Cloud Project: Data Pipeline

1. Introduction

This report details the process of setting up a cloud-based data pipeline utilizing Microsoft Azure services, specifically focusing on deploying a Virtual Machine (VM), establishing a queue storage system, processing data, and utilizing monitoring tools like Grafana. The pipeline simulates real-time data ingestion and storage, which can be extended for further analytical and monitoring purposes.

2. Virtual Machine Setup

**2.1. VM Creation**

A Virtual Machine (VM) was created on Microsoft Azure. The VM chosen for this project runs **Ubuntu**, and it was accessed using SSH (Secure Shell) from a local machine.

1. **Public and Private Key Generation:** A public-private key pair was generated using the following command:

*ssh-keygen -t rsa -b 4096 -C "Misbah" -N "" -f ~/.ssh/id\_rsa*

The public key was used during the VM creation in Azure to enable secure login.

1. **VM Login:** Once the VM was set up, the public IP address of the machine was used to log into the VM through Bash or PowerShell.

*ssh azureuser@[public\_ip\_address]*

**2.2. VM Configuration**

After logging into the VM, the following commands were executed to update the system, install

*sudo apt update  
sudo apt -y upgrade  
sudo apt -y autoremove  
sudo apt -y install nano wget curl htop tmux net-tools  
pip install azure storage queue*

This ensured that the VM was up to date and had the required tools for further tasks.

3. Azure Storage Setup

**3.1. Storage Account Creation**

A **Storage Account** was created in Azure to serve as a destination for the data being generated in this pipeline. A **Queue Storage** system was established, which can store data in the form of messages.

**3.2. Data Ingestion**

Data was simulated as sensor readings and then sent to the queue storage system using Python code. The **azure.storage.queue** library was used to establish a connection to the Azure queue. Here’s an example of the code used for this connection and data sending:

import os

import random

from datetime import datetime

from azure.storage.queue import QueueClient, BinaryBase64EncodePolicy

from dotenv import load\_dotenv

load\_dotenv()

def generate\_sample\_data():

data = {

'id': random.randint(1, 100),

'timestamp': datetime.utcnow().isoformat(),

'temperature': random.uniform(20.0, 30.0),

'humidity': random.uniform(30.0, 60.0)

}

return data

# Connection to Queue

queue\_client = QueueClient.from\_connection\_string(

os.getenv('AZURE\_STORAGE\_CONNECTION\_STRING'),

os.getenv('AZURE\_QUEUE\_NAME'),

message\_encode\_policy=BinaryBase64EncodePolicy()

)

def send\_data\_to\_queue():

try:

for \_ in range(10):

data = generate\_sample\_data()

message = json.dumps(data)

queue\_client.send\_message(message)

print(f"Data sent: {message}")

except Exception as e:

print(f"Error sending data: {str(e)}")

This script generates random sensor data (temperature and humidity readings) and sends them to the Azure Queue Storage.

4. Database Setup

**4.1. Database Creation**

A SQL-based database was created to store the ingested data for further analysis. The following query was used to create the table structure:

CREATE TABLE dbo.SensorData (

id INT PRIMARY KEY NOT NULL,

timestamp DATETIME NOT NULL,

temperature FLOAT NOT NULL,

humidity FLOAT NOT NULL

);

This table can be populated with the data from the Azure queue as it arrives, allowing for future querying and analysis.

**4.2. User Permissions**

To manage database access securely, a user named ApplicationUser was created, and specific roles were assigned:

CREATE USER ApplicationUser

WITH PASSWORD = 'password';

ALTER ROLE db\_datareader ADD MEMBER ApplicationUser;

ALTER ROLE db\_datawriter ADD MEMBER ApplicationUser;

This ensured that the user had permission to read and write data to the database.

**5. Logic App (**Automating Data Insertion with Azure Logic App**)**

To automate data transfer from Azure Queue Storage to the SQL Database, an Azure Logic App was created. The Logic App triggers when new data arrives in the queue and inserts it into the SQL Database automatically.

Steps:

* Create Logic App: In Azure, create a Logic App and set the trigger to activate when a message is received from the Azure Queue Storage.
* Parse Data: Add a Parse JSON action to process the message, extracting fields like id, timestamp, temperature, and humidity.
* Insert Data into SQL: Use the Insert row action to map the parsed fields to the corresponding columns in the SensorData table of the SQL Database.
* Test and Monitor: Test the workflow with sample data to ensure proper functioning. Azure’s built-in monitoring tools help track execution and troubleshoot any issues.

This setup fully automates the data pipeline, ensuring real-time updates to the SQL Database with minimal manual intervention.

