UNIVERSITY OF CALGARY

Analyzing Traffic Incidents in Calgary, Alberta

Jaden Der, Poonam Phagura, and James Ding

Developing Big Data Applications

Data 608 Summer 2024

Final Project

Presented to Dr. Usman Alim

Contents

Section 1: Introduction, Problem statement, and concise summary of the results

Introduction:

The City of Calgary is one of Canada's fastest-growing municipalities, driven largely by a significant influx of interprovincial migration. Many people are relocating to Calgary from other provinces, particularly from cities where the cost of living has become increasingly burdensome. Calgary's appeal lies in its blend of economic opportunity and relatively affordable living, attracting those seeking a better balance between career prospects and quality of life.

As Calgary's population continues to swell, the city is experiencing increasing urban density, which presents both opportunities and challenges. One of the most pressing concerns is the impact on the transportation infrastructure. As more people move to the city, the demand on Calgary's road networks will intensify. Over time, this is likely to lead to greater congestion, longer commute times, and more frequent delays, especially during peak hours. The growing strain on transportation systems underscores the need for strategic urban planning and investment in public transit and road improvements to accommodate the city's rapid expansion while maintaining the quality of life that draws so many new residents. To analyze this ongoing issue, we will be utilizing the Traffic Incident dataset found on the city’s open data portal.

Problem Statement:

The project aims to analyze traffic incidents in Calgary using AWS services to automate data ingestion, storage, and processing, and finally visualize the results using Streamlit. The objective is to create a seamless pipeline that handles data from its source to a public-facing dashboard, allowing users to understand traffic patterns and incidents more effectively.

Summary of Results

The report outlines the data engineering lifecycle for a project involving traffic incident data. It begins with the collection of unrecorded traffic camera data, followed by data cleaning and processing using AWS services like Lambda and Glue. Data ingestion is automated through an EC2-hosted Python script, scheduled via cron jobs, and organized into structured datasets in S3. The data undergoes transformation using AWS Glue ETL jobs before being stored in an S3 bucket and cataloged for analysis. The processed data is then served to users through a Streamlit application, providing interactive visualizations of traffic incidents. The report highlights the successful automation of the data pipeline, ensuring efficient data management and delivery, and discusses limitations, potential improvements, and future steps such as real-time data processing and advanced visualizations. The project demonstrates the effectiveness of AWS services in creating a scalable and user-friendly data pipeline while emphasizing the importance of automation and the challenges of integrating multiple AWS services.

Section 2: Data engineering lifecycle

In this project, the data engineering lifecycle starts with the collection of raw data, particularly incidents captured from unrecorded traffic camera footage. In our case this footage is monitored in real time but not stored for later use. Once gathered, the data will undergo cleaning and processing, utilizing Lambda functions and AWS Glue to ensure accuracy and consistency. After processing, the data will be transformed and prepared for analysis, ultimately being served to end users through a Streamlit application. This process should become clearer as we progress through this report.

Section 2.0: Pipeline diagram

The following is the diagram showing the flow from data ingestion through processing and storage to serving.

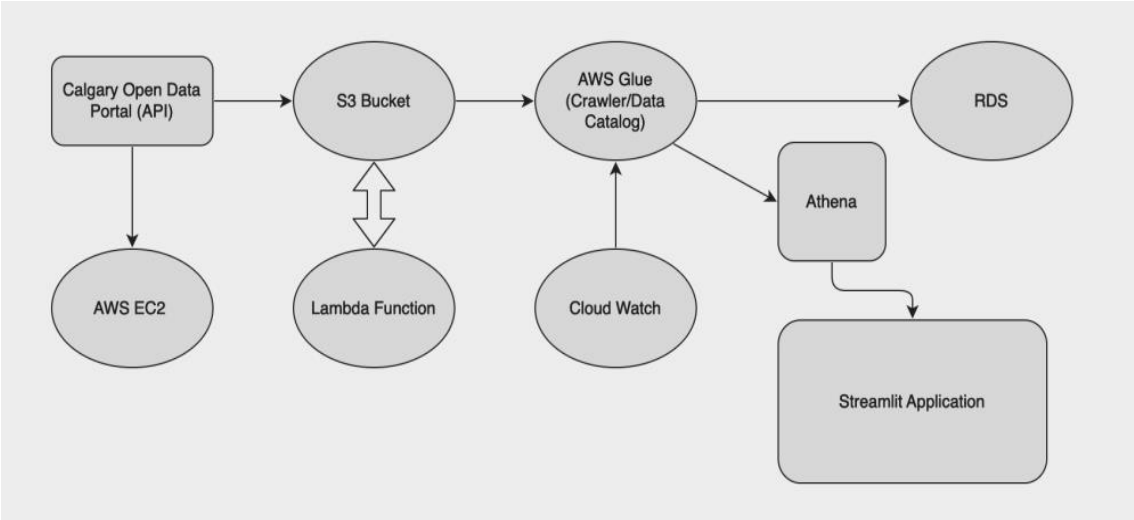


Figure 1: Flowchart of data pipeline

Section 2.1: Data generation

When we start to build on the cloud we'll have to manage access to services. In AWS, this is achieved through IAM roles. Unfortunately, we are not able to create IAM roles in Learner Lab so we have to use one created for us. It is called “LabRole”, and we need to assign this role to our EC2 instance so that it can access our S3 object storage.

From there you should see some options, select LabInstanceProfile

Click *Update*

Now your EC2 instance can access other AWS services that have the same IAM role

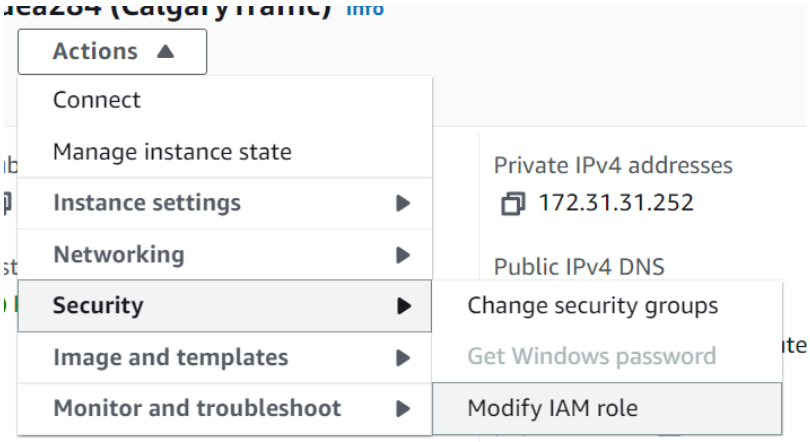


Figure 2: screen capture of IAM role in AWS

The following is installed on the EC2 instance before the data ingestion/generation starts:

1. sudo snap install aws-cli –classic

Data Source

The data is generated from the API on City of Calgary's open data portal, providing the bulk of traffic incident reports in CSV format.

Type of Data

The data is structured, containing fields such as incident description, start date, location coordinates, and unique identifiers for each incident. The first few rows of our main dataset are provided below.

	INCIDENT INFO	DESCRIPTION	START_DT	MODIFIED_DT	QUADRANT	Longitude	Latitude	Count	id	Point
0	Westbound 16 Avenue at Deerfoot Trail NE	Stalled vehicle. Partially blocking the right...	2022/06/21 07:31:40 AM	2022/06/21 07:33:16 AM	NE	-114.026687	51.067485	1	2022-06-21T07:31:4051.067485129276236-114.0266...	POINT (-114.02668672232672 51.067485129276236)
1	11 Avenue and 4 Street SW	Traffic incident. Blocking multiple lanes	2022/06/21 04:02:11 AM	2022/06/21 04:12:38 AM	SW	-114.071481	51.042624	1	2022-06-21T04:02:1151.042624492614 62-114.07148...	POINT (-114.07148057660925 51.042624492614 62)
2	68 Street and Memorial Drive E	Traffic incident.	2022/06/20 11:53:08 PM	2022/06/20 11:55:42 PM	NE	-113.935553	51.052474	1	2022-06-20T23:53:0851.0524735056658-113.935553...	POINT (-113.9355533257 51 51.0524735056658)
3	Eastbound 16 Avenue and 36 Street NE	Traffic incident. Blocking the left shoulder	2022/06/20 04:43:21 PM	2022/06/20 05:17:05 PM	NE	-113.989219	51.067086	1	2022-06-20T16:43:2151.06708565896752-113.98921...	POINT (-113.98921905311566 51.06708565896752)
4	Barlow Trail and 61 Avenue SE	Traffic incident.	2022/06/20 04:42:12 PM	2022/06/20 05:28:21 PM	SE	-113.985727	50.998727	1	2022-06-20T16:42:1250.99872748477766-113.98572...	POINT (-113.98572655353505 50.99872748477766)

Figure 3: Snapshot of dataset

Volume and Velocity

The data is relatively moderate in volume but can vary with traffic patterns and incidents. The cron job schedules data retrieval biweekly, ensuring that the system handles updates efficiently without overloading resources.

