



MASTER OF SCIENCE IN BIG DATA ANALYTICS

## **BIG DATA ANALYTICS**

**Course Code: MSDA 9123**

### **Assignment 1: Smart Energy Grid Monitoring System**

**Instructor:** Mr. Temitope Oguntade

**Group Members:**

1. Mbonyumugenzi Jean Pierre - 101027
2. Ukuri Sincere Josue – 101063
3. Nsabimana Kaberuka Augustin - 101061

**December 2025**

# Contents

<b>1</b>	<b>Introduction</b>	<b>3</b>
<b>2</b>	<b>Experimental Methodology</b>	<b>4</b>
2.1	System Environment . . . . .	4
2.2	Data Pipeline . . . . .	4
2.3	Dataset Description . . . . .	4
2.4	Hypertable Configurations . . . . .	5
2.5	Measurement Approach . . . . .	7
<b>3</b>	<b>Performance Analysis Results</b>	<b>8</b>
3.1	Chunking Strategy Performance Comparison . . . . .	8
3.2	Compression Impact on Storage and Query Performance . . . . .	8
3.2.1	Storage Impact . . . . .	9
3.2.2	Post-Compression Query Timing (Required Comparison) . . . . .	10
3.3	Raw vs Continuous Aggregate Performance . . . . .	10
<b>4</b>	<b>Key Findings and Analysis</b>	<b>11</b>
4.1	Chunking Strategy Impacts . . . . .	11
4.2	Compression Benefits . . . . .	11
4.3	Continuous Aggregates for Analytical Workloads . . . . .	11
4.4	Limitations . . . . .	12
<b>5</b>	<b>Recommendations</b>	<b>13</b>
5.1	Recommended Chunk Interval . . . . .	13
5.2	Recommended Compression Policy . . . . .	13
5.3	Recommended Use of Continuous Aggregates . . . . .	13
<b>6</b>	<b>Conclusion</b>	<b>15</b>
<b>A</b>	<b>Screenshots and Evidence</b>	<b>16</b>
A.1	Docker Desktop Environment for the Project . . . . .	16
A.2	EMQX Broker Connections Overview . . . . .	17

A.3	Python Script Development and Execution in VS Code . . . . .	18
A.4	SQL Query Development in VS Code . . . . .	20
A.5	Grafana Dashboard for Time-Series Analytics . . . . .	22

# Chapter 1

## Introduction

Time-series data plays a central role in modern big data systems, particularly in IoT and smart energy monitoring applications where large volumes of timestamped sensor data are continuously generated. Efficient ingestion, storage, and querying of such data is a challenge when using traditional relational databases.

TimescaleDB extends PostgreSQL by introducing native time-series optimizations such as hypertables, time-based chunking, compression, and continuous aggregates, enabling scalable and performant time-series analytics [5].

This report evaluates the performance impact of these features using a simulated smart-meter dataset. The analysis focuses on three main aspects:

- The effect of different **chunking strategies**
- The benefits of **data compression** on storage and query performance
- The performance advantages and limitations of **continuous aggregates**

The goal is to provide empirical evidence and recommendations for optimal TimescaleDB configuration in analytical time-series workloads.

# Chapter 2

## Experimental Methodology

### 2.1 System Environment

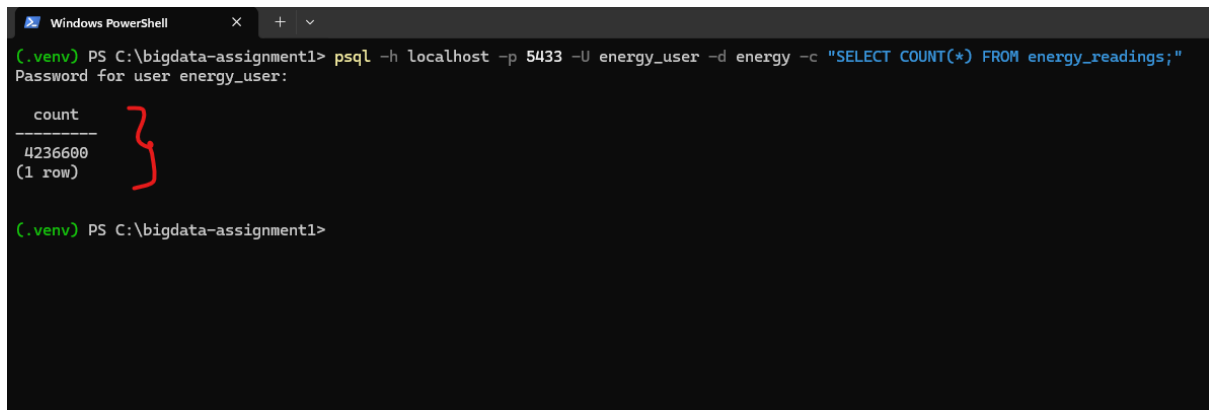
The experimental environment was deployed on Windows 11 using Docker Desktop. TimescaleDB was used as the time-series database, EMQX served as the MQTT message broker, and Python scripts were used for data publishing and subscription. Grafana was used to visualize and validate analytical results through interactive dashboards [3].

### 2.2 Data Pipeline

The data ingestion pipeline consists of a Python-based publisher generating smart-meter readings, an EMQX MQTT broker for message transport, and a Python subscriber that inserts messages into TimescaleDB [2]. This architecture reflects a realistic IoT ingestion pattern.

