

VIET NAM NATIONAL UNIVERSITY HO CHI MINH CITY
HO CHI MINH CITY UNIVERSITY OF TECHNOLOGY
FACULTY OF COMPUTER SCIENCE AND ENGINEERING



SEMESTER 241 - CO3071

DISTRIBUTED SYSTEMS

Developing A Smart Energy Consumption Prediction Pipeline

Instructor: PROF. Thoại Nam

Đỗ Quang Hào	- 2252180
Tạ Gia Khang	- 2152642
Nguyễn Quang Thiện	- 2152994

HO CHI MINH CITY, DECEMBER 2024



Contents

1	Introduction	3
2	Architecture	3
2.1	Apache Kafka	3
2.1.1	Definition	3
2.1.2	Mechanism	3
2.1.3	Highlights	4
2.2	Grafana	4
2.2.1	Definition	4
2.2.2	Operating Mechanism	4
2.2.3	Highlights	6
2.3	Our Baseline	6
3	Configuration	8
3.1	Docker Compose	8
3.2	Kafka Configuration	9
3.3	Database Setup	10
3.3.1	Significance of the Approach	11
3.4	Grafana Setup	12
3.4.1	Creating Dashboards and Visualizations	12
3.4.2	Importing a Dashboard	12
3.4.3	Data Files	14
3.4.4	Alerting in Grafana	14
4	Implementation	15
4.1	Data Production and Consumption in Kafka Topics	15
4.2	Machine Learning-based Prediction and Analytics	17
4.2.1	Data Preparation	17
4.2.2	Model Architecture and Training	17
4.2.3	Model Export and Deployment	18
4.2.4	Significance and Advantages	18
4.2.5	Experimental Results	19
4.3	Data Visualization in Grafana	19
4.3.1	Visualize Graph for each type of Data	19
4.3.2	Some limitation of Grafana	21
5	Conclusion	23

Listings

1	Commands for configuration	8
2	Specific JSON message example	16

1 Introduction

As climate change concerns intensify and regulatory requirements become more stringent, governments must adapt swiftly to remain competitive and sustainable. This paradigm shift necessitates innovative solutions that can process vast amounts of data, derive meaningful insights, and facilitate informed decision-making in real-time.

Our AI-driven Data Management Platform (DMP) revolutionizes sustainable energy practices. This cutting-edge system addresses the critical challenges of sustainable transformation (SX) and digital transformation (DX). Leveraging Kafka Streaming for efficient time-series data management and Spark for advanced predictive modeling, our platform seamlessly integrates with existing infrastructure to drive innovation in the energy sector.

2 Architecture

2.1 Apache Kafka

2.1.1 Definition

Apache Kafka is an open-source event streaming platform designed with a distributed architecture, providing scalability and high performance for processing real-time streaming data. Currently, Kafka is widely used for real-time streaming processing and is gradually replacing traditional message queue systems.

2.1.2 Mechanism

According to the architecture illustrated in the figure below, the Kafka system is designed with the following main components:

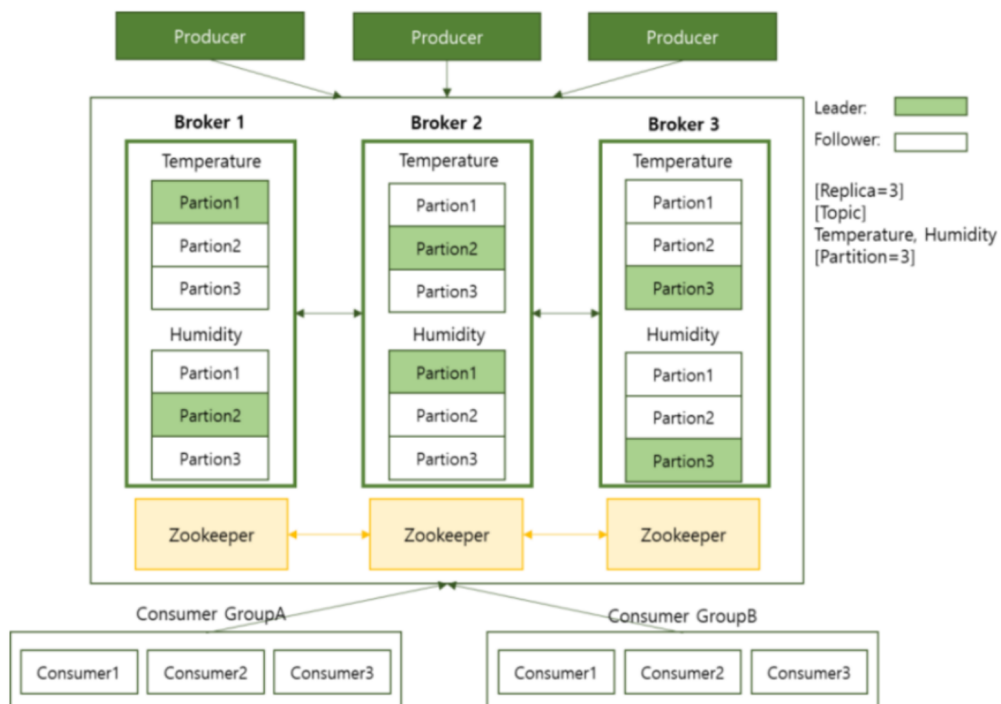


Figure 1: General Kafka Architecture

- **Producer:** A component that generates messages and sends them to corresponding *topics* on the Kafka Broker. The producer is responsible for generating data and pushing it into the Kafka system.

- **Consumer:** A component that subscribes to *topics* on the Kafka Broker to receive messages produced by the Producer. Consumers retrieve data from topics and process it according to system requirements.
- **Consumer Group:** A group of consumers collaboratively consuming messages from the Kafka Broker. Consumers within the same *consumer group* coordinate to process data, ensuring that each message is consumed only once within the group, thus distributing the load and improving processing efficiency.
- **Topic:** A collection of messages with related content, used to categorize and group data in Kafka. Each topic can be divided into multiple *partitions* to enhance storage and parallel processing capabilities.
- **Partition:** Data in a topic is divided into smaller parts called *partitions*. A topic can have one or more partitions, and each partition stores data with a fixed ID called an *offset*. Partitions can be replicated across the Kafka cluster to ensure availability and data safety. Among these replicas, one is designated as the *Leader* to handle read/write requests, while the others serve as *Followers* for redundancy. When the Leader fails, a Follower is automatically promoted to become the Leader.
- **Broker:** A Kafka cluster is a collection of interconnected servers, where each server is referred to as a *Broker*. Brokers are responsible for storing partition data and managing read/write operations for producers and consumers.
- **Zookeeper:** Used to manage and monitor the activity of brokers in Kafka. Zookeeper ensures cluster consistency, assists in distributing tasks among brokers, and helps recover from failures when a broker experiences issues.

This architectural model enables Kafka to handle real-time data with high scalability, ensuring data integrity and reliability during transmission.

2.1.3 Highlights

- Highly scalable due to its distributed cluster and broker mechanism.
- High reliability because stored data is replicated across multiple partitions on different brokers. Additionally, data is stored on disk on each broker and managed via offsets.
- Efficiently handles large volumes of data simultaneously with very high processing speed.
- Well-suited for data streaming processing.

2.2 Grafana

2.2.1 Definition

Grafana is an open-source analytics and interactive visualization web application, primarily designed for time series data visualization and monitoring. It provides a flexible platform for metrics visualization, allowing users to create comprehensive dashboards and alerts for various data sources.

2.2.2 Operating Mechanism

According to the architecture illustrated in the figure below, the Grafana system is designed with the following main components:

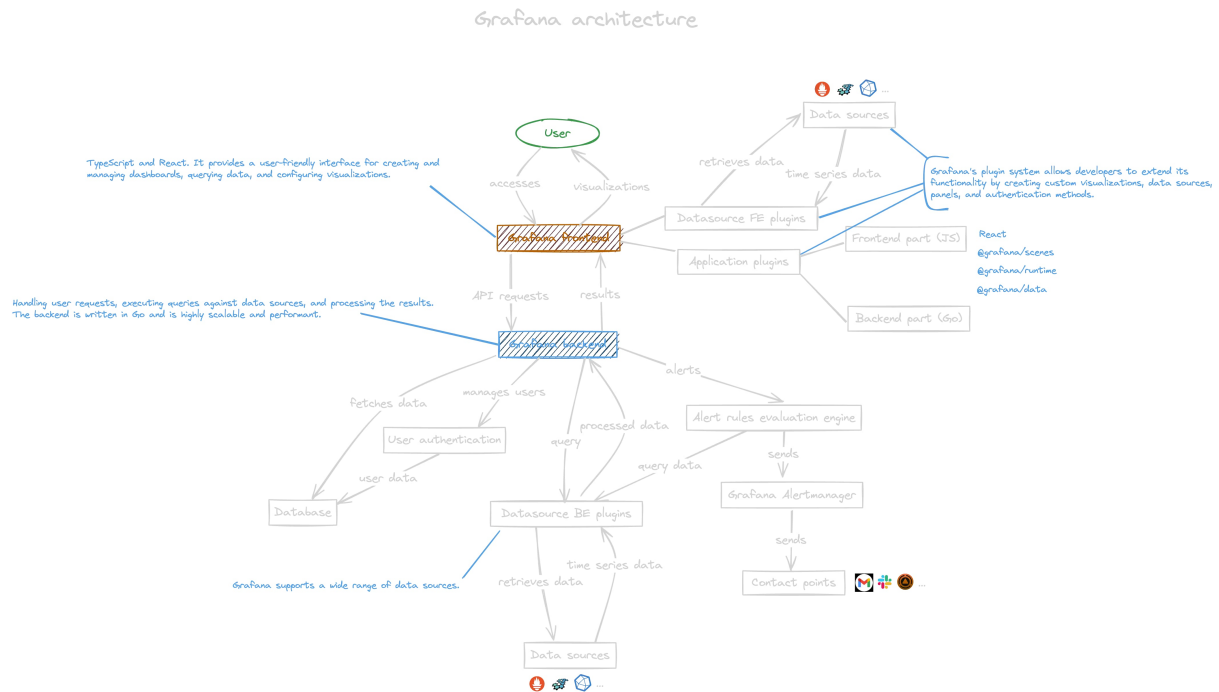


Figure 2: General Grafana Architecture

- **Data Sources:** Components that connect Grafana to various types of databases and services. Grafana supports multiple data sources including Prometheus, Elasticsearch, InfluxDB, MySQL, and many others. Each data source has its specific query language and capabilities.
- **Panels:** The basic visualization units in Grafana. Each panel represents a specific visualization (such as graphs, tables, heatmaps, or alerts) and can query data from any configured data source. Panels are highly customizable with various visualization options and query builders.
- **Dashboards:** Collections of panels arranged in a grid layout. Dashboards serve as the main interface for viewing and analyzing data. They can be organized with folders, shared among team members, and exported/imported across different Grafana instances.
- **Organizations:** Logical groupings that separate users, dashboards, and data sources. Each organization can have its own users, dashboards, and data source configurations, enabling multi-tenancy support.
- **Users and Teams:** Entities that access and manage Grafana resources. Users can be organized into teams, and both users and teams can be assigned specific roles and permissions within organizations.
- **Alerting Engine:** A system that continuously evaluates alert rules based on the data being visualized. When conditions are met, it can trigger notifications through various channels such as email, Slack, or webhook calls.
- **Plugin System:** An extensible architecture that allows for the addition of new data sources, panel types, and applications. Plugins can be officially supported, community-contributed, or enterprise-exclusive.

This architectural model enables Grafana to serve as a comprehensive monitoring and visualization platform, capable of handling diverse data sources and visualization needs.

2.2.3 Highlights

- Highly flexible with support for multiple data sources and query languages, allowing integration with virtually any time-series database or data source.
- Rich visualization capabilities with a wide range of built-in panels and visualization options, complemented by a robust plugin ecosystem.
- Powerful alerting system with support for complex alert rules and multiple notification channels, enabling proactive monitoring and incident response.
- Enterprise-ready features including fine-grained access control, authentication integration, and team-based organization structure.

2.3 Our Baseline

In our architecture, the data source comes from various IoT devices such as temperature and humidity sensors. It gathers real-life environmental data and also some numerical data like renewable energy and lighting usage in previous day, we convert all of it into serial data represented as numeric values.

Our system uses **HiveMQ** as middleware between the data sources and databases, enabling efficient data exchange. The sensor data is managed under corresponding topics in HiveMQ and synchronized to Kafka topics using HiveMQ's Kafka Connector tool and *Pub/Sub* operations. **Kafka** plays a critical role in the system as a robust message queue, enabling both storage and processing of streaming data. While the MQTT Broker serves as the aggregation point for data from a large number of connections via the MQTT protocol, the Kafka Broker is responsible for supporting storage, analysis, and processing of this data. Kafka's features, such as partitioning within topics, message management through offsets, and data replication across partitions on multiple brokers, make it an exceptionally efficient tool for streaming data processing, particularly for real-time processing.

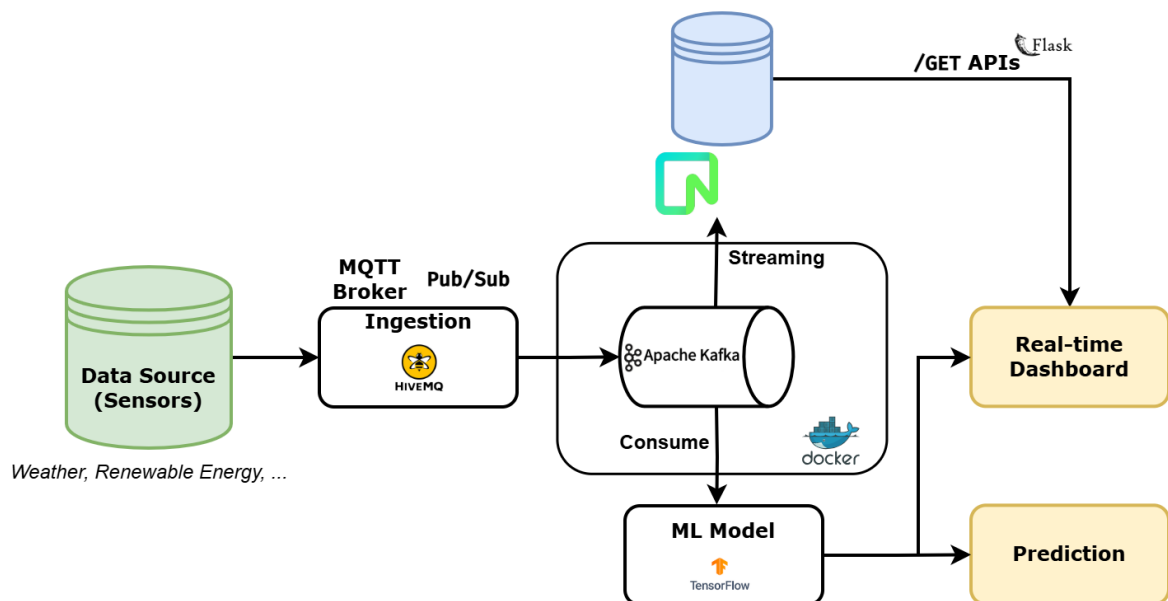


Figure 3: Our project's architecture design

Data from Kafka is then clean and transform before stored in the database. The data retrieved from the databases is accessible via the server-side Backend through the **GET** method call. This data is then forwarded to the Frontend Side, represented by the **Dashboard** tier.

Moreover, we introduce our **AI Module** consume data from the message queue to provide real-time predictions, particularly in forecasting energy consumption in the next day and in the next seven-days. These predictions are sent and displayed on the Dashboard, showing some valuable information about power consumption.