Section 2.2: Data ingestion

Ingestion Methods

Data is ingested via a Python script executed on an EC2 instance. The script downloads the CSV file from the public API and uploads it to an S3 bucket. A cron job ensures this process runs automatically every two weeks.

1. Create an upload.py and add the following code to the upload file. This code takes the data from API And uploads it to the S3 bucket using csv format

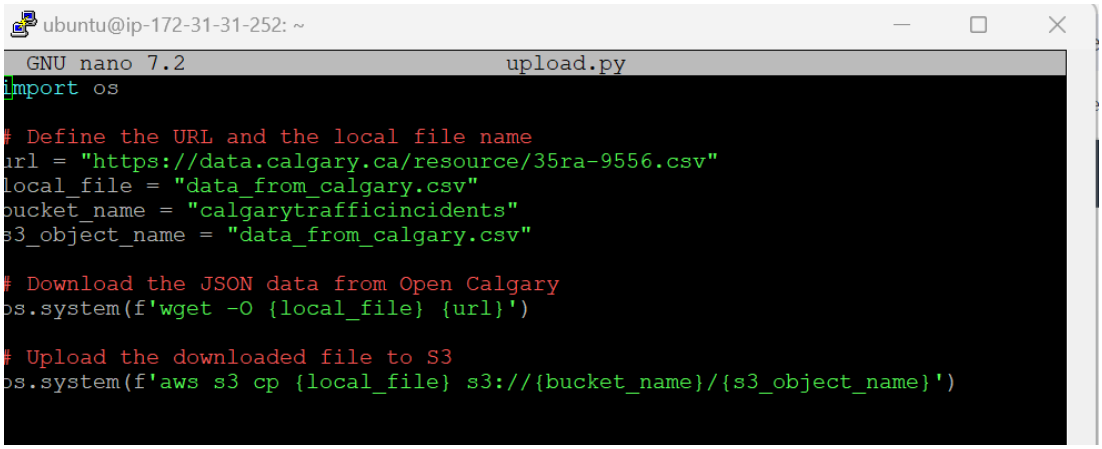


Figure 5: "upload.py"

2. Run_biweekly shell script is used to call upload.py and also log the details from the upload step.

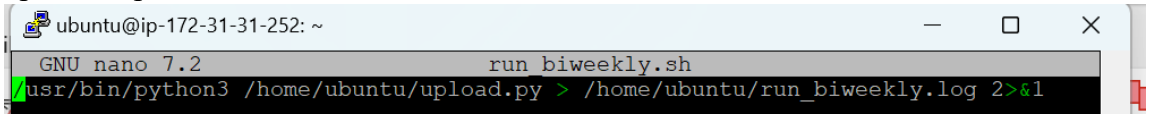


Figure 6: Executing "run_biweekly"

3. This script is automated using the crontab -e. The run_biweekly script has been scheduled to run at 5:00 PM on 1st and 15th of every month.

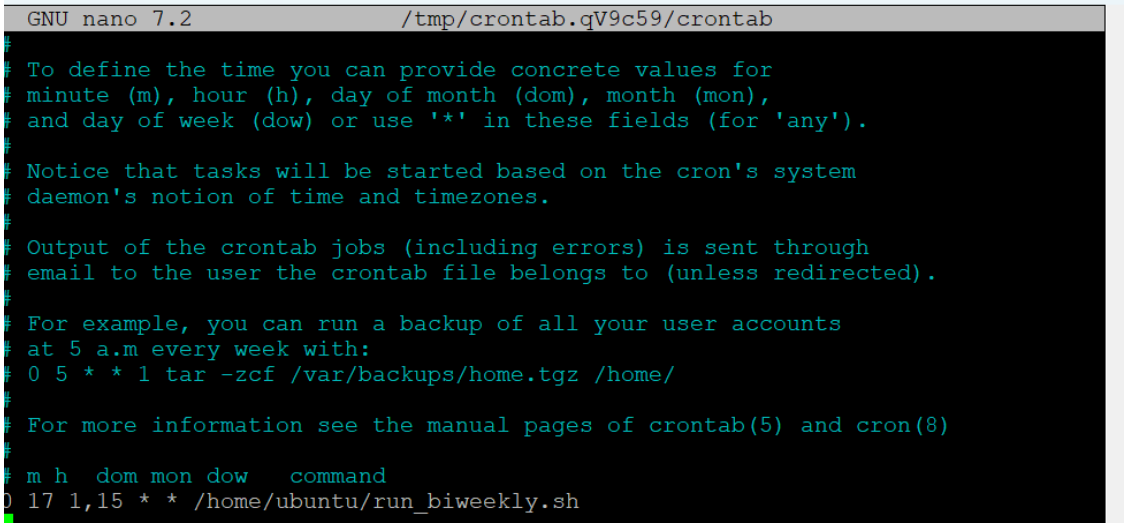


Figure 7: Scheduling script run times

Tools and Technologies

- **AWS Lambda:** Triggers processing of new data as it arrives in S3.
- **AWS CLI:** Facilitates interaction with AWS services from the EC2 instance.
- **Cron:** Schedules the biweekly execution of the ingestion script.

Docker Container Setup

To streamline the data ingestion process and ensure a consistent environment across different stages of the pipeline, we utilized Docker to create a containerized environment on the EC2 instance. This approach helps in maintaining a uniform setup and simplifies the deployment of our data ingestion scripts.

Install Docker: Docker is installed on the EC2 instance using the following commands:

1. sudo apt install docker.io
2. sudo systemctl start docker $ sudo systemctl enable docker
3. sudo usermod -aG docker $USER
4. newgrp docker

Dockerfile creation: A Dockerfile is created to define the environment, and dependencies required for the data ingestion script. This includes installing necessary Python packages and setting up any environment variables.

```
# Use the official Ubuntu base image
FROM ubuntu:24.04

# Update the package list and install necessary packages
RUN apt-get update && \
    apt-get install -y \
    python3 \
    python3-venv \
    wget \
    cron \
    automake \
    autotools-dev \
    fuse \
    g++ \
    git \
    libcurl4-openssl-dev \
    libfuse-dev \
    libssl-dev \
    libxml2-dev \
    make \
    pkg-config \
    nano && \
    apt-get clean && \
    rm -rf /var/lib/apt/lists/* /tmp/* /var/tmp/*

# Install s3fs
RUN git clone https://github.com/s3fs-fuse/s3fs-fuse.git && \
    cd s3fs-fuse && \
    ./autogen.sh && \
    ./configure --prefix=/usr && \
    make && \
    make install && \
    cd .. && rm -rf s3fs-fuse

# Create a virtual environment and install Python packages
RUN python3 -m venv /opt/venv && \
    /opt/venv/bin/pip install --upgrade pip && \
    /opt/venv/bin/pip install jupyter pandas streamlit folium plotly boto3 streamlit-folium

# Create necessary directories
RUN mkdir -p /mnt/s3

# Copy the upload script
COPY upload.py /home/ubuntu/upload.py
```

Figure 8: Dockerfile creation

```
# Copy the cron job script
COPY run_biweekly.sh /home/ubuntu/run_biweekly.sh

# Copy the check script
COPY check_mount.sh /home/ubuntu/check_mount.sh

# Add cron job
RUN chmod +x /home/ubuntu/run_biweekly.sh /home/ubuntu/check_mount.sh && \
    echo "0 17 1,15 * * /home/ubuntu/run_biweekly.sh" > /etc/cron.d/run_biweekly && \
    chmod 0644 /etc/cron.d/run_biweekly && \
    crontab /etc/cron.d/run_biweekly

# Copy the Streamlit app script
COPY app.py /home/ubuntu/app.py

# Expose Streamlit port
EXPOSE 8501

# Set the entrypoint to mount the S3 bucket and start Jupyter
ENTRYPOINT ["/bin/bash", "-c", "s3fs calgarytrafficincidents /mnt/s3 -o iam_role=LabRole -o use_cache=/tmp -o allow_other -o uid=1000 -o mp_umask=002 -o mul
```

Figure 9: Running cron job script

Build and Run Docker Image: The Docker image is built and run on the EC2 instance, ensuring that the data ingestion script executes in a consistent and isolated environment.

```
ubuntu@ip-172-31-31-252:~$ docker build -t project608:latest .
```

```
ubuntu@ip-172-31-31-252:~$ docker run --env-file=.env -p 8501:8501 --privileged --cap-add SYS_ADMIN --device /dev/fuse project608:latest
```

