Rewrite the following documentation in a professional and concise manner. Apply any suggestion to improve the document or reorganize the thought process in a way that is simpler or more organized.

## Overview Mine:

This project involves multiple ETL processes that extract data from the NYC Open Data Platform, transform it and load it into a Postgres Database and a Redshift Database. Specifically, it involves extracting, transforming, and loading monthly Fire Incident Dispatch Data and Automated Traffic Volume Counts from the NYC Open Data platform to a Postgres Database for transactional and storage purposes. The data from both tables in the database are extracted as separate files and then loaded to an AWS S3 Bucket via an automated Airflow DAG job, AWS Glue Jobs then automatically run when the files are dropped into the S3 bucket transform the data (such as joining both tables into a single table via SQL statement) and then loaded to a Redshift database. A Power Bi dashboard connects to the redshift database tables and visualizations show the summarized finding for both the Fire Incident Dispatch Data and the Traffic Data.

The Fire Incident Dispatch Data contains data that is generated by the Starfire Computer Aided Dispatch System. The data spans from the time the incident is created in the system to the time the incident is closed in the system. It covers information about the incident as it relates to the assignment of resources and the Fire Department’s response to the emergency. The Automated Traffic Volume Counts contain the following: New York City Department of Transportation (NYC DOT) uses Automated Traffic Recorders (ATR) to collect traffic sample volume counts at bridge crossings and roadways.

The purpose of this project is to allow stakeholders such as city planners, public safety officials, emergency services, first responders, transportation authorities, urban data analysts/researchers, policy makers, and residents/local communities to visualize data that show monthly Fire Incident Data and Traffic Data to get insight and make informed decisions about resource allocations, planning optimal routes and response strategies, and identifying traffic bottlenecks that may cause response time delays.

A screenshot of a computer

AI-generated content may be incorrect.

## Tools and Technologies Leveraged (Mine):

* Docker
* Airflow
* PySpark
* Socrata
* Pandas
* Sodapy
* Sqlalchemy
* Psycopg2-binary
* Pyarrow
* Retry
* Tenacity
* Boto3
* Postgres
* AWS S3 Bucket
* AWS Glue
* AWS Lambda
* AWS Redshift
* AWS IAM

# ETL Process Configuration Document

## Overview

This project implements a scalable batch-processing ETL pipeline to process Fire Incident Dispatch Data and Automated Traffic Volume Counts sourced from the NYC Open Data Platform. The pipeline efficiently ingests, transforms, and stores data across Postgres (transactional storage) and Amazon Redshift (analytical storage), enabling data visualization in Power BI for actionable insights.

**ETL Process Overview**

1. **Data Extraction & Ingestion (Automated Airflow DAGs in Docker)**
   * Two automated Airflow DAGs, running in a Docker container, extract monthly Fire Incident Dispatch Data and Traffic Volume Counts from the NYC Open Data Platform.
   * The extracted data undergoes transformations using PySpark before being loaded into a Postgres database, serving as the transactional storage layer.
2. **Data Transfer & Monitoring (Airflow DAG with External Task Sensor)**
   * A third Airflow DAG, equipped with an External Task Sensor, monitors the completion of the first two DAGs.
   * Once both DAGs finish processing, the third DAG initiates, extracting the processed data from Postgres, converting it into CSV files, and uploading them to AWS S3.
3. **Data Processing in AWS (Glue Jobs & Redshift Integration)**
   * **AWS Glue Jobs** are triggered automatically upon file arrival in S3:
     + The first two jobs standardize the individual datasets and load them into Amazon Redshift.
     + The third Glue Job executes an SQL-based transformation, joining both datasets based on Borough and derived date fields, applying schema consistency before loading the final table into Redshift.
4. **Data Analysis & Visualization (Power BI)**
   * Power BI connects to Amazon Redshift, generating interactive dashboards to analyze incident dispatch trends and traffic volumes.
   * The visualizations help stakeholders monitor monthly insights and make data-driven decisions.

**Data Breakdown**

* **Fire Incident Dispatch Data:** Captured from the Starfire Computer-Aided Dispatch System, tracking incidents from creation to resolution. Provides insight into response times, resource allocation, and emergency patterns.
* **Automated Traffic Volume Counts**: NYC DOT collects vehicle volume data via Automated Traffic Recorders (ATR) at key crossings and roadways, assisting in congestion analysis.

