Final Project Summary – INFO-535 - Owen Randolph – 4/13/2025

Title: Customer Service Data Pipeline Using AWS

Introduction

The objective is to build a serverless, event-driven data pipeline that ingests, processes, stores, and analyzes customer service interactions using AWS. The goal is to allow data being handed off from business stakeholders to be adequately processed and transformed to provide actionable insights. This system supports customer support operations and performance evaluation for service representatives. It also provides a foundation for analytics applications used by data scientists and sales leadership.

Background

This data pipeline serves as a strategic bridge between raw business data and the teams that rely on it—namely, data scientists, analysts, and decision-makers. Understanding data pipelines and cloud service technologies has become an essential skill set for modern data professionals, particularly data scientists and machine learning engineers. This pipeline represents a key component of data architecture and infrastructure.

The system offers a modern, serverless cloud-based infrastructure that automates data ingestion and transformation, making data readily available for downstream use. By handling multiple data types, the architecture ensures that files are categorized, processed, and stored efficiently. AWS Athena further optimizes the data by converting it into Parquet format, enabling faster and more efficient SQL querying. This prepares the data not only for analytical tasks but also for machine learning, reporting, and data visualization—making it a critical enabler for data-driven decision-making in businesses.

Methodology

AWS Services Used

- **Amazon API Gateway**: REST API for data ingestion (e.g., chat messages, support forms)
- **Amazon S3**: Primary storage for CSVs, JSONs, Parquet files, and Athena query results
- **AWS Lambda**: Event-driven compute logic
    - Moves CSV files to a staging bucket
    - Runs Athena queries and stores results in S3
    - Inserts JSON files into DynamoDB
- **AWS Glue**:
    - Converts CSV files to Parquet format (ETL Jobs)
    - Crawlers update the Glue Data Catalog with schema
- **Glue Data Catalog**: Metadata store for Athena
- **Amazon Athena**: SQL querying engine for analyzing Parquet files in S3
- **Amazon DynamoDB**: NoSQL database for structured JSON data
- **AWS IAM**: Permissions and access control for all components

Data Flow & Processing Pipeline

1. **Ingestion Stage:**
    - Users or systems upload .csv or .json files into data-bucket-1988.
    - The API Gateway triggers a Lambda function based on file type.
2. **Storage and Transformation:**
    - CSV files are moved to staging-bucket-1988.
    - A Glue job transforms these files into Parquet format and stores them in parquet-output-bucket-1988.
3. **Cataloging:**
    - Glue Crawlers (csv-staging-crawler and parquet-output-crawler) scan respective directories and update the etl_data database in the Glue Data Catalog.
4. **Query and Reporting:**
    - Athena runs SQL queries over the Parquet files.
    - A Lambda function named runAthenaQuery is used to automate and schedule query executions.
    - Query results are stored in a designated Athena results bucket.
5. **DynamoDB Integration:**
    - JSON files from data-bucket-1988 are processed through the modified checkANDStage Lambda function.
    - Valid entries are inserted into the CustomerServiceReps table.

Datasets used as sample and testing data

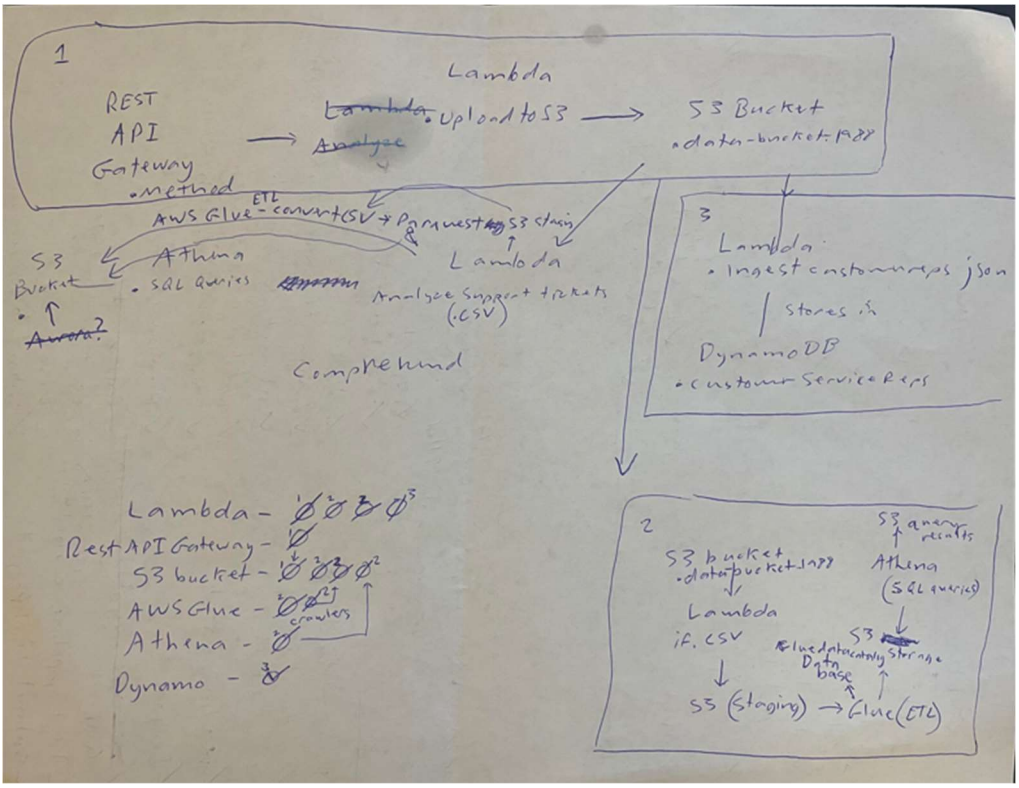
The datasets used are synthetic data created by me in relatively small sizes reflecting an example of real data that could use produced by a customer service department.