6. Monitoring and Visualization

**6.1. Grafana Setup**

Grafana was chosen as the monitoring tool to visualize the data stored in the database. The following commands were used to install and configure Grafana on the VM:

sudo apt update

sudo apt upgrade -y

sudo apt install -y software-properties-common

sudo add-apt-repository ppa:grafana/Grafana

sudo apt update

sudo apt install -y grafana

sudo systemctl status grafana-server

sudo systemctl enable grafana-server

sudo systemctl start grafana-server

To access Grafana, installed SQL plugins and allowed port 3000 to have Grafan

sudo grafana-cli plugins install grafana-sqlserver-datasource

sudo systemctl restart grafana-server

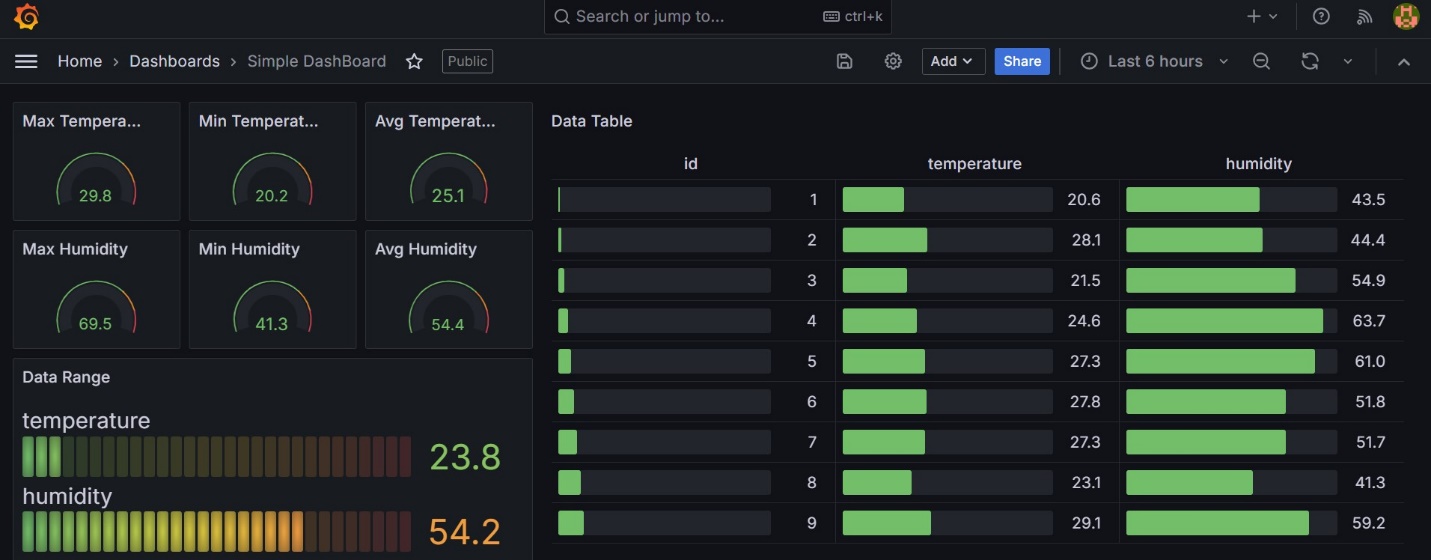
sudo apt install net-tools

sudo ufw allow 3000/tcp

Then used Grafana through the following address

http://[my\_IP\_address]:3000/

Once the server was running, it was linked to the database, allowing real-time visualization of the sensor data, including metrics like temperature and humidity trends.



**6.2. Power BI (Locally)**

Alternatively we use Power BI to show visualization of the data.

The Data was imported from the SQL server directly using the authentication and access key and the data was directly accessible to be transformed to our desired visualization.

Following is the sample of the visualization of the data

**A screenshot of a computer

Description automatically generated**

**6.3 Report on Creating a Sensor Data Visualization Webpage**

the creation of a web application that visualizes sensor data using Flask and Chart.js, fetching data from an Azure SQL Database.

