29" and "BLACKBOX Agent".

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.

Create data stream

Data stream configuration

Data stream name: taxi-trip-stream

Capacity mode

- On-demand
- Provisioned

Provisioned shards

The total capacity of a stream is the sum of the capacities of its shards. Enter number of provisioned shards to see total data stream capacity.

1 **Shard estimator**

Total data stream capacity

Shard capacity is determined by the number of provisioned shards. Each shard ingests up to 1 MiB/second and 1,000 records/second and emits up to 2 MiB/second. If writes and reads exceed capacity, the application will receive throttles.

Write capacity

Maximum: 1 MiB/second and 1,000 records/second

Read capacity

Maximum: 2 MiB/second

Provisioned mode has a fixed-throughput pricing model. See Kinesis pricing for Provisioned mode.

taxi-trip-stream-1

Data stream summary

Status: Active

Capacity mode: On-demand

ARN: arn:aws:kinesis:ap-south-1:487615745519:stream/taxi-trip-stream-1

Creation time: October 19, 2025 at 22:07 GMT+5:30

Applications **Monitoring** **Configuration** **Enhanced fan-out (0)** **Data viewer** **Data analytics - new** **Data stream sharing** **EventBridge Pipes**

Producers

- Amazon Kinesis Agent
- AWS SDK
- Amazon Kinesis Producer Library (KPL)

Consumers

- Managed Apache Flink
- Amazon Data Firehose
- Amazon Kinesis Client Library (KCL)

Name	Status	Capacity mode	Provisioned shards	Sharing policy	Data retention period	Encryption	Consumers with
taxi-trip-stream-1	Active	On-demand	-	No	1 day	Disabled	0

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.

The image shows two screenshots of the AWS S3 interface. The top screenshot is the 'Create bucket' wizard, where a bucket named 'taxi-trip-staging-yashwanth' is being created in the 'Asia Pacific (Mumbai) ap-south-1' region. It's set to be a 'General purpose' bucket. The bottom screenshot shows the 'Amazon S3 Buckets' dashboard, displaying the newly created 'taxi-trip-staging-yashwanth' bucket under the 'General purpose buckets' tab. The bucket details show it was created on October 19, 2025, at 20:44:19 (UTC+05:30).

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.

Create Firehose stream

Choose source and destination
Specify the source and the destination for your Firehose stream. You cannot change the source and destination of your Firehose stream once it has been created.

Source: Direct PUT
Destination: Amazon Redshift

Firehose stream name
Firehose stream name: Lambda-Redshift-Firehouse
Acceptable characters are uppercase and lowercase letters, numbers, underscores, hyphens, and periods.

Transform records - optional
Configure Amazon Data Firehose to transform and convert your record data.
Transform source records with AWS Lambda

Turn on data transformation

Serverless workgroup
Specify the Amazon Redshift Serverless workgroup to which S3 bucket data is copied.
default-workgroup

Database
Specify the Amazon Redshift Serverless database to where the data is copied.
dev

Authentication
Specify how you want to configure the authentication to access your destination.

- Use username and password
- Use AWS Secrets Manager - new

To learn more, see AWS Secrets Manager pricing

User name
Specify an Amazon Redshift Serverless user with permissions to access the Amazon Redshift Serverless workgroup.
admin

Password
Specify the password for the user who has permissions to access the workgroup.
Yashwanth4905

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 '@'.

Show password

Table
Specify the Amazon Redshift Serverless table to where the data is copied.
enriched_trip_data

Columns - optional

Buffer hints, compression and encryption
The fields below are pre-populated with the recommended default values for S3. Pricing may vary depending on storage and request costs.

S3 buffer hints
Amazon Data Firehose buffers incoming records before delivering them to your S3 bucket. Record delivery is triggered once the value of either of the specified buffering hints is reached.

Buffer size
The higher buffer size may be lower in cost with higher latency. The lower buffer size will be faster in delivery with higher cost and less latency.
5 MiB

Minimum: 1 MiB, maximum: 128 MiB. Recommended: 5 MiB.

Buffer interval
The higher interval allows more time to collect data and the size of data may be bigger. The lower interval sends the data more frequently and may be more advantageous when looking at shorter cycles of data activity.
60 seconds

Minimum: 0 seconds, maximum: 900 seconds. Recommended: 300 seconds.

S3 compression and encryption
Amazon Data Firehose can compress records before delivering them to your S3 bucket. Records can also be encrypted in the S3 bucket using an AWS Key Management Service (KMS) key.

Compression for data records
Amazon Data Firehose can compress records before delivering them to your S3 bucket.

- Not enabled
- GZIP

Encryption for data records

```
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.

The screenshot shows the AWS DynamoDB Tables page. On the left, there's a sidebar with 'DynamoDB' selected under 'Tables'. The main area shows a table named 'TaxiBilling' with one item. The table details are as follows:

Name	Status	Partition key	Sort key	Indexes	Replication Regions	Deletion protection	Favorite	Read capacity mode	Write capacity mode	Total size	Table class
TaxiBilling	Active	trip_id (\$)	pickup_datetime (\$)	0	0	Off		On-demand	On-demand	0 bytes	Standard

A green banner at the top says 'The TaxiBilling table was created successfully.'

The screenshot shows the 'TaxiBilling' table settings page. The left sidebar has 'DynamoDB' selected under 'Tables'. The main area has tabs for 'Settings', 'Indexes', 'Monitor', 'Global tables', 'Backups', 'Exports and streams', and 'Permissions'. Under 'General Information', the table details are:

- Partition key:** trip_id (String)
- Sort key:** pickup_datetime (String)
- Point-in-time recovery (PITR):** Off
- Capacity mode:** On-demand
- Item count:** 12
- Table status:** Active
- Table size:** 3.5 kilobytes

Under 'Read/write capacity', it says 'The read/write capacity mode controls how you are charged for read and write throughput and how you manage capacity.' The 'Capacity mode' is set to 'On-demand'.

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.

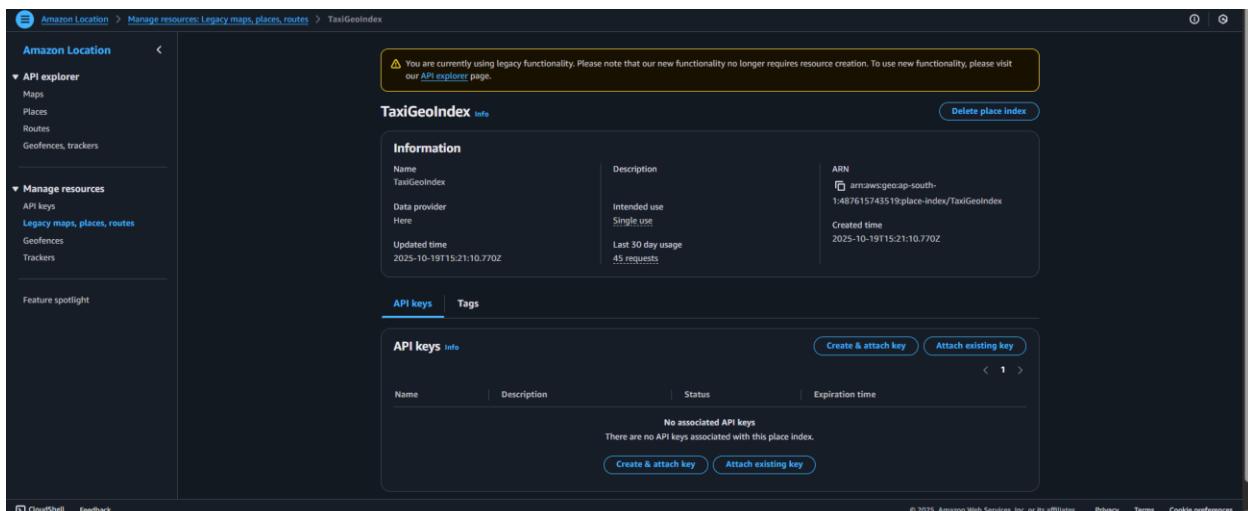
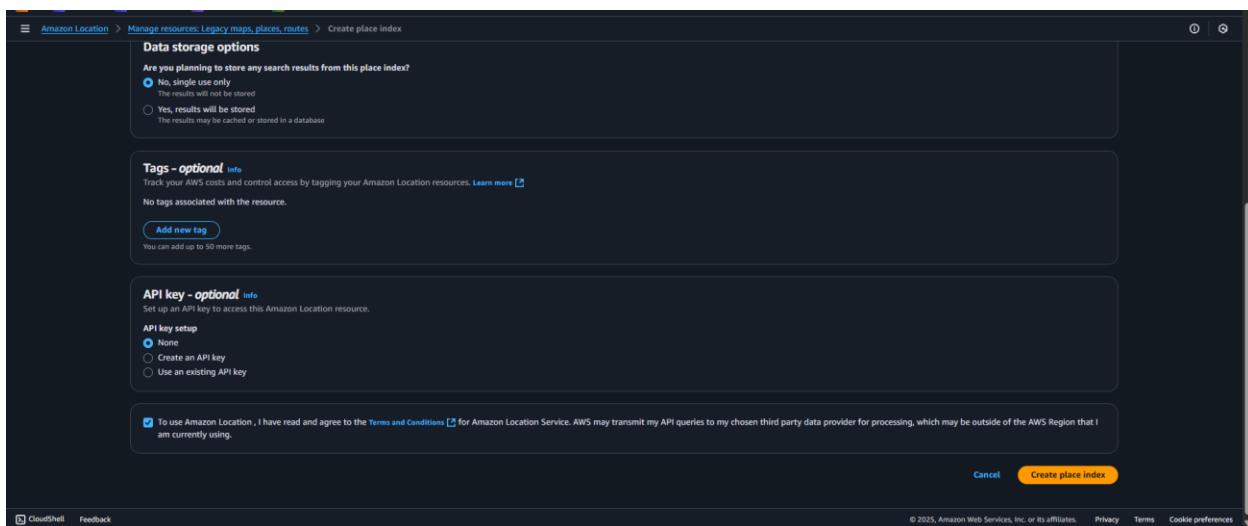
Purpose:

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

The screenshot shows the Amazon Location Service console under the 'Places' tab. A prominent callout box titled 'Migrate to new APIs to access improved functionality' contains three sections: 'Add permissions' (with a key icon), 'Select new API' (with an SDK icon), and 'Get the SDK' (with a laptop and smartphone icon). Below the callout, a table titled 'Place indexes' shows one entry: 'No place indexes'. At the bottom of the page is a 'Create place index' button.

This screenshot shows the 'Create place index' wizard. Step 1, 'Name and description', has 'TaxiGeoIndex' entered in the 'Name' field. Step 2, 'Data provider', shows 'HERE' selected, with a detailed description of HERE's service. Step 3, 'Data storage options', shows 'No, single use only' selected. The bottom of the screen includes standard AWS navigation links like CloudShell, Feedback, and a cookie preference banner.



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.

taxi-trip-dlq

Details Info

Name taxi-trip-dlq

Type Standard

Encryption Amazon SQS key (SSE-SQS)

URL https://sqs.ap-south-1.amazonaws.com/487615743519/taxi-trip-dlq

Dead-letter queue -

ARN copied. [arn:aws:sqscap-south-1:487615743519:taxis-trip-dlq](#)

Access policy Info

Define who can access your queue.

