# Azure Data Pipeline Project Report

Nora Ayaz

October 10, 2024

## Contents

# 1 Initial Setup and Configuration

Before implementing the cloud-based data pipeline, several initial configurations were required on the Azure Portal. These steps were crucial for setting up the resources that enabled the pipeline's functionality. The following key components were created and configured:

## 1.1 Azure Student Account

An Azure Student account was created to gain access to a free tier of Azure services, which includes free credits and access to essential services like Azure Storage, Azure SQL Database, and Function Apps.

## 1.2 Resource Group

A Resource Group was created to organize and manage all related Azure resources under one grouping. This helps with administration, cost tracking, and resource management.

## 1.3 Azure Storage Account

An Azure Storage Account was provisioned to serve as the backbone for the queue service. Within the storage account:

- A **Queue Service** was created to hold sensor data messages sent by the data producer.
- The **Queue** was named `myqueue1090` to store and queue incoming data for processing.

## 1.4 Azure SQL Database and Server

An Azure SQL Database was set up to store the sensor data processed by the Azure Function. The following steps were taken:

- A **SQL Server** was created to manage the database. The server was configured with appropriate firewall rules to allow access from specific IP addresses.
- A **SQL Database** named `mydatabase1090` was created to store the processed sensor data (temperature, humidity, and timestamp).
- The database was set to the **serverless tier** to enable cost-effective scaling and automatic pausing during periods of inactivity.

## 1.5 Azure Function App

An **Azure Function App** was created to handle the processing of data messages from the queue. The function app automatically triggers whenever new data arrives in the queue, processes the message, and inserts the data into the Azure SQL Database.

### 1.6 Power BI and Grafana Setup

To visualize the stored data:

- **Power BI** was connected to the Azure SQL Database to create reports and visualizations based on the stored sensor data.
- **Grafana** was installed on a cloud-hosted environment and configured to connect to the Azure SQL Database, providing real-time data dashboards.

This initial setup allowed for the creation of a seamless data pipeline from data generation at the edge, through processing and storage in the cloud, to visualization using various tools.

# 2 Introduction

The goal of this project was to design and implement a serverless data pipeline using Azure services, demonstrating cloud computing skills in data generation, ingestion, storage, and consumption. The project simulates a real-world IoT-like system where data is generated at the edge, sent to the cloud for storage, and later consumed by data consumers for analysis and visualization.

The key components of this project include:

**Data Producer:** A Python script (`send_to_queue.py`) generates random sensor data (temperature, humidity, and timestamps) and sends it to an Azure Queue. This simulates the edge data generation and transmission to the cloud.

**Message Queue:** Azure Queue Storage is used as the queuing service to temporarily store the sensor data messages from the producer before processing.

**Data Ingestion:** An Azure Function (`function_app.py`) is triggered automatically when new data arrives in the Azure Queue. This serverless function processes the data and inserts it into an Azure SQL Database, ensuring that the data is ingested and stored efficiently.

**Data Storage:** The ingested data is stored in an Azure SQL Database (serverless tier), which provides a scalable and reliable cloud-based storage solution for the processed sensor data.

**Data Consumers:**

- **Grafana:** Connects to the Azure SQL Database to visualize the stored data in real-time dashboards, providing meaningful insights from the sensor data.
- **Power BI:** Retrieves data from the Azure SQL Database for advanced data analysis and reporting, enabling further business intelligence capabilities.
- **Python Data Consumer:** A custom Python script (`python_data_consumer.py`) connects to the Azure SQL Database to fetch the sensor data and visualizes it using `matplotlib`, providing an additional method of exploring the data.

This project demonstrates the integration of multiple Azure services to build a serverless, cost-efficient, and scalable cloud-based data pipeline. It showcases how serverless functions, cloud-based databases, and various data consumption tools (Grafana, Power BI, and a custom Python script) work together to process, store, and visualize data in near real-time, simulating a complete IoT data pipeline.

## 3 System Overview

This project implements a cloud-based data pipeline using Azure services. The system consists of a data producer that generates sensor data (temperature, humidity, and timestamps) and sends it to an Azure Queue. An Azure Function is automatically triggered by the arrival of new data in the queue. The Azure Function processes the data and inserts it into an Azure SQL Database.

Once the data is stored in the Azure SQL Database, it is visualized using various tools:

- **Grafana:** Visualizes real-time sensor data from the SQL Database in interactive dashboards, providing an intuitive way to monitor trends.
- **Power BI:** Retrieves data from the SQL Database to create detailed reports and conduct advanced data analysis.
- **Python Data Consumer:** A custom Python script fetches data from the SQL Database and visualizes it using `matplotlib`, offering a flexible data exploration method.

The entire system leverages serverless components such as Azure Functions and Azure SQL Database for scalability, cost-efficiency, and ease of management, ensuring secure and reliable data flow from generation to consumption. This architecture simulates a robust and scalable IoT data pipeline, enabling near real-time insights into sensor data.

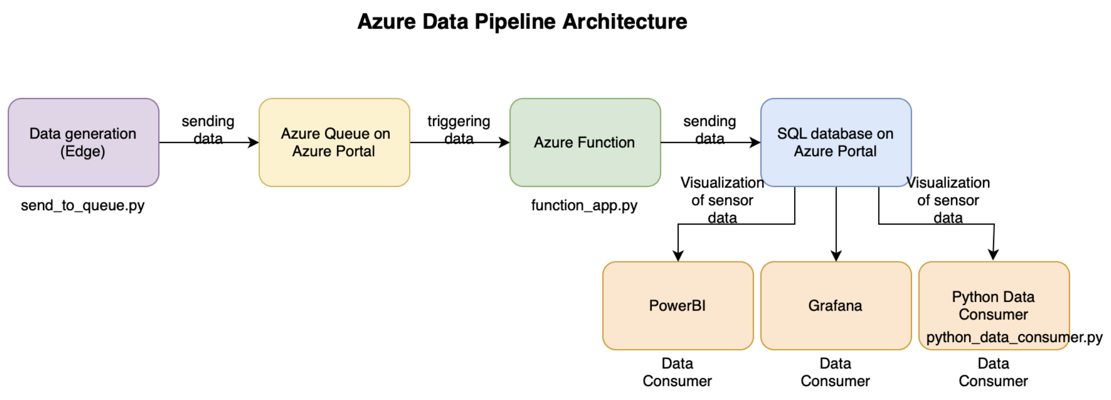


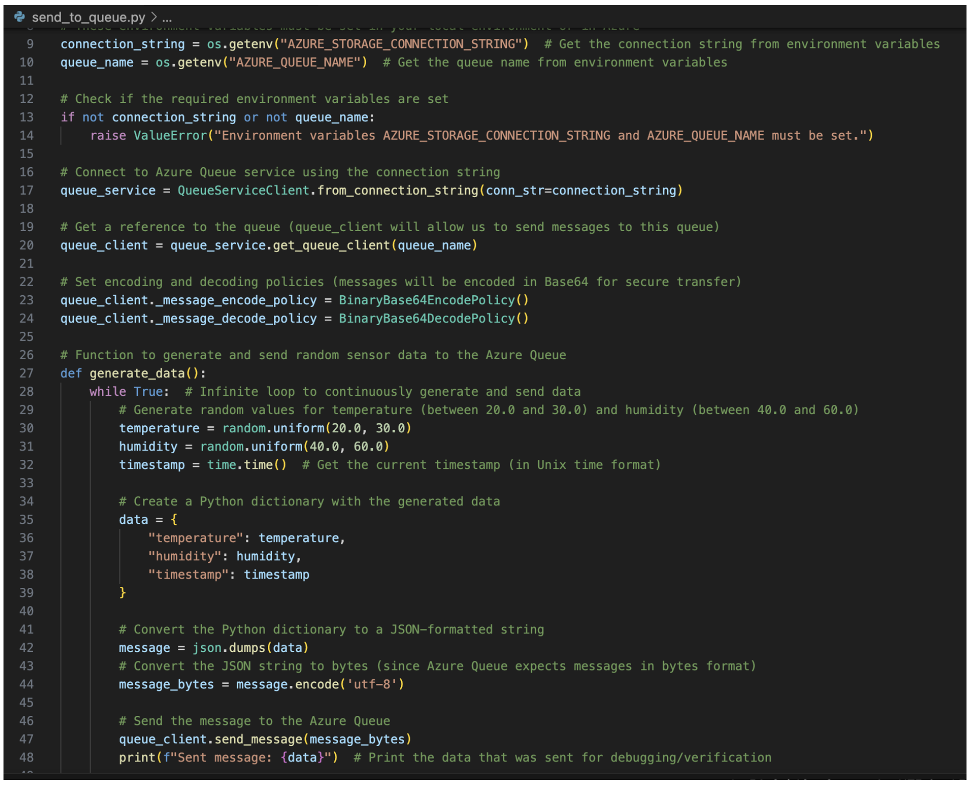
Figure 1: Data Pipeline Overview.