Installed necessary packages using Bash:

```bash  
pip install flask pyodbc python-dotenv  
```

Created a Flask application to connect to the Azure SQL Database and retrieve sensor data.

```python  
import os  
from flask import Flask, render\_template, jsonify  
import pyodbc  
from dotenv import load\_dotenv  
  
load\_dotenv()  
app = Flask(\_\_name\_\_)  
  
# SQL Server connection parameters  
sql\_server = os.getenv('SQL\_SERVER')  
sql\_database = os.getenv('SQL\_DATABASE')  
sql\_username = os.getenv('SQL\_USERNAME')  
sql\_password = os.getenv('SQL\_PASSWORD')  
sql\_driver = os.getenv('SQL\_DRIVER')  
  
def create\_sql\_connection():  
 connection\_string = (  
 f"DRIVER={sql\_driver};"  
 f"SERVER=tcp:{sql\_server},1433;"  
 f"DATABASE={sql\_database};"  
 f"UID={sql\_username};"  
 f"PWD={sql\_password};"  
 f"Encrypt=yes;"  
 f"TrustServerCertificate=no;"  
 f"Connection Timeout=30;"  
 )  
 return pyodbc.connect(connection\_string)  
  
@app.route('/data')  
def data():  
 conn = create\_sql\_connection()  
 cursor = conn.cursor()  
 cursor.execute("SELECT TOP 20 timestamp, temperature, humidity FROM [dbo].[SensorData] ORDER BY timestamp DESC")  
 data = cursor.fetchall()  
 conn.close()  
 return jsonify(data)  
```

Developed the frontend to visualize data using index.html.

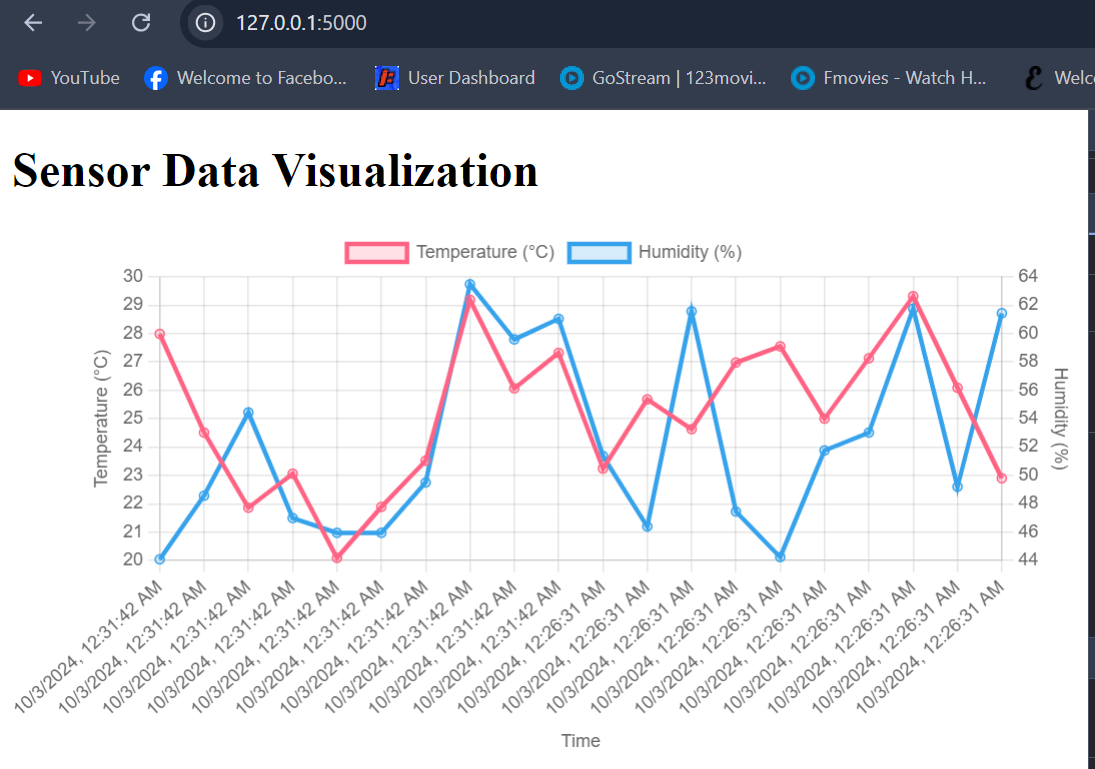
```html  
<!DOCTYPE html>  
<html lang="en">  
<head>  
 <meta charset="UTF-8">  
 <meta name="viewport" content="width=device-width, initial-scale=1.0">  
 <title>Sensor Data Visualization</title>  
 <script src="https://cdn.jsdelivr.net/npm/chart.js"></script>  
</head>  
<body>  
 <h1>Sensor Data Visualization</h1>  
 <canvas id="sensorChart" width="800" height="400"></canvas>  
 <script>  
 fetch('/data')  
 .then(response => response.json())  
 .then(data => {  
 const ctx = document.getElementById('sensorChart').getContext('2d');  
 new Chart(ctx, {  
 type: 'line',  
 data: {  
 labels: data.map(item => new Date(item.timestamp).toLocaleString()),  
 datasets: [  
 {  
 label: 'Temperature (°C)',  
 data: data.map(item => item.temperature),  
 borderColor: 'rgb(255, 99, 132)',  
 backgroundColor: 'rgba(255, 99, 132, 0.2)',  
 },  
 {  
 label: 'Humidity (%)',  
 data: data.map(item => item.humidity),  
 borderColor: 'rgb(54, 162, 235)',  
 backgroundColor: 'rgba(54, 162, 235, 0.2)',  
 }  
 ]  
 }  
 });  
 });  
 </script>  
</body>  
</html>  
```