Automating Data Processing with AWS Lamda

To ensure our data ingestion process is both scalable and efficient, we have implemented an AWS Lambda function that automates the processing of incoming CSV data files. Below, we describe the key steps performed by this Lambda function:

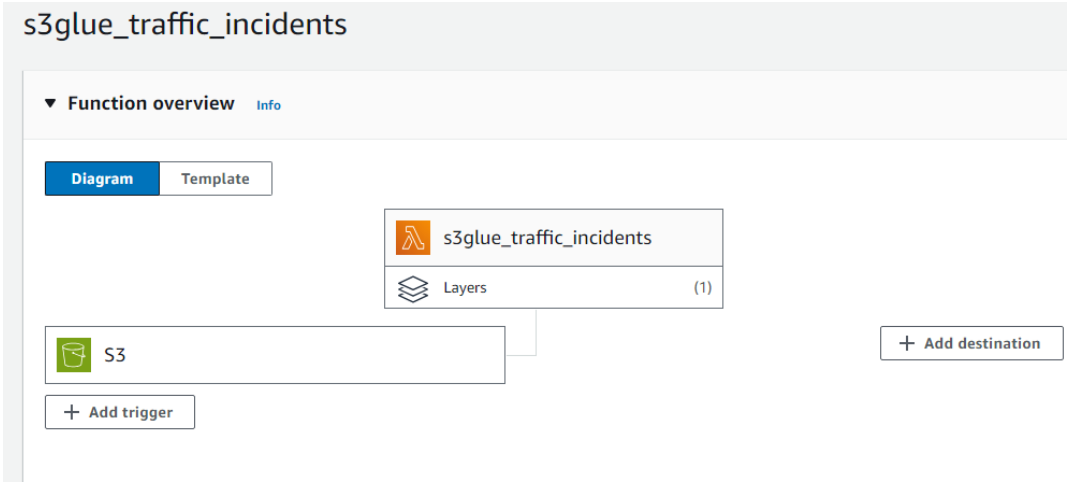


Figure 10: AWS Lambda function configuration

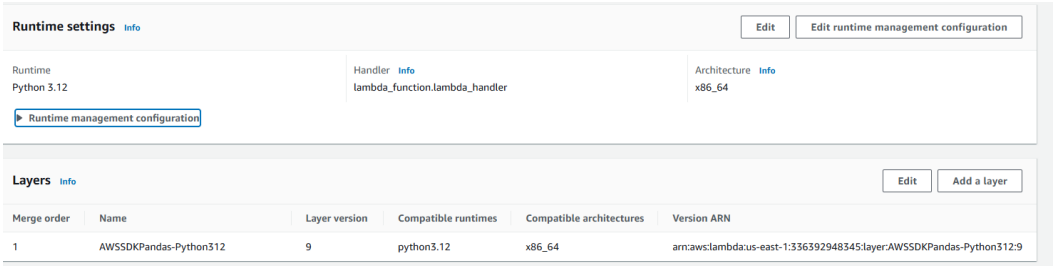


Figure 11: Deploying the function

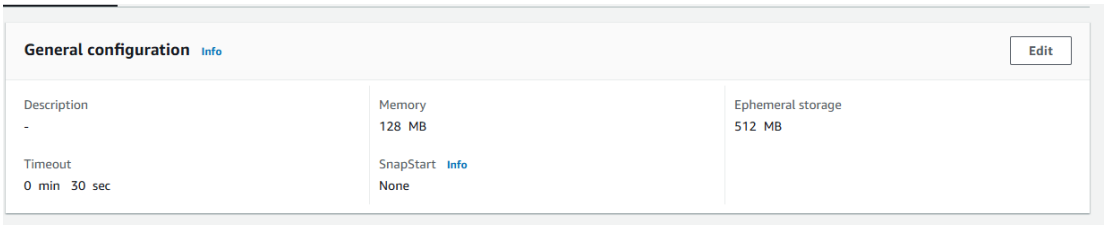


Figure 12: Update timeout duration

Lambda Function Overview:

Objective: Automatically read, process, and store CSV files from an AWS S3 bucket.

Trigger: The function is triggered whenever a new CSV file is uploaded to the S3 bucket, ensuring immediate processing.

lambda_function × Environment Var ×

```python
import json
import boto3
import pandas as pd
from io import StringIO

def lambda_handler(event, context):
    try:
        # Define the S3 bucket and file
        bucket = 'calgarytrafficincidents'
        key = 'data_from_calgary.csv'

        # Create an S3 client
        s3 = boto3.client('s3')

        # Read the CSV file from S3
        response = s3.get_object(Bucket=bucket, Key=key)
        csv_content = response['Body'].read().decode('utf-8')

        # Load the CSV content into a DataFrame
        df = pd.read_csv(StringIO(csv_content))

        # Verify and print the columns in the CSV file
        columns = df.columns.tolist()
        print("Columns in CSV:", columns)

        # Check if 'START_DT' column exists in the DataFrame (case insensitive)
        date_column = next((col for col in columns if col.lower() == 'start_dt'.lower()), None)
        if date_column is None:
            raise ValueError("Column 'START_DT' not found in the CSV file.")

        # Convert START_DT to datetime and extract the year and month
        df[date_column] = pd.to_datetime(df[date_column], errors='coerce')
        df['year'] = df[date_column].dt.year
        df['month'] = df[date_column].dt.month

        # Drop rows with invalid dates
        df = df.dropna(subset=[date_column])

        # Convert the DataFrame back to CSV
        csv_buffer = StringIO()
        df.to_csv(csv_buffer, index=False)

        # Write the updated CSV back to S3
        s3.put_object(Bucket=bucket, Key='calgarytrafficincidents_clean.csv', Body=csv_buffer.getvalue())

        return {
            'statusCode': 200,
            'body': json.dumps('CSV updated with year and month columns and saved to S3')
        }
    except Exception as e:
        return {
            'statusCode': 500,
            'body': json.dumps(f'Error processing CSV file: {str(e)}')
        }
```

Figure 13: Code for Lambda function

Function Workflow:

1. **Read CSV File:** The function begins by reading a CSV file named 'data_from_calgary.csv' from the 'calgarytrafficincidents' S3 bucket.

2. **Data Transformation:** It converts the 'START_DT' column to a datetime object to extract the 'year' and 'month' from each entry, adding these as new columns to the DataFrame. This step is crucial for subsequent time-based analysis.

3. **Clean Data:** The function also removes rows with invalid dates to maintain data quality.

4. **Write Back to S3:** After processing, the updated CSV is written back to the S3 bucket, overwriting the original file or creating a new file named 'calgarytrafficincidents_clean.csv'.

Section 2.3: Data storage

Storage Mechanism

Data is stored in an S3 bucket (calgarytrafficincidents). AWS Glue is used to catalog and prepare data for analysis, allowing easy access and management through Athena.

Tools and Technologies

- **Amazon S3:** Provides scalable and secure storage for raw and processed data.

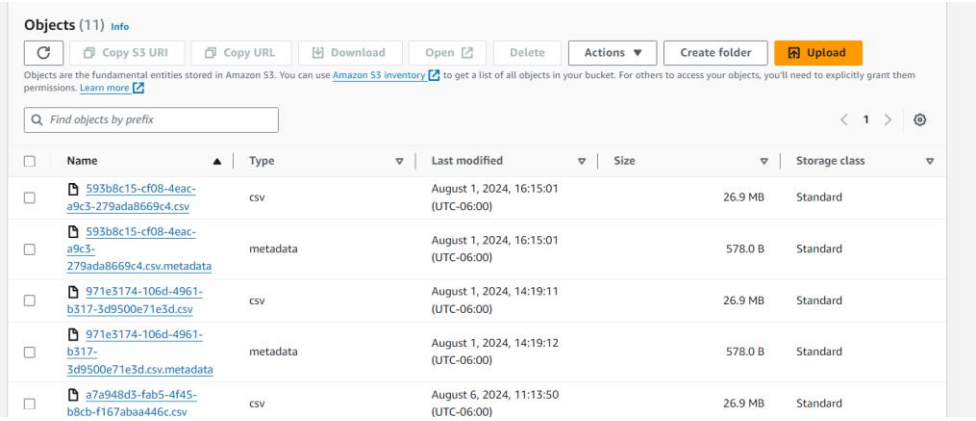


Figure 14: Amazon S3

1. Navigate to the S3 Console
2. Select Your Bucket to set up the event notification for
3. Go to the Properties Tab
4. Scroll down to the Event notifications section and click Create event notification.
5. Configure Event Notification:
    - Name: Give your event a name.
    - Event types: Select the event types you want to trigger the Lambda function.
    - Specify a prefix and/or suffix to filter the events.
    - Choose Lambda Function.
    - Select the Lambda function you created.
    - Click Save to create the event notification.