## 4 Components

### 4.1 Data Producer

The data producer script (`send_to_queue.py`) generates random temperature, humidity, and timestamp data in JSON format and sends it to an Azure Queue. The script uses environment variables to securely store sensitive connection information such as the queue name and connection string. By continuously sending data, this component simulates an

IoT-like edge device in the data pipeline, providing real-time input for the system to process.



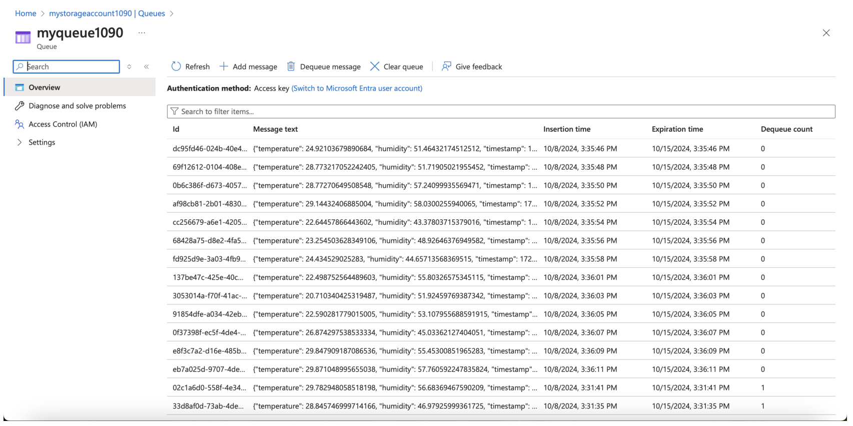
send_to_queue.py > ...

```python
connection_string = os.getenv("AZURE_STORAGE_CONNECTION_STRING")  # Get the connection string from environment variables
queue_name = os.getenv("AZURE_QUEUE_NAME")  # Get the queue name from environment variables

# Check if the required environment variables are set
if not connection_string or not queue_name:
    raise ValueError("Environment variables AZURE_STORAGE_CONNECTION_STRING and AZURE_QUEUE_NAME must be set.")

# Connect to Azure Queue service using the connection string
queue_service = QueueServiceClient.from_connection_string(conn_str=connection_string)

# Get a reference to the queue (queue_client will allow us to send messages to this queue)
queue_client = queue_service.get_queue_client(queue_name)

# Set encoding and decoding policies (messages will be encoded in Base64 for secure transfer)
queue_client._message_encode_policy = BinaryBase64EncodePolicy()
queue_client._message_decode_policy = BinaryBase64DecodePolicy()

# Function to generate and send random sensor data to the Azure Queue
def generate_data():
    while True:  # Infinite loop to continuously generate and send data
        # Generate random values for temperature (between 20.0 and 30.0) and humidity (between 40.0 and 60.0)
        temperature = random.uniform(20.0, 30.0)
        humidity = random.uniform(40.0, 60.0)
        timestamp = time.time()  # Get the current timestamp (in Unix time format)

        # Create a Python dictionary with the generated data
        data = {
            "temperature": temperature,
            "humidity": humidity,
            "timestamp": timestamp
        }

        # Convert the Python dictionary to a JSON-formatted string
        message = json.dumps(data)
        # Convert the JSON string to bytes (since Azure Queue expects messages in bytes format)
        message_bytes = message.encode('utf-8')

        # Send the message to the Azure Queue
        queue_client.send_message(message_bytes)
        print(f"Sent message: {data}")  # Print the data that was sent for debugging/verification
```

Figure 2: Data Producer Script.

## 4.2 Azure Queue

In this project, an Azure Storage Account was used to create a Queue service, allowing for reliable message queuing between the data producer and the next stages in the pipeline. The producer script sends JSON messages containing sensor data to this queue. The messages are then picked up by an Azure Function, which processes and inserts them into the Azure SQL Database. This ensures asynchronous and scalable communication between the components in the data pipeline.



**Authentication method:** Access key (Switch to Microsoft Entra user account)

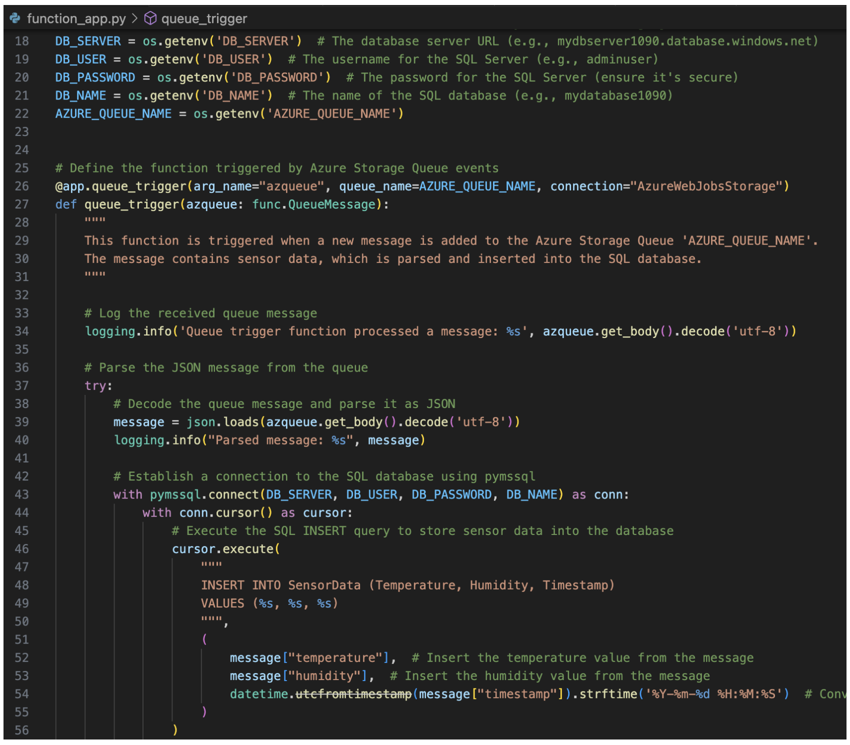
Search to filter items...

