THE UNIVERSITY OF SYDNEY

DATA ENGINEERING COMP5339

ASSIGNMENT 2 GROUP 88

DETAILS: COMP5339-GROUP 88

No	Unikey	SID
1.	msha0502	530792813
2.	psin0239	530627113

Table of Contents:

Figures:

1. Introduction

The project focuses on building a complete data pipeline for the real time fuel pricing data from the New South Wales(NSW) Government's FuelCheck platform. The project focuses on applying core data engineering techniques which include data retrieval, integration, transformation, storage, real time publishing , subscribing and interactive visualization using python.

The project comprises of comprehensive system that aggregates real-time gasoline price updates, organises the data in a structured format, disseminates each update via MQTT, and presents the information in an interactive, dynamic dashboard with Streamlit and Folium which simulates a real-world data streaming environment and enhances comprehension of the integration of batch and stream data processing for a seamless user experience.

2. Data Retrieval

The project began by acquiring fuel price datasets from the NSW Government's FuelCheck API. We automated the data collection through the NSW Web API. The system first authenticates OAuth 2.0 client credentials flow. The process first involves getting the access which is used to authorize subsequent API requests. The script initiated with process by calling **fetch_access_token()** which uses the base url as **https://api.onegov.nsw.gov.au**, utilizing the request library to send a GET request to the token URL, to encode the API key and secret with **base64** for Basic Authentication also included error handling with logging to handle any potential failures to ensure the API calls are reliable when called subsequently.
The access token once secured, the script fetched fuel price data using **fetch_fuel_data()**. For the initial run it used /FuelPriceCheck/v1/fuel/prices to retrieve the full dataset while followed by subsequent iterations for getting updated data via /FuelPriceCheck/v1/fuel/prices/new which simulates an unbounded data stream. Each request headers with api key, a unique transaction id from uuid and a request timestamp from datetime using **datetime.now()** in (UTC format), which ensures compliance requirement of API which is achieved through subscribing to the NSW Fuel API.

3. Data Integration and Storage

The data integration process began with fetching initial fuel price data from the NSW Government's FuelCheck API, focusing on two endpoints: 1.) **/FuelPriceCheck/v1/fuel/prices** endpoint for initial data , which returned a JSON response containing two main components: a list of stations and a list of prices. The sample data provided for this endpoint included stations with details such as brandid, station_id, brand, code, name, address, location (with latitude and longitude), and isAdBlueAvailable, alongside prices with stationcode, fueltype, price, and last_updated. 2.) The update endpoint, **/FuelPriceCheck/v1/fuel/prices/new,** provided data for stations and prices that had changed since the last call, with a similar structure but only for updated records.

The Data was fetched using the **fetch_fuel_data()** for the initial endpoint which parses the JSON response into a list of stations and prices. The data transformed into the dataframe using **pandas**: for storing the details of the station detail station_df and prices_df for price information. The **process_fuel_data()** handles the merging of both the columns on **station_code**. For cleaning the raw data the steps involved were as follows filling missing values (e.g., brandid and station_id as empty string were filled with defaults or left as it is), converted the latitude, longitude and price to numeric type with pd.to_numeric and using **pd.to_datetime** to convert last_updated to datetime. The address column was split into components using string manipulation: address, suburb, state and postal_code, ensuring a structured output. The whole merged dataframe is stored and consolidated into a single csv named **fuelprice.csv**. The script is updated from the /FuelPriceCheck/v1/fuel/prices/new using the same **fetch_fuel_data()** every 60 seconds , which processes the new data coming into new_df with the same cleaning steps. Using **pd.read_csv**, the existing fuelprice.csv was loaded into existing_df and last_updated was converted to datetime for the consistency of the data throughout. The new data was concatenated with the **existing_df** to form **combined_df** which ensures that all records were considered. The most recent data was retained to handle the duplicacy of the data, combined_df was **sorted** using **df.sort_values()** by station_code, fuel_type and last_updated, then **grouped** by **station_code** and **fuel_type**, keeping the last row(with the latest record) for each group using **df.groupby()** and **last()**.The combined DataFrame was updated to fuelprice.csv using the **update_fuel_csv_and_publish()**, guaranteeing the file consistently held the most current information for all stations and fuel kinds.