**Stakeholder Benefits – PBI Dashboard**

* **City Planners & Policy Makers** – Optimize resource allocation and traffic infrastructure planning.
* **Public Safety Officials & First Responders** – Improve emergency response strategies.
* **Transportation Authorities** – Identify traffic bottlenecks affecting emergency routes.
* **Urban Data Analysts & Researchers** – Gain insights into fire dispatch trends and traffic density for policy recommendations.
* **Local Communities & Residents** – Understand city-wide emergency and traffic patterns for safer mobility.

This structured pipeline enhances data-driven decision-making, providing a real-time, automated workflow that bridges transactional and analytical databases for urban insights.

## Technical Configuration

**1. Overview**

* **Purpose:** Simulating a batch-processing pipeline that integrates a transactional storage database (Postgres) and an analytical database (Amazon Redshift) for visualization in Power BI.
* **Technologies Used:**
  + Docker
  + Airflow
  + PySpark
  + Socrata
  + Pandas
  + Sodapy
  + Sqlalchemy
  + Psycopg2-binary
  + Pyarrow
  + Retry
  + Tenacity
  + Boto3
  + Postgres
  + AWS S3 Bucket
  + AWS Glue
  + AWS Lambda
  + AWS Redshift
  + AWS IAM

**2. Architecture Diagram**

**A screenshot of a computer

AI-generated content may be incorrect.**

**3. Environment Setup**

**Docker Configuration**

* Base images used: postgres:13, dpage/pgadmin4, airflow-custom:latest,
* Click to view more details of the docker-compose settings.

**Postgres Database**

* Schema definitions:
  + Fire Incidents Table:

|  |  |
| --- | --- |
| **Column Name** | **Data Type** |
| alarm\_box\_borough | text |
| alarm\_box\_location | text |
| alarm\_box\_number | bigint |
| alarm\_level\_index\_description | text |
| alarm\_source\_description\_tx | text |
| category\_dispatch\_response\_seconds\_qy | text |
| category\_engines\_assigned\_quantity | text |
| category\_incident\_response\_seconds\_qy | text |
| category\_incident\_travel\_tm\_seconds\_qy | text |
| category\_ladders\_assigned\_quantity | text |
| category\_other\_units\_assigned\_quantity | text |
| citycouncildistrict | bigint |
| communitydistrict | bigint |
| communityschooldistrict | bigint |
| congressionaldistrict | bigint |
| dispatch\_response\_seconds\_qy | bigint |
| engines\_assigned\_quantity | double precision |
| first\_activation\_datetime | timestamp without time zone |
| first\_assignment\_datetime | timestamp without time zone |
| first\_on\_scene\_datetime | timestamp without time zone |
| highest\_alarm\_level | text |
| incident\_borough | text |
| incident\_classification | text |
| incident\_classification\_group | text |
| incident\_close\_datetime | timestamp without time zone |
| incident\_datetime | timestamp without time zone |
| incident\_response\_seconds\_qy | double precision |
| incident\_travel\_tm\_seconds\_qy | double precision |
| index | bigint |
| ladders\_assigned\_quantity | double precision |
| other\_units\_assigned\_quantity | double precision |
| policeprecinct | bigint |
| starfire\_incident\_id | text |
| total\_avg\_dispatch\_response\_seconds\_qy\_per\_borough | double precision |
| total\_avg\_incident\_response\_seconds\_qy\_per\_borough | double precision |
| total\_avg\_incident\_travel\_tm\_seconds\_qy\_per\_borough | double precision |
| total\_resources\_assigned\_quantity | double precision |
| valid\_dispatch\_rspns\_time\_indc | text |
| valid\_incident\_rspns\_time\_indc | text |
| zipcode | bigint |

* + NYC Traffic Table:

|  |  |
| --- | --- |
| **Column Name** | **Data Type** |
| boro | text |
| d | bigint |
| direction | text |
| fromst | text |
| hh | bigint |
| index | bigint |
| m | bigint |
| mm | bigint |
| report\_date\_time | timestamp with time zone |
| requestid | bigint |
| segmentid | bigint |
| street | text |
| tost | text |
| vol | bigint |
| wktgeom | text |
| yr | bigint |

* Connection details and credentials:
  + Server Name: Docker
  + Host Name/Address: fire\_incidents\_db\_container
  + Port: 5432
  + Username: root
  + Password: root

