RBB2013: Digital Twin - May 2025

Ts Dr Ho Tatt Wei

Name	ID	Program
Oussama Shebaro	22004122	Computer Engineering
Mohammed Adel Sinnokrot	22005106	Computer Engineering
Abdullah Shahid Ali	22010729	Computer Engineering
Hafdoon Muhammed	22010898	Computer Engineering

Table Of Contents

Introduction

Nowadays, in the automotive industry, electric vehicles (EVs) are relying more on sensor data to monitor internal systems, ensure safety, and optimize performance. With the rapid spread of Internet of Things (IoT) technologies, it has become possible to collect real-time data from various EV components such as battery health, motor condition, brake wear, and tire pressure. However, simply collecting this data is not enough as there is also a growing need to analyze it in real time to predict faults before they occur. This is where predictive maintenance, supported by machine learning and cloud computing, becomes highly valuable.

This project is a digital twin that helps replicate that real life scenario which could cost hundreds of thousands of dollars if we want to test out with real vehicles, instead we simulate the predictions virtually as this minimizes costs a ton. Our project focuses on building a real-time predictive maintenance system for electric vehicles using AWS cloud services. The system simulates the behavior of an actual EV by sending sensor data at fixed intervals and feeding it into a digital twin environment where predictive analytics are applied. The solution integrates several AWS services including Lambda, S3, IoT SiteWise, IoT TwinMaker, IoT Events, Systems Manager, and EventBridge. The main goal is to use a trained deep learning model to predict the vehicle's Time To Failure (TTF) and classify the likelihood of incoming failure based on live data.

By using AWS IoT TwinMaker and Grafana, the system also provides us with an interactive dashboard that displays both the raw sensor data and the predictive insights. This enables proactive monitoring and decision-making. Not only does this implementation replicate real-world IoT deployment, but it also demonstrates how predictive analytics can be applied to actually improve vehicle reliability, reduce downtime, and improve overall operational efficiency.

Objective and Outcome

Objective

The main objective of this project is to design and implement a scalable, serverless architecture that simulates real-time electric vehicle sensor data and performs predictive maintenance using machine learning. To be more specific, the system should be able to:

- Simulate the streaming of time-series EV sensor data in 15-minute intervals using pre-recorded datasets stored in Amazon S3.
- Store and manage real-time telemetry in AWS IoT SiteWise to enable structured access to the sensor readings.
- Automatically trigger a machine learning inference pipeline whenever new data is sent, using AWS Lambda, Systems Manager, and EventBridge.
- Apply a trained machine learning model (Random Forest) to perform regression and classification tasks:
    - Predict the Time To Failure (TTF) of the electric vehicle using stored sensor values
    - Classify whether the vehicle is likely to fail soon, based on whether TTF falls below a critical threshold (48 hours)
- Display current and predicted data within a 3D digital twin model in AWS IoT TwinMaker and Grafana dashboards.

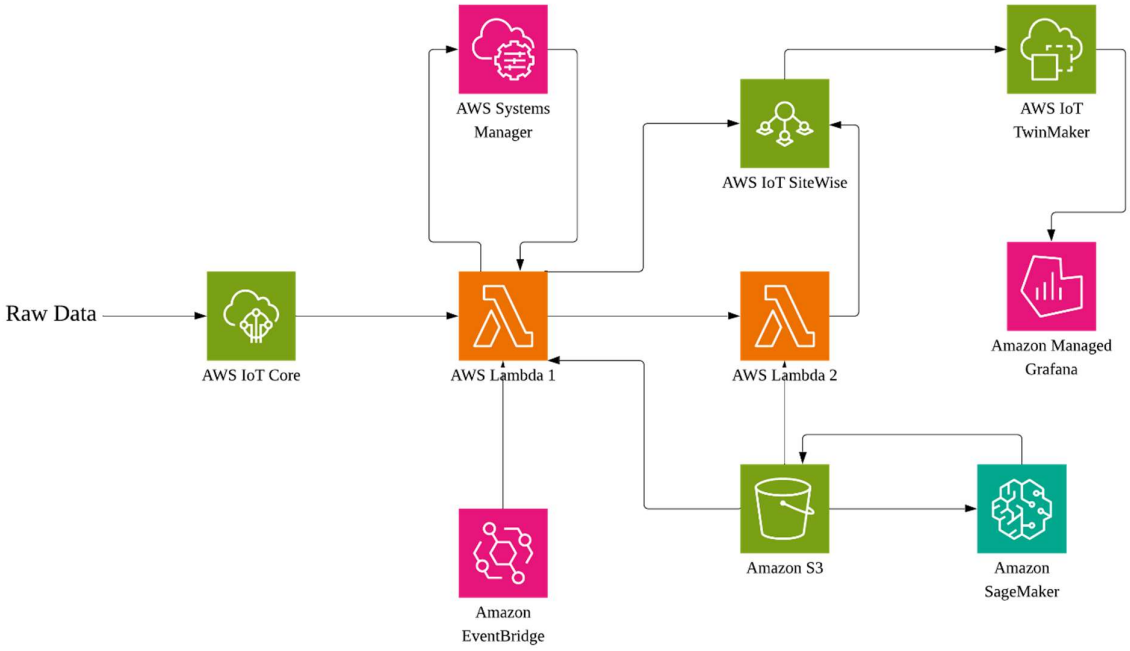
Expected Outcome

By the completion of this project, the following outcomes are expected:

- A functional simulation pipeline that processes new incoming EV sensor data every 15 minutes and pushes it to AWS IoT SiteWise.
- An automated trigger mechanism that detects when a prediction should be made and executes the machine learning model on the latest data.
- Accurate TTF predictions and binary failure classification output sent back to IoT SiteWise for visualization and monitoring.
- A fully integrated digital twin visualization of the electric vehicle system using AWS IoT TwinMaker, fed with both live sensor readings and prediction data.
- A Grafana dashboard that provides real-time insights into sensor health, TTF trends, and potential failure warnings.

This system demonstrates how IoT data, serverless computing, and machine learning can be integrated into a unified predictive maintenance solution for smart mobility applications.

System Architecture Diagram



The architecture diagram represents the complete data and control flow within the predictive maintenance system. It shows how we integrate several AWS services to facilitate real-time data ingestion, storage, processing, and visualization.

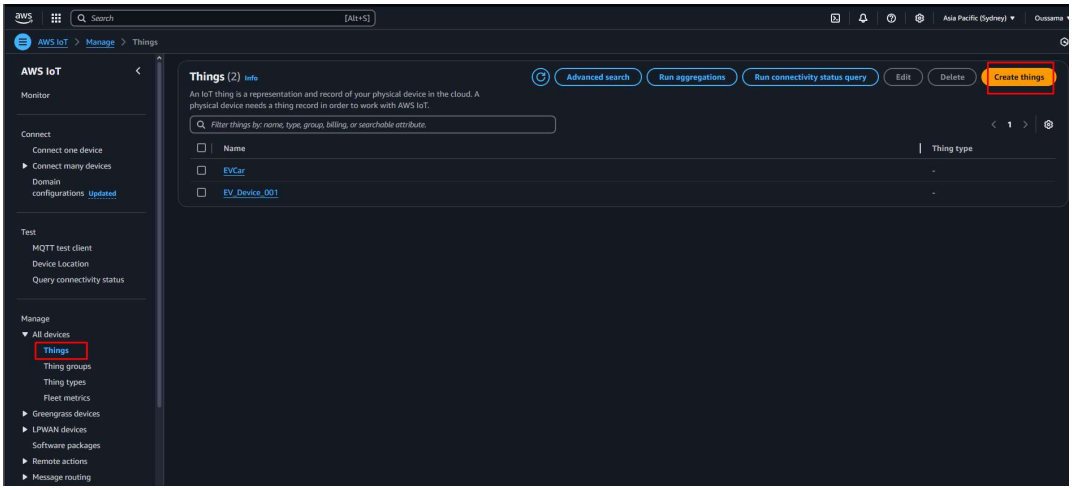
1. **Amazon S3** acts as the data source and model storage layer. It contains two primary assets: a CSV file with pre-recorded EV sensor data (daily_user.csv, moderate_user.csv, rare_user.csv, heavy_user.csv) and the trained machine learning model script (train_script.py).
2. **Amazon EventBridge** is configured to trigger a Lambda function every 15 minutes. This Lambda simulates a real-time device by reading the next row of the CSV file from S3 and preparing it for submission to the monitoring system.
3. The **first AWS Lambda function** is responsible for:
    - Reading a single row of sensor data from the CSV.
    - Sending the data to **AWS IoT SiteWise**, which serves as the time-series database for storing telemetry from simulated EV components.
    - Setting a boolean flag (TriggerPrediction = true) in SiteWise to indicate that a prediction should be executed.
    - Updating the row index in **AWS Systems Manager Parameter Store**, ensuring that each Lambda invocation sends the next row in sequence.

4. **AWS Systems Manager (SSM)** is used to store the current row index after each time the Lambda is executed. Since Lambda is a stateless service, storing this index ensures that the simulation progresses in a controlled and trackable manner.
5. **AWS IoT SiteWise** functions as the central data repository. It receives the sensor data, stores the TriggerPrediction flag, and holds the output predictions once they are generated. It provides a structured interface for querying historical and real-time values.
7. The **second AWS Lambda function** performs the machine learning prediction. It:
    a. Fetches the most recent values from SiteWise, including the last 24-time steps if required by the model.
    b. Calls the SageMaker endpoint to do the prediction.
    c. Performs preprocessing and inference to generate a PredictedTTF value (a double) and a FailureSoon flag (a boolean classification).
    d. Sends the prediction results back to SiteWise.
    e. Resets the TriggerPrediction flag to false to indicate that the prediction is complete.
8. **AWS SageMaker** is used for model deployment, after being triggered by the second Lambda to do the prediction.
9. **AWS IoT TwinMaker** reads from SiteWise and provides a 3D visualization of the digital twin of the electric vehicle system. It allows real-time monitoring of both raw and predicted data in the context of the vehicle's structure.
10. **Amazon Managed Grafana** connects directly to TwinMaker to display a dashboard that shows live sensor trends, predicted time to failure, and alert conditions such as imminent failure warnings. This serves as a monitoring tool for users or maintenance personnel.

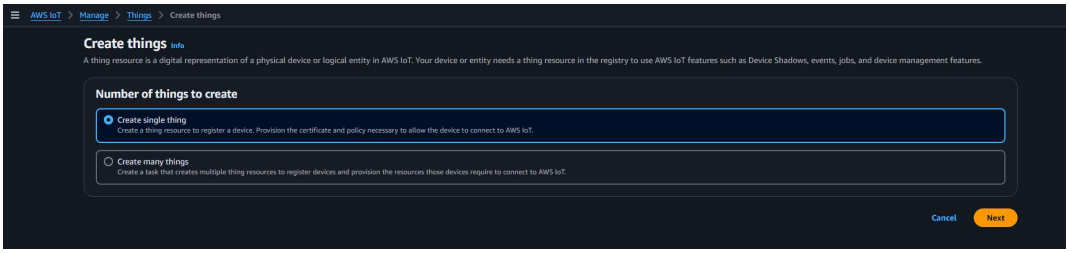
The architecture is modular, scalable, and serverless, making it suitable for real-time digital twin simulations and predictive maintenance in smart mobility and industrial applications.

