Chicago Crime Data ETL Pipeline Project

Nishan Khanal

April 24, 2025

Contents

1 Introduction **2**

2 Technology Stack **3**

3 Project Motivation and Planning **3**

4 Airflow Automation and Orchestration **4**

5 Pipeline Overview **5**

6 Data Extraction **6**

7 Data Transformation **8**

8 Data Loading **10**

9 Analysis & Visualization **13**

10 Looker Studio Integration **15**

11 Monitoring and Logging **17**

12 Security and Secrets Management **19**

13 Conclusion and Future Work **20**

References **21**

Abstract

This report describes the disign developement and deployment of a fully automated ETL pipeline orchestrated using Airflow to aid in the analysis and visualization of the Chicago crime dataset. The pipeline is designed to be lightweight, and fault tolorent - ready to be deployed in a prdocution envrironment. It integrates various components including extracting data from the official Chicago open data API, applying different transformations to clean and standardize the data, enriching it with spatial context (i.e., missing community areas), and loading the processed data efficiently into a PostgreSQL database hosted on Aiven (DigitalOcean). It is deployed on a Google Cloud micro VM (e2-micro instance) to ensure persistent, cloud-based data availability independent of local or VM storage with a robust logging and monitoring mechanism.

1 Introduction

Crime prediction and public safety analysis rely heavily on clean, timely, and spatially accurate data. This project aims to make that process easir by automating the ETL (Extract, Transform, Load) pipeline for the Chicago Crime dataset. It fouses on periodically fetching the latest crime data from the Chicogo's official data api, cleaning and standardizing it, enriching the data with spatial context (i.e., missing community areas), and storing the processed data in a reliable, queryable cloud database. The pipeline is designed to be lightweight, fault-tolerant, and ready for production deployment. he goal is to create a system that can continuously fetch the latest crime reports from Chicago's official data portal, clean and standardize them, enrich the data with spatial context (i.e., missing community areas), and store the processed data in a reliable, queryable cloud database.

This project focuses on setting up an ETL pipeline that can continuously fetch the latest crime reports from Chicago's official data portal, clean and standardize them, enrich the data with spatial context (i.e., missing community areas), and store the processed data in a reliable, queryable cloud database.

Dataset Choice

Based on my proposal I've built the ETL pipeline for the **Chicago Crime dataset**. This dataset contains the reported incidents of crime in the City of Chicago from 2001 to present (minus the most recent 7 days). The dataset is deidentified in the spatial domain, i.e, the addresses are only accurate to the block level. This dataset will be helpful for analyzing crime patterns, and trends, and for predicting future crime hotspot. It is available in the official Chicago open data portal here. The dataset is updated daily, and the API allows for incremental updates, making it ideal for a continuous ETL pipeline.

A supplementary dataset **Chicago Community Areas** was also used which aided in the visualization of the crime location in different communities. This dataset contains geographic information (GIS) about the different communities in Chicago. This dataset is available in the official Chicago open data portal here. This dataset was also used to impute the missing community area values in the crime dataset.

The core goals of this project were:

- To automate the end-to-end ETL process without manual intervention
- To optimize the pipeline for execution on limited infrastructure (micro VM)
- To decouple the pipeline from local resources for availability and maintainability
- To ensure scalability, observability, and auditability in real-world conditions

Throughout this document, each component of the system will be described in detail, along with rationale, challenges, and architectural decisions made along the way.

2 Technology Stack

Languages and Frameworks

- Python 3.10
- Apache Airflow (version 2.7.2)
- Pandas, GeoPandas, sodapy , Psycopg2
- PostgreSQL (Aiven Free Tier)

Infrastructure

- Google Cloud VM (e2-micro) running Ubuntu
- Aiven PostgreSQL database (hosted on DigitalOcean)
- GitHub for version control. (Link)

Development Tools

- VS Code for development and SSH access
- Tmux for persistent process handling on the VM
- Systemd for automating Airflow as a background service

3 Project Motivation and Planning

Initially, I began working with the dataset on my local machine. I used `ngrok` to tunnel my PostgreSQL port and connect external services (like Google Looker Studio) for demo purposes. While this worked, it was not suitable for a long-running, always-on pipeline:

- **Ngrok is ideal for development**, but the connection times out frequently unless you're on a paid plan.

- Relying on a **local database** meant the pipeline couldn't run unless my laptop was powered and connected.

To overcome this, I migrated to a **Google Cloud micro VM (e2-micro)** — a lightweight and cost-effective instance — to host my Airflow-based pipeline. I also moved my database to **Aiven's free-tier PostgreSQL instance (hosted on DigitalOcean)** to ensure persistent, cloud-based data availability independent of local or VM storage.

4 Airflow Automation and Orchestration

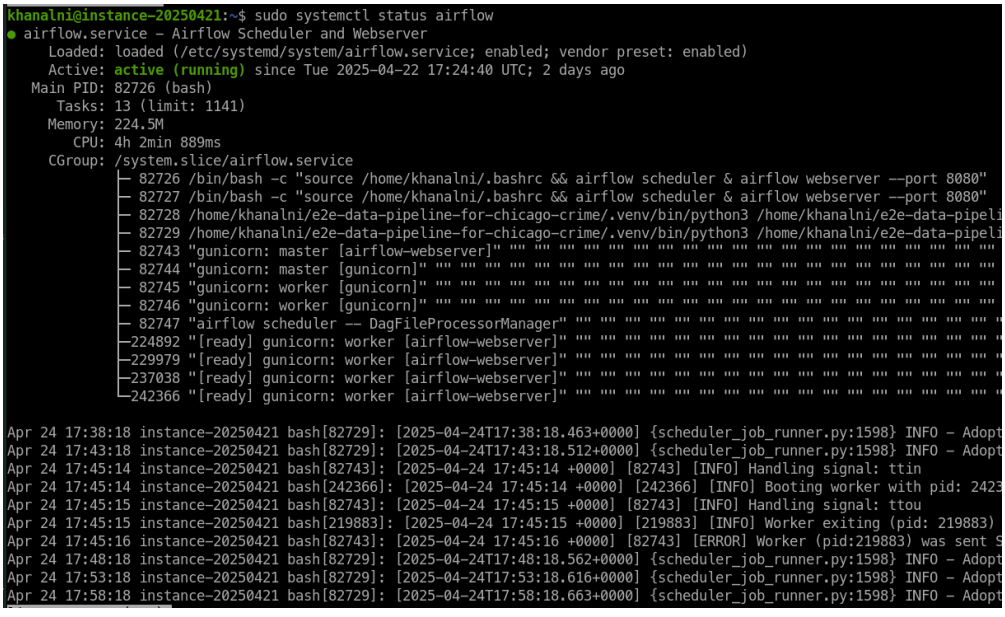
Airflow was chosen to automate and orchestrate the pipeline due to its flexibility, native scheduling, and observability features. Tasks are defined as Python callables, and dependencies are managed explicitly. The DAG consists of three tasks — `extract`, `transform`, and `load` — each executing the respective Python functions.

Airflow DAG Design

The DAG is scheduled to run daily. A simple linear dependency is enforced: `extract` → `transform` → `load`.

Systemd Automation

To ensure Airflow runs continuously without manual intervention, a systemd service was created to launch the scheduler and webserver in the background when the VM boots.