**4. ETL Job Details**

**Airflow DAG Automation:**

* DAG 1 – fire\_incidents\_dag.py:
  + **Extract Function and Task** (extract\_fire\_incidents\_task):
    - **Batch Extraction Process:** The extract\_data function calls extract\_data\_via\_api(), processing data in batches by parsing configured variables. This executes the extract script, detailed in Dag 1 and Dag 2 sections.
    - **Task Instance Management:** A task\_instance is created (kwargs['ti']), leveraging Airflow’s context variables to manage execution efficiently.
    - **XCom for Batch Coordination:** The offset\_counter variable is stored in Airflow XCom, enabling the transform task to retrieve batched extracted data.
    - **Extract Task Definition:** The task is registered as extract\_data\_task, ensuring batch-wise processing and seamless data flow between ETL stages.
  + **Transform Function and Task** (transform\_fire\_incidents\_task):
    - **Retrieve Extracted Data:** The function transform\_data() accesses the task instance (kwargs['ti']) and pulls the offset\_counter from XCom (extract\_data\_xcom). This is used to determine the number of rows in the data.
    - **Apply PySpark Transformations:** Calls main\_pyspark\_transformations(extracted\_data, data\_source), performing transformations on the batch of extracted data for scalable processing. (Found in the Dag 1 and Dag 2 Transform section below)
    - **Define Transform Task:** Registers transform\_fire\_incidents\_task as a PythonOperator, linking it to the transform\_data() function for execution within the Airflow DAG.
  + **Load Function and Task** (load\_fire\_incidents\_task):
    - **Retrieve Transformation Status:** The load\_data() function accesses the task instance (kwargs['ti']) and pulls the XCom variable (transformed\_data\_xcom), which contains a "Transformations Completed" message from the previous task.
    - **Initiate Data Loading:** Calls load\_data\_to\_postgres(), passing the transformation status along with database connection parameters (username, password, host\_name, port, database, tbl\_name, data\_source, schema\_name).
    - **Define Load Task:** Registers load\_fire\_incidents\_task as a PythonOperator, linking it to the load\_data() function for execution within the Airflow DAG.
  + **Task dependencies are configured as follows:** 
    - Ensures sequential batch execution → extract\_data\_task >> transform\_data\_task >> load\_data\_task. (as seen in the image below)

A screenshot of a computer

AI-generated content may be incorrect.

* DAG 2 – traffic\_dag.py:
  + **Define ETL Variables:** Configures API extraction parameters, database credentials, and schema names, ensuring the pipeline processes data in batches (offset=1000).
  + **Set Default DAG Arguments:** Establishes retry logic, execution ownership, and error handling to ensure fault-tolerant ETL execution.
  + **Configure DAG:** Defines 'etl\_nyc\_traffic\_dag', scheduling monthly batch executions (schedule\_interval="0 0 1 \* \*") with a single active run at a time (max\_active\_runs=1).
  + **Extract Stage:**
    - Calls extract\_data\_via\_api() to fetch batched NYC traffic data.
    - Stores the offset counter in XCom (extract\_data\_xcom) to track processed batches dynamically.
    - Registers the task as extract\_data\_task using PythonOperator for execution.
  + **Transform Stage:**
    - Pulls the offset counter from XCom, ensuring batch continuity.
    - Applies PySpark transformations using main\_traffic\_nyc\_pyspark\_transformations(), leveraging distributed processing.
    - Defines the task as transform\_data\_task using PythonOperator for scalable execution.
  + **Load Stage:**
    - Loads transformed batch data into PostgreSQL using load\_data\_to\_postgres(), ensuring efficient storage.
    - Registers the task as load\_data using PythonOperator, finalizing the ETL workflow.
  + **Task Dependencies:**
    - Ensures sequential batch execution → extract\_data\_task >> transform\_data\_task >> load\_data\_task. (as seen in the image below)

A screenshot of a computer

AI-generated content may be incorrect.