IoT Integration

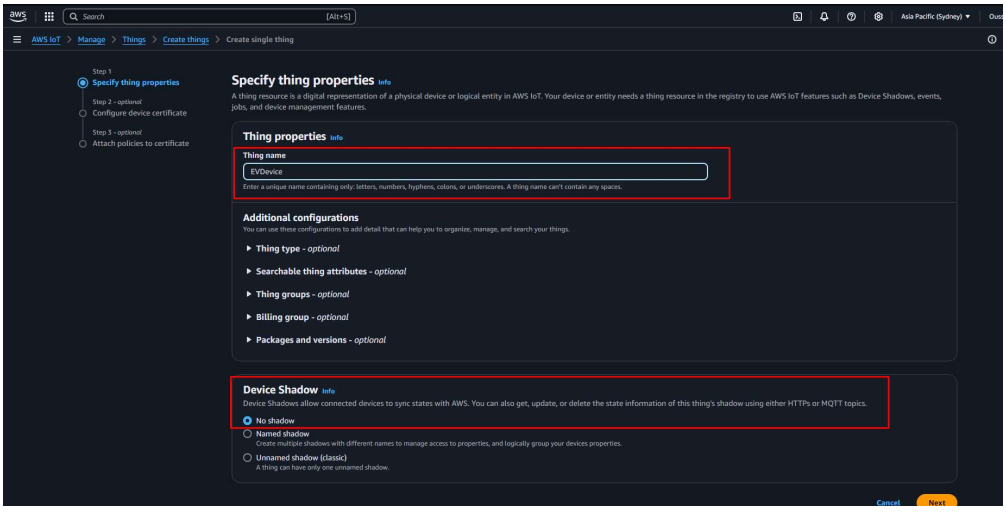
1. First, head to IoT Core dashboard, then click on Manage -> All Devices -> Things
2. Click on Create Things



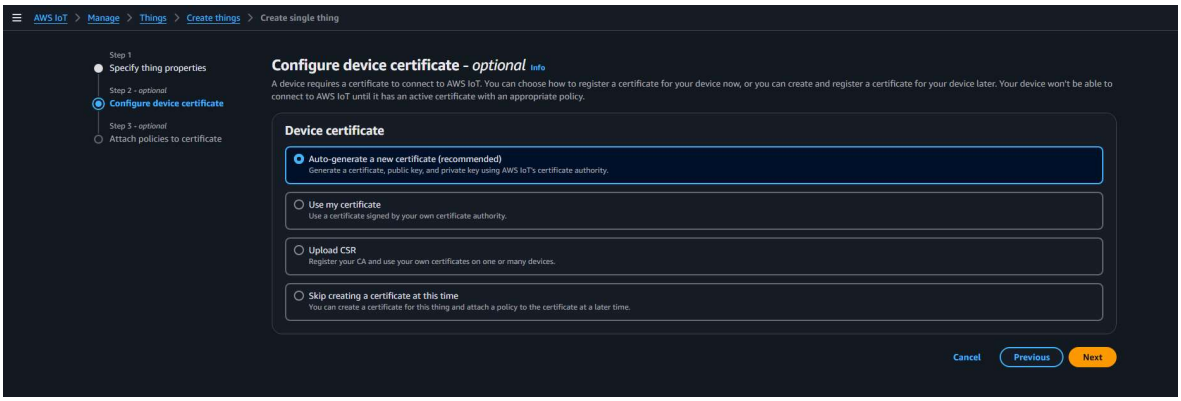
3. Click on Create single thing



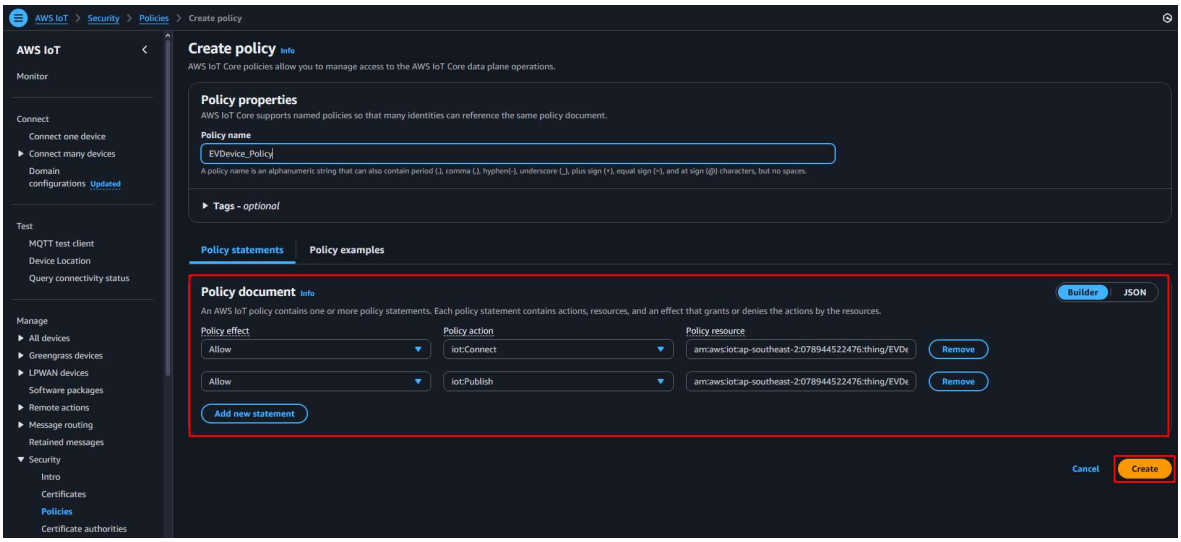
4. Give your Thing a name, make sure “No shadow” selected



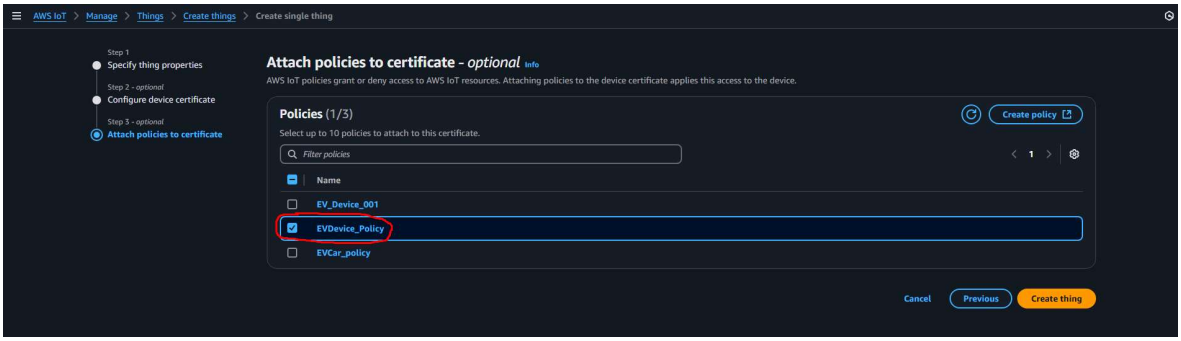
5. After clicking on Next, you will see this page where you will auto generate a new certificate



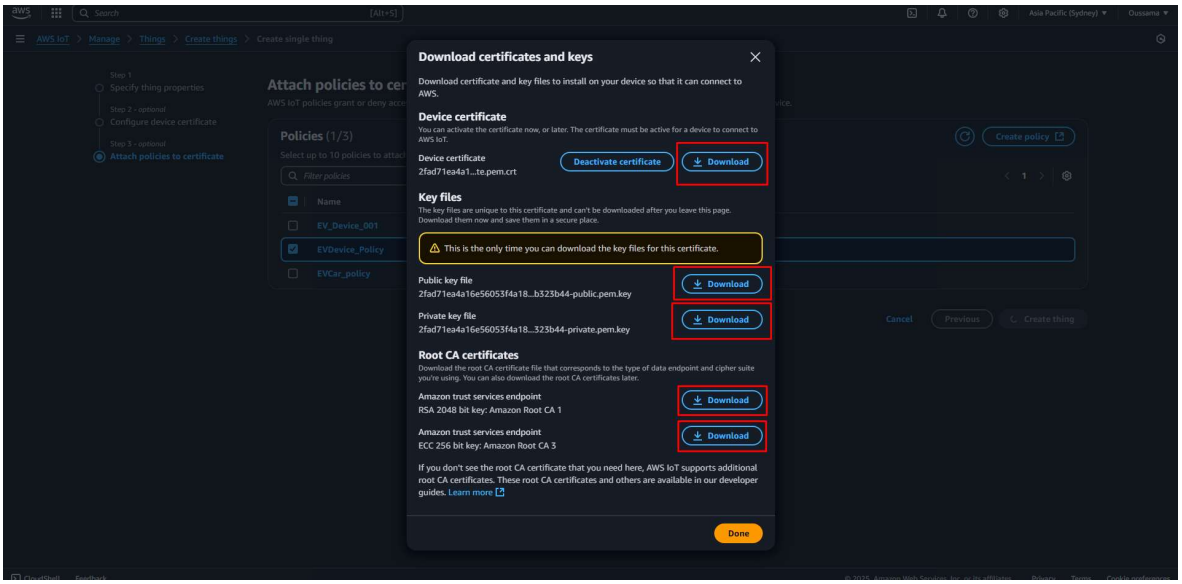
6. After you hit next, you have to create a policy for your thing



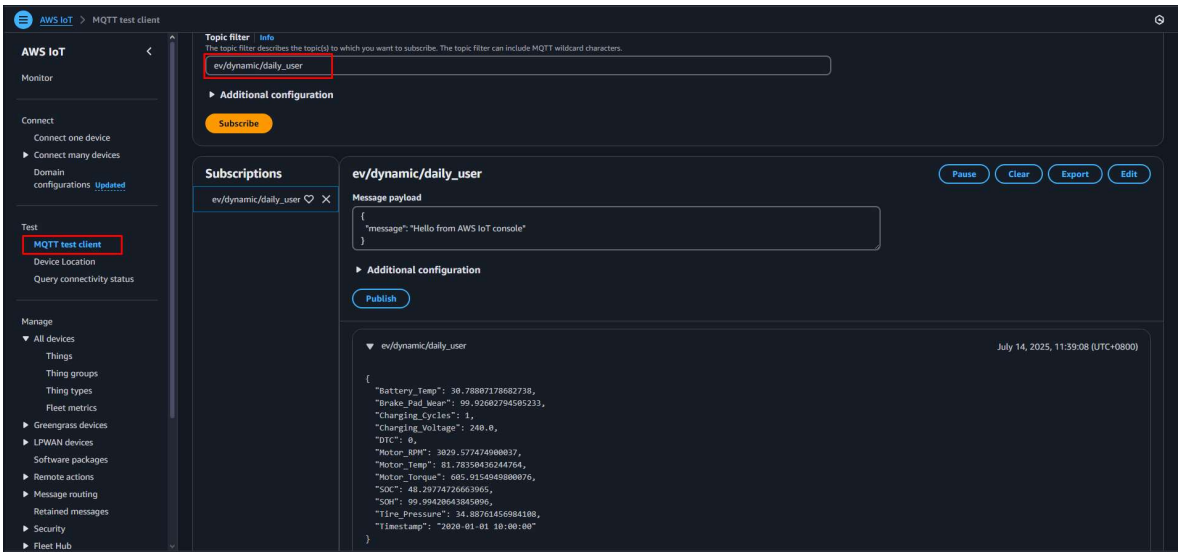
7. After clicking create, you will be redirected to the page we were on earlier, where we can select the policy, we just created to attach it to our Thing.



8. Now after you hit Create Thing, you will be shown a popup message with important certificates and key files you must download in this step, as keys won't be available later.



9. After creating a Thing, we can go to Test -> MQTT Test Client
10. You can now subscribe to the topic by adding a name that we will use in our Lambda code later to simulate data streams to MQTT.
11. As you can see in the image below, we are publishing the daily_user data



12. Lambda code used to publish to MQTT

```
import json
import boto3
import pandas as pd
import time
from datetime import datetime
import pytz

# AWS Clients
s3 = boto3.client('s3')
client = boto3.client('iot-data', region_name='ap-southeast-2')

# S3 details
bucket = 'ev-iot-dataset'
key = 'daily_user.csv'


# ==============================
# Lambda Handler
# ==============================
def lambda_handler(event, context):
    try:
        publish_csv_to_mqtt()
        return {'statusCode': 200, 'body': json.dumps('Daily user data
published to MQTT')}
    except Exception as e:
        print(f"[ERROR] Lambda error: {e}")
        return {'statusCode': 500, 'body': json.dumps(str(e))}


# ==============================
# Publish CSV Data to MQTT
# ==============================
def publish_csv_to_mqtt():
    df = load_dataset()
    if df.empty:
        print("[ERROR] Dataset is empty, nothing to publish.")
        return

    print(f"[INFO] Publishing {len(df)} rows to MQTT...")

    for _, row in df.iterrows():
        message = {
            "Battery_Temp": float(row["Battery_Temp"]),
            "Brake_Pad_Wear": float(row["Brake_Pad_Wear"]),
            "Charging_Cycles": int(row["Charging_Cycles"]),
            "Charging_Voltage": float(row["Charging_Voltage"]),
            "DTC": int(row["DTC"]),
            "Motor_RPM": float(row["Motor_RPM"]),
            "Motor_Temp": float(row["Motor_Temp"]),
            "Motor_Torque": float(row["Motor_Torque"]),
            "SOC": float(row["SOC"]),
            "SOH": float(row["SOH"]),
            "Tire_Pressure": float(row["Tire_Pressure"]),
            "Timestamp": row["Timestamp"].strftime("%Y-%m-%d %H:%M:%S")
        }

        client.publish(topic="ev/dynamic/daily_user", qos=0,
payload=json.dumps(message))
        time.sleep(0.2)
```

```
    print("[INFO] All daily user data published.")


# ==============================
# Load Dataset
# ==============================
def load_dataset():
    try:
        obj = s3.get_object(Bucket=bucket, Key=key)
        df = pd.read_csv(obj['Body'])
        df.rename(columns={df.columns[0]: 'Timestamp'}, inplace=True)
        df['Timestamp'] = pd.to_datetime(df['Timestamp'], format="%Y-%m-%d
%H:%M:%S").dt.tz_localize('Asia/Kuala_Lumpur').dt.tz_convert('UTC')
        print(f"[INFO] Loaded {len(df)} rows from {key}")
        return df
    except Exception as e:
        print(f"[ERROR] Failed to load dataset: {e}")
        return pd.DataFrame()
```

IoT SiteWise

What is AWS IoT SiteWise used for in the context of our project?