```
{
  "Version": "2012-10-17",
  "Id": "default_policy_ID",
  "Statement": [
    {
      "Sid": "owner_statement",
      "Effect": "Allow",
      "Principal": "owner",
      "Action": "SQS:SendMessage",
      "Resource": "arn:aws:sqscap-south-1:487615743519:taxis-trip-dlq",
      "Condition": {}
    }
  ]
}
```

Queue policies **Monitoring** **SNS subscriptions** **Lambda triggers** **EventBridge Pipes** **Dead-letter queue** **Tagging** **Encryption** **Dead-letter queue redrive tasks**

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

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

Maximum receives

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	Value - optional
<input type="text" value="Enter key"/>	<input type="text" value="Enter value"/>

Remove

The screenshot shows the AWS SQS console with the title "Amazon SQS > Queues". There are two queues listed:

Name	Type	Created	Messages available	Messages in flight	Encryption	Content-based deduplication
taxi-trip-dlq	Standard	2025-10-19T23:16+05:50	0	0	Amazon SQS key (SSE-SQS)	-
taxi-trip-processing-queue	Standard	2025-10-19T23:17+05:50	0	0	Amazon SQS key (SSE-SQS)	-

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('sns')

# 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'
        except Exception as e:
            print(f"Error processing record: {e}")

```

```

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

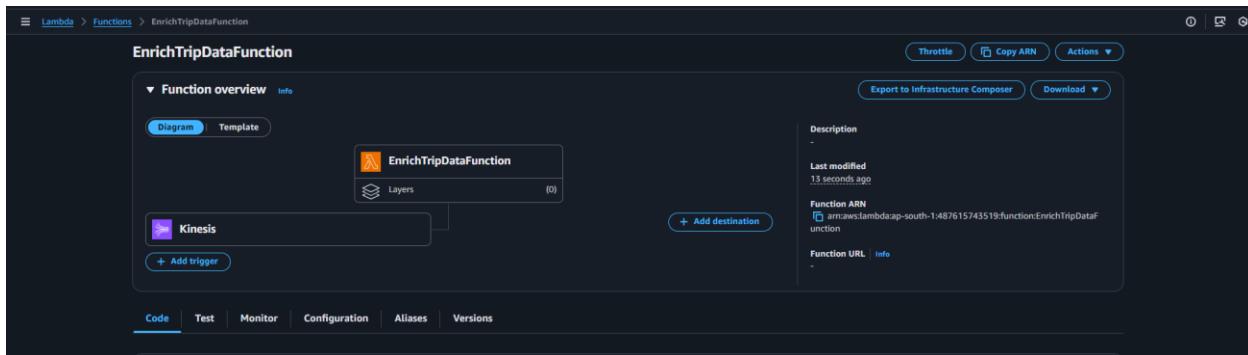
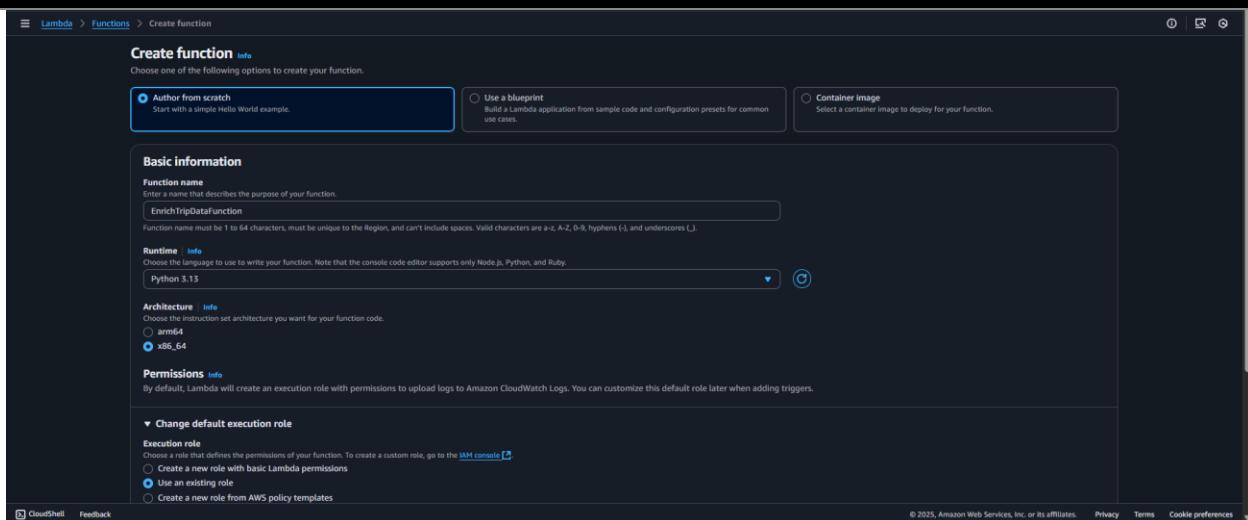
# 4. Send the enriched data to the SQS queue
# The message body must be a string
message_body = json.dumps(trip_data)

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.")
}

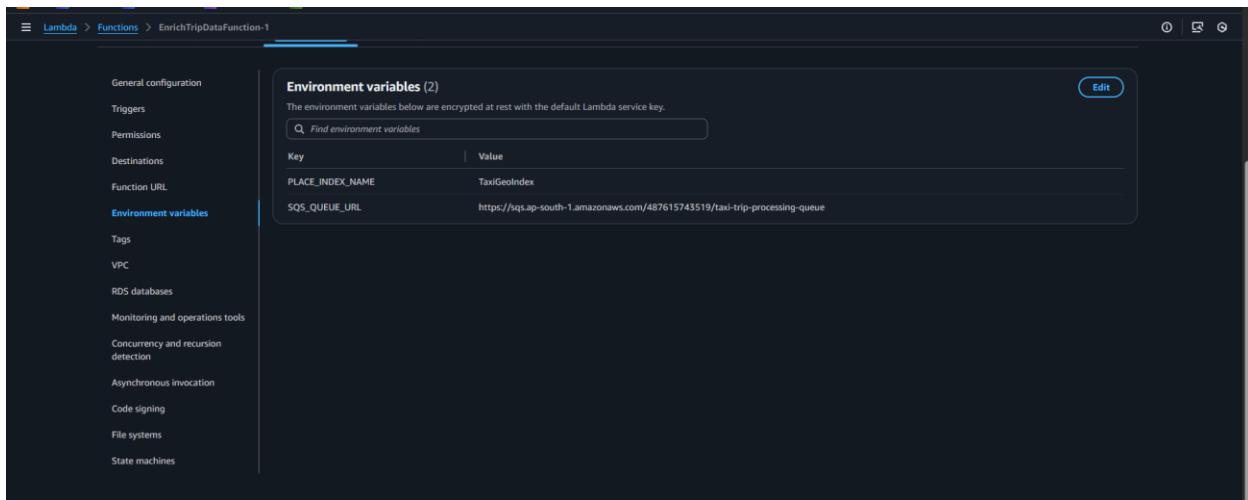
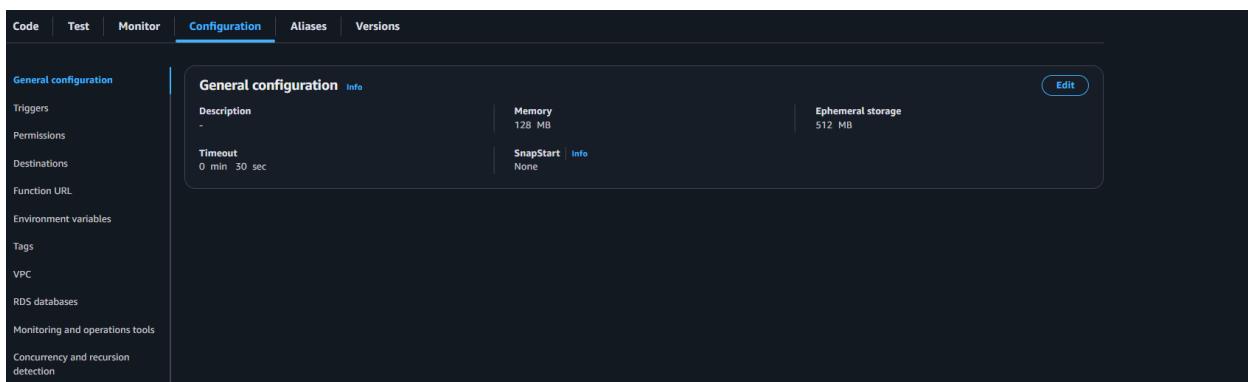
```



```

lambda_function.py
1 import base64
2 import json
3 import os
4 import boto3
5
6 # Initialize AWS clients outside the handler for better performance
7 # Boto3 will reuse these clients across multiple invocations
8 LOCATION_CLIENT = boto3.client('location')
9 LAMBDA_CLIENT = boto3.client('lambda')
10
11 # Get environment variables
12 # These must be set in your Lambda function's configuration
13 PLACE_INDEX_NAME = os.environ.get('PLACE_INDEX_NAME')
14 PROCESS_LAMBDA_NAME = os.environ.get('PROCESS_LAMBDA_NAME')
15
16 def lambda_handler(event, context):
17     """
18     This function is triggered by a Kinesis Data Stream.
19     It enriches the trip data with a zone name from AWS Location Service
20     and then invokes the next Lambda function in the chain.
21     """
22
23     if not PLACE_INDEX_NAME or not PROCESS_LAMBDA_NAME:
24         print("Error: Environment variables PLACE_INDEX_NAME and PROCESS_LAMBDA_NAME must be set.")
25         return {'statusCode': 500}
26
27     # Kinesis events contain a list of records
28     for record in event['Records']:

```



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_datetimestamp', '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"X 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("\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'])
    })
}
}

```

REAL-TIME TAXI TELEMETRY STREAMING | YASHWANTH ARAVANTI

The screenshot shows the 'Create function' wizard. It has three main options for creating a 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: ProcessAndStoreFunction

Runtime: Python 3.13

Architecture: x86_64

Permissions: 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.

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.

The screenshot shows the 'Add trigger' configuration page for an SQS queue. The selected trigger type is SQS, specifically event-source-mapping polling queue. The ARN of the queue is listed as 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. Set to 10.

Batch window - optional: The maximum amount of time to gather records before invoking the function, in seconds. Set to 0.

Maximum concurrency - optional: The maximum number of concurrent function instances that the SQS event source can invoke. Set to 1.

The screenshot shows the details page for the 'ProcessAndStoreFunction'. The function name is ProcessAndStoreFunction.

Function overview

- Diagram**: Shows a visual representation of the function architecture.
- Template**: Provides a template for the function.

Triggers

- SQS**: An SQS trigger named ProcessAndStoreFunction.

Actions

- Throttle
- Copy ARN
- Actions
- Export to Infrastructure Composer
- Download

Description

Last modified: 1 minute ago

Function ARN: arn:aws:lambda:ap-south-1:487615743519:function:ProcessAndStoreFunction

Function URL: Info

Code, **Test**, **Monitor**, **Configuration** (selected), **Aliases**, **Versions**

Lambda Chaining Summary:

```

[Source: Producer Script]
  |
  v
[Kinesis Data Stream: taxi-trip-stream]
  |
  v (Triggers)
[Lambda: EnrichTripDataFunction]
  |
  v (Sends message to)
[SQS Queue: taxi-trip-processing-queue] ----> [SQS DLQ: taxi-trip-dlq]
                                                (For failed messages)
  |
  v (Triggers)
[Lambda: ProcessAndStoreFunction]
  |
  -----> [Destination 1: DynamoDB Table (TaxiBilling)]
  |
  -----> [Destination 2: Kinesis Firehose]
    |
    v
    [S3 Bucket (Data Lake)]
    |
    v
    [Amazon Redshift (Data Warehouse)]
  
```

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.

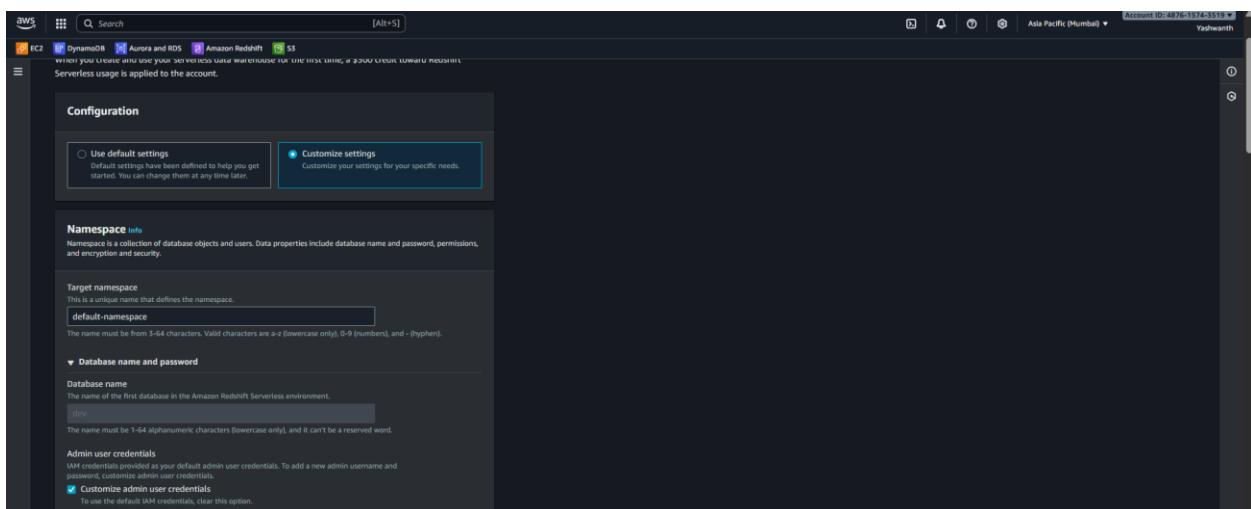
```

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.



REAL-TIME TAXI TELEMETRY STREAMING | YASHWANTH ARAVANTI

The screenshot shows the AWS Lambda console interface. At the top, there's a navigation bar with 'Lambda' selected. Below it, a search bar and a 'Create function' button. The main area is titled 'Real-time taxi telemetry streaming'. It shows the configuration for a new function:

- Runtime:** Python 3.8
- Handler:** app.lambda_handler
- Role:** Lambda execution role - *lambda-redshift-streaming-role*
- Code:** *lambda/redshift-streaming*
- Environment:** *environment*
- Logs:** CloudWatch Logs
- Deployment:** *Deploy*

Below the configuration, there's a section for 'AWS Lambda layers' which is currently empty.