(a) Systemd service status for Airflow

Figure 1: Automation of background execution using systemd

5 Pipeline Overview

Figure 2 shows the ETL workflow. The data will be extracted from the Chicago Crime dataset and the Chicago Community Areas dataset. The data was be transformed through cleaning, normalization, enrichment, and data type conversion. The transformed data was then loaded into a PostgreSQL database hosted on Aiven. Finally, the data was be analyzed and visualized using Python libraries and GCP Looker Studio.

Figure 3 shows the Airflow DAG for the ETL pipeline. The DAG is scheduled to run daily, and it consists of three tasks: `extract`, `transform`, and `load`. Each task is defined as a Python callable.

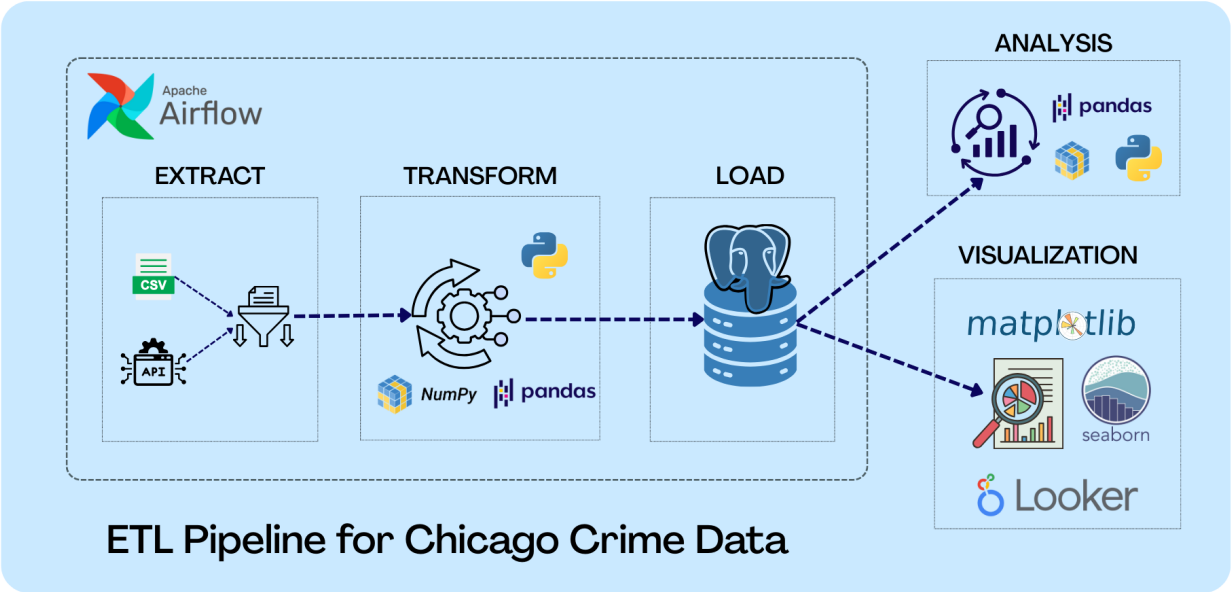


Figure 2: ETL Workflow

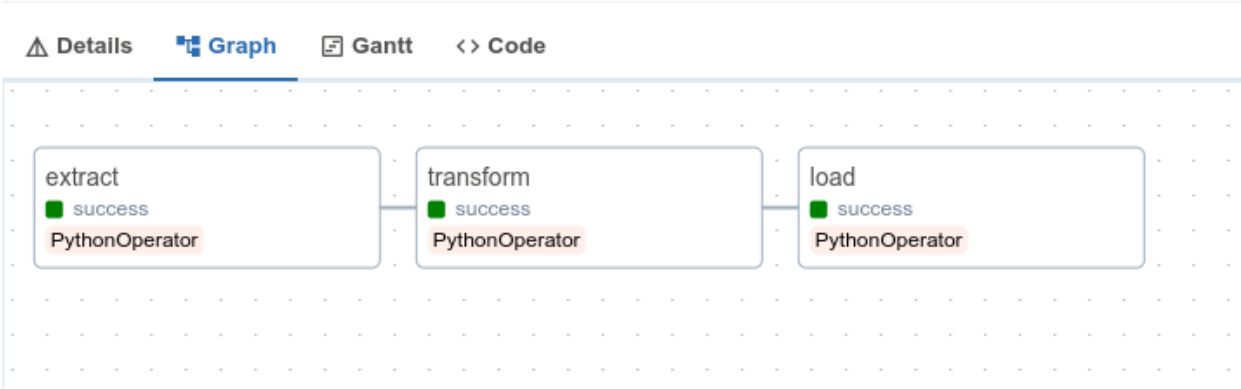


Figure 3: Airflow DAG for ETL pipeline

6 Data Extraction

The `extract()` function is the first step in the ETL pipeline. Its responsibility is to connect to the Socrata Open Data API hosted by the City of Chicago, retrieve all crime records that have been added or updated since the last successful load, and return them in memory as a list of dictionaries as shown in figure 4. Figure 4 shows the logs of the extract phase with its successfull completion.

API Integration with Socrata

The City of Chicago crime dataset is accessible via an open data API. This function authenticates using an application token and sends paginated requests, each limited to a batch size (default: 1000). A retry mechanism with **exponential backoff** ensures fault tolerance in the event of timeouts or transient network errors.

Figure 4: Output of the extract phase

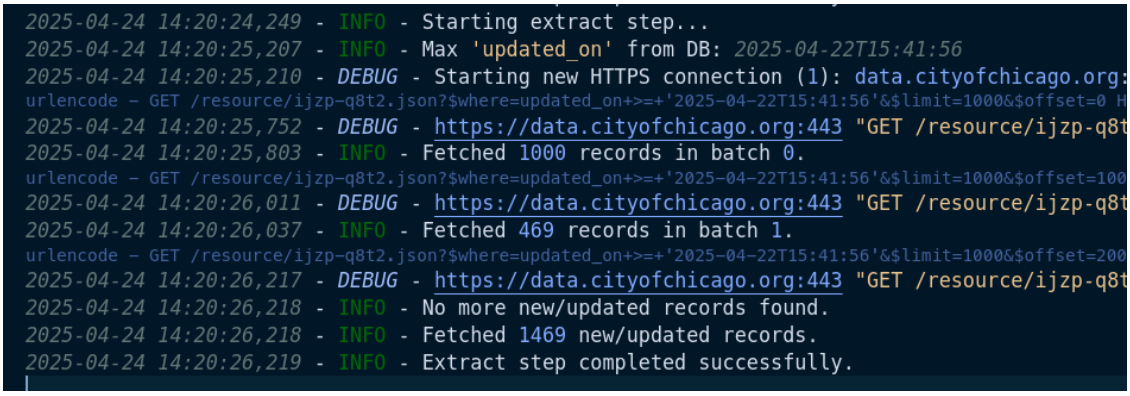


Figure 5: logs of the extract phase after of successfull execution

Tracking Incremental Updates

To avoid re-fetching already-processed data, the extractor first queries the PostgreSQL database to find the maximum `updated_on` timestamp in the `crimes` table. This value is used in a `WHERE` clause to fetch only new or modified records.