1. Storing and Structuring Sensor Data

We are simulating an electric vehicle fleet by sending rows of sensor data (from CSV) every 15 minutes. IoT SiteWise acts as a data historian, storing this incoming time-series data from each EV as structured, timestamped properties.

Example sensor properties stored:

- Battery_Temp
- Brake_Pad_Wear
- SOC (State of Charge)
- SOH (State of Health)
- Motor_RPM, Motor_Torque, etc.

SiteWise organizes this data under "Assets" and "Properties", making it easy to manage and track changes over time.

2. Triggering ML-Based Predictions

We have a pipeline setup where:

1. A Lambda function pushes new sensor data (e.g row 1) to SiteWise.
2. It also sets a flag called TriggerPrediction = true.
3. AWS System Manager will get a message from lambda that row 1 has been sent to SiteWise, so the System Manager will increment the row parameter to 2 to prepare for the next call after 15 minutes
4. A second lambda (Prediction) will detect TriggerPrediction was set True by the first one
5. Prediction will happen using the Model we deploy, the data will be returned to SiteWise as PredictedTTF (double) and FailureSoon (True/False).
6. TriggerPrediction will reset to 0
7. After 15 minutes passed, AWS EventBridge will trigger the first lambda to be called again, and the process repeats

This wouldn't be possible without IoT SiteWise acting as the source of truth for both raw sensor values and control signals. This data stream simulation can be applied to real sensor data where we receive sensor data every set amount of time which can range from a few seconds all the way to a few minutes or hours. We can even receive it live (random timing) if the system was set up to accept that. This allows us to visualize the output of our prediction and sensor data in Twin Maker and Grafana.

3. Providing Data for Visualization in Grafana

Using the predictions our ML model makes:

- PredictedTTF (Time To Failure in hours)
- FailureSoon (True/False)
- These are then visualized in real-time dashboards using Amazon Managed Grafana, allowing us to:
  - » Monitor EV health status
  - » Spot upcoming failures
  - » See prediction trends

4. Enabling Digital Twin Simulation via TwinMaker

SiteWise is the backend data source for AWS IoT TwinMaker in our demo. While TwinMaker shows the 3D visualization and entity relationships, SiteWise feeds the live and historical sensor data into it.

Why SiteWise instead of just pushing data directly from S3 using Lambda?

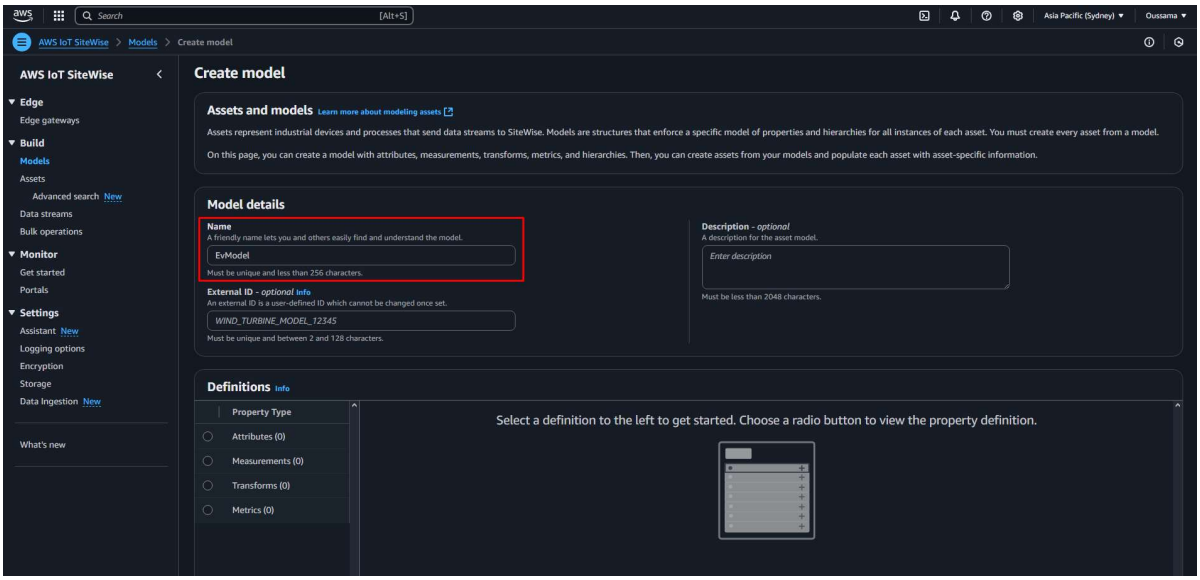
SiteWise offers built-in time-series support (timestamps, interpolation, etc.) which lambda did not do a great job at when trying to stream timeseries data to TwinMaker. It would frequently show blank blocks and fail to grab the data at the set interval, using SiteWise solved this problem. Since it allows us to get real-time updates, integrate smoothly with IoT Core, TwinMaker, EventBridge, and Systems Manager.

Summary

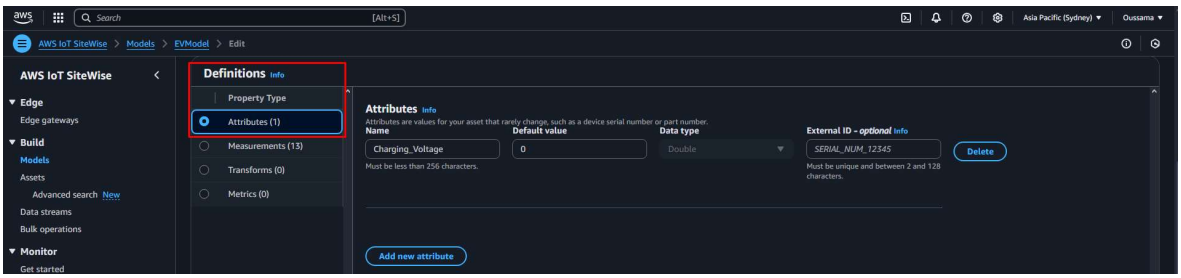
For our project, AWS IoT SiteWise is the real-time sensor data backbone. Since it receives and organizes simulated EV sensor data, stores both raw and predicted data for inspection, enables triggering predictions via IoT EventBridge, and powers TwinMaker scene and Grafana dashboards. Without SiteWise, we'd have to manually build a complex data ingestion and time-series handling system.

Setting up IoT SiteWise

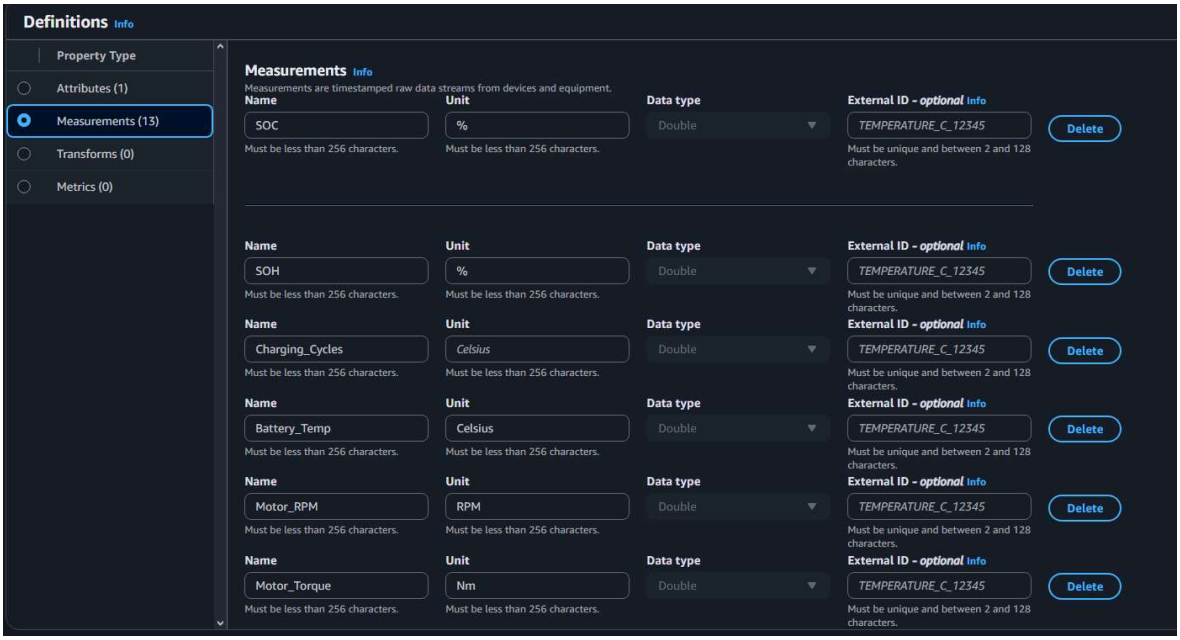
1. Go to AWS IoT SiteWise -> Build -> Models
2. Click on **Create asset model**
3. Enter a name for your Car Model

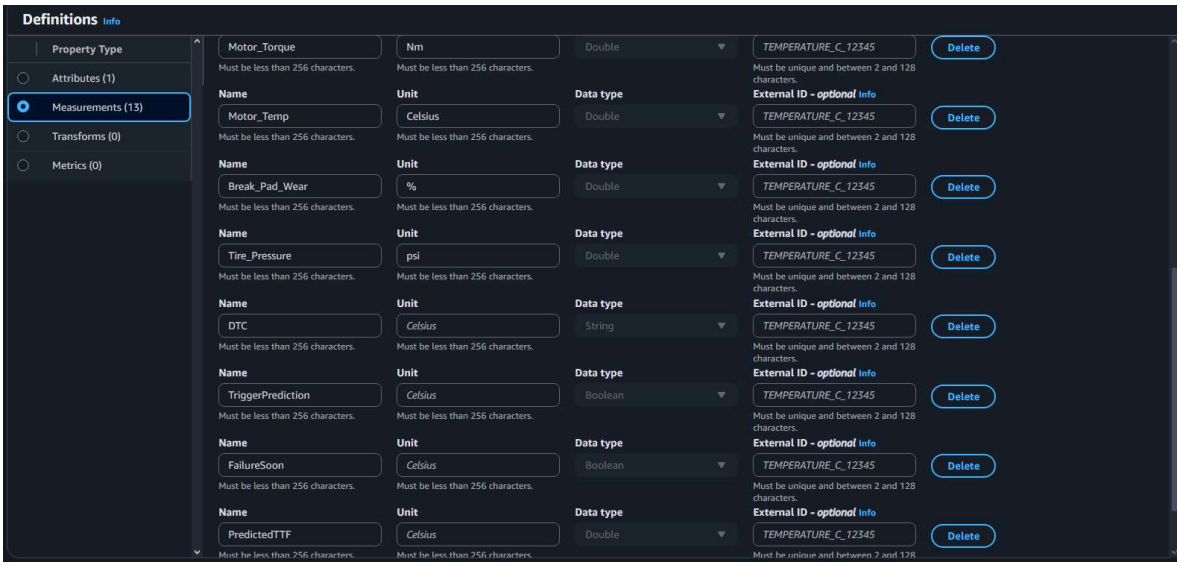


4. Below definitions, click on **Attributes**
    - » Attributes are values for our asset that do not change, such as charging voltage for our case, we will initialize it for the model at 0, then for each asset like daily user, moderate user, rare user, we will specify the charging voltage accordingly and it remains consistent across the entire data stream for that asset.