- **AWS Glue Crawlers:** AWS Crawlers scans the S3 bucket to catalog the data into tables, which are then utilized in the AWS Glue Data Catalog. This setup simplifies the management and accessibility of data across AWS services. There are 2 crawlers which infer the schema, create 2 tables, Staging and Main for further processing and transformation of data by AWS Glue ETL jobs.

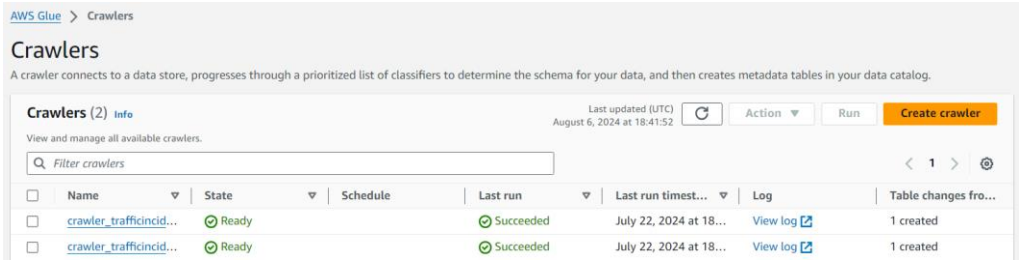


Figure 15: 2 crawlers which can be scheduled to run on regular basis

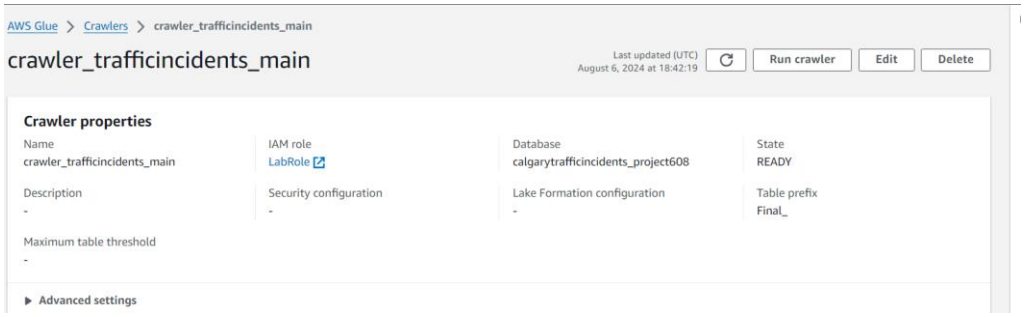


Figure 16: Crawlers continued

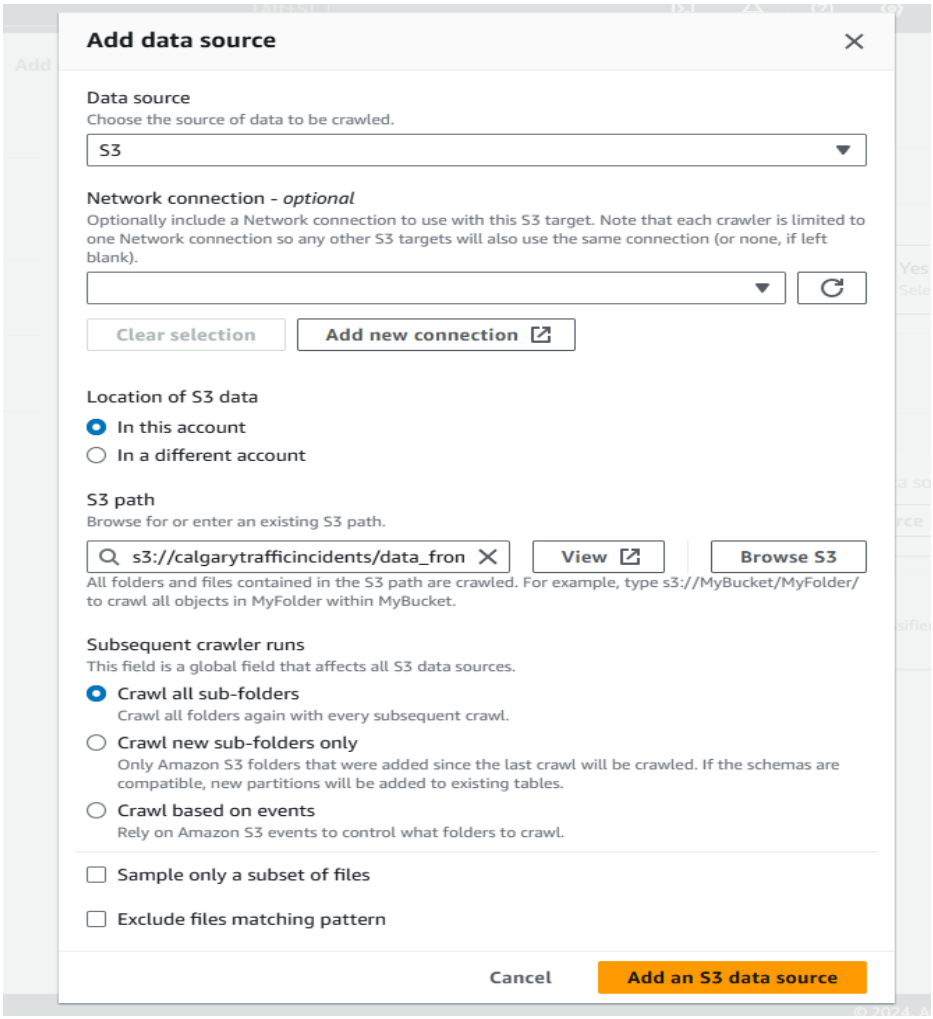


Figure 17: Screen capture of crawler data source configuration

Section 2.4: Data transformation

Transformation Processes

AWS Glue ETL jobs are used to clean and transform the data. The ETL process involves removing rows with null values and duplicates. There are 3 ETL jobs set up for data transformation.

The process of managing data flows within AWS Glue, specifically transitioning from staging to a main data environment:

Load staging ETL:

This ETL job takes the data from S3 bucket and moves it into the staging table.

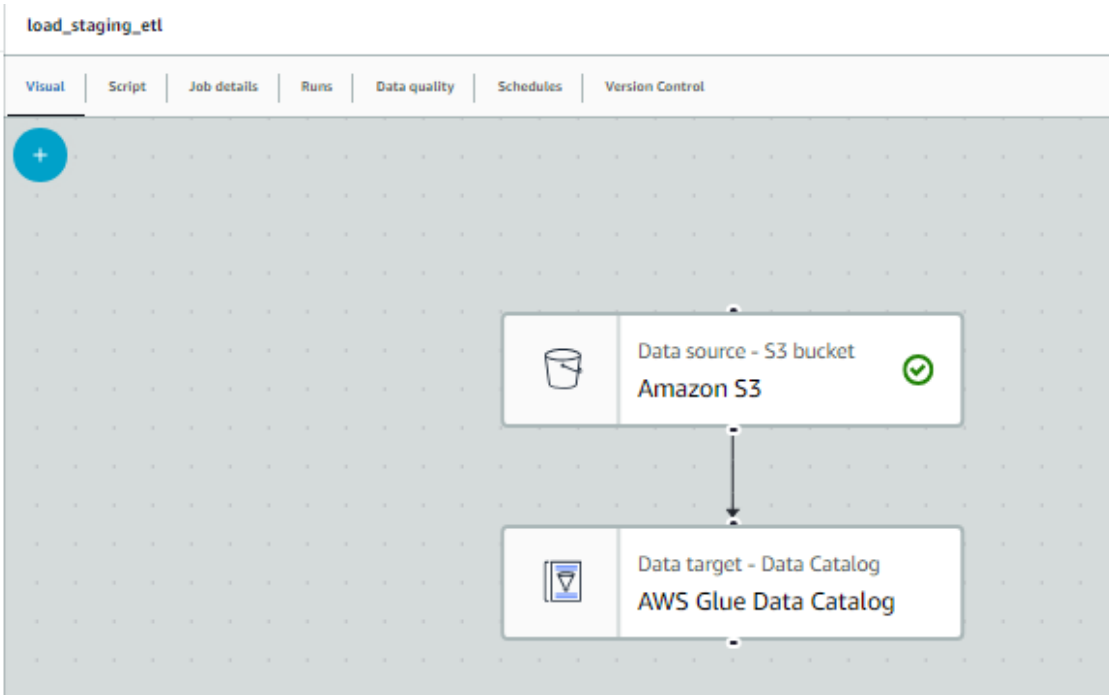


Figure 18: Staging

Staging to Main:

Start with the data sourced from staging table

Transformation Logic: The ETL job (staging to main) combines data from the staging area to the main catalog. Detail the de-duplication process and how data is combined, ensuring the most recent records are retained.