4. Data Publishing via MQTT

In this project, MQTT(Message Queuing Telemetry Transport) was used to publish the fuel price updates in real time retrieved from the NSW FuelCheck API, enabling downstream consumers to receive live data changes efficiently. The **paho-mqtt** Python library was installed via **pip install paho-mqtt** to handle MQTT client operations. MQTT (Message Queuing Telemetry Transport) is a lightweight protocol designed for real-time data transmission, particularly suited for IoT applications. It employs a publish-subscribe architecture in which publishers transmit messages to topics, and subscribers obtain them through a broker (Anna, Vijayalakshmi, & Kota, 2024).

Mosquitto, which is an open source MQTT broker, is utilized to manage the distribution of messages between the publishing **data_retrieval.py** and subscribing **visualization.py** scripts. It has some callbacks: 1.) **on connect**: subscribed to **nsw/fuel/prices** upon successful connection. 2.) **on_message**: Logged received messages (used for debugging, thoroughly during the subscribing part in visualizatoin.py). These both enable transmission of fuel price updates in real time visualization. Managing a high volume of data while publishing multiple fuel types per station could overload the broker which is prevented by 0.1 second delay between messages using **time.sleep(0.1)** which makes the script stable and balanced through without and crashes. Station_code, brand, name, address, suburb, state, postal_code, latitude, longitude, fuel_type, price, and last_updated are among the fields that are expected in JSON messages. With each station having a dictionary of gasoline prices, it updated **map_data** to retain a nested structure and made sure that all available fuel types (DL, E10, P95, P98, U91, and PDL, for example) were tracked with their timestamps a detail that had not been thoroughly examined previously.

It processes the incoming MQTT messages and updates the dashboard's data state and dynamic changes updates are rendered. It parses the JSON payload using the **json.load(msg.payload.decode())** which extracts the station code and fuel type and updates it using the **st.session_state.map_data** accordingly. It handles last_updated (date and time) by firstly converting it to datetime object and then creates a new dataframe row to concatenate with the **st.session_state.fuel_data** followed by incrementing **st.session_state.update_trigger** to force a re rendering via **st.experimental_rerun()** make sure that streamlit app re- ran to reflect new data, leveraging **update_trigger** as key for **st_folium** which is further used in map rendering.

Continuous Execution

The MQTT was initiated by setting up the broker with Mosquitto on **localhost:1883**. Then MQTT client was created and client initialization was done by connecting with the broker using the **initialize_mqtt_client()**.Each fuel price record is retrieved in the json format and then published to **nsw/fuel/prices** with 0.1 second delay each row in a dataframe published as a json message to MQTT along with the continuous updates were published every 60 seconds, ensuring a real time data stream. If at all there during the specific 60 seconds there are no updated or new records, the script runs continuously till we receive the record and keeps on running. If, within the designated 60 seconds, there are no updated or new records, the script continues to execute until a record is received, displaying empty fields in the station and price attributes in the JSON output for no updates .

5. Data Subscribing and Visualization

We created a data subscribing and visualization pipeline using MQTT and Streamlit to simulate a real-time streaming environment for live insights into NSW fuel prices. The main objective of this stage involved creating a real-time interactive map that showed fuel prices received from live feeds with minimal delay while ensuring full interactivity to match the NSW FuelCheck interface.

MQTT Subscribing Logic

We implemented an MQTT client using the paho.mqtt.client Python library. This client subscribes to the topic nsw/fuel/prices and listens for published messages that were previously broadcast by our data publisher (from dataretrieval.py). Each message is a JSON object containing the station ID, brand, geographic coordinates, fuel type, price, and the last updated timestamp.