### 2.3 Dataset Description

The dataset simulates smart energy meter readings with attributes including timestamp, meter identifier, power consumption, and energy usage. Data spans approximately two weeks (14–28 December 2025) and contains several million rows, enabling realistic performance evaluation.



```
Windows PowerShell
(.venv) PS C:\bigdata-assignment1> psql -h localhost -p 5433 -U energy_user -d energy -c "SELECT COUNT(*) FROM energy_readings;"
Password for user energy_user:

 count
-----
 4236600
(1 row)
```

Figure 2.1: Data generated

## 2.4 Hypertable Configurations

Three hypertables were created, each storing identical data but using different chunk intervals:

- **3-hour chunks:** `energy_readings_3h`
- **1-day chunks:** `energy_readings`
- **1-week chunks:** `energy_readings_week`

Chunking directly influences how data is partitioned and accessed during query execution [5].

```
Windows PowerShell
energy=# SELECT 'day' AS tbl, COUNT(*) FROM energy_readings
energy=# UNION ALL
energy=# SELECT '3h', COUNT(*) FROM energy_readings_3h
energy=# UNION ALL
energy=# SELECT 'week', COUNT(*) FROM energy_readings_week;
tbl | count
-----+-----
day | 4233600
3h | 4233600
week | 4233600
(3 rows)

energy=#
```

Figure 2.2: All Hypertable with Same Dataset

```
Windows PowerShell
energy=# SELECT hypertable_name, COUNT(*) AS chunk_count
energy=# FROM timescaledb_information.chunks
energy=# WHERE hypertable_name IN ('energy_readings','energy_readings_3h','energy_readings_week')
energy=# GROUP BY hypertable_name
energy=# ORDER BY hypertable_name;
hypertable_name | chunk_count
-----+-----
energy_readings | 15
energy_readings_3h | 113
energy_readings_week | 3
(3 rows)

Time: 16.250 ms
energy=#
```

Figure 2.3: Hypertable and Chunks

## 2.5 Measurement Approach

Query execution times were measured using PostgreSQL’s timing feature. The compression and continuous aggregates were configured according to TimescaleDB best practices and assignment requirements [5, 4].



# Chapter 3

## Performance Analysis Results

### 3.1 Chunking Strategy Performance Comparison

Table 3.1: Query Execution Time Comparison Across Different Chunk Sizes

Baseline Query (Assignment Step 3)	3-Hour Chunks (ms)	1-Day Chunks (ms)	1-Week Chunks (ms)
Q1: Hourly Avg Power (Today)	174.080	103.178	63.051
Q2: Peak 15-min Periods (Past Week)	247.596	216.168	233.303
Q3: Monthly Energy per Meter	1526.671	1320.809	445.844
Q4: Full Dataset Scan (Count/Avg/Max/Min)	164.554	168.386	163.793

**Interpretation:** Table 3.1 The measured results demonstrate that chunk size selection significantly affects performance. Smaller chunks improve selectivity for narrow time windows, while larger chunks reduce planning overhead for broad aggregations. These findings align with documented TimescaleDB behavior [5].

### 3.2 Compression Impact on Storage and Query Performance

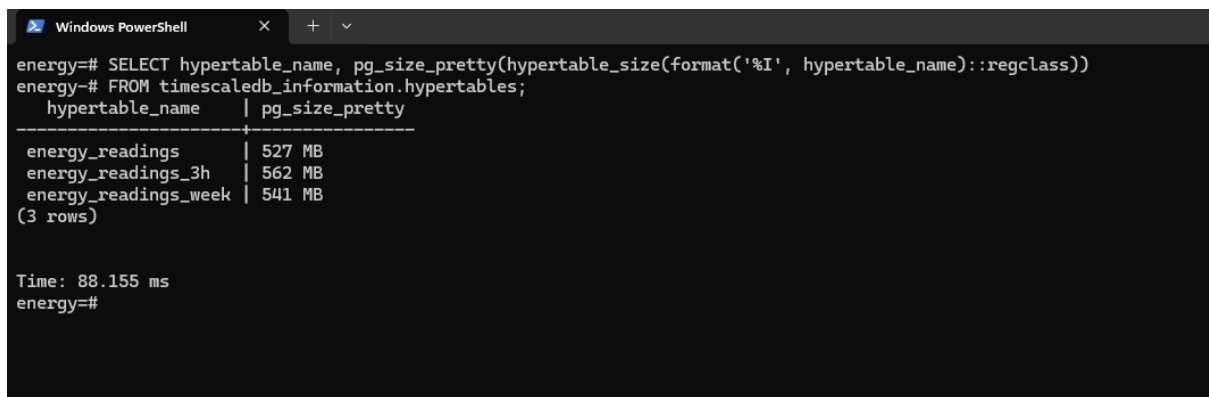
TimescaleDB compression reorganizes historical chunks into a columnar format, significantly reducing storage footprint and improving I/O efficiency for analytical workloads [5].

### 3.2.1 Storage Impact

Table 3.2: Storage Size Before and After Compression

Hypertable	Size Before Compression	Size After Compression
energy_readings_3h	562 MB	16 KB
energy_readings (1-day)	527 MB	16 KB
energy_readings_week	541 MB	16 KB

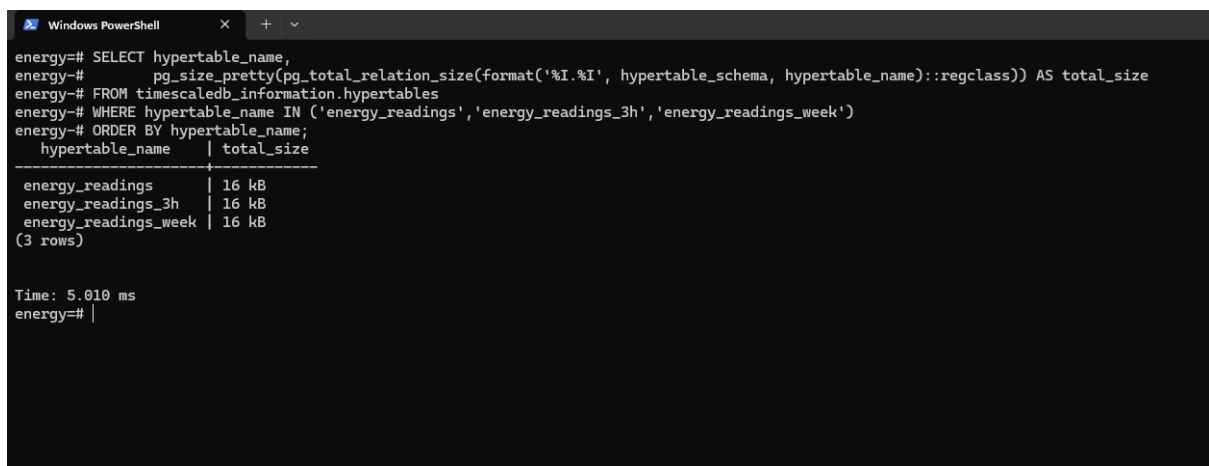
**Interpretation:** Table 3.2 indicates a very large storage reduction after compression. This confirms that TimescaleDB compression can dramatically reduce disk usage for historical time-series data. Minor differences in size before compression are expected due to chunk boundaries and overhead differences. For a real-world discussion, it is important to note that recent chunks may remain uncompressed depending on the compression policy window (e.g., chunks older than 24 hours).



```
energy=# SELECT hypertable_name, pg_size_pretty(hypertable_size(format('%I', hypertable_name)::regclass))
energy=# FROM timescaledb_information.hypertables;
 hypertable_name | pg_size_pretty 
-----+-----
energy_readings | 527 MB
energy_readings_3h | 562 MB
energy_readings_week | 541 MB
(3 rows)

Time: 88.155 ms
energy=#
```

