Rain and City Infrastructure: Analyzing NYC 311 Service Requests and Weather Data

Group 4:

Adrian Garcia – adrian.garcia1@baruchmail.cuny.edu

Neha Machamada – neha.machamada@baruchmail.cuny.edu

Adnan Quayyum – adnan.quayyum@baruchmail.cuny.edu

Roland Thomas – roland.thomas@baruchmail.cuny.edu

Zicklin School of Business, Baruch College, City University of New York

CIS4400: Data Warehousing for Analytics

Section: CMWA [44276]

Professor Richard Holowczak

Abstract

This project analyzes water-related 311 complaints in New York City, such as sewer issues, water system problems, water conservation concerns, and storm/flooding complaints, in conjunction with daily rainfall and snowfall data for each of the five boroughs. The goal is to identify patterns and correlations between precipitation events and strain on city infrastructure, and to understand how service responses vary geographically.

We will integrate the 311 Service Requests dataset from NYC Open Data with borough-level weather datasets from Open-Meteo API, transform them into separate data marts, and then combine them into a centralized data warehouse. Analyses will focus on identifying communities that are at higher risk for water-related issues during heavy rain or snow, and evaluating response times by borough and type of complaint. These insights may support data-driven decisions about resource allocation and infrastructure planning to reduce vulnerabilities during severe weather events.

The Problem Statement

NYC experiences extreme weather throughout the year, from heat waves in summer to snowstorms in winter. Heavy rain and snow can overwhelm city infrastructure, leading to flooding, sewer backups, and service interruptions. A major concern is how water-related issues affect neighborhoods differently across boroughs and whether the city responds consistently across geographic areas. By focusing on four categories of water-related 311 complaints: Sewer, Water System, Water Conservation, and Storm/Flooding, and linking them with borough-level daily weather data, we can examine how precipitation correlates with the volume and severity of complaints, as well as with response times. Key performance indicators (KPIs) will measure complaint volumes, rainfall and snowfall levels, geographic clustering of incidents, and average service response times during normal vs. extreme weather days. This analysis will allow us to identify boroughs and communities most affected by water-related issues and suggest where the city may need to improve its preparedness and responsiveness.

Part I: Data Sources

The Main Dataset

NYC Open Data – 311 Service Requests (2010 to Present)

- Rows: A total of about 41 million rows, where each row is a 311 service request.
- Columns: A total of 41 columns, including a unique identifier for each service request.
- Columns of Interest or KPIs: Complaint Type, Incident ZIP, Borough, Longitude, Latitude, Descriptor, Created Date, Closed Date, Resolution Description.
- Filter: Complaint Types = Sewer, Water System, Water Conservation, Storm/Street Flooding

NYC Open Data 311 Service Requests Columns (Excluding Unique ID):

Created Date	Closed Date	Agency	Agency Name
Complaint Type	Descriptor	Location Type	Incident Zip
Incident Address	Street Name	Cross Street 1	Cross Street 2
Intersection Street 1	Intersection Street 2	Address Type	City
Landmark	Facility Type	Status	Due Date
Resolution Description	Resolution Action Updated Date	Community Board	BBL
Borough	X Coordinate (State Plane)	Y Coordinate (State Plane)	Open Data Channel Type
Park Facility Name	Park Borough	Vehicle Type	Taxi Company Borough
Taxi Pick Up Location	Bridge Highway Name	Bridge Highway Direction	Road Ramp
Bridge Highway Segment	Latitude	Longitude	Location

Unique Key	Created Date	Closed Date	Agency	Agency Name	Complaint Type	Descriptor	Location Type	Incident Zip	Incident Address	Street Name
65905898	08/20/2025 03:22:00 PM	08/20/2025 07:00:00 PM	DEP	Department of Environmental Protect	Water System	Excessive Water In Basement (WEF		11411	216-26 121 AVENUE	121 AVENUE
65349315	06/23/2025 10:38:00 PM	06/24/2025 12:50:00 AM	DEP	Department of Environmental Protect	Water System	Excessive Water In Basement (WEF		11221	436 EVERGREEN AVENUE	EVERGREEN AVENUE
65151592	06/03/2025 12:32:00 PM	06/03/2025 01:50:00 PM	DEP	Department of Environmental Protect	Sewer	Sewer Backup (Use Comments) (SA		11215	417 CARROLL STREET	CARROLL STREET
65012511	05/20/2025 01:26:00 PM	05/20/2025 01:40:00 PM	DEP	Department of Environmental Protect	Sewer	Sewer Backup (Use Comments) (SA		10004	44 WATER STREET	WATER STREET
64854981	05/05/2025 11:56:00 AM	05/05/2025 02:25:00 PM	DEP	Department of Environmental Protect	Sewer	Sewer Odor (SA2)		11201	59 PEARL STREET	PEARL STREET
63743414	01/12/2025 04:17:00 PM	01/12/2025 07:50:00 PM	DEP	Department of Environmental Protect	Sewer	Sewer Backup (Use Comments) (SA		11360	215-24 23 AVENUE	23 AVENUE
63702064	01/09/2025 07:35:00 PM	01/09/2025 09:45:00 PM	DEP	Department of Environmental Protect	Sewer	Sewer Backup (Use Comments) (SA		11360	215-20 23 ROAD	23 ROAD
63636640	01/05/2025 08:50:00 AM	01/05/2025 10:40:00 AM	DEP	Department of Environmental Protect	Sewer	Sewer Backup (Use Comments) (SA		10461	1728 EASTCHESTER ROAD	EASTCHESTER ROAD
62924682	10/30/2024 11:59:00 AM	10/30/2024 02:05:00 PM	DEP	Department of Environmental Protect	Sewer	Sewer Odor (SA2)		10302	241 CRYSTAL AVENUE	CRYSTAL AVENUE
62805459	10/18/2024 05:17:00 PM	10/18/2024 08:05:00 PM	DEP	Department of Environmental Protect	Sewer	Sewer Odor (SA2)		10004	1 STATE STREET	STATE STREET
62211943	08/23/2024 04:50:00 PM	08/23/2024 06:35:00 PM	DEP	Department of Environmental Protect	Sewer	Sewer Backup (Use Comments) (SA		10302	133 BRYSON AVENUE	BRYSON AVENUE
62068632	08/09/2024 01:05:00 PM	08/09/2024 02:15:00 PM	DEP	Department of Environmental Protect	Sewer	Sewer Backup (Use Comments) (SA		10004	125 BROAD STREET	BROAD STREET
62057870	08/08/2024 09:44:00 AM	08/08/2024 12:00:00 PM	DEP	Department of Environmental Protect	Sewer	Sewer Backup (Use Comments) (SA		10314	271 BIDWELL AVENUE	BIDWELL AVENUE
61376188	06/05/2024 11:53:00 AM	06/05/2024 01:20:00 PM	DEP	Department of Environmental Protect	Sewer	Sewer Backup (Use Comments) (SA		11201	312 WATER STREET	WATER STREET
61270153	05/25/2024 04:08:00 PM	05/25/2024 05:55:00 PM	DEP	Department of Environmental Protect	Water System	Excessive Water In Basement (WEF		11429	112-29 219 STREET	219 STREET
61073022	05/06/2024 11:17:00 PM	05/07/2024 09:25:00 AM	DEP	Department of Environmental Protect	Water System	LOW WATER PRESSURE - WLWP		11219	1249 42 STREET	42 STREET
61013228	04/29/2024 04:08:00 PM	04/29/2024 05:45:00 PM	DEP	Department of Environmental Protect	Sewer	Sewer Odor (SA2)		11201	244 WATER STREET	WATER STREET
60683140	03/25/2024 03:38:00 PM	03/25/2024 06:25:00 PM	DEP	Department of Environmental Protect	Sewer	Manhole Overflow (Use Comments		11360	215-44 23 ROAD	23 ROAD
59647996	12/05/2023 02:08:00 PM	12/05/2023 05:00:00 PM	DEP	Department of Environmental Protect	Sewer	Sewer Backup (Use Comments) (SA		10314	239 BIDWELL AVENUE	BIDWELL AVENUE
59377038	11/09/2023 09:08:00 AM	11/09/2023 10:20:00 AM	DEP	Department of Environmental Protect	Sewer	Sewer Backup (Use Comments) (SA		10314	245 WILLARD AVENUE	WILLARD AVENUE

Weather Data Source:

Open-Meteo API

Rows: A total of about 87,700 rows, with each row representing an hourly weather observation over a 10 year period.

Columns: 10 columns, including time, temperature, and precipitation measurements.

Columns of Interest or KPIs: Time, Precipitation, Rainfall, Windspeed.