* DAG 3 – postgres\_to\_s3\_task\_sensor\_dag.py:
  + **Define ETL Variables:** Configures parameters for database extraction and AWS S3 uploads, ensuring batch-wise processing.
  + **Set Default DAG Arguments:** Establishes retry logic, error handling, and task ownership for fault-tolerant execution.
  + **Configure DAG:** Defines 'postgres\_to\_s3\_task\_sensor\_dag', scheduling batch executions every month (0 0 1 \* \*) with controlled concurrency (max\_active\_runs=1).
  + **Ensure Dependency Completion:**
    - ExternalTaskSensor waits for Fire Incidents DAG (etl\_nyc\_fire\_incidents\_dag) and Traffic Data DAG (etl\_nyc\_traffic\_dag) to complete before proceeding.
    - Uses poke mode to periodically check DAG completion, preventing premature execution.
  + **Extract Stage:**
    - Calls extract\_data\_from\_postgres() to pull batched NYC Fire Incident and Traffic data from PostgreSQL and convert to CSV files.
    - Defines extract\_data\_task using PythonOperator for execution.
  + **Load Stage:**
    - Calls load\_data\_to\_s3() to upload extracted CSV files to an AWS S3 bucket, ensuring scalable storage.
    - Registers load\_data as a PythonOperator, marking the final ETL step.
  + **Task Dependencies:**
    - Ensures sequential execution → wait\_for\_fire\_incidents\_dag, wait\_for\_traffic\_data\_dag >> extract\_data\_task >> load\_data\_task. (as seen in the image below)

A screenshot of a computer

AI-generated content may be incorrect.

* DAG 4 (Not Used) – postgres\_to\_s3\_dag.py:
  + **Set Default DAG Arguments:** Defines retry logic, execution ownership, and failure handling (retries=1, retry\_delay=5 minutes) to ensure robust processing.
  + **Configure DAG:**
    - Establishes 'postgres\_to\_s3\_dag', orchestrating monthly batch execution (schedule\_interval="0 0 1 \* \*").
    - Ensures only one active run at a time (max\_active\_runs=1), preventing overlapping ETL runs.
  + **Extract Stage:**
    - Calls extract\_data\_from\_postgres() to pull batched data from PostgreSQL.
    - Defines the task as extract\_data\_task, using PythonOperator for execution.
  + **Load Stage:**
    - Calls load\_data\_to\_s3() to upload extracted data to an AWS S3 bucket.
    - Registers the task as load\_data, ensuring scalable cloud storage integration.
  + **Task Dependencies:**
    - Ensures sequential execution → extract\_data\_task >> load\_data\_task. (as seen in the image below)

A screenshot of a computer

AI-generated content may be incorrect.