Created a `.env` file for sensitive information:

```  
SQL\_SERVER=misbahserver.database.windows.net  
SQL\_DATABASE=My\_database  
SQL\_USERNAME=Application\*\*\*\*  
SQL\_PASSWORD=\*\*\*\*\*\*  
SQL\_DRIVER={ODBC Driver 17 for SQL Server}  
```

Executed the application:

```bash  
python3 app.py  
```

****

7. Cost Comparison Report:   
Azure SQL Database (Serverless) vs. Azure VM with PostgreSQL

**Technical Cost Analysis for 10TB Database**

|  |  |
| --- | --- |
| **1. Azure SQL Database (Serverless)**  #### Compute Costs:  - vCore range: 0.5 - 40 vCores  - Pricing: $0.5218 per vCore-hour  - Estimated average usage: 20 vCores  - Monthly compute cost: 20 vCores \* $0.5218 \* 730 hours ≈ $7,618  #### Storage Costs:  - 10TB at $0.115 per GB/month  - Monthly storage cost: 10,240 GB \* $0.115 ≈ $1,178  #### Backup Storage:  - Assumed 100% of DB size  - 10TB at $0.11 per GB/month  - Monthly backup cost: 10,240 GB \* $0.11 ≈ $1,126 | **2. Azure VM with PostgreSQL**  #### Compute Costs:  - VM Size: D16s v3 (16 vCPUs, 64 GB RAM)  - Pricing: $0.768 per hour  - Monthly compute cost: $0.768 \* 730 hours ≈ $560  #### Storage Costs:  - Premium SSD: P60 (8,192 GB) + P50 (4,096 GB)  - P60 cost: $1,098.24/month  - P50 cost: $571.52/month  - Total storage cost: $1,669.76/month  #### Backup and Transaction Log Storage:  - 10TB at $0.02 per GB/month (Azure Blob Storage)  - Monthly backup cost: 10,240 GB \* $0.02 ≈ $204.80  #### Networking:  - Outbound data transfer: Assumed 5TB/month  - First 5GB free, then $0.087 per GB  - Monthly networking cost: (5120 GB - 5 GB) \* $0.087 ≈ $444.31  #### OS and Management:  - Azure Monitor: $15/month  - OS support: Included in VM price |
| \*\*Total Estimated Monthly Cost: $9,922\*\* | \*\*Total Estimated Monthly Cost: $2,893.87\*\* |

While the Azure VM with PostgreSQL appears more cost-effective based on raw numbers, the Azure SQL Database (Serverless) option offers significant technical advantages in terms of automatic scaling, built-in HA/DR, and reduced management overhead. The choice between these options should be based on specific workload characteristics, required features, and available technical expertise for database management.

**8. Conclusion**

This cloud-based data pipeline project demonstrates how various Microsoft Azure services can be integrated to create an efficient and scalable system for real-time data ingestion, processing, and monitoring. The pipeline was built by deploying an Ubuntu-based Virtual Machine (VM), creating Azure Queue Storage for data ingestion, and setting up a SQL database for long-term storage of sensor data.

Automation was a key aspect of this pipeline, achieved through the use of Azure Logic Apps. The Logic App was configured to trigger when new data was added to the Azure Queue, automatically inserting it into the SQL database without requiring manual intervention. This workflow ensures data is continuously updated in real time, making the system more responsive and reducing the likelihood of delays or errors.

To monitor the data, Grafana was configured to visualize metrics like temperature and humidity stored in the SQL database. This real-time visualization allows users to track data trends and system performance. Power BI was also considered for local data visualization, offering an alternative method for displaying insights.

Finally, a cost comparison analysis was performed between using Azure SQL Database (Serverless) and Azure Virtual Machines with PostgreSQL, highlighting the trade-offs between cost-effectiveness and technical benefits such as automatic scaling and reduced management overhead.

Overall, this project provides a scalable and automated pipeline for managing and processing large volumes of data, with potential for further expansion into machine learning, advanced analytics, or more sophisticated alerting systems. The infrastructure is suitable for real-time data applications, ensuring seamless data flow from ingestion to storage and visualization.