| Id | Message text | Insertion time | Expiration time | Dequeue count |
|---|---|---|---|---|
| dc95fd46-024b-40e4... | {"temperature": 24.92103679890684, "humidity": 51.46432174512512, "timestamp": 1... | 10/8/2024, 3:35:46 PM | 10/15/2024, 3:35:46 PM | 0 |
| 69f12612-0104-408e... | {"temperature": 28.773217052242405, "humidity": 51.71905021955452, "timestamp": ... | 10/8/2024, 3:35:48 PM | 10/15/2024, 3:35:48 PM | 0 |
| 0b6c386f-d673-4057... | {"temperature": 28.77270649508548, "humidity": 57.24099935569471, "timestamp": 1... | 10/8/2024, 3:35:50 PM | 10/15/2024, 3:35:50 PM | 0 |
| af98cb81-2b01-4830... | {"temperature": 29.14432406885004, "humidity": 58.0300255940065, "timestamp": 17... | 10/8/2024, 3:35:52 PM | 10/15/2024, 3:35:52 PM | 0 |
| cc256679-a6e1-4205... | {"temperature": 22.64457866443602, "humidity": 43.37803715379016, "timestamp": 1... | 10/8/2024, 3:35:54 PM | 10/15/2024, 3:35:54 PM | 0 |
| 68428a75-d8e2-4fa5... | {"temperature": 23.254503628349106, "humidity": 48.92646376949582, "timestamp": ... | 10/8/2024, 3:35:56 PM | 10/15/2024, 3:35:56 PM | 0 |
| fd925d9e-3a03-4fb9... | {"temperature": 24.434529025283, "humidity": 44.65713568369515, "timestamp": 172... | 10/8/2024, 3:35:58 PM | 10/15/2024, 3:35:58 PM | 0 |
| 137be47c-425e-40c... | {"temperature": 22.498752564489603, "humidity": 55.80326575345115, "timestamp": ... | 10/8/2024, 3:36:01 PM | 10/15/2024, 3:36:01 PM | 0 |
| 3053014a-f70f-41ac-... | {"temperature": 20.710340425319487, "humidity": 51.92459769387342, "timestamp": ... | 10/8/2024, 3:36:03 PM | 10/15/2024, 3:36:03 PM | 0 |
| 91854dfe-a034-42eb... | {"temperature": 22.590281779015005, "humidity": 53.107955688591915, "timestamp"... | 10/8/2024, 3:36:05 PM | 10/15/2024, 3:36:05 PM | 0 |
| 0f37398f-ec5f-4de4-... | {"temperature": 26.874297538533334, "humidity": 45.03362127404051, "timestamp": ... | 10/8/2024, 3:36:07 PM | 10/15/2024, 3:36:07 PM | 0 |
| e8f3c7a2-d16e-485b... | {"temperature": 29.847909187086536, "humidity": 55.45300851965283, "timestamp": ... | 10/8/2024, 3:36:09 PM | 10/15/2024, 3:36:09 PM | 0 |
| eb7a025d-9707-4de... | {"temperature": 29.871048995655038, "humidity": 57.760592247835824, "timestamp"... | 10/8/2024, 3:36:11 PM | 10/15/2024, 3:36:11 PM | 0 |
| 02c1a6d0-558f-4e34... | {"temperature": 29.782948058518198, "humidity": 56.68369467590209, "timestamp": ... | 10/8/2024, 3:31:41 PM | 10/15/2024, 3:31:41 PM | 1 |
| 33d8af0d-73ab-4de... | {"temperature": 28.845746999714166, "humidity": 46.97925999361725, "timestamp": ... | 10/8/2024, 3:31:35 PM | 10/15/2024, 3:31:35 PM | 1 |

Figure 3: Azure Queue Overview.

## 4.3 Azure Function and Data Ingestion

An Azure Function (`function_app.py`) is used to ingest data from the Azure Queue into the Azure SQL Database. The function is triggered automatically whenever new data arrives in the queue, ensuring real-time processing of sensor data. The function processes the JSON message, extracts the temperature, humidity, and timestamp, and inserts this data into the `SensorData` table in the Azure SQL Database. Environment variables are used to store sensitive information like database connection strings, ensuring security and scalability without relying on virtual machines.



function_app.py > queue_trigger

```python
DB_SERVER = os.getenv('DB_SERVER')  # The database server URL (e.g., mydbserver1090.database.windows.net)
DB_USER = os.getenv('DB_USER')  # The username for the SQL Server (e.g., adminuser)
DB_PASSWORD = os.getenv('DB_PASSWORD')  # The password for the SQL Server (ensure it's secure)
DB_NAME = os.getenv('DB_NAME')  # The name of the SQL database (e.g., mydatabase1090)
AZURE_QUEUE_NAME = os.getenv('AZURE_QUEUE_NAME')


# Define the function triggered by Azure Storage Queue events
@app.queue_trigger(arg_name="azqueue", queue_name=AZURE_QUEUE_NAME, connection="AzureWebJobsStorage")
def queue_trigger(azqueue: func.QueueMessage):
    """
    This function is triggered when a new message is added to the Azure Storage Queue 'AZURE_QUEUE_NAME'.
    The message contains sensor data, which is parsed and inserted into the SQL database.
    """

    # Log the received queue message
    logging.info('Queue trigger function processed a message: %s', azqueue.get_body().decode('utf-8'))

    # Parse the JSON message from the queue
    try:
        # Decode the queue message and parse it as JSON
        message = json.loads(azqueue.get_body().decode('utf-8'))
        logging.info("Parsed message: %s", message)

        # Establish a connection to the SQL database using pymssql
        with pymssql.connect(DB_SERVER, DB_USER, DB_PASSWORD, DB_NAME) as conn:
            with conn.cursor() as cursor:
                # Execute the SQL INSERT query to store sensor data into the database
                cursor.execute(
                    """
                    INSERT INTO SensorData (Temperature, Humidity, Timestamp)
                    VALUES (%s, %s, %s)
                    """,
                    (
                        message["temperature"],  # Insert the temperature value from the message
                        message["humidity"],  # Insert the humidity value from the message
                        datetime.~~utcfromtimestamp~~(message["timestamp"]).strftime('%Y-%m-%d %H:%M:%S')  # Conv
                    )
                )
```

Figure 4: Azure Function for Data Ingestion.

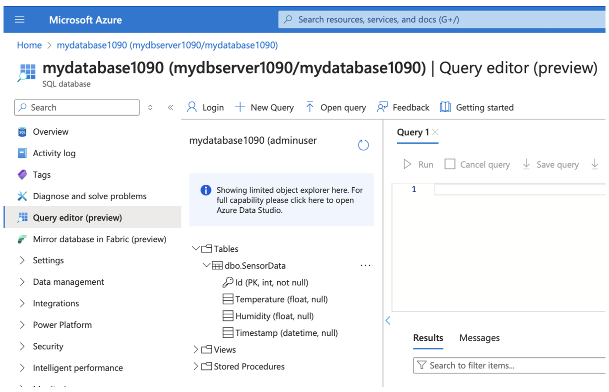
## 4.4 Azure SQL Database

The Azure SQL Database acts as the central storage component in the data pipeline. It provides a serverless and scalable solution for storing the processed sensor data. The serverless tier was selected to minimize costs while maintaining the ability to scale automatically based on demand, and it pauses during inactivity.

The data from the Azure Queue is processed by the Azure Function and inserted into the `SensorData` table in the SQL Database, which stores:

**Temperature:** The sensor-generated temperature data.
**Humidity:** The humidity levels from the sensors.
**Timestamp:** The exact time when the data was generated.

The database offers a secure and high-availability storage solution, ensuring that the sensor data can be easily accessed by data consumers like Grafana, Power BI, and a custom Python Data Consumer for visualization and analysis. This component is critical for managing and storing data in the pipeline.


mydatabase1090 (mydbserver1090/mydatabase1090) | Query editor (preview)

SQL database

mydatabase1090 (adminuser

Showing limited object explorer here. For full capability please click here to open Azure Data Studio.

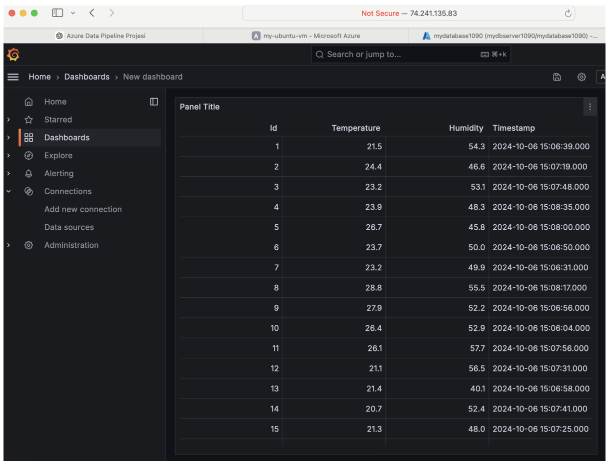
```
Tables
  dbo.SensorData
    Id (PK, int, not null)
    Temperature (float, null)
    Humidity (float, null)
    Timestamp (datetime, null)
Views
Stored Procedures
```

Figure 5: Azure SQL Database Overview.