* **ETL Detailed Process for Dag 1 and 2:** Monthly ETL processes capturing the previous month’s data for NYC Fire Incidents and NYC Traffic data.
  + **Extract**
    - **Initialize Client:** The Client variable instantiates the Socrata class, enabling interaction with the open data portal using api\_url and token.
    - **Pagination & Date Filtering:** A while loop iterates through the data in 1,000-row increments (due to NYC Open Data platform limits) using an offset variable. Data extraction is dynamically configured via param\_from and param\_to, defining the date range (e.g., "01/01/2017 to 01/31/2017").
    - **Data Retrieval:** Within the loop, get\_data\_from\_api() fetches data based on the selected DAG (fire\_incident\_data or traffic\_data). A GET request stores the response as JSON. At the end of the entire function it returns offset\_counter variable to be used in the transformation stage.
    - **Retry Mechanism:** get\_data\_from\_api() uses the tenacity library for automatic retries using the retry decorator (up to 5 attempts). Wait times progressively double—from 2 seconds to a maximum of 16 seconds—to handle connection failures efficiently.
    - **Temporary Storage:** Extracted data is temporarily stored in temp/extract within Docker before transformation. Files are removed after processing.
    - **Error Handling:** The while loop runs within a try-except block, ensuring successful execution writes files to temp/extract, while failures are caught and logged.
  + **Transform** – There are two transformation stages because there are two types of data being extracted, NYC Fire Incident Data and NYC Traffic Data.
    - **NYC Fire Incident Data Transformations**
      * **Initialize SparkSession:** Creates a SparkSession instance for processing DataFrames, allocating 4GB of memory each for the driver and executors, with a maximum result size of 4GB.
      * **Read Temporary Files:** A while loop iterates through temp files, appending JSON outputs to read\_json\_data.
      * **Create DataFrame:** A PySpark DataFrame (df) is instantiated for transformations.
      * **Partitioning:** The DataFrame is split into 8 partitions for parallel processing, each handled by a Spark executor for distributed computation.
      * **Persisting Data:** The DataFrame is cached in memory and disk to avoid redundant recomputation, ensuring efficient reuse of intermediate results.
      * **Data Transformations:**
        + **Handle Null Values:** Assigns Borough-specific default values to missing fields (e.g., zipcode, police precinct, district codes).
        + **Convert Numerical Fields:** Casts fields like dispatch\_response\_qy and incident\_response\_seconds\_qy to float types.
        + **Categorize Fields:** Groups numerical values into predefined tiers (e.g., response time categories: Very Low, Low, Medium, High, etc.).
        + **Calculate Averages:** Computes borough-level averages for key time-related metrics (dispatch\_response\_seconds\_qy, incident\_travel\_tm\_seconds\_qy).
        + **Summarize Resources:** Aggregates total resources per incident (engines\_assigned\_quantity, ladders\_assigned\_quantity).
      * **Optimize Partitions:** Uses coalesce() to reduce partition count for efficient data processing.
      * **Write Transformed Data:** Converts the final DataFrame to JSON (json\_string) and stores it in temp/transform for the load stage, removing files once processed.
      * **Cleanup Temporary Files:** A while loop deletes extracted 1,000-row JSON files from temp/extract, as they are no longer needed.
    - **NYC Traffic Data Transformations**
      * **Initialize SparkSession:** Creates a SparkSession instance for processing DataFrames, allocating 4GB of memory each for the driver and executors, with a maximum result size of 4GB.
      * **Read Temporary Files:** A while loop iterates through temp files, appending JSON outputs to read\_json\_data.
      * **Create DataFrame:** A PySpark DataFrame (df) is instantiated for transformations.
      * **Partitioning:** The DataFrame is split into 8 partitions for parallel processing, each handled by a Spark executor for distributed computation.
      * **Persisting Data:** The DataFrame is cached in memory and disk to avoid redundant recomputation, ensuring efficient reuse of intermediate results.
      * **Data Transformations:**
        + **Format Date-Time:** Creates report\_date\_time by concatenating year, month, day, hour, and minute fields.
        + **Standardize Borough Names:** Converts Boro field to uppercase for consistency with NYC Fire Incidents data in SQL joins.
        + **Normalize Staten Island Naming:** Replaces "Staten Island" in boro field with "Richmond / Staten Island" for uniformity.
        + **Convert Volume Field:** Casts vol to integer type for proper numerical processing.
      * **Optimize Partitions:** Uses coalesce() to reduce partition count for efficient data processing.
      * **Write Transformed Data:** Converts the final DataFrame to JSON (json\_string) and stores it in temp/transform for the load stage, removing files once processed.
      * **Cleanup Temporary Files:** A while loop deletes extracted 1,000-row JSON files from temp/extract, as they are no longer needed.
  + **Load** –The load stage takes the transformation file and loads it to the postgres database running on the Docker Container.
    - **Read & Prepare Data:** Reads transformed data from temp/transform into a Pandas DataFrame.
    - **Ensure Date Consistency:** Standardizes date fields (incident\_datetime, first\_assignment\_datetime, etc.) for uniform formatting.
    - **Initialize Database Engine:** Uses sqlalchemy.create\_engine() with predefined parameters (username, password, host, port, and database) from the Airflow DAG.
    - **Create PostgreSQL Table:** Defines fire\_incidents\_tbl, replacing existing tables if needed (if\_exists='replace').
      * engine = create\_engine(f'postgresql://{username}:{password}@{host\_name}:{port}/{database}'))
        + Username = root
        + Password = root
        + Host Name = fire\_incidents\_db\_container
        + Port Number = 5432
        + Database = fire\_incidents\_db
    - **Batch Processing:** Splits the DataFrame into 1,000-row batches using create\_batches\_of\_rows(), storing them in a python list (batches).
    - **Append Data Efficiently:** Iterates over batches, using .to\_sql() to append records into the PostgreSQL table (fire\_incidents\_tbl or nyc\_traffic\_tbl).
    - **Clean Up Temporary Files:** Executes remove\_temp\_file() to delete processed files from temp/transform.
    - **ETL Completion:** Confirms all stages (Extract, Transform, Load) are successfully executed.
* **ETL Detailed Process for Dag 3:** Monitors the completion of the first two DAGs via an **External Task Sensor** (poke mode enabled, timeout = 600ms). Final Step: Extracts transformed data from Postgres, converts it to CSV, and loads it into the AWS S3 bucket.
  + **Extract:**
    - **Initialize Connection:** Calls extract\_data\_from\_postgres() and sets connection variables (database, user, password, host, port, and table names).
    - **Export Data to CSV:** Calls export\_data\_to\_csv(), passing connection parameters.
    - **Database Connection:** Uses psycopg2.connect() to establish connection with PostgreSQL.
    - **Execute Query:** Creates a cursor instance and runs SELECT \* FROM tbl\_name to retrieve all records.
    - **Write to CSV:** If a temp folder doesn’t exist, it is created. The script then defines the CSV path and writes data from cursor.description.
    - **Close Connections:** Closes both the database and cursor connections after writing the file.
    - **Prepare for S3 Upload:** Completes the extraction process by storing the data in a temporary CSV file, ready for uploading to an AWS S3 bucket. Once uploaded, the temporary file is deleted
  + **Load:**
    - **Initialize STS Client:** Creates an AWS STS client (sts\_client) for managing temporary credentials.
    - **Assume IAM Role:** Sets the required RoleArn and RoleSessionName for authentication.
    - **Retrieve Credentials:** Extracts temporary credentials from a directory location in the docker container. The directory was preconfigured in the docker-compose file.
    - **Initialize S3 Client:** Uses boto3.client('s3') to interact with S3, passing temporary credentials (aws\_access\_key\_id, aws\_secret\_access\_key, aws\_session\_token). This ensures secure, short-lived access.
    - **Upload Data to S3:** Calls upload\_file\_to\_s3(), passing data names (nyc\_fire\_incidents\_data, nyc\_traffic\_data) along with the S3 client instance.
    - **Process Files:** Iterates through the data using .upload\_file() inside a loop, transferring files from the temporary folder (extract stage) to S3.
    - **Completion:** Confirms successful upload of both CSV files to AWS S3, marking the end of the load stage.

