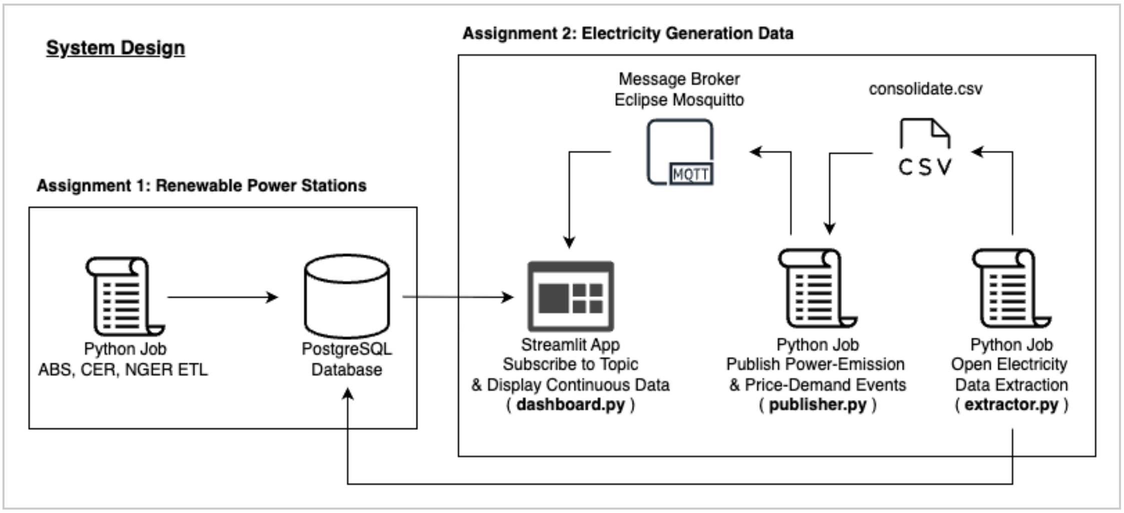
# Electricity Sector Data Streaming and Analysis

COMP5339 Assignment 2
Tutorial 1 group 8

Jonas Lim (jlim0114, 540885684)
Alok Das (adas061, 540932863)

## System Description

There are 4 essential components in this system that display continuous power generation data for the dashboard. The first component (extractor.py) is an ETL job that retrieves the last 1 week of per-facility power-emission and per-region price-demand data at a 5min interval; and processes it into a clean and consolidated csv file. The second component (publisher.py) is a persistent job which loads the csv file into memory, and publishes the per-facility or per-region events, on a 0.1 second delay,  to a MQTT broker. To simulate continuous data, the second component republishes events from the csv file again after a 60 second delay. The third component is the MQTT broker, which for simplicity, we leverage on the publicly available (Eclipse Mosquitto) MQTT broker. The fourth component is the streamlit dashboard application which subscribes to our MQTT topic and processes the events before rendering on the dashboard. The processing may require data from the PostgreSQL database completed from Assignment 1 (A1). To conform with the required submission format, we put the required data tables from A1 into csv files for this assignment and simulate querying from them.



## Data Retrieval And Integration (extractor.py)

**Getting list of NEM facility codes from Open Electricity (OE)**

The first step was to identify a list of facility codes in NEM (Eastern Australia) using OE's API [1]. However, there seems to be an issue accessing the data with python requests, and thus, we used postman to download the data. The data was (nem_faclities_json in context.py) was in unit level as a facility could have multiple generation units. We aggregated and filtered the facilities data to facility level by defining a facility as operational if at least one of its units has an operating status. Only facilities that are operational are retained. Of the 514 facilities in NEM, 419 had at least 1 operating unit. For each unit, we map the fuel type (fueltech_code) to a broader category (fueltech label) as provided on OE [2]. The result is a facility level data table about NEM facilities, which comprises name, code, latitude, longitude, region, and an array of fuel types, and is stored in PostgreSQL database under the table name, ***facility_lookup***. While iterating across the unit level facilities data, we built a hash map of unit codes to facility code. This unit-to-facility hashmap would be useful downstream when handling power and emission data, which are at unit level.