- Upon receiving a message, the subscriber.
- Decodes the JSON payload
- Parses the last_updated field into a datetime object
- Appends the data into a global Pandas DataFrame for raw data storage (fuel_data_global)

- Updates an in-memory dictionary (map_data_global) that holds the most recent prices per station and fuel type

Our approach guaranteed that updates remained idempotent while preventing data duplication and preserving only the most recent information. The **client.loop_start()** function enables asynchronous processing for all MQTT operations to maintain a non-blocking interface.

Dynamic and Interactive Map Visualization

We employed Streamlit to build the frontend UI and used Folium to create a live map presentation of the data. The map displays markers which indicate each fuel station through icons and labels that show their fuel type and price. Key features of the visualization:

Dynamic Marker Rendering: Map markers appear dynamically at runtime without necessitating a complete rerendering of the map. Markers on the map get updated with new additions or modifications whenever new MQTT messages come in.

Brand Icons and Price Labels: Each marker uses a customized DivIcon which displays the current fuel price above the brand logo. Brand icons are embedded using base64 encoding and retrieved from a local /static directory.

Interactive Popups: Clicking a marker opens a popup showing detailed information including:

- Station name and brand
- Address (parsed into street, suburb, state, and postcode)
- All available fuel types at that station, along with their current prices and last updated timestamps

Fuel Type Selector: The dropdown menu lets users select fuel types such as E10, Diesel, Premium 98 to filter map markers which refresh automatically.

Brand Filter: By selecting multiple brands through a multi-select input users can compare prices and detect pricing patterns between different brands.