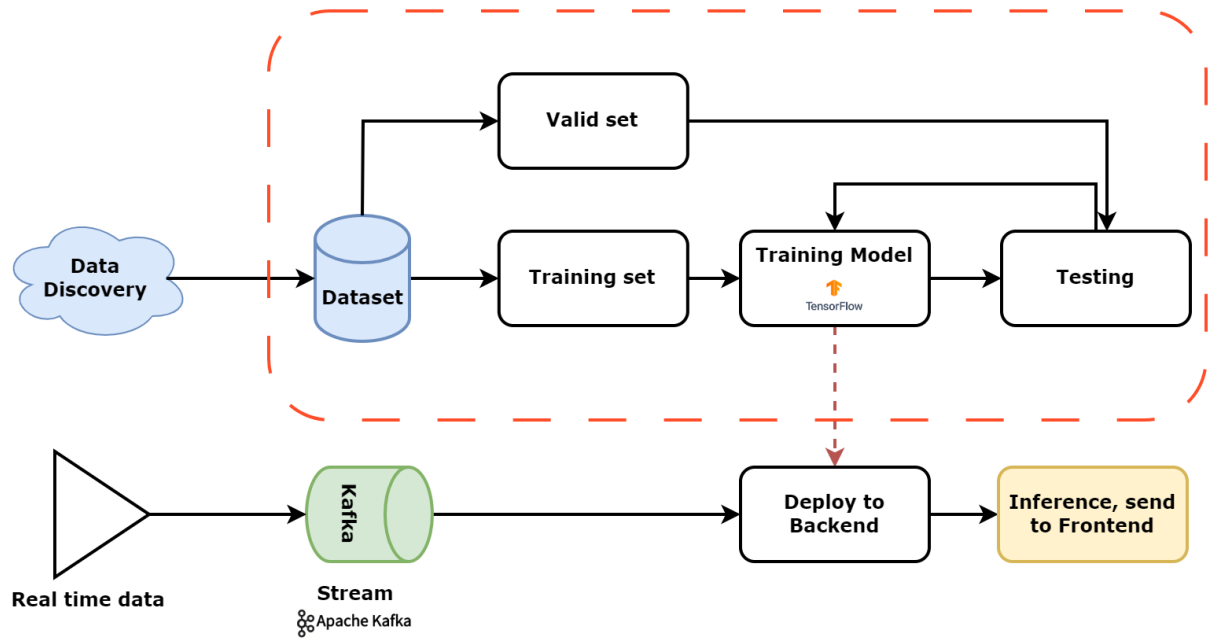


Figure 4: Our design for ML Module

3 Configuration

3.1 Docker Compose

Set up for generating necessary containers: **3 brokers**, **kafka-ui**, and **hivemq**.

```

1 # Initialize network
2 docker network create smart_weather_network
3
4 # Running all service
5 docker-compose up -d --build
6
7 # Create kafka topic
8
9 # Open connector to Kafka
10 docker cp ./kafka-configuration.xml hivemq:/opt/hivemq/extensions/hivemq
    -kafka-extension
11 docker cp ./config.xml hivemq:/opt/hivemq/extensions/hivemq-kafka-
    extension/conf
12
13 docker exec -ti hivemq bash
14
15 # Inside hivemq container
16 cd /opt/hivemq/extensions/hivemq-kafka-extension/
17 cd /opt/hivemq/extensions/hivemq-kafka-extension/conf
18 rm -rf DISABLED
  
```

Listing 1: Commands for configuration

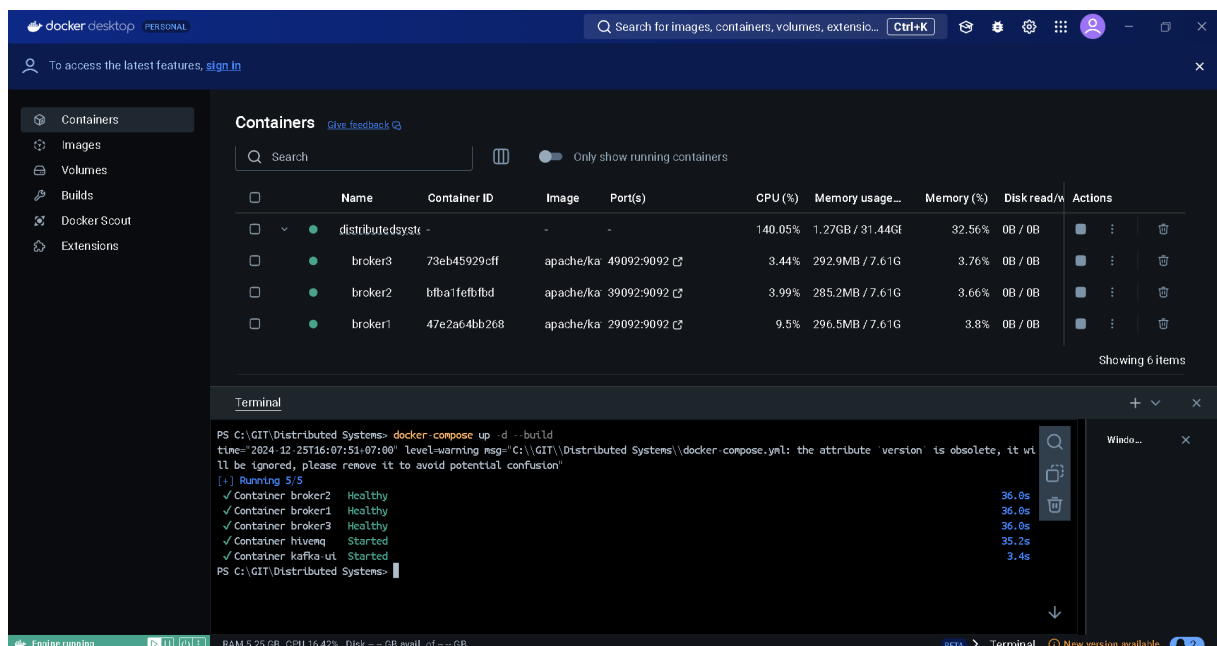


Figure 5: Executing Docker-Compose File (.yaml)

3.2 Kafka Configuration

Initialize Kafka Cluster **Smart_Weather_Network**, which having 3 brokers: **broker1**, **broker2**, **broker3** with ports **29092**, **39092**, **49092**, respectively.

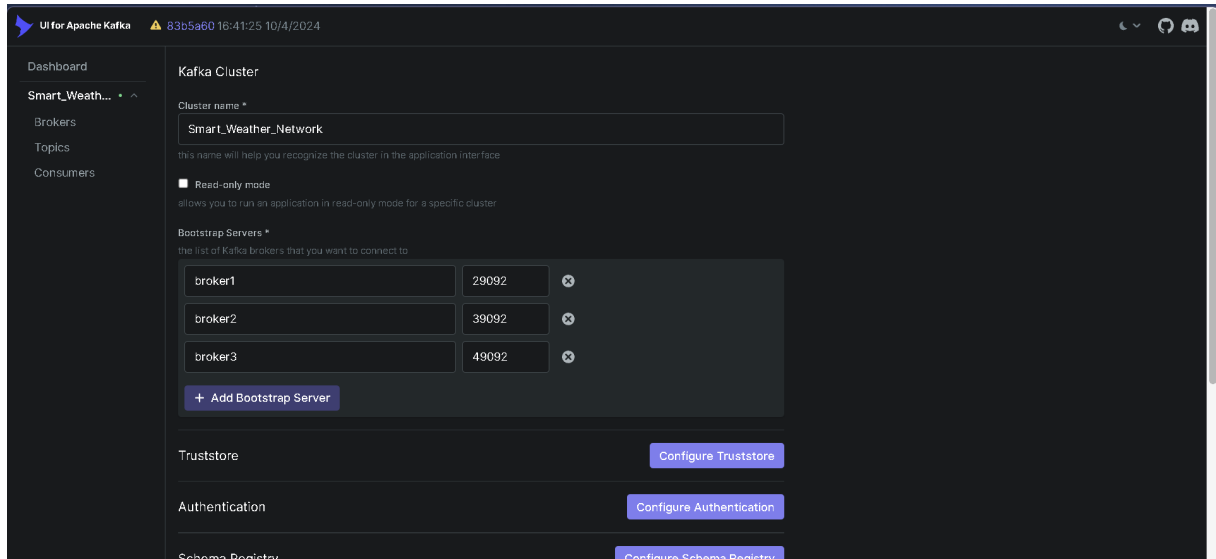


Figure 6: Setting up Kafka Cluster

Then, constructing topics that need storage and analysis with **4 partitions** and **2 replication factors** (for preventing data loss)

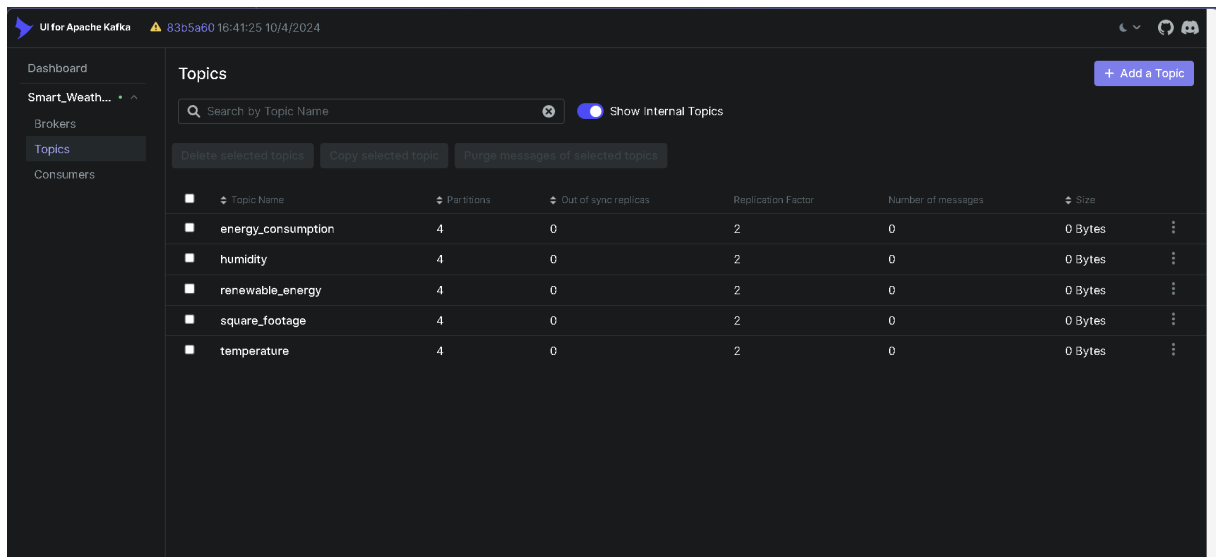


Figure 7: Generating Topics

Furthermore, observing the log below which corresponds to replicas and leader partitions of Kafka topics in the specified broker container, it is obvious that this log in general and all logs of brokers in detail play a vital role in Kafka's replication and fault tolerance mechanisms.