**Getting the power and emission data for each facility**

With a list of facility codes, the second step would be to extract the power and emission data using OE's API [3]. As the API would not allow specifying all the facility codes at once, we batched the facility codes into 17 batches of 25 each. Each request was with a date start of today, date end of 7 days ago, and an aggregating time interval of 5 minutes. We observe the response data was provided at the unit level. To construct a facility-level dataset, we iterate through the 17 response batches and, within each batch, first zip the data to pair each unit's power and emission records. We then perform a second zip operation to group the data by timestamp, aligning all uni-level information under the same timestamp. Each record, representing unit level power and emission for a timestamp is appended to a list, loaded as a pandas dataframe, and mapped with the unit-to-facility hashmap obtained earlier, to obtain its respective facility code. Lastly, we group by the facility code and timestamp and sum the power and emission data to obtain a facility level timestamp table for power and emission. At this stage, we observed 335 facilities with data during this date range.

**Getting the price and demand data for each region**

The third step involves extracting the price and demand data using OE's API [4]. Apart from setting 1 week date range and 5 minute aggregating time interval in the request, the primary group of network regions was also specified to retrieve the market data group by NEM regions. Using the double zip method, similarly used in transforming power and emission data we group the price and demand data for each region, and group the data for each timestamp, respectively. Each region's price-demand-timestamp record was appended into a list and the resultant list was loaded as a pandas dataframe.

**Consolidating into a single cache csv**

The facility-level power-emission and region-level price-demand dataframes were pivoted with timestamps as the index and their respective codes (facility or region) as columns. For the facility dataframe, the values are power and emission, while for the region dataframe, the values are price and demand. The resulting pivot tables have rows representing unique timestamps and columns corresponding to each facility's or region's respective values, with the codes successfully embedded into the column headers. Both pivot tables were merged on the timestamp column using a left join. This combines the facility power-emission data with the region price-demand data for each timestamp, ensuring that all facility records are retained even if some timestamps do not have corresponding region data. The final joint table is saved as ***consolidate.csv.***

## Data Publishing and Continuous Execution (publisher.py)

A connection was established with the Eclipse Mosquitto MQTT broker, and the consolidated CSV file was loaded into a Pandas dataframe. The column headers were used to extract the lists of facility codes and region codes, and the rows were already sorted in ascending timestamp order. For each timestamp, we first iterate over the facility codes to construct a per-facility power-emission event payload containing the facility code, timestamp, power, and emission. We then iterate over the region codes to construct a per-region price-demand event payload containing the region code, timestamp, price, and demand. Events for each timestamp are sent in sequence before moving to the next timestamp, ensuring that all messages are published in chronological order. Each event is published with a 0.1-second delay. Once all events from the dataframe have been published using the MQTT client with QoS 1, the system waits 60 seconds before starting the next round of republishing.

In a production environment, to avoid publishing duplicate events, a set is maintained to track each published event by (timestamp, facility code). After an event is published, its (timestamp, facility code) pair is added to the set. Before publishing subsequent events, the system checks this set and skips any event that has already been published.