## 4.5 Data Consumers

The stored data in the Azure SQL Database is visualized and analyzed using three different data consumers: **Grafana**, **Power BI**, and a **Custom Python Data Consumer**. These tools provide insight into the data, confirming that the pipeline is functioning as expected and allowing users to monitor the real-time flow of sensor data.

**Grafana:** Grafana was connected to the Azure SQL Database to create a real-time dashboard. It displays key metrics like temperature, humidity, and timestamp in an interactive interface, helping to monitor the incoming data. Grafana provides a clear, visual representation of the data pipeline's performance and ensures that the data flow is functioning correctly.

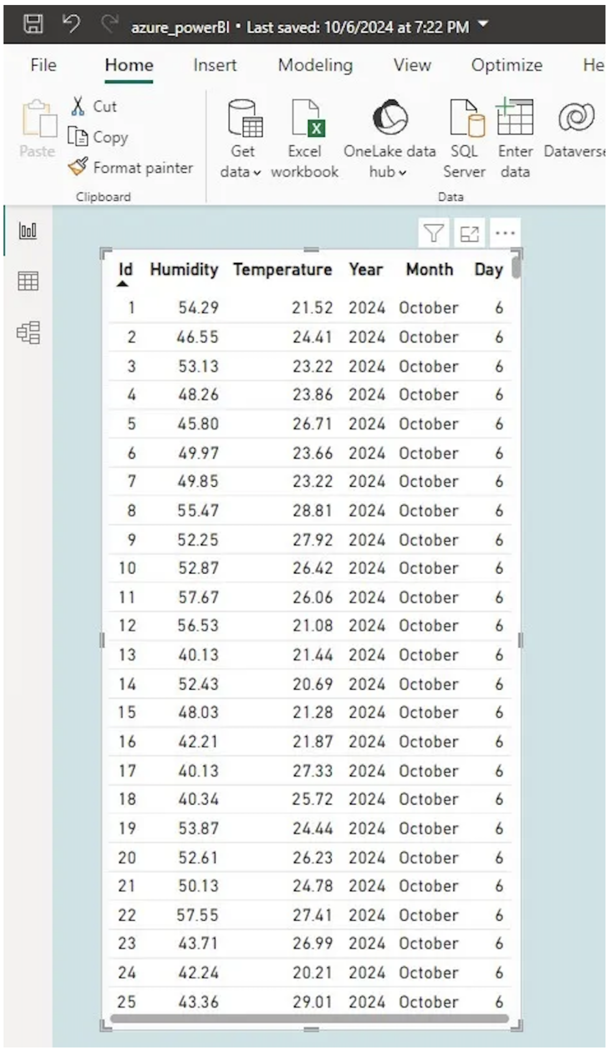

Not Secure — 74.241.135.83
Azure Data Pipeline Projesi
my-ubuntu-vm - Microsoft Azure
mydatabase1090 (mydbserver1090/mydatabase1090) -...
Search or jump to...
Home › Dashboards › New dashboard
Home
Starred
Dashboards
Explore
Alerting
Connections
Add new connection
Data sources
Administration

Panel Title

| Id | Temperature | Humidity | Timestamp |
|---|---|---|---|
| 1 | 21.5 | 54.3 | 2024-10-06 15:06:39.000 |
| 2 | 24.4 | 46.6 | 2024-10-06 15:07:19.000 |
| 3 | 23.2 | 53.1 | 2024-10-06 15:07:48.000 |
| 4 | 23.9 | 48.3 | 2024-10-06 15:08:35.000 |
| 5 | 26.7 | 45.8 | 2024-10-06 15:08:00.000 |
| 6 | 23.7 | 50.0 | 2024-10-06 15:06:50.000 |
| 7 | 23.2 | 49.9 | 2024-10-06 15:06:31.000 |
| 8 | 28.8 | 55.5 | 2024-10-06 15:08:17.000 |
| 9 | 27.9 | 52.2 | 2024-10-06 15:06:56.000 |
| 10 | 26.4 | 52.9 | 2024-10-06 15:06:04.000 |
| 11 | 26.1 | 57.7 | 2024-10-06 15:07:56.000 |
| 12 | 21.1 | 56.5 | 2024-10-06 15:07:31.000 |
| 13 | 21.4 | 40.1 | 2024-10-06 15:06:58.000 |
| 14 | 20.7 | 52.4 | 2024-10-06 15:07:41.000 |
| 15 | 21.3 | 48.0 | 2024-10-06 15:07:25.000 |

Figure 6: Grafana Dashboard Visualization.

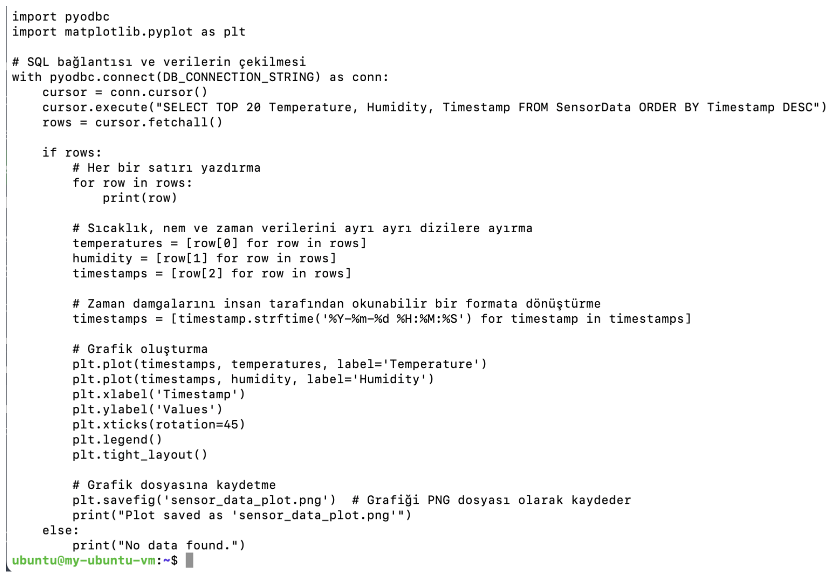
**Power BI:** Power BI was used to visualize the same data stored in the SQL Database. Similar to Grafana, it retrieves and displays temperature, humidity, and timestamp data, providing advanced data analytics and reporting capabilities. Power BI's flexibility allows for more detailed reporting and further insights into the collected data.



| Id | Humidity | Temperature | Year | Month | Day |
|---|---|---|---|---|---|
| 1 | 54.29 | 21.52 | 2024 | October | 6 |
| 2 | 46.55 | 24.41 | 2024 | October | 6 |
| 3 | 53.13 | 23.22 | 2024 | October | 6 |
| 4 | 48.26 | 23.86 | 2024 | October | 6 |
| 5 | 45.80 | 26.71 | 2024 | October | 6 |
| 6 | 49.97 | 23.66 | 2024 | October | 6 |
| 7 | 49.85 | 23.22 | 2024 | October | 6 |
| 8 | 55.47 | 28.81 | 2024 | October | 6 |
| 9 | 52.25 | 27.92 | 2024 | October | 6 |
| 10 | 52.87 | 26.42 | 2024 | October | 6 |
| 11 | 57.67 | 26.06 | 2024 | October | 6 |
| 12 | 56.53 | 21.08 | 2024 | October | 6 |
| 13 | 40.13 | 21.44 | 2024 | October | 6 |
| 14 | 52.43 | 20.69 | 2024 | October | 6 |
| 15 | 48.03 | 21.28 | 2024 | October | 6 |
| 16 | 42.21 | 21.87 | 2024 | October | 6 |
| 17 | 40.13 | 27.33 | 2024 | October | 6 |
| 18 | 40.34 | 25.72 | 2024 | October | 6 |
| 19 | 53.87 | 24.44 | 2024 | October | 6 |
| 20 | 52.61 | 26.23 | 2024 | October | 6 |
| 21 | 50.13 | 24.78 | 2024 | October | 6 |
| 22 | 57.55 | 27.41 | 2024 | October | 6 |
| 23 | 43.71 | 26.99 | 2024 | October | 6 |
| 24 | 42.24 | 20.21 | 2024 | October | 6 |
| 25 | 43.36 | 29.01 | 2024 | October | 6 |

Figure 7: Power BI Dashboard Visualization.