- “customer_support_tickets”: contains customer profile data and ticket information with ticket_id as the primary key
- “ticket_sentiments.csv_1”, “ticket_sentiments.csv_2”, “ticket_sentiments.csv_3”: provide sentiment scores mapped to ticket IDs
- “customer_service_reps.json”, “customer_service_reps_1.json”, “customer_service_reps.json_2”: contain support representative details with rep_id as the key

Methodology Hands On: Notes and Pictures from Build

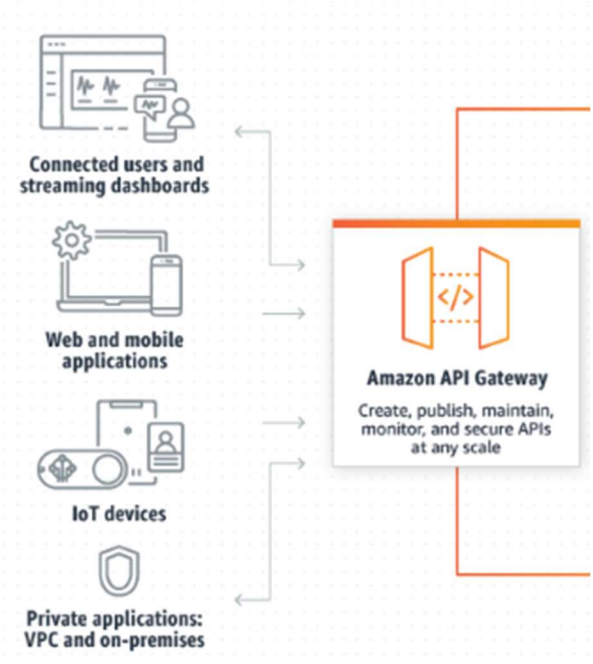
Updated Services & Flow

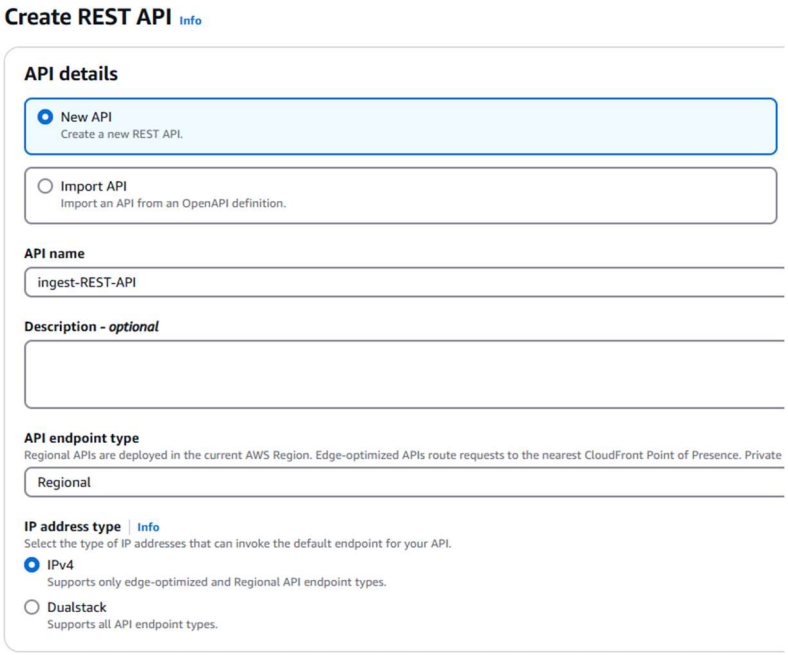
I started with drawing out a flow chart by hand after I researched a basic architecture and included as much as I thought I would need. Some modification was necessary as will be seen in the following methodology.



1. Data Ingestion

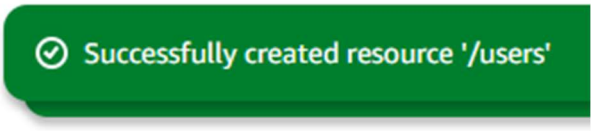
- Amazon API Gateway
  → Use for capturing incoming customer service document data from outside stakeholder





- Create REST API Gateway instance

Create a resource (resource path (URL Endpoint defined by the user) and resource name (“users”))

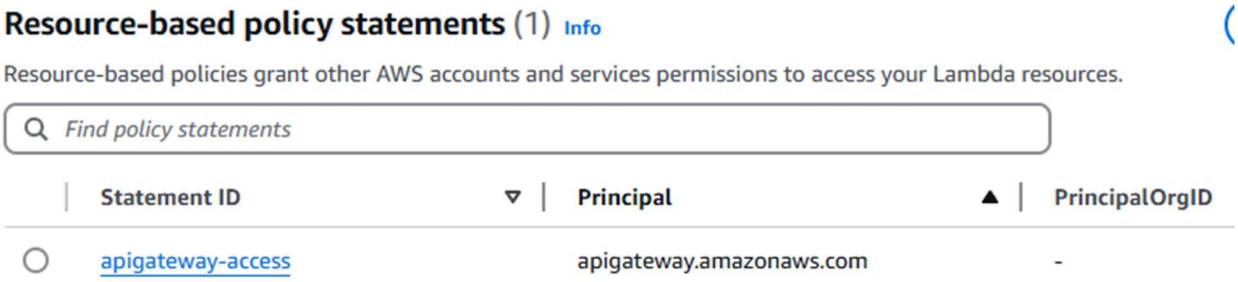


- Create a GET method with Lambda as integration type

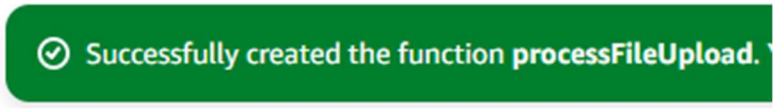
HTTP Request Headers for the two file types

- Create Lambda function

Add permissions for API Gateway to invoke it by using a Resource-based policy permission (“apigateway-access”):



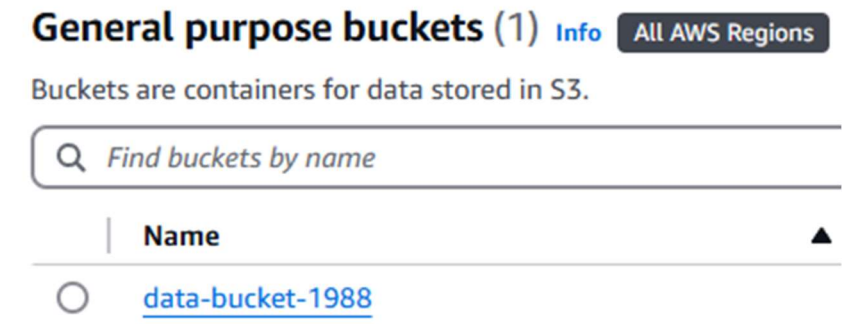
*Able to use CloudWatch for debugging from here