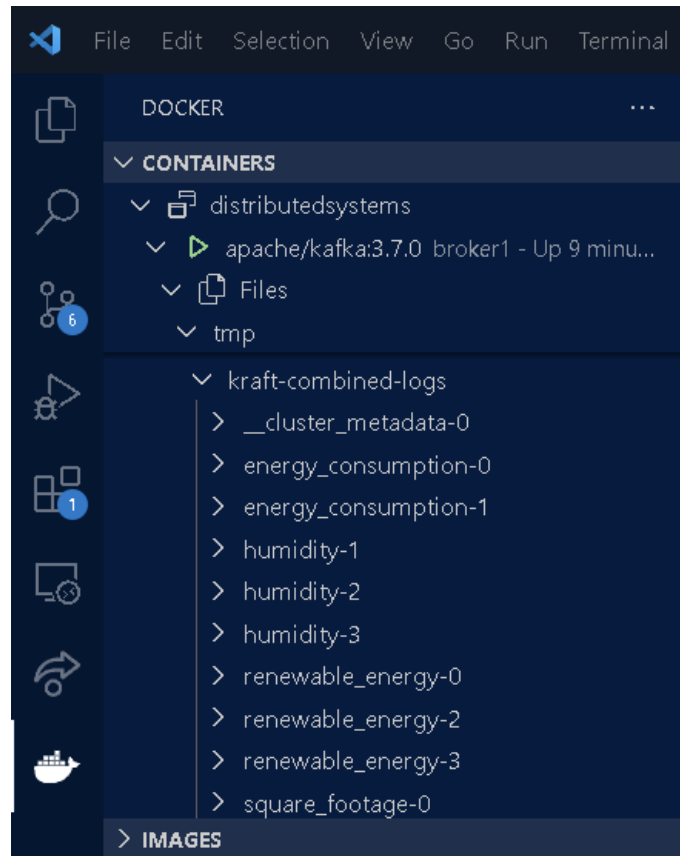


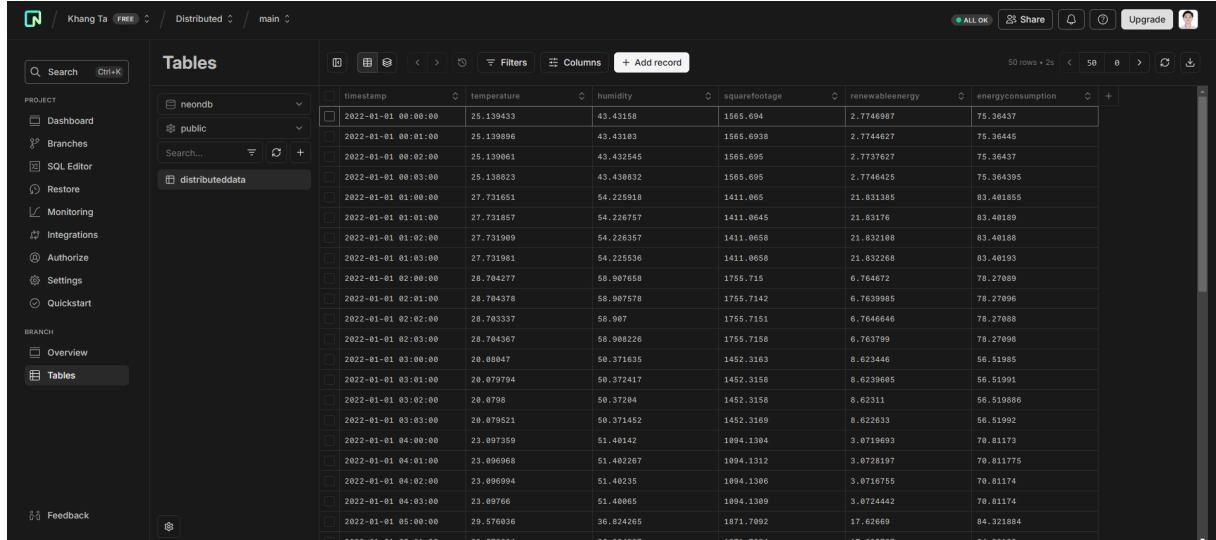
Figure 8: Replicas and Leader Partitions Topics in Broker

3.3 Database Setup

To support data ingestion and preparation for machine learning predictions, a table was designed and deployed in NeonDB. The schema of the table, named `DistributedData`, is defined using the following SQL command:

```
1 CREATE TABLE IF NOT EXISTS DistributedData (
2   Timestamp TIMESTAMP NOT NULL,
3   Temperature REAL NOT NULL,
4   Humidity REAL NOT NULL,
5   SquareFootage REAL NOT NULL,
6   RenewableEnergy REAL NOT NULL,
7   EnergyConsumption REAL NOT NULL,
8   PRIMARY KEY (Timestamp)
9 );
```

This schema is optimized for storing time-series data, with features such as temperature, humidity, square footage, renewable energy contribution, and energy consumption. The `Timestamp` column serves as the primary key, ensuring temporal uniqueness for each record. A visualization of the table as seen in NeonDB is provided in Figure 9.



timestamp	temperature	humidity	squarefootage	renewableenergy	energyconsumption
2022-01-01 00:00:00	25.139433	43.43158	1565.694	2.7746987	75.36437
2022-01-01 00:01:00	25.139896	43.43183	1565.6938	2.7744627	75.36445
2022-01-01 00:02:00	25.139061	43.432545	1565.695	2.7737627	75.36437
2022-01-01 00:03:00	25.138823	43.438832	1565.696	2.7746425	75.364395
2022-01-01 01:00:00	27.731651	54.225918	1411.065	21.831385	83.401855
2022-01-01 01:01:00	27.731857	54.226757	1411.0645	21.83176	83.40189
2022-01-01 01:02:00	27.731909	54.226357	1411.0658	21.832188	83.40188
2022-01-01 01:03:00	27.731081	54.225536	1411.0658	21.832268	83.40193
2022-01-01 02:00:00	28.784277	58.987658	1755.715	6.764872	78.27089
2022-01-01 02:01:00	28.784378	58.987578	1755.7142	6.7639985	78.27096
2022-01-01 02:02:00	28.783337	58.987	1755.7151	6.7646646	78.27088
2022-01-01 02:03:00	28.784367	58.988226	1755.7158	6.763799	78.27098
2022-01-01 03:00:00	28.88847	58.371835	1452.3163	8.623446	56.51985
2022-01-01 03:01:00	28.879794	58.372417	1452.3158	8.6239685	56.51991
2022-01-01 03:02:00	28.8798	58.37284	1452.3158	8.62311	56.519886
2022-01-01 03:03:00	28.879521	58.371452	1452.3169	8.622833	56.51992
2022-01-01 04:00:00	23.897359	51.48142	1894.1384	3.8719693	70.81173
2022-01-01 04:01:00	23.896968	51.482267	1894.1312	3.8728197	70.811775
2022-01-01 04:02:00	23.896994	51.48335	1894.1386	3.8716755	70.81174
2022-01-01 04:03:00	23.89766	51.48865	1894.1389	3.8724442	70.81174
2022-01-01 05:00:00	29.576836	36.824265	1871.7092	17.62669	84.321884

Figure 9: Table structure in NeonDB

To interact with the database, the `psycopg2` library was utilized for establishing connections via a secure database URL (`DB_URL`). For each machine learning prediction, the system retrieves the seven most recent hourly averages of the data. The SQL query used for this purpose is shown below:

```

1 WITH HourlyData AS (
2     SELECT
3         date_trunc('hour', timestamp) AS hour_bucket,
4         AVG(temperature) AS avg_temperature,
5         AVG(humidity) AS avg_humidity,
6         AVG(squarefootage) AS avg_squarefootage,
7         AVG(renewableenergy) AS avg_renewableenergy,
8         AVG(energyconsumption) AS avg_energyconsumption
9     FROM DistributedData
10    GROUP BY hour_bucket
11    ORDER BY hour_bucket DESC
12    LIMIT 7
13 )
14 SELECT *
15 FROM HourlyData
16 ORDER BY hour_bucket ASC;
```

The query employs a common table expression (`WITH` clause) to calculate hourly averages for each feature. The resulting dataset includes the most recent seven hourly intervals, sorted chronologically to align with the requirements of time-series modeling. By aggregating the data, this approach reduces noise and enhances the model's robustness to fluctuations in raw data.

This query is critical for preparing the input data used in subsequent machine learning predictions. It ensures that the model receives a well-structured and temporally consistent dataset, thereby enabling accurate and reliable forecasting.

3.3.1 Significance of the Approach

The database setup and data preparation process are designed to achieve the following:

- **Efficiency:** The use of indexed primary keys and structured queries ensures fast data retrieval, even

for large datasets.