**Python Data Consumer:** A custom Python script (`python_data_consumer.py`) was developed and executed on an Azure Virtual Machine (VM) to connect to the Azure SQL Database and retrieve sensor data. The script securely connects to the database using environment variables and fetches the latest 20 records of temperature, humidity, and timestamp data. The retrieved data is then visualized using `matplotlib`, which generates line plots showing trends of temperature and humidity over time. The generated plot is saved on the VM for further analysis or reporting.



```
import pyodbc
import matplotlib.pyplot as plt

# SQL bağlantısı ve verilerin çekilmesi
with pyodbc.connect(DB_CONNECTION_STRING) as conn:
    cursor = conn.cursor()
    cursor.execute("SELECT TOP 20 Temperature, Humidity, Timestamp FROM SensorData ORDER BY Timestamp DESC")
    rows = cursor.fetchall()

    if rows:
        # Her bir satırı yazdırma
        for row in rows:
            print(row)

        # Sıcaklık, nem ve zaman verilerini ayrı ayrı dizilere ayırma
        temperatures = [row[0] for row in rows]
        humidity = [row[1] for row in rows]
        timestamps = [row[2] for row in rows]

        # Zaman damgalarını insan tarafından okunabilir bir formata dönüştürme
        timestamps = [timestamp.strftime('%Y-%m-%d %H:%M:%S') for timestamp in timestamps]

        # Grafik oluşturma
        plt.plot(timestamps, temperatures, label='Temperature')
        plt.plot(timestamps, humidity, label='Humidity')
        plt.xlabel('Timestamp')
        plt.ylabel('Values')
        plt.xticks(rotation=45)
        plt.legend()
        plt.tight_layout()

        # Grafik dosyasına kaydetme
        plt.savefig('sensor_data_plot.png')  # Grafiği PNG dosyası olarak kaydeder
        print("Plot saved as 'sensor_data_plot.png'")
    else:
        print("No data found.")
ubuntu@my-ubuntu-vm:~$
```

Figure 8: Execution of the `python_data_consumer.py` script on the Azure VM.

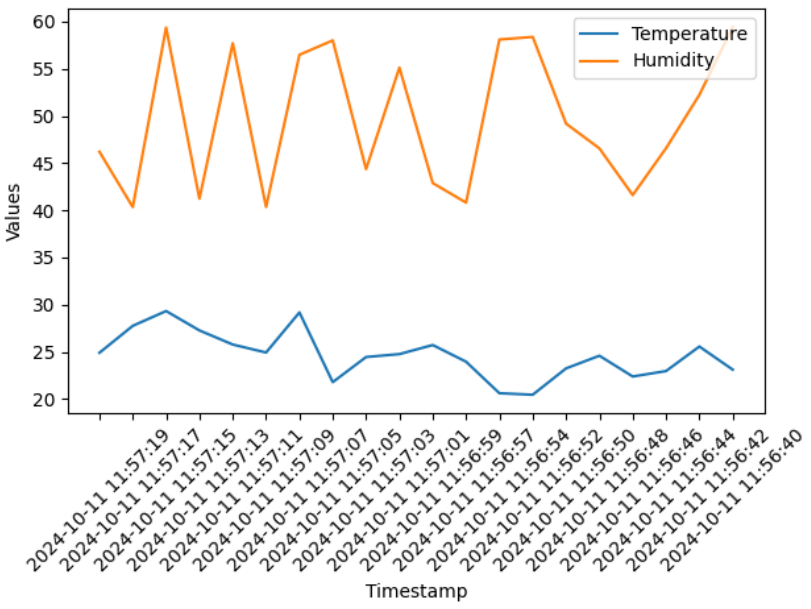


Figure 9: Visualization of sensor data using `matplotlib`, showing trends in temperature and humidity.

Both Grafana, Power BI, and the Python script confirm the proper flow of data from generation through to storage and visualization, ensuring the pipeline's integrity and performance.

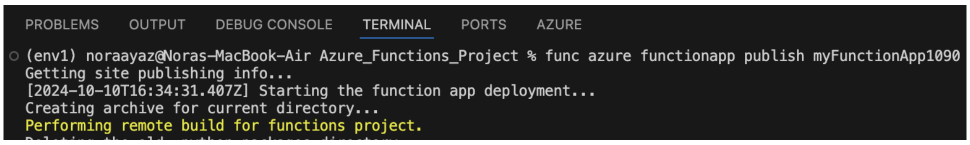
## 5 Results

The data pipeline was successfully implemented using Azure services, enabling the end-to-end flow of data from generation to storage and consumption. The Python producer script (`send_to_queue.py`) generated random sensor data and sent it to the Azure Queue.

An Azure Function (`function_app.py`) was automatically triggered by the arrival of new data in the queue, processing the messages and inserting the data into the Azure SQL Database.

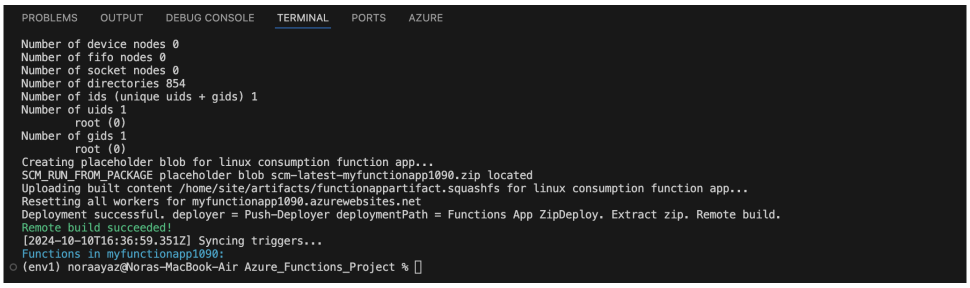
Data consumers like Grafana, Power BI, and a custom Python Data Consumer were successfully configured to visualize the data stored in the database. Grafana provided real-time data insights using simple charts and graphs, while Power BI presented the data in a structured tabular format. The custom Python Data Consumer (`python_data_consumer.py`) retrieved the data from the Azure SQL Database and generated line plots using `matplotlib` to visualize temperature and humidity trends over time.

The system ran efficiently and without any major issues, with data being continuously inserted into the Azure SQL Database. The successful insertion of data was verified by querying the SQL Database and viewing it in the data consumer tools. This pipeline demonstrates the scalability and flexibility of a serverless architecture with Azure's integration, seamlessly handling data ingestion, storage, and visualization.