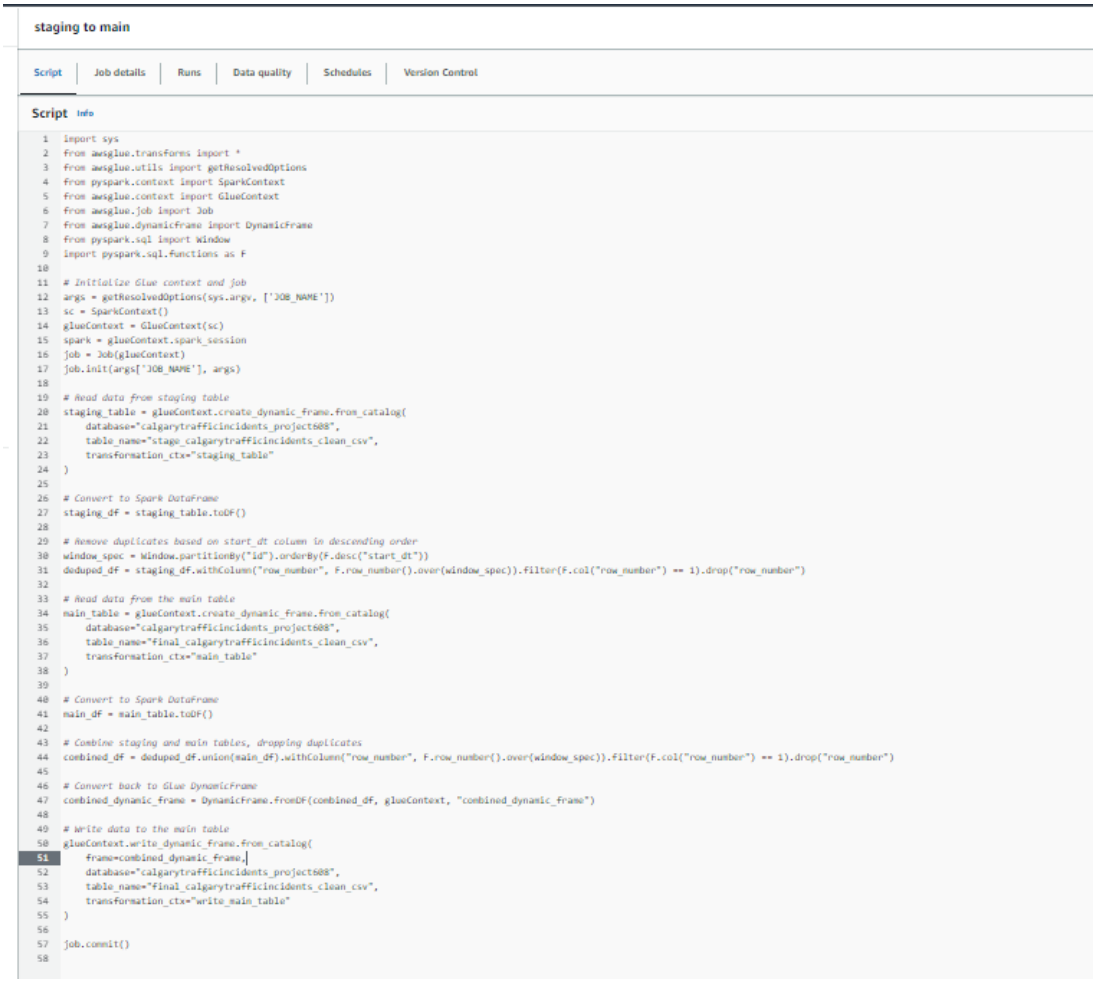


Figure 19: Loading, cleaning and writing to main table

MySQL:

cleaned and processed data is written to an external MySQL database hosted on AWS RDS

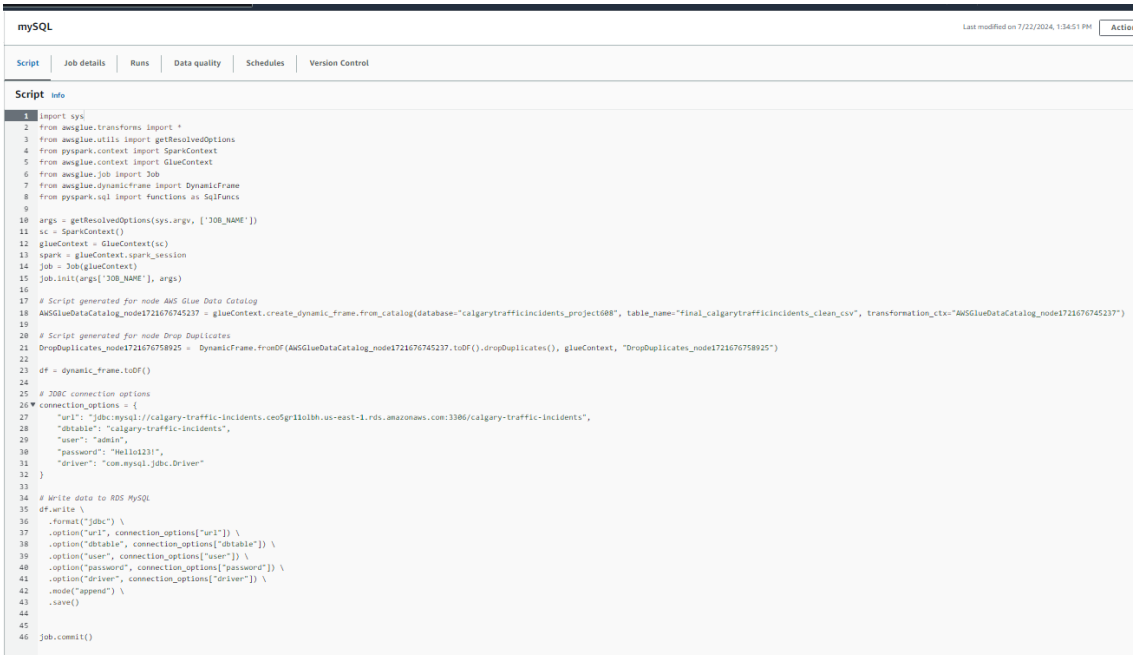


Figure 20: Writing to external database

Tools and Technologies

- **AWS Glue:** Handles ETL tasks, creating transformed datasets ready for analysis.

Section 2.5: Data serving

Serving Layer

The transformed data is served via a Streamlit application hosted on an EC2 instance. The application allows users to interact with visualizations, exploring incidents by time and location.

Tools and Technologies

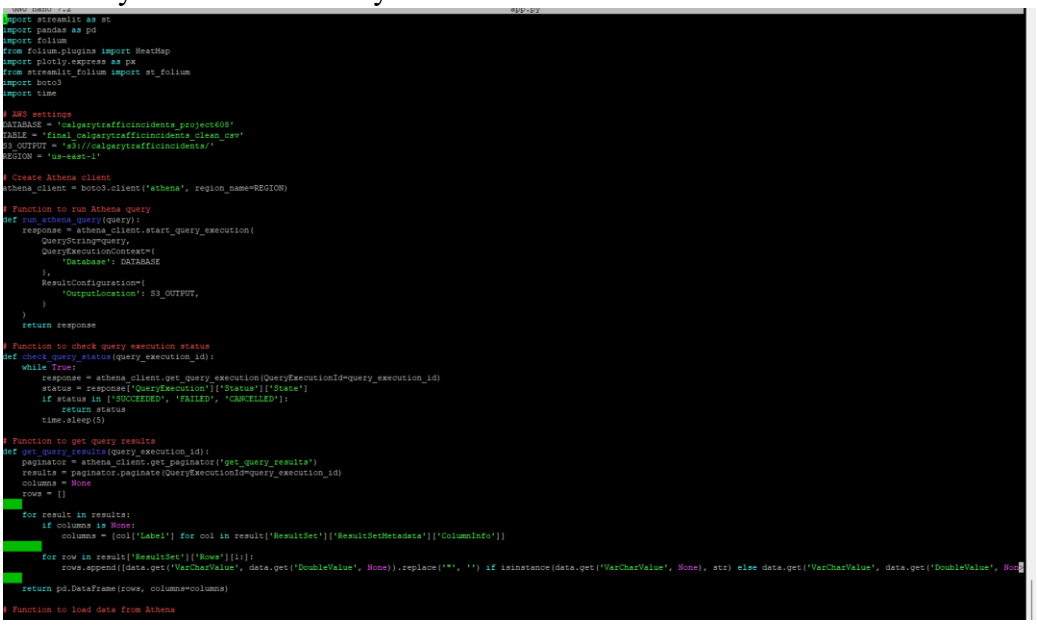
- **Streamlit:** Provides an interactive web interface for data visualization.
- **Plotly and Folium:** Used within Streamlit to create interactive maps and charts.
- **Amazon EC2:** Hosts the Streamlit application, making it publicly accessible.