The screenshot shows the AWS Redshift console interface. At the top, there's a navigation bar with 'Redshift' selected. Below it, a search bar and a 'Create workgroup' button. The main area is titled 'Workgroup - default-workgroup'. It shows the configuration for a new workgroup:

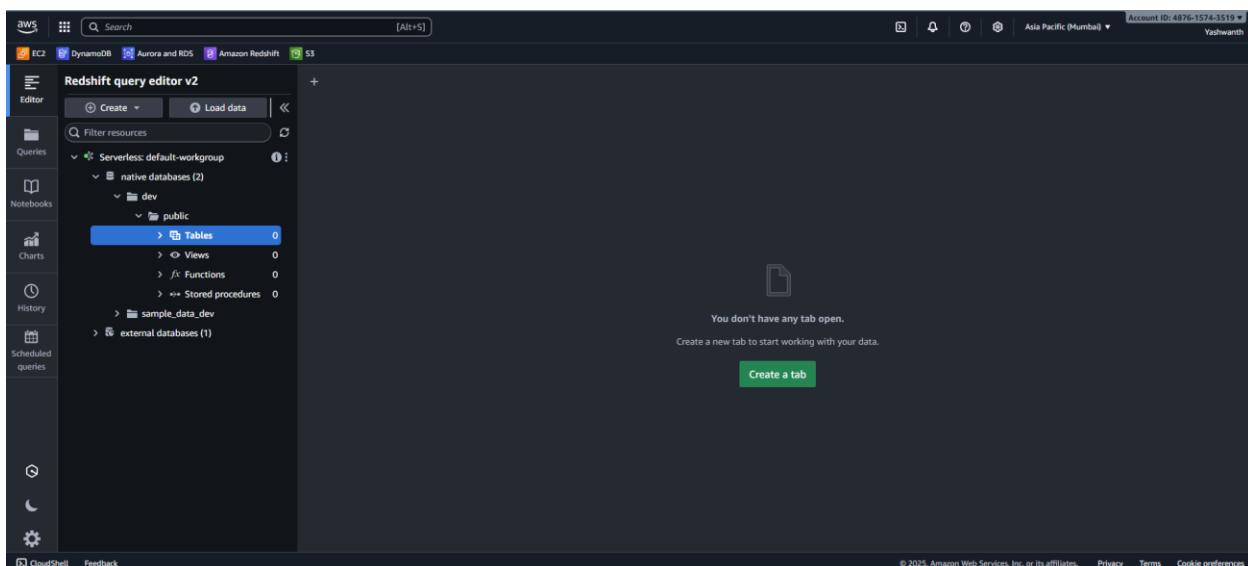
- Name:** default-workgroup
- Compute capacity:** 4 RPU
- Track:** Current
- Network and security:** IP address type: Info

Below the configuration, there's a section for 'Logs' which is currently empty.

The screenshot shows the AWS Redshift Serverless dashboard. At the top, there's a navigation bar with 'Amazon Redshift Serverless' selected. Below it, a search bar and a 'Create workgroup' button. The main area is divided into several sections:

- Namespace overview:** Shows 0 total snapshots, 0 data shares in my account, 0 requiring authorization, 0 from other accounts, and 0 requiring association.
- Namespaces / Workgroups:** Shows namespaces: default-namespace (Available) and default-workgroup (Available). It includes a line chart for average query duration.
- Queries metrics:** Shows metrics for running and queued queries, completed and failed queries, and number of queries.
- Total compute usage - new:** Shows total consumed RPU hours and a note about retrieving costs.
- Free trial:** Shows a link to the free trial information.

REAL-TIME TAXI TELEMETRY STREAMING | YASHWANTH ARAVANTI



This screenshot shows the AWS Redshift query editor v2 interface with a running query. The query is a 'CREATE TABLE' statement for a table named 'enriched_trip_data'. The table has 22 columns with various data types like VARCHAR, DOUBLE PRECISION, and TIMESTAMP. The statement includes options like 'IF NOT EXISTS', 'DISTSTYLE AUTO', and 'SORTKEY(pickup_datetime)'. The interface shows tabs for 'Untitled 1', 'Untitled 3', and 'Untitled 2'. Below the code, the 'Result 1' tab is active, displaying a 'Summary' section with an info message: 'Relation "enriched_trip_data" already exists and will be skipped'. The bottom of the screen shows standard AWS footer links for CloudShell, Feedback, and copyright information.

```
1 CREATE TABLE IF NOT EXISTS enriched.trip_data (
2     trip_id           VARCHAR(50),
3     taxi_id           VARCHAR(50),
4     pickup_datetime   TIMESTAMP,
5     dropoff_datetime  TIMESTAMP,
6     passenger_count   INT DEFAULT 0,
7     distance_km       DOUBLE PRECISION,
8     pickup_longitude  DOUBLE PRECISION,
9     pickup_latitude   DOUBLE PRECISION,
10    dropoff_longitude DOUBLE PRECISION,
11    dropoff_latitude  DOUBLE PRECISION,
12    zone_name          VARCHAR(255),
13    payment_type       VARCHAR(20),
14    fare_amount        DOUBLE PRECISION,
15    extra_charges      DOUBLE PRECISION DEFAULT 0,
16    tip_amount         DOUBLE PRECISION DEFAULT 0,
17    toll_amount        DOUBLE PRECISION DEFAULT 0,
18    total_amount       DOUBLE PRECISION DEFAULT 0,
19    created_at         TIMESTAMP DEFAULT CURRENT_TIMESTAMP
20 )
21 DISTSTYLE AUTO
22 SORTKEY(pickup_datetime);
```

Run Producer.py

```

producer.py > send_data_in_batches
13  def send_data_in_batches(file_path, stream_name):
22      try:
23          with open(file_path, 'r', encoding='utf-8') as csv_file:
24              csv_reader = csv.DictReader(csv_file)
25
26              for row in csv_reader:
27                  # Ensure pickup_datetime exists
28                  if 'pickup_datetime' not in row or not row['pickup_datetime'].strip():
29                      print(f"Skipping trip_id {row.get('trip_id', 'N/A')}: missing pickup_datetime")
30                      continue
31
32                  # Convert numeric fields
33                  for col in ['latitude', 'longitude', 'pickup_lat', 'pickup_long', 'drop_lat', 'drop_long', 'distance_km']:
34                      row[col] = float(row.get(col, 0))
35
36
37      ...
38
39  <>Finished sending all data.
PS C:\Users\ADMIN\Desktop\DETraining\AWS Data Engineering\Projects> python -u "c:\Users\ADMIN\Desktop\DETraining\AWS Data Engineering\Projects\producer.py"
Sending data from 'demo.csv' to Kinesis stream 'taxi-trip-stream-1'...
Sent batch of 8 records.
Finished sending all data.
PS C:\Users\ADMIN\Desktop\DETraining\AWS Data Engineering\Projects>

```

EnrichTripDataFunction Lambda Logs

Timestamp	Message
2025-10-19T20:12:27.316Z	INIT_START Runtime Version: python3.13.v64 Runtime Version ARN: arn:aws:lambda:ap-south-1:runtime:r7e78198cc1837d99c8e576c9b94c4d0b8ff8067b86f7d3481b75df6d76c24
2025-10-19T20:12:27.785Z	START RequestId: 2202e952-ff66-4fa5-bbe4-7df0fb2fb1 Version: \$LATEST
2025-10-19T20:12:27.785Z	Processing Trip ID: T0083
2025-10-19T20:12:28.180Z	Found Zone: 411, Bharati Vidyapeeth University, Dhankawadi, Pune 411043, India
2025-10-19T20:12:28.398Z	Successfully sent message to SQS for Trip ID: T0083
2025-10-19T20:12:28.398Z	Processing Trip ID: T0083
2025-10-19T20:12:28.533Z	Found Zone: Stand Bank Road, Chitpur, Kolkata 700002, India
2025-10-19T20:12:28.539Z	Successfully sent message to SQS for Trip ID: T0085
2025-10-19T20:12:28.539Z	Processing Trip ID: T0085
2025-10-19T20:12:28.671Z	Found Zone: 498 Buning Chat Road, Rayillyars Society, Koregaon Park, Pune 411001, India
2025-10-19T20:12:28.678Z	Successfully sent message to SQS for Trip ID: T0086
2025-10-19T20:12:28.699Z	END RequestId: 2202e952-ff66-4fa5-bbe4-7df0fb2fb1
2025-10-19T20:12:28.699Z	REPORT RequestId: 2202e952-ff66-4fa5-bbe4-7df0fb2fb1 Duration: 914.24 ms Billed Duration: 1379 ms Memory Size: 128 MB Max Memory Used: 88 MB Init Duration: 464.51 ms

No newer events at this moment. Auto retry paused. Resume

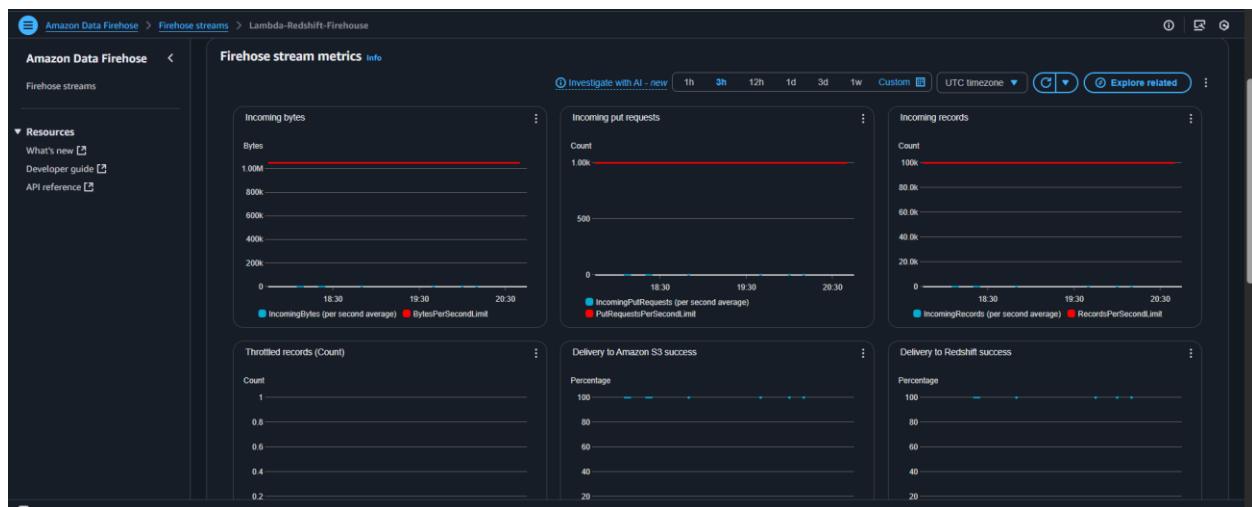
ProcessAndStoreFunction Lambda Logs:

The screenshot shows the AWS CloudWatch Log groups interface. The left sidebar navigation includes CloudWatch, Favorites and recents, Dashboards, AI Operations, Alarms, Logs, Log groups, Metrics, and more. The main area displays 'Log events' for the '/aws/lambda/ProcessAndStoreFunction' log group. The log entries are timestamped and show the execution of the Lambda function, including the processing of enriched data, calculating a fare of 375.6, successfully writing to the 'TaxiBilling' table in DynamoDB, and successfully sending data to the 'Lambda-Redshift-Firehose' stream.