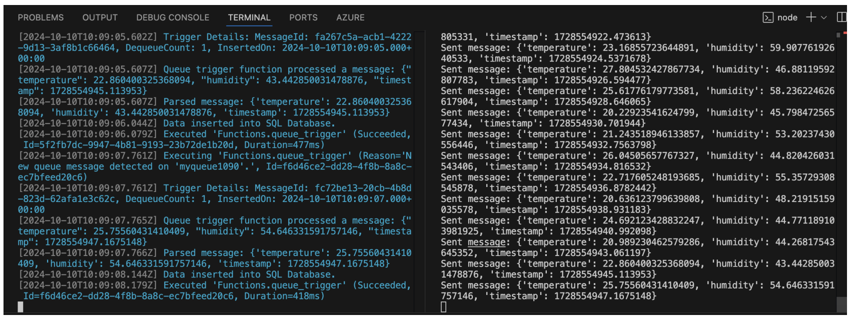
```
(env1) noraayaz@Noras-MacBook-Air Azure_Functions_Project % func azure functionapp publish myFunctionApp1090
Getting site publishing info...
[2024-10-10T16:34:31.407Z] Starting the function app deployment...
Creating archive for current directory...
Performing remote build for functions project.
```

Figure 10: Deployment command initiated via Azure CLI.



```
Number of device nodes 0
Number of fifo nodes 0
Number of socket nodes 0
Number of directories 854
Number of ids (unique uids + gids) 1
Number of uids 1
        root (0)
Number of gids 1
        root (0)
Creating placeholder blob for linux consumption function app...
SCM_RUN_FROM_PACKAGE placeholder blob scm-latest-myfunctionapp1090.zip located
Uploading built content /home/site/artifacts/functionappartifact.squashfs for linux consumption function app...
Resetting all workers for myfunctionapp1090.azurewebsites.net
Deployment successful. deployer = Push-Deployer deploymentPath = Functions App ZipDeploy. Extract zip. Remote build.
Remote build succeeded!
[2024-10-10T16:36:59.351Z] Syncing triggers...
Functions in myfunctionapp1090:
(env1) noraayaz@Noras-MacBook-Air Azure_Functions_Project %
```

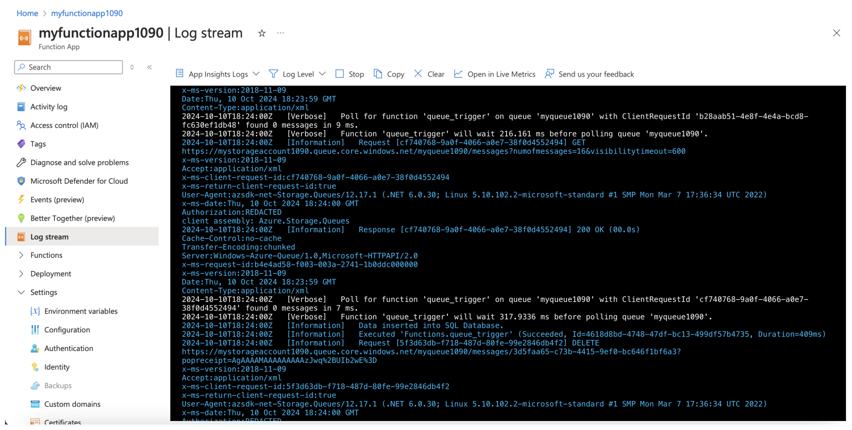
Figure 11: Successful deployment of Azure Function via Azure CLI.



```
[2024-10-10T10:09:05.602Z] Trigger Details: MessageId: fa267c5a-acb1-4222
-9d13-3af8b1c66464, DequeueCount: 1, InsertedOn: 2024-10-10T10:09:05.000+
00:00
[2024-10-10T10:09:05.607Z] Queue trigger function processed a message: {"
temperature": 22.860400325368094, "humidity": 43.442850031478876, "timest
amp": 1728554945.113953}
[2024-10-10T10:09:05.607Z] Parsed message: {'temperature': 22.86040032536
8094, 'humidity': 43.442850031478876, 'timestamp': 1728554945.113953}
[2024-10-10T10:09:06.044Z] Data inserted into SQL Database.
[2024-10-10T10:09:06.079Z] Executed 'Functions.queue_trigger' (Succeeded,
 Id=5f2fb7dc-9947-4b81-9193-23b72de1b20d, Duration=477ms)
[2024-10-10T10:09:07.761Z] Executing 'Functions.queue_trigger' (Reason='N
ew queue message detected on 'myqueue1090'.', Id=f6d46ce2-dd28-4f8b-8a8c-
ec7bfeed20c6)
[2024-10-10T10:09:07.761Z] Trigger Details: MessageId: fc72be13-20cb-4b8d
-823d-62afa1e3c62c, DequeueCount: 1, InsertedOn: 2024-10-10T10:09:07.000+
00:00
[2024-10-10T10:09:07.765Z] Queue trigger function processed a message: {"
temperature": 25.75560431410409, "humidity": 54.646331591757146, "timesta
mp": 1728554947.1675148}
[2024-10-10T10:09:07.766Z] Parsed message: {'temperature': 25.75560431410
409, 'humidity': 54.646331591757146, 'timestamp': 1728554947.1675148}
[2024-10-10T10:09:08.144Z] Data inserted into SQL Database.
[2024-10-10T10:09:08.179Z] Executed 'Functions.queue_trigger' (Succeeded,
 Id=f6d46ce2-dd28-4f8b-8a8c-ec7bfeed20c6, Duration=418ms)
```

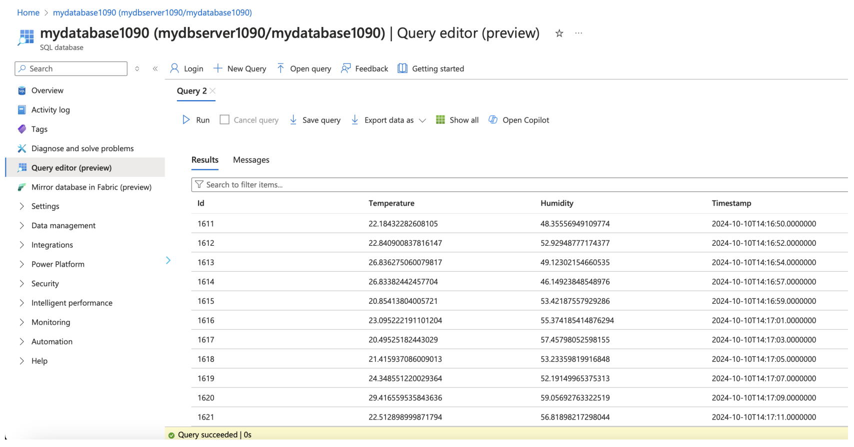
```
805331, 'timestamp': 1728554922.473613}
Sent message: {'temperature': 23.16855723644891, 'humidity': 59.907761926
40533, 'timestamp': 1728554924.5371678}
Sent message: {'temperature': 27.804532427867734, 'humidity': 46.88119592
807783, 'timestamp': 1728554926.594477}
Sent message: {'temperature': 25.61776179773581, 'humidity': 58.236224626
617904, 'timestamp': 1728554928.646065}
Sent message: {'temperature': 20.22923541624799, 'humidity': 45.798472565
77434, 'timestamp': 1728554930.701944}
Sent message: {'temperature': 21.243518946133857, 'humidity': 53.20237430
556446, 'timestamp': 1728554932.7563798}
Sent message: {'temperature': 26.04505657767327, 'humidity': 44.820426031
543406, 'timestamp': 1728554934.816532}
Sent message: {'temperature': 22.717605248193685, 'humidity': 55.35729308
545878, 'timestamp': 1728554936.8782442}
Sent message: {'temperature': 20.636123799639808, 'humidity': 48.21915159
035578, 'timestamp': 1728554938.931183}
Sent message: {'temperature': 24.692123428832247, 'humidity': 44.77118910
3981925, 'timestamp': 1728554940.992098}
Sent message: {'temperature': 20.989230462579286, 'humidity': 44.26817543
645352, 'timestamp': 1728554943.061197}
Sent message: {'temperature': 22.860400325368094, 'humidity': 43.44285003
1478876, 'timestamp': 1728554945.113953}
Sent message: {'temperature': 25.75560431410409, 'humidity': 54.646331591
757146, 'timestamp': 1728554947.1675148}
```