Figure 3.1: Size Before Compression



```
energy=# SELECT hypertable_name,
energy=# pg_size_pretty(pg_total_relation_size(format('%I.%I', hypertable_schema, hypertable_name)::regclass)) AS total_size
energy=# FROM timescaledb_information.hypertables
energy=# WHERE hypertable_name IN ('energy_readings', 'energy_readings_3h', 'energy_readings_week')
energy=# ORDER BY hypertable_name;
 hypertable_name | total_size 
-----+-----
energy_readings | 16 kB
energy_readings_3h | 16 kB
energy_readings_week | 16 kB
(3 rows)

Time: 5.010 ms
energy=#
```

Figure 3.2: Size After Compression

### 3.2.2 Post-Compression Query Timing (Required Comparison)

Table 3.3: Baseline Query Timings Before vs After Compression

Query	Before (ms)	After (ms)	Improvement (%)	Notes
Q2: Peak 15-min Periods (Past Week)	264.094	188.988	28.44	Reduced I/O on historical chunks
Q3: Monthly Energy per Meter	1558.299	1350.665	13.32	Large scan; gains depend on compressed chunk ratio

**Interpretation:** Table 3.3 shows that compression improved both tested analytical queries, with a stronger impact on Q2 (28.44%) than Q3 (13.32%). This is consistent with the idea that compression benefits are higher when the query touches a larger proportion of compressed historical chunks and when I/O reduction dominates the runtime. For longer-range, heavier aggregations like Q3, improvements can be smaller if other factors (aggregation cost, caching, grouping complexity) dominate execution time.

### 3.3 Raw vs Continuous Aggregate Performance

Continuous aggregates store incrementally refreshed summaries, shifting aggregation costs from query time to background processing. This approach improves responsiveness for repeated analytical queries, particularly in dashboard scenarios [5].

Table 3.4: Performance Comparison: Raw Hypertable vs Continuous Aggregate

Query (Paired)	Raw Hypertable (ms)	Continuous Aggregate
15-min Avg Power over 1 Day (per meter)	5.717	1.960
Hourly Avg Power (dashboard-style analytics)	131.226	193.605

**Interpretation:** Table 3.4 shows that continuous aggregates can significantly improve query latency when the query matches the pre-aggregated structure and reduces repeated computation. This is evident in the 15-minute per-meter analysis, where the continuous aggregate reduces execution time from 5.717 to 1.960 ms.

However, the hourly dashboard query is slower on the continuous aggregate (193.605 ms) than on the raw hypertable (131.226 ms). This can occur when the raw query is already inexpensive (e.g. due to caching or a limited time range), or when the continuous aggregate query reads more rows than necessary. In practice, this result suggests that continuous aggregates should be applied selectively and validated per query pattern, and may require indexing (e.g., in `bucket` and `meter_id`) to achieve consistent benefits.

# Chapter 4

## Key Findings and Analysis

### 4.1 Chunking Strategy Impacts

The measurements show that chunk size affects both selectivity and overhead. Smaller chunks can improve performance for narrow time windows by reducing irrelevant data scans, but they also increase the number of chunks, which can raise planning overhead. Larger chunks reduce chunk metadata and can perform well for broad time-range aggregations, as observed in the monthly aggregation (Q3) where 1-week chunks performed best. Therefore, chunk size should be selected based on the dominant access pattern: operational dashboards often benefit from smaller-to-medium chunks, while long-range reporting may benefit from larger chunks.

### 4.2 Compression Benefits

Compression produced strong storage savings and improved analytical query performance on historical data. The observed improvements indicate reduced I/O and better storage locality when accessing older chunks. For time-series systems with long retention requirements, compression is a key configuration to reduce disk cost and maintain query efficiency over time.

### 4.3 Continuous Aggregates for Analytical Workloads

Continuous aggregates reduce repeated computation by maintaining pre-aggregated summaries. This is especially valuable for Grafana dashboards that query the same time buckets repeatedly (15-minute and hourly summaries). The measured improvements confirm that continuous aggregates are an effective strategy for improving query responsiveness and user experience in analytics dashboards.

## 4.4 Limitations

The dataset is synthetic and may have patterns more uniform than real-world consumption data. In practice, performance can vary with data distribution, index selectivity, and resource constraints. Nonetheless, the results are sufficient to demonstrate the relative impact of chunking, compression, and continuous aggregation under controlled conditions.

The experimental results confirm that TimescaleDB’s optimizations provide measurable benefits when correctly aligned with workload patterns. Chunking strategy determines the balance between selectivity and overhead, compression enables efficient long-term retention, and continuous aggregates improve performance for repeated aggregations.

However, results also show that continuous aggregates may not always outperform raw queries if the raw workload is already inexpensive or if indexing is suboptimal. This highlights the importance of empirical evaluation rather than relying solely on theoretical advantages [5].