- **Scalability:** The schema and query design support seamless scaling, making it suitable for high-frequency data ingestion and retrieval in real-world applications.
- **Data Integrity:** Timestamp-based uniqueness and hourly aggregation enhance data consistency and reduce redundancy.
- **Model Readiness:** The processed data is aligned with the temporal and statistical requirements of machine learning models, ensuring optimal prediction accuracy.

This streamlined pipeline bridges the gap between raw data collection and predictive modeling, laying a solid foundation for data-driven decision-making in real-time systems.

3.4 Grafana Setup

3.4.1 Creating Dashboards and Visualizations

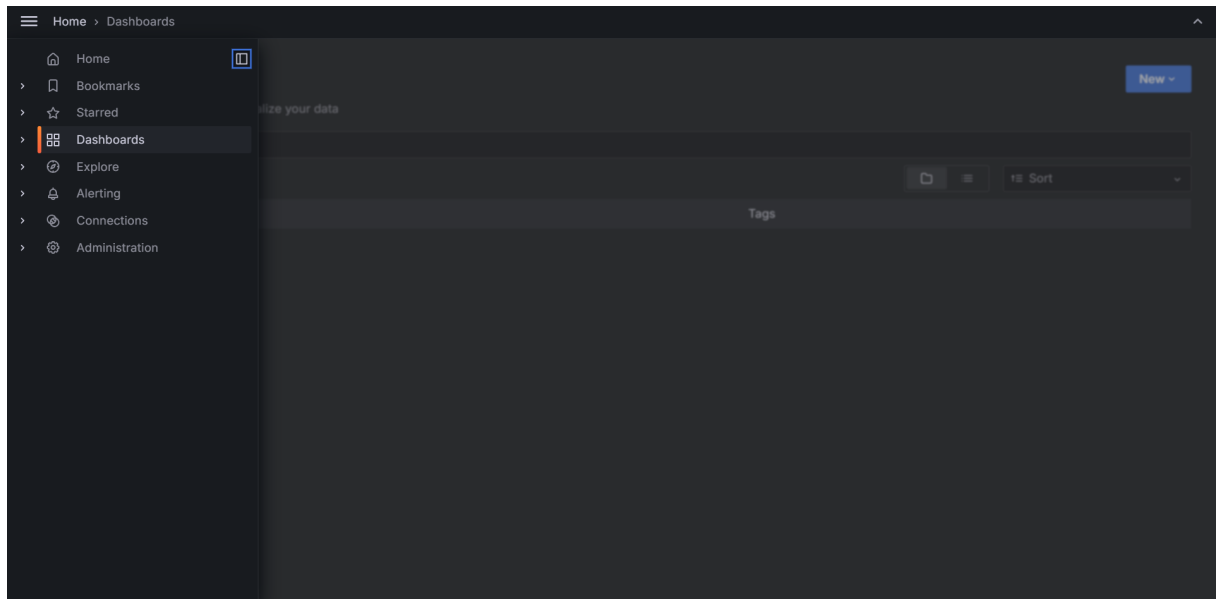
Creating dashboards in Grafana involves the following steps:

1. **Add Panels:** Each panel consists of panels, which are the building blocks to visualize data.
2. **Choose Visualization Types:** Select from various visualization options such as time series graphs, bar charts, pie charts, and single stats.
3. **Query Configuration:** Use Grafana's query editor to fetch data from the connected data sources.
4. **Apply Transformations:** Transformations allow users to modify data directly within Grafana to better suit visualization needs.
5. **Customize Panels:** Use panel settings to adjust appearance, colors, labels, and thresholds.

3.4.2 Importing a Dashboard

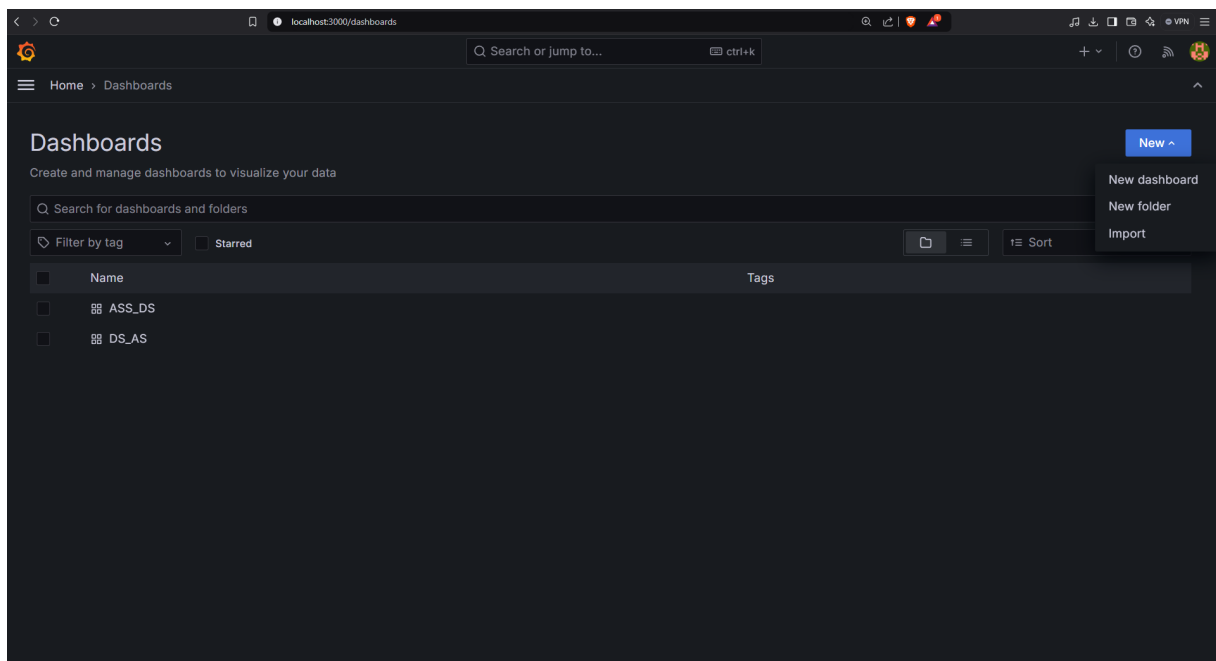
To import a pre-defined dashboard:

1. Ensure that you have the file 'DS_AS-173495038224.json' that contains the dashboard template.
2. Open Grafana in your browser (e.g., <http://localhost:3000>).
3. Navigate to the dashboard page.



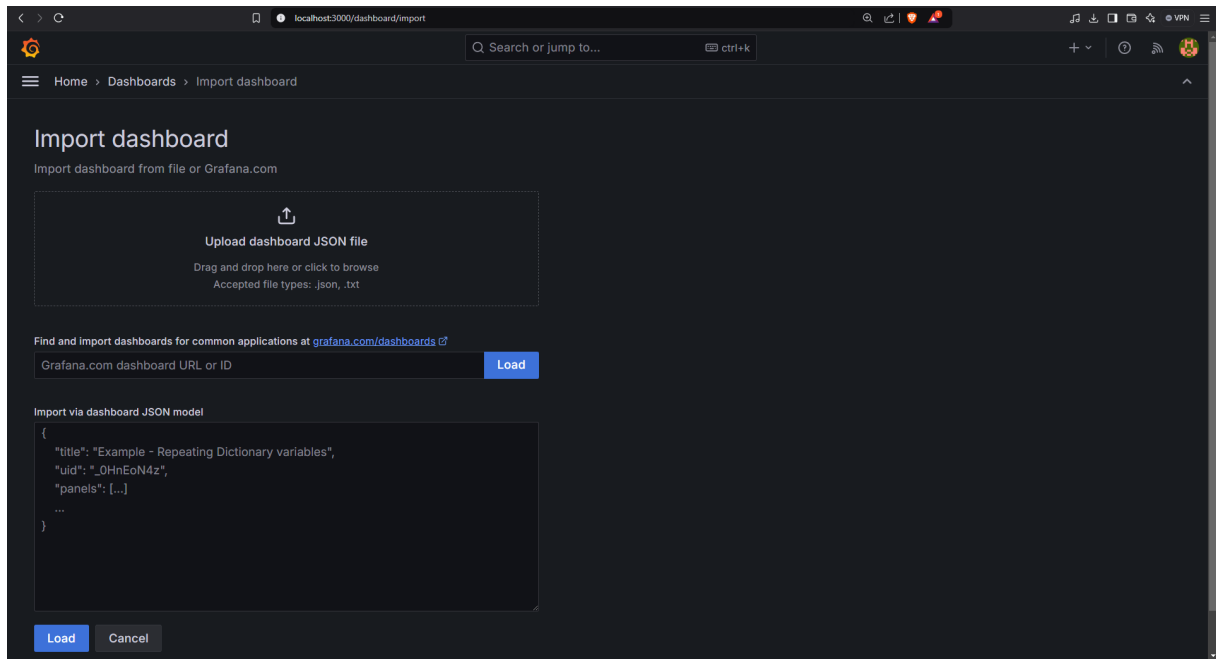
Home window of Grafana

4. Click on **New** in the left-hand corner and choose **Import**.



Click on New then Import at the left-corner

5. Upload the provided JSON file to load the dashboard template.



Import json file that I provide

3.4.3 Data Files

Ensure the following JSON files are available and formatted correctly:

- **Historical Data:** 'neon_db_data.json'
- **Predict Data:** 'next_days.json'

All these file should be store in the same folder and then in terminal that now direct to that folder please enter: `python -m http.server 8000`. Now use can you use your browser to enter this file through `http://localhost:8000/<file name>`

These files should be uploaded to the appropriate data source to populate the dashboard.