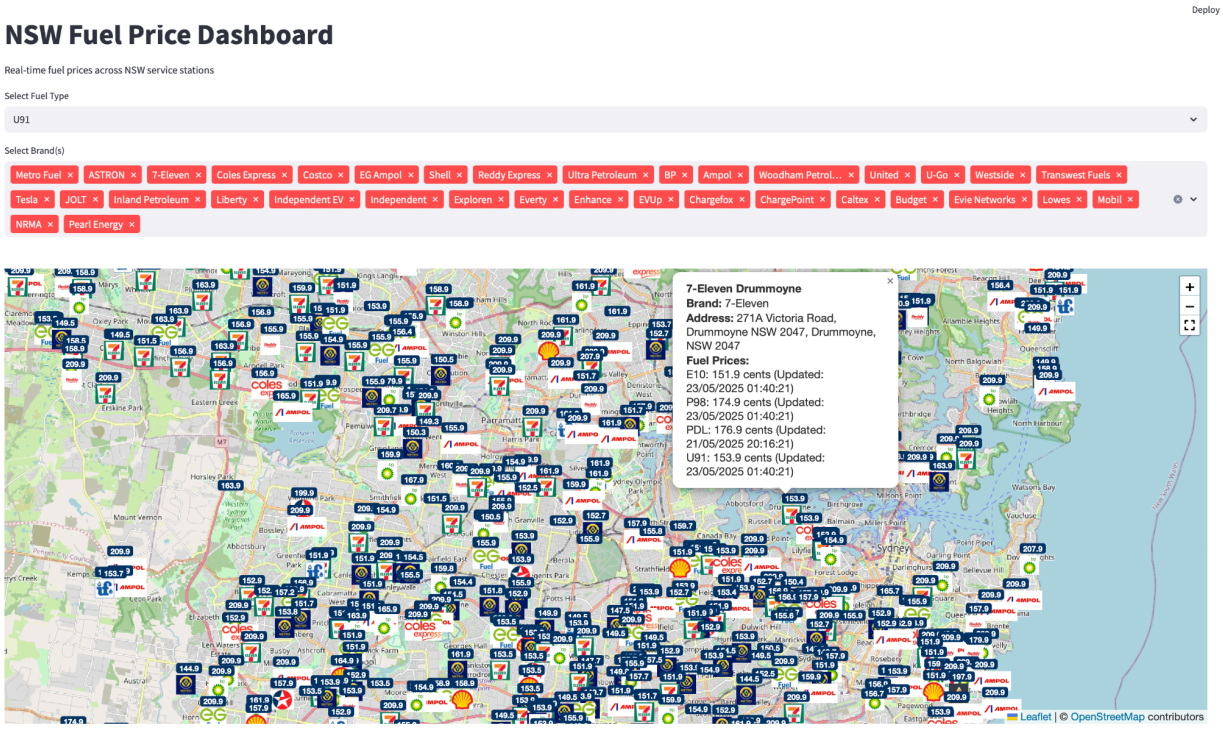

Fig1. Fuel Price Dashboard with dynamic markers, brand icons, filters and popups

Live Session Syncing

The global data (fuel_data_global, map_data_global) is synced with Streamlit's session_state to ensure live updates without page refresh. The Streamlit app is continuously listening and updating the map as new MQTT data arrives.