Error Handling

The function includes structured exception handling. The logs generated by retry logic is also demonstrated in figure6:

- Raises `RuntimeError` if database connection fails or returns an invalid timestamp
- Raises `RuntimeError` if the API client fails to initialize
- Raises `RuntimeError` if retry limit is reached during a batch request
- Raises `ValueError` if no data is returned (indicating a possible upstream issue)

Design Decisions

- Raising exceptions (instead of returning empty lists) ensures that Airflow correctly marks the task as failed
- Pagination and retry logic are encapsulated in a nested `fetch_with_retry()` function for clarity and reuse. Pagination can also be seen on figure 5 with the first page fetching 1000 records and the second page fetching 469 records.
- Logs are written for each API call and each critical step, including retry attempts and batch sizes

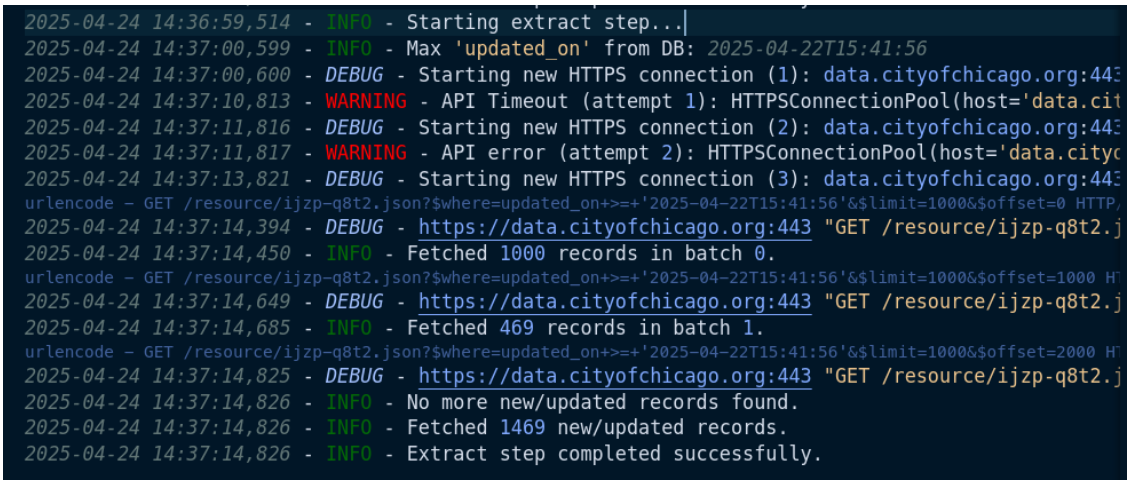


Figure 6: Demonstration of system behavior during timeouts and transient n/w failure

7 Data Transformation

The `transform()` function is utilized to convert raw API data into clean, enriched,and structured form to be loaded into the database. This stage comprises rigorous datacleaning, standardization, and geospatial enrichment.

Input Validation and Schema Enforcement

The function first validates the structure of the input `DataFrame`. If the DataFrame is empty or `None`, it raises a `ValueError`, immediately halting the DAG. After that, it ensures all the expected columns are present in the data frame. Then only the columns that are present in the database schema are selected, which is defined by the expected columns

Data Type Casting

To enforce consistency and avoid downstream type mismatch errors, the function calls `cast_column_types()`, which converts each column to a specific type:

- `Int64` for primary identifiers and codes
- `Float64` for coordinates
- `string` for categorical fields
- `boolean` for flags like `arrest` and `domestic`
- `datetime64[ns]` for `date` and `updated_on`

String Normalization

For all the string columns, leading/trainling whitespaces are stripped off, internal whitespaces are collapsed using regex and finally are converted to uppercase. This ensures uniform grouping and filtering duing the analytics and visualization phase.

Spatial Imputation of Missing Community Areas

Some records lack the `community_area` field, which is critical for geographic analysis if we are to consider community areas as nodes for feeding the data to a graph nerural network. These are imputed based on the crime's `x_coordinate` and `y_coordinate` using a geospatial join with a geojson file of Chicago's community area boundaries. If the location of a crime incidden falls inside the area of a community area, it's `community_area` attribute is imputed using the Community Area it falls inside. For example, in figure 7, we can see that the crime incident with `case_number=JJ219799` falls inside the community area `51`, sot its `community_area` attribute is set as `51`

Row Cleaning and Deduplication

The function removes rows based on configurable rules. For now, the function can remove all the rows for which any of the specified columns(drop_if_any_null_columns) have missing values, and all the rows for which all the values in the specified columns(drop_if_all_null_columns) are missing:

- Drops rows with missing values in critical columns (optional logic)
- Drops rows where all coordinate fields are missing
- Deduplicates based on all columns except `id`

Dropped rows are saved to audit files named `audit_missing_drops_{timestamp}.csv` and `audit_duplicate_drops_{timestamp}.csv`, making the cleaning process fully transparent and reversible.

In figure 8, we can see the output of the transform fucntion for a select group of columns. Only these columns were included in the figure due to the space constraint. Here, we can see that, for some rows, latitude and longitude columns have missing values even after transformation; this is by design. Although the code to remove rows based on missing values of specific columns is present, I didn't specify any columns to check for null values for this task because these rows can still be important in the downstream task because they have the non-null values for other spatial columns like community_area, beat, district, etc.

Final Checks

If all rows are dropped during the cleaning process, the function raises a `ValueError`, which signals Airflow to halt the DAG. Otherwise, it logs the final row count as shown in figure 9 and returns the cleaned DataFrame for loading.