5 Data Sets, For Each Borough - Data from October 1, 2015 to October 1, 2025

time	temperature_2m (°C)
precipitation (mm)	rain (mm)
cloudcover (%)	cloudcover_low (%)
cloudcover_mid (%)	cloudcover_high (%)
windspeed_10m (km/h)	winddirection_10m (°)

Sample - Queens Data

time	temperature_2m (Â°C)	precipitation (mm)	rain (mm)	cloud_cover (%)	cloud_cover_low (%)	cloud_cover_mid (%)	cloud_cover_high (%)	wind_speed_10m (km/h)	wind_direction_10m (Â°)
2015-10-03T00:00	0.4	0	0	58	5	9	53	4.3	175
2015-10-03T01:00	0.3	0	0	68	19	20	60	4.7	171
2015-10-03T02:00	0.9	0.3	0.3	88	25	37	83	3.4	162
2015-10-03T03:00	5.2	0.2	0.1	94	23	48	90	4	175
2015-10-03T04:00	6.6	0.3	0.2	96	18	82	93	4.8	228
2015-10-03T05:00	7.4	0.2	0.2	98	74	93	89	6.8	238
2015-10-03T06:00	8.2	0.1	0.1	99	76	93	60	10.3	245
2015-10-03T07:00	9.5	0.1	0.1	95	72	92	37	10.8	249
2015-10-03T08:00	8.4	0.1	0.1	94	69	81	35	9.9	251
2015-10-03T09:00	8.6	0.7	0.7	77	65	51	27	10.2	247
2015-10-03T10:00	8.9	0.2	0.2	60	48	24	20	7.4	247
2015-10-03T11:00	7.6	0.1	0.1	48	27	2	27	6.7	234
2015-10-03T12:00	7.3	0.1	0	51	22	15	41	5.8	240
2015-10-03T13:00	6.6	0.1	0.1	67	33	39	46	4.3	222
2015-10-03T14:00	5.7	0.1	0.1	66	37	60	1	3.4	212
2015-10-03T15:00	5.1	0.2	0.2	67	51	49	0	4	185
2015-10-03T16:00	4.8	0.4	0.4	59	50	36	0	5	180
2015-10-03T17:00	4.6	0	0	55	51	28	0	5.4	172
2015-10-03T18:00	3.7	0	0	69	65	29	1	5.1	172
2015-10-03T19:00	3	0	0	66	65	25	0	5.2	168
2015-10-03T20:00	2.8	0	0	52	49	16	0	5.4	160
2015-10-03T21:00	2.3	0	0	40	39	8	0	5.5	157
2015-10-03T22:00	1.9	0	0	29	22	13	0	4.4	171
2015-10-03T23:00	1.4	0	0	22	21	1	0	4.9	163

Potential Data Source 2:

NYC Open Data – Street Flooding (2010 to Present)

- Rows: A total of about 43 thousand rows where each row is a report.
- Columns: A total of 39 columns, including a unique identifier.
- Columns of Interest or KPIs: Complaint Type, Incident ZIP, Borough, Longitude, Latitude, Descriptor, Created Date, Closed Date, Resolution Description.

NYC Open Data Street Flooding Dataset Columns:

Unique Key	Created Date	Closed Date	Agency
Agency Name	Complaint Type	Descriptor	Location Type
Incident Zip	Incident Address	Street Name	Cross Street 1
Cross Street 2	Intersection Street 1	Intersection Street 2	Address Type
City	Landmark	Facility Type	Status
Due Date	Resolution Description	Resolution Action Updated Date	Community Board
Borough	X Coordinate (State Plane)	Y Coordinate (State Plane)	Park Facility Name
Park Borough	Vehicle Type	Taxi Company Borough	Taxi Pick Up Location
Bridge Highway Name	Bridge Highway Direction	Road Ramp	Bridge Highway Segment
Latitude	Longitude	Location	–

Other:

Potential APIs:

- Socrata Open Data API (SODA) – NYC Open Data API
- National Centers for Environmental Information API (NOAA) – Extreme Weather Events

Part II: Requirements Gathering & KPIs

Data Sources –

1. **311 Service Complaints (NYC Open Data)**
   Source:https://data.cityofnewyork.us/Social-Services/311-Service-Requests-from-2010-to-Present
   Update Frequency: Daily
   Time Period Being Used: Oct 2015 - Oct 2025
   Columns Being Used: Unique Key, Created Date, Closed Date, Agency, Agency Name, Complaint Type, Descriptor, Location Type, Incident Zip, City, Status, Resolution Action Updated, Borough, Latitude, Longitude

2. **Hourly Weather Data by Borough**
   Source: Open Meteo API
   Update Frequency: Daily
   Time Period Being Used: Oct 2015 - Oct 2025
   Columns Being Used: Time, Temperature, Precipitation, Rain, Wind Speed

Key Performance Indicators (KPIs) –

311 Water Complaints KPIs

1. Daily total water-related complaints by borough.
2. Complaint count per 10,000 residents by borough/ZIP.
3. Average response time (Created → Closed) for water-related complaints.
4. Percent of complaints resolved within SLA (e.g., 7 days).
5. Severity breakdown by descriptor (flooding, sewer backup, leak, etc.).

Weather KPIs

1. Daily precipitation (rainfall + snowfall) per borough.
2. Number of extreme weather days (e.g., >2 inches rain, >6 inches snow).
3. Average precipitation per month by borough.

Integrated KPIs (311 + Weather) –

1. Correlation between daily precipitation and complaint counts by borough.
2. Average response times on extreme precipitation days vs. normal days.
3. Borough ranking of water complaints during heavy rain events.
4. Hotspot analysis: ZIP codes with highest complaint counts per inch of rain.
5. Service gap KPI: Difference in average resolution time during storm events between boroughs.

Part III: Dimensional Modeling

Kimball's Four Step Process

Step 1: Selecting the Business Processes – There are two main processes that are going to be modeled – 311 Complaints, and Weather Monitoring.

The 311 Complaints Process – This process will be modeled to capture incidents reported to NYC's 311 line, specifically complaints related to the water system, sewer system, and storm or flooding complaints. Each complaint includes information such as the complaint type, the location of the reported incident, the agency responsible for resolving the issue. This process provides insight into how city infrastructure is affected by water-related issues.

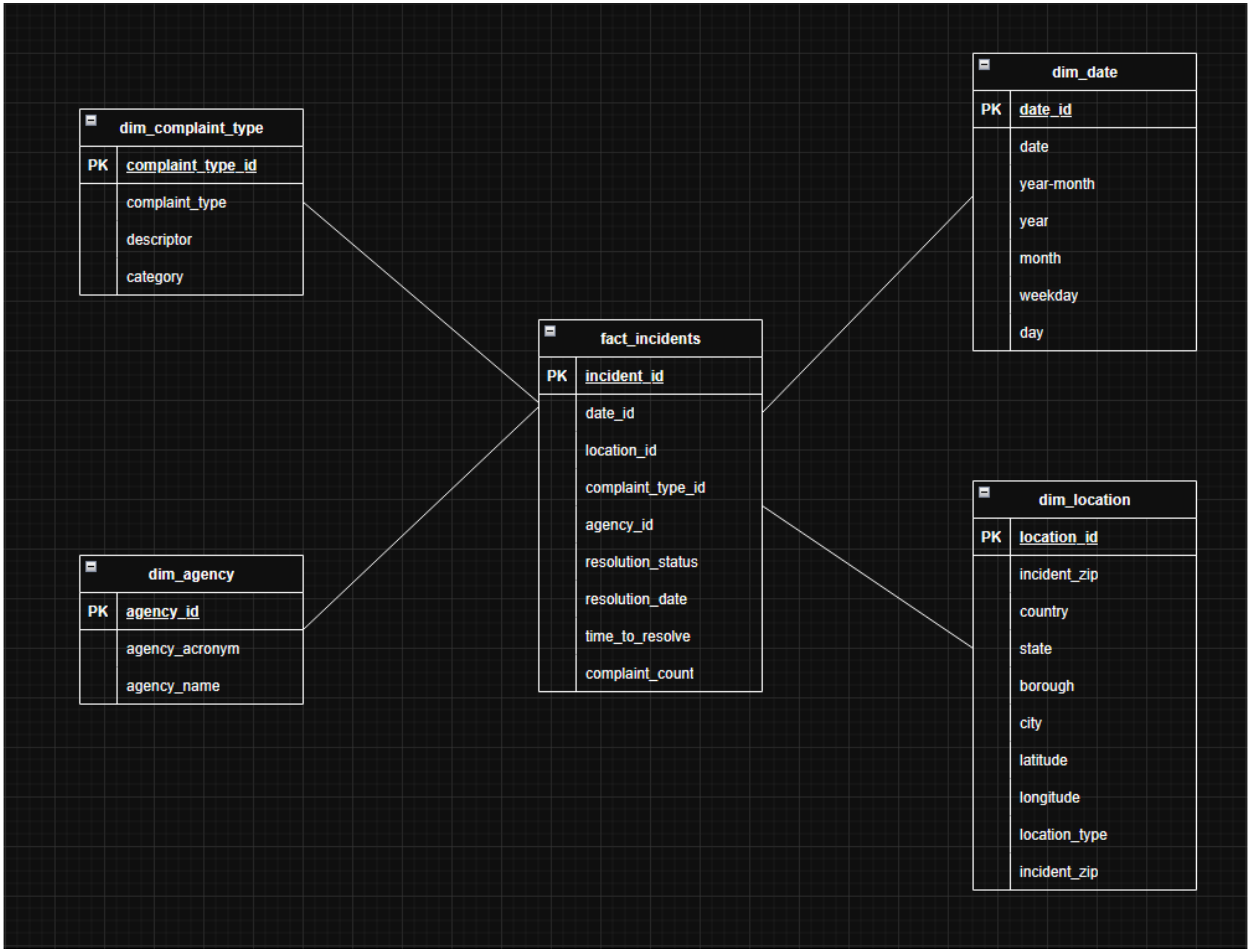
The Weather Monitoring Process – This process tracks hourly and daily weather data for each borough within the same time period as the 311 complaints. This process contains measures such as precipitation, rainfall, temperature, and wind speed.