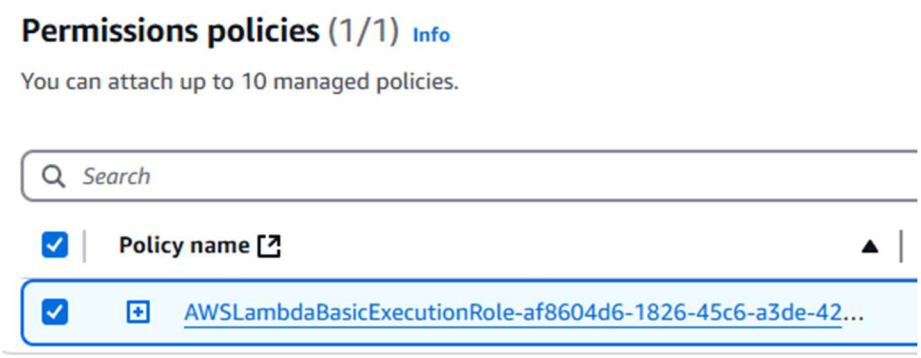
- Triggered when new messages arrive

2. Data Storage

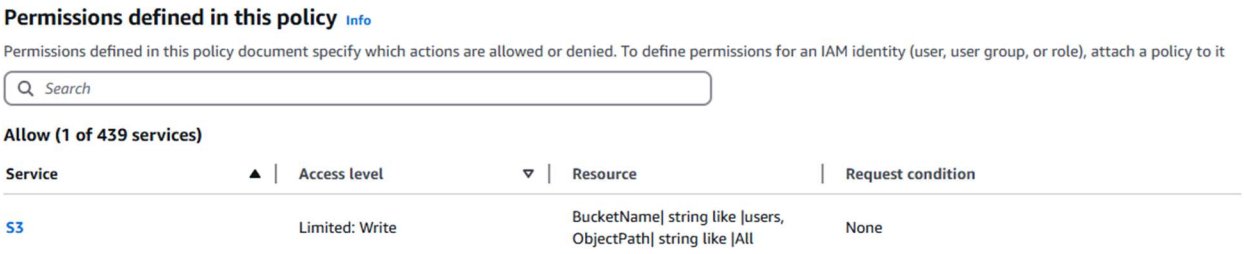
- Create an s3 bucket for raw storage of files (“data-bucket-1988”)



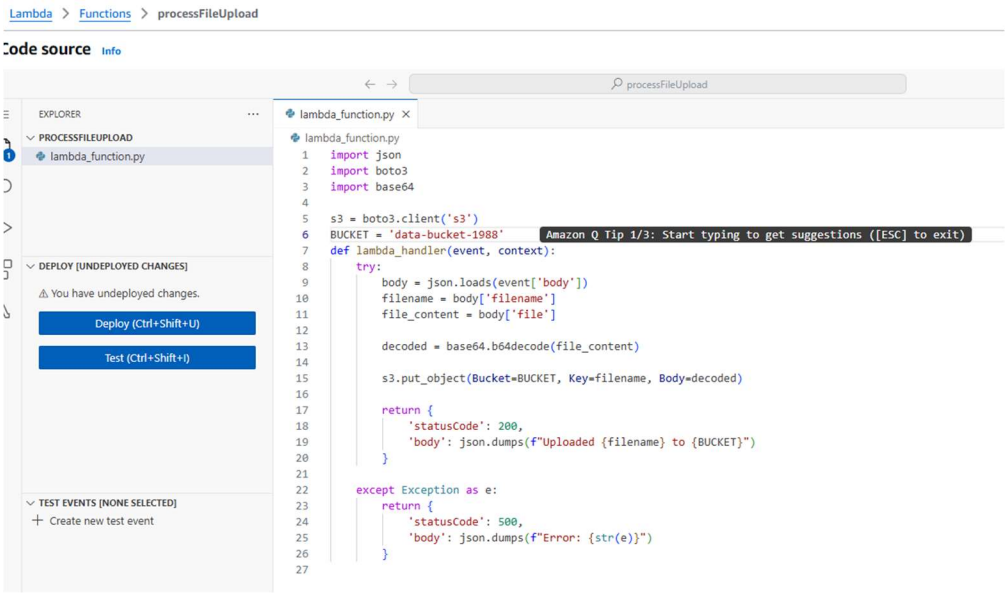
Connect the S3 bucket to lambda function by adding permission to upload in IAM



- Configure policy and specify permissions

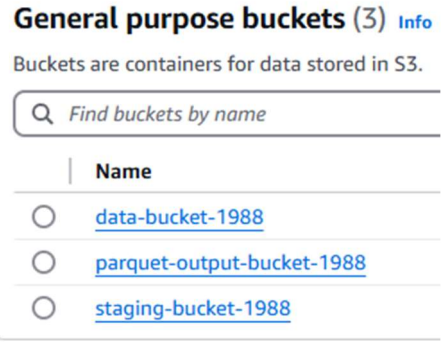


Add code for to move the files using the Lambda function to S3 raw storage data-bucket-1988

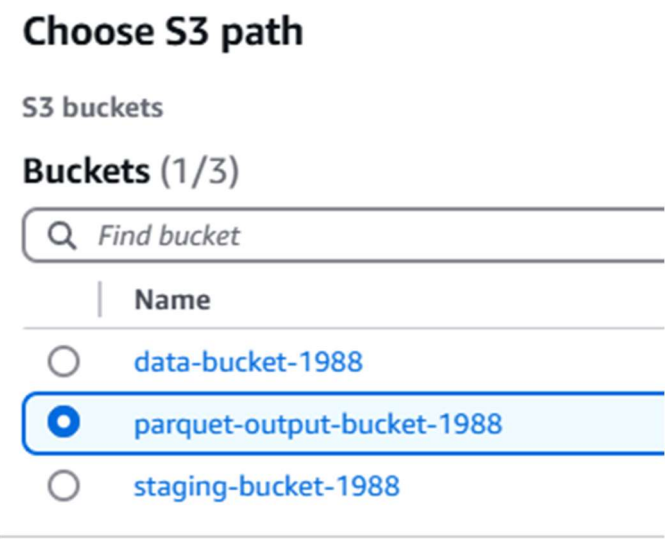


- Set trigger function for checkANDStageCsv lambda function to trigger the data-bucket-1988 PUT event.

Create staging and parquet file (“staging-file-1988” and “parquet-output-bucket-1988 buckets for csv file ETL process storage

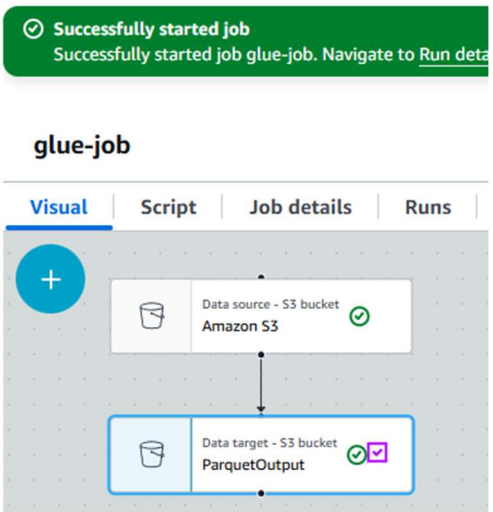


Use AWS Glue to change data from CSV to Parquet. Create a crawler for the Parquet outputs (“csv-staging-crawler”) and specify the path to the output bucket.