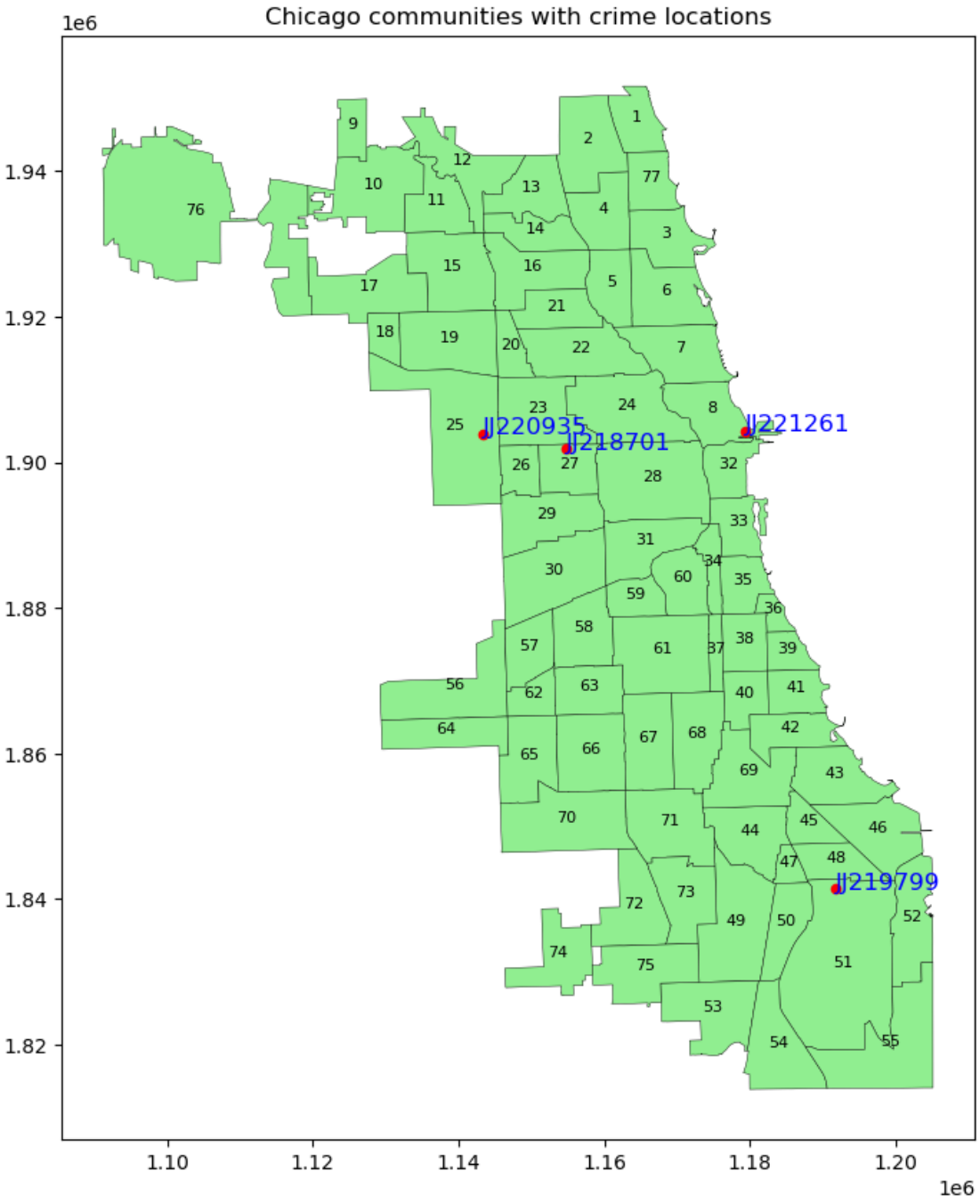


Figure 7: Demonstration of `community_area` imputation

8 Data Loading

The `load()` function performs the final step in the ETL pipeline: inserting the cleaned and transformed data into the PostgreSQL database hosted on Aiven. Since, I used this function to load both the continuous data from api and the historical data (which is massive with 8 million rows), the function was optimised for bulk inserts at the same time supporting

```
df[['id','case_number','date', 'community_area', 'primary_type', 'latitude', 'longitude']].head(10)
✓ 0.0s
```

	id	case_number	date	community_area	primary_type	latitude	longitude
0	13812394	JJ226208	2023-11-29 00:00:00	49	DECEPTIVE PRACTICE	NaN	NaN
1	13805266	JJ217373	2025-04-14 07:29:00	25	THEFT	41.908453	-87.765881
2	13812197	JJ225931	2024-03-07 00:01:00	6	OTHER OFFENSE	NaN	NaN
3	13806051	JJ218345	2025-04-14 18:40:00	35	NARCOTICS	41.841727	-87.609351
4	13805425	JJ217539	2025-04-14 10:10:00	49	ASSAULT	41.721604	-87.632724
5	13805354	JJ217424	2025-04-14 08:06:00	46	ASSAULT	41.751112	-87.557730
6	13806379	JJ218913	2025-04-14 19:30:00	21	THEFT	41.943984	-87.712635
7	13806016	JJ218058	2025-04-14 16:20:00	15	ASSAULT	41.941173	-87.746930
8	13805992	JJ218432	2025-04-14 21:23:00	29	NARCOTICS	41.862408	-87.714515
9	13805766	JJ217970	2025-04-14 13:30:00	44	ROBBERY	41.747171	-87.605120

Figure 8: Output of the transform function

```
2025-04-24 15:38:44,014 - INFO - Starting transform step...
2025-04-24 15:38:44,014 - INFO - Transforming raw data with 1469 rows.
2025-04-24 15:38:44,059 - INFO - Data type casting complete.
2025-04-24 15:38:44,087 - INFO - Cleaned string columns: ['case_number', 'block', 'iucr', 'primary_type'
2025-04-24 15:38:44,088 - INFO - No missing community_area values. Skipping spatial imputation.
2025-04-24 15:38:44,089 - INFO - No rows dropped for null values.
2025-04-24 15:38:44,112 - INFO - Dropped 25 duplicate rows based on columns: ['case_number', 'date', 'bl
2025-04-24 15:38:44,112 - INFO - Row count after dropping duplicates: 1444
2025-04-24 15:38:44,117 - INFO - Dropped 0 outlier rows based on columns: ['x_coordinate', 'y_coordinate
2025-04-24 15:38:44,123 - INFO - Transform complete. Final row count: 1444
```

Figure 9: Logs generated by the transform function

UPSERT behavior to ensure that the existing records are updated rather than duplicated.

Preparation for Insertion

Before insertion, the function converts all null-like values (e.g., `pd.NA`, `NaN`) to Python-native `None` values using the helper function `prepare_for_postgres_insertion()`. This avoids type mismatches when communicating with the database driver.

Conflict Detection

To log how many rows will be inserted versus updated, the function queries the destination table for all existing `id` values in the current batch. It compares this list against the incoming data to log the following as shown in figure 10:

- Number of records that will be inserted (new)
- Number of records that will be updated (conflicts)

Batching and Performance

The function uses `psycopg2.extras.execute_values()` to insert data in batches (default: 50,000 rows per batch). This approach is significantly faster than inserting rows individually.

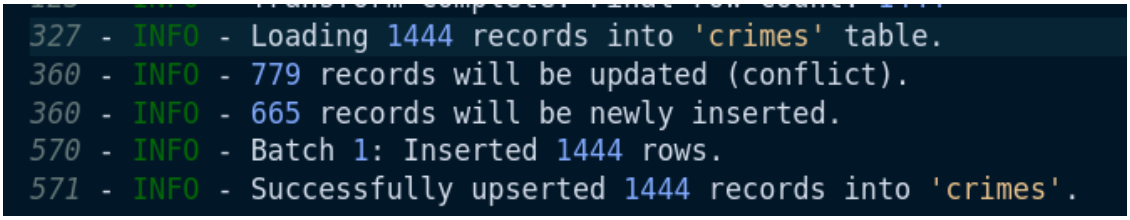


Figure 10: Log output showing insert vs. update count

If a batch fails, it is rolled back and written to a CSV file named `failed_batch_{n}_{timestamp}.csv` for later inspection or reprocessing.

Error Handling and Robustness

The function raises exceptions under the following conditions:

- Failure to connect to the database
- Failure in all batches (complete load failure)
- Partial load success (some batches failed): a `RuntimeError` is raised after successful ones are committed, signaling Airflow while preserving inserted data

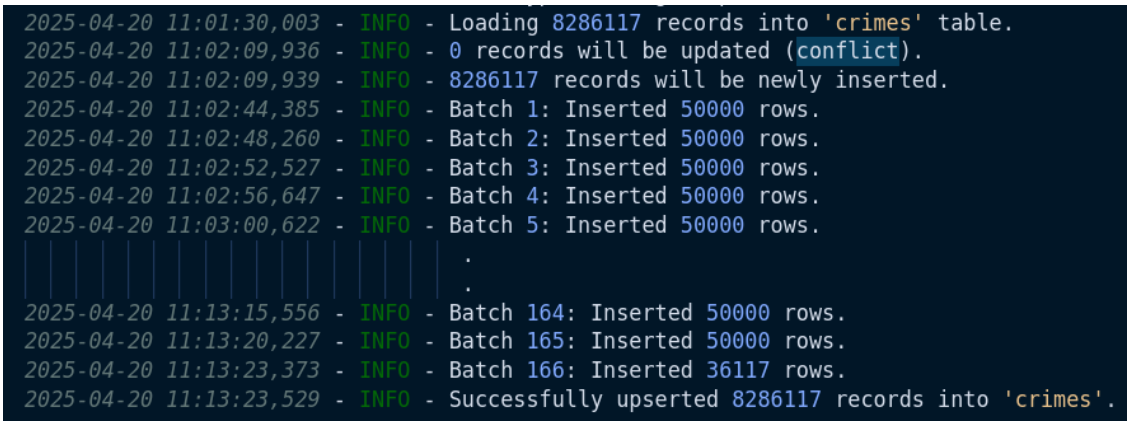


Figure 11: Log output showing multiple batch insert for historical data

Figure 11 show the logs of loading the entire historical crime data using the load() function. This shows that the function is capable of loading not only the data from the api but also huge historical data. I've manually trancated the logs to show only the first and last few batches.