## Data Subscribing & Dynamic Visualization (dashboard.py)

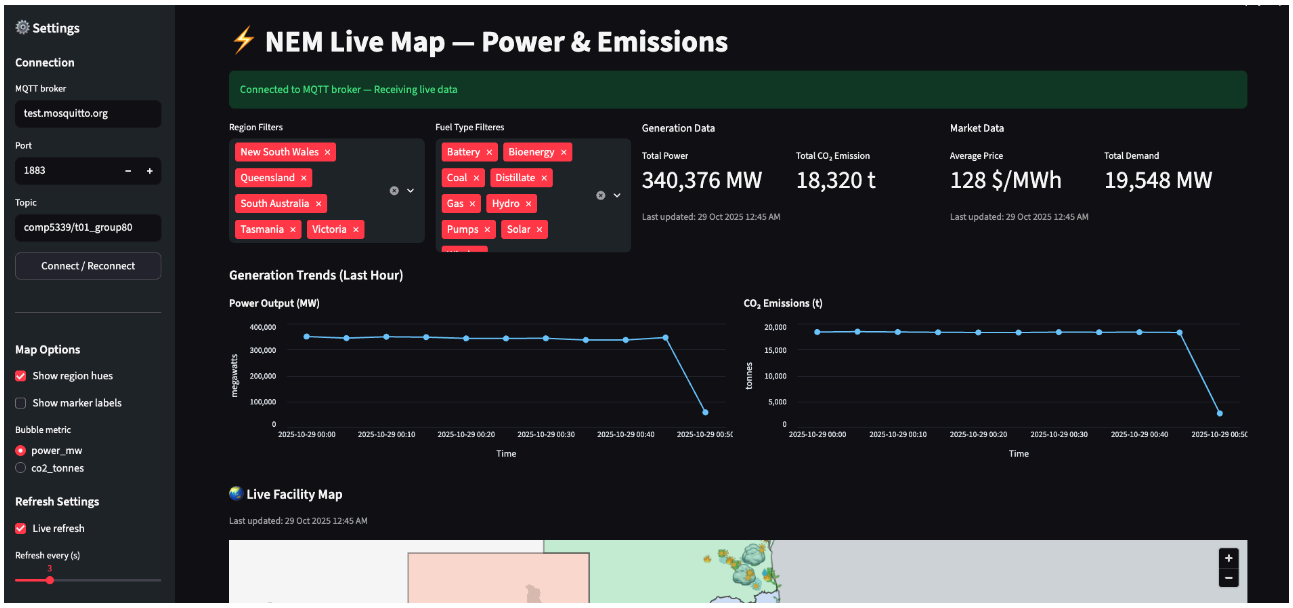
We built a streamlit dashboard application which is multi-threaded. Upon initialization, a cached application state instance is created to serve as a central data repository, fetching static lookup tables for facilities and regions from PostgreSQL's ***oem*** schema. For the sake of submission, we simulate the query by pre-loading the lookup tables from ***context.py***. When the user clicks "Connect/Reconnect" in the sidebar, the app spawns a MQTT client, connects to the public Eclipse Mosquitto broker and subscribes to our configured topic with QoS 1, while running in a background thread

Each message is differentiated to either a facility or market event, and validated with Pydantic data models. Each message is enriched with facility or region metadata from the corresponding lookup tables, using their respective identifiers. If a facility is not found in the OE facility lookup table, the app queries PostgreSQL's ***nger*** schema from Assignment 1 to check whether the facility's information is available. If the data exists, a new record is inserted into the facility lookup table. Finally, under a thread lock to ensure concurrency safety, the message is used to update the shared state dictionaries that map facilities and regions to their latest readings, as well as the bounded deques that maintain rolling historical windows of recent data.

When live refresh is enabled, Streamlit automatically reruns at the configured refresh interval (default: 3 seconds). On each refresh, it reads data from the shared state to construct a dataframe, which is then used to compute aggregated metrics for power, emissions, price, and demand, as well as to render the interactive PyDeck map. As shown in Figure 1, users can filter the displayed data by region or fuel type through a multi-select field. Both the map visualization and the aggregated metrics dynamically update to reflect the selected filters. Time series trends for Power output and CO2 emission are generated as line plots using historical data collected in 5-minute intervals.