The data transformed in the previous stages is made accessible and queryable using AWS Athena, which connects directly to the AWS Glue Data Catalog. This approach provides a highly efficient way to manage and analyze large datasets without the need for traditional database infrastructure. Below is the app.py which is deployed via docker container on EC2 instance. The provided Python script is a crucial component of the data serving layer, effectively bridging the gap between raw data stored in AWS and actionable insights presented to the user. The script integrates various AWS services and Python libraries to create a robust and user-friendly data serving solution.

1. Data Fetching

The script begins by establishing a connection to AWS Athena, the service responsible for querying the data stored in AWS S3. The process includes:

- **Connection Setup**: Connect to Athena, specifying the necessary credentials and query configurations.
- **Running SQL Queries**: SQL queries are executed against the data stored in AWS S3 via Athena. These queries are designed to extract the required traffic incident data, filtered and formatted according to the needs of the analysis.
- **Handling Responses**: The results from Athena are fetched and loaded into a Pandas DataFrame for further processing. The script ensures that the data is accurately retrieved and ready for transformation.



```
import streamlit as st
import pandas as pd
import folium
from folium.plugins import HeatMap
import plotly.express as px
from streamlit_folium import st_folium
import boto3
import time

# AWS settings
DATABASE = 'calgarytrafficincidents_project608'
TABLE = 'final_calgarytrafficincidents_clean_csv'
S3_OUTPUT = 's3://calgarytrafficincidents/'
REGION = 'us-east-1'

# Create Athena client
athena_client = boto3.client('athena', region_name=REGION)

# Function to run Athena query
def run_athena_query(query):
    response = athena_client.start_query_execution(
        QueryString=query,
        QueryExecutionContext={
            'Database': DATABASE
        },
        ResultConfiguration={
            'OutputLocation': S3_OUTPUT,
        }
    )
    return response

# Function to check query execution status
def check_query_status(query_execution_id):
    while True:
        response = athena_client.get_query_execution(QueryExecutionId=query_execution_id)
        status = response['QueryExecution']['Status']['State']
        if status in ['SUCCEEDED', 'FAILED', 'CANCELLED']:
            return status
        time.sleep(5)

# Function to get query results
def get_query_results(query_execution_id):
    paginator = athena_client.get_paginator('get_query_results')
    results = paginator.paginate(QueryExecutionId=query_execution_id)
    columns = None
    rows = []

    for result in results:
        if columns is None:
            columns = [col['Label'] for col in result['ResultSet']['ResultSetMetadata']['ColumnInfo']]

        for row in result['ResultSet']['Rows'][1:]:
            rows.append([data.get('VarCharValue', data.get('DoubleValue', None)).replace('"', '') if isinstance(data.get('VarCharValue', None), str) else data.get('VarCharValue', data.get('DoubleValue', None)

    return pd.DataFrame(rows, columns=columns)

# Function to load data from Athena
```

Figure 21: Athena utilization

2. Data Processing

Once the data is fetched from Athena, the script leverages Pandas for various data transformations, particularly focusing on date-time processing:

- **Date-Time Transformations**: The script converts date-time strings into Pandas DateTime objects, enabling more complex time-based analysis. This transformation is crucial for generating time-series visualizations and identifying trends over specific periods.
- **Preparation for Visualization**: The processed data is then prepared for visualization, with the script ensuring that it is in the appropriate format for generating the desired plots and maps. This includes aggregating data, calculating necessary statistics, and setting up data structures for visual tools like Plotly and Folium.

```
@st.cache_data
def load_data():
    query = f'SELECT * FROM "{DATABASE}"."{TABLE}";'
    response = run_athena_query(query)
    query_execution_id = response['QueryExecutionId']
    status = check_query_status(query_execution_id)
    if status == 'SUCCEEDED':
        df = get_query_results(query_execution_id)
        return df
    else:
        st.error("Query failed")
        return pd.DataFrame()

# Load data
df = load_data()

# Convert start_dt to datetime
df['start_dt'] = pd.to_datetime(df['start_dt'], errors='coerce')
```

Figure 22: Loading data

3. Visualization

The script utilizes Streamlit, along with Folium and Plotly libraries, to present the processed data through interactive and insightful visualizations:

- **Data Overview**: The script generates an overview of the traffic incidents, presenting key statistics such as total incidents, incident types, and peak times. This summary provides users with a quick snapshot of the data.
- **Trend Lines Over Time**: Using Plotly, the script creates interactive trend lines that show the progression of traffic incidents over time. These visualizations help users identify patterns, such as increases or decreases in incidents during specific months or days.

- **Geographical Heatmaps**: Folium is employed to create geographical heatmaps that display the density of traffic incidents across Calgary. These maps are embedded in the Streamlit application, allowing users to explore the spatial distribution of incidents interactively.
- **User-Friendly Interface**: The integration with Streamlit ensures that the entire visualization process is interactive and user-friendly. Users can adjust filters, zoom into specific data points, and explore the data in a highly intuitive manner.

Figure 22: Streamlit and Folium utilization

Tools and Technologies

1. **AWS Athena**:
    - **Functionality**: AWS Athena is utilized for querying the transformed data stored in AWS S3 via the Glue Data Catalog. As a serverless interactive query service, Athena allows users to execute SQL queries directly on data stored in S3, without requiring any underlying database management.
    - **Benefits**:
        - **Scalability**: Athena's serverless architecture automatically scales based on the volume and complexity of queries, making it well-suited for handling large-scale datasets.
        - **Cost-Efficiency**: Users only pay for the queries they run, which makes Athena a cost-effective solution for large, infrequent queries.
2. **Streamlit**:
    - **Functionality**: Streamlit is used as the front-end application framework, providing an interactive platform for users to engage with the data through custom web apps.
    - **Integration**: Streamlit directly interacts with AWS Athena to fetch the required data. It is used to create and share interactive visualizations and

dashboards, which are crucial for data analysis and decision-making processes.

- **Capabilities**:
  - **Real-Time Interaction**: Streamlit's interface is designed to update interactively based on user inputs, allowing for a dynamic data exploration experience.
  - **Caching**: Streamlit's @st.cache_data decorator is employed to cache data, which minimizes the need to re-run long queries, thereby enhancing the user experience by speeding up data retrieval.

3. **Folium**:
   - **Functionality**: Folium is a Python library used to generate maps. In this project, it is specifically utilized for creating heatmaps that visualize the distribution and density of traffic incidents across Calgary.
   - **Integration**: The maps generated by Folium are integrated into the Streamlit application, providing users with a geographic representation of traffic incidents.
4. **Plotly**:
   - **Functionality**: Plotly is used for creating interactive graphs and plots. These visualizations are embedded within the Streamlit application, offering users the ability to analyze trends, patterns, and anomalies in traffic incidents.
   - **Capabilities**:
     - **Interactivity**: Plotly's interactive capabilities allow users to zoom, hover, and filter data points, providing a deeper insight into the traffic data.

Performance and Scalability

- **Athena's Serverless Architecture**: The serverless nature of Athena ensures that it can scale automatically to meet the demands of large and complex queries. This eliminates the need for manual scaling, which would otherwise be necessary with traditional database systems.
- **Cache Utilization**: By using Streamlit's caching capabilities, the application avoids redundant computations and data fetching, which is especially important for improving the speed and responsiveness of the user interface.
- **Real-Time Interaction**: The combination of Athena's fast query execution and Streamlit's responsive interface creates a real-time data exploration environment. Users can interactively explore the data, adjusting parameters and filters on the fly to generate new insights.

Section 2.6: Any additional steps (if applicable)

AWS Glue crawlers automatically detect changes in the data schema and update the data catalog, ensuring seamless integration and accessibility of data.

Model Overview

This model integrates AWS services (Athena, S3) and Python libraries (Pandas, Streamlit, Folium, Plotly) to create a comprehensive data serving layer. The approach is designed to be both efficient and user-friendly, enabling dynamic data exploration and insightful visualizations. This model not only handles the complexities of data fetching and processing but also ensures that the insights are presented in a format that is easy to understand and interact with.

This integration of technologies results in a powerful tool for analyzing traffic incidents in Calgary, providing users with the insights they need to make informed decisions in their everyday lives.