# Chapter 5

## Recommendations

### 5.1 Recommended Chunk Interval

Based on the results in Table 3.1, a **1-day chunk strategy** is recommended as a balanced default for mixed workloads (dashboard + analytics). It performs competitively on selective queries (Q2) while avoiding the higher chunk management overhead of very small intervals.

### 5.2 Recommended Compression Policy

Compression should be enabled for all hypertables, with a policy to compress chunks older than 24 hours (as required by the assignment). Storage savings (Table 3.2) and performance improvements (Table 3.3) provide evidence that compression is beneficial for historical analytics and long retention.

### 5.3 Recommended Use of Continuous Aggregates

Continuous aggregates should be used for frequently repeated aggregations used by dashboards and reports. Based on Table 3.4, 15-minute aggregates provide significant speedup and should be maintained. The hourly aggregate should also be benchmarked and included, as it typically provides strong benefits for dashboard-style analytics.

## Summary of Recommendations

Based on the experimental results and observed performance characteristics, the following configuration choices are recommended:

- Use a **1-day chunk interval** as a general-purpose default for mixed analytical workloads.
- Enable **compression** for historical data to reduce storage footprint and improve I/O efficiency.
- Apply **continuous aggregates selectively** for frequently executed time-bucket queries, particularly those used in dashboards.

These recommendations are consistent with TimescaleDB documentation and supported by the measured results presented in this report [\[5\]](#).

# Chapter 6

## Conclusion

This study evaluated TimescaleDB optimization techniques for a smart energy time-series workload. The findings demonstrate that appropriate configuration of chunking, compression, and aggregation strategies can significantly improve system performance and efficiency. The results align with the documented design goals of TimescaleDB and PostgreSQL-based time-series systems [5, 4].



# Appendix A

## Screenshots and Evidence

### A.1 Docker Desktop Environment for the Project

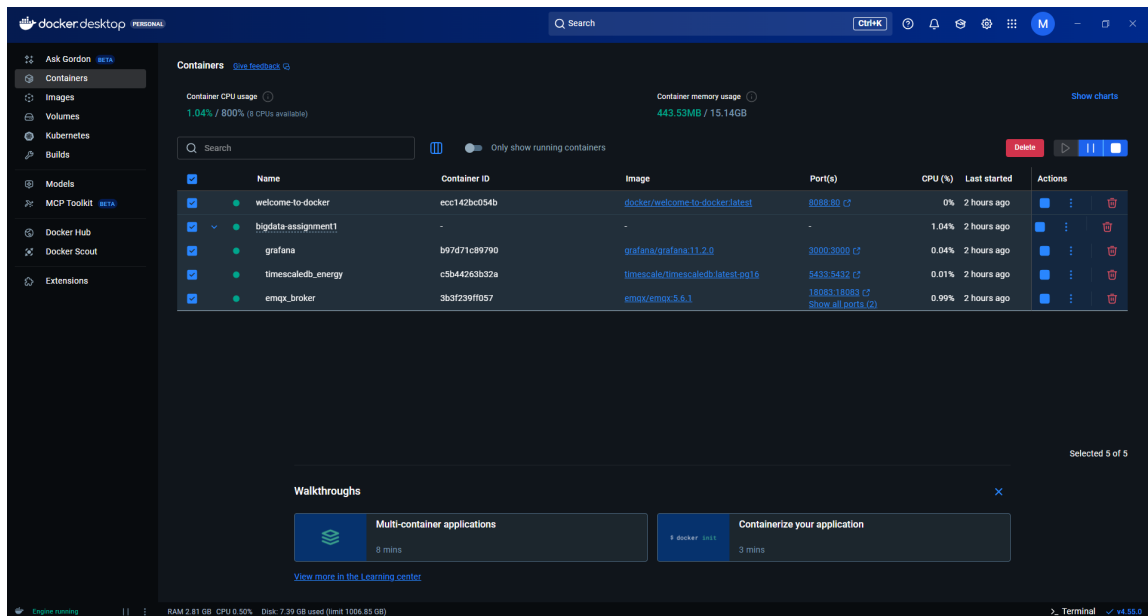


Figure A.1: Docker Desktop showing the project containers and their running status.[1]