As facility events stream into the dashboard, the geographic map (Figure 2) dynamically updates to display the latest information. Each icon on the map represents a facility, with a marker showing its name and either its most recent power generation or emission value, selectable via a radio button group. The marker size also adjusts dynamically based on the chosen metric. To reduce visual clutter, users can toggle the display of markers and labels using a checkbox on the sidebar. The regions are segregated using geojson data [5], improving visual clarity with a legend that explains what each icon, hue and size represents..

Users can hover over the facility icon, highlighting and showing a pop-up in the form of a tooltip showing the facility's name, fuel types, and most recent power and emission data, as illustrated in Figure 2. A tooltip format was chosen for the pop-up due to constraints in the PyDeck library. A table of the latest subscribed events is also displayed (switchable between facility and market data) as shown in Figure 3 of the Annex. Meanwhile, the MQTT thread continues running asynchronously in the background, updating the shared application state to ensure that each map refresh reflects the latest available data.



⚙️ Settings

Connection

MQTT broker: test.mosquitto.org

Port: 1883

Topic: comp5339/t01_group80

Connect / Reconnect

Map Options

- ✓ Show region hues
- ☐ Show marker labels

Bubble metric: ● power_mw ○ co2_tonnes

Refresh Settings

- ✓ Live refresh

Refresh every (s): 3

⚡ NEM Live Map — Power & Emissions

Connected to MQTT broker — Receiving live data

Region Filters: New South Wales × Queensland × South Australia × Tasmania × Victoria ×

Fuel Type Filteres: Battery × Bioenergy × Coal × Distillate × Gas × Hydro × Pumps × Solar ×