Section 3: Evaluation

The application was evaluated on its ability to handle data updates, performance in delivering visualizations, and user feedback on interface usability. It efficiently managed data updates without disruption. We found that visualizations loaded quickly and accurately, even with our relatively large datasets.

The automated data pipeline effectively managed the data lifecycle from ingestion to visualization. This seamless automation met our project objectives, providing users with reliable, up-to-date insights. Overall, the application proved to be a robust, user-friendly tool that delivered our intended purpose.

Section 4: Limitations and possible next steps

Limitations

- **Real-time Processing:** The current setup processes data biweekly, which may not capture the latest incidents immediately.
- **Scalability:** The setup is limited by the current EC2 instance capacity and could face performance bottlenecks with increased data volume.

Possible Next Steps

- **Real-time Data Ingestion:** Implement streaming solutions such as AWS Kinesis to handle real-time data updates.
- **Scalability Improvements:** Explore serverless options with AWS Lambda for the Streamlit app or use AWS Elastic Beanstalk for auto-scaling.
- **Enhanced Visualizations:** Integrate more complex analytics and visualizations, such as clustering algorithms to identify incident hotspots.

Future steps:

- **Machine Learning Models**: The next phase of the project could involve developing and deploying machine learning models to predict traffic incidents based on historical data, weather conditions, and other relevant factors. This predictive capability would enable proactive traffic management and potentially reduce the number of incidents.
- **Real-Time Data Processing**: Enhancing the current pipeline to support real-time data processing would allow for near-instant insights and alerts for new traffic incidents. Integrating technologies such as AWS Kinesis could facilitate this capability. In terms of real-world applications, improving the cadence of updates would provide more meaningful data for the end user to make informed decisions.
- **Advanced Visualization**: Incorporating more advanced data visualization techniques, such as time series analysis or predictive modeling outputs, could provide deeper insights into traffic patterns and help in identifying long-term trends.
- **Extended Data Sources**: Expanding the dataset to include additional sources, such as weather data or roadwork schedules, could provide a more comprehensive view of the factors influencing traffic incidents, leading to more accurate analyses and predictions.

Section 5: Conclusions

The project demonstrates the power of AWS services in creating a scalable and efficient data pipeline. By automating the ingestion, processing, and serving of traffic incident data, users can gain valuable insights into traffic patterns and incidents in Calgary. This project is open ended in the sense that there is no perfect product. There is ample opportunity to continually add resources to provide clarity and provide more insightful information beyond what we have portrayed thus far. This could mean importing data from external or joining within the resources the City of Calgary already provides.

Enhanced Data Accessibility

The project successfully automates the entire data pipeline, from ingestion to serving, ensuring that traffic incident data is continuously and reliably available for analysis. The use of AWS services such as S3, Glue, and Athena significantly reduces the manual effort required to maintain the data pipeline, while also enhancing data reliability and accessibility.

Lessons Learned

- **Importance of Automation**: The project demonstrates the critical role of automation in maintaining data freshness and reducing human error. By automating the ingestion, transformation, and serving processes using AWS services and CRON jobs, the project achieves scalability and efficiency.
- **Integration Complexity**: Integrating various AWS services requires a deep understanding of AWS Identity and Access Management (IAM) roles, permissions, and configurations. The challenges encountered with permissions highlighted the need for thorough planning and understanding of AWS security configurations.

Section 6: Shareable Resources

The following is the link to a YouTube video, which shows the streamlit app deployed on EC2 instance. The data is pulled from Athena using a query in the background and then data is cached.

Demo Video: https://www.youtube.com/watch?v=Y8v4tYaSSTA

Section 7: Code Appendix

Functions/Sections will have names bolded, code will follow. Can be used as reference for continuation.

Upload.py

```
# Define the URL and the local file name
url = "https://data.calgary.ca/resource/35ra-9556.csv"
local_file = "data_from_calgary.csv"
bucket_name = "calgarytrafficincidents"
s3_object_name = "data_from_calgary.csv"


# Download the JSON data from Open Calgary
os.system(f'wget -O {local_file} {url}')


# Upload the downloaded file to S3
os.system(f'aws s3 cp {local_file} s3://{bucket_name}/{s3_object_name}')
```

run_biweekly.sh

```
/usr/bin/python3 /home/ubuntu/upload.py > /home/ubuntu/run_biweekly.log 2>&1
```

Dockerfile

```
# Use the official Ubuntu base image
FROM ubuntu:24.04


# Update the package list and install necessary packages
RUN apt-get update && \
    apt-get install -y \
    python3 \
    python3-venv \
    wget \
    cron \
    automake \
    autotools-dev \
    fuse \
    g++ \
    git \
    libcurl4-openssl-dev \
    libfuse-dev \
    libssl-dev \
    libxml2-dev \
    make \
    pkg-config \
    nano && \
    apt-get clean && \
    rm -rf /var/lib/apt/lists/* /tmp/* /var/tmp/*


# Install s3fs
RUN git clone https://github.com/s3fs-fuse/s3fs-fuse.git && \
    cd s3fs-fuse && \
    ./autogen.sh && \
    ./configure --prefix=/usr && \
    make && \
    make install && \
    cd .. && rm -rf s3fs-fuse


# Create a virtual environment and install Python packages
RUN python3 -m venv /opt/venv && \
    /opt/venv/bin/pip install --upgrade pip && \
    /opt/venv/bin/pip install jupyter pandas streamlit folium plotly boto3 streamlit-folium
```

```
# Create necessary directories
RUN mkdir -p /mnt/s3

# Copy the upload script
COPY upload.py /home/ubuntu/upload.py

# Copy the cron job script
COPY run_biweekly.sh /home/ubuntu/run_biweekly.sh

# Copy the check script
COPY check_mount.sh /home/ubuntu/check_mount.sh

# Add cron job
RUN chmod +x /home/ubuntu/run_biweekly.sh /home/ubuntu/check_mount.sh && \
    echo "0 17 1,15 * * /home/ubuntu/run_biweekly.sh" > /etc/cron.d/run_biweekly && \
    chmod 0644 /etc/cron.d/run_biweekly && \
    crontab /etc/cron.d/run_biweekly

# Copy the Streamlit app script
COPY app.py /home/ubuntu/app.py

# Expose Streamlit port
EXPOSE 8501

# Set the entrypoint to mount the S3 bucket and start Jupyter
ENTRYPOINT ["/bin/bash", "-c", "s3fs calgarytrafficincidents /mnt/s3 -o iam_role=LabRole -o
use_cache=/tmp -o allow_other -o uid=1000 -o mp_umask=002 -o mul>
```

checkmount.sh

```
#!/bin/bash


# Check if the S3 bucket is mounted
if mountpoint -q /mnt/s3; then
    echo "S3 bucket is mounted successfully."
    ls -la /mnt/s3
else
    echo "Failed to mount S3 bucket."
    exit 1
fi
```