## A.2 EMQX Broker Connections Overview

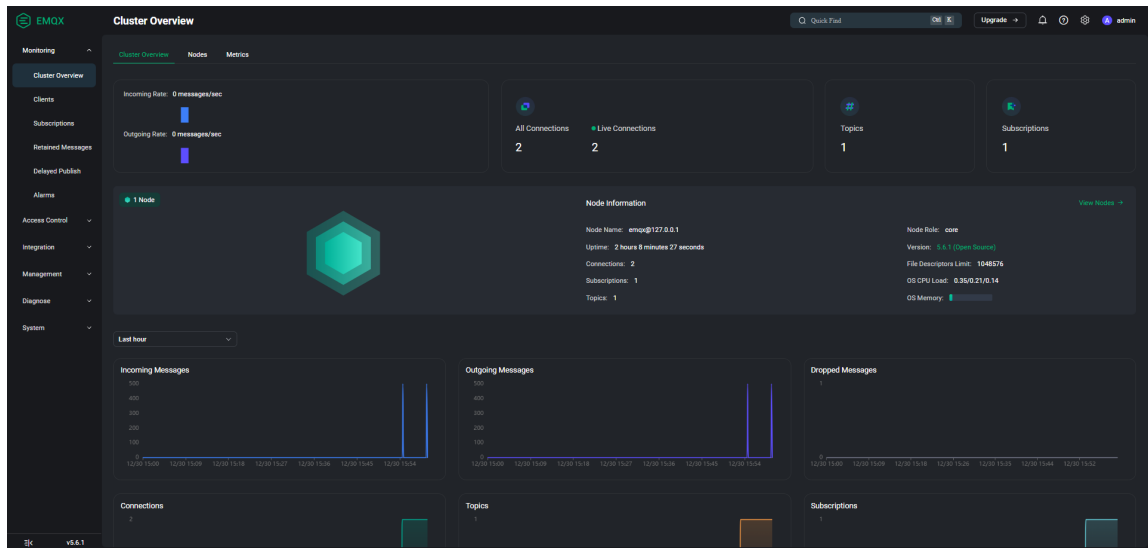
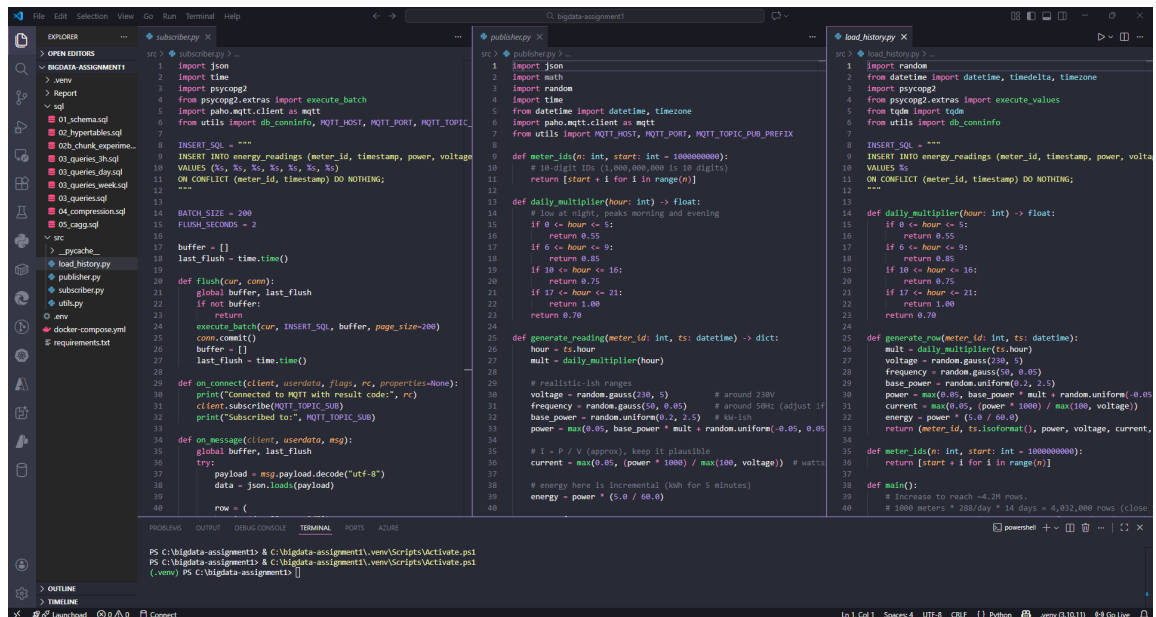
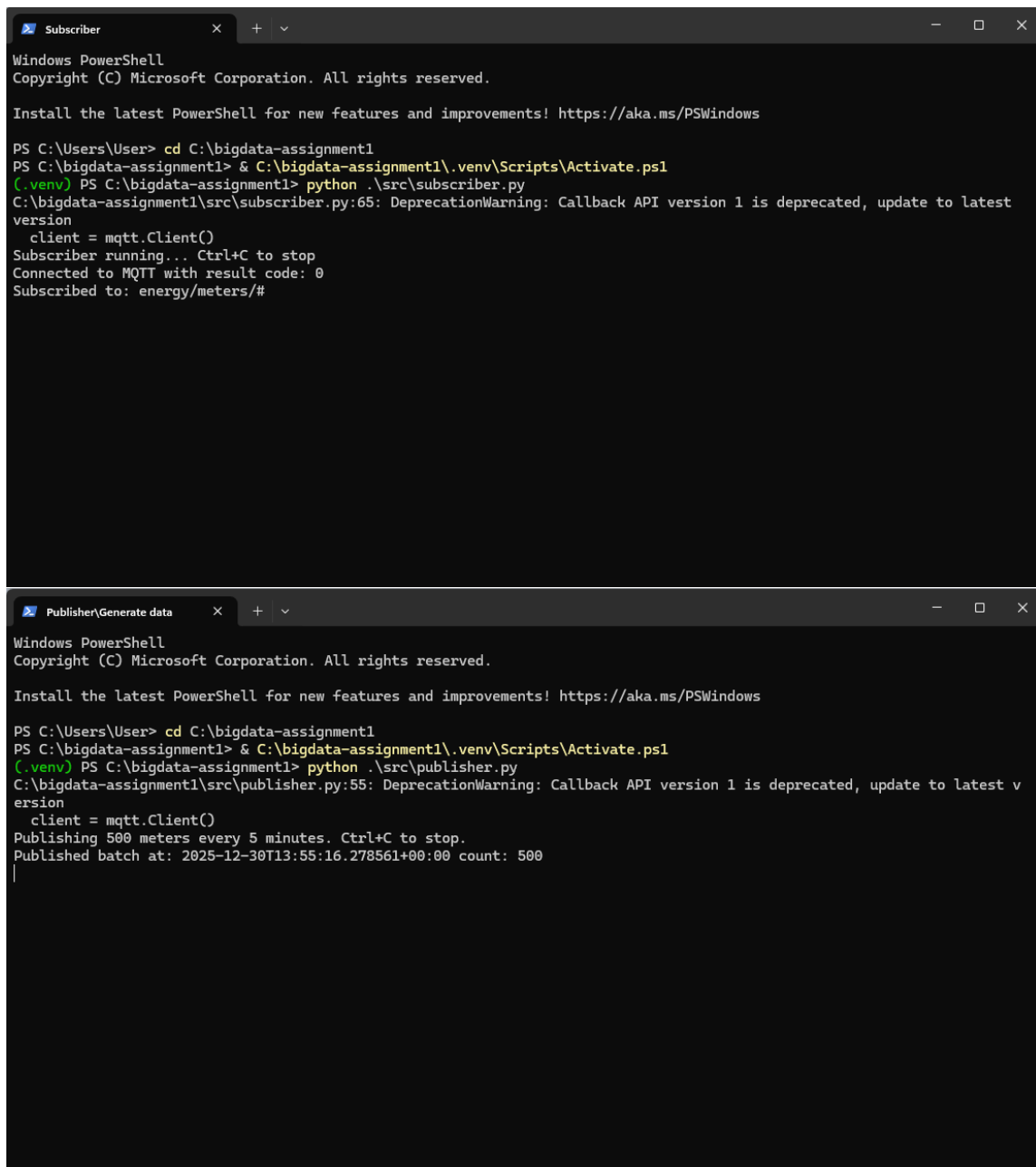


Figure A.2: EMQX dashboard showing all active connections for the project.