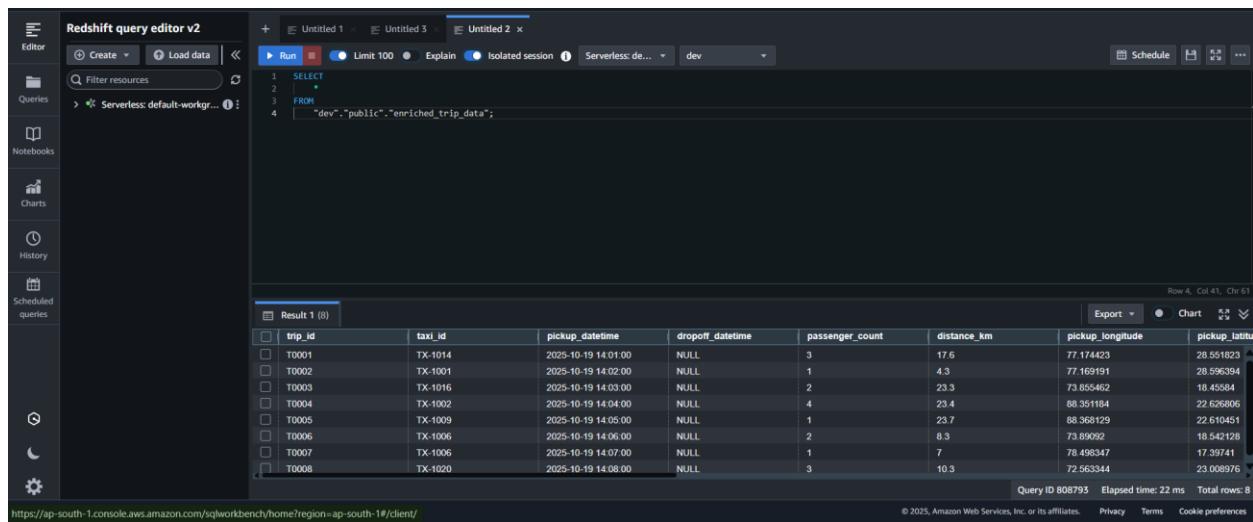
DynamoDB Table:

The screenshot shows the AWS DynamoDB Items page for the 'TaxiBilling' table. A green header bar indicates 'Completed - Items returned: 8 - Items scanned: 8 - Efficiency: 100% - RCU consumed: 2'. The table data is as follows:

trip_id (String)	pickup_datetime (String)	distance_km	drop_lat	drop_long	extra_charges	fare_amount	passenger_count
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:



The screenshot shows the Redshift query editor interface. On the left, there's a sidebar with options like Editor, Queries, Notebooks, Charts, History, and Scheduled queries. The main area has tabs for Untitled 1, Untitled 3, and Untitled 2. Untitled 2 is active, showing a query:

```

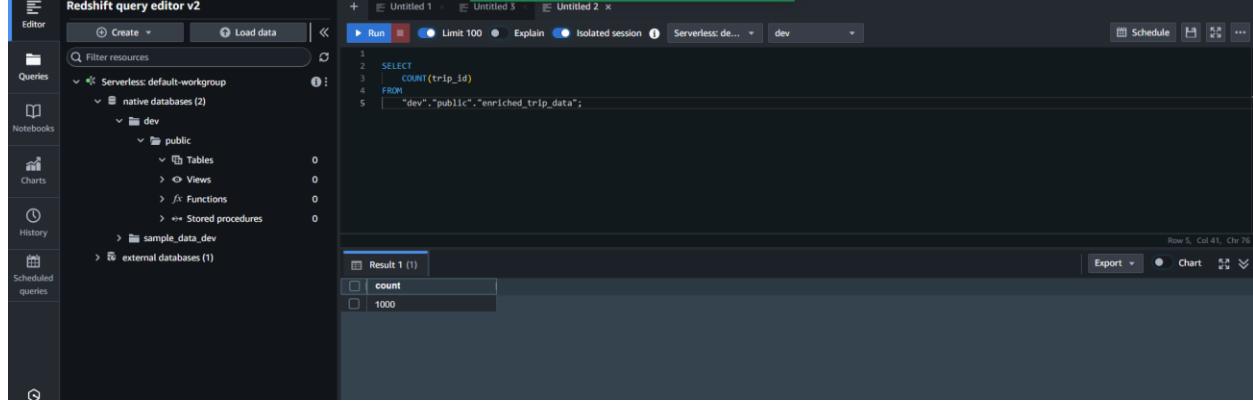
1 SELECT
2   *
3 FROM
4   "dev"."public"."enriched_trip_data";
    
```

Below the query is a results pane titled "Result 1 (8)" showing 8 rows of trip data:

trip_id	taxi_id	pickup_datetime	dropoff_datetime	passenger_count	distance_km	pickup_longitude	pickup_latitude
T0001	TX-1014	2025-10-19 14:01:00	NULL	3	17.6	77.174423	28.551023
T0002	TX-1001	2025-10-19 14:02:00	NULL	1	4.3	77.169191	28.596394
T0003	TX-1016	2025-10-19 14:03:00	NULL	2	23.3	73.855462	18.45584
T0004	TX-1002	2025-10-19 14:04:00	NULL	4	23.4	88.351184	22.626806
T0005	TX-1009	2025-10-19 14:05:00	NULL	1	23.7	88.368129	22.610451
T0006	TX-1006	2025-10-19 14:06:00	NULL	2	8.3	73.89092	18.542128
T0007	TX-1006	2025-10-19 14:07:00	NULL	1	7	78.495347	17.39741
T0008	TX-1020	2025-10-19 14:08:00	NULL	3	10.3	72.563344	23.008976

At the bottom, it says "Query ID 808793 Elapsed time: 22 ms Total rows: 8".

<https://ap-south-1.console.aws.amazon.com/sqlworkbench/home?region=ap-south-1#/client/>



This screenshot shows the Redshift query editor with a more complex navigation sidebar. It lists "Serverless: default-workgroup" and "native databases (2)". Under "native databases (2)", it shows "dev" which contains "public" (Tables: 0, Views: 0, Functions: 0, Stored procedures: 0) and "sample_data_dev". It also lists "external databases (1)".

The main query editor window shows a query:

```

1 SELECT
2   COUNT(trip_id)
3 FROM
4   "dev"."public"."enriched_trip_data";
5   
```

The results pane shows a single row with a count of 1000.

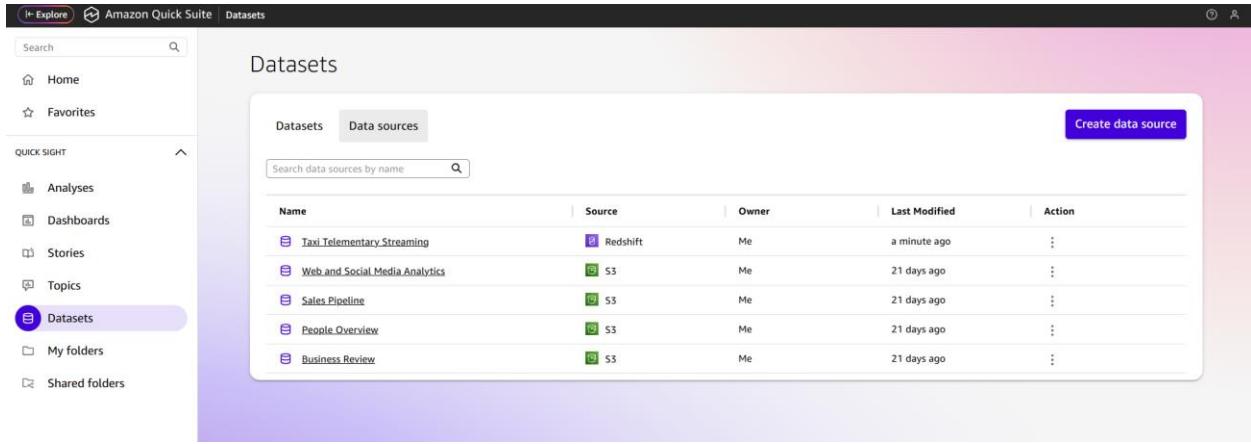
Step 10: Create Amazon QuickSight Dashboard (Visualization Layer)

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:
 - o Trip volume by zone
 - o Fare distribution
 - o Real-time revenue by time of day

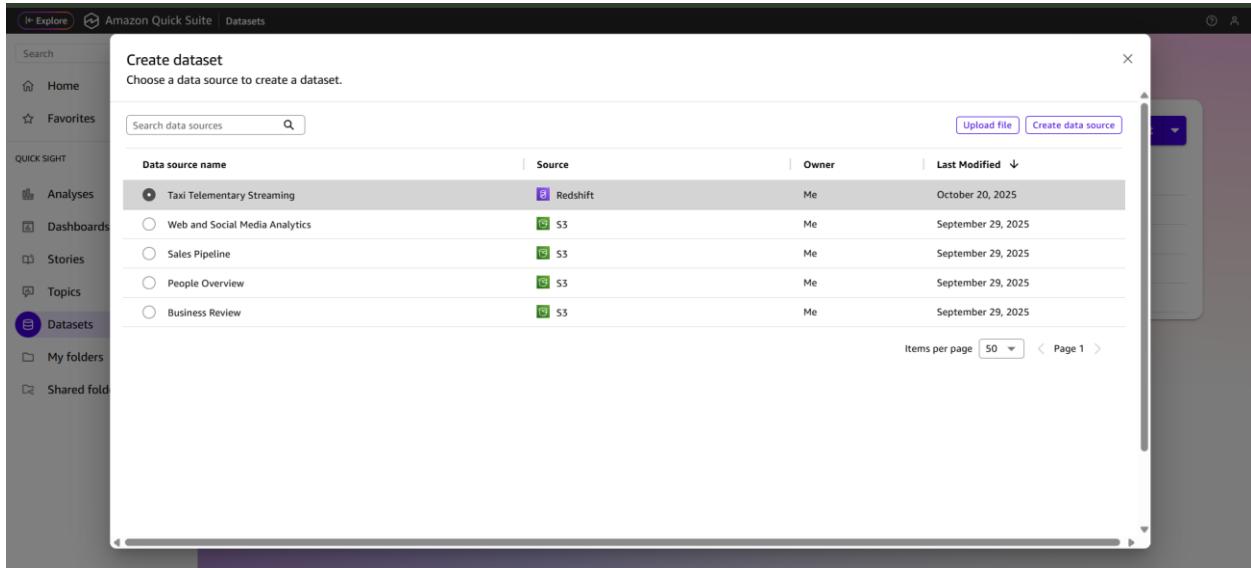
Purpose:

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



The screenshot shows the Amazon QuickSight interface with the 'Datasets' tab selected. On the left, there's a sidebar with 'Home', 'Favorites', 'Analyses', 'Dashboards', 'Stories', 'Topics', and 'Datasets' (which is highlighted). The main area displays a table of datasets with columns for Name, Source, Owner, Last Modified, and Action. The datasets listed are:

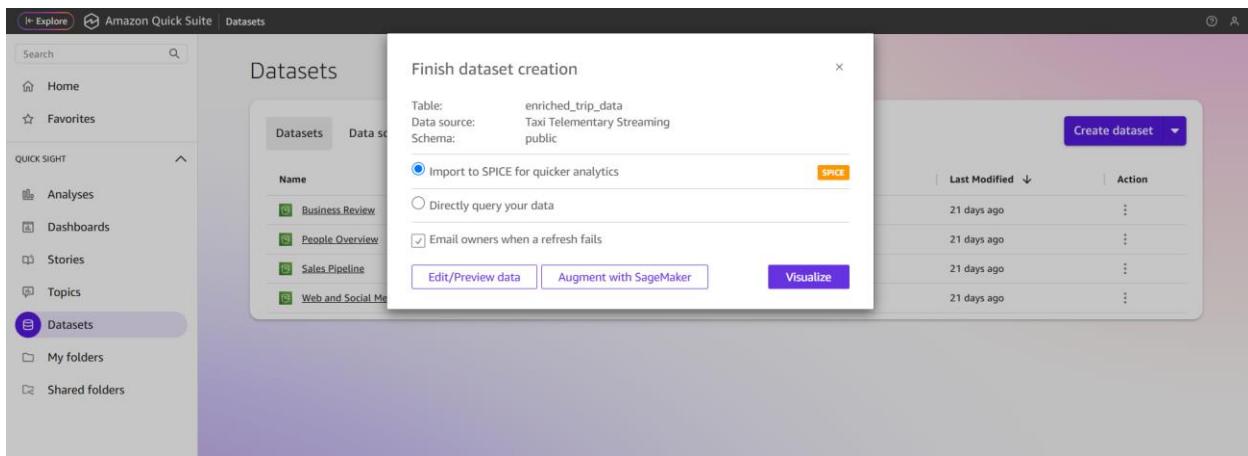
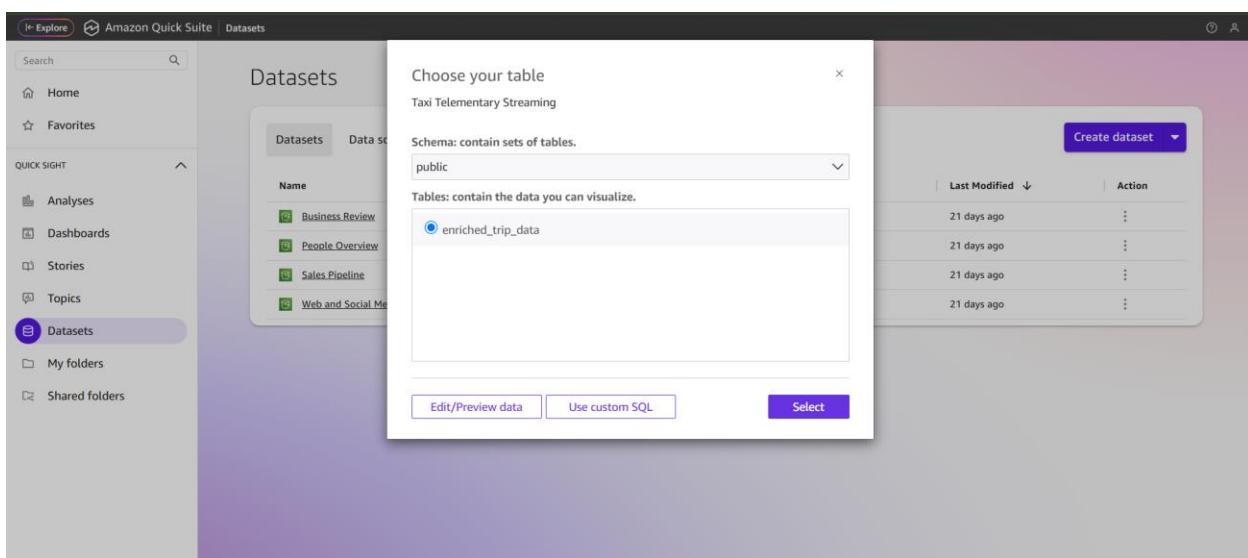
Name	Source	Owner	Last Modified	Action
Taxi Telemetry Streaming	Redshift	Me	a minute ago	⋮
Web and Social Media Analytics	S3	Me	21 days ago	⋮
Sales Pipeline	S3	Me	21 days ago	⋮
People Overview	S3	Me	21 days ago	⋮
Business Review	S3	Me	21 days ago	⋮