**AWS**

**AWS S3**

* Bucket structure – Contains one bucket where the two files are dropped:
  + nyc-fire-incidents-s3/exported\_nyc\_fire\_incidents.csv
  + nyc-fire-incidents-s3/exported\_nyc\_trafic\_data.csv
* Event notifications – When the files are dropped into the S3 bucket a notification is triggered and sent to the lambda function destination. The Lambda function will then run Glue Jobs which will take the CSV files, transform the data, and load to Redshift.
  + All object create events: s3:ObjectCreated:\*
    - Notification is sent when any of the object create events occur in the bucket.
    - The destination of the notification is the trigger\_glue\_job\_nyc\_fire\_traffic\_incidents lambda function.
    - IAM Role For Notifications: AWSLambdaBasicExecutionRole-48318d02-1540-424a-ad4e-f243564947a2.
      * Allows to CreateLogStream
      * Allows to PutLogEvents
* **Lambda Trigger:** S3 event notification (s3:ObjectCreated:\*) triggers Lambda function. (Lambda function can be found under AWS/lambda/lambda\_function.py)
  + 1. Gets the file\_name from the event
  + 2. If the file\_name is exported\_nyc\_traffic\_data.csv then it starts two glue jobs.
  + 3. If the file\_name is exported\_nyc\_fire\_incidents\_data.csv then it starts the last glue job.
* **Glue Job 1 (NYC\_Fire\_Traffic\_ETL\_Job):** Loads data from S3 into Redshift, maintaining schema and data types.

**A screenshot of a computer

AI-generated content may be incorrect.**

* **Glue Job 2 (NYC\_Traffic\_Data\_ETL\_Job):** Loads data from S3 into Redshift, maintaining schema and data types.

**A screenshot of a computer

AI-generated content may be incorrect.**

* **Glue Job 3 (Join\_NYC\_Fire\_Incident\_Traffic\_Data\_ETL\_Job):**

A screenshot of a computer

AI-generated content may be incorrect.

* + Extracts both CSV files from S3 Bucket
  + **Derives** date fields from the files.
    - incident\_datetime\_yr\_month\_day: CONCAT(CAST(YEAR(incident\_datetime) AS VARCHAR(10)),'-',CAST(MONTH(incident\_datetime) AS VARCHAR(10)) ,'-',CAST(DAY(incident\_datetime) AS VARCHAR(10)))
    - report\_date\_time\_yr\_month\_day: CONCAT(CAST(YEAR(report\_date\_time) AS VARCHAR(10)),'-',CAST(MONTH(report\_date\_time) AS VARCHAR(10)) ,'-',CAST(DAY(report\_date\_time) AS VARCHAR(10)))
  + **Aggregates** the average volume of traffic by report\_date\_time\_yr\_month\_day and borough. To get daily averages per NYC Borough.
  + A screenshot of a computer

    AI-generated content may be incorrect.
  + **Joins** tables via SQL using Borough & derived date field as joining keys.

select \*

From nyc\_fire\_incidents

Inner Join nyc\_traffic\_incidents

On nyc\_fire\_incidents.incident\_borough = nyc\_traffic\_incidents.boro

AND nyc\_fire\_incidents.incident\_datetime\_yr\_month\_day = nyc\_traffic\_incidents.report\_date\_time\_yr\_month\_day;

A screenshot of a computer

AI-generated content may be incorrect.