Step 2: Declaring the Grain
For the 311 complaints, the grain is defined as one row for each individual complaint made that can be uniquely identified with the "Unique Key" provided by NYC Open Data. This allows measures such as resolution time and counts of incidents to be calculated accurately for each complaint. For the weather data, the grain is one row per hour for each borough, capturing weather conditions such as precipitation, rainfall, and windspeed.

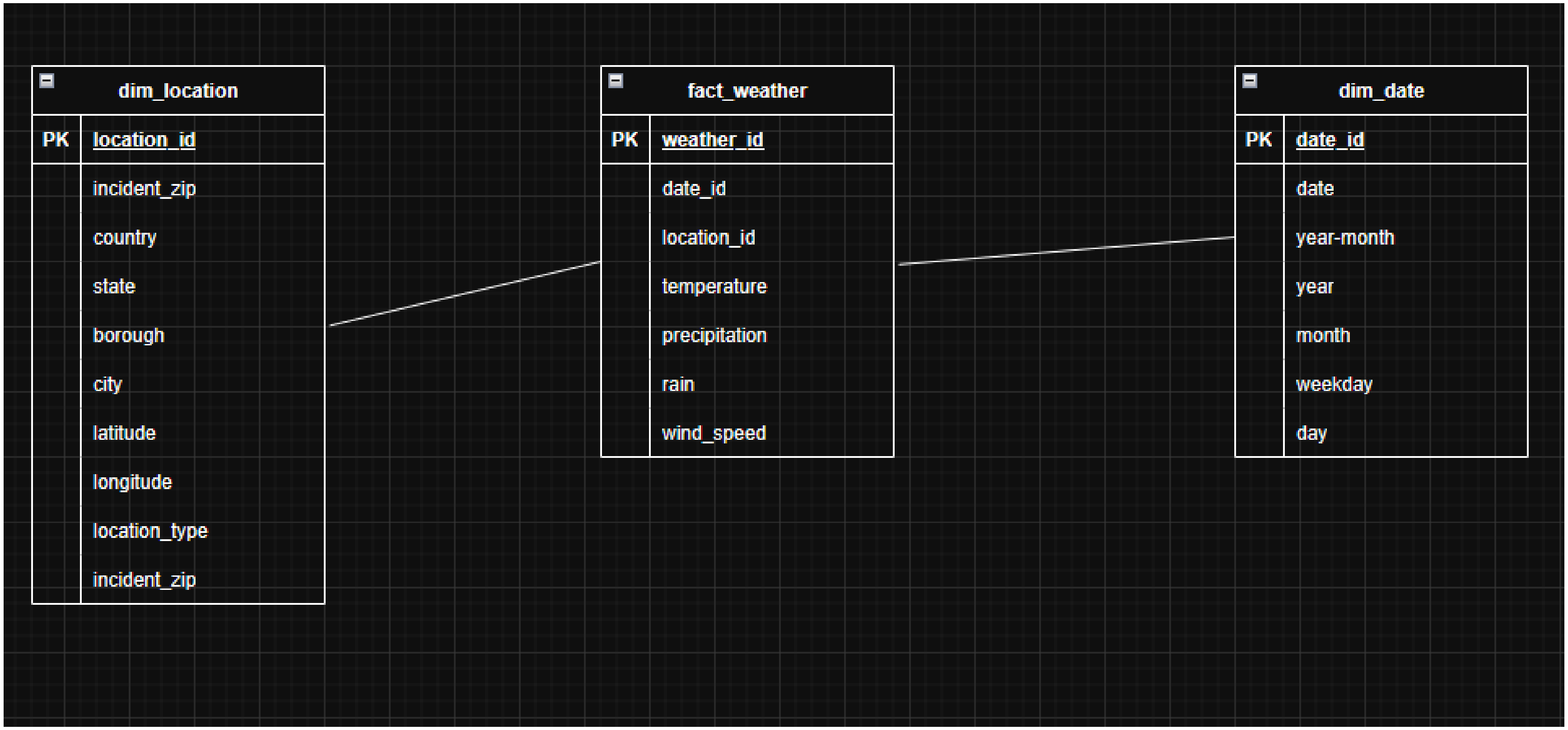
Step 3: Identifying the Dimensions
There will be two dimensions shared between both fact tables – dim_date, a dimension containing time related attributes, and dim_location, which includes location attributes of the both the reported incidents and the weather.
There will also be a dim_complaint_type, which includes the type of complaint made and description of the complaint. A dim_agency dimension will also be included to capture the agencies responsible for resolving the complaint.

Step 4: Identifying the Facts
Fact_Incidents: Measures include incident_count (usually 1 per complaint) and Resolution_Time, calculated from Created Date to Resolution Action Updated Date. These measures allow for evaluation of complaint volume, response efficiency, and trends in water-related issues.
Fact_Weather: Measures include Temperature, Precipitation, Rain, Wind_Speed, Wind_Direction, and Cloud_Cover, enabling analysis of weather patterns and correlation with 311 complaints.

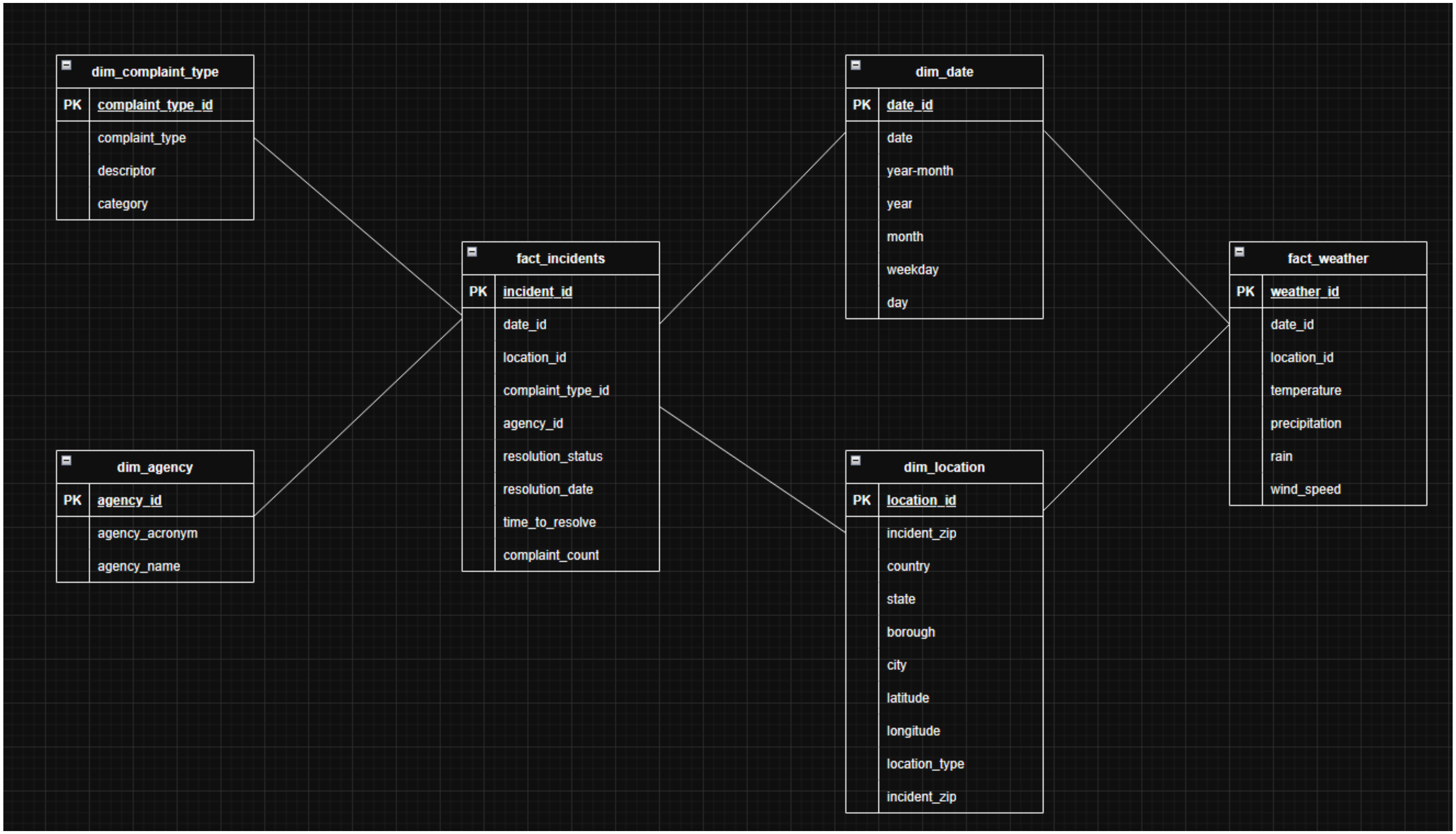
NYC Water-Related Complaints Star Schema