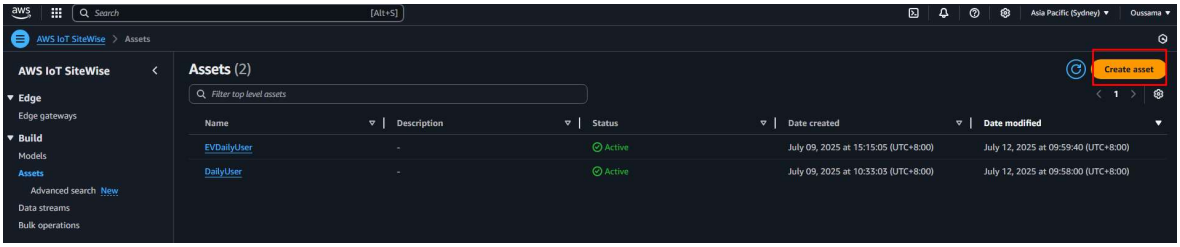


5. After setting the Attributes, go to **Measurements** to add the data that changes frequently in time-series.
    - ◊ TriggerPrediction is a True/False Boolean which we trigger when we invoke our first lambda which sends the next stream of data
    - ◊ FailureSoon is a True/False Boolean which is an indicator for our Classification model (True if <48 hours till Failure & False if >48 hours till Failure)
    - ◊ PredictedTTF is a double value which displays the predicted time till failure of the vehicle depending on the sensor data

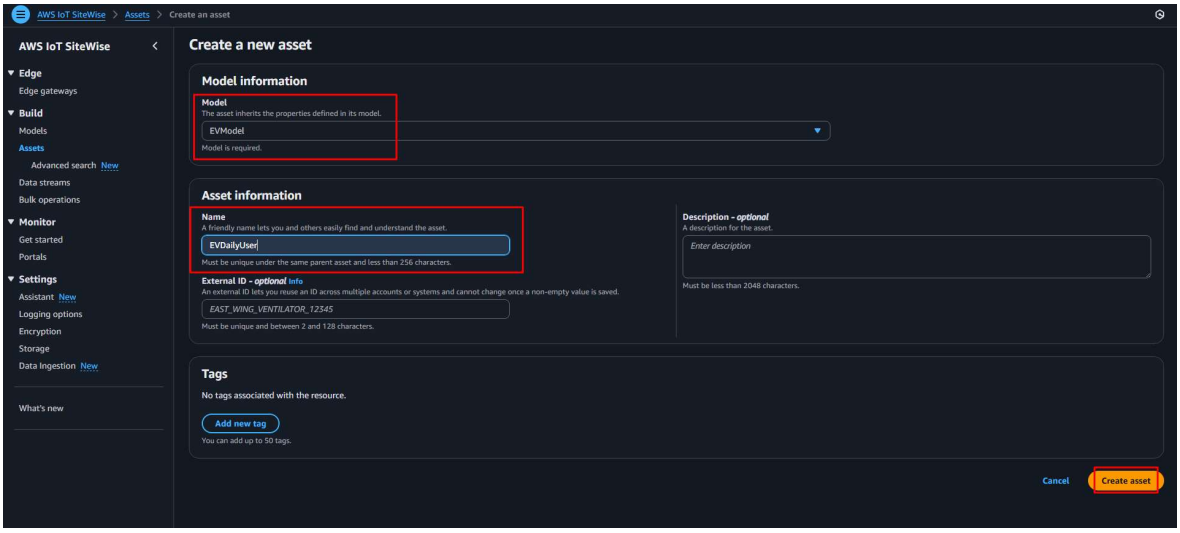




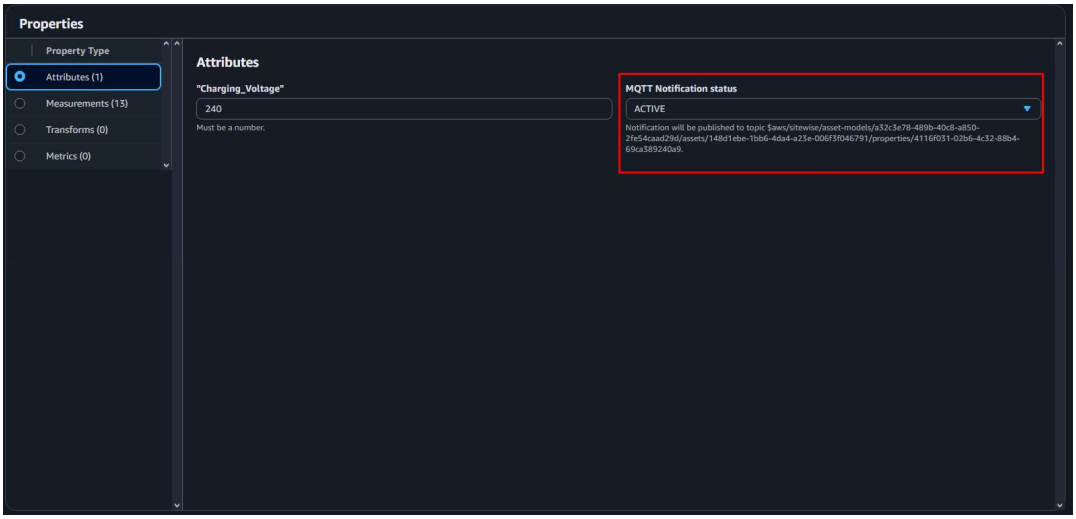
6. After creating the car model, we can now create the Asset.
   Go to Build -> Assets -> **Create asset**

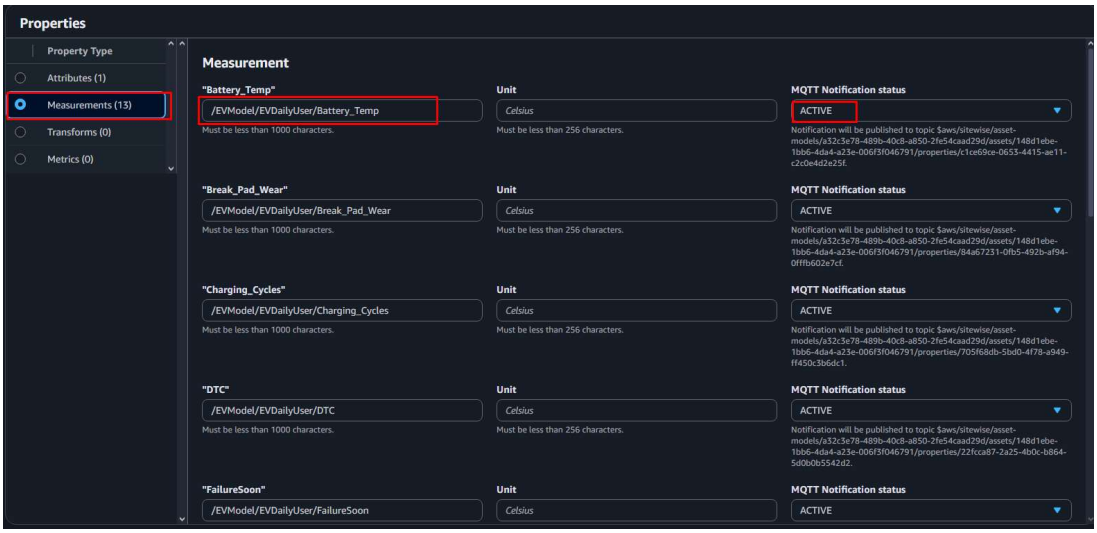


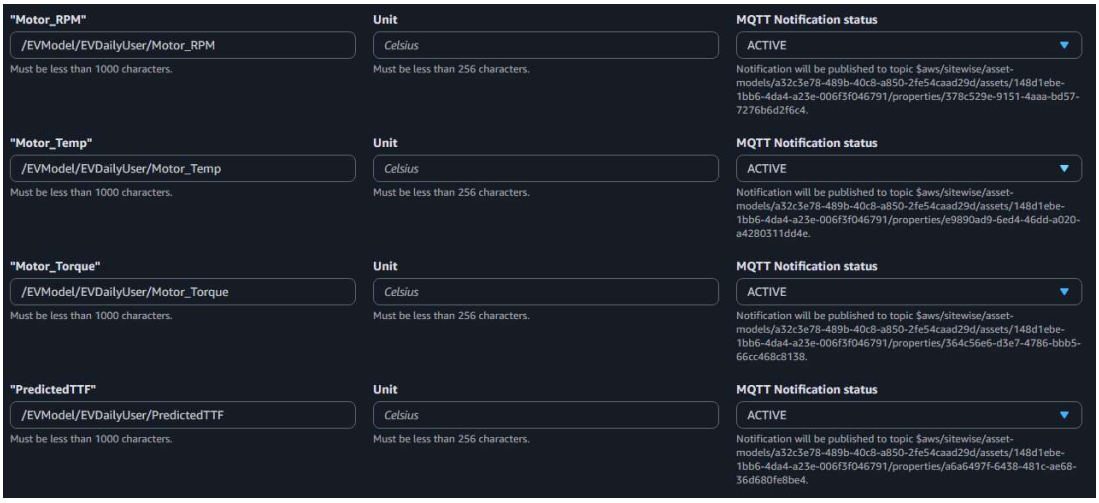
7. Make sure you select the model we just created (**EVModel**)
8. After selecting the model, enter a name for your asset
9. Click on **Create asset**

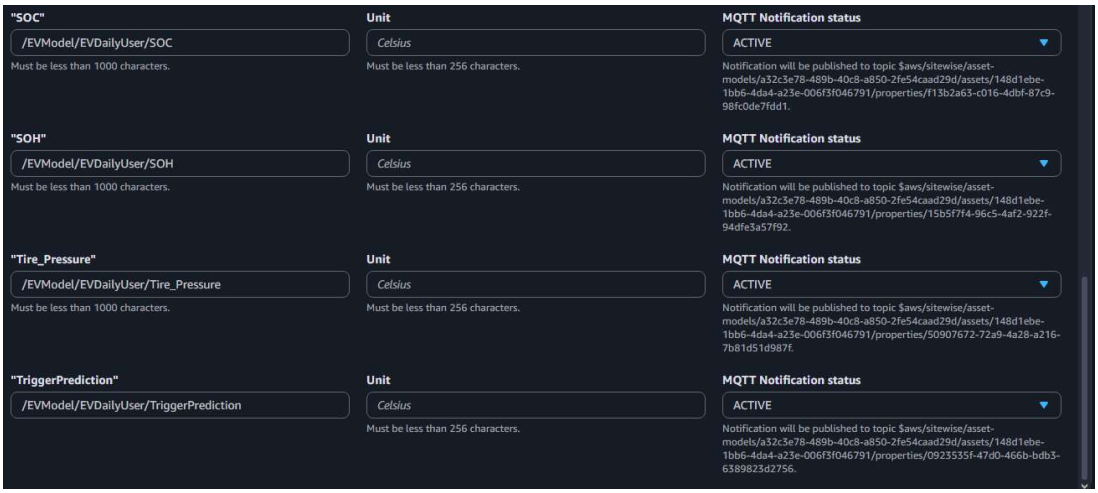


10. After creating the asset, you should now be able to see the attributes and measurements inherited from our EVModel asset model. The next step is to go to each of the attributes and measurements, enable MQTT notifications, and give it an alias as shown below.