3. Extract, Transform, Load (ETL)

- Create a new IAM role for this. Create a Glue Data Catalog database for the table (“etl_data”).
- Create an ETL job using AWS Glue Studio, and configure the visual studio with a workflow.
- Configure the transform path to write from CSV to Parquet.



4. Query & Reporting

- Amazon Athena *(Free: 1TB scanned per month)*
  → Run SQL queries on data stored in S3 (must convert to partitioned format for efficiency).

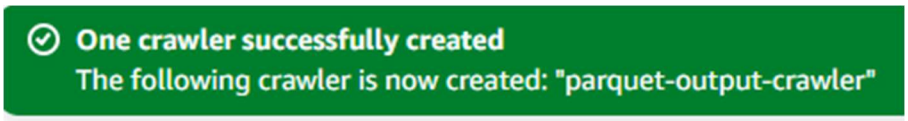
Add another S3 bucket for query results ("s3://parquet-output-bucket-1988/athena-results/"). Configure Athena to be able to query over the Parquet files and put the query results in the results bucket.

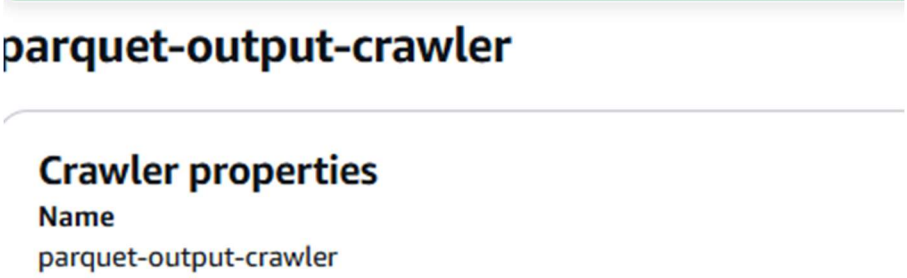
Set up another Glue crawler for Athena to be used effectively ("parquet-output-crawler"). It scans the Parquet files in parquet-output-bucket-1988/.

The crawler automatically:

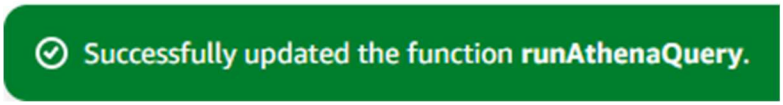
- Reads the file format (Parquet in your case)
- Detects the schema (column names, data types)
- Handles partitions if you use them
- Catalogs Parquet files into a table Glue can read from to do ETL

The purpose of changing the csv files into parquet files is that it will be a more efficient SQL data structure to query from. It will know exactly what column to go to rather than having to run through the whole table when doing a query (as would be the case with Aurora).





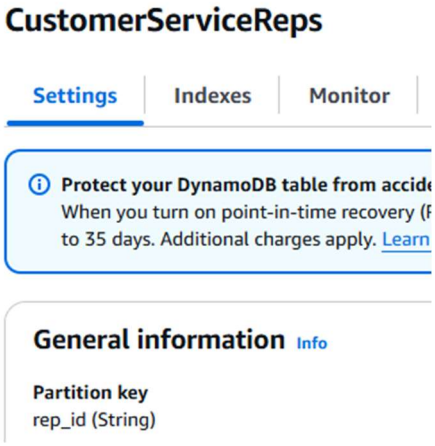
- Create new Lambda function ("runAthenaQuery") and add code for the function to be triggered by S3 when a new file is uploaded, checks if the file is a csv file and copied the file the S3 bucket staging-bucket-1988
- Change permission, create new role with basic Lambda permissions:



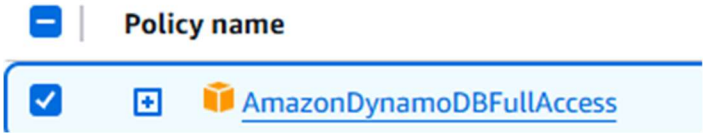
DynamoDB Stage

DynamoDB: NoSQL database

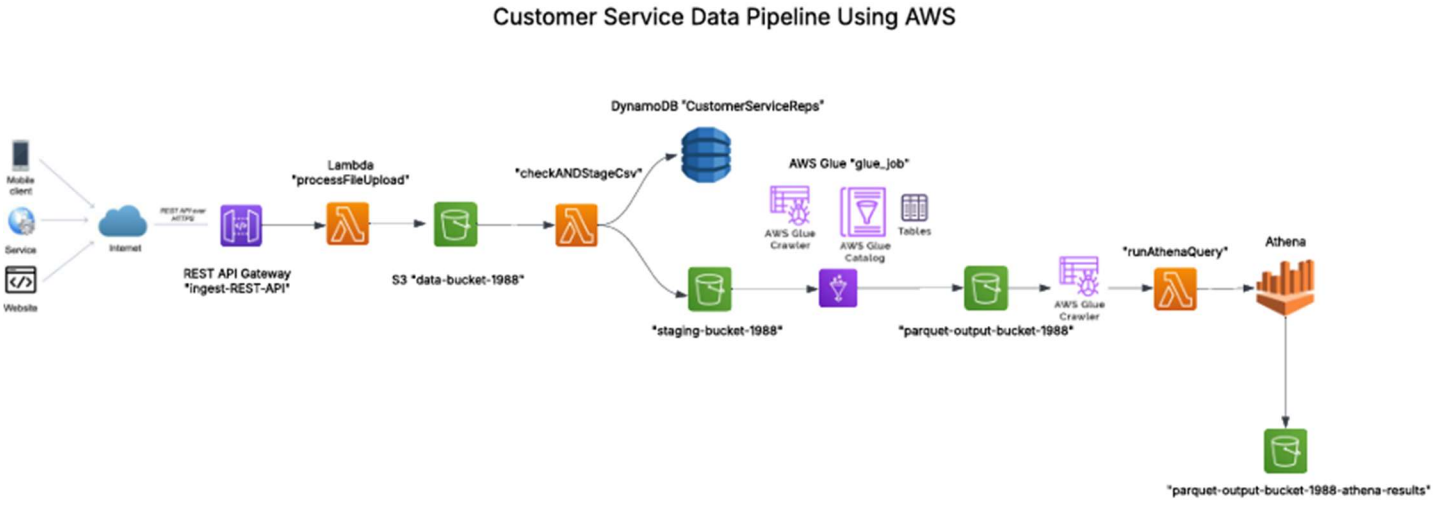
- Create a DynamoDB table (“CustomerServiceReps”) with partition key (“rep_id” in our json files) and set table schema:



- Modify the lambda function checkandstagecsv to check for json files and put them in the dynamodb NoSQL database.
- Configure the Lambda’s IAM role “lambda-role-checkANDStageCsv” and add full access as a permission:



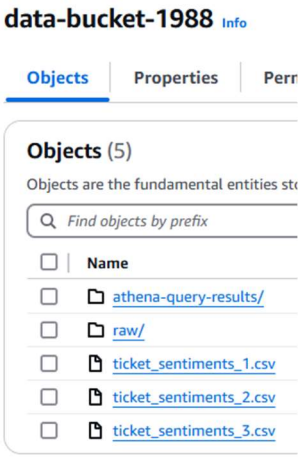
Flow Chart:



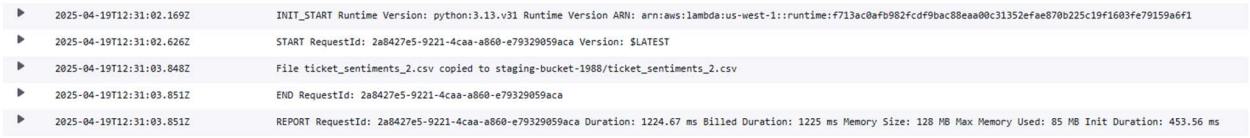
Results

The first step after building this cloud architecture is to test it. During testing, I did a lot of troubleshooting, such as reconfiguring scripts, adjusting IAM role permissions, and reloading files for testing. The following is a workflow for the data pipeline from data ingestion to the end of the csv file workflow. It will be followed by the json file workflow from the checkANDStagecsv lambda function that separates the json files and sends them to the DynamoDB database:

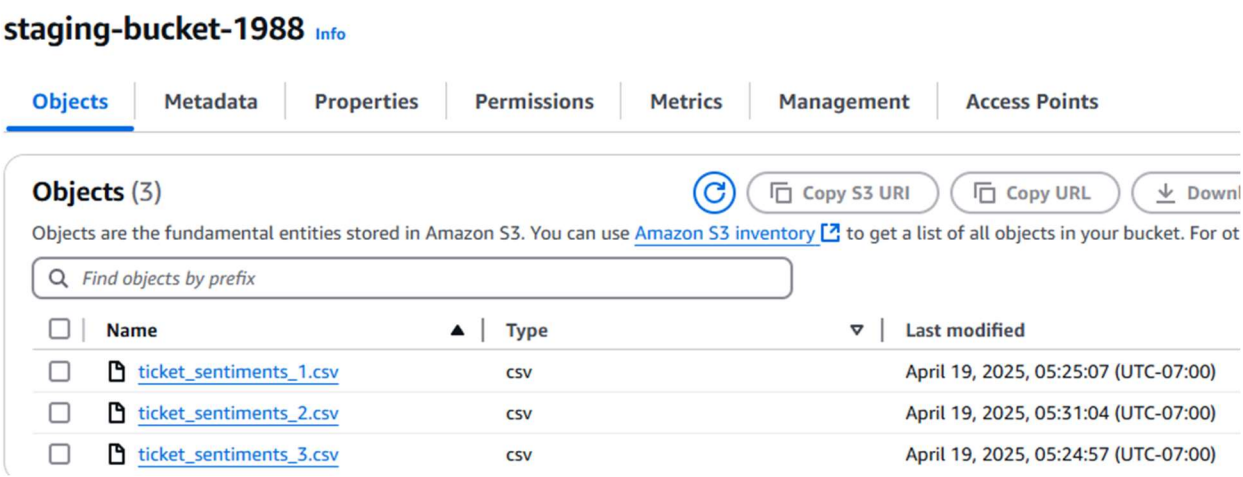
✅ Start by uploading a .csv file to the API Gateway endpoint and make sure it is in my first s3 raw storage bucket by using curl shell command. They are uploaded to the first S3 bucket:



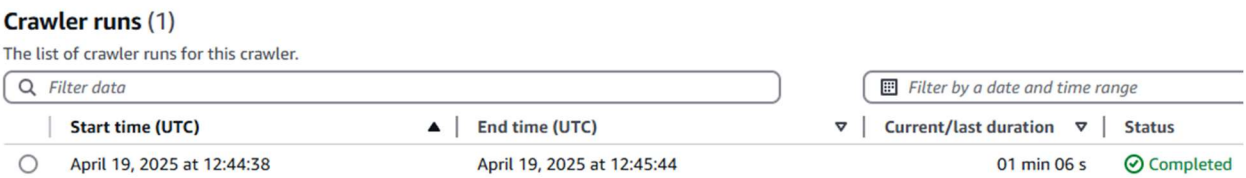
✅ And verifying the log in CloudWatch to see if this flow of the pipeline worked



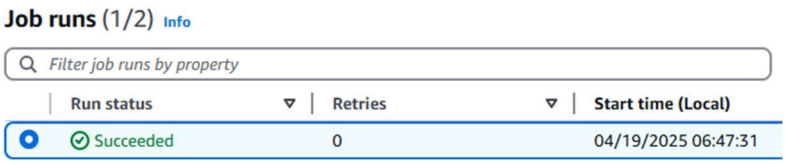
✅ Now it's in staging-bucket-1988:



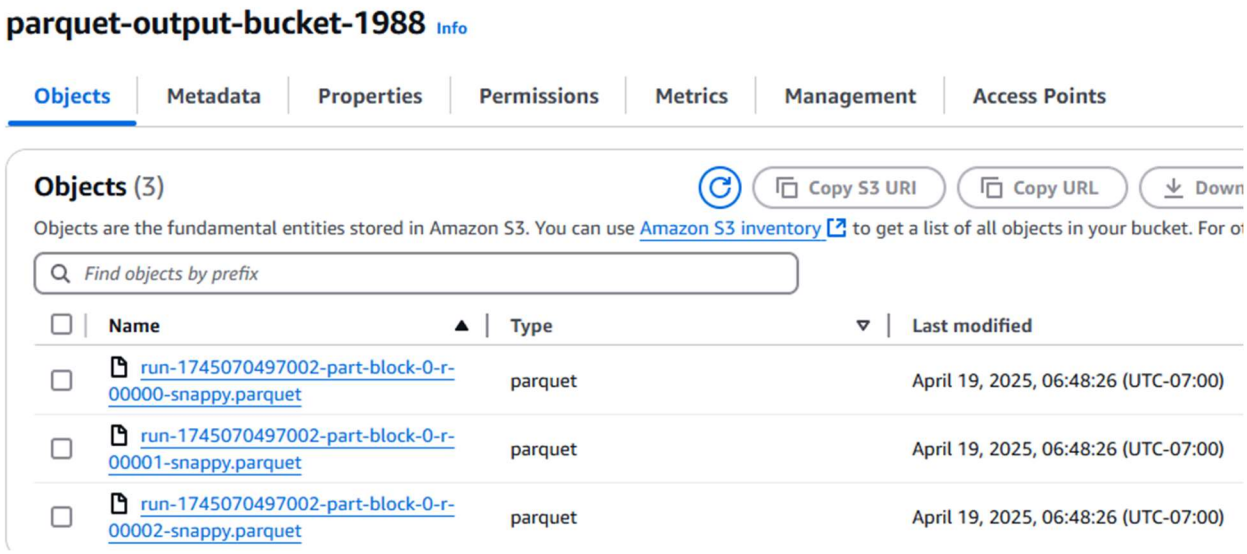
This staging isn't automated. I had to run the Glue crawler:



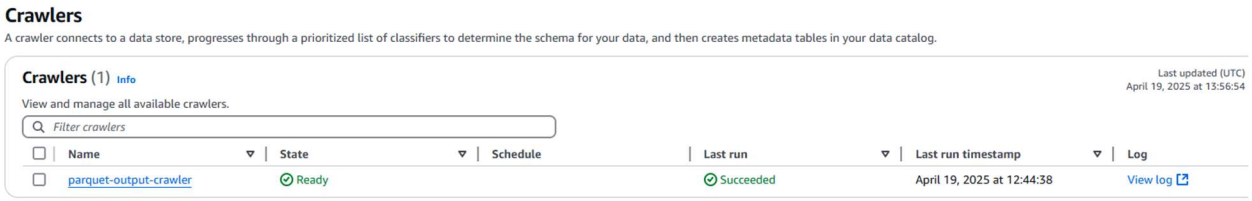
✅ Check Glue Data Catalog to verify ETL job:



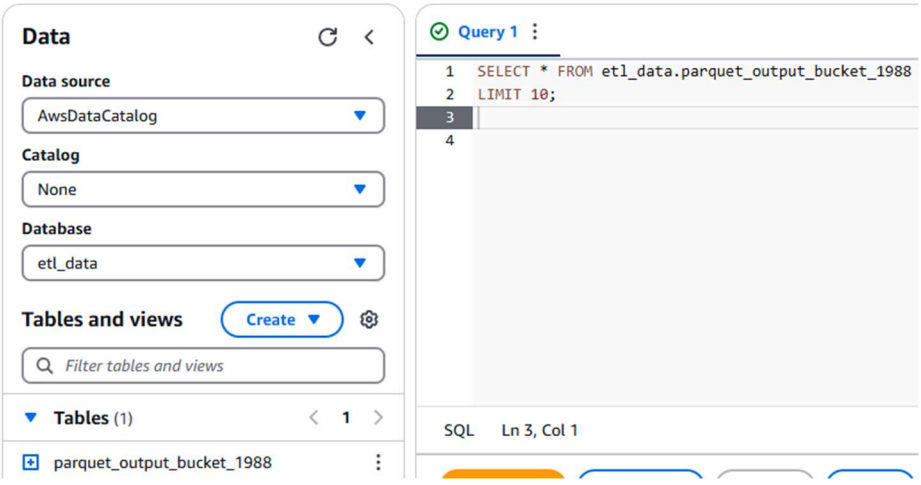
✅ Check the next S3 bucket for the parquet files:



✅ Check the Next Glue Crawler to see that the parquet files were worked enacted on the create a table schema in Glue Data Catalog:



✅ Test the Athena Query service by running a SQL query on a parquet file, ensuring that the etl_data database where the parquet files are located has the bucket access:



Query results | Query stats

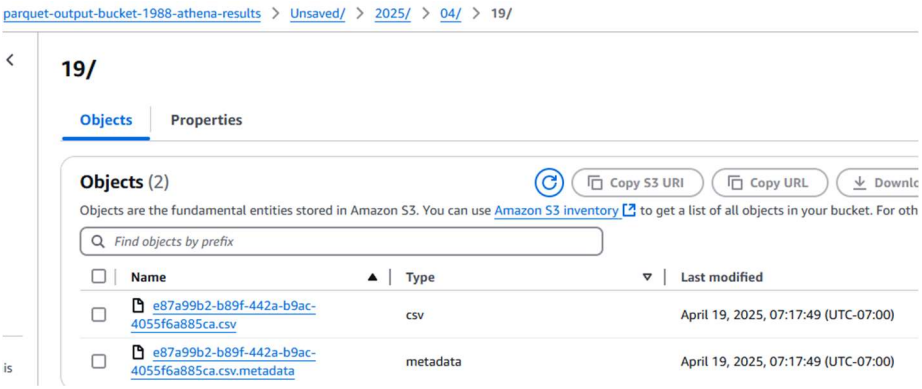
Completed

Results (10)

Search rows

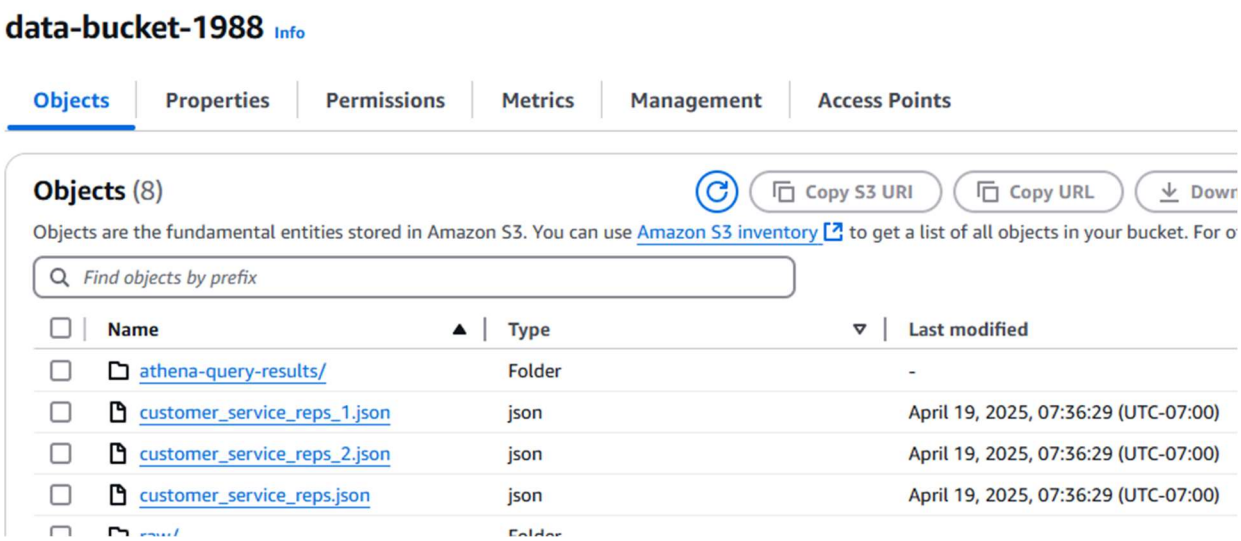
#	ticket_id	sentiment_score
1	TKT3000	-0.9
2	TKT3001	0.66
3	TKT3002	-0.02
4	TKT3003	0.6
5	TKT3004	0.63
6	TKT3005	-0.38
7	TKT3006	-0.53
8	TKT3007	-0.79
9	TKT3008	0.9
10	TKT3009	0.07