Weather Patterns Star Schema



NYC Water-Related Complaints & Weather Patterns Star Schema



Part IV: Technical Architecture

Target Data Warehouse: Google BigQuery
Hosting Environment: Google Cloud
Automation & Orchestration: Airflow

ETL in Python (Libraries & Frameworks):

- Extract: Urllib, PyArrow, Parquet, SODA, openmeteo-requests
- Transform: Pandas, Polars, DuckDB, Great Expectations
- Load: Google-Cloud-BigQuery

Data from the NYC 311 dataset will be extracted with Python by querying the Socrata Open Data API using URL queries with urllib to encode queries directly in the URL. The data is loaded in chunks and in parallel using Polars to handle large in-memory datasets efficiently. Each chunk is then converted into an Apache Arrow table using PyArrow for optimized constant reading/writing operations, and it will be written to a Parquet file, which provides efficient compression for storage. Weather data will be pulled using openmeteo-requests, processed with Pandas, and then saved as CSVs.

Data will be processed, profiled, and transformed using a combination of Polars for large in-memory preprocessing, and Pandas for exploratory data analysis. Visualization libraries such as Matplotlib and Seaborn (which integrate with Pandas) will be used to visualize missingness and cardinality. Great Expectations will be used for data validation and data quality assurance. DuckDB will be used as a staging database file to hold the table data for the final dimension and fact tables from the processed data.

Each DuckDB table will be downloaded as an individual Parquet file for efficient storage, and for easy integration with the target data warehouse. The data will then be loaded into the target data warehouse, Google BigQuery, with Google Cloud as the hosting environment for the data. The Google BigQuery Python API will be used for loading the data into the target data warehouse. Apache Airflow will be used for automation and orchestration of the ETL process once the initial pipeline is created.

Part V: ETL Programming

Our ETL pipeline was implemented in Python and organized into modular steps: (1) extracting raw 311 and weather data from the API's (2) Validating and transforming into clean and consistent schemas, (3) Staging curated output in efficient file formats(Parquet) and a staging engineer (DuckDB) and (4) Loading the final dimension and fact tables into Google BigQuery for analytics and dashboarding. This section documents the major ETL modules and highlights key custom code.

Link to Repo: https://github.com/AGxData/nyc-311-weather-etl
Look from left to right

Extract code for 311 dataset

```python
# Importing libraries needed for extraction
import polars as pl
import pyarrow as pa
import pyarrow.parquet as pq
import urllib.parse
import json
import os
from datetime import datetime
from concurrent.futures import ThreadPoolExecutor, as_completed
from frictionless import Resource, Schema, Field
from logger.etl_logger import ETLLogger
from pathlib import Path


# Setting logging for extraction
extract_logger = ETLLogger("extract").get()
extract_logger.info("Starting 311 data extraction")

# Paths for project root, metadata, and full data
project_root = Path(__file__).resolve().parents[2]
main_parquet = project_root / "data" / "nyc_311_full_preprocessed.parquet
metadata_folder = project_root / "metadata"

# Columns for extraction
columns = [
    "unique_key", "created_date", "closed_date", "agency", "agency_name",
    "complaint_type", "descriptor", "location_type", "incident_zip",
    "city", "status", "resolution_action_updated_date", "borough",
    "latitude", "longitude"
]

# columns needed for overwriting
slowly_changing_dimensions = [
    "created_date",
    "closed_date",
    "resolution_action_updated_date"
]

```

--- This section sets up the ETL process by importing the libraries used for data processing, API calls, and validation. We

```python
# Frictionless schema for validation
schema = Schema(fields = [
    Field(name = "unique_key", type = "string"),
    Field(name = "created_date", type = "datetime"),
    Field(name = "closed_date", type = "datetime"),
    Field(name = "agency", type = "string"),
    Field(name = "agency_name", type = "string"),
    Field(name = "complaint_type", type = "string"),
    Field(name = "descriptor", type = "string"),
    Field(name = "location_type", type = "string"),
    Field(name = "incident_zip", type = "string"),
    Field(name = "city", type = "string"),
    Field(name = "status", type = "string"),
    Field(name = "resolution_action_updated_date", type ="datetime"),
    Field(name = "borough", type = "string"),
    Field(name = "latitude", type = "number"),
    Field(name = "longitude", type = "number"),
])

# Settings for extraction
base_url = r"https://data.cityofnewyork.us/resource/erm2-nwe9.csv"
chunk_size = 100_000
max_workers = 4



# Frictionless schema for validation upon extraction
schema = Schema(fields = [
    Field(name = "unique_key", type = "string"),
    Field(name = "created_date", type = "datetime"),
    Field(name = "closed_date", type = "datetime"),
    Field(name = "agency", type = "string"),
    Field(name = "agency_name", type = "string"),
    Field(name = "complaint_type", type = "string"),
    Field(name = "descriptor", type = "string"),
    Field(name = "location_type", type = "string"),
    Field(name = "incident_zip", type = "string"),
    Field(name = "city", type = "string"),
```

also define a schema that lists the columns we expect from the 311 dataset and their data types. This schema is later used to check that the data we extract matches the expected structure. Defining the schema early helps catch data issues before the data is saved or merged.

```python
    Field(name = "status", type = "string"),
    Field(name = "resolution_action_updated_date", type = "datetime"),
    Field(name = "borough", type = "string"),
    Field(name = "latitude", type = "number"),
    Field(name = "longitude", type = "number"),
])


# Function for downloading by chunks from Socrata API via URL (Faster i/o)
def download_chunk(offset, latest_date):
    soql = f"""
        SELECT {', '.join(columns)}
        WHERE created_date > '{latest_date}'
        LIMIT {chunk_size} OFFSET {offset}
    """
    encoded_query = urllib.parse.quote(soql, safe='')
    url = f"{base_url}?$query={encoded_query}"

    try:
        df_chunk = pl.read_csv(url, columns=columns, dtypes={"incident_zip": pl.Utf8})
        if df_chunk.height == 0:
            return None
        return df_chunk
    except Exception as e:
        extract_logger.error(f"Error at offset {offset}: {e}")
        return None



def extract_311():
    # Determining latest date in metadata extraction files
    metadata_folder.mkdir(parents = True, exist_ok = True)
    metadata_file = metadata_folder / "last_date.json"

    if metadata_file.exists():
        with open(metadata_file) as f:
            report_json = json.load(f)
            latest_date = report_json.get("last_date")
```

--- The 311 dataset is very large, so we do not download it all at once. Instead, we pull the data in smaller chunks using LIMIT and OFFSET. Each chunk is read into memory using Polars, which is faster and more memory-efficient than standard tools for large datasets. Chunking allows us to safely extract millions of rows without running out of memory.

--- This step allows the ETL process to only pull new 311 records. The pipeline checks a metadata file that stores the most recent created_date

```python
            extract_logger.info(f"Using latest_date from metadata: {latest_date}")
    else:
        latest_date = "2025-09-25T01:44:42"  # default start date
        extract_logger.info(f"No metadata found, using default start date: {latest_date}")

    offset = 0
    all_chunks = []
    finished = False

    while not finished:
        offsets = [offset + i * chunk_size for i in range(max_workers)]
        results = []

        with ThreadPoolExecutor(max_workers=max_workers) as executor:
            futures = {executor.submit(download_chunk, o, latest_date): o for o in offsets}
            for future in as_completed(futures):
                df_chunk = future.result()
                if df_chunk is not None and df_chunk.height > 0:
                    results.append(df_chunk)

        if not results:  # Stopping if all chunks are empty
            finished = True
        else:
            all_chunks.extend(results)
            offset += max_workers * chunk_size

    if not all_chunks:
        extract_logger.info("No new 311 data to extract.")
        return None

    # Concatenating data
    new_data = pl.concat(all_chunks, rechunk=True)

    # Schema Validation
    try:
        resource = Resource(data = new_data.to_dicts(), schema = schema)
        validation_report = resource.validate()
        if validation_report.valid:
```

--- To speed up the extraction process, multiple chunks are downloaded at the same time using a thread pool. This means the program can request data from the API in parallel instead of waiting for each request to finish one at a time. This is because parallel downloads significantly reduce the total runtime of the ETL process.