3.4.4 Alerting in Grafana

Grafana's alerting system enables users to set up alerts based on defined conditions. These alerts can be sent to various notification channels.

- **Define Alert Rules:** Set up rules in panel settings for triggering alerts.
- **Notification Channels:** Integrate notification channels like email, Slack, and PagerDuty.
- **Monitor Alerts:** View active and historical alerts in the Alerting tab.

4 Implementation

4.1 Data Production and Consumption in Kafka Topics

Producing Data of Each Sensor to the corresponding Topics in Kafka, 4 *device_id* for every matching partition (fundamental custom key value: *device_id_0* suit with partition 0, etc.), which ensures an even load distribution and optimal usage of resources in the Kafka cluster. Notice that having the number of sensor types equal to the evaluated attribute values, and in one type of sensor having 4 identical devices with the id *0,1,2,3* to push data and analyze them for handling the deviations in reality.

The image shows a Python IDE (VS Code) and the Apache Kafka UI. The IDE displays a script named `main.py` that publishes data to Kafka topics. The script uses the `pika` library to connect to a Kafka cluster and publishes messages for four different sensors (Humidity, Temperature, Renewable Energy, and Footage) across four partitions. The Kafka UI shows the `energy_consumption` topic with messages being consumed. The messages contain JSON data with fields like `deviceId`, `timestamps`, `value`, and `unit`.

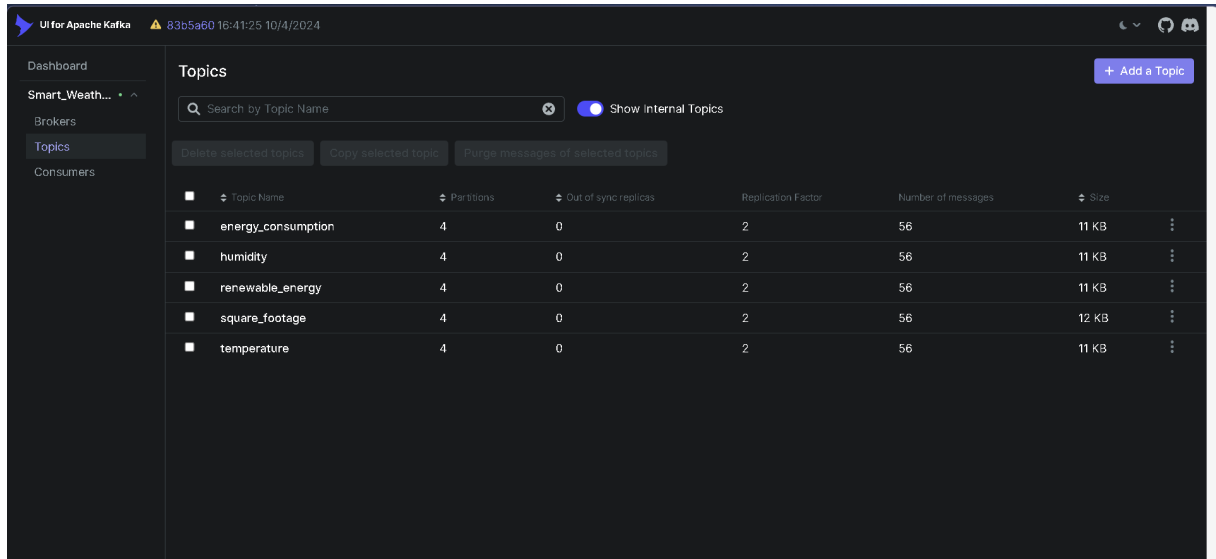
Python Script Snippet:

```
current_directory = os.path.dirname(os.path.abspath(__file__))
python_interpreter = r".\Scripts\python.exe"
scripts = [
    '..\Producer\producer_humid.py',
    '..\Producer\producer_temp.py',
    '..\Producer\producer_renew.py',
    '..\Producer\producer_footage.py',
    '..\Producer\producer_consum.py',
]
```

Kafka UI Snippet:

Offset	Partition	Timestamp	Key	Value
0	0	16:14:15 25/12/2024		{"deviceId": "0", "timestamps": "01/01/2022 00:00:00", "value": np.float64(25.13943344), "unit": "C"}
0	1	16:14:15 25/12/2024		{"deviceId": "1", "timestamps": "01/01/2022 00:01:00", "value": np.float64(43.43102984380294), "unit": "H"}
0	3	16:14:15 25/12/2024		{"deviceId": "3", "timestamps": "01/01/2022 00:00:00", "value": np.float64(75.3644483896419), "unit": "kWh"}

Figure 10: Publishing Data



Topic Name	Partitions	Out of sync replicas	Replication Factor	Number of messages	Size
energy_consumption	4	0	2	56	11 KB
humidity	4	0	2	56	11 KB
renewable_energy	4	0	2	56	11 KB
square_footage	4	0	2	56	12 KB
temperature	4	0	2	56	11 KB

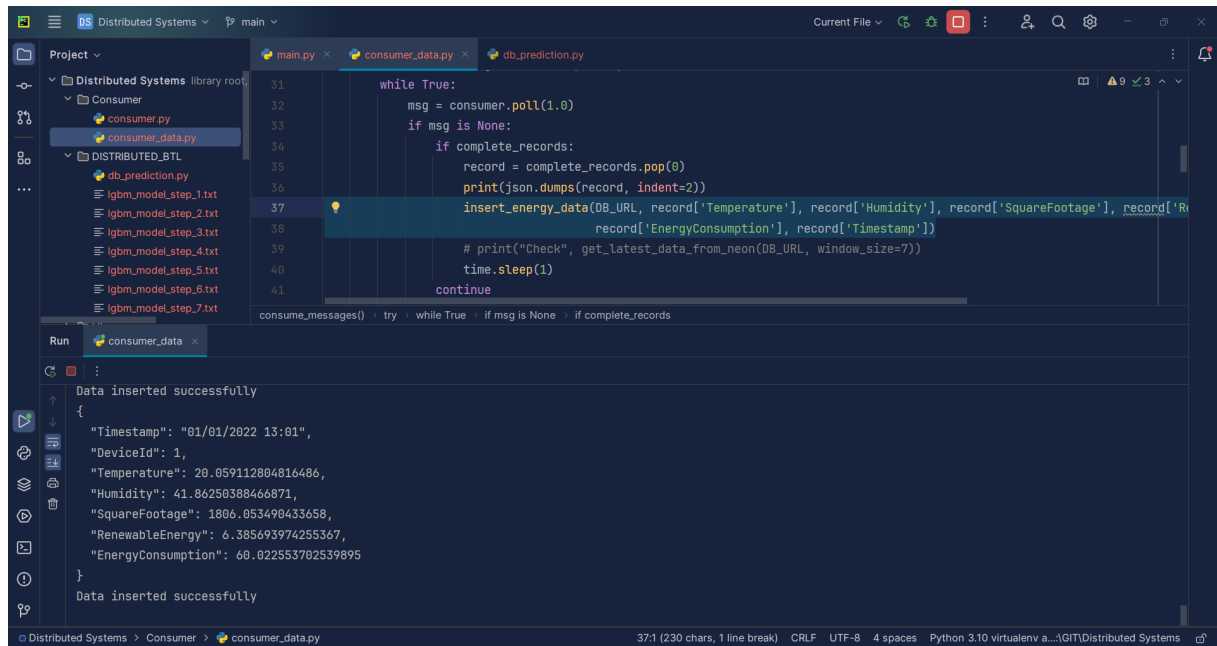
Figure 11: Overview of Data Publishing

After that, consume Data from each topic and push it to the **NeonDB database**. In addition, construct a specific JSON message format from data of the consumer topic to facilitate processing more efficiently is also essential.

```

1 {
2   "Timestamp": "01/01/2025 13:00",
3   "DeviceId": 1,
4   "Temperature": 20.056789,
5   "Humidity": 44.056789,
6   "SquareFootage": 1506.056789,
7   "RenewableEnergy": 6.056789,
8   "EnergyConsumption": 60.056789
9 }
```

Listing 2: Specific JSON message example



```

31 while True:
32     msg = consumer.poll(1.0)
33     if msg is None:
34         if complete_records:
35             record = complete_records.pop(0)
36             print(json.dumps(record, indent=2))
37             insert_energy_data(DB_URL, record['Temperature'], record['Humidity'], record['SquareFootage'], record['RenewableEnergy'], record['EnergyConsumption'], record['Timestamp'])
38             record['EnergyConsumption'], record['Timestamp']
39             # print("Check", get_latest_data_from_neon(DB_URL, window_size=7))
40             time.sleep(1)
41             continue

```

Run consumer_data

```

Data inserted successfully
{
  "Timestamp": "01/01/2022 13:01",
  "DeviceId": 1,
  "Temperature": 20.059112884816486,
  "Humidity": 41.86250388466871,
  "SquareFootage": 1806.053490433658,
  "RenewableEnergy": 6.385693974255367,
  "EnergyConsumption": 60.822553702539895
}
Data inserted successfully

```

Figure 12: Consuming Data from Kafka and Pushing to NeonDB Database

4.2 Machine Learning-based Prediction and Analytics

To predict time-series targets based on sensor historical data, we employed the **LightGBM** framework with a tailored regression setup. The workflow included preprocessing input and target sequences, training individual models for each prediction step, and evaluating model performance using standard metrics.