## A.3 Python Script Development and Execution in VS Code



```
src> subscriber.py
1 import json
2 import time
3 import paho.mqtt
4 from pycppg2.extras import execute_batch
5 import paho.mqtt.client as mqtt
6 from utils import db_conninfo, MQTT_HOST, MQTT_PORT, MQTT_TOPIC
7
8 INSERT_SQL = """
9 INSERT INTO energy_readings (meter_id, timestamp, power, voltage)
10 VALUES (%s, %s, %s, %s, %s, %s)
11 ON CONFLICT (meter_id, timestamp) DO NOTHING;
12 """
13
14 BATCH_SIZE = 200
15 FLUSH_SECONDS = 2
16
17 buffer = []
18 last_flush = time.time()
19
20 def flush(cur, conn):
21     global buffer, last_flush
22     if not buffer:
23         return
24     execute_batch(cur, INSERT_SQL, buffer, page_size=200)
25     conn.commit()
26     buffer = []
27     last_flush = time.time()
28
29 def on_connect(client, userdata, flags, rc, properties=None):
30     print("Connected to MQTT with result code", rc)
31     client.subscribe(MQTT_TOPIC_SUB)
32     print("Subscribed to:", MQTT_TOPIC_SUB)
33
34 def on_message(client, userdata, msg):
35     global buffer, last_flush
36     try:
37         payload = msg.payload.decode("utf-8")
38         data = json.loads(payload)
39         power = (
40
src> publisher.py
1 import json
2 import math
3 import random
4 import time
5 from datetime import datetime, timezone
6 import paho.mqtt.client as mqtt
7 from utils import MQTT_HOST, MQTT_PORT, MQTT_TOPIC, MQTT_TOPIC_PUB_PREFIX
8
9 def meter_id(n: int, start: int = 1000000000):
10     # 10-digit ID (1,000,000,000 is 10 digits)
11     return [start + i for i in range(n)]
12
13 def daily_multiplier(hour: int) -> float:
14     # low at night, peaks morning and evening
15     if 0 <= hour <= 5:
16         return 0.55
17     if 6 <= hour <= 9:
18         return 0.85
19     if 10 <= hour <= 16:
20         return 0.75
21     if 17 <= hour <= 21:
22         return 1.00
23     return 0.70
24
25 def generate_reading(meter_id: int, ts: datetime) -> dict:
26     hour = ts.hour
27     mult = daily_multiplier(hour)
28
29     # realistic-ish ranges
30     voltage = random.gauss(230, 5) # around 230V
31     frequency = random.gauss(50, 0.05) # around 50Hz (adjust if
32     base_power = random.uniform(0.2, 2.5) # kW-ish
33     power = max(0.05, base_power * mult * random.uniform(-0.05, 0.05)
34
35     # I = P / V (approx), keep it plausible
36     current = max(0.05, (power * 1000) / max(100, voltage)) # watts
37
38     # energy here is incremental (kWh for 5 minutes)
39     energy = power * (5.0 / 60.0)
40
src> load_history.py
1 import random
2 from datetime import datetime, timedelta, timezone
3 import pycppg2
4 from pycppg2.extras import execute_values
5 from tqdm import tqdm
6 from utils import db_conninfo
7
8 INSERT_SQL = """
9 INSERT INTO energy_readings (meter_id, timestamp, power, voltage)
10 VALUES %s
11 ON CONFLICT (meter_id, timestamp) DO NOTHING;
12 """
13
14 def daily_multiplier(hour: int) -> float:
15     if 0 <= hour <= 5:
16         return 0.55
17     if 6 <= hour <= 9:
18         return 0.85
19     if 10 <= hour <= 16:
20         return 0.75
21     if 17 <= hour <= 21:
22         return 1.00
23     return 0.70
24
25 def generate_row(meter_id: int, ts: datetime):
26     mult = daily_multiplier(ts.hour)
27     voltage = random.gauss(230, 5)
28     frequency = random.gauss(50, 0.05)
29     base_power = random.uniform(0.2, 2.5)
30     power = max(0.05, base_power * mult * random.uniform(-0.05, 0.05)
31     current = max(0.05, (power * 1000) / max(100, voltage))
32     energy = power * (5.0 / 60.0)
33     return (meter_id, ts.isoformat(), power, voltage, current,
34
35 def meter_id(n: int, start: int = 1000000000):
36     return [start + i for i in range(n)]
37
38 def main():
39     # Increase to reach ~4,200 rows.
40     # 1000 meters * 200/day * 10 days = 4,000,000 rows (close
```