Generation Data

Total Power: 340,376 MW

Total $CO_2$ Emission: 18,320 t

Last updated: 29 Oct 2025 12:45 AM

Market Data

Average Price: 128 $/MWh

Total Demand: 19,548 MW

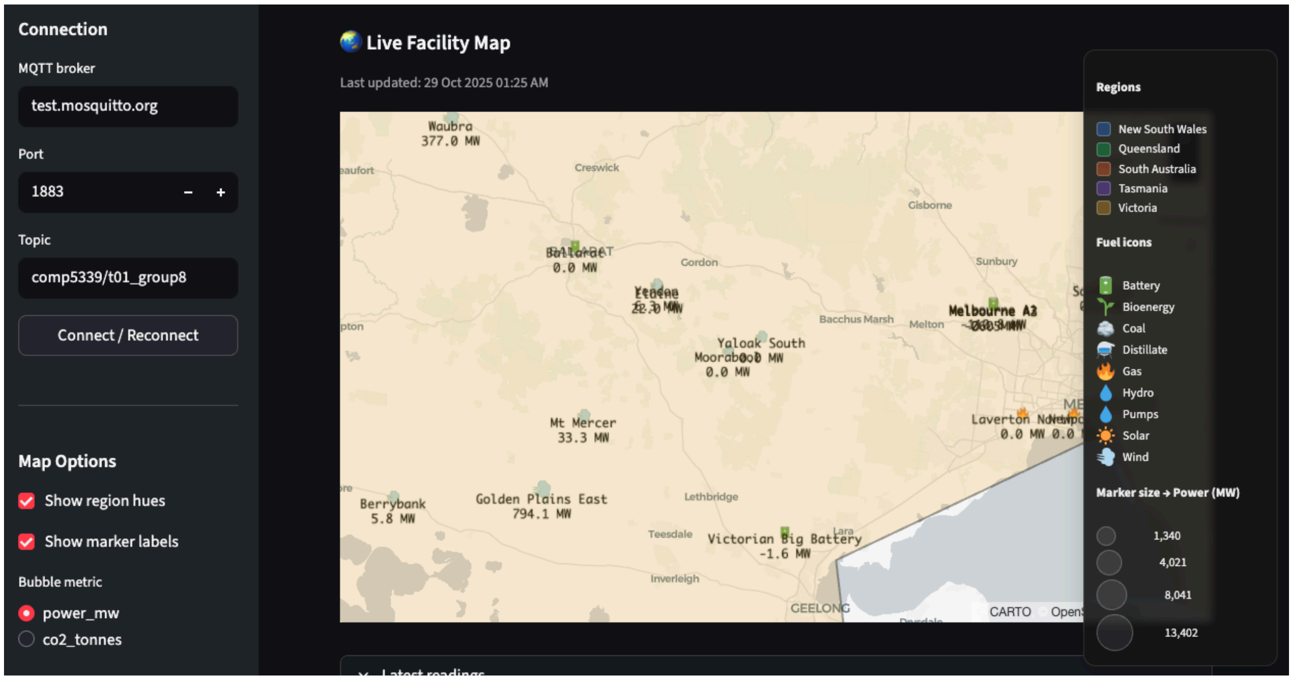
Last updated: 29 Oct 2025 12:45 AM

Generation Trends (Last Hour)

🌏 Live Facility Map

Last updated: 29 Oct 2025 12:45 AM

**Figure 1: Dashboard Filters, Metrics and Trend**



Connection

MQTT broker: test.mosquitto.org

Port: 1883

Topic: comp5339/t01_group8

Connect / Reconnect

Map Options

- ✓ Show region hues
- ✓ Show marker labels

Bubble metric: ● power_mw ○ co2_tonnes

🌏 Live Facility Map

Last updated: 29 Oct 2025 01:25 AM

Regions

- 🟦 New South Wales
- 🟩 Queensland
- 🟫 South Australia
- 🟪 Tasmania
- 🟨 Victoria

Fuel icons

- 🔋 Battery
- 🌱 Bioenergy
- ☁️ Coal
- ☁️ Distillate
- 🔥 Gas
- 💧 Hydro
- 💧 Pumps
- ☀️ Solar
- 💨 Wind

Marker size → Power (MW)

- ● 1,340
- ● 4,021
- ● 8,041
- ● 13,402

Latest readings

**Figure 2: Dashboard Live Facility Map**

## Optimization and Efficiency

**Data Extraction**
While extracting price and demand data from OE, we observed their API server can be unstable at times where the same request may succeed or fail. In response, we optimized our request function with a retry mechanism of 10  seconds delay and retry limit of 3. We extracted the price and demand at region level, instead of market level, so that the dashboard can achieve a region level filter for the displayed price and demand metric.

**Data Integration**
Transforming the facility and market data into pivot tables and merging them together was an optimization to reduce data redundancy. Each row now represents a unique timestamp, while each column shows either a facility's power or emission value, or a region's price or demand, corresponding to that specific time. Each ISO-format timestamp occupies 27 bytes, and each facility code averages 6 bytes. Over a one-week period with 2016 five-minute intervals, this corresponds to roughly 24MB or redundant data that is avoided by not repeatedly storing these values for every record. As a result, the final CSV file is only 8MB, representing a 75% reduction in data size and a reduction in memory requirements at the event publishing service.

**Data Publishing**
Storing the facility level facilities data in PostgreSQL has enabled us to keep the event message, to be published, lightweight. The event message only needs to comprise of an identifier and the variable data respective of the identifier. For power and emission, the identifier would be facility id, while for price and demand, the identifier would be region id. A lightweight event message has a smaller payload size, which means less data needs to be transmitted across the network. This results in faster transfer times, reduced chances of retransmission in poor network conditions, and lower overall bandwidth usage. In turn, it improves efficiency and responsiveness, especially in environments with limited network capacity or when data must be relayed from MQTT publisher client to subscriber clients, through the broker.

**Data Visualization**
When the dashboard application starts up, it queries the PostgreSQL database once, to retrieve the facility and region information table (***facility_lookup & region_lookup***) and stores it in memory. This is an optimization because it avoids the need to repeatedly query the database for the same static facility information every time a new event payload arrives. By keeping the data in memory, the application can perform instant lookups instead of waiting for database round trips, which reduces latency, data load, network traffic and makes the dashboard more responsive. To ensure the integrity of the subscribed events, we use Pydantic models to validate the incoming data's structure and types before processing. This guarantees that each event conforms to the expected schema, preventing malformed or inconsistent data from entering the system.