The screenshot shows the 'Create dataset' dialog box. It has a search bar at the top and a table below it. The table lists data sources with columns for Data source name, Source, Owner, and Last Modified. The data sources listed are:

Data source name	Source	Owner	Last Modified
Taxi Telemetry Streaming	Redshift	Me	October 20, 2025
Web and Social Media Analytics	S3	Me	September 29, 2025
Sales Pipeline	S3	Me	September 29, 2025
People Overview	S3	Me	September 29, 2025
Business Review	S3	Me	September 29, 2025

REAL-TIME TAXI TELEMETRY STREAMING | YASHWANTH ARAVANTI



Charts & Visuals:

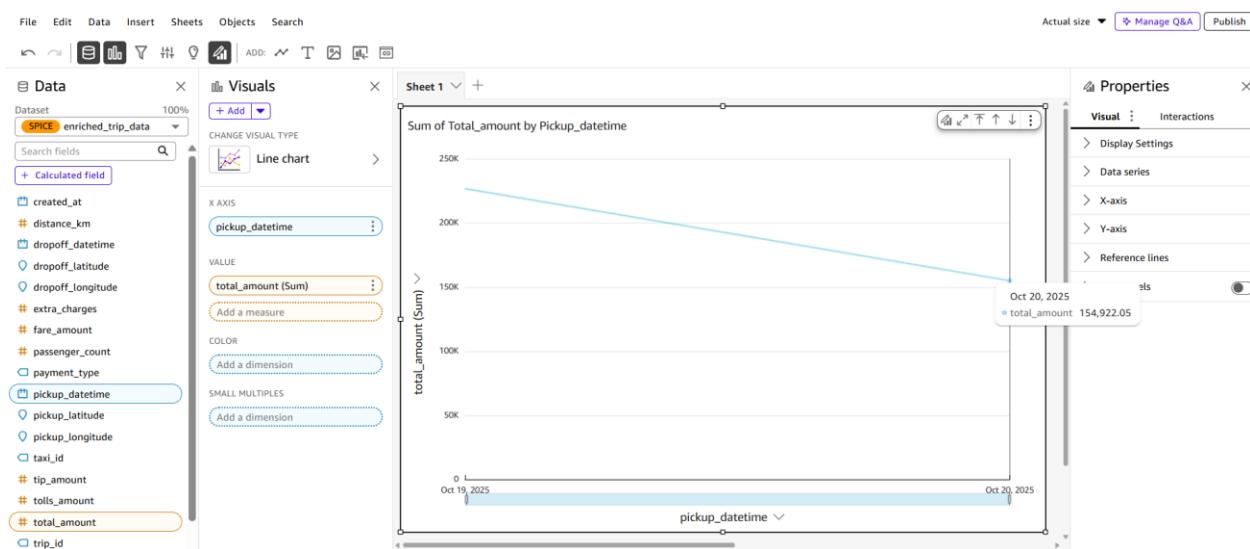
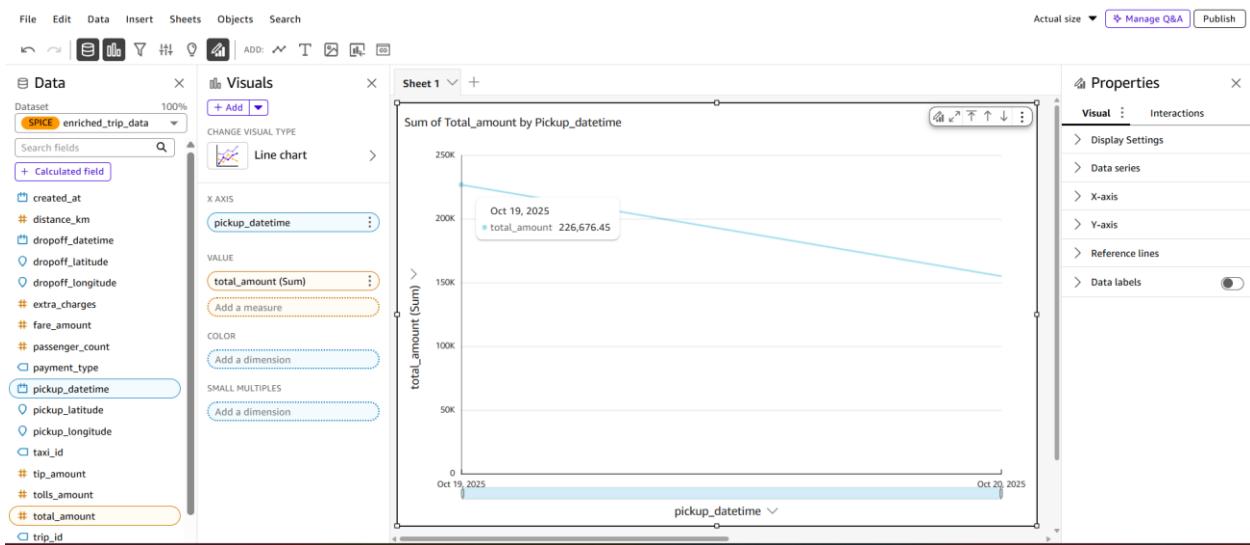
1. Revenue & Trip Trends

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.

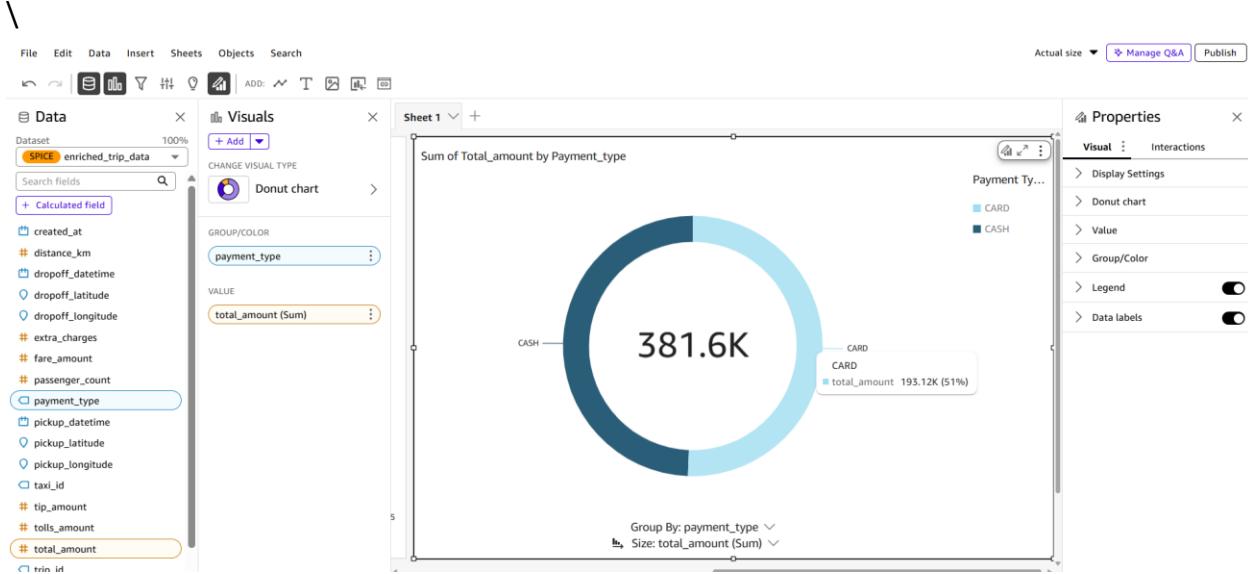
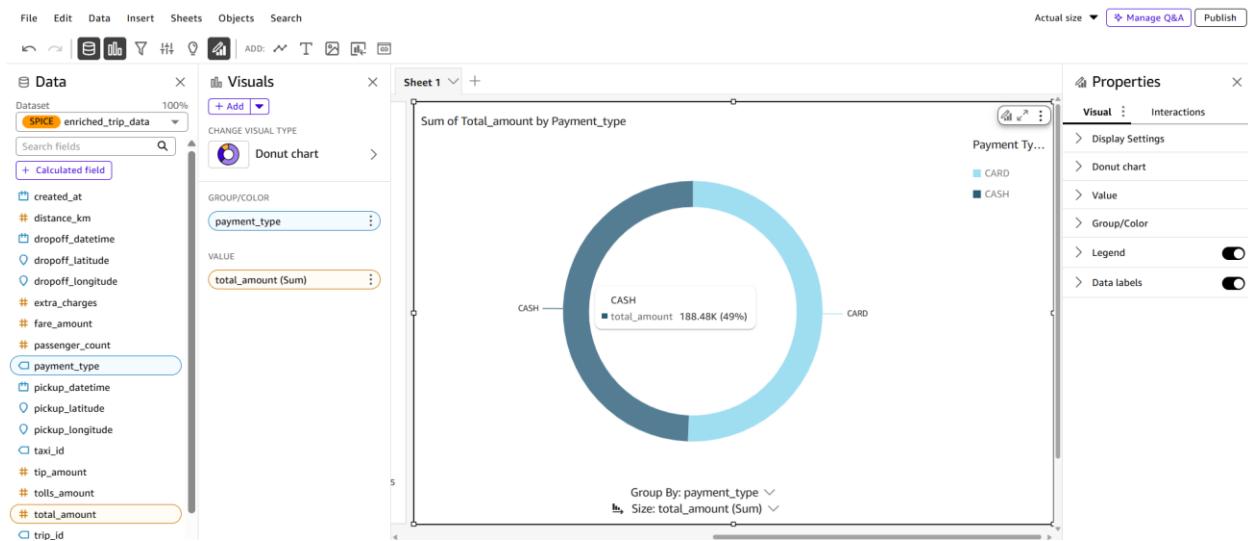


2. Payment Type Breakdown

Chart Type: Donut / Pie Chart

Data: payment_type vs SUM(total_amount)

Insight: Cash vs Card usage ratio.



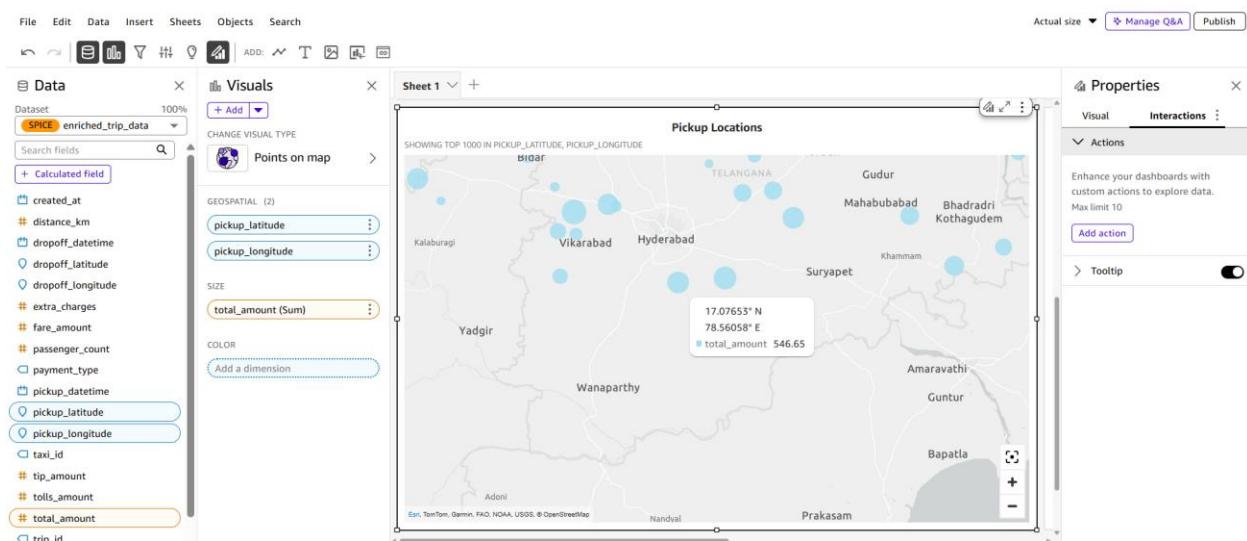
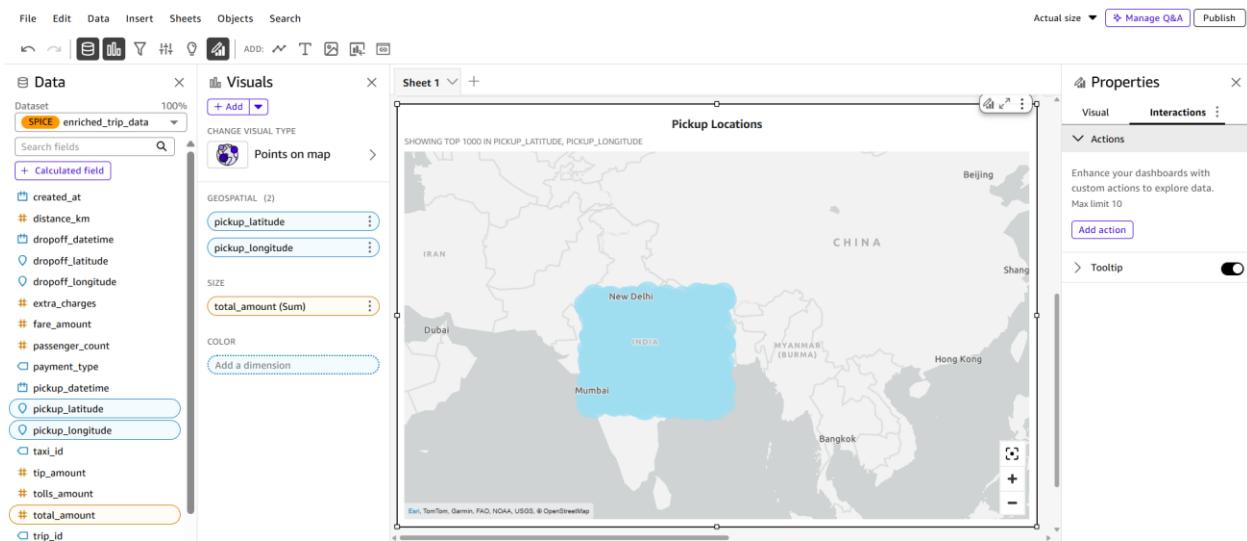
3. Points on Map (Pickup Locations)

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.)