Post-Load Outcome

If at least one batch succeeds, the function logs the total number of records inserted and updated as shown in figure 10, and returns cleanly. This enables downstream systems (e.g., dashboards or reports) to use the updated data immediately after load completion.

Figure 12 shows the crime incidents orderd by date column for a select few columns before the latest load() operation and figure 13 shows the crime incidents after the latest load() operation.

Query Query History

```
1 select id, case_number, date, primary_type, community_area, latitude, longitude, updated_on from crimes
2 order by date desc
3 limit 1000;
```

Data Output Messages Notifications

Showing rows: 1 to 1000 Page No: 1

	id [PK] bigint	case_number character varying (20)	date timestamp without time zone	primary_type character varying (100)	community_area integer	latitude double precision	longitude double precision	updated_on timestamp without time zone
1	13809488	JJ222713	2025-04-14 00:00:00	THEFT	43	41.760160698	-87.583520062	2025-04-21 15:41:50
2	13805877	JJ218300	2025-04-14 00:00:00	OTHER OFFENSE	28	41.862218265	-87.639087164	2025-04-21 15:41:50
3	13805733	JJ218142	2025-04-14 00:00:00	DECEPTIVE PRACTICE	24	41.911223127	-87.694568274	2025-04-21 15:41:50
4	13807498	JJ220238	2025-04-14 00:00:00	THEFT	60	41.847565598	-87.645749253	2025-04-21 15:41:50
5	13806309	JJ218683	2025-04-14 00:00:00	MOTOR VEHICLE THEFT	4	41.962081333	-87.676018515	2025-04-21 15:41:50
6	13805209	JJ217448	2025-04-14 00:00:00	THEFT	28	41.880194	-87.665148099	2025-04-21 15:41:50
7	13809348	JJ221677	2025-04-14 00:00:00	OFFENSE INVOLVING CHILDREN	58	41.814396114	-87.70374836	2025-04-21 15:41:50
8	13806011	JJ217958	2025-04-14 00:00:00	THEFT	16	41.950426051	-87.736709828	2025-04-21 15:41:50
9	13805308	JJ217506	2025-04-14 00:00:00	BATTERY	41	41.801838953	-87.599791667	2025-04-21 15:41:50

Figure 12: State of crimes table before latest load() operation

Query Query History

```
1 select id, case_number, date, primary_type, community_area, latitude, longitude, updated_on from crimes
2 order by date desc
3 limit 1000;
```

Data Output Messages Notifications

Showing rows: 1 to 1000 Page No: 1

	id [PK] bigint	case_number character varying (20)	date timestamp without time zone	primary_type character varying (100)	community_area integer	latitude double precision	longitude double precision	updated_on timestamp without time zone
1	13808203	JJ220960	2025-04-16 00:00:00	ASSAULT	25	41.873185004	-87.743795434	2025-04-23 15:41:12
2	13807360	JJ219954	2025-04-16 00:00:00	OTHER OFFENSE	66	41.762243047	-87.691741219	2025-04-23 15:41:12
3	13808555	JJ221312	2025-04-16 00:00:00	THEFT	1	42.016119861	-87.664120865	2025-04-23 15:41:12
4	13807268	JJ219819	2025-04-16 00:00:00	BATTERY	7	41.929723017	-87.651331865	2025-04-23 15:41:12
5	13808467	JJ221456	2025-04-16 00:00:00	THEFT	7	41.93247541	-87.663911788	2025-04-23 15:41:12
6	13808007	JJ220791	2025-04-16 00:00:00	OTHER OFFENSE	7	41.930925342	-87.66351272	2025-04-23 15:41:12
7	13807767	JJ220606	2025-04-16 00:00:00	INTERFERENCE WITH PUBLIC OFFICER	45	41.751416835	-87.592747031	2025-04-23 15:41:12
8	13807192	JJ219811	2025-04-16 00:00:00	BATTERY	46	41.728852906	-87.554118677	2025-04-23 15:41:12
9	13808030	JJ220126	2025-04-16 00:00:00	THEFT	67	41.764264563	-87.667444617	2025-04-23 15:41:12

Figure 13: State of crimes table after latest load() operation

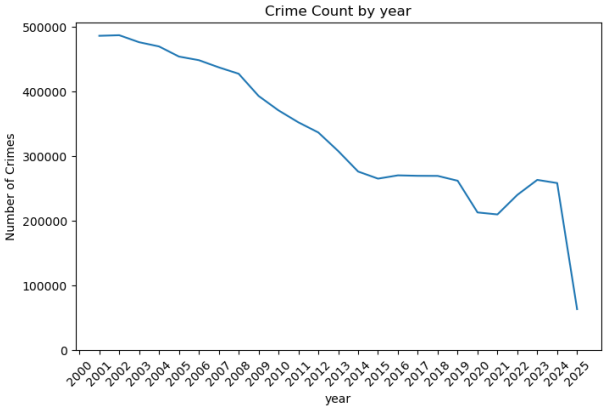
9 Analysis & Visualization

Exploratory Data Analysis

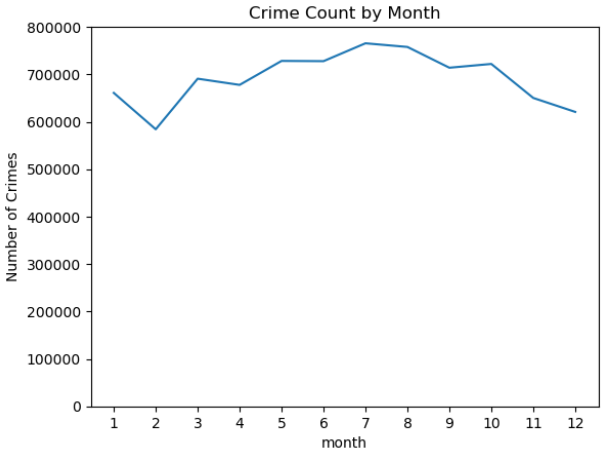
I performed exploratory data analysis after completing the ETL phase for the chicago crime data. This can be viewed in the ETL.ipynb notebook in the github repository. Few insights drawn from the analysis are listed below along with their supporting visualizations:

- The overall crime count has been decreasing from 2001 to 2024 as shown in figure 14a.
- There is seasonal trend present as shown by the figure 14b where the number of crimes peaks around summer.