## Contributions

Jonas designed the overall system and its components, implementing the complete data pipeline from acquisition to visualization. He developed the data acquisition, cleaning, and integration processes in extractor.py, and implemented both the publishing (publisher.py) and subscribing (on_message in dashboard.py) mechanisms. He also refactored dashboard.py to remove redundant code and integrate the subscribed event payloads into the dashboard's application state. In addition, Jonas authored the System Description, Data Retrieval and Integration, Data Publishing and Continuous Execution, Optimization and Efficiency, and the subscriber section of the Data Subscribing & Dynamic Visualization parts of the report.

Alok built the frontend Streamlit dashboard and worked on theData Subscribing & Dynamic Visualization parts of the report .

## Reference List

1. OpenElectricity, “Get Facilities.” [Online]. Available: https://docs.openelectricity.org.au/api-reference/facilities/get-facilities

2. OpenElectricity, “Fueltechs and Fueltech Groups.” [Online]. Available: https://docs.openelectricity.org.au/guides/fueltechs

3. OpenElectricity, “Get Facility Data.” [Online]. Available: https://docs.openelectricity.org.au/api-reference/data/get-facility-data

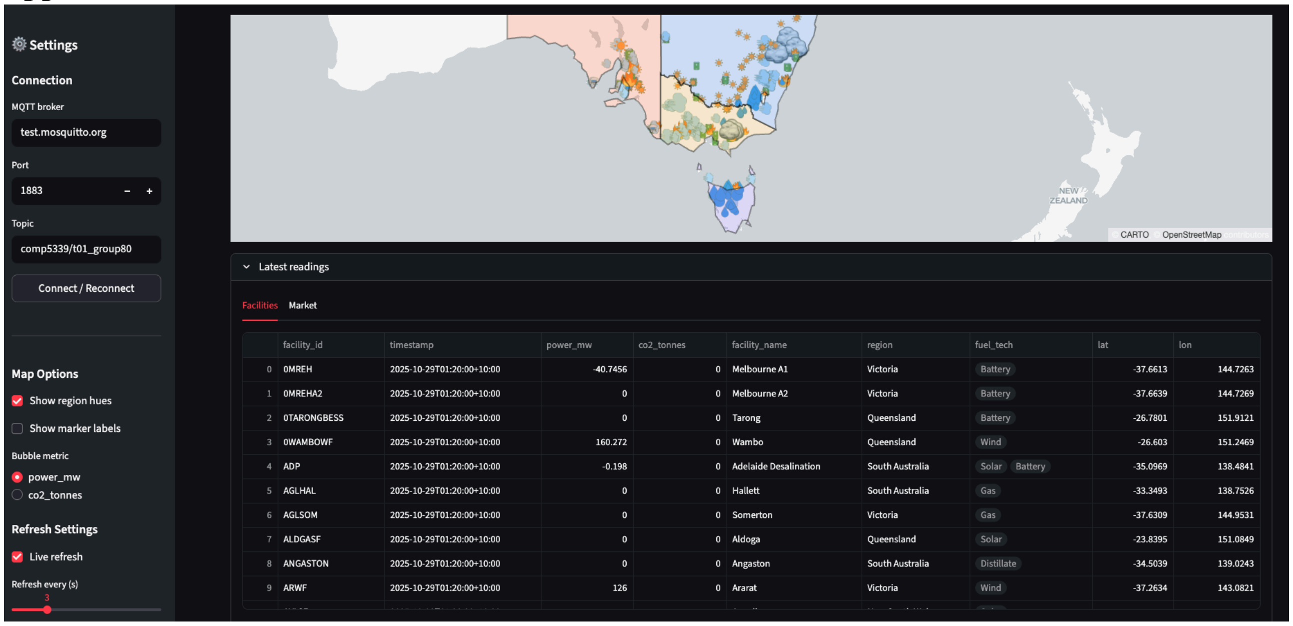
4. OpenElectricity, “Get Network Data.” [Online]. Available: https://docs.openelectricity.org.au/api-reference/market/get-network-data