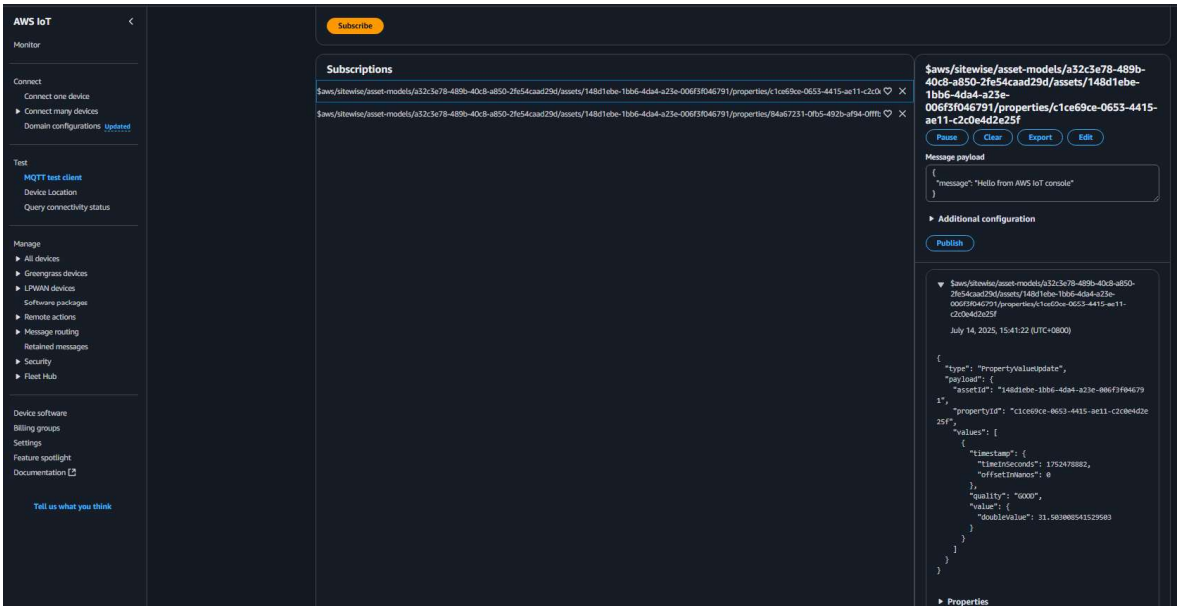



Properties
Property Type
Attributes (1)
Measurements (13)
Transforms (0)
Metrics (0)
Measurement
"Battery_Temp"
/EVModel/EVDailyUser/Battery_Temp
Must be less than 1000 characters.
Unit
Celsius
Must be less than 256 characters.
MQTT Notification status
ACTIVE
Notification will be published to topic $aws/sitewise/asset-models/a32c3e78-489b-40c8-a850-2fe54caad29d/assets/148d1ebe-1bb6-4da4-a23e-006f3f046791/properties/c1ce69ce-0653-4415-ae11-c2c0e4d2e25f.
"Break_Pad_Wear"
/EVModel/EVDailyUser/Break_Pad_Wear
Must be less than 1000 characters.
Unit
Celsius
Must be less than 256 characters.
MQTT Notification status
ACTIVE
Notification will be published to topic $aws/sitewise/asset-models/a32c3e78-489b-40c8-a850-2fe54caad29d/assets/148d1ebe-1bb6-4da4-a23e-006f3f046791/properties/84a67231-0fb5-492b-af94-0fffb602e7cf.
"Charging_Cycles"
/EVModel/EVDailyUser/Charging_Cycles
Must be less than 1000 characters.
Unit
Celsius
Must be less than 256 characters.
MQTT Notification status
ACTIVE
Notification will be published to topic $aws/sitewise/asset-models/a32c3e78-489b-40c8-a850-2fe54caad29d/assets/148d1ebe-1bb6-4da4-a23e-006f3f046791/properties/705f68db-5bd0-4f78-a949-ff450c3b6dc1.
"DTC"
/EVModel/EVDailyUser/DTC
Must be less than 1000 characters.
Unit
Celsius
Must be less than 256 characters.
MQTT Notification status
ACTIVE
Notification will be published to topic $aws/sitewise/asset-models/a32c3e78-489b-40c8-a850-2fe54caad29d/assets/148d1ebe-1bb6-4da4-a23e-006f3f046791/properties/22fcca87-2a25-4b0c-b864-5d0b0b5542d2.
"FailureSoon"
/EVModel/EVDailyUser/FailureSoon
Unit
Celsius
MQTT Notification status
ACTIVE


"Motor_RPM"
/EVModel/EVDailyUser/Motor_RPM
Must be less than 1000 characters.
Unit
Celsius
Must be less than 256 characters.
MQTT Notification status
ACTIVE
Notification will be published to topic $aws/sitewise/asset-models/a32c3e78-489b-40c8-a850-2fe54caad29d/assets/148d1ebe-1bb6-4da4-a23e-006f3f046791/properties/378c529e-9151-4aaa-bd57-7276b6d2f6c4.
"Motor_Temp"
/EVModel/EVDailyUser/Motor_Temp
Must be less than 1000 characters.
Unit
Celsius
Must be less than 256 characters.
MQTT Notification status
ACTIVE
Notification will be published to topic $aws/sitewise/asset-models/a32c3e78-489b-40c8-a850-2fe54caad29d/assets/148d1ebe-1bb6-4da4-a23e-006f3f046791/properties/e9890ad9-6ed4-46dd-a020-a4280311dd4e.
"Motor_Torque"
/EVModel/EVDailyUser/Motor_Torque
Must be less than 1000 characters.
Unit
Celsius
Must be less than 256 characters.
MQTT Notification status
ACTIVE
Notification will be published to topic $aws/sitewise/asset-models/a32c3e78-489b-40c8-a850-2fe54caad29d/assets/148d1ebe-1bb6-4da4-a23e-006f3f046791/properties/364c56e6-d3e7-4786-bbb5-66cc468c8138.
"PredictedTTF"
/EVModel/EVDailyUser/PredictedTTF
Must be less than 1000 characters.
Unit
Celsius
Must be less than 256 characters.
MQTT Notification status
ACTIVE
Notification will be published to topic $aws/sitewise/asset-models/a32c3e78-489b-40c8-a850-2fe54caad29d/assets/148d1ebe-1bb6-4da4-a23e-006f3f046791/properties/a6a6497f-6438-481c-ae68-36d680fe8be4.


"SOC"
/EVModel/EVDailyUser/SOC
Must be less than 1000 characters.
Unit
Celsius
Must be less than 256 characters.
MQTT Notification status
ACTIVE
Notification will be published to topic $aws/sitewise/asset-models/a32c3e78-489b-40c8-a850-2fe54caad29d/assets/148d1ebe-1bb6-4da4-a23e-006f3f046791/properties/f13b2a63-c016-4dbf-87c9-98fc0de7fdd1.
"SOH"
/EVModel/EVDailyUser/SOH
Must be less than 1000 characters.
Unit
Celsius
Must be less than 256 characters.
MQTT Notification status
ACTIVE
Notification will be published to topic $aws/sitewise/asset-models/a32c3e78-489b-40c8-a850-2fe54caad29d/assets/148d1ebe-1bb6-4da4-a23e-006f3f046791/properties/15b5f7f4-96c5-4af2-922f-94dfe3a57f92.
"Tire_Pressure"
/EVModel/EVDailyUser/Tire_Pressure
Must be less than 1000 characters.
Unit
Celsius
Must be less than 256 characters.
MQTT Notification status
ACTIVE
Notification will be published to topic $aws/sitewise/asset-models/a32c3e78-489b-40c8-a850-2fe54caad29d/assets/148d1ebe-1bb6-4da4-a23e-006f3f046791/properties/50907672-72a9-4a28-a216-7b81d51d987f.
"TriggerPrediction"
/EVModel/EVDailyUser/TriggerPrediction
Unit
Celsius
Must be less than 256 characters.
MQTT Notification status
ACTIVE
Notification will be published to topic $aws/sitewise/asset-models/a32c3e78-489b-40c8-a850-2fe54caad29d/assets/148d1ebe-1bb6-4da4-a23e-006f3f046791/properties/0923535f-47d0-466b-bdb3-6389823d2756.

11. Now to make sure you are receiving the data streams, copy the notification topic then go to AWS IoT Core -> MQTT test client -> Subscribe to a topic -> Topic filter, paste the topic



Below is the **Lambda 1** python code that next row of sensor data from a CSV stored in S3 to IoT SiteWise (as if it's coming from a live device). It will then set the flag TriggerPrediction to True in SiteWise to trigger a prediction workflow. Lastly another function (Lambda 2) will be triggered when TriggerPrediction is True to perform machine learning prediction using current sensor values.

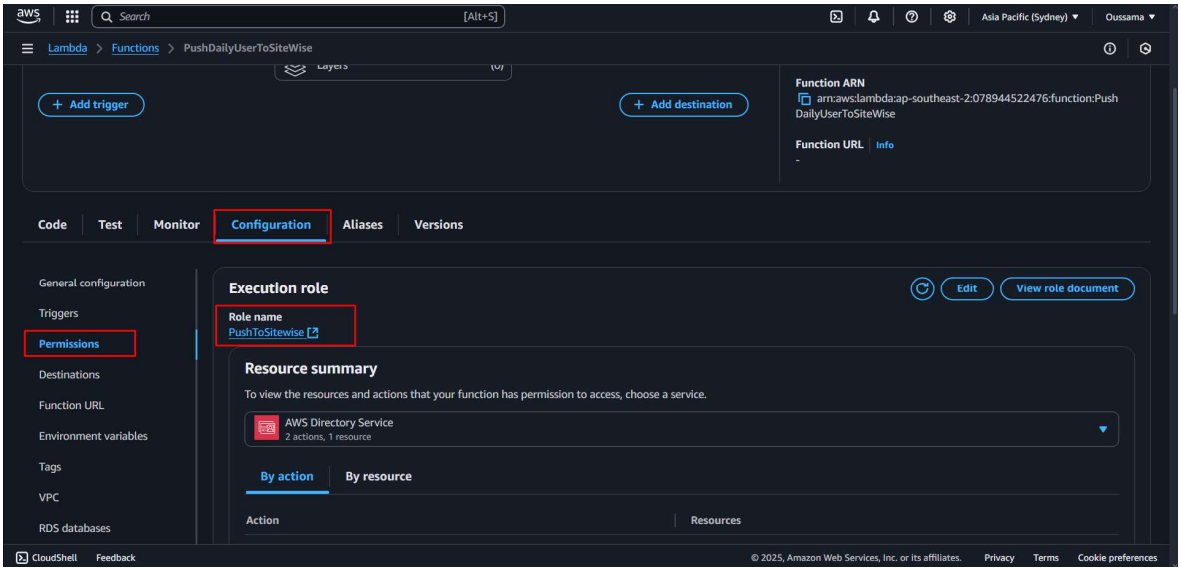
Lambda 1 Function

```
import boto3
import csv
import time
import uuid
from io import StringIO

# AWS clients
s3 = boto3.client('s3')
sitewise = boto3.client('iotsitewise', region_name='ap-southeast-2')
ssm = boto3.client('ssm')
lambda_client = boto3.client('lambda')
PREDICTION_LAMBDA_NAME = 'DailyUserPrediction'

# Configurations
BUCKET_NAME = 'ev-iot-dataset'
CSV_FILE_KEY = 'daily_user.csv'
ASSET_ID = '148d1ebe-1bb6-4da4-a23e-006f3f046791'
SSM_PARAM = '/eviot/row_index'

# Property IDs
PROPERTY_ID_MAP = {
    'SOC': 'f13b2a63-c016-4dbf-87c9-98fc0de7fdd1',
    'SOH': '15b5f7f4-96c5-4af2-922f-94dfe3a57f92',
    'Charging_Cycles': '705f68db-5bd0-4f78-a949-ff450c3b6dc1',
    'Battery_Temp': 'c1ce69ce-0653-4415-ae11-c2c0e4d2e25f',
    'Motor_RPM': '378c529e-9151-4aaa-bd57-7276b6d2f6c4',
    'Motor_Torque': '364c56e6-d3e7-4786-bbb5-66cc468c8138',
    'Motor_Temp': 'e9890ad9-6ed4-46dd-a020-a4280311dd4e',
    'Brake_Pad_Wear': '84a67231-0fb5-492b-af94-0fffb602e7cf',
    'Tire_Pressure': '50907672-72a9-4a28-a216-7b81d51d987f',
    'DTC': '22fcca87-2a25-4b0c-b864-5d0b0b5542d2',
    'TriggerPrediction': '0923535f-47d0-466b-bdb3-6389823d2756'
}

def load_csv_from_s3():
    response = s3.get_object(Bucket=BUCKET_NAME, Key=CSV_FILE_KEY)
    content = response['Body'].read().decode('utf-8')
    return list(csv.DictReader(StringIO(content)))

def send_row_to_sitewise(row, timestamp_epoch):
```

