**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**

* customer\_support\_tickets.csv: Contains customer profile data and ticket information with ticket\_id as the primary key.
* ticket\_sentiments.csv: Provides sentiment scores mapped to ticket IDs. Useful for merging with support tickets.
* customer\_service\_reps (JSON format): Contains support representative details with rep\_id as the key. Includes nested lists such as assigned\_ticket\_ids.

**Results**

**Methodology Hands On: Notes and Pictures from Build**

**Updated Services & Flow**

1. Data Ingestion

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

A diagram of a computer application

AI-generated content may be incorrect.

A screenshot of a computer

AI-generated content may be incorrect.

* Create REST API Gateway instance

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

A green rectangle with white text

AI-generated content may be incorrect.

* 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”):

A screenshot of a computer

AI-generated content may be incorrect.

\*Able to use CloudWatch for debugging from here

A green rectangle with white text

AI-generated content may be incorrect.

* Triggered when new messages arrive

2. Data Storage

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

A screenshot of a computer

AI-generated content may be incorrect.

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

A screenshot of a computer

AI-generated content may be incorrect.

Configure policy and specify permissions

A screenshot of a computer

AI-generated content may be incorrect.

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

A screenshot of a computer

AI-generated content may be incorrect.

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

A screenshot of a computer

AI-generated content may be incorrect.

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.

A screenshot of a computer

AI-generated content may be incorrect.

* 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.

A screenshot of a computer

AI-generated content may be incorrect.

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).

A screenshot of a computer

AI-generated content may be incorrect.

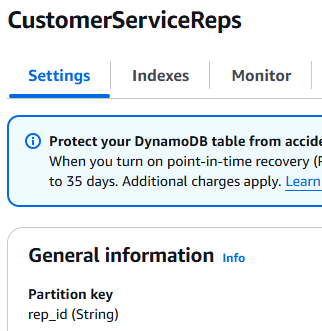
* Create new Lambda function (“runAthenaQuery”) and add code for the function to be triggered by S3 when a new file is uploaded, checks if the file is a csv file and copied the file the S3 bucket staging-bucket-1988
* Change permission, create new role with basic Lambda permissions:

A green rectangle with white text

AI-generated content may be incorrect.

**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:

A close-up of a logo

AI-generated content may be incorrect.

**Flow Chart:**

**A computer screen shot of a diagram

AI-generated content may be incorrect.**

**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 endpointCSV files are stored in S3 and transformed into analytics-ready Parquet files. Processing and Analytics: ETL for better analytics

**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**

**References**

**Amazon Web Services. “AWS Management Console.”** [**https://us-east-1.console.aws.amazon.com/console/home?region=us-east-1**](https://us-east-1.console.aws.amazon.com/console/home?region=us-east-1)**.**

**Amazon Web Services.** [**https://aws.amazon.com**](https://aws.amazon.com)**.**

**ChatGPT. OpenAI.** [**https://chatgpt.com**](https://chatgpt.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**](https://technologymagazine.com/top10/top-10-biggest-cloud-providers-in-the-world-in-2023)**.**

**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/**](https://aws.amazon.com/blogs/big-data/analyze-data-in-amazon-dynamodb-using-amazon-sagemaker-for-real-time-prediction/)**.**

**Appendix**: 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 urllib.parse

s3 = boto3.client('s3')

dynamodb = boto3.resource('dynamodb')

DEST\_BUCKET = 'staging-bucket-1988'

DYNAMO\_TABLE\_NAME = 'CustomerServiceReps'

def lambda\_handler(event, context):

    try:

        for record in event['Records']:

            src\_bucket = record['s3']['bucket']['name']

            src\_key = urllib.parse.unquote\_plus(record['s3']['object']['key'])

            print(f"Received file: {src\_key} from {src\_bucket}")

            # Handle .csv files → copy to staging

            if src\_key.lower().endswith('.csv'):

                copy\_source = {'Bucket': src\_bucket, 'Key': src\_key}

                s3.copy\_object(Bucket=DEST\_BUCKET, Key=src\_key, CopySource=copy\_source)

                print(f"Copied {src\_key} to {DEST\_BUCKET}")

            # Handle .json files → insert into DynamoDB

            elif src\_key.lower().endswith('.json'):

                response = s3.get\_object(Bucket=src\_bucket, Key=src\_key)

                content = response['Body'].read().decode('utf-8')

                json\_data = json.loads(content)

                table = dynamodb.Table(DYNAMO\_TABLE\_NAME)

                if isinstance(json\_data, list):

                    with table.batch\_writer() as batch:

                        for item in json\_data:

                            if 'rep\_id' in item:

                                batch.put\_item(Item=item)

                            else:

                                print(f"Missing 'rep\_id' in: {item}")

                elif isinstance(json\_data, dict):

                    if 'rep\_id' in json\_data:

                        table.put\_item(Item=json\_data)

                    else:

                        print(f"Missing 'rep\_id' in: {json\_data}")

                else:

                    print("Unsupported JSON format")

                print(f"Inserted data from {src\_key} into DynamoDB")

            else:

                print(f"Skipped unsupported file type: {src\_key}")

        return {

            'statusCode': 200,

            'body': json.dumps('Processed S3 event successfully.')

        }

    except Exception as e:

        print(f"Error: {str(e)}")

        return {

            'statusCode': 500,

            'body': json.dumps(f"Error: {str(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}'

    }