5.tonywr71, “GeoJson-Data: Australian states.” GitHub. [Online]. Available: https://github.com/tonywr71/GeoJson-Data/blob/master/australian-states.json

## Acknowledgements

AI tools such as ChatGPT have been used to help improve fluency of certain sections in this report so as to achieve better clarity. We also used ChatGPT to generate an initial code skeleton for the streamlit dashboard before modifying it to suit our specific use case. Copilot was used to clarify certain python syntax structures.

## Appendix



⚙ Settings

Connection

MQTT broker: test.mosquitto.org

Port: 1883

Topic: comp5339/t01_group80

Connect / Reconnect

Map Options

[✓] Show region hues

[ ] Show marker labels

Bubble metric: [✓ power_mw] co2_tonnes

Refresh Settings

[✓] Live refresh

Refresh every (s): 3

Latest readings

Facilities | Market

| | facility_id | timestamp | power_mw | co2_tonnes | facility_name | region | fuel_tech | lat | lon |
|---|---|---|---|---|---|---|---|---|---|
| 0 | 0MREH | 2025-10-29T01:20:00+10:00 | -40.7456 | 0 | Melbourne A1 | Victoria | Battery | -37.6613 | 144.7263 |
| 1 | 0MREHA2 | 2025-10-29T01:20:00+10:00 | 0 | 0 | Melbourne A2 | Victoria | Battery | -37.6639 | 144.7269 |
| 2 | 0TARONGBESS | 2025-10-29T01:20:00+10:00 | 0 | 0 | Tarong | Queensland | Battery | -26.7801 | 151.9121 |
| 3 | 0WAMBOWF | 2025-10-29T01:20:00+10:00 | 160.272 | 0 | Wambo | Queensland | Wind | -26.603 | 151.2469 |
| 4 | ADP | 2025-10-29T01:20:00+10:00 | -0.198 | 0 | Adelaide Desalination | South Australia | Solar Battery | -35.0969 | 138.4841 |
| 5 | AGLHAL | 2025-10-29T01:20:00+10:00 | 0 | 0 | Hallett | South Australia | Gas | -33.3493 | 138.7526 |
| 6 | AGLSOM | 2025-10-29T01:20:00+10:00 | 0 | 0 | Somerton | Victoria | Gas | -37.6309 | 144.9531 |
| 7 | ALDGASF | 2025-10-29T01:20:00+10:00 | 0 | 0 | Aldoga | Queensland | Solar | -23.8395 | 151.0849 |
| 8 | ANGASTON | 2025-10-29T01:20:00+10:00 | 0 | 0 | Angaston | South Australia | Distillate | -34.5039 | 139.0243 |
| 9 | ARWF | 2025-10-29T01:20:00+10:00 | 126 | 0 | Ararat | Victoria | Wind | -37.2634 | 143.0821 |

**Figure 3: Subscribed Events**

### Guide to run assignment 2 project

Please set up your python virtual environment using venv or an isolated environment using miniconda.
Our assignment 2 project uses python 3.12.12. For more information, refer to the requirements.txt

*Terminal 1*
~~~
``` bash
cd project_directory
# activate python environment
python extractor.py
~~~

*Terminal 2*
~~~
``` bash
cd project_directory
# activate python environment
streamlit run dashboard.py
~~~

*Terminal 3*
~~~
``` bash
cd project_directory
# activate python environment
python publisher.py
~~~

``` # Click on the CONNECT button ```
```
```