processed in the last run. If the file exists, extraction starts from that date instead of reloading all historical data.

```python
            extract_logger.info("Extracted data matches schema.")
        else:
            extract_logger.warning("Schema validation failed. Check extracted data.")
    except Exception as e:
        extract_logger.error(f"Schema validation failed: {e}")

    # Merging with main dataset with incremental updatation
    if main_parquet.exists():
        df_main = pl.read_parquet(main_parquet)

        # Joining on SCD columns
        df_update = new_data.select(["unique_key"] + slowly_changing_dimensions)
        df_main = df_main.join(df_update, on="unique_key", how="left")

        # Overwriting SCD columns if new value exists
        for col in slowly_changing_dimensions:
            right_col = f"{col}_right"
            if right_col in df_main.columns:
                df_main = df_main.with_columns(
                    pl.when(pl.col(right_col).is_not_null())
                    .then(pl.col(right_col))
                    .otherwise(pl.col(col))
                    .alias(col)
                ).drop(right_col)

        # Adding new rows that do not exist
        new_rows = new_data.join(df_main, on="unique_key", how="anti")
        if new_rows.height > 0:
            df_main = pl.concat([df_main, new_rows], rechunk=True)

        combined = df_main
    else:
        combined = new_data

    # Updating metadata files
    last_date = new_data.select(pl.col("created_date").max()).item()
    with open(metadata_file, "w") as f:
        json.dump({"last_date": last_date}, f)
```

--- This part of the code merges newly extracted data with the existing dataset. If a complaint already exists, certain fields such as dates are updated when newer values are available. Complaints that do not already exist are added as new rows. This keeps the dataset accurate and up to date without creating duplicates.

```python

    # Saving updated dataset
    combined.write_parquet(main_parquet)
    extract_logger.info(f"Extraction completed. Total records: {combined.height}")

    return combined


# Entry point for the extraction function
if __name__ == "__main__":
    extract_311()
```

--- The final cleaned dataset is saved as a Parquet file, which is efficient for storage and analytics. The metadata file is then updated with the latest processed date so future runs know where to resume. This makes the ETL process repeatable and efficient over time.

Extract code for Weather dataset

```python
import polars as pl
from datetime import datetime, timedelta
from concurrent.futures import ThreadPoolExecutor, as_completed
import requests
import json
from pathlib import Path
from logger.etl_logger import ETLLogger

# Logger settings
extract_logger = ETLLogger("extract_weather").get()
extract_logger.info("Starting weather data extraction")

# Settings for file path locations
project_root = Path(__file__).resolve().parents[2]
metadata_folder = project_root / "metadata"
metadata_folder.mkdir(parents=True, exist_ok=True)
metadata_file = metadata_folder / "weather_last_date.json"

if metadata_file.exists():
    with open(metadata_file) as f:
        metadata = json.load(f)
    last_date = datetime.fromisoformat(metadata.get("last_date"))
    extract_logger.info(f"Using last_date from metadata: {last_date.date()}")
else:
    last_date = datetime(2010, 1, 1)
    extract_logger.info(f"No metadata found. Starting from default date: {last_date.date()}")



# Settings for pulling data
chunk_days = 30
max_workers = 4
end_date = datetime(2025, 9, 25)



# Open-Meteo variables of interest
variables = [
    "temperature_2m_max",
    "temperature_2m_min",
    "precipitation_sum",
    "precipitation_hours",
    "rain_sum",
    "showers_sum",
    "snowfall_sum",
    "windspeed_10m_max",
    "windgusts_10m_max"
]

# Open-Meteo base URL
base_url = "https://archive-api.open-meteo.com/v1/archive"



# Centroid lat/lon coordinates for each borough
borough_coords = {
    "Manhattan": (40.7831, -73.9712),
    "Brooklyn": (40.6782, -73.9442),
    "Queens": (40.7282, -73.7949),
    "Bronx": (40.8448, -73.8648),
    "Staten Island": (40.5795, -74.1502)
}



# Function to help pull data in chunks based on date ranges
def daterange_chunks(start, end, days_per_chunk = 30):
    current_start = start
    while current_start <= end:
        current_end = min(current_start + timedelta(days = days_per_chunk-1), end)
        yield current_start, current_end
        current_start = current_end + timedelta(days = 1)



# Function to help pull lat/lon borough centroid data
def fetch_weather(borough, lat, lon, start, end):
    params = {
        "latitude": lat,
        "longitude": lon,
        "start_date": start.strftime("%Y-%m-%d"),
        "end_date": end.strftime("%Y-%m-%d"),
        "daily": ",".join(variables),
        "timezone": "America/New_York"
    }
    try:
        resp = requests.get(base_url, params=params, timeout=60)
        resp.raise_for_status()
        data = resp.json()
        if "daily" not in data:
            extract_logger.warning(f"No daily data for {borough} {start.date()} to {end.date()}")
            return None
        df = pl.DataFrame(data["daily"])
        df = df.with_columns([
            pl.lit(borough).alias("borough"),
            pl.lit(lat).alias("latitude"),
            pl.lit(lon).alias("longitude")
        ])
        return df
    except Exception as e:
        extract_logger.error(f"Error fetching {borough} {start.date()} to {end.date()}: {e}")
        return None



# Main function for weather extraction
def extract_weather():
    current_max_date = last_date
    all_chunks = []

    for chunk_start, chunk_end in daterange_chunks(last_date + timedelta(days = 1), end_date, chunk_days):
        extract_logger.info(f"Fetching data from {chunk_start.date()} to {chunk_end.date()}")
        dfs = []

        with ThreadPoolExecutor(max_workers=max_workers) as executor:
            futures = {executor.submit(fetch_weather, b, *borough_coords[b], chunk_start, chunk_end): b
                    for b in borough_coords}
            for future in as_completed(futures):
                result = future.result()
                if result is not None:
                    dfs.append(result)

        if dfs:
            chunk_df = pl.concat(dfs, rechunk=True)
            all_chunks.append(chunk_df)

            max_chunk_date = max(chunk_df["time"].to_list())
            if isinstance(max_chunk_date, str):
                max_chunk_date = datetime.fromisoformat(max_chunk_date)
            if max_chunk_date > current_max_date:
                current_max_date = max_chunk_date

    if not all_chunks:
        extract_logger.info("No new weather data to extract.")
        return None

    combined_df = pl.concat(all_chunks, rechunk=True)

    # Updating metadata files
    with open(metadata_file, "w") as f:
        json.dump({"last_date": current_max_date.isoformat()}, f)
    extract_logger.info(f"Weather extraction complete. last_date updated to {current_max_date.date()}")

    return combined_df
```

	`` # Entry point for weather extraction function if __name__ == "__main__": weather_df = extract_weather() ```

Transform code for 311 dataset

```python
# Functions needed for transformation
import polars as pl
import json
from pathlib import Path
from logger.etl_logger import ETLLogger
from rapidfuzz import process, fuzz


# Settings for logging transformation
transform_logger = ETLLogger("transform_311").get()
transform_logger.info("Starting 311 data transformation")

# Settings for file path locations
project_root = Path(__file__).resolve().parents[2]
mapping_dir = project_root / "mappings"
metadata_folder = project_root / "metadata"
metadata_folder.mkdir(parents=True, exist_ok=True)

# Loading mappings
mappings = {}
for file in mapping_dir.glob("*.json"):
    with open(file, "r") as f:
        mappings[file.stem] = json.load(f)



# Function to set data types for each column
def data_type_transformer(df: pl.DataFrame) -> pl.DataFrame:
    df = df.with_columns([
        pl.col("unique_key").cast(pl.Utf8),
        pl.col("created_date").str.strptime(pl.Datetime, fmt = r"%Y-%m-%dT%H:%M:%S", strict = False),
        pl.col("closed_date").str.strptime(pl.Datetime, fmt = r"%Y-%m-%dT%H:%M:%S", strict = False),
        pl.col("agency").cast(pl.Utf8),
        pl.col("agency_name").cast(pl.Utf8),
        pl.col("complaint_type").cast(pl.Utf8),
        pl.col("descriptor").cast(pl.Utf8),
        pl.col("location_type").cast(pl.Utf8),
        pl.col("incident_zip").cast(pl.Utf8),
```

```python
        pl.col("city").cast(pl.Utf8),
        pl.col("status").cast(pl.Utf8),
        pl.col("resolution_action_updated_date").str.strptime(pl.Datetime, fmt = r"%Y-%m-%dT%H:%M:%S", strict = False),
        pl.col("borough").cast(pl.Utf8),
        pl.col("latitude").cast(pl.Float64),
        pl.col("longitude").cast(pl.Float64)
    ])
    transform_logger.info("Data types transformed")
    return df




# Funciton to clean string columns before mapping them
def clean_strings_before_mapping(df: pl.DataFrame, cols) -> pl.DataFrame:
    for col in cols:
        if col in df.columns:
            df = df.with_columns(
                pl.col(col)
                .str.strip_chars()
                .str.replace_all(r"\s+", " ")
                .alias(col)
            )
    transform_logger.info("String columns cleaned")
    return df




# Function to filter newly pulled data with complaint types not relevant.
def filter_relevant_complaints(df: pl.DataFrame, relevant_complaints) -> pl.DataFrame:
    df = df.filter(
        (pl.col("complaint_type").str.contains(r"^[a-zA-Z0-9\s\.,\-\(\)&]+$")) &
        (pl.col("complaint_type").is_in(relevant_complaints))
    )
    transform_logger.info(f"Filtered relevant complaints: {df.height} rows remain")
    return df