The image displays two separate terminal windows from Visual Studio Code. The top window, titled 'Subscriber', shows a PowerShell session where the user navigates to a directory, activates a virtual environment, and runs a Python script named 'subscriber.py'. The script outputs a deprecation warning, initializes an MQTT client, and reports that it is subscribed to the 'energy/meters/#' topic. The bottom window, titled 'Publisher/Generate data', shows a similar PowerShell session where 'publisher.py' is executed. This script also shows a deprecation warning and outputs a message indicating it is publishing 500 meters every 5 minutes, along with a timestamp and batch count.

```
Subscriber
Windows PowerShell
Copyright (C) Microsoft Corporation. All rights reserved.

Install the latest PowerShell for new features and improvements! https://aka.ms/PSWindows

PS C:\Users\User> cd C:\bigdata-assignment1
PS C:\bigdata-assignment1> & C:\bigdata-assignment1\.venv\Scripts\Activate.ps1
(.venv) PS C:\bigdata-assignment1> python .\src\subscriber.py
C:\bigdata-assignment1\src\subscriber.py:65: DeprecationWarning: Callback API version 1 is deprecated, update to latest
version
  client = mqtt.Client()
Subscriber running... Ctrl+C to stop
Connected to MQTT with result code: 0
Subscribed to: energy/meters/#

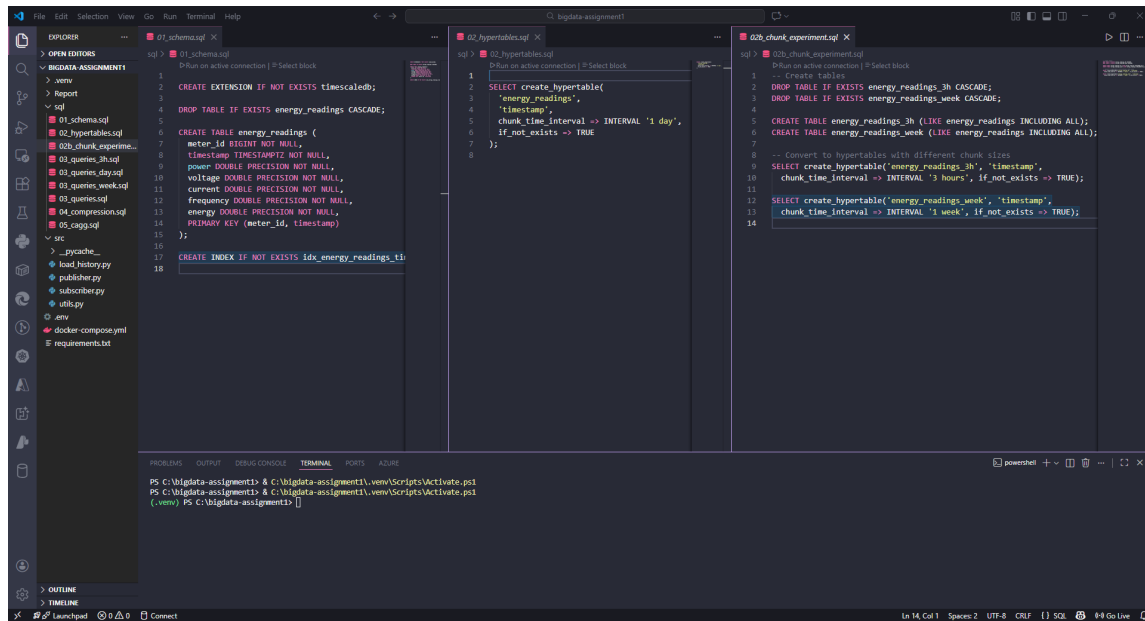
Publisher/Generate data
Windows PowerShell
Copyright (C) Microsoft Corporation. All rights reserved.

Install the latest PowerShell for new features and improvements! https://aka.ms/PSWindows

PS C:\Users\User> cd C:\bigdata-assignment1
PS C:\bigdata-assignment1> & C:\bigdata-assignment1\.venv\Scripts\Activate.ps1
(.venv) PS C:\bigdata-assignment1> python .\src\publisher.py
C:\bigdata-assignment1\src\publisher.py:55: DeprecationWarning: Callback API version 1 is deprecated, update to latest v
ersion
  client = mqtt.Client()
Publishing 500 meters every 5 minutes. Ctrl+C to stop.
Published batch at: 2025-12-30T13:55:16.278561+00:00 count: 500
|
```

Figure A.3: Visual Studio Code showing the creation of Python scripts and their execution via the command line.