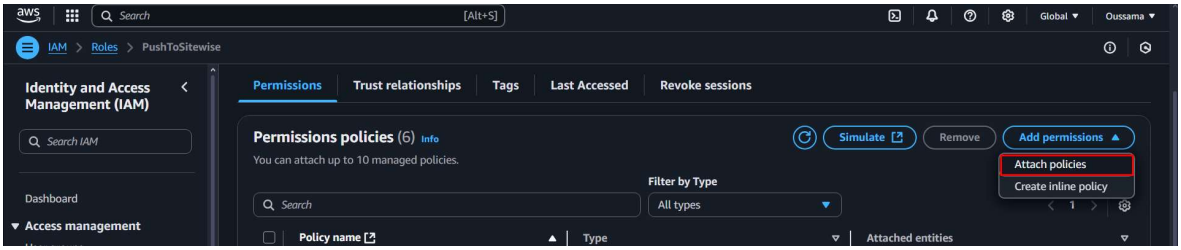
```
    entries = []
    for key, property_id in PROPERTY_ID_MAP.items():
        value_str = row.get(key)
        if value_str is None or value_str.strip() == '':
            continue

        try:
            value = str(value_str) if key == 'DTC' else float(value_str)
            value_key = 'stringValue' if key == 'DTC' else 'doubleValue'
        except ValueError:
            print(f"⚠️ Invalid {key} → '{value_str}'")
            continue

        entries.append({
            'entryId': str(uuid.uuid4()),
            'assetId': ASSET_ID,
            'propertyId': property_id,
            'propertyValues': [{
                'value': {value_key: value},
                'timestamp': {'timeInSeconds': timestamp_epoch, 'offsetInNanos': 0},
                'quality': 'GOOD'
            }]
        })

    # 🔁 Add TriggerPrediction = 1
    trigger_property_id = PROPERTY_ID_MAP.get('TriggerPrediction')
    if trigger_property_id:
        entries.append({
            'entryId': str(uuid.uuid4()),
            'assetId': ASSET_ID,
            'propertyId': trigger_property_id,
            'propertyValues': [{
                'value': {'booleanValue': True},
                'timestamp': {'timeInSeconds': timestamp_epoch, 'offsetInNanos': 0},
                'quality': 'GOOD'
            }]
        })

    if entries:
        for i in range(0, len(entries), 10):
            try:
                sitewise.batch_put_asset_property_value(entries=entries[i:i + 10])
            except Exception as e:
                print(f"❌ Batch send failed: {e}")

def get_next_index():
    try:
        param = ssm.get_parameter(Name=SSM_PARAM)
        return int(param['Parameter']['Value'])
    except ssm.exceptions.ParameterNotFound:
        ssm.put_parameter(Name=SSM_PARAM, Value='0', Type='String')
        return 0

def update_index(new_index):
    ssm.put_parameter(Name=SSM_PARAM, Value=str(new_index), Overwrite=True)

def lambda_handler(event, context):
    rows = load_csv_from_s3()
    index = get_next_index()

    if index >= len(rows):
        print("🎉 All rows sent.")
        return {'statusCode': 200, 'body': '✅ All rows have been sent.'}

    row = rows[index]
    ts = int(time.time())
    send_row_to_sitewise(row, ts)
    update_index(index + 1)

    print(f"✅ Sent row {index} at {ts}")

    # 🔁 Invoke Prediction Lambda
    try:
        lambda_client.invoke(
            FunctionName=PREDICTION_LAMBDA_NAME,
            InvocationType='Event'  # Async call
        )
        print("🚀 Triggered Prediction Lambda")
    except Exception as e:
        print(f"❌ Failed to trigger Prediction Lambda: {e}")

    return {'statusCode': 200, 'body': f"✅ Sent row {index} and triggered prediction"}
```

We must give the lambda function appropriate permissions now.

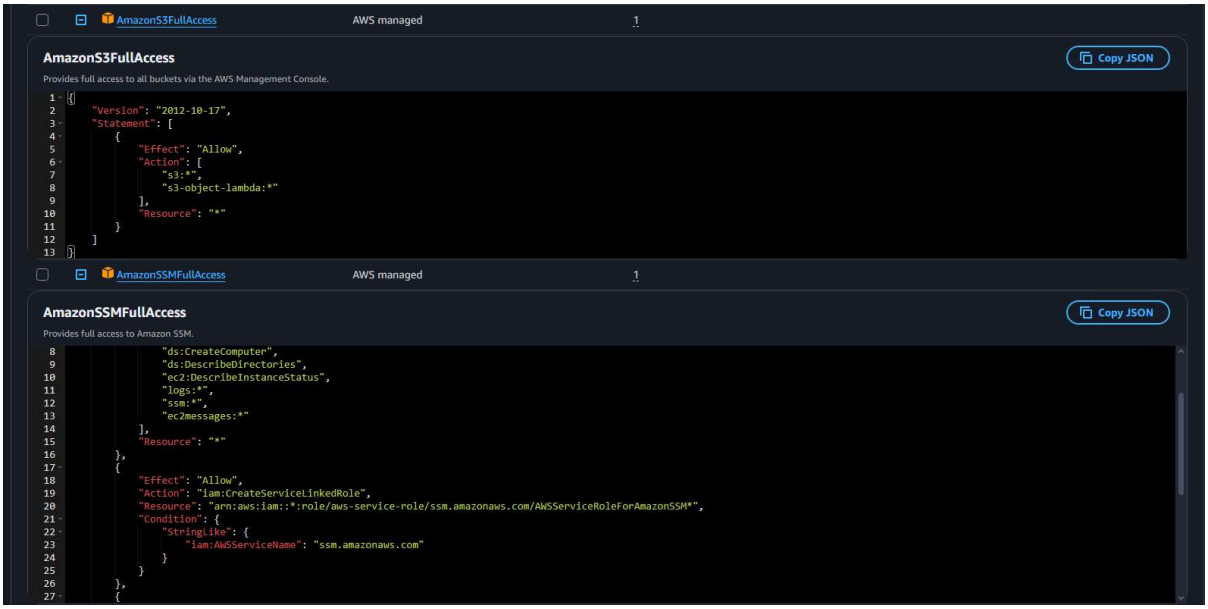
Go to Lambda 1 -> Configuration -> Permissions -> **Execution Role**

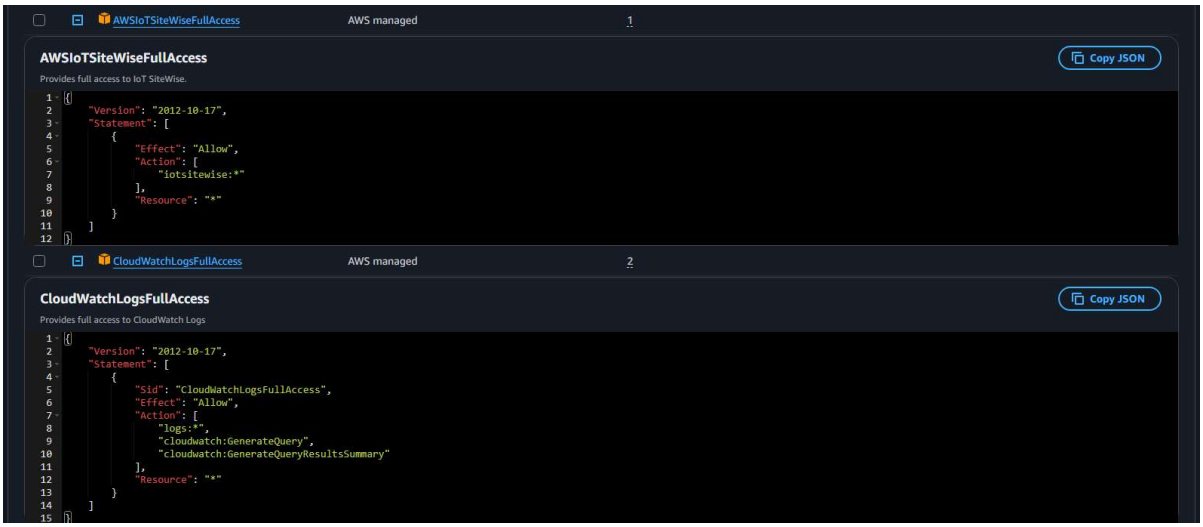


Click on Attach policies

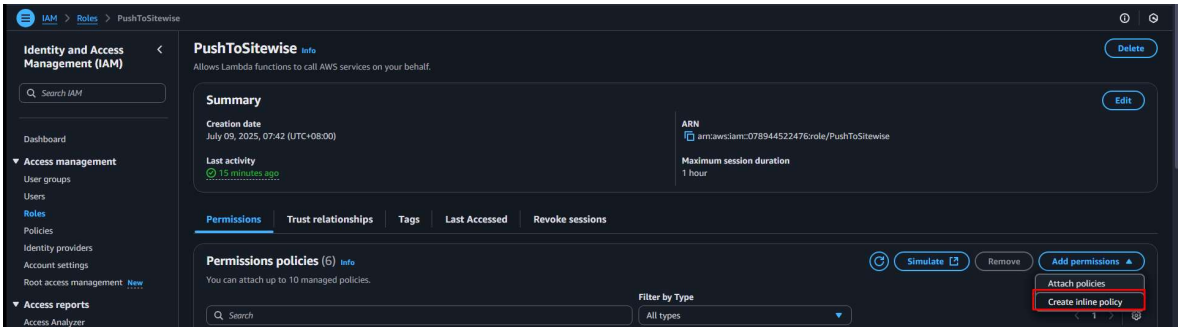


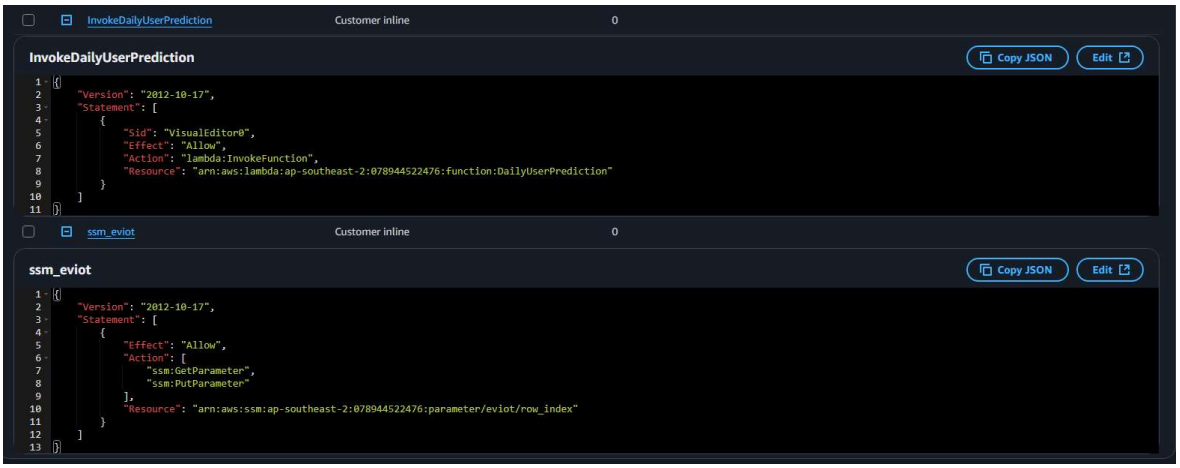
Attach the following policies as shown below.