```

```python
# Function to deduplicate data based duplicate "unique_key", keeping the first record
def dedupe(df: pl.DataFrame) -> pl.DataFrame:
    df = df.sort("created_date", reverse = False)
    transform_logger.info(f"Deduplicated complaints: {df.height} rows remain")
    return df.unique(subset = "unique_key", keep = "first")




# Function to clean zip codes - replacing zip codes under 4 digits to null, and replacing zip codes with over 5 to only the first 5
def clean_zip_codes(df: pl.DataFrame, zip_col: str = "incident_zip") -> pl.DataFrame:
    if zip_col in df.columns:
        df = df.with_columns(
            pl.when(pl.col(zip_col).str.len_chars() < 5)
            .then(None)
            .otherwise(pl.col(zip_col).str.slice(0, 5))
            .alias(zip_col)
        )
    transform_logger.info("Zip codes cleaned")
    return df




# Function to make a select list of string columns to be title cased
def title_casing(df: pl.DataFrame) -> pl.DataFrame:
    title_case_columns = [
        "descriptor", "location_type", "city", "status",
        "borough", "complaint_category", "agency_name", "complaint_type"
    ]
    for col in title_case_columns:
        if col in df.columns:
            df = df.with_columns(
                pl.col(col)
                .str.to_titlecase()
                .alias(col)
            )
```

```python
    # Making the agency acronym column all caps
    if "agency" in df.columns:
        df = df.with_columns(
            pl.col("agency").str.to_uppercase().alias("agency")
        )

    transform_logger.info("Applied title casing")
    return df



# Function to apply mappings based on JSON files in order to reduce computation for common mispellings
# Falls back on fuzzy string matching if there are categories that do not match keys (80% similiarity score matching)
def apply_mapping(df: pl.DataFrame, column, mapping, use_fuzzy = True, score_cutoff = 80) -> pl.DataFrame:
    if column not in df.columns or not mapping:
        return df

    # Applying mappings before falling back to fuzzy string matching
    df = df.with_columns(
        pl.col(column)
        .map_elements(lambda x: mapping.get(x, x))
        .alias(column)
    )

    if use_fuzzy:
        # Collecting unique values that still are not mapped after applying mappings (still unclean categories)
        unmapped = [x for x in df[column].unique().to_list() if x not in mapping.values() and x is not None]

        # Dictionary for fuzzy mapping
        fuzzy_map = {}
        for val in unmapped:
            # Finding the best match based on an 80% similiary cutoff score to exisiting categories.
            best_match = process.extractOne(val, mapping.keys(), scorer = fuzz.ratio)
            if best_match and best_match[1] >= score_cutoff:
                fuzzy_map[val] = mapping[best_match[0]]

        if fuzzy_map:
            df = df.with_columns(
```

```python
                pl.col(column)
                .map_elements(lambda x: fuzzy_map.get(x, x))
                .alias(column)
            )
    transform_logger.info(f"Applied mapping on {column} with fuzzy matching: {use_fuzzy}")
    return df



def transform_311(df: pl.DataFrame, mappings: dict) -> pl.DataFrame:
    df = data_type_transformer(df)

    df = clean_strings_before_mapping(df, [
        "complaint_type", "location_type", "city", "borough", "agency_name", "descriptor"
    ])

    df = filter_relevant_complaints(df, mappings["relevant_complaints"])

    df = dedupe(df)

    mapping_columns = {
        "complaint_type": "complaint_mapping",
        "complaint_category": "complaint_categories",
        "agency": "agency_mapping",
        "agency_name": "agency_name_mapping",
        "city": "city_mapping",
        "borough": "borough_mapping",
        "location_type": "location_type_mapping"
    }
    for col, mapping_name in mapping_columns.items():
        df = apply_mapping(df, col, mappings.get(mapping_name, {}))

    df = clean_zip_codes(df)

    df = title_casing(df)

    str_columns = [col for col, dtype in zip(df.columns, df.dtypes) if dtype == pl.Utf8]
    df = df.with_columns([
        pl.when(pl.col(col) == "missing")
        .then(None)
        .otherwise(pl.col(col))
        .alias(col)
        for col in str_columns
    ])

    return df


# Function entry point for the main 311 transformation functi
if __name__ == "__main__":
    transform_311()
```