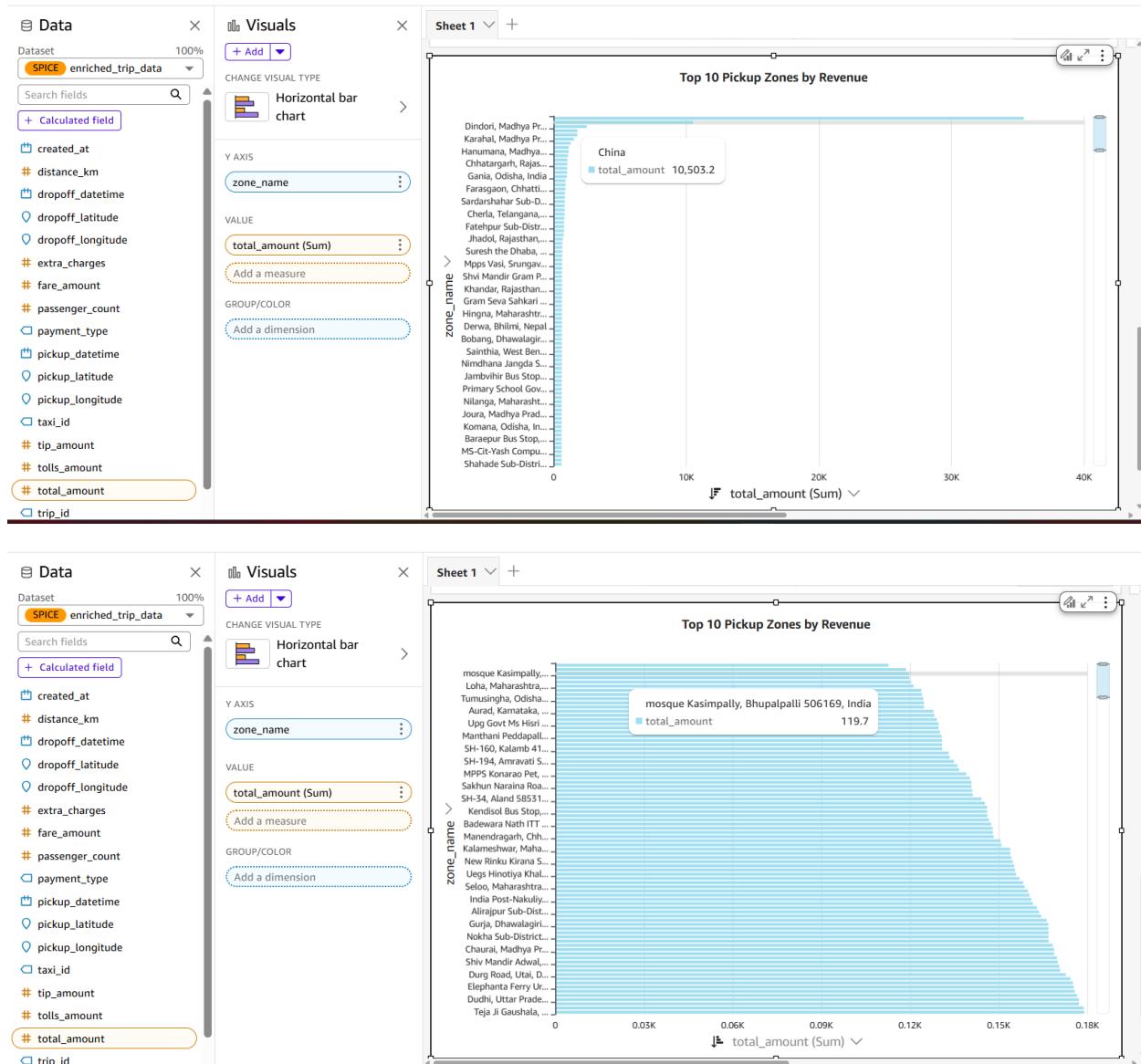
4. Top 10 Pickup Zones by Revenue

Chart Type: Horizontal Bar Chart

X-axis: SUM(total_amount)

Y-axis: zone_name (Top 10)

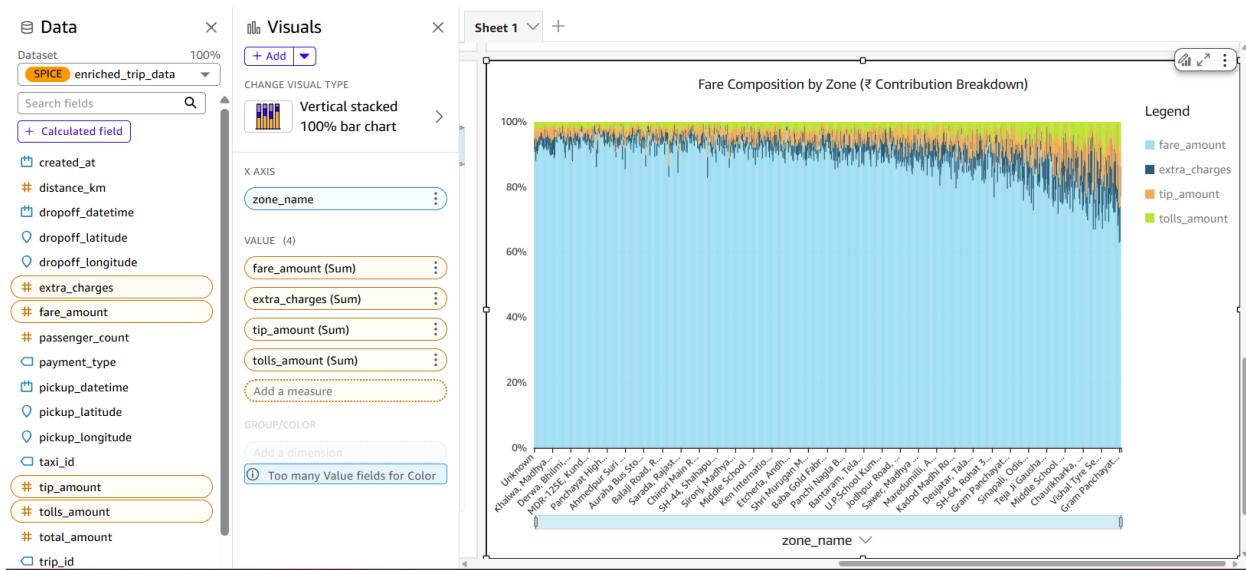
Insight: Which areas generate the most business.



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



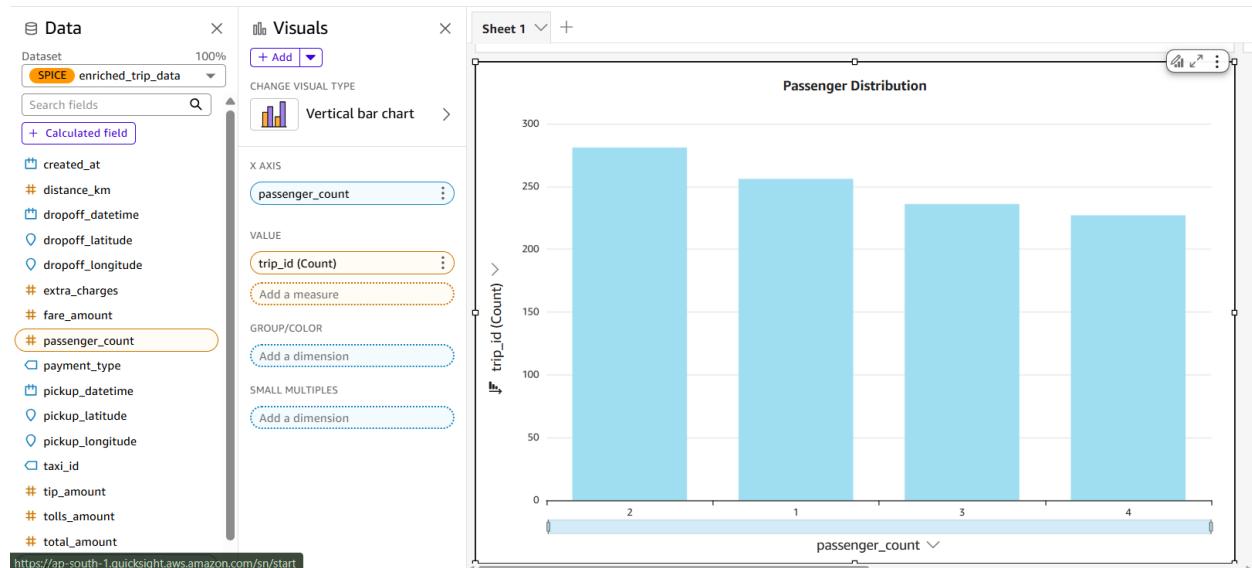
6. Passenger Distribution

Chart Type: Bar / Histogram

X-axis: passenger_count

Y-axis: COUNT(trip_id)

Insight: Typical passenger loads per trip.