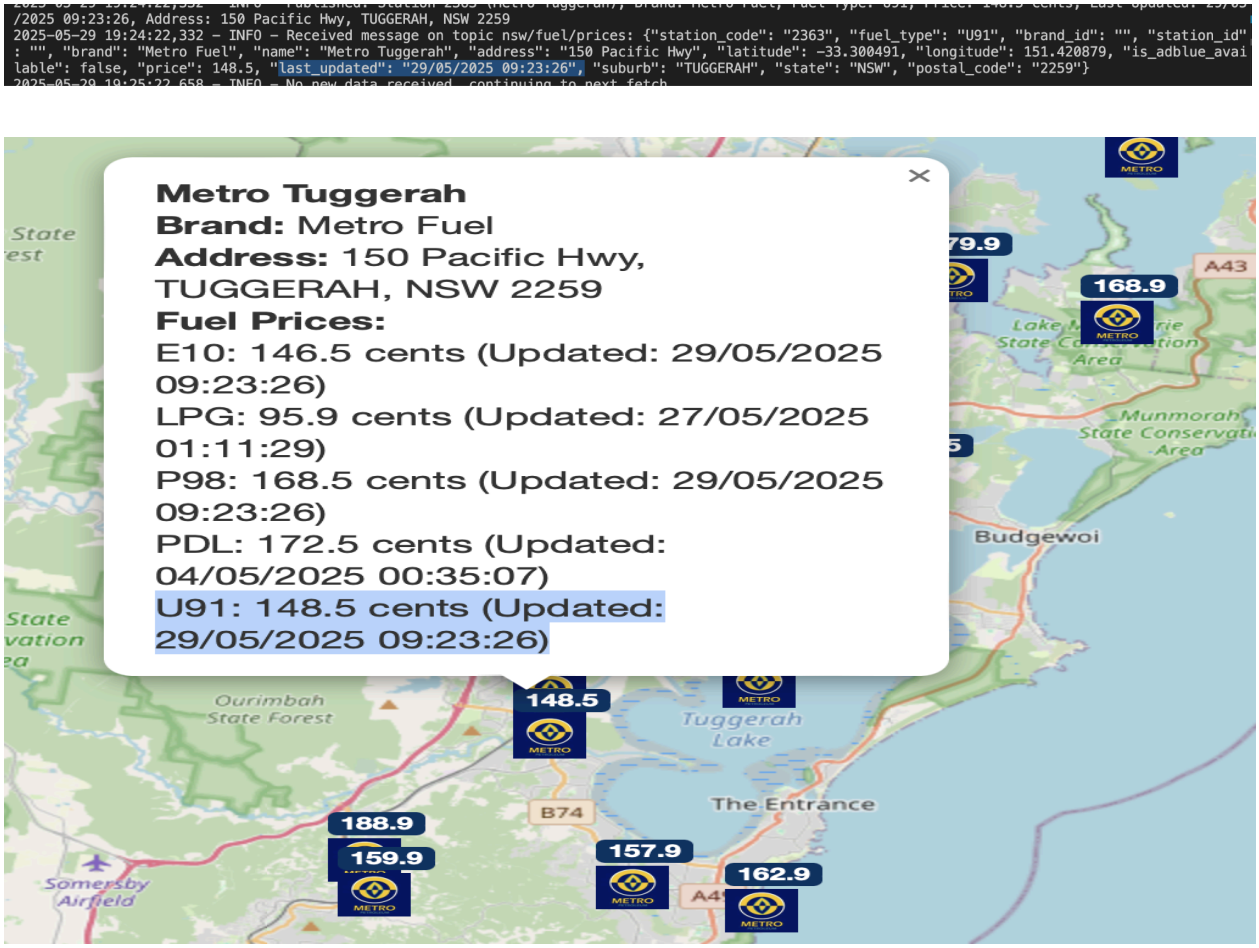

Fig2. Live update of Prices

Robustness and Performance

Our implementation covers error handling mechanisms for malformed messages together with fallback procedures for missing icon files. The map displayed only valid geographical coordinates and filtered out any rows lacking price or fuel type information.

Our solution shows how MQTT-based message handling and real-time mapping together enable effective subscription and visualization of live fuel price data. Our system enables dynamic updates along with interactive filtering and branded icon support which closely replicates the NSW FuelCheck user experience. The dashboard merges effective data management with user-focused design to deliver a strong scalable interface for fuel price monitoring that satisfies the project's technical requirements and user experience targets.

6. Key Insights and Findings

We derived multiple significant insights from our examination of the NSW FuelCheck dataset.

1. **Sparse Availability of Certain Fuel Types:** According to the dataset there are only two stations in NSW that distribute B20 biodiesel making it the least available fuel type in the dataset. A total of 28 stations provide E85 (ethanol) availability. Regular unleaded (U91) fuel can be purchased at more than 2,000 locations across the network making it the most accessible option.
2. **Brand Dominance and Distribution:** The brands Shell, Ampol, and BP maintain extensive coverage across different locations whereas smaller regional brands like Reddy Express and Ultra Petroleum function in limited areas typically found within specific suburbs.
3. **Electric Vehicle (EV) Chargers:** The electric vehicle network includes 758 stations which demonstrates growing support for EVs but still falls short compared to the 2,000 U91 stations.
4. **Data Quality Variation:** Multiple stations occupy identical geographic coordinates which indicates that they may operate under different brands or multiple fuel types together. Address formatting inconsistencies existed within the dataset and were resolved throughout the preprocessing stage.

7. Challenges Faced and Solutions

1. Data Retrieval and Initial JSON Collation

The API response separated station and fuel prices information into distinct list stations and prices which only had one shared key as **station_code**. Merging these two dataframe accurately to ensure consistency and completeness was important and challenging so converting all the values to compatible data types and joining the **station_df** and **prices_df** on **station_code** was a crucial step for creating a single consolidated view of fuel data.

2.Increment Updates from /FuelPriceCheck/v1/fuel/prices/new endpoint

While working with the /FuelPriceCheck/v1/fuel/prices/new endpoint. Unlike the initial full dataset, this incremental endpoint provided only the updated stations and price records. However, the data was often incomplete as the record like brandid and station_id was difficult to update with the previously stored data. While addressing this was retained and reused the existing metadata from the initial dataset wherever possible. Additionally, updating the fuelprice.csv file with only the most recent and valid records required careful deduplication logic. Further, when combining the new data (**new_df**) with the existing record (**existing_df**), sorting the combined data by station_code, fuel_type and last_updated and then grouping by station and fuel type to only keep the most recent record using **groupby().last()** which ensures that the CSV file remained accurate and up to date.