Figure 12: Logs from the local execution of the Azure Function, confirming successful data ingestion and processing during testing.



```
x-ms-version:2018-11-09
Date:Thu, 10 Oct 2024 18:23:59 GMT
Content-Type:application/xml
2024-10-10T18:24:00Z   [Verbose]   Poll for function 'queue_trigger' on queue 'myqueue1090' with ClientRequestId 'b28aab51-4e8f-4e4a-bcd8-
fc630ef1db48' found 0 messages in 9 ms.
2024-10-10T18:24:00Z   [Verbose]   Function 'queue_trigger' will wait 216.161 ms before polling queue 'myqueue1090'.
2024-10-10T18:24:00Z   [Information]   Request [cf740768-9a0f-4066-a0e7-38f0d4552494] GET
https://mystorageaccount1090.queue.core.windows.net/myqueue1090/messages?numofmessages=16&visibilitytimeout=600
x-ms-version:2018-11-09
Accept:application/xml
x-ms-client-request-id:cf740768-9a0f-4066-a0e7-38f0d4552494
x-ms-return-client-request-id:true
User-Agent:azsdk-net-Storage.Queues/12.17.1 (.NET 6.0.30; Linux 5.10.102.2-microsoft-standard #1 SMP Mon Mar 7 17:36:34 UTC 2022)
x-ms-date:Thu, 10 Oct 2024 18:24:00 GMT
Authorization:REDACTED
client assembly: Azure.Storage.Queues
2024-10-10T18:24:00Z   [Information]   Response [cf740768-9a0f-4066-a0e7-38f0d4552494] 200 OK (00.0s)
Cache-Control:no-cache
Transfer-Encoding:chunked
Server:Windows-Azure-Queue/1.0,Microsoft-HTTPAPI/2.0
x-ms-request-id:b4e4ad58-f003-003a-2741-1b0ddc000000
x-ms-version:2018-11-09
Date:Thu, 10 Oct 2024 18:23:59 GMT
Content-Type:application/xml
2024-10-10T18:24:00Z   [Verbose]   Poll for function 'queue_trigger' on queue 'myqueue1090' with ClientRequestId 'cf740768-9a0f-4066-a0e7-
38f0d4552494' found 0 messages in 7 ms.
2024-10-10T18:24:00Z   [Verbose]   Function 'queue_trigger' will wait 317.9336 ms before polling queue 'myqueue1090'.
2024-10-10T18:24:00Z   [Information]   Data inserted into SQL Database.
2024-10-10T18:24:00Z   [Information]   Executed 'Functions.queue_trigger' (Succeeded, Id=4618d8bd-4748-47df-bc13-499df57b4735, Duration=409ms)
2024-10-10T18:24:00Z   [Information]   Request [5f3d63db-f718-487d-80fe-99e2846db4f2] DELETE
https://mystorageaccount1090.queue.core.windows.net/myqueue1090/messages/3d5faa65-c73b-4415-9ef0-bc646f1bf6a3?
popreceipt=AgAAAAMAAAAAAAAAzJwq%2BUIb2wE%3D
x-ms-version:2018-11-09
Accept:application/xml
x-ms-client-request-id:5f3d63db-f718-487d-80fe-99e2846db4f2
x-ms-return-client-request-id:true
User-Agent:azsdk-net-Storage.Queues/12.17.1 (.NET 6.0.30; Linux 5.10.102.2-microsoft-standard #1 SMP Mon Mar 7 17:36:34 UTC 2022)
x-ms-date:Thu, 10 Oct 2024 18:24:00 GMT
```

Figure 13: Azure Portal Log Stream displaying real-time execution logs of the deployed Azure Function.



| Id | Temperature | Humidity | Timestamp |
|---|---|---|---|
| 1611 | 22.18432282608105 | 48.35556949109774 | 2024-10-10T14:16:50.0000000 |
| 1612 | 22.840900837816147 | 52.92948777174377 | 2024-10-10T14:16:52.0000000 |
| 1613 | 26.836275060079817 | 49.12302154660535 | 2024-10-10T14:16:54.0000000 |
| 1614 | 26.83382442457704 | 46.14923848548976 | 2024-10-10T14:16:57.0000000 |
| 1615 | 20.85413804005721 | 53.42187557929286 | 2024-10-10T14:16:59.0000000 |
| 1616 | 23.095222191101204 | 55.374185414876294 | 2024-10-10T14:17:01.0000000 |
| 1617 | 20.49525182443029 | 57.45798052598155 | 2024-10-10T14:17:03.0000000 |
| 1618 | 21.415937086009013 | 53.23359819916848 | 2024-10-10T14:17:05.0000000 |
| 1619 | 24.348551220029364 | 52.19149965375313 | 2024-10-10T14:17:07.0000000 |
| 1620 | 29.416559535843636 | 59.05692763322519 | 2024-10-10T14:17:09.0000000 |
| 1621 | 22.512898999871794 | 56.81898217298044 | 2024-10-10T14:17:11.0000000 |

Query succeeded | 0s

Figure 14: Query results from the Azure SQL Database showing successfully ingested sensor data.

Screenshot showing the data successfully inserted into the Azure SQL Database by the Azure Function, confirming that the end-to-end data flow was successful.

## 6 Cost Analysis

For the cost analysis, I compared the pricing of a 10TB Azure SQL Database (serverless tier) against Amazon RDS for SQL server, focusing on monthly costs for both storage and compute resources.

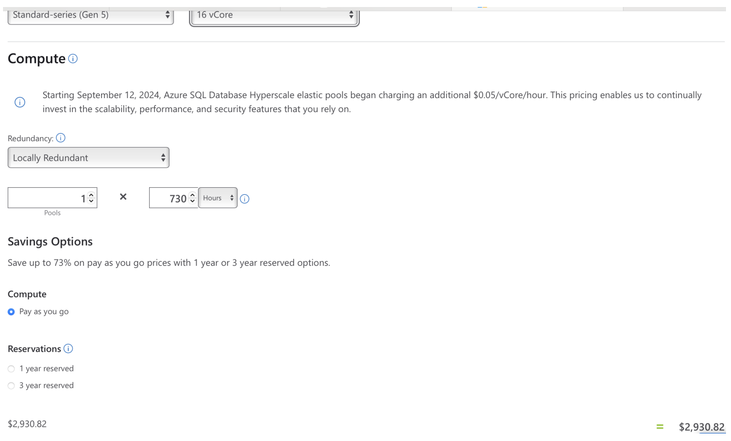
**Azure SQL Database (Serverless):**
**4 vCores:**
Compute (Pay-as-you-go): $732/month$
Storage (10TB): $2,648/month$
Total: $3,385/month(Pay-as-you-go)$
With 1-year reservation: $527/month(Compute), reducingthetotalto$3,175/month.

**16 vCores:**
Compute (Pay-as-you-go): $2,648/month$
Storage (10TB): $2,648/month$
Total: $5,583/month(Pay - as - you - go)$
With 1-year reservation: $2,109/month(Compute), reducingthetotalto$4,757/month.

**Amazon RDS for SQL server:**
**16 vCores with 64 GiB memory:**
Compute: $6,990/month$
Storage (10TB): $2,457/month$
Total: $9,448/month(Pay - as - you - go)$.

**Comparison and Conclusion:**

Azure SQL Database offers a significantly lower total cost compared to AWS RDS for similar compute and storage configurations. Using Azure's reserved pricing model further reduces costs, making Azure the more economical choice, especially for long-term use. Backup storage costs were minimal in both cases, with Azure offering a slightly cheaper option ($4.46/monthforAzurevs$0.45/month for AWS). Overall, Azure SQL Database provides a more cost-effective solution with better pricing for reserved compute instances, particularly for high storage scenarios like a 10TB database.


Standard-series (Gen 5) | 16 vCore

Compute

Starting September 12, 2024, Azure SQL Database Hyperscale elastic pools began charging an additional $0.05/vCore/hour. This pricing enables us to continually invest in the scalability, performance, and security features that you rely on.

Redundancy: Locally Redundant

1 Pools × 730 Hours

Savings Options

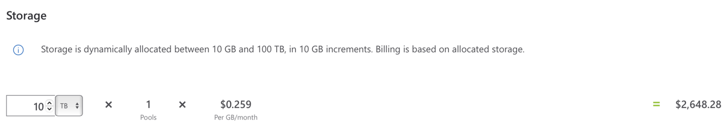
Save up to 73% on pay as you go prices with 1 year or 3 year reserved options.