4.2.1 Data Preparation

The raw time-series data was smoothed using a Savitzky-Golay filter to mitigate noise while preserving the signal's trends and patterns. The smoothing parameters were set to a window size of 5 and a polynomial order of 2. This approach enhanced the model's ability to capture underlying temporal dynamics effectively.

Next, sliding window sequences were generated:

- Input sequences (**X_sequences**) were constructed by grouping the smoothed data into non-overlapping windows of size 7.
- Corresponding target sequences (**y_sequences**) were defined as the subsequent 7 time steps, ensuring alignment for supervised learning.

These sequences were converted into numpy arrays and split into training (80%) and testing (20%) sets. For compatibility with LightGBM, the input sequences were flattened into two-dimensional arrays, where each row represented a concatenated sequence of features. The target sequences were also reshaped into 1D arrays for each prediction step.

4.2.2 Model Architecture and Training

We utilized the **LGBMRegressor** from the LightGBM library, a gradient-boosting framework optimized for speed and scalability. The model was configured with the following hyperparameters:

- **Objective:** Regression
- **Metric:** Root Mean Squared Error (RMSE)
- **Learning Rate:** 0.05

- **Number of Estimators:** 200
- **Maximum Depth:** 5
- **Subsampling Rate:** 0.8
- **Feature Subsampling Rate:** 0.8
- **Maximum Leaves:** 31

Separate models were trained for each step in the prediction horizon, with each model predicting a specific output time step based on the input sequence. During training, the following process was applied:

- The model was evaluated on both the training and testing sets using the RMSE metric.
- An iterative approach was used to optimize performance for each prediction step.
- Models were saved in text format for reproducibility and further analysis.

This multi-step approach ensures that the model effectively captures the temporal dependencies in the data while maintaining high accuracy across all prediction steps.

4.2.3 Model Export and Deployment

Trained models were serialized and exported for deployment using LightGBM's native model-saving functionality. Each model corresponding to a prediction step was stored independently, enabling modular integration into downstream applications. This strategy ensures scalability and facilitates updates for individual steps without retraining the entire pipeline.

4.2.4 Significance and Advantages

The proposed model development pipeline offers the following benefits:

- **Efficiency:** LightGBM's gradient boosting approach provides fast training and prediction times.
- **Scalability:** Separate models for each prediction step allow for modular expansion and adaptation.
- **Accuracy:** Smoothing the data before training enhances prediction robustness by reducing noise-related artifacts.
- **Interpretability:** Feature importances can be extracted from LightGBM models, providing insights into the factors influencing predictions.

4.2.5 Experimental Results

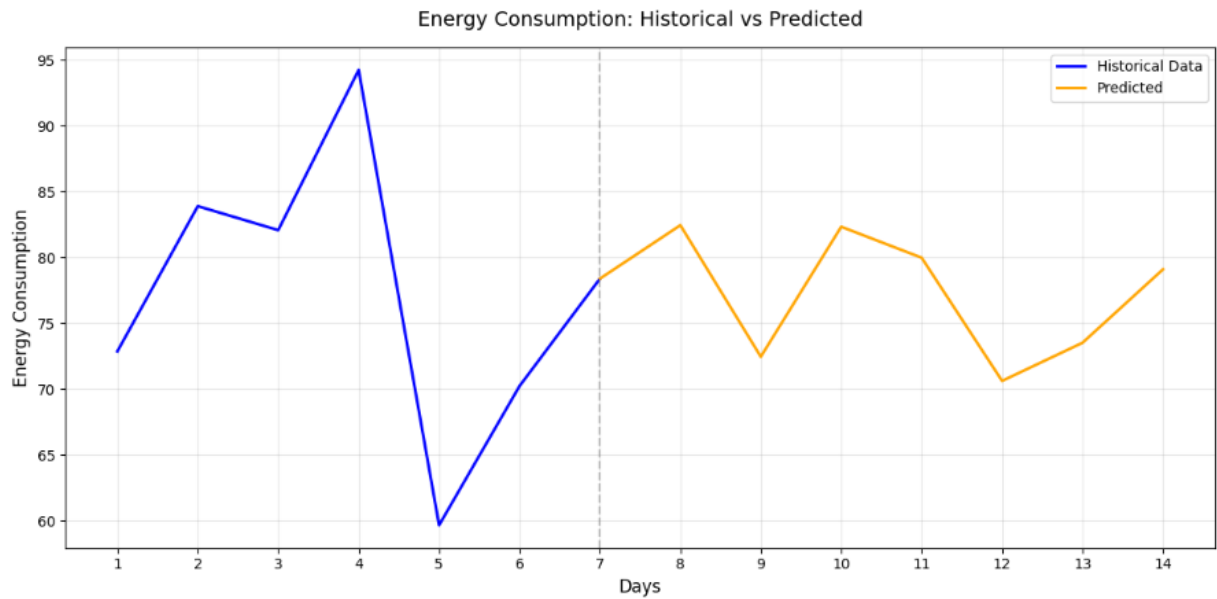


Figure 13: Our demo with ML Model for 7-to-7 prediction

4.3 Data Visualization in Grafana

4.3.1 Visualize Graph for each type of Data

In this project, I will visualize 5 graph: Temperature, Humidity, Square Footage, Renewable Energy, Energy Consumption and it prediction