After attaching the policies provided by AWS, now create 2 Inline Policies and paste:

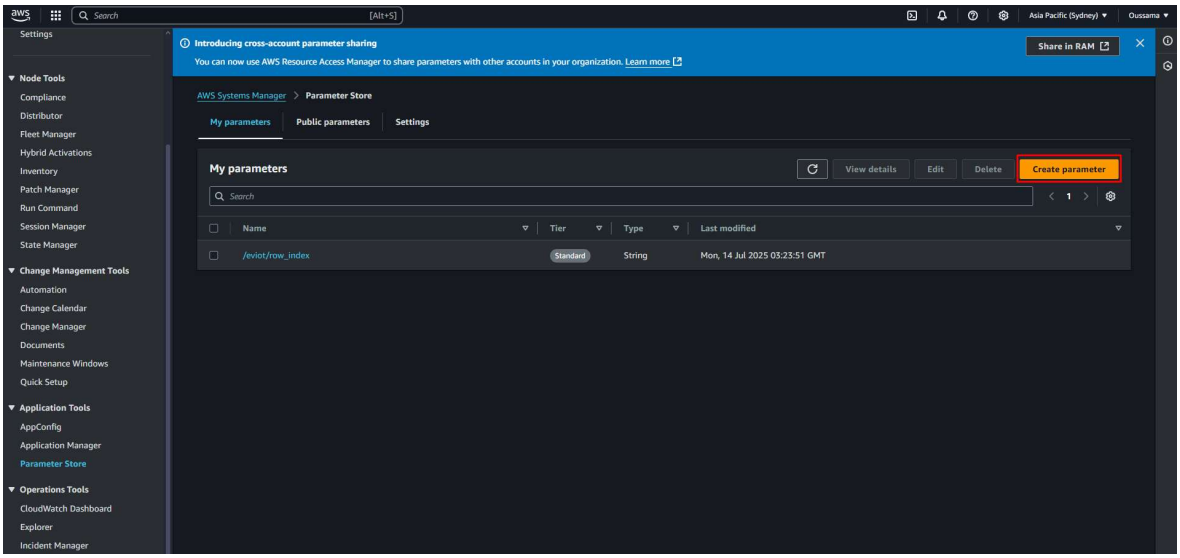




AWS Systems Manager

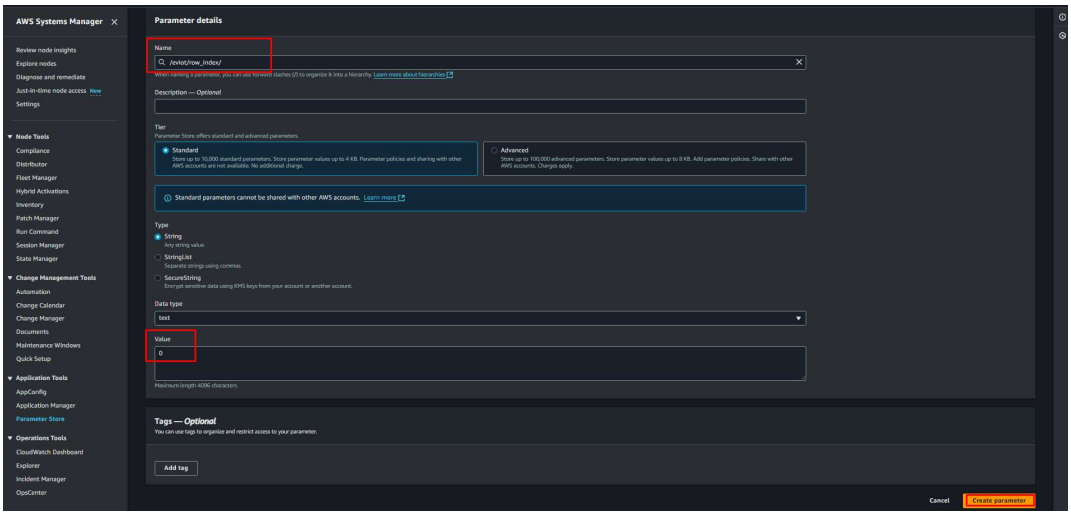
AWS Systems Manager (SSM) is a service that helps manage and automate AWS infrastructure, and in our project, we are using its **Parameter Store** feature to keep track of which row in our CSV file has already been sent to IoT SiteWise. Since AWS Lambda is stateless and doesn't remember previous executions, this parameter (/eviot/row_index) acts as a counter, storing the current row number. Each time the Lambda runs, it reads this index from SSM, sends the corresponding sensor data row to SiteWise, then updates the index for the next run. This allows us to simulate real-time streaming of data across multiple executions.

1. Go to Systems Manager -> Parameter Store -> **Create Parameter**



2. Decide a name for your parameter
3. Choose **Standard Tier**
4. Select type **String**
5. Data type: **text**
6. Value: **0**
    - This value is initialized at 0 which indicated the data row that we are at in the CSV, from there it will increment by 1 every time the first lambda is triggered.

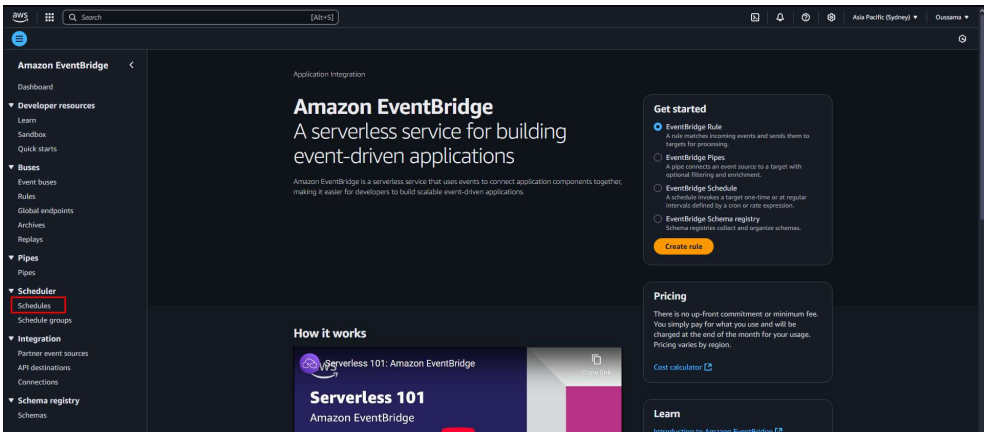
7. Create parameter



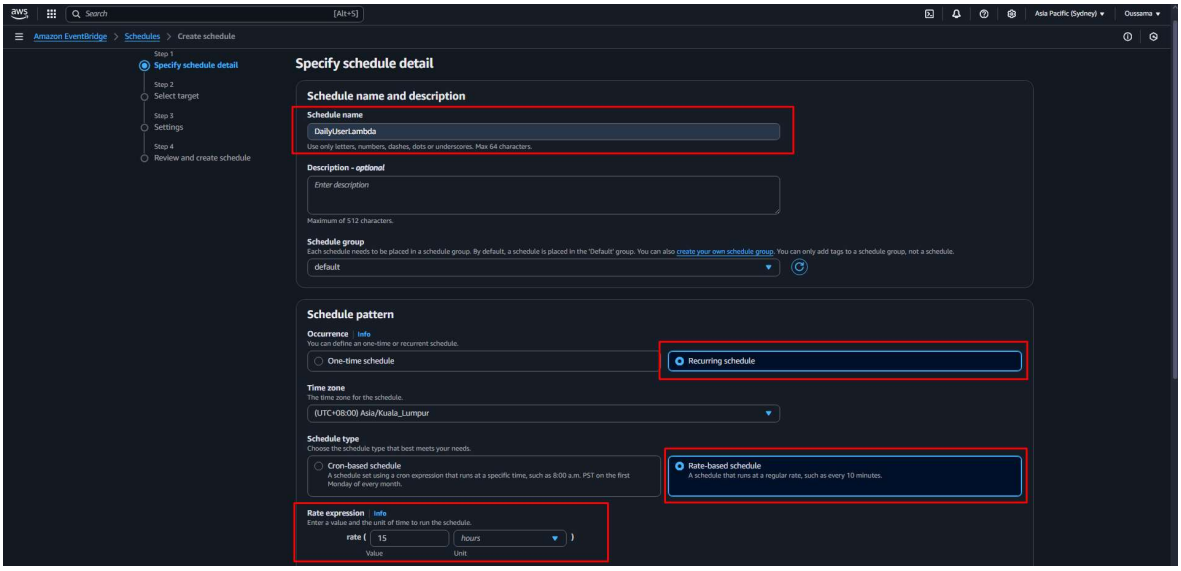
AWS EventBridge

Amazon EventBridge is what we use to automatically trigger our Lambda function every 15 minutes, simulating a real-time data stream from our electric vehicle sensor dataset. Instead of running it manually, we set up a schedule in EventBridge so that it consistently calls our Lambda, which sends the next row of sensor data to AWS IoT SiteWise. This way, we keep our simulation running smoothly and regularly, just like a real vehicle would send updates over time.

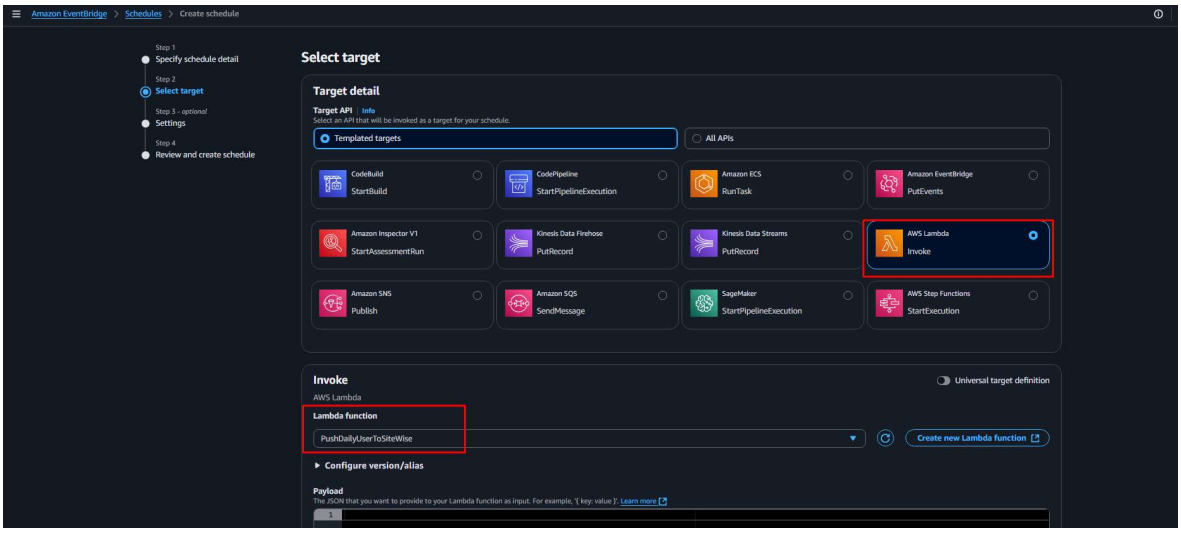
1. Go to EventBridge -> Scheduler -> **Schedules**



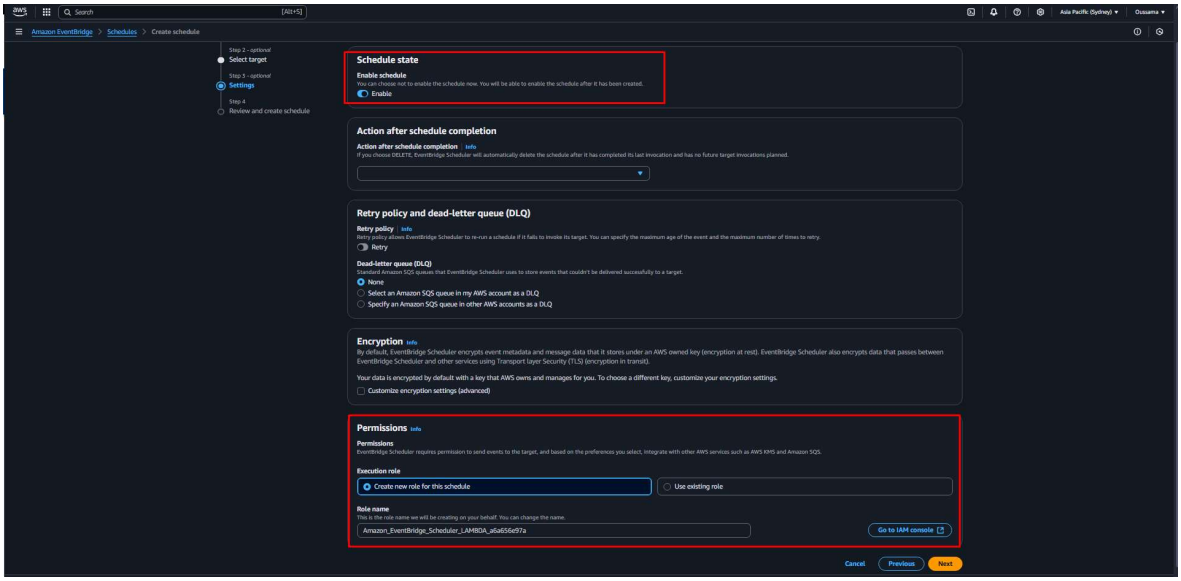
2. Enter Schedule name
3. Schedule pattern -> Occurrence -> Recurring schedule
4. Schedule pattern -> Schedule type -> Rate-based schedule
5. Schedule pattern -> Rate expression -> 15 minutes