Compute
[✓ Pay as you go]

Reservations
1 year reserved
3 year reserved

$2,930.82 = $2,930.82

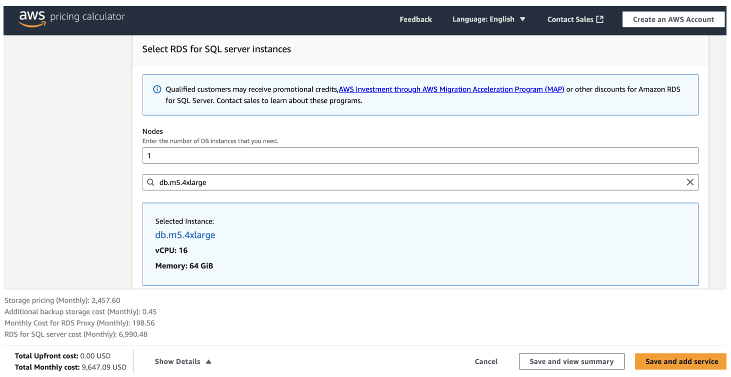
Figure 15: Azure SQL Database 16 vCores Estimated Cost


Storage

Storage is dynamically allocated between 10 GB and 100 TB, in 10 GB increments. Billing is based on allocated storage.

10 TB × 1 Pools × $0.259 Per GB/month = $2,648.28

Figure 16: Azure SQL Database Storage Estimated Cost



Select RDS for SQL server instances

Qualified customers may receive promotional credits, AWS Investment through AWS Migration Acceleration Program (MAP) or other discounts for Amazon RDS for SQL Server. Contact sales to learn about these programs.

Nodes
Enter the number of DB instances that you need.
1

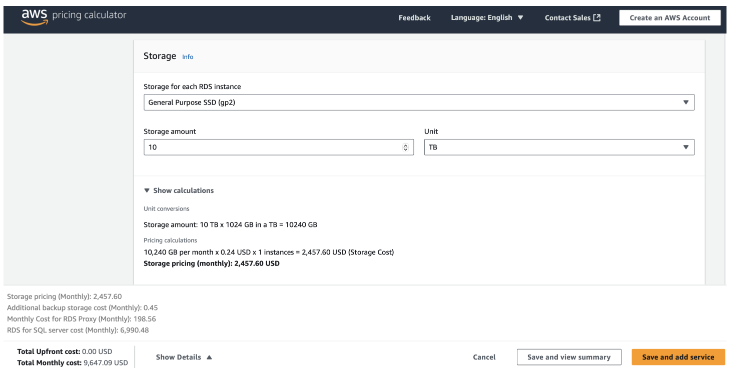
db.m5.4xlarge

Selected Instance:
db.m5.4xlarge
**vCPU: 16**
**Memory: 64 GiB**

Storage pricing (Monthly): 2,457.60
Additional backup storage cost (Monthly): 0.45
Monthly Cost for RDS Proxy (Monthly): 198.56
RDS for SQL server cost (Monthly): 6,990.48

**Total Upfront cost:** 0.00 USD
**Total Monthly cost:** 9,647.09 USD

Figure 17: Amazon RDS for SQL server Estimated Cost



Storage Info

Storage for each RDS instance
General Purpose SSD (gp2)

Storage amount
10

Unit
TB

Show calculations

Unit conversions
Storage amount: 10 TB x 1024 GB in a TB = 10240 GB

Pricing calculations
10,240 GB per month x 0.24 USD x 1 instances = 2,457.60 USD (Storage Cost)
**Storage pricing (monthly): 2,457.60 USD**

Storage pricing (Monthly): 2,457.60
Additional backup storage cost (Monthly): 0.45
Monthly Cost for RDS Proxy (Monthly): 198.56
RDS for SQL server cost (Monthly): 6,990.48

**Total Upfront cost:** 0.00 USD
**Total Monthly cost:** 9,647.09 USD

Figure 18: Amazon RDS for SQL server storage Estimated Cost

## 7 Conclusion

In this project, I successfully implemented a cloud-based data pipeline using Azure services. The system efficiently handled data production, queuing, and storage, with real-time sensor data being ingested from a Python-based producer into an Azure Queue. This data was then processed by an Azure Function, which inserted it into an Azure SQL Database, enabling serverless, automated data ingestion. The stored data was visualized using Grafana and Power BI, providing meaningful insights through real-time dashboards and reports. Additionally, a custom Python data consumer was developed to visualize the data using `matplotlib`.

The project demonstrated the benefits of a serverless architecture for scalability, flexibility, and cost-effectiveness. By leveraging Azure Functions, Azure SQL Database, and data visualization tools, we ensured secure, reliable, and real-time data flow from generation to consumption. This implementation highlights how cloud-based solutions can simplify the management of data pipelines while ensuring efficient processing and visualization of data.

## 8 Challenges and Solutions

Throughout the implementation of this cloud-based data pipeline, I encountered several challenges, each contributing to valuable learning experiences. Below are the key difficulties I faced and the solutions I applied:

## 8.1 Data Generation and Queue Poison Issues

While developing the `send_to_queue.py` script to generate and send data to the Azure Queue, I initially faced a problem where the generated data was repeatedly sent to the **poison queue** instead of the target queue. This issue was quite challenging and time-consuming, as I struggled to understand the root cause. After extensive research, I came across a YouTube video that explained the problem was related to how Azure handles encoding for queue messages. The solution involved explicitly setting the message encoding and decoding policies with the following lines of code:

```
queue_client._message_encode_policy = BinaryBase64EncodePolicy()
queue_client._message_decode_policy = BinaryBase64DecodePolicy()
```

Once I implemented these policies, the data was successfully sent to the target queue.

## 8.2 Azure Function Deployment

Deploying the Azure Function was another significant challenge, as I had no prior experience with function deployment on the Azure platform. Initially, I struggled to understand how to deploy the function both locally and to the cloud. With the help of ChatGPT, I learned how to deploy the function using both the **Azure CLI** and **local deployment** methods. This guidance helped me navigate the Azure portal and correctly deploy my function, allowing it to process messages from the queue and insert them into the SQL database.

## 8.3 Database Connectivity Issues

One of the most frustrating challenges was related to transferring data from the Azure Function to the SQL database. Initially, I attempted to use the **ODBC driver** to connect to the database, but encountered compatibility issues. Since Azure Function Apps in Python are Linux-based, the ODBC driver was not supported in my environment, leading to repeated connection failures. I even attempted using Docker as an alternative, but that didn't resolve the issue either.

After more troubleshooting and assistance from ChatGPT, I was advised to switch from the ODBC driver to the **pymssql** library. After restructuring my code to use **pymssql**, I was finally able to connect to the Azure SQL Database successfully, resolving the connectivity issues.

## 8.4 Learning and Persistence

Each of these challenges required substantial research and persistence. From figuring out how to send messages to the correct queue to deploying the Azure Function and handling database connectivity, every step of this project taught me something new. While these obstacles consumed a significant amount of time, they also provided me with a deep understanding of how a data pipeline works in Azure, from data generation to ingestion, storage, and visualization. The experience of troubleshooting and learning from multiple sources, including ChatGPT, enhanced my technical knowledge and problem-solving skills.

## 9 Final Thoughts

This project provided a valuable hands-on experience in designing and implementing a serverless data pipeline using Azure cloud services. Each phase of the project, from setting up the environment to overcoming technical challenges, contributed to a deeper understanding of how cloud platforms can handle real-time data processing and visualization.

By working through the issues with queuing, database connectivity, and function deployment, I developed not only technical skills but also resilience and problem-solving abilities. The ability to research and troubleshoot effectively, coupled with leveraging tools like ChatGPT, allowed me to complete the pipeline and understand its full lifecycle from data generation to consumption.

This project reinforced the importance of serverless architecture for scalability, flexibility, and cost-efficiency, and it equipped me with practical knowledge of how to integrate multiple Azure components into a cohesive system. These skills will be invaluable in future cloud-based projects.