✅Check the final S3 bucket to see if the Athena query output results were stored there:

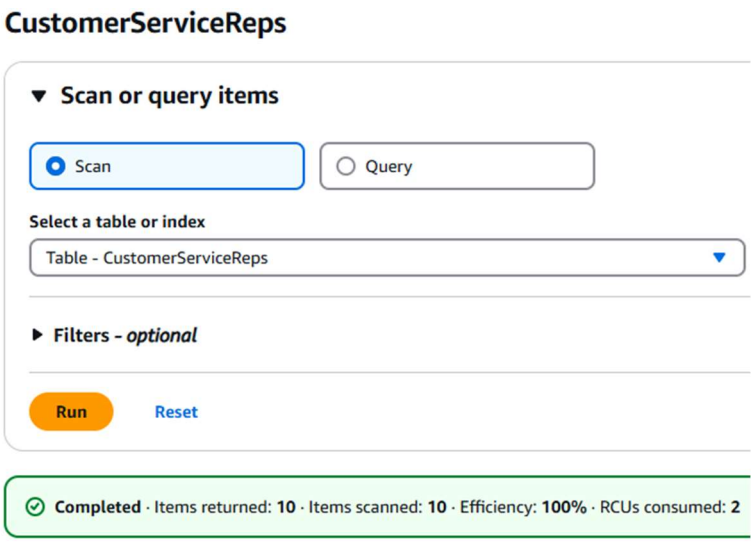


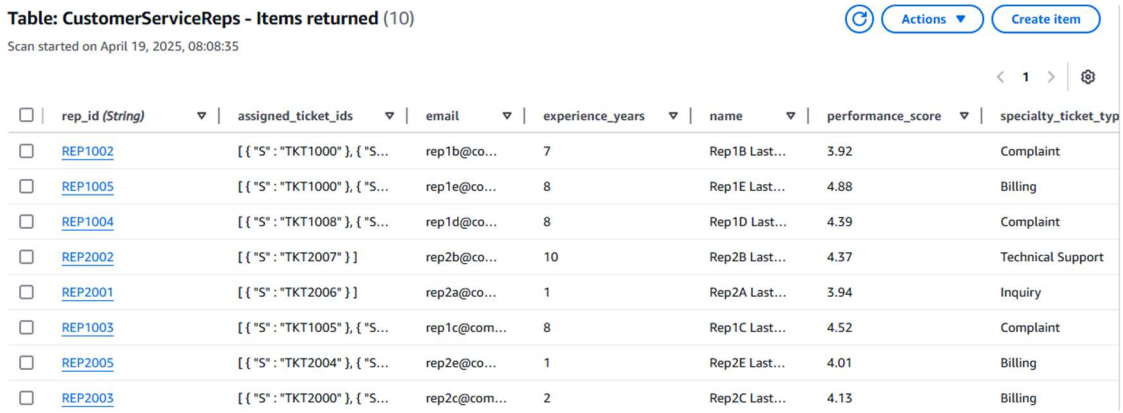
DynamoDB workflow:

✅Json file ingested into data uploaded from data-bucket-1988 S3 bucket:



✅ Check DynamoDB to see if the files were uploaded:





Discussion

This project used:

Data Types and Sources: csv, json, and parquet are different data types that need to be processed differently for use.

- **CSV files** are structured, tabular data often used for exporting records from business systems.
- **JSON files** are semi-structured data, commonly used for API-based communication or nested records.
- **Parquet** is a columnar, compressed file format ideal for analytics.

Cloud Computing: providing resources from AWS, a technology that is the most relevant to today's cloud standards and is the largest cloud provider in the world. AWS scales with demand and provides reliability and durability. The price structure is pay-as-you-go, so it offers a good opportunity for a company of any size to automate in this way.

Data Lake: Amazon S3 is a data lake. It acts as a central repository for this data pipeline. There are four S3 buckets in this architecture serving numerous functions: raw injection "data-bucket-1988", staging for ETL ("staging-bucket-1988"), analytics-ready data lake ("parquest-output-bucket-1988"), and SQL query results ("parquet-output-bucket-1988-athena-results")

Data Pipeline: the architecture for this project. Ingestion-Transform-Storage-Query

Ingest and Storage: API Gateway accepts incoming data (or can be extended to do so) as a REST endpoint CSV files are stored in S3 and transformed into analytics-ready Parquet files. Processing and Analytics: ETL for better analytics. Lambda functions trigger a movement of data through the pipeline from the API gateway and storage buckets.

Further possible developments:

*Connect QuickSight or a BI tool to Athena for dashboarding

*Add batch functionality to enable scheduled, bulk, or event-driven processing of multiple files, instead of processing each one instantly

*Add aggregate storage for repeated reports to be readied for machine learning tasks and model building. DynamoDB can be used by AWS Sagemaker for real-time prediction. DynamoDB benefits from low-latency reads and writes.

Impression

AWS console was quite easy to pick up because I am already familiar with the foundations of AWS. I hadn't really deployed much except in a controlled beginner environment, so this was a good hands-on exercise. The other way I planned this project was with a flow chart. They are commonly used in cloud architecture, so I just drafted one and reworked it as the project structure became clearer and solid with reading, Q&A, experimenting. It's confusing for totally new users but is easy to navigate with guidance as a novice, especially if you know what services you are supposed to use and how to configure them.

Conclusion

This project effectively implements a data pipeline that ingests and processes two file formats—CSV and JSON—for storage and analysis. CSV files are transformed into Parquet format, enabling efficient SQL querying and result storage. JSON files are routed to a NoSQL database (DynamoDB) for downstream use. The pipeline architecture leverages AWS Lambda functions written in Python to automate file flow, with Amazon S3 serving as the intermediate storage layer. The system is accessed via an API Gateway, with final data endpoints in DynamoDB and an S3 bucket. Designed for use by a business team, the solution supports seamless document submission, transformation, and analysis.