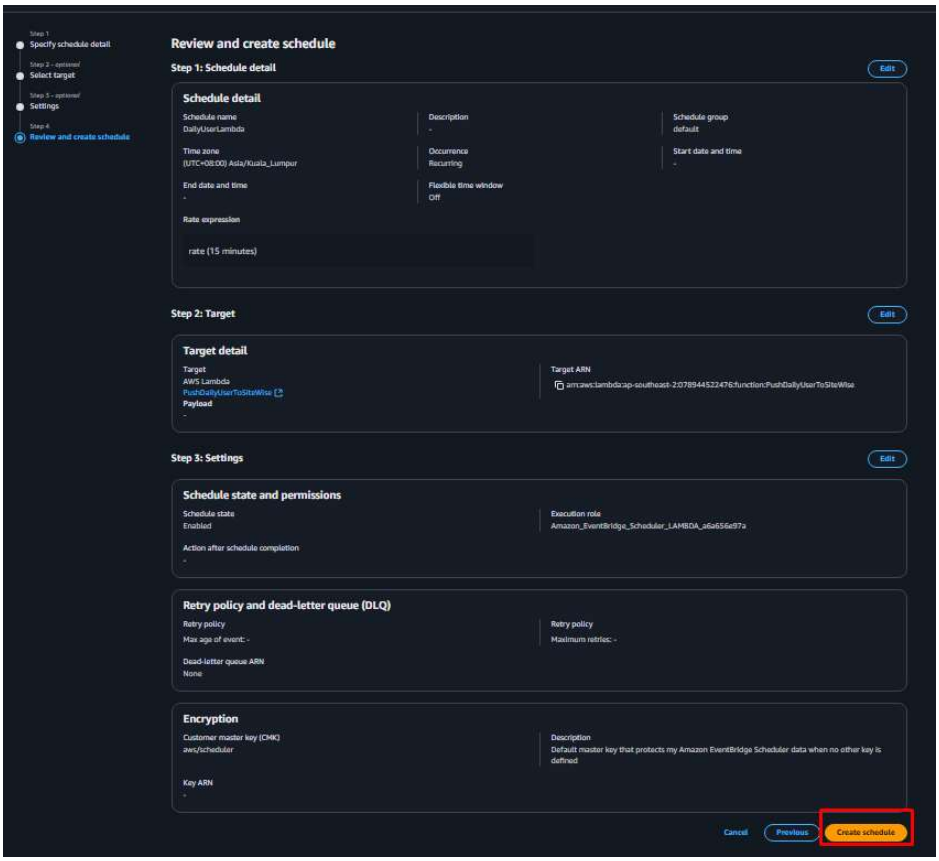
6. After clicking next, select AWS Lambda for the target
7. Under Invoke, enter your Lambda function name (the one pushing csv data)



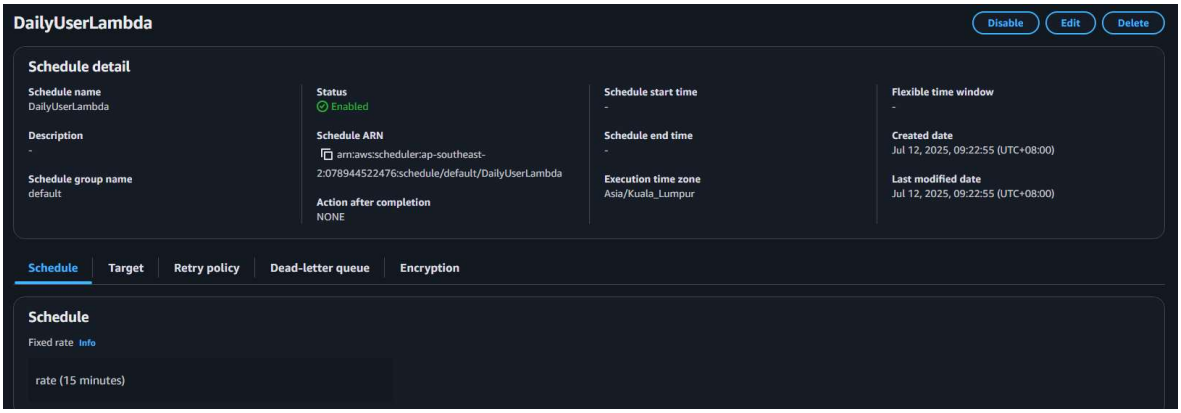
8. Make sure schedule is enabled
9. Let it create a role for you as well



10. Click on **Create schedule**



As you can see, the rate is 15 minutes, we can disable it at any time if we want to stop simulating the sensor data flow.



Google Colab
ML Model Training

Model Overview: Predicting Time-To-Failure (TTF) for Electric Vehicle Users

This machine learning project focuses on predictive maintenance using real-world IoT sensor data collected from electric vehicles (EVs). The goal is to accurately predict Time-To-Failure (how many hours remain before a vehicle is going to require maintenance) using a regression model and classify "early failures" (defined as failures expected within the next 48 hours).

What the Model Does:

- Trains on sensor data (e.g., SOC, SOH, RPM, torque, temperature).
- Learns to predict how many hours remain until the vehicle is going to fail.
- Post-processes the regression output into a binary classification: whether a failure will happen in less than 48 hours.
- This helps fleet operators and maintenance teams act before a fault occurs, reducing downtime and cost.

Dataset Evolution and Motivation for Change

Initially, training was done on another dataset which had near 30 features, and 175,000 sensor data. However, that dataset had several limitations:

- It lacked variety in user behavior.
- It failed to represent long-term operational patterns.
- It contained many noisy or sparse failure events.

After hours of experimenting with multiple regression and classification models (Random Forest, XGBoost, Linear Regression, etc.), it became clear that the original dataset could not support a highly accurate early failure detection model. Many of the models consistently misclassified failure events, either underestimating or overreacting to minor sensor variations.

To overcome these limitations, we had to switch to a new dataset. This dataset is split into 4 CSV files which are rare, moderate, heavy, and daily user data. This gave the model better diversity with different operational profiles and produced significantly better results when predicting failures under various real-world conditions.

Model Comparisons and Observed Issues

Random Forest Regressor

- **Observed Issues:**
  - Produced noisy and inconsistent TTF predictions.
  - Overreacted to minor anomalies such as voltage or temperature spikes.
  - Generated false alarms under normal conditions.
- **Analysis:**
  - Lacked sequential context or memory of historical trends.
  - Highly reactive to short-term variations in sensor values.

XGBoost / Gradient Boosting

- **Observed Issues:**
  - TTF outputs were irregular.
  - Model frequently misclassified normal operations as failure events.
- **Analysis:**
  - While able to model nonlinear feature interactions, these models could not effectively distinguish gradual failure trends from short-term fluctuations.

Linear Regression / SVR

- **Observed Issues:**
  - Predictions lacked variability; many were close to the average TTF.
  - Failed to capture both normal and critical operating conditions.
- **Analysis:**
  - These models were too simplistic to handle the more complex, nonlinear relationships present in multivariate sensor data.

Final Model and Evaluation: Random Forest Regressor

After extensive experimentation with various machine learning models, the team ultimately deployed a Random Forest Regressor to predict Time To Failure (TTF) of electric vehicles using IoT sensor data. This decision was primarily driven by deployment constraints involving deep learning models, particularly the CNN + BiLSTM model, which could not be effectively deployed on SageMaker due to TensorFlow container and size limitations. Even though Random Forest has lower performance compared to deep learning models, the Random Forest model offers a usable level of accuracy and was significantly easier to integrate into the real-time AWS pipeline.

```
Mean Absolute Error (Regression): 28.14216324200913
Accuracy (Classification): 0.6109589041095891
              precision    recall  f1-score   support

           0       0.65      0.89      0.75      5690
           1       0.33      0.10      0.16      3070

    accuracy                           0.61      8760
   macro avg       0.49      0.49      0.45      8760
weighted avg       0.53      0.61      0.54      8760
```

Performance Overview (Random Forest)

Regression Evaluation:

Mean Absolute Error (MAE): 28.14

- On average, the model's predicted TTF values deviated by 28.14 hours from the actual values.
- While this level of error may not be suitable for precise scheduling, it still provides value as an early warning system for preventive maintenance, mainly within the critical 48-hour window.

Classification Evaluation (Failure Soon: TTF < 48 hours)

To support alerting and visualization, the regression output was converted into a binary classification:

- 1 = FailureSoon (TTF < 48 hours)
- 0 = Safe (TTF ≥ 48 hours)

Results on a test set of 8,760 samples:

The classification results of the Random Forest model, which labels the electric vehicles as "FailureSoon" (TTF < 48 hours) or "Safe" (TTF ≥ 48 hours) shows several strengths and limitations. Below is a breakdown of what each metric means and how it applies to the use case:

Accuracy: 0.611

- Explanation: About 61.1% of the model's predictions (both Safe and FailureSoon) matched the actual labels in the test dataset.
- Insight: While better than random guessing (especially with imbalanced data), this moderate accuracy indicates the model is possibly making frequent misclassifications, especially when distinguishing vehicles close to the 48-hour TTF threshold.

Precision (FailureSoon = 1): 0.33

- Explanation: Only 33% of the vehicles predicted to fail soon were actually true positives.
- Insight: The model frequently raises false alarms, in other words it is predicting failures that don't happen within 48 hours. This could reduce trust in the system or cause unnecessary maintenance alerts.

Recall (FailureSoon = 1): 0.10

- Explanation: Out of all actual FailureSoon cases, the model correctly identified only 10%.
- Insight: This is a serious limitation, since 90% of vehicles that were genuinely close to failure were missed by the model. It highlights that the Random Forest model is poor at capturing rare but critical failure cases, which is dangerous in a predictive maintenance context where early intervention is important.

F1 Score (FailureSoon = 1): 0.16

- Explanation: This score balances precision and recall for the failure class. A value of 0.16 shows poor overall detection of at-risk vehicles.
- Insight: The model fails to keep a good balance between raising alarms and catching real failures. This confirms that the model is conservative, in other words it hesitates to label a case as FailureSoon unless it is very obvious.

Precision (Safe = 0): 0.65

- Explanation: When the model predicts a vehicle is "Safe", it's correct 65% of the time.
- Insight: This is a decent result since most vehicles labeled safe really are safe. It suggests the model has better confidence in normal operating conditions and tends to favor predicting the safer class when uncertain.

Recall (Safe = 0): 0.89

- Explanation: The model successfully identifies 89% of all truly "Safe" vehicles.
- Insight: This high recall indicates the model is strongly biased toward recognizing vehicles that don’t need maintenance. However, this comes at the cost of missing real failure risks (low recall for class 1). This kind of imbalance is typical in highly skewed datasets, especially where the majority of vehicles operate normally.

Interpretation & Analysis

Random Forest Regressor (TTF Prediction)

- **Advantage:** Handles high-dimensional and noisy sensor data well.
- **Output:** A continuous-valued prediction of how many hours remain until vehicle failure.
- **Interpretation:** Lower predicted TTF values indicate the vehicle is closer to failure.
- **Deployment Benefit:** Fast inference via joblib, with no need for GPU or TensorFlow.

Random Forest Classifier (FailureSoon)

- **Logic:** Converts TTF into a binary flag, show 1 if TTF < 48 hours.
- **Accuracy:** Evaluated using accuracy score and classification report (precision, recall, F1-score).
- **Interpretation:** Flags a vehicle as "at risk" of failing soon, supporting maintenance planning.
- **Handling Imbalance:** We used class_weight='balanced' to account for any imbalance in the FailureSoon labels.

Although the Random Forest model did not produce the highest accuracy, it was ultimately chosen due to:

- Faster prediction time and lightweight resource consumption.
- Straightforward integration with AWS SageMaker, Lambda and SiteWise.
- Robustness to missing or noisy data.

It served as a practical starting point for deploying a predictive maintenance system capable of highlighting potentially at-risk vehicles based on recent sensor readings.

Performance of CNN + BiLSTM (Highest, but Not Deployed)

The CNN + BiLSTM architecture yielded the best results in testing:

Component	Contribution
CNN	Captured short-term local patterns and anomalies in time-series data.
BiLSTM	Modeled long-term dependencies and trends over time.
Sequential Input	Trained on 24-time steps (6-hour window) to reflect historical behavior.
Overall Accuracy	Provided more stable and accurate TTF predictions and better classification balance.