Transform code for Weather dataset

```python
# transform/transform_311_weather.py
import polars as pl

def transform_combined(cases: pl.DataFrame, weather: pl.DataFrame) -> dict:
    # Ensuring data types
    cases = cases.with_columns([
        pl.col("unique_key").cast(pl.Int64),
        pl.col("created_date").cast(pl.Date),
        pl.col("closed_date").cast(pl.Date),
        pl.col("resolution_action_updated_date").cast(pl.Date),
        pl.col("agency").cast(pl.Utf8),
        pl.col("agency_name").cast(pl.Utf8),
        pl.col("complaint_type").cast(pl.Utf8),
        pl.col("descriptor").cast(pl.Utf8),
        pl.col("location_type").cast(pl.Utf8),
        pl.col("incident_zip").cast(pl.Utf8),
        pl.col("city").cast(pl.Utf8),
        pl.col("status").cast(pl.Utf8),
        pl.col("borough").cast(pl.Utf8),
        pl.col("latitude").cast(pl.Float64),
        pl.col("longitude").cast(pl.Float64),
        pl.col("complaint_category").cast(pl.Utf8),
    ])
    cases = cases.with_columns([
        pl.when(pl.col("closed_date") < pl.col("created_date"))
        .then(None)
        .otherwise(pl.col("closed_date"))
        .alias("closed_date")
    ])

    weather = weather.with_columns([
        pl.col("time").str.strptime(pl.Date, "%Y-%m-%d").alias("date"),
        pl.col("temperature_2m_max").cast(pl.Float64),
        pl.col("temperature_2m_min").cast(pl.Float64),
        pl.col("precipitation_sum").cast(pl.Float64),
        pl.col("precipitation_hours").cast(pl.Int64),
        pl.col("rain_sum").cast(pl.Float64),
        pl.col("showers_sum").cast(pl.Float64),
        pl.col("snowfall_sum").cast(pl.Float64),
        pl.col("windspeed_10m_max").cast(pl.Float64),
        pl.col("windgusts_10m_max").cast(pl.Float64),
        pl.col("borough").cast(pl.Utf8),
        pl.col("latitude").cast(pl.Float64),
        pl.col("longitude").cast(pl.Float64),
    ])

    # dim_date
    dim_date = pl.concat([
        cases.select(["created_date", "closed_date"]).melt().select("value").rename({"value":"date"}),
        weather.select(["time"]).rename({"time":"date"})
    ]).unique().with_columns([
        pl.col("date").cast(pl.Date),
        pl.arange(1, pl.count() + 1).alias("date_id")
    ]).sort("date")

    dim_date = dim_date.with_columns([
        pl.col("date").dt.day().alias("day"),
        pl.col("date").dt.month().alias("month"),
        pl.col("date").dt.year().alias("year"),
        pl.col("date").dt.weekday().alias("weekday"),
        pl.col("date").dt.strftime("%A").alias("weekday_name")
    ])

    # dim_borough
    dim_borough = pl.DataFrame({
        "borough_id": [1, 2, 3, 4, 5],
        "borough_name": ["Manhattan", "Brooklyn", "Queens", "Bronx", "Staten Island"]
    })

    # dim_location
    weather_loc = weather.select([
        pl.col("borough"),
        pl.col("latitude"),
        pl.col("longitude"),
        pl.lit(None).cast(pl.Utf8).alias("city"),
        pl.lit(None).cast(pl.Utf8).alias("location_type"),
```

```python
        pl.lit(None).cast(pl.Utf8).alias("incident_zip"),
    ])
    dim_location = pl.concat([
        cases.select(["borough","latitude","longitude","city","location_type","incident_zip"]),
        weather_loc
    ]).unique().with_columns([
        pl.arange(1, pl.count() + 1).alias("location_id")
    ])
    dim_location = dim_location.select([
        "location_id","incident_zip","borough","city","location_type","latitude","longitude"
    ])

    # dim_agency
    dim_agency = cases.select(["agency","agency_name"]).unique().with_columns([
        pl.arange(1, pl.count() + 1).alias("agency_id")
    ])

    # dim_complaint_type
    dim_complaint_type = cases.select(["complaint_type","descriptor","complaint_category"]).unique().with_columns([
        pl.arange(1, pl.count() + 1).alias("complaint_type_id")
    ])


    # fact_incidents
    fact_incidents = cases.join(
        dim_date.rename({"date": "created_date", "date_id": "created_date_id"}),
        on="created_date", how="left"
    ).join(
        dim_date.rename({"date": "closed_date", "date_id": "closed_date_id"}),
        on="closed_date", how="left"
    ).join(
        dim_location,
        on=["borough", "latitude", "longitude"], how="left"
    ).join(
        dim_borough, on="borough", how="left"
    ).join(
        dim_agency, on="agency", how="left"
    ).join(
        dim_complaint_type,
        on=["complaint_type", "descriptor", "complaint_category"], how="left"
    ).with_columns([
        ((pl.col("closed_date") - pl.col("created_date"))).alias("time_to_resolve_interval"),
        ((pl.col("closed_date") == pl.col("created_date")).cast(pl.Int64)).alias("is_resolved_same_day"),
        pl.lit(1).alias("complaint_count")
    ])

    # Dropping unnecessary columns
    drop_cols_incidents = [
        "agency_name", "complaint_type", "descriptor", "location_type",
        "incident_zip", "city", "status", "resolution_action_updated_date",
        "borough", "latitude", "longitude"
    ]
    fact_incidents = fact_incidents.drop([col for col in drop_cols_incidents if col in fact_incidents.columns])

    # fact_weather
    weather = weather.with_columns([pl.col("time").cast(pl.Date).alias("date")])

    fact_weather = weather.rename({
        "temperature_2m_max": "temperature_max",
        "temperature_2m_min": "temperature_min",
        "precipitation_sum": "precipitation_total",
        "rain_sum": "rain_total",
        "showers_sum": "showers_total",
        "snowfall_sum": "snowfall_total",
        "windspeed_10m_max": "windspeed_max",
        "windgusts_10m_max": "windgust_max"
    }).join(
        dim_date.rename({"date": "date", "date_id": "date_id"}), on="date", how="left"
    ).join(
        dim_borough, on="borough", how="left"
    ).with_columns([
        (pl.col("rain_total") > 0).cast(pl.Int64).alias("rain_flag"),
        (pl.col("showers_total") > 0).cast(pl.Int64).alias("showers_flag"),
        (pl.col("snowfall_total") > 0).cast(pl.Int64).alias("snow_flag"),
        ((pl.col("windspeed_max") > 15) | (pl.col("windgust_max") > 20)).cast(pl.Int64).alias("high_wind_flag")
    ])
    # Filtering cols
    fact_weather = fact_weather.select([
        "date_id", "borough_id", "temperature_max", "temperature_min", "precipitation_total",
        "precipitation_hours", "rain_total", "showers_total", "snowfall_total", "windspeed_max",
        "windgust_max", "rain_flag", "showers_flag", "snow_flag", "high_wind_flag"
    ])

    # fact_daily_summary
    daily_incidents = fact_incidents.group_by(["created_date_id","borough_id"]).agg([
        pl.count("incident_id").alias("total_incidents"),
        (pl.col("is_resolved_same_day").mean() * 100).alias("percent_resolved_same_day")
    ])

    daily_weather = fact_weather.group_by(["date_id","borough_id"]).agg([
        pl.col("temperature_max").mean().alias("temperature_max"),
        pl.col("temperature_min").mean().alias("temperature_min"),
        ((pl.col("temperature_max") + pl.col("temperature_min"))/2).mean().alias("temperature_avg"),
        pl.col("precipitation_total").sum().alias("precipitation_total"),
        (pl.when(pl.col("precipitation_total") > 0).then(pl.col("precipitation_total")/24).otherwise(0)).alias("precipitation_per_hour"),
        pl.col("rain_total").sum().alias("rain_total"),
        pl.col("showers_total").sum().alias("showers_total"),
        pl.col("snowfall_total").sum().alias("snowfall_total"),
        pl.col("windspeed_max").max().alias("windspeed_max"),
        pl.col("windgust_max").max().alias("windgust_max"),
        pl.col("rain_flag").max().alias("rain_flag"),
        pl.col("showers_flag").max().alias("showers_flag"),
        pl.col("snow_flag").max().alias("snow_flag"),
        pl.col("high_wind_flag").max().alias("high_wind_flag")
    ])

    fact_daily_summary = daily_incidents.join(
        daily_weather, left_on=["created_date_id","borough_id"], right_on=["date_id","borough_id"], how="left"
    )

    # Renamung columns to match BigQuery schema
    dim_date = dim_date.rename({
        "date_id":"date_id","date":"date","day":"day","month":"month",
        "year":"year","weekday":"weekday","weekday_name":"weekday_name"
    })
    dim_borough = dim_borough.rename({"borough_id":"borough_id","borough_name":"borough_name"})
    dim_location = dim_location.rename({
        "location_id":"location_id","incident_zip":"incident_zip","borough":"borough",
        "city":"city","location_type":"location_type","latitude":"latitude","longitude":"longitude"
    })
    dim_agency = dim_agency.rename({
        "agency_id":"agency_id","agency":"agency","agency_name":"agency_name"
    })
    dim_complaint_type = dim_complaint_type.rename({
        "complaint_type_id":"complaint_type_id","complaint_type":"complaint_type",
        "complaint_category":"complaint_category","descriptor":"complaint_descriptor"
    })
    fact_incidents = fact_incidents.rename({
        "unique_key":"incident_id","created_date_id":"created_date_id","closed_date_id":"closed_date_id",
        "agency_id":"agency_id","complaint_type_id":"complaint_type_id","date_id":"date_id",
        "location_id":"location_id","resolution_status":"resolution_status",
        "time_to_resolve_interval":"time_to_resolve_interval",
        "is_resolved_same_day":"is_resolved_same_day",
        "complaint_count":"complaint_count"
    })
    fact_weather = fact_weather.rename({
        "date_id":"date_id","borough_id":"borough_id","temperature_max":"temperature_max",
        "temperature_min":"temperature_min","precipitation_total":"precipitation_total",
        "precipitation_hours":"precipitation_hours","rain_total":"rain_total",
        "showers_total":"showers_total","snowfall_total":"snowfall_total","windspeed_max":"windspeed_max",
        "windgust_max":"windgust_max","rain_flag":"rain_flag","showers_flag":"showers_flag",
        "snow_flag":"snow_flag","high_wind_flag":"high_wind_flag"
    })
    fact_daily_summary = fact_daily_summary.rename({
        "created_date_id":"date_id","borough_id":"borough_id","total_incidents":"total_incidents",
        "percent_resolved_same_day":"percent_resolved_same_day","temperature_avg":"temperature_avg",
        "temperature_max":"temperature_max","temperature_min":"temperature_min",
        "precipitation_total":"precipitation_total","precipitation_per_hour":"precipitation_per_hour",
        "rain_total":"rain_total","showers_total":"showers_total","snowfall_total":"snowfall_total",
        "windspeed_max":"windspeed_max","windgust_max":"windgust_max",
        "rain_flag":"rain_flag","showers_flag":"showers_flag","snow_flag":"snow_flag",
        "high_wind_flag":"high_wind_flag"
    })

    # Return all tables in a dict
    return {
        "dim_date": dim_date,
        "dim_borough": dim_borough,
        "dim_location": dim_location,
        "dim_agency": dim_agency,
        "dim_complaint_type": dim_complaint_type,
        "fact_incidents": fact_incidents,
        "fact_weather": fact_weather,
        "fact_daily_summary": fact_daily_summary
    }
```