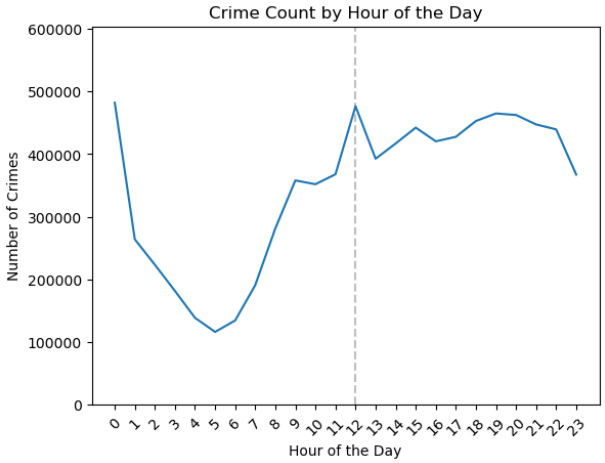
- From figure 14c, we can see that crime incidents clearly depend on the time of the day and number of crime peaks aroudn 12 PM.
- Theft is the most common crime type base on the figure 15a which shows the top 10 crime types by crime count.
- Street is the most common location where crime occurs based on the figure 15c visualizing the top 10 location by crime count.
- Among the top 10 crime type by crime counts `Narcotics` has the highest arrest percentage as shown in figure 15b.
- Figure 15d shows the distribution of crimes in 2024 in chicago.



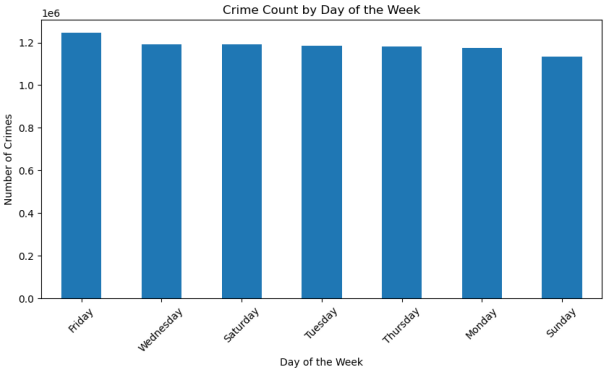
(a) Overall crime count over the years



(b) Seasonal trend of crime count

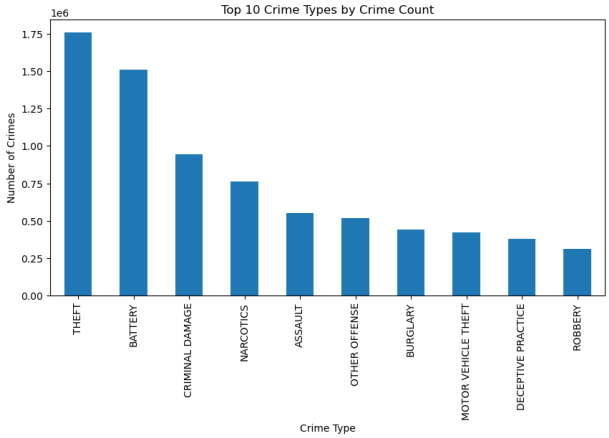


(c) Seasonal trend of crime count by hour of day

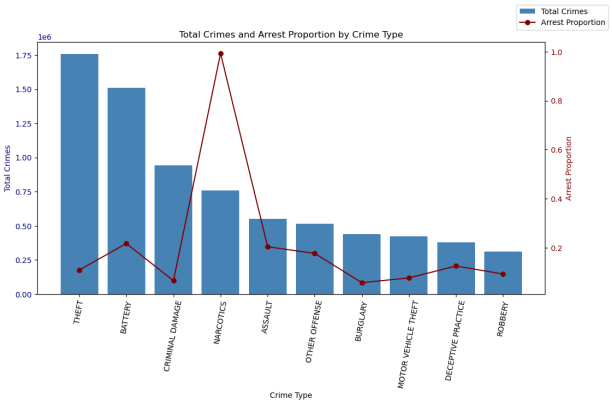


(d) Seasonal trend of crime count by day of the week

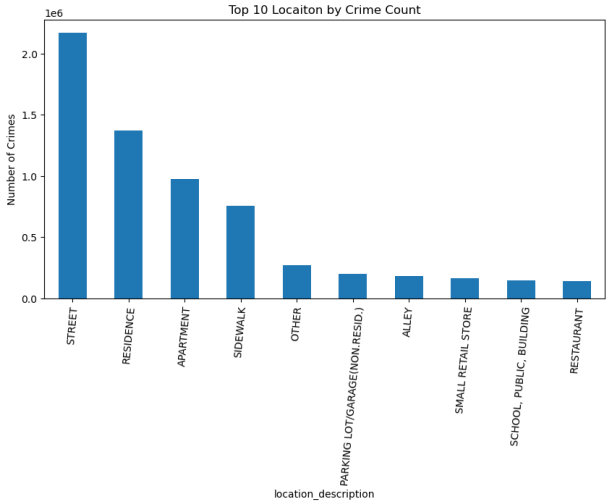
Figure 14: Crime count trends across different time dimensions



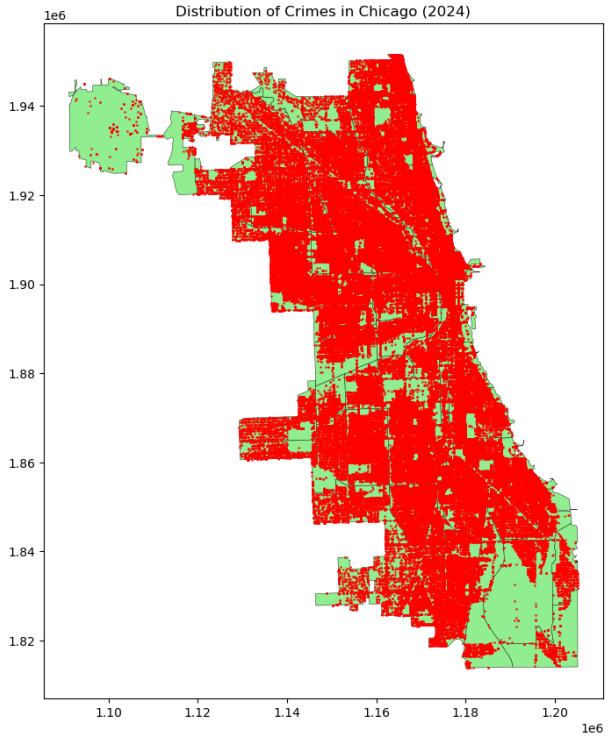
(a) Top 10 crime types by crime count



(b) Arrest percentage by crime type



(c) Top 10 locations by crime count



(d) Distribution of crime incidents in 2024

Figure 15: Crime data visualizations: crime types, locations, arrest proportions, and 2024 distribution

10 Looker Studio Integration

To make the transformed and enriched crime data easily consumable, I connected the PostgreSQL database (hosted on Aiven) to Google Looker Studio to create dynamic visual dashboards. The dashboard allows users to explore crime trends, types, and locations interactively. It can be accessed from this link: here.