References

Amazon Web Services. *Analyze Data in Amazon DynamoDB Using Amazon SageMaker for Real-Time Prediction*. AWS Big Data Blog. https://aws.amazon.com/blogs/big-data/analyze-data-in-amazon-dynamodb-using-amazon-sagemaker-for-real-time-prediction/.

Amazon Web Services. *AWS Management Console*. https://us-east-1.console.aws.amazon.com/console/home?region=us-east-1.

Amazon Web Services. https://aws.amazon.com.

ChatGPT. OpenAI. https://chatgpt.com.

Lucid Chart. https://lucidchart.com.

Technology Magazine. *Top 10 Biggest Cloud Providers in the World in 2023*. https://technologymagazine.com/top10/top-10-biggest-cloud-providers-in-the-world-in-2023.

Appendix A: Lambda Function Python Scripts

processFileUpload

```
import boto3
import os
import urllib.parse

s3 = boto3.client('s3')

DEST_BUCKET = 'staging-bucket-1988'

def lambda_handler(event, context):
  try:
    # Get the object details from the S3 event
    source_bucket = event['Records'][0]['s3']['bucket']['name']
    object_key = urllib.parse.unquote_plus(event['Records'][0]['s3']['object']['key'])

    # Check if it's a CSV file
    if object_key.lower().endswith('.csv'):
      # Construct copy source
      copy_source = {
        'Bucket': source_bucket,
        'Key': object_key
      }

      # Define destination key (you can modify the folder structure here if desired)
      dest_key = f"{object_key}"

      # Perform the copy
      s3.copy_object(Bucket=DEST_BUCKET, Key=dest_key, CopySource=copy_source)

      print(f"File {object_key} copied to {DEST_BUCKET}/{dest_key}")
    else:
      print(f"Skipped non-CSV file: {object_key}")

  except Exception as e:
    print(f"Error: {str(e)}")
    raise e
```

checkANDStageCsv

```
import json
import boto3
import os
import urllib.parse
from decimal import Decimal
```

```
s3 = boto3.client('s3')
dynamodb = boto3.resource('dynamodb')

DEST_BUCKET = 'staging-bucket-1988'
DDB_TABLE_NAME = 'CustomerServiceReps'

def lambda_handler(event, context):
  try:
    source_bucket = event['Records'][0]['s3']['bucket']['name']
    object_key = urllib.parse.unquote_plus(event['Records'][0]['s3']['object']['key'])

    if object_key.lower().endswith('.csv'):
      s3.copy_object(
        Bucket=DEST_BUCKET,
        Key=object_key,
        CopySource={'Bucket': source_bucket, 'Key': object_key}
      )
      print(f"File {object_key} copied to {DEST_BUCKET}/{object_key}")

    elif object_key.lower().endswith('.json'):
      response = s3.get_object(Bucket=source_bucket, Key=object_key)
      content = response['Body'].read().decode('utf-8')

      # Convert floats to Decimal
      data = json.loads(content, parse_float=Decimal)

      table = dynamodb.Table(DDB_TABLE_NAME)
      if isinstance(data, list):
        for item in data:
          table.put_item(Item=item)
      else:
        table.put_item(Item=data)

      print(f"Inserted JSON data into DynamoDB from {object_key}")

    else:
      print(f"Skipped unsupported file type: {object_key}")

  except Exception as e:
    print(f"Error: {str(e)}")
    raise e
```

runAthenaQuery

```
import boto3
import time

ATHENA_DB = 'etl_data'  # change if different
ATHENA_QUERY = 'SELECT * FROM your_table_name LIMIT 10;'  # customize
ATHENA_OUTPUT_BUCKET = 's3://parquet-output-bucket-1988-athena-results/'  # must end with /
```

```
def lambda_handler(event, context):
  client = boto3.client('athena')

  response = client.start_query_execution(
    QueryString=ATHENA_QUERY,
    QueryExecutionContext={
      'Database': ATHENA_DB
    },
    ResultConfiguration={
      'OutputLocation': ATHENA_OUTPUT_BUCKET
    }
  )

  query_execution_id = response['QueryExecutionId']
  print(f'Started Athena query: {query_execution_id}')

  # Optional: wait for result to complete (useful for short queries)
  while True:
    status = client.get_query_execution(QueryExecutionId=query_execution_id)['QueryExecution']['Status']['State']
    if status in ['SUCCEEDED', 'FAILED', 'CANCELLED']:
      break
    print("Query still running...")
    time.sleep(2)

  print(f'Query {status}. Results at: {ATHENA_OUTPUT_BUCKET}{query_execution_id}.csv')

  return {
    'statusCode': 200,
    'body': f'Query {status}. Execution ID: {query_execution_id}'
  }
```

Appendix B: Glue Job Python script

```
import sys
from awsglue.transforms import *
from awsglue.utils import getResolvedOptions
from pyspark.context import SparkContext
from awsglue.context import GlueContext
from awsglue.job import Job
from awsgluedq.transforms import EvaluateDataQuality

args = getResolvedOptions(sys.argv, ['JOB_NAME'])
sc = SparkContext()
glueContext = GlueContext(sc)
spark = glueContext.spark_session
job = Job(glueContext)
job.init(args['JOB_NAME'], args)
```

```
# Default ruleset used by all target nodes with data quality enabled
DEFAULT_DATA_QUALITY_RULESET = """
  Rules = [
    ColumnCount > 0
  ]
"""

# Script generated for node Amazon S3
AmazonS3_node1744568210464 = glueContext.create_dynamic_frame.from_options(
  connection_type="s3",
  connection_options={
    "paths": ["s3://staging-bucket-1988/"],
    "recurse": True
  },
  format="csv",
  format_options={
    "withHeader": True,
    "separator": ",",
    "quoteChar": "\""
  },
  transformation_ctx="AmazonS3_node1744568210464"
)


# Script generated for node ParquetOutput
EvaluateDataQuality().process_rows(frame=AmazonS3_node1744568210464,
ruleset=DEFAULT_DATA_QUALITY_RULESET, publishing_options={"dataQualityEvaluationContext":
"EvaluateDataQuality_node1744568100171", "enableDataQualityResultsPublishing": True},
additional_options={"dataQualityResultsPublishing.strategy": "BEST_EFFORT", "observations.scope": "ALL"})
ParquetOutput_node1744568749168 =
glueContext.write_dynamic_frame.from_options(frame=AmazonS3_node1744568210464, connection_type="s3",
format="glueparquet", connection_options={"path": "s3://parquet-output-bucket-1988/", "partitionKeys": []},
format_options={"compression": "snappy"}, transformation_ctx="ParquetOutput_node1744568749168")

job.commit()
```