3. Synchronizing MQTT Messages Between Publisher and Subscriber Scripts

Managing real time MQTT message flow between the data_retrieal.py and visualization.py script was a one of the crucial challenges with major complexities.Given that MQTT functions asynchronously, ensuring synchronisation between message publication and real-time dashboard rendering necessitated meticulous design. We faced timing complications where messages were received in an incorrect sequence or not rendered accurately due to inadequate session state initialisation in Streamlit. To address this, we implemented several callback functions like **on_message()** to ensure structure implementation of JSON message and adding error handling is done along with proper receiving to the correctly formatted messages. Additionally, we added 0.1 second delay using **time.sleep(0.1)** as mentioned between the messages to prevent broker overload which increased the stability. The **update_trigger** mechanism in Streamlit was employed to compel re-renders and ensure UI responsiveness with each incoming update.

4. Dynamic Visualization Lag

The visualization.py script encountered both performance slowdowns and brand icon display errors at startup.The issue was that the display of brand icons was incorrect because certain icons were absent from the static directory or they had incorrect mappings.Our solution addressed for missing files included implementing a base64 fallback system with detailed logging to manage these scenarios smoothly. The map functioned properly to display prices when necessary icons were absent without any system disruptions. Additionally,the map performance got better because we introduced a global station info cache (**map_data_global**) that made updates only for markers that had changed.

5. Message Decoding and Synchronization

The issue with rapid arrival of MQTT messages initially led to overlapping updates and session-state inconsistencies within Streamlit. To address this solution, we implemented session-level synchronization by separating incoming data streams into two distinct layers: raw data (fuel_data_global) and deduplicated display-ready data (map_data_global).

8. Future recommendations and Improvements

- **Real-Time Clustering:** Implement clustering on the Folium map to prevent marker overlap in dense urban areas, especially for common fuel types like U91.
- **Historical Trends:** The dashboard should include features to display price trends over time with line charts based on stored CSV history data.
- **Alert System:** Users can set up price drop notifications for their preferred fuel brands and stations to help maintain fuel budgeting.
- **Icon Mapping Automation:** The mapping between brands and icon filenames needs to be automated through a central metadata table to eliminate manual mistakes.

9. Conclusion

This project demonstrated the end to end engineering of real time fuel price monitoring systems for NSW service fuel stations, from acquiring data to creating an interactive dashboard with visualization. We integrated live FuelCheck API data, cleaned and transformed it for consistency and published real time fuel data via MQTT mechanism. Our implementation handles data pipelining of real world data coming in high frequency in an interactive environment. The Streamlit dashboard subscribed to this data stream and dynamically rendered fuel station and prices with filtering and visualization which all together lays foundation for implementing efficient data preprocessing which is scalable and real time processing.

10. Team Contribution

Mohammed Junaid Shaikh (msha0502): Developed the development of data retrieval, integration, cleansing, and storage. Created the MQTT publishing pipeline, processed the initial and incremental data streams, and put the FuelCheck API access into practice. maintained effective logging and data management procedures and made sure the data pipeline ran continuously.
Paramvir Singh (psin0239): Created the dashboard for data subscribing and visualisation using Streamlit. Real-time map updates, interactive user interface elements, and MQTT subscription logic were implemented along with incorporating dynamic filtering, marker updates, and branded visual elements to improve the dashboard's responsiveness and user experience.

11. References

1. Anna, D. M., Vijayalakshmi, M. N., & Kota, S. R. (2024). Enabling lightweight device authentication in message queuing telemetry transport protocol. IEEE Internet of Things Journal, 11(9), 15792-15807.

2. Basic concepts of Streamlit - Streamlit Docs. (n.d.)https://docs.streamlit.io/get-started/fundamentals/main-concepts

(Basic Concepts of Streamlit - Streamlit Docs, n.d.)

3. Map — Folium 0.19.6 documentation. (n.d.). ttps://python-visualization.github.io/folium/latest/user_guide/map.html

(Map — Folium 0.19.6 Documentation, n.d.)

4. Mosquitto man page. (2021, November 3). Eclipse Mosquitto. https://mosquitto.org/man/mosquitto-8.html

(Mosquitto Man Page, 2021)