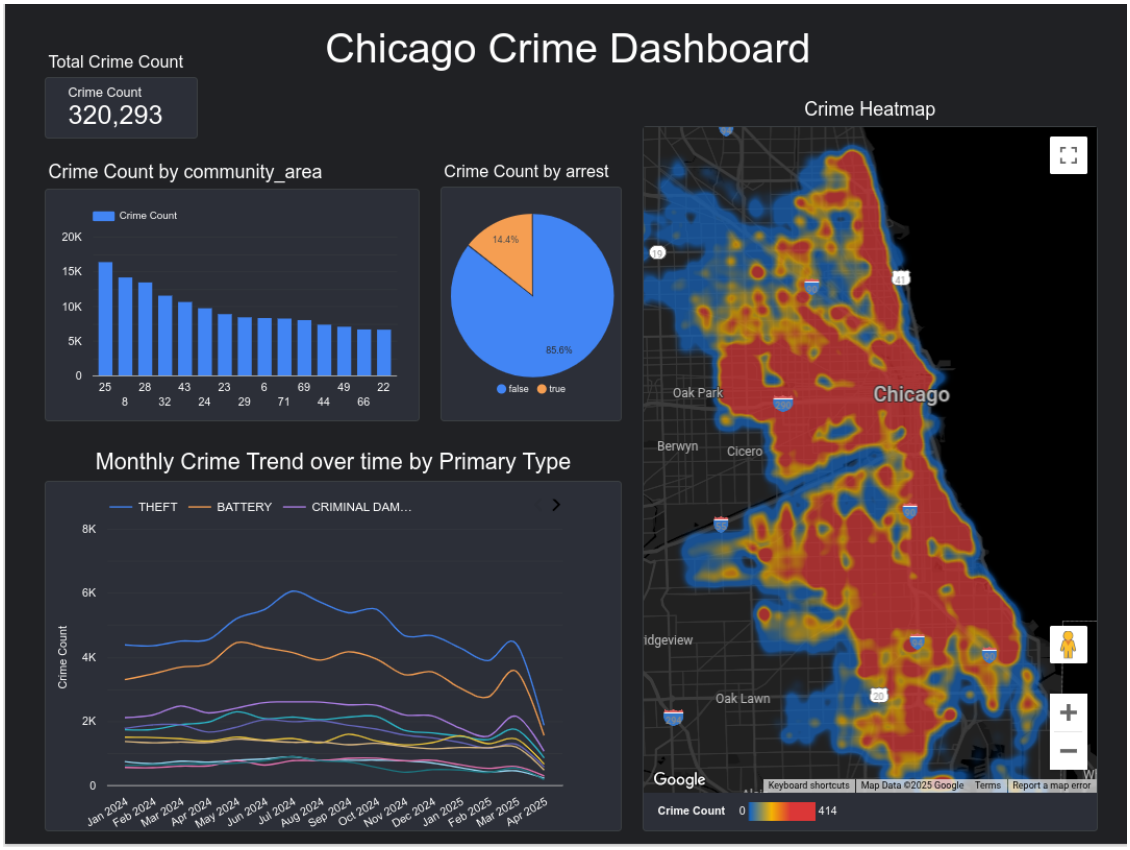


Figure 16: Looker Studio Geo Map showing crime intensity by location

Data Source Configuration

Initially, I attempted to expose my local PostgreSQL server to Looker Studio using Ngrok. However, this approach was abandoned for the following reasons:

- Public exposure of local resources is not secure for production
- Requires a continuously running local machine
- Ngrok tunnels on the free tier expire frequently

Instead, I setup the database in Aiven's managed PostgreSQL instance. This cloud-hosted database ensures consistent availability and does not depend on my local infrastructure.

Creating the Connection

- In Looker Studio, I selected the "PostgreSQL" connector
- I entered the Aiven host, port, user, and password
- I chose views such as `crimes_by_month`, `crimes_by_location`, and `crimes_by_type` to power the dashboard

Enabling Geospatial Visualization

Looker Studio requires a single "Location" field for geo maps. Since my data stores latitude and longitude separately, I created a calculated field:

Listing 1: Looker Studio calculated field for geo mapping

```
CONCAT(latitude , "," ,longitude)
```

This field was used as the "Location" dimension in geo maps to visualize hotspots of criminal activity.

11 Monitoring and Logging

Systemd Status Monitoring

Airflow was configured to run as a systemd service, making it easy to check the status at any time:

```
sudo systemctl status airflow
```

If Airflow or the system is restarted, the scheduler and webserver start automatically in the background.

Airflow Logs

Each task logs detailed output visible from the Airflow UI or saved to disk under the `logs/` directory. These logs include:

- Retry attempts
- Record counts
- Timestamps
- Error tracebacks

Audit Logs

Transformation and loading steps generate CSV files for any dropped or failed records:

- `audit_missing_drops_{timestamp}.csv`
- `audit_duplicate_drops_{timestamp}.csv`
- `failed_batch_{n}_{timestamp}.csv`

These files are timestamped and saved locally, allowing me to inspect issues post-run without affecting the main pipeline.

Resource Monitoring

Because the VM is a Google Cloud e2-micro instance, I used `htop` to monitor CPU and memory usage. Since the system only has 1GB of RAM, it frequently crashed when I tried to ssh into it through vscode, so, I created a 4GB of swap space to mitigate those crashes. The pipeline is optimized for low memory usage by processing data in batches, avoiding unnecessary copies, and running on minimal infrastructure.

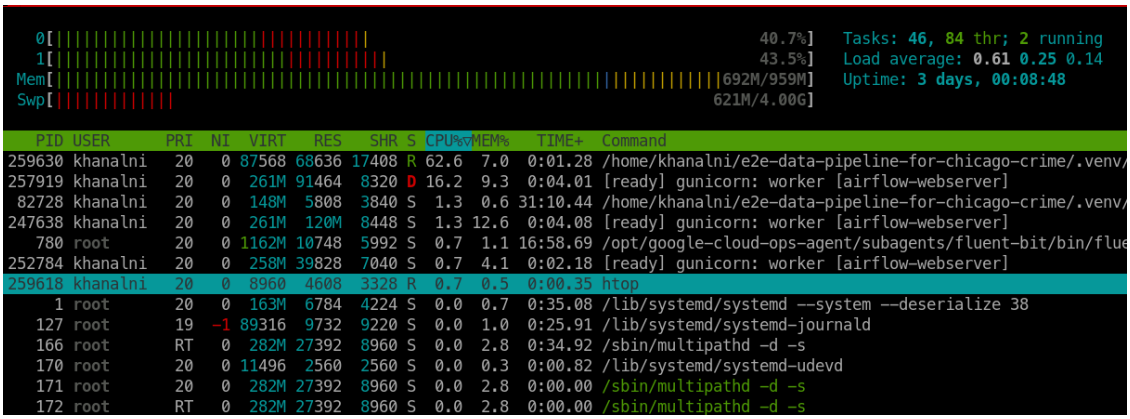


Figure 17: Resource usage during load task, monitored using `htop`

Accessing the Airflow Dashboard Securely

Since the Airflow webserver was running on port 8080 inside a remote VM (Google Cloud), I used SSH port forwarding to securely access the dashboard from my local browser without exposing it to the public internet.

The command I used was:

`ssh -L 8080:localhost:8080 khanalni@34.56.242.64`

This forwards the VM's port `8080` to my local machine's port `8080`, allowing me to view the dashboard by navigating to `http://localhost:8080` in my browser.