* + Applies schema changes before loading the final table into Amazon Redshift. Truncates the target table before loading data to table.

A screenshot of a computer

AI-generated content may be incorrect.

* + IAM Role for Glue Jobs: AWSServiceRole\_S3\_Redshift\_Role\_2
    - AmazonRedshiftFullAccess
    - AmazonS3FullAccess
    - AWSGlueConsoleFullAccess
    - AWSGlueServiceRole
    - AWSKeyManagementServicePowerUser
    - SecretsManagerReadWrite

**5. Power BI Integration**

* Connection settings for Redshift.
* Schema mapping for visualization.
* Overview of the 3 dashboards created.

**6. Logging & Monitoring**

* Airflow DAG execution logs.
* AWS CloudWatch monitoring for Lambda and Glue jobs.
* Postgres logs (via Docker).
* Glue job execution tracking.

**7. Performance Considerations**

* Optimization techniques used (partitioning, indexing).
* Scalability considerations.
* Potential bottlenecks and mitigation strategies.

**8. Future Enhancements**

## Summary of docker-compose settings

**Docker Configuration Summary**

* **Docker Compose Version:** 3.8
* **Services:**
  + **Postgres Database (fire\_incidents\_db\_container)**
    - Image: postgres:13
    - Environment Variables:
      * POSTGRES\_USER: root
      * POSTGRES\_PASSWORD: root
      * POSTGRES\_DB: fire\_incidents\_db
    - Data Persistence: Mounted volume (./fire\_incidents\_postgres:/var/lib/postgresql/data)
    - Ports: 5432:5432
    - Network: pg-network-fire-incidents
  + **PgAdmin (pgadmin-fire-incidents-container)**
    - Image: dpage/pgadmin4
    - Environment Variables:
      * PGADMIN\_DEFAULT\_EMAIL: admin@admin.com
      * PGADMIN\_DEFAULT\_PASSWORD: root
    - Ports: 8080:80
    - Network: pg-network-fire-incidents
* **Network Configuration:**
  + pg-network-fire-incidents (Externally defined network)

**Airflow Configuration Summary**

* **Airflow Image:** airflow-custom:latest
* **Executor Type:** CeleryExecutor
* **Database Connection:**
  + SQLAlchemy: postgresql+psycopg2://airflow:airflow@postgres/airflow
  + Celery Result Backend: db+postgresql://airflow:airflow@postgres/airflow
  + Celery Broker URL: redis://:@redis:6379/0
* **Scheduler Health Check:** Enabled (AIRFLOW\_\_SCHEDULER\_\_ENABLE\_HEALTH\_CHECK=true)
* **Security & Credentials:**
  + Uses **Fernet encryption** (AIRFLOW\_\_CORE\_\_FERNET\_KEY='')
  + **AWS credentials** stored in /opt/airflow/.aws/credentials
* **Volumes & Mounted Paths:**
  + DAGs: /opt/airflow/dags
  + Logs: /opt/airflow/logs
  + Config files: /opt/airflow/config
  + Plugins: /opt/airflow/plugins
  + Requirements: /requirements.txt
  + Entrypoint script: /entrypoint.sh
  + AWS Credentials: ~/.aws:/opt/airflow/.aws:ro
* **User Permissions:** Airflow UID = 50000, Docker Group = 1001

**Service Dependencies**

* **Postgres Database (airflow service)**
  + Image: postgres:13
  + Environment:
    - POSTGRES\_USER=airflow
    - POSTGRES\_PASSWORD=airflow
    - POSTGRES\_DB=airflow
  + Volume: postgres-db-volume:/var/lib/postgresql/data
  + Health Check:
    - Command: pg\_isready -U airflow
    - Interval: 10s
    - Retries: 5
    - Start Period: 5s
  + Restart Policy: always
  + Network: pg-network-fire-incidents
* **Redis Service**
  + Image: redis:7.2-bookworm
  + Exposed Ports: 6379
  + Health Check:
    - Command: redis-cli ping
    - Interval: 10s
    - Timeout: 30s
    - Retries: 50
    - Start Period: 30s
  + Restart Policy: always
  + Network: pg-network-fire-incidents