app.py

```
 import streamlit as st
import pandas as pd
import folium
```

```
from folium.plugins import HeatMap
import plotly.express as px
from streamlit_folium import st_folium
import boto3
import time


# AWS settings
DATABASE = 'calgarytrafficincidents_project608'
TABLE = 'final_calgarytrafficincidents_clean_csv'
S3_OUTPUT = 's3://calgarytrafficincidents/'
REGION = 'us-east-1'


# Create Athena client
athena_client = boto3.client('athena', region_name=REGION)


# Function to run Athena query
def run_athena_query(query):
    response = athena_client.start_query_execution(
        QueryString=query,
        QueryExecutionContext={
            'Database': DATABASE
        },
        ResultConfiguration={
            'OutputLocation': S3_OUTPUT,
        }
    )
    return response


# Function to check query execution status
def check_query_status(query_execution_id):
    while True:
        response = athena_client.get_query_execution(QueryExecutionId=query_execution_id)
        status = response['QueryExecution']['Status']['State']
        if status in ['SUCCEEDED', 'FAILED', 'CANCELLED']:
            return status
        time.sleep(5)


# Function to get query results
def get_query_results(query_execution_id):
    paginator = athena_client.get_paginator('get_query_results')
```

```
    results = paginator.paginate(QueryExecutionId=query_execution_id)
    columns = None


    rows = []


    for result in results:
        if columns is None:
            columns = [col['Label'] for col in result['ResultSet']['ResultSetMetadata']['ColumnInfo']]

        for row in result['ResultSet']['Rows'][1:]:
            rows.append([data.get('VarCharValue', data.get('DoubleValue', None)).replace("", '') if
isinstance(data.get('VarCharValue', None), str) else da>

    return pd.DataFrame(rows, columns=columns)

# Function to load data from Athena
@st.cache_data
def load_data():
    query = f'SELECT * FROM "{DATABASE}"."{TABLE}";'
    response = run_athena_query(query)
    query_execution_id = response['QueryExecutionId']
    status = check_query_status(query_execution_id)
    if status == 'SUCCEEDED':
        df = get_query_results(query_execution_id)
        return df
    else:
        st.error("Query failed")
        return pd.DataFrame()


# Load data
df = load_data()


# Convert start_dt to datetime
df['start_dt'] = pd.to_datetime(df['start_dt'], errors='coerce')




# Drop rows with null datetime values
df = df.dropna(subset=['start_dt'])
```

```
# Extract year and month as integers
df['Year'] = df['start_dt'].dt.year.astype(int)
df['Month'] = df['start_dt'].dt.month.astype(int)


# Streamlit app
st.title("Calgary Traffic Incidents")


# Filters

st.sidebar.header("Filters")


selected_year = st.sidebar.selectbox("Select Year", sorted(df['Year'].unique(), reverse=True))
selected_month = st.sidebar.selectbox("Select Month", sorted(df['Month'].unique(), reverse=True))


# Convert selected values to integers
selected_year = int(selected_year)
selected_month = int(selected_month)


# Filter data based on selections
filtered_df = df[(df['Year'] == selected_year) & (df['Month'] == selected_month)]


st.header(f"Data Overview for {selected_year}-{selected_month:02d}")
st.write(filtered_df.head())


# Line graph
st.header("Incidents Over Time")
filtered_df['Date'] = filtered_df['start_dt'].dt.date
daily_counts = filtered_df.groupby('Date').size().reset_index(name='Count')
fig = px.line(daily_counts, x='Date', y='Count', title='Daily Incident Counts')
st.plotly_chart(fig)


# Heatmap
st.header("Incident Heatmap")


# Calculate map center
map_center = [filtered_df['latitude'].astype(float).mean(), filtered_df['longitude'].astype(float).mean()]
```

```
m = folium.Map(location=map_center, zoom_start=12)
heat_data = [[row['latitude'], row['longitude']] for index, row in filtered_df.iterrows()]
HeatMap(heat_data).add_to(m)


st_folium(m, width=700, height=500)
```

s3glue_traffic_incidents - Lamda function

```
import json
import boto3
import pandas as pd
from io import StringIO


def lambda_handler(event, context):
    try:
        # Define the S3 bucket and file
        bucket = 'calgarytrafficincidents'
        key = 'data_from_calgary.csv'

        # Create an S3 client
        s3 = boto3.client('s3')

        # Read the CSV file from S3
        response = s3.get_object(Bucket=bucket, Key=key)
        csv_content = response['Body'].read().decode('utf-8')

        # Load the CSV content into a DataFrame
        df = pd.read_csv(StringIO(csv_content))

        # Verify and print the columns in the CSV file
        columns = df.columns.tolist()
        print("Columns in CSV:", columns)

        # Check if 'START_DT' column exists in the DataFrame (case insensitive)
        date_column = next((col for col in columns if col.lower() == 'start_dt'.lower()), None)
        if date_column is None:
            raise ValueError("Column 'START_DT' not found in the CSV file.")

        # Convert START_DT to datetime and extract the year and month
        df[date_column] = pd.to_datetime(df[date_column], errors='coerce')
        df['year'] = df[date_column].dt.year
        df['month'] = df[date_column].dt.month
```

```
        # Drop rows with invalid dates
        df = df.dropna(subset=[date_column])

        # Convert the DataFrame back to CSV
        csv_buffer = StringIO()
        df.to_csv(csv_buffer, index=False)

        # Write the updated CSV back to S3
        s3.put_object(Bucket=bucket, Key='calgarytrafficincidents_clean.csv', Body=csv_buffer.getvalue())

        return {
            'statusCode': 200,
            'body': json.dumps('CSV updated with year and month columns and saved to S3')
        }
    except Exception as e:
        return {
            'statusCode': 500,
            'body': json.dumps(f'Error processing CSV file: {str(e)}')
        }
```

AWS ELT Glue - load staging etl

```
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


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


# Script generated for node Amazon S3
AmazonS3_node1721617562081 =
glueContext.create_dynamic_frame.from_options(format_options={"quoteChar": "\"", "withHeader":
True, "separator": ",", "optimizePerformance": False}, connection_type="s3", format="csv",
connection_options={"paths": ["s3://calgarytrafficincidents/calgarytrafficincidents_clean.csv"], "recurse":
True}, transformation_ctx="AmazonS3_node1721617562081")
```

```
# Script generated for node AWS Glue Data Catalog
AWSGlueDataCatalog_node1721622343671 =
glueContext.write_dynamic_frame.from_catalog(frame=AmazonS3_node1721617562081,
database="calgarytrafficincidents_project608", table_name="stage_calgarytrafficincidents_clean_csv",
transformation_ctx="AWSGlueDataCatalog_node1721622343671")

 job.commit()
```

AWS ETL Glue - staging to main

```
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 awsglue.dynamicframe import DynamicFrame
from pyspark.sql import Window
import pyspark.sql.functions as F


# Initialize Glue context and job
args = getResolvedOptions(sys.argv, ['JOB_NAME'])
sc = SparkContext()
glueContext = GlueContext(sc)
spark = glueContext.spark_session
job = Job(glueContext)
job.init(args['JOB_NAME'], args)


# Read data from staging table
staging_table = glueContext.create_dynamic_frame.from_catalog(
    database="calgarytrafficincidents_project608",
    table_name="stage_calgarytrafficincidents_clean_csv",
    transformation_ctx="staging_table"
)


# Convert to Spark DataFrame
staging_df = staging_table.toDF()


# Remove duplicates based on start_dt column in descending order
window_spec = Window.partitionBy("id").orderBy(F.desc("start_dt"))
```

```
deduped_df = staging_df.withColumn("row_number",
F.row_number().over(window_spec)).filter(F.col("row_number") == 1).drop("row_number")


# Read data from the main table
main_table = glueContext.create_dynamic_frame.from_catalog(
    database="calgarytrafficincidents_project608",
    table_name="final_calgarytrafficincidents_clean_csv",
    transformation_ctx="main_table"
)


# Convert to Spark DataFrame
main_df = main_table.toDF()


# Combine staging and main tables, dropping duplicates
combined_df = deduped_df.union(main_df).withColumn("row_number",
F.row_number().over(window_spec)).filter(F.col("row_number") == 1).drop("row_number")


# Convert back to Glue DynamicFrame
combined_dynamic_frame = DynamicFrame.fromDF(combined_df, glueContext,
"combined_dynamic_frame")


# Write data to the main table
glueContext.write_dynamic_frame.from_catalog(
    frame=combined_dynamic_frame,
    database="calgarytrafficincidents_project608",
    table_name="final_calgarytrafficincidents_clean_csv",
    transformation_ctx="write_main_table"
)

job.commit()
```

AWS ETL Glue - my SQL

```
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 awsglue.dynamicframe import DynamicFrame
from pyspark.sql import functions as SqlFuncs
```

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


# Script generated for node AWS Glue Data Catalog
AWSGlueDataCatalog_node1721676745237 =
glueContext.create_dynamic_frame.from_catalog(database="calgarytrafficincidents_project608",
table_name="final_calgarytrafficincidents_clean_csv",
transformation_ctx="AWSGlueDataCatalog_node1721676745237")


# Script generated for node Drop Duplicates
DropDuplicates_node1721676758925
=  DynamicFrame.fromDF(AWSGlueDataCatalog_node1721676745237.toDF().dropDuplicates(),
glueContext, "DropDuplicates_node1721676758925")


df = dynamic_frame.toDF()


# JDBC connection options
connection_options = {
    "url": "jdbc:mysql://calgary-traffic-incidents.ceo5gr11olbh.us-east-1.rds.amazonaws.com:3306/calgary-
traffic-incidents",
    "dbtable": "calgary-traffic-incidents",
    "user": "admin",
    "password": "Hello123!",
    "driver": "com.mysql.jdbc.Driver"
}

# Write data to RDS MySQL
df.write \
  .format("jdbc") \
  .option("url", connection_options["url"]) \
  .option("dbtable", connection_options["dbtable"]) \
  .option("user", connection_options["user"]) \
  .option("password", connection_options["password"]) \
  .option("driver", connection_options["driver"]) \
  .mode("append") \
  .save()

job.commit()
```

Section 8: References

"OPEN CALGARY TERMS OF USE." DATA.CALGARY.CA, DATA.CALGARY.CA/STORIES/S/U45N-7AWA. ACCESSED 10 JULY 2024.

"TRAFFIC INCIDENTS | OPEN CALGARY." DATA.CALGARY.CA, DATA.CALGARY.CA/TRANSPORTATIONTRANSIT/TRAFFIC-INCIDENTS/35RA-9556. ACCESSED 10 JULY 2024.