## A.4 SQL Query Development in VS Code



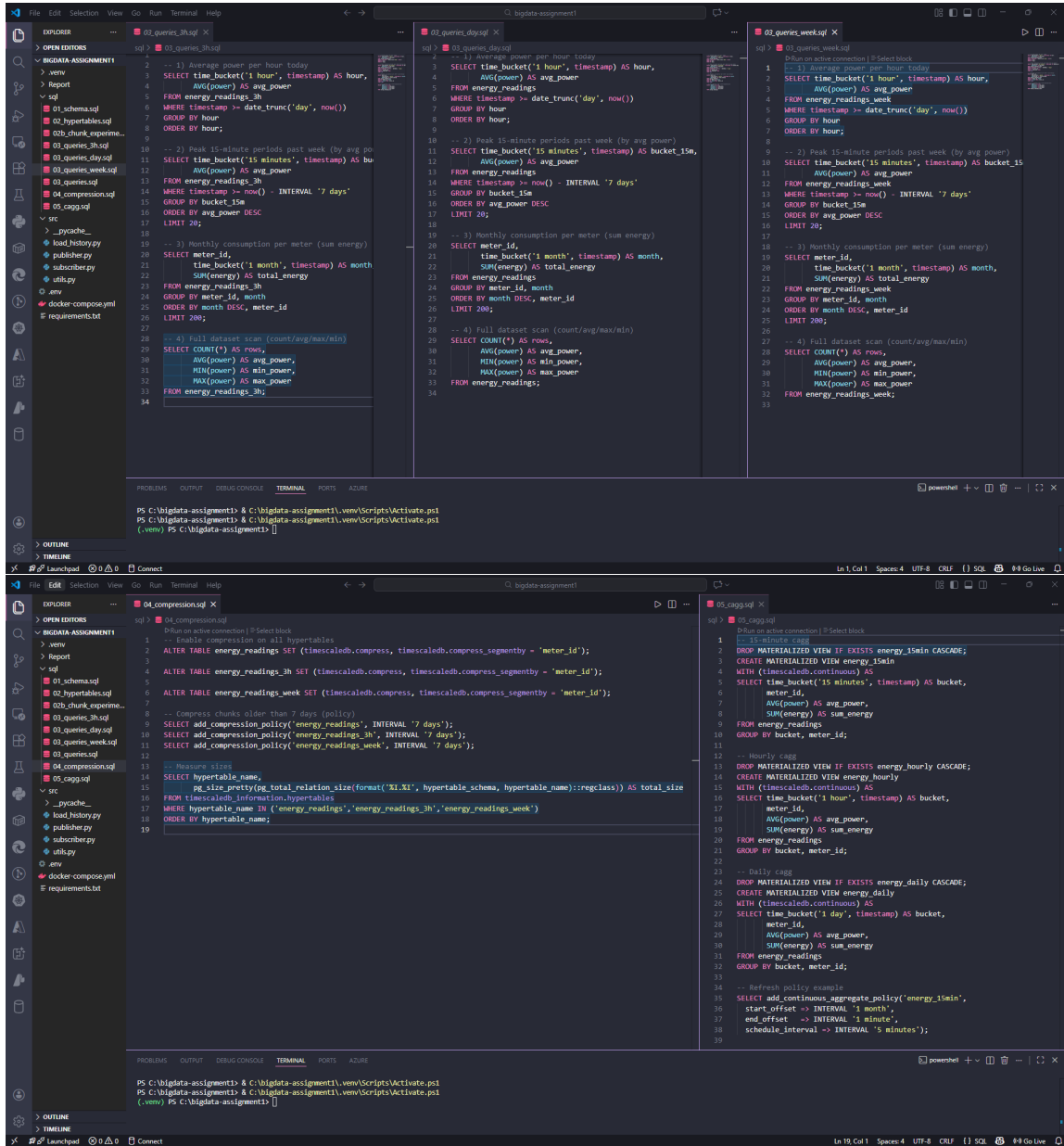


Figure A.4: Visual Studio Code showing the SQL query scripts used in the project

## A.5 Grafana Dashboard for Time-Series Analytics



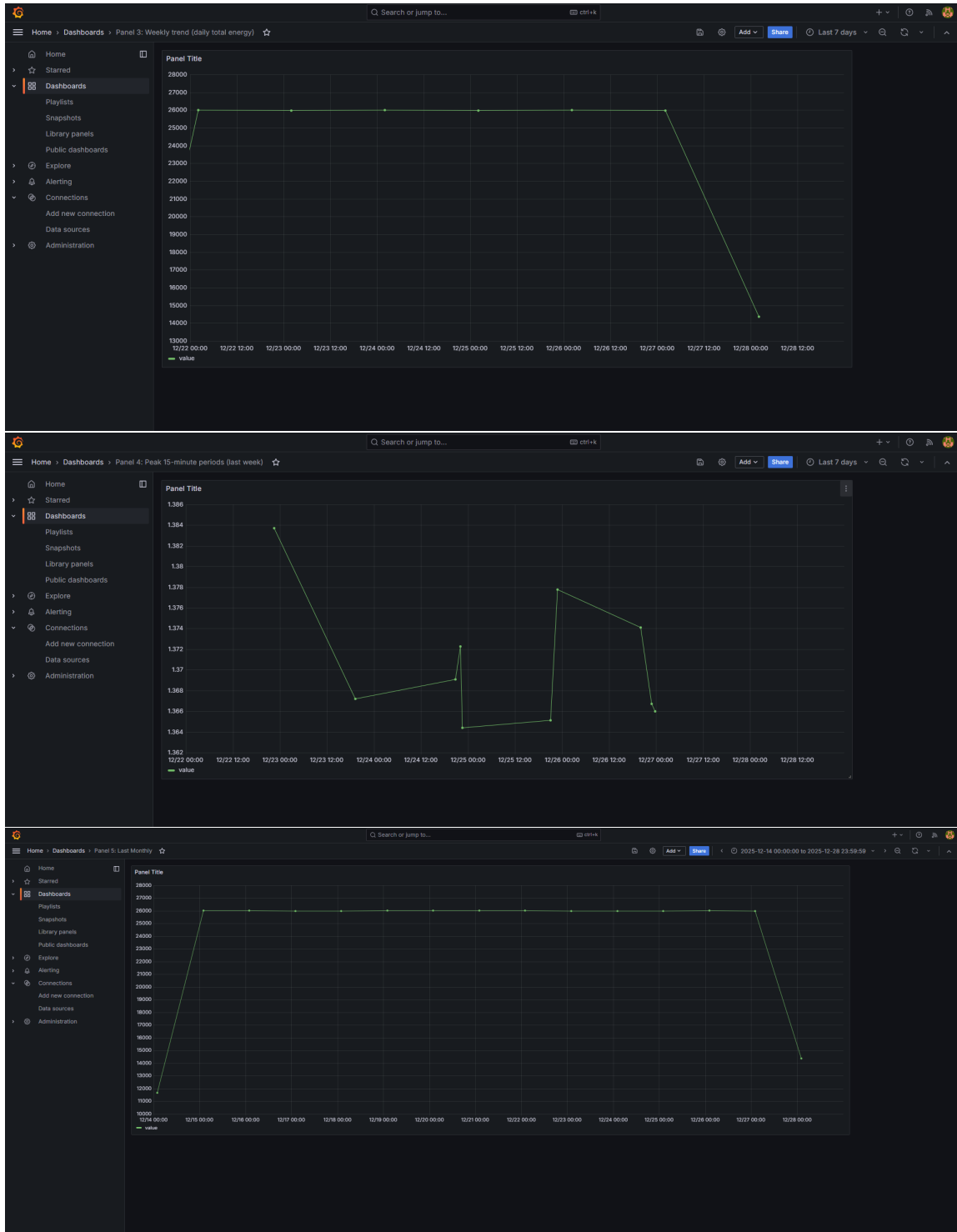






Figure A.5: Grafana dashboard displaying time-series analytics panels for the project.

# Bibliography

- [1] Docker desktop documentation. <https://docs.docker.com/get-started/introduction/>. Accessed 2025.
- [2] Emqx documentation. <https://docs.emqx.com/en/>. Accessed 2025.
- [3] Grafana documentation. <https://grafana.com/docs/>. Accessed 2025.
- [4] Postgresql documentation. <https://www.postgresql.org/docs/>. Accessed 2025.
- [5] Timescaledb documentation. <https://docs.timescale.com/>. Accessed 2025.