**Airflow Components**

* **Airflow Webserver**
  + Command: airflow webserver
  + Entrypoint Script: /entrypoint.sh
  + Ports: 9090:8080
  + Health Check:
    - Command: curl --fail http://localhost:9090/health
    - Interval: 30s
    - Timeout: 10s
    - Retries: 5
    - Start Period: 30s
  + Restart Policy: always
  + Depends On:
    - **Redis** (healthy state required)
    - **Postgres** (healthy state required)
    - **Airflow Init** (service\_completed\_successfully)
  + Network: pg-network-fire-incidents
* **Airflow Scheduler**
  + Command: airflow scheduler
  + Entrypoint Script: /entrypoint.sh
  + Health Check:
    - Command: curl --fail http://localhost:8974/health
    - Interval: 30s
    - Timeout: 10s
    - Retries: 5
    - Start Period: 30s
  + Restart Policy: always
  + Depends On:
    - **Redis** (healthy state required)
    - **Postgres** (healthy state required)
    - **Airflow Init** (service\_completed\_successfully)
  + DNS Configuration:
    - 8.8.8.8 (Google DNS)
    - 1.1.1.1 (Cloudflare DNS)
  + Network: pg-network-fire-incidents
  + Extra Hosts: "spark-master:127.0.0.1"
* **Airflow Worker**
  + Command: airflow celery worker
  + Entrypoint Script: /entrypoint.sh
  + Volumes:
    - /var/run/docker.sock:/var/run/docker.sock
    - ./dags:/opt/airflow/dags
    - ./logs:/opt/airflow/logs
    - ~/.aws:/opt/airflow/.aws:ro
    - ./plugins:/opt/airflow/plugins
    - ./requirements.txt:/requirements.txt
    - ./entrypoint.sh:/entrypoint.sh
  + Health Check:
    - Command:
      * celery --app airflow.providers.celery.executors.celery\_executor.app inspect ping -d "celery@$${HOSTNAME}" || celery --app airflow.executors.celery\_executor.app inspect ping -d "celery@$${HOSTNAME}"
    - Interval: 30s
    - Timeout: 10s
    - Retries: 5
    - Start Period: 30s
  + Restart Policy: always
  + Depends On:
    - **Redis** (healthy state required)
    - **Postgres** (healthy state required)
  + Network: pg-network-fire-incidents
* **Airflow Triggerer**
  + Command: triggerer
  + Health Check:
    - Command: airflow jobs check --job-type TriggererJob --hostname "$${HOSTNAME}"
    - Interval: 30s
    - Timeout: 10s
    - Retries: 5
    - Start Period: 30s
  + Restart Policy: always
  + Depends On:
    - **Redis** (healthy state required)
    - **Postgres** (healthy state required)
    - **Airflow Init** (service\_completed\_successfully)
  + Network: pg-network-fire-incidents
* **Airflow Init**
  + Entry Point: /bin/bash
  + Command:
    - Verifies system resources (memory, CPU, disk space)
    - Creates directories (/sources/logs, /sources/dags, /sources/plugins)
    - Adjusts permissions (chown -R "${AIRFLOW\_UID}:0")
    - Initializes Airflow database migrations (\_AIRFLOW\_DB\_MIGRATE=true)
    - Creates default user (\_AIRFLOW\_WWW\_USER\_CREATE=true)
  + Volumes:
    - ${AIRFLOW\_PROJ\_DIR:-.}:/sources
  + Network: pg-network-fire-incidents
* **Airflow CLI**
  + Command: bash -c airflow
  + Debug Profile Enabled
  + Network: pg-network-fire-incidents
* **Flower Service**
  + Command: celery flower
  + Ports: 5555:5555
  + Health Check:
    - Command: curl --fail http://localhost:5555/
    - Interval: 30s
    - Timeout: 10s
    - Retries: 5
    - Start Period: 30s
  + Restart Policy: always
  + Depends On:
    - **Redis** (healthy state required)
    - **Postgres** (healthy state required)
    - **Airflow Init** (service\_completed\_successfully)
  + Network: pg-network-fire-incidents
  + Profile-Based Activation (docker-compose --profile flower up)

**Volumes & Networks**

* **Persistent Volume:** postgres-db-volume
* **Network:** pg-network-fire-incidents (external)