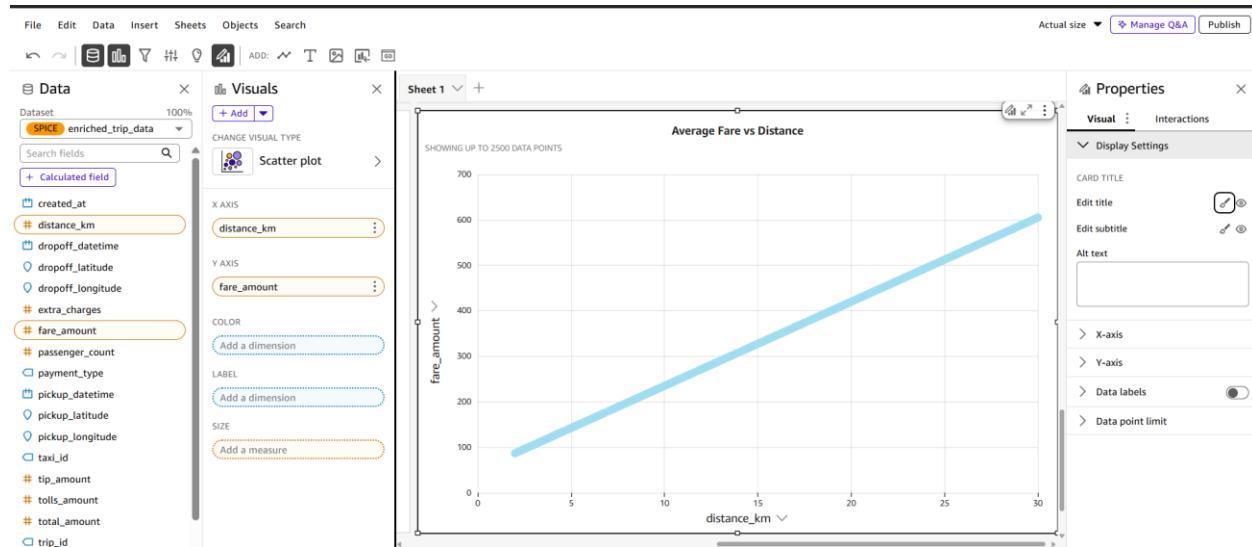
7. Average Fare vs Distance

Chart Type: Scatter Plot

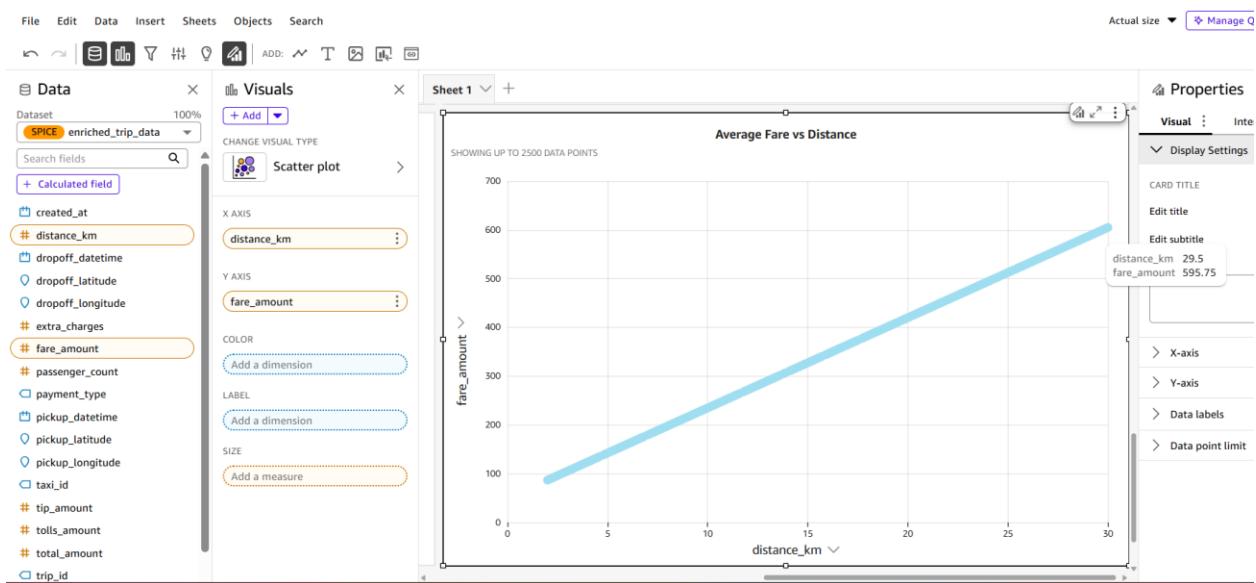
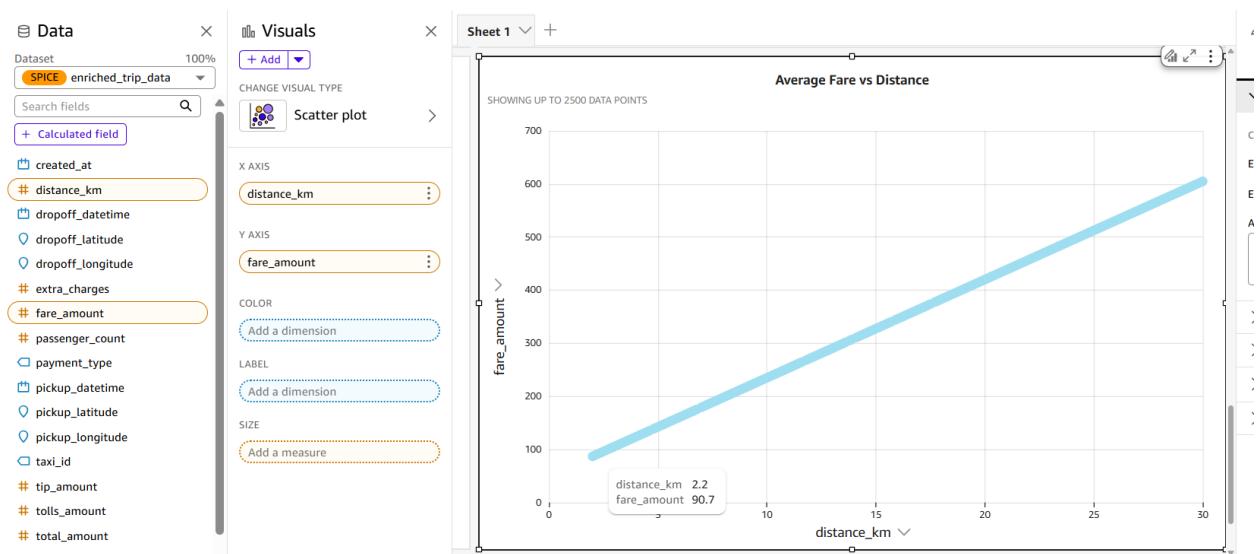
X-axis: distance_km

Y-axis: fare_amount

Insight: Identify any outlier trips (too expensive or too cheap for distance).



REAL-TIME TAXI TELEMETRY STREAMING | YASHWANTH ARAVANTI



Pipeline 2: Raw Data Archiving Pipeline

Goal:

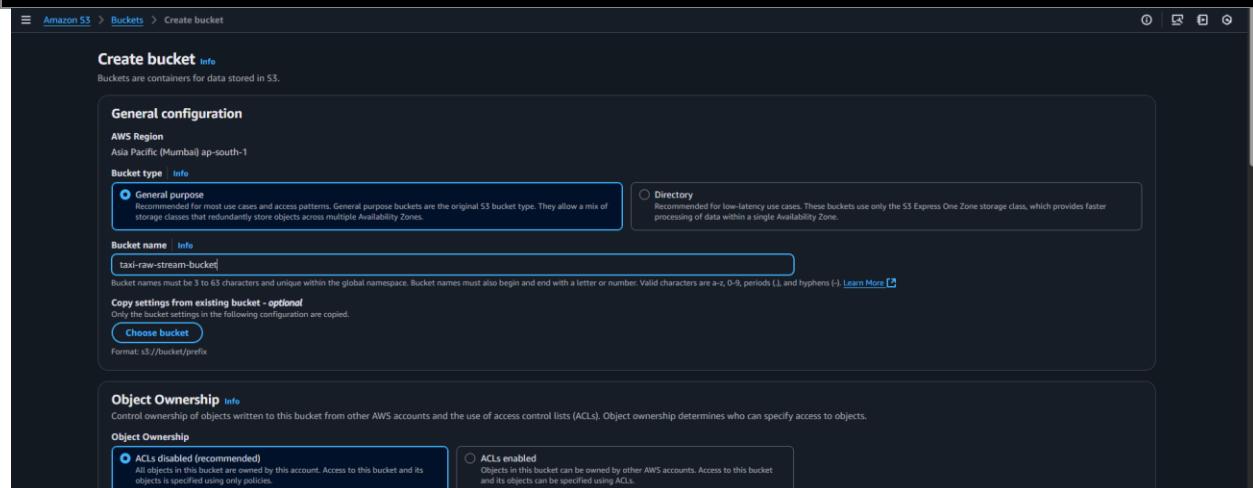
Store all raw taxi/transportation telemetry data reliably and automatically transition older data to low-cost archival tiers (S3 Glacier / Deep Archive) for long-term retention and auditing.

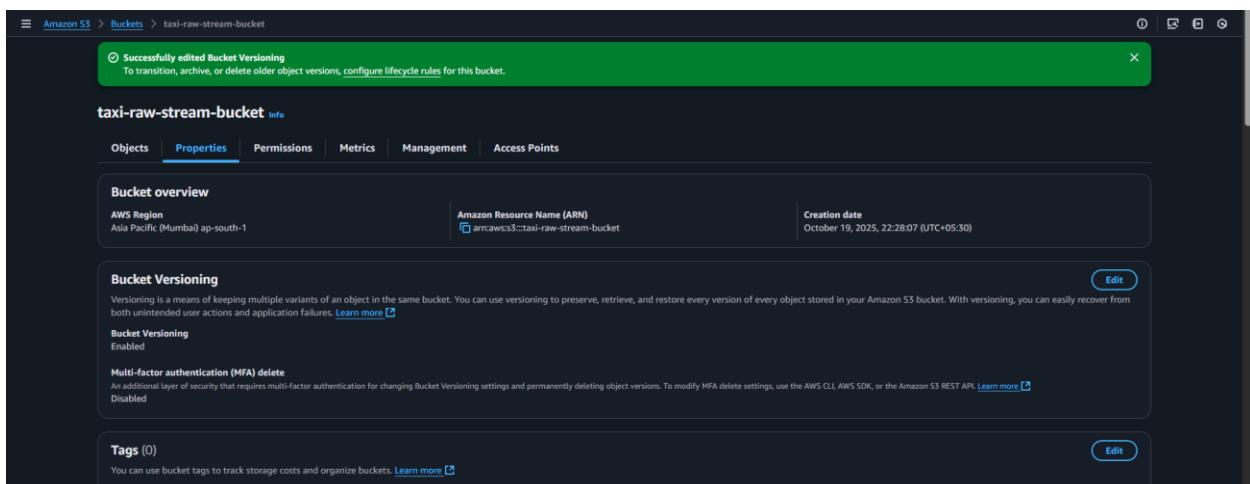
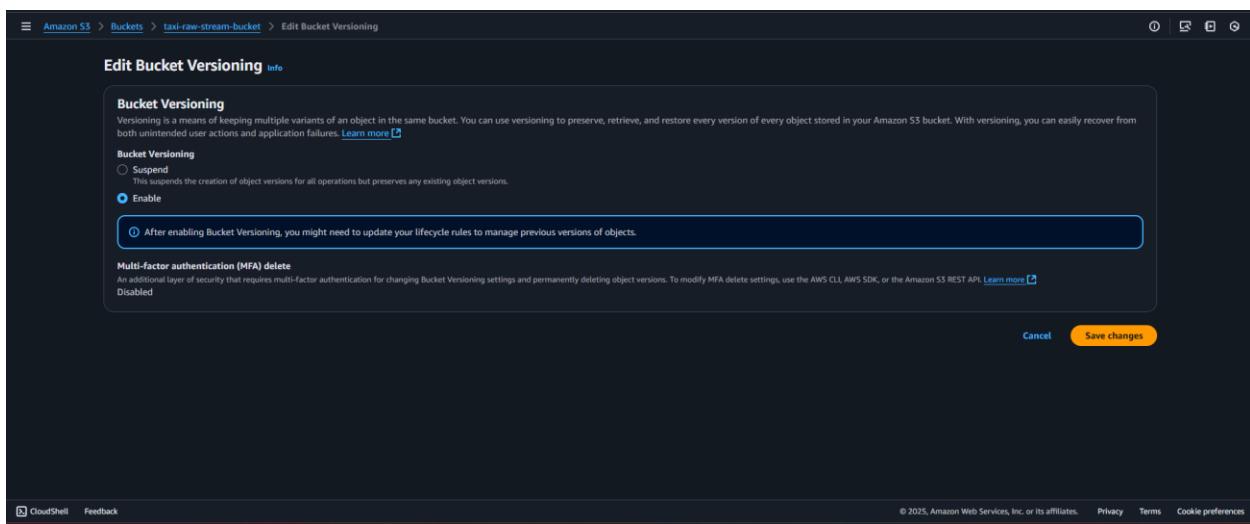
Step-by-Step Implementation:

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"
        },
        {
          "Days": 180,
          "StorageClass": "DEEP_ARCHIVE"
        }
      ]
    }
  ]
}
```

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

Amazon S3 > Buckets > taxi-raw-stream-bucket > Lifecycle configuration > Create lifecycle rule

Create lifecycle rule Info

Lifecycle rule configuration

Lifecycle rule name
DeepArchivePolicy
Up to 255 characters

Choose a rule scope
 Limit the scope of this rule using one or more filters
 Apply to all objects in the bucket

⚠️ Apply to all objects in the bucket
 If you want the rule to apply to specific objects, you must use a filter to identify those objects. Choose "Limit the scope of this rule using one or more filters". [Learn more](#)
 I acknowledge that this rule will apply to all objects in the bucket.

Lifecycle rule actions
 Choose the actions you want this rule to perform.

Transition current versions of objects between storage classes
 This action will move current versions.
 Transition noncurrent versions of objects between storage classes
 This action will move noncurrent versions.
 Expire current versions of objects
 Permanently delete noncurrent versions of objects
 Delete expired object delete markers or incomplete multipart uploads
 These actions are not supported when filtering by object tags or object size.

CloudShell Feedback © 2025, Amazon Web Services, Inc. or its affiliates. Privacy Terms Cookie preferences

Transition current versions of objects between storage classes
 Choose transitions to move current versions of objects between storage classes based on your use case scenario and performance access requirements. These transitions start from when the objects are created and are consecutively applied. [Learn more](#)

Choose storage class transitions

Glacier Flexible Retrieval (formerly Glacier)	Days after object creation 30	Remove
Glacier Deep Archive	180	Remove

Add transition

Permanently delete noncurrent versions of objects
 Choose when Amazon S3 permanently deletes specified noncurrent versions of objects. [Learn more](#)

Days after objects become noncurrent 730	Number of newer versions to retain - Optional Number of versions
---	---

Review transition and expiration actions

Current version actions Day 0	Noncurrent versions actions Day 0
---	---

Permanently delete noncurrent versions of objects
 Choose when Amazon S3 permanently deletes specified noncurrent versions of objects. [Learn more](#)

Days after objects become noncurrent 730	Number of newer versions to retain - Optional Number of versions
---	---

Review transition and expiration actions

Current version actions	Noncurrent versions actions
Day 0 <ul style="list-style-type: none"> Objects uploaded 	Day 0 <ul style="list-style-type: none"> Objects become noncurrent
Day 30 <ul style="list-style-type: none"> Objects move to Glacier Flexible Retrieval (formerly Glacier) 	Day 730 <ul style="list-style-type: none"> 0 newest noncurrent versions are retained All other noncurrent versions are permanently deleted
Day 180	
<ul style="list-style-type: none"> Objects move to Glacier Deep Archive 	

Cancel Create rule

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/

Choose source and destination
Specify the source and the destination for your Firehose stream. You cannot change the source and destination of your Firehose stream once it has been created.

Source: Amazon Kinesis Data Streams

Destination: Amazon S3

Firehose stream name
Firehose stream name: taxi-raw-firehose
Acceptable characters are uppercase and lowercase letters, numbers, underscores, hyphens, and periods.

Source settings
Kinesis data stream: arn:aws:kinesis:ap-south-1:487615743519:stream/taxi-trip-stream-1
Format: arn:aws:kinesis:[Region]:[AccountId]:stream/[StreamName]

Transform and convert records - optional

Destination settings
Specify the destination settings for your Firehose stream.

S3 bucket: s3://taxi-raw-stream-bucket
Format: s3://[bucket]

New line delimiter
You can configure your Firehose stream to add a new line delimiter between records in objects that are delivered to Amazon S3.
 Not enabled
 Enabled

Dynamic partitioning
Dynamic partitioning enables you to create targeted data sets by partitioning streaming S3 data based on partitioning keys. You can partition your source data with inline parsing and/or the specified AWS Lambda function. You can enable dynamic partitioning only when you create a new Firehose stream. You cannot enable dynamic partitioning for an existing Firehose stream. Enabling dynamic partitioning incurs additional costs per GB of partitioned data. For more information, see [Amazon Data Firehose pricing](#).

S3 bucket prefix - optional
By default, Amazon Data Firehose appends the prefix "YYYY/MM/dd/HH" (in UTC) to the data it delivers to Amazon S3. You can override this default by specifying a custom prefix that includes expressions that are evaluated at runtime.

S3 bucket error output prefix - optional
You can specify an S3 bucket error output prefix to be used in error conditions. This prefix can include expressions for Amazon Data Firehose to evaluate at runtime.

S3 buffer hints
Amazon Data Firehose buffers incoming records before delivering them to your S3 bucket. Record delivery is triggered once the value of either of the specified buffering hints is reached.

Buffer size
The higher buffer size may be lower in cost with higher latency. The lower buffer size will be faster in delivery with higher cost and less latency.
5 MiB
Minimum: 1 MiB, maximum: 128 MiB. Recommended: 5 MiB.

Buffer interval
The higher interval allows more time to collect data and the size of data may be bigger. The lower interval sends the data more frequently and may be more advantageous when looking at shorter cycles of data activity.
30 seconds
Minimum: 0 seconds, maximum: 900 seconds. Recommended: 300 seconds.

Compression for data records
Amazon Data Firehose can compress records before delivering them to your S3 bucket.

Not enabled
 GZIP
 Snappy
 Zip
 Hadoop-Compatible Snappy

File extension format - optional
Enter the full extension format. If you want to add 'gzip file extension' for your 'json' files, enter 'json.gz'
Enter a file extension (for example, json, zip, json.gz)

File extension must start with a period and may only contain allowed characters: 0-9a-z_~^". File extension cannot exceed 128 characters.

⚠️ If you specify a file extension, it will override the default file extension that Amazon Data Firehose adds when data format conversion or compression is enabled.

Encryption for data records
Record gets encrypted in the S3 bucket using an AWS Key Management Service (KMS) key.

Use the encryption setting of the S3 bucket
Default encryption for S3 bucket is server-side encryption with Amazon S3 managed keys (SSE-S3). See [S3: Protecting data with server-side encryption](#) for other encryption options.

Use SSE-KMS
Server-side encryption with AWS Key Management Service (KMS) keys.

Step 4 – Configure Permissions

- IAM role FirehoseToS3Role must have:

{

```
"Effect": "Allow",
```

```
"Action": ["s3:PutObject", "s3:PutObjectAcl"],
```

```
"Resource": "arn:aws:s3:::taxi-raw-stream-bucket/*"
```

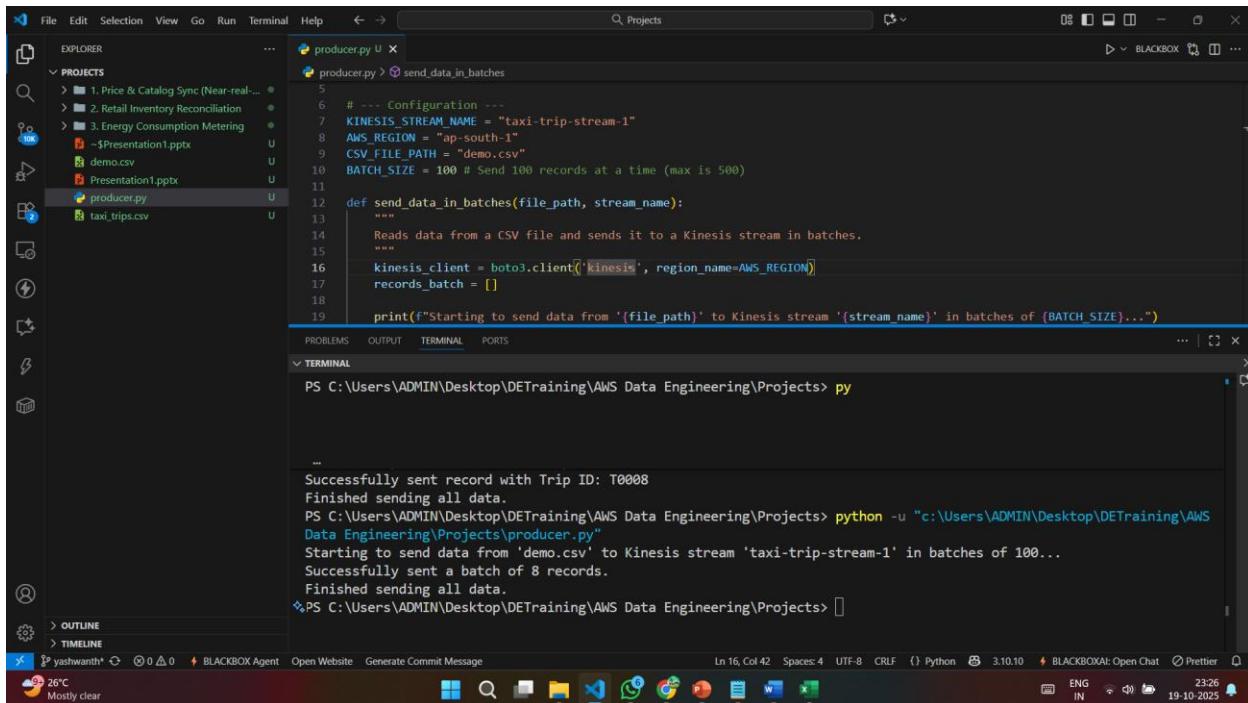
}

- Add Kinesis Stream read permissions too.

Step 5 – Validate Data Flow

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

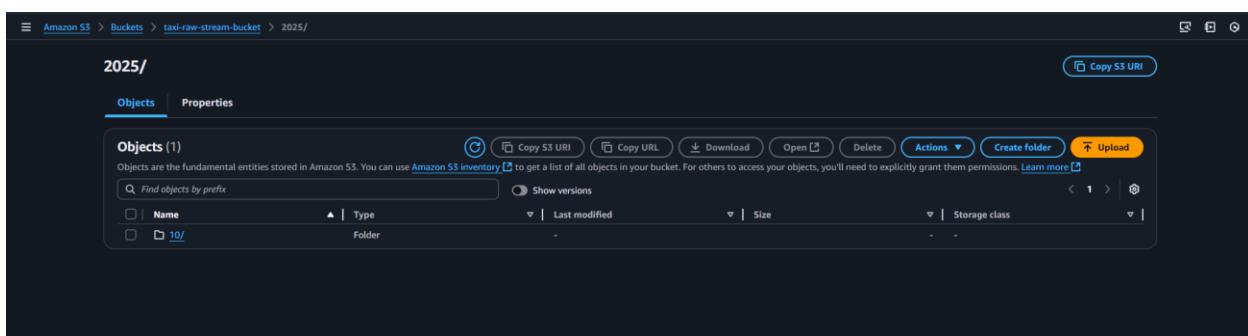
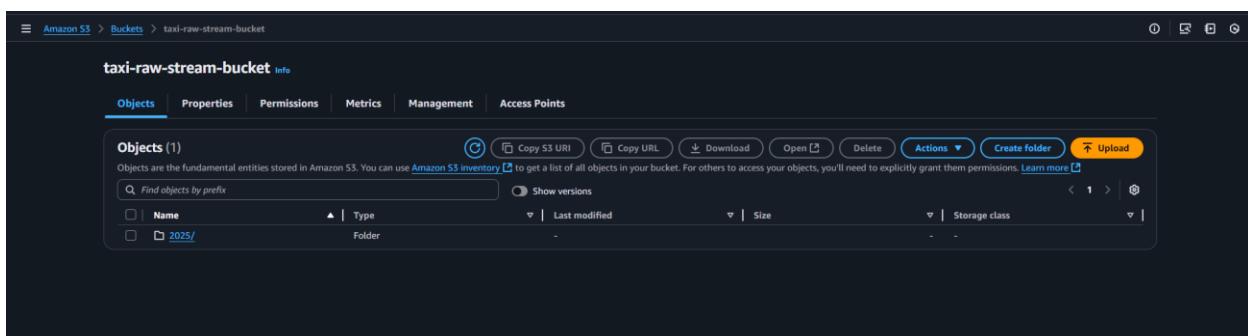
- Objects are created in partitions (date/hour folders).
- Lifecycle rules are automatically applied.



```

producer.py > send_data_in_batches
5
6 # --- Configuration ---
7 KINESIS_STREAM_NAME = "taxi-trip-stream-1"
8 AWS_REGION = "ap-south-1"
9 CSV_FILE_PATH = "demo.csv"
10 BATCH_SIZE = 100 # Send 100 records at a time (max is 500)
11
12 def send_data_in_batches(file_path, stream_name):
13     """
14     Reads data from a CSV file and sends it to a Kinesis stream in batches.
15     """
16     kinesis_client = boto3.client('kinesis', region_name=AWS_REGION)
17     records_batch = []
18
19     print(f"Starting to send data from '{file_path}' to Kinesis stream '{stream_name}' in batches of {BATCH_SIZE}...")
...
Successfully sent record with Trip ID: T0008
Finished sending all data.
PS C:\Users\ADMIN\Desktop\DETraining\AWS Data Engineering\Projects> python -u "c:\Users\ADMIN\Desktop\DETraining\AWS Data Engineering\Projects\producer.py"
Starting to send data from 'demo.csv' to Kinesis stream 'taxi-trip-stream-1' in batches of 100...
Successfully sent a batch of 8 records.
Finished sending all data.
PS C:\Users\ADMIN\Desktop\DETraining\AWS Data Engineering\Projects>

```



Amazon S3 > Buckets > taxi-raw-stream-bucket > 2025/ > 10/

10/

Objects (1)

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](#)

[Show versions](#)

Name	Type	Last modified	Size	Storage class
19/	Folder	-	-	-

Amazon S3 > Buckets > taxi-raw-stream-bucket > 2025/ > 10/ > 19/

19/

Objects (1)

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](#)

[Show versions](#)

Name	Type	Last modified	Size	Storage class
17/	Folder	-	-	-

Amazon S3 > Buckets > taxi-raw-stream-bucket > 2025/ > 10/ > 19/ > 17/

17/

Objects (4)

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](#)

[Show versions](#)

Name	Type	Last modified	Size	Storage class
taxi-raw-firehose-1-2025-10-19-17-54-5b-15e8caf-9eda-4793-b626-6e2193e96169.parquet	parquet	October 19, 2025, 23:25:29 (UTC+05:30)	768.0 B	Standard
taxi-raw-firehose-1-2025-10-19-17-54-5b-c845e210-c65c-47ff-999f-3ee41337270a.parquet	parquet	October 19, 2025, 23:25:29 (UTC+05:30)	515.0 B	Standard
taxi-raw-firehose-1-2025-10-19-17-54-5b-c9ec8b4-38fc-47d8-b0d8-be393d59328.parquet	parquet	October 19, 2025, 23:25:29 (UTC+05:30)	253.0 B	Standard
taxi-raw-firehose-1-2025-10-19-17-54-5b-f0bbdbd6-1490-40b3-a4c5-8665237f05e0.parquet	parquet	October 19, 2025, 23:25:29 (UTC+05:30)	515.0 B	Standard

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.