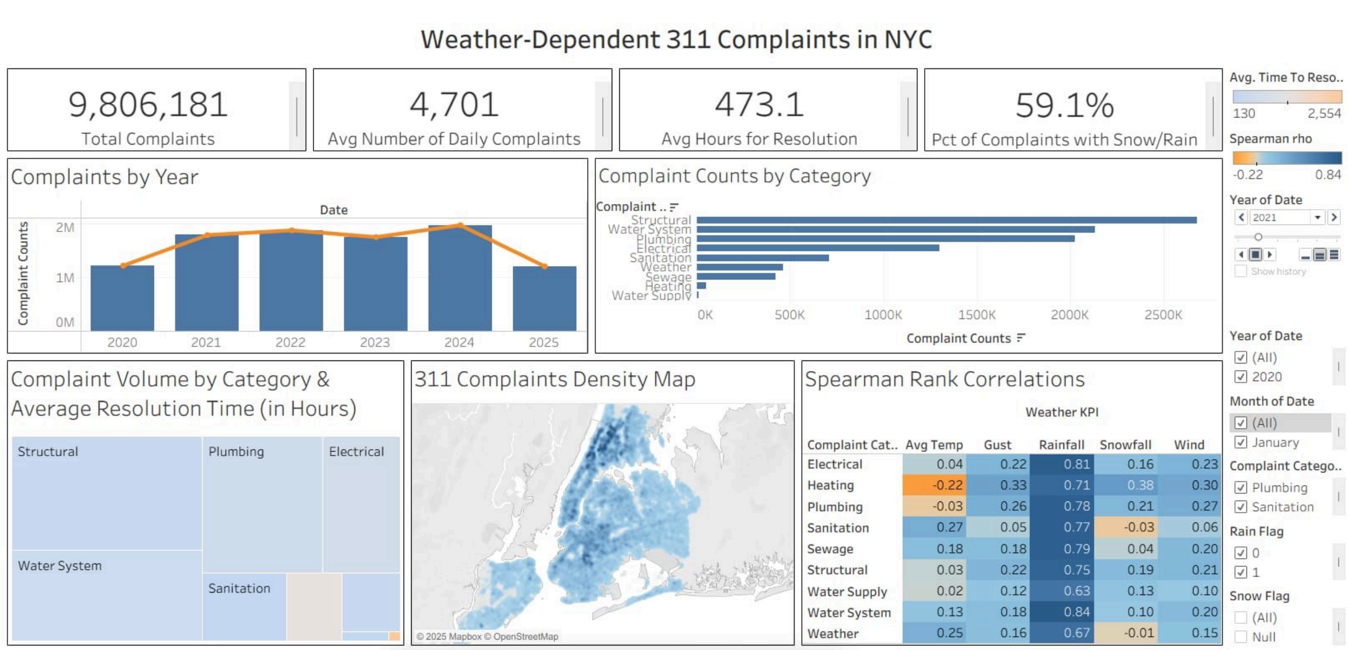
Load code for data

```python
from google.cloud import bigquery
from google.oauth2 import service_account
import polars as pl
from pathlib import Path
from logger.etl_logger import ETLLogger

# Initializing logger
load_logger = ETLLogger("load").get()
load_logger.info("Starting data loading")

# BigQuery client setup
script_dir = Path(__file__).parent
credentials_path = script_dir.parent / "credentials" / "bigquery_nyc_weather_etl_credentials.json"
credentials = service_account.Credentials.from_service_account_file(credentials_path)
client = bigquery.Client(project="nyc-311-weather-etl", credentials=credentials)
dataset_id = "nyc_311_weather"

def load_to_bigquery(df_dict, chunk_size = 10_000):
    for table_name, df in df_dict.items():
        if df is None or df.height == 0:
            load_logger.info(f"No data for table {table_name}, skipping...")
            continue
        load_logger.info(f"Starting load for table {table_name} ({df.height} rows)")
        table_ref = f"{client.project}.{dataset_id}.{table_name}"
        # Check if table exists
        try:
            table = client.get_table(table_ref)
            table_exists = True
        except Exception:
            table_exists = False
            load_logger.info(f"Table {table_name} does not exist. Will create new table automatically.")
        # Convert Polars to pandas for BigQuery
        df_pd = df.to_pandas()

        if table_name.startswith("dim_") and table_exists:
            # For dimension tables, remove duplicates based on primary key
            pk_field = table.schema[0].name
            # Load existing primary keys
            existing_df = client.list_rows(table).to_dataframe()
            df_pd = df_pd.merge(existing_df[[pk_field]], on=pk_field, how="left", indicator=True)
            df_pd = df_pd[df_pd["_merge"] == "left_only"].drop(columns=["_merge"])
            load_logger.info(f"{len(df_pd)} new rows detected for {table_name}")

        if len(df_pd) == 0:
            load_logger.info(f"No new rows to load for {table_name}")
            continue

        # Loading in chunks
        for start in range(0, len(df_pd), chunk_size):
            end = start + chunk_size
            chunk_df = df_pd.iloc[start:end]

            try:
                job = client.load_table_from_dataframe(
                    chunk_df,
                    table_ref,
                    job_config=bigquery.LoadJobConfig(write_disposition="WRITE_APPEND")
                )
                job.result()  # Wait for completion
                load_logger.info(f"Loaded rows {start} to {end} into {table_name}")
            except Exception as e:
                load_logger.error(f"Error loading rows {start} to {end} into {table_name}: {e}")

        load_logger.info(f"Finished loading table {table_name}")

if __name__ == "__main__":
    load_to_bigquery()
```

Part VI: Visuals and Dashboard

[BI Application]

Our Tableau dashboard uses data from the integrated data warehouse to analyze water-related 311 complaints and their relationship to weather conditions across New York City. Below, we describe four key visualizations in detail, followed by brief descriptions of the remaining dashboard elements.



Summary KPI Metrics

This visualization shows high-level summary metrics, including total water-related 311 complaints, average daily complaints, average resolution time in hours, and the percentage of complaints that occurred on days with rain or snow. These KPIs update based on selected filters and provide a quick overview of complaint volume, response efficiency, and weather-related impact.

Complaints by Year

This chart displays the total number of water-related 311 complaints by year using bars, with a line showing the overall trend. It allows us to compare complaint volume across years and identify periods where water-related issues increased or decreased.

Complaint Counts by Category

This horizontal bar chart shows the total number of complaints by category, such as Structural, Water System, Plumbing, and Sanitation. The categories are ordered by volume, making it easy to see which types of issues are reported most frequently.

Complaint Volume by Category and Average Resolution Time

This heatmap visualizes complaint volume by category, where the size of each block represents the number of complaints and the color reflects the average resolution time. It helps compare both workload and service efficiency across different types of water-related issues.

311 Complaints Density Map

This map shows the geographic density of water-related 311 complaints across New York City. Darker areas indicate higher concentrations of complaints, allowing us to identify neighborhoods where water-related infrastructure issues are more common.

Spearman Rank Correlations Between Weather and Complaints

This heatmap displays the Spearman rank correlation between complaint categories and weather variables such as rainfall, snowfall, wind and temperature. The stronger color shading indicates stronger relationships, helping us understand which complaint types are most influenced by weather conditions.

Part VI: Conclusion

For our group, we used multiple tools to complete the project. Our main form of communication was WhatsApp, which we used for instant messaging and to set up Zoom or Google Meet calls. Through these platforms, we decided on our project and which softwares to use.

- Open-Meteo API: We used this to gather data on weather in the 5 boroughs, to correspond to our 311 Complaint data
- draw.io: We used this to design our dimensional models
- python: We used this to design and complete our ETL Programming
- Tableau: We used this to create visualizations for our data

The most difficult step in this process was completing the ETL Programming. We experienced a lot of issues with loading the data via Python, which suffered from local hardware limits. The easiest step was designing the Dimensional Model because we had already decided on the KPIs and relevant dimensions.

The results of our business application show that the proposed benefits of the system can be realized. The dashboard indicates that water-related 311 complaints increase on days with rain or snow, and that certain complaint categories, such as structural issues, appear more frequently during periods of heavy rainfall. The analysis also shows that some complaint types not only occur more often but also take longer to resolve, suggesting that severe weather places additional strain on city infrastructure and services. The category level and correlation analyses further support this by highlighting clear relationships between precipitation levels and specific types of complaints.

Overall, this project demonstrates the value of combining 311 service request data with weather data in a centralized data warehouse. By integrating multiple data sources and using business intelligence tools, we were able to identify patterns that would not be visible using a single dataset. The results of this application can be used to better understand how weather-related events impact NYC residents and to prepare more targeted responses as rainfall is expected to increase. For example, when heavy rainfall is forecasted, additional attention and resources can be directed toward areas that frequently report structural complaints, allowing the city to focus on failing infrastructure and provide increased support where it is most needed.

Part VII: Meeting Log

Meeting Log			
Date/Time	Modality	Attendees	Objective
9/9 16:00-17:00	Zoom	Adrian, Neha, Adnan, Roland	Filter 311 Data Set
9/10 14:30-15:15	Whatsapp	Adrian, Neha, Adnan, Roland	Discuss Secondary Data Sets
9/12 18:00-23:00	Whatsapp	Adrian, Neha, Adnan, Roland	Finalize Milestone 1 Submission
9/20 13:30-14:30	Zoom	Adrian, Neha, Adnan, Roland	Discuss Secondary Data Sets, Pick out KPIs
9/26 14:00-16:30	Zoom	Adrian, Neha, Adnan, Roland	Data Mart Schema
10/3 18:00-23:00	Zoom	Adrian, Neha, Adnan, Roland	Finalize Milestone 2 Submission
10/21 18:00-20:00	Zoom	Adrian, Neha, Adnan, Roland	Discuss and Finalize Milestone 3 Submission
11/7 18:00-19:00	Whatsapp	Adrian, Neha, Adnan, Roland	Discuss ETL Tools and DBMS
11/14 17:00-18:00	Zoom	Adrian, Neha, Adnan, Roland	Discussed Technical Architecture Components
12/1 18:00-20:00	Whatsapp	Adrian, Neha, Adnan, Roland	Discussed ETL Programming

12/5 16:00-20:00	Zoom	Adrian, Neha, Adnan, Roland	Discussed ETL Programming
12/19 16:00-18:00	Zoom	Adrian, Neha, Adnan, Roland	Finalized Visuals and Dashboard on Tableau

References

Data Source - 311 Complaint Data
https://data.cityofnewyork.us/Social-Services/311-Service-Requests-from-2010-to-Present/erm2-nwe9/about_data

Data Source - Weather In 5 Boroughs https://open-meteo.com/

Technique - Designing Star Schema
https://www.geeksforgeeks.org/data-analysis/designing-the-star-schema-in-data-warehousing/

Technique - ETL Programing in Python https://www.datacamp.com/courses/etl-and-elt-in-python

Technique - Creating a Dashboard https://youtu.be/vDgBCgxLWPY?si=5AW5bkl5S4PbOd5I