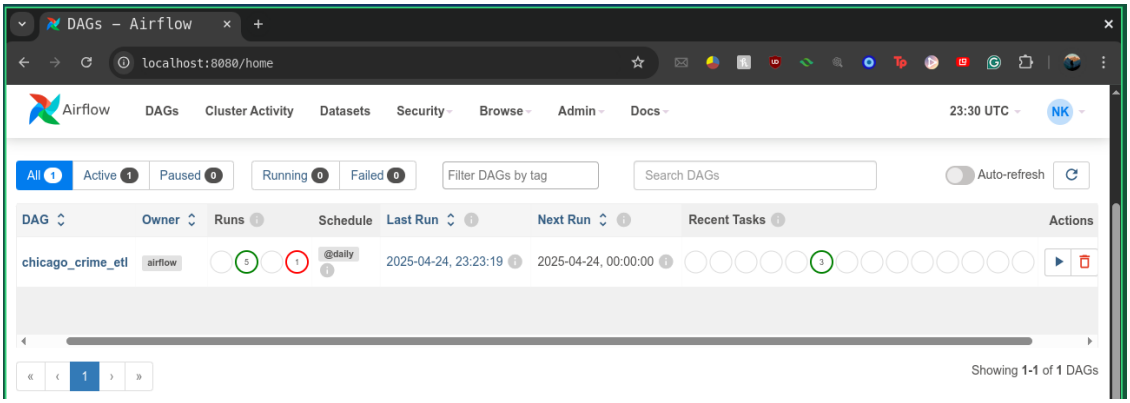


Figure 18: Airflow dashboard running remotely, accessed via SSH port forwarding

Using SSH port forwarding avoided the need for opening firewall rules or exposing the webserver publicly, which is particularly important for VMs without HTTPS or basic authentication.

12 Security and Secrets Management

In this project, several sensitive credentials are used, including:

- PostgreSQL database credentials for the Aiven instance
- API tokens for the Socrata open data platform
- Airflow configurations that may store DAG execution metadata

To avoid hardcoding secrets in source files, I followed best practices for managing credentials securely.

Using `.env` Files

All sensitive information was stored in a `.env` file at the root of the project directory. This file is read at runtime using the `python-dotenv` library.

```
# .env (never committed to version control)
DB_HOST=db.aiven.io
DB_USER=myuser
DB_PASS=secure_password_here
DB_NAME=crime_data
APP_TOKEN=my_socrata_token
```

In Python, these values were loaded as:

```
from dotenv import load_dotenv
import os

load_dotenv()

db_config = {
    "host": os.getenv("DB_HOST"),
    "port": os.getenv("DB_PORT"),
    "dbname": os.getenv("DB_NAME"),
    "user": os.getenv("DB_USER"),
    "password": os.getenv("DB_PASS")
}

app_token = os.getenv("APP_TOKEN")
```

This ensures that secrets are never exposed in the codebase, nor are they accidentally committed to GitHub.

Version Control

The `.env` file and other sensitive scripts were added to `.gitignore` to prevent accidental inclusion in the repository.

```
# .gitignore
.env
*.log
*.csv
logs/
```

Cloud Credentials

For the Aiven PostgreSQL database:

- A dedicated user with limited privileges was created
- SSL was enabled by default
- The connection string was never exposed publicly

Airflow Configuration

For local development and production automation:

- Airflow connections and variables were avoided in plaintext

These practices helped ensure that the pipeline could run securely without exposing critical credentials during deployment, development, or daily automation.

13 Conclusion and Future Work

This project was able to deliver a fully functional and automated ETL pipeline for Chicago crime data processing. The pipeline was designed for reliability, scalability, and ease of maintenance when running on low infrastructure. It includes real-time API extraction, intensive transformation with spatial enrichment, batch-optimized PostgreSQL loading, and visualization with Looker Studio — all orchestrated with Airflow on a micro GCP VM.

Key Accomplishments

- Built a modular ETL framework using Python, with well-separated concerns (extract, transform, load)
- Implemented robust exception handling and audit logging to ensure traceability
- Enabled geospatial enrichment of missing fields via shapefile-based joins

- Migrated from local development to a fully cloud-hosted pipeline with no dependency on local resources
- Used Airflow for orchestration, systemd for automation, and Looker Studio for visualization

Lessons Learned

- Development tools like `ngrok` are useful for prototyping but unsuited for production pipelines
- Memory-efficient techniques such as chunked CSV processing and in-place DataFrame transformations are essential on constrained VMs
- Investing in early error handling and audit logging dramatically reduces debugging overhead
- Choosing managed services like Aiven allows for a highly available system with minimal DevOps effort

Future Enhancements

- **Dockerize the pipeline**: Containerize the ETL pipeline and run it with Docker Compose or Kubernetes for portability and environment isolation
- **Slack/Email alerts**: Configure Airflow to send failure notifications to Slack or email for real-time monitoring
- **Add support for other cities**: Generalize the pipeline to support similar datasets from other U.S. cities

The result is a cloud-native, resource-efficient data pipeline that can grow to support more complex workflows and richer analytics in the future.

References

[1] Aiven. Aiven for postgresql: Managed postgresql on your favorite cloud. `https://aiven.io/postgresql`, 2024. Accessed: 2024-04-20.

[2] Apache Software Foundation. *Apache Airflow Documentation*. Apache Software Foundation, 2024. `https://airflow.apache.org/docs/`.

[3] City of Chicago. Boundaries - community areas (current). `https://data.cityofchicago.org/Facilities-Geographic-Boundaries/Boundaries-Community-Areas-current-/cauq-8yn6`, 2024. Accessed: 2024-04-06.

[4] City of Chicago. Crimes - 2001 to present. https://data.cityofchicago.org/Public-Safety/Crimes-2001-to-Present/ijzp-q8t2, 2024. Accessed: 2024-04-06.

[5] Federico Di Gregorio and Contributors. *psycopg: PostgreSQL database adapter for Python*. The Psycopg Project, 2024. https://www.psycopg.org/.

[6] Google LLC. *Looker Studio: Google's data visualization and reporting tool*. Google, 2024. https://lookerstudio.google.com/.

[7] Charles R. Harris, K. Jarrod Millman, Stéfan J. van der Walt, Ralf Gommers, et al. *NumPy: Fundamental package for scientific computing with Python*. NumPy Developers, 2024. https://numpy.org/.

[8] Kelsey Jordahl and Contributors. *GeoPandas: Python tools for geographic data*. GeoPandas Developers, 2024. https://geopandas.org/.

[9] Wes McKinney and The Pandas Development Team. *pandas: Powerful Python data analysis toolkit*. Pandas Community, 2024. https://pandas.pydata.org/.

[10] Ngrok. Ngrok: Secure introspectable tunnels to localhost. https://ngrok.com, 2024. Accessed: 2024-04-20.

[11] Socrata and Contributors. *Sodapy: A Python client for the Socrata Open Data API*. Socrata, 2024. https://github.com/socrata/sodapy.

[12] The PostgreSQL Global Development Group. *PostgreSQL: The world's most advanced open source relational database*. PostgreSQL Global Development Group, 2024. https://www.postgresql.org/docs/.

[13] Michael Waskom and the Seaborn Development Team. *Seaborn: Statistical Data Visualization*. Seaborn Developers, 2024. https://seaborn.pydata.org/.