Figure 14: Temperature Visualization

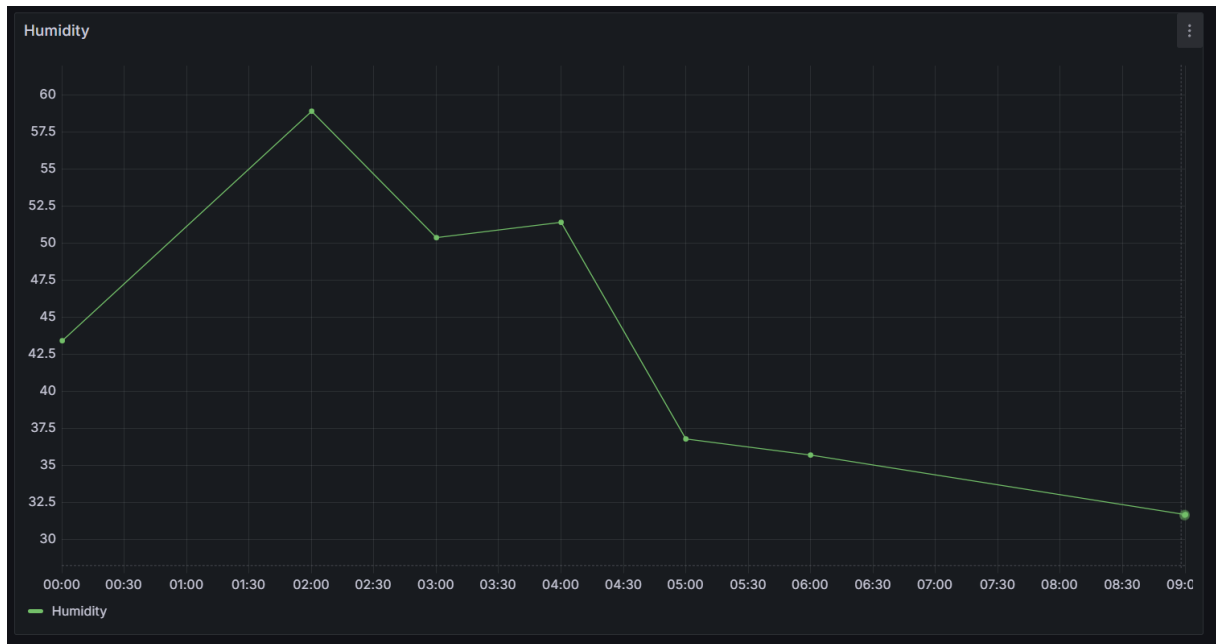


Figure 15: Humidity Visualization

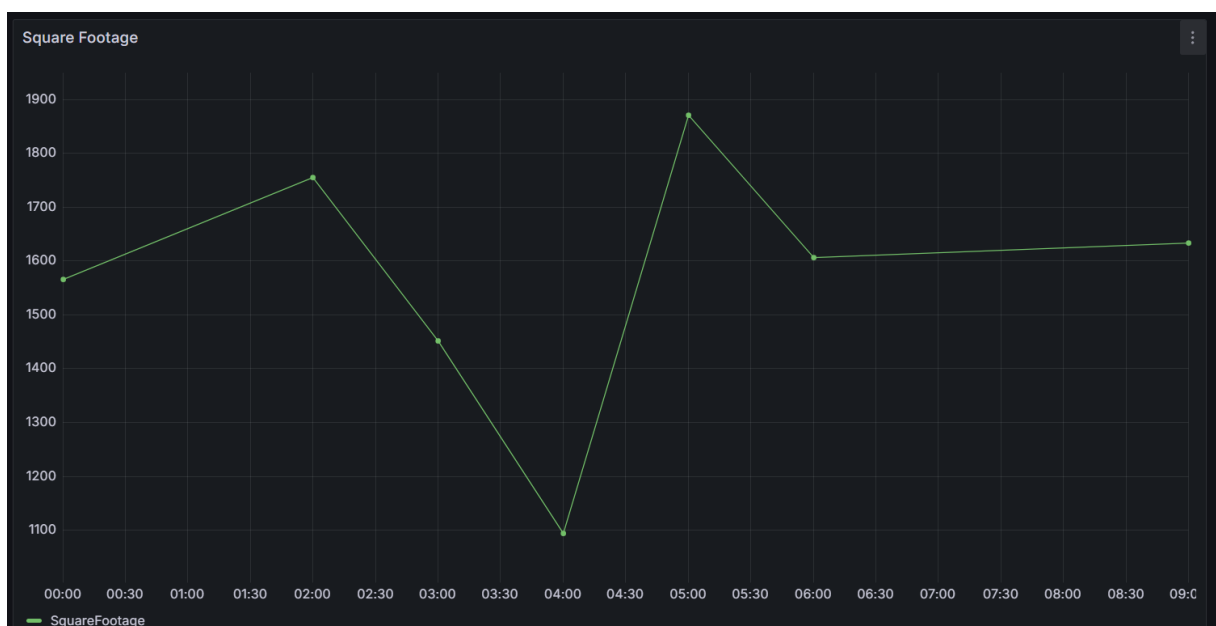


Figure 16: Square-footage Visualization



Figure 17: Renewable Energy Visualization

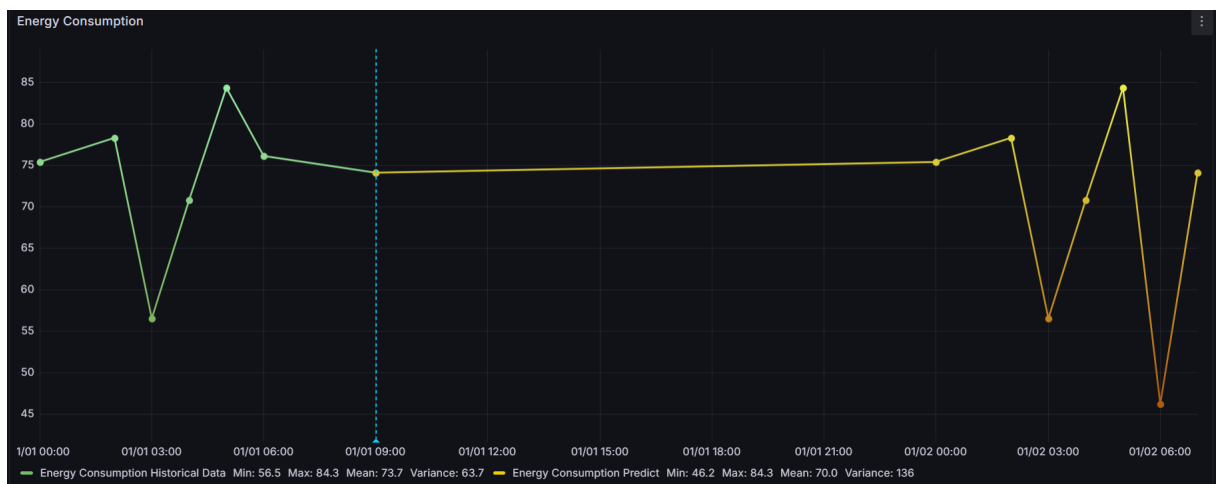


Figure 18: Energy Consumption and its prediction visualization

4.3.2 Some limitation of Grafana

- **Dependency on Data Sources:**

- Grafana relies on external data sources (e.g., Prometheus, InfluxDB) for data ingestion and storage.
- Performance of real-time streaming depends on the capabilities of the underlying data source.

- **Update Frequency:**

- Dashboards refresh at intervals (default 5 seconds, configurable), which may not provide true real-time updates.
- Shorter intervals increase system load and may not meet ultra-low latency requirements.

- **Data Source Support:**

- Not all supported data sources are optimized for high-frequency real-time streaming.

- For example, Prometheus is better suited for metrics than for real-time event streaming.
- SQL databases or Elasticsearch may face challenges with frequent updates or large datasets.
- **High-Latency Issues:**
 - Large volumes of streaming data can degrade visualization performance.
 - Complex queries or transformations increase latency, delaying real-time updates.
- **Limited Stream Handling Features:**
 - Grafana lacks native support for handling event streams like Kafka or MQTT.
 - Middleware tools such as Loki or Fluentd are often required for stream processing.
- **Scalability Challenges:**
 - High-frequency data streams with thousands of events per second can slow down dashboards.
 - Custom plugins or external optimizations may be needed to improve scalability.
- **Complexity of Query Configuration:**
 - Configuring queries for streaming requires advanced knowledge of Grafana and the data source query language (e.g., PromQL or SQL).
 - Lack of built-in tools for optimizing queries for real-time data.
- **Limited Interactive Streaming Features:**
 - Grafana primarily focuses on visualization rather than interactivity.
 - Features like pausing, rewinding, or fast-forwarding through a data stream are not available.
- **Cost and Resource Usage:**
 - High-frequency streaming increases resource usage (CPU, memory, and bandwidth) for Grafana and data sources.
 - Operational costs can escalate in resource-intensive environments.
- **Alerting Constraints:**
 - Alerting capabilities on streaming data depend on the data source and may be limited in real-time scenarios.
 - Complex alert conditions may not perform efficiently in high-frequency streaming contexts.
- **Alternatives for High-Performance Streaming:**
 - Use tools like **Apache Kafka** for event streaming.
 - Consider high-speed time-series databases such as **TimescaleDB**, **Druid**, or **InfluxDB**.
 - Employ **Fluentd** or **Logstash** for data aggregation and pre-processing.
 - Use streaming frameworks like **Apache Flink**, **Kafka Streams**, or **Apache NiFi**.

As that, we used the json file so it will take more space in the hard disk, and the Grafana have to base on datasource so much so that one json file is not enough in reality, which our team is use for our future plan for this project

5 Conclusion

In this report, we present a comprehensive architecture for a smart energy consumption prediction pipeline that integrates real-time data streaming, efficient message queuing, and predictive analytics. Our system leverages modern distributed technologies, including Apache Kafka, HiveMQ, Docker, and NeonDB, to effectively handle and process large-scale time-series data generated by IoT devices in real time.

The pipeline is designed with high availability, scalability, and data reliability in mind, employing essential features such as Kafka's partitioning, replication, and fault tolerance mechanisms. Furthermore, Docker facilitates dynamic scaling and streamlined deployment, providing the flexibility and robustness necessary to meet growing demands.

By incorporating an AI-based prediction model, the platform enhances its functionality through real-time energy consumption forecasts, which are critical for informed energy management decisions. These forecasts are presented via a user-friendly dashboard interface, enabling end users to monitor energy trends and make adjustments as needed.

Additionally, we explore the use of Kafka Connect to enable seamless data flow from Kafka to NeonDB, facilitating advanced analytics on the streamed data. To further enhance the system's extensibility, we developed a Flask API that allows for efficient querying and retrieval of data from NeonDB, positioning the pipeline for integration with larger applications.

Security considerations are also paramount; we have implemented measures to protect data flowing through Kafka, NeonDB and Grafana via authentication, encryption, and least-privilege access protocols, thereby ensuring privacy and data integrity.

In conclusion, this pipeline exemplifies a highly scalable, efficient, and secure approach to managing and predicting energy consumption data. It illustrates the potential of combining distributed systems with AI-driven analytics to foster innovation and efficiency in smart energy management, paving the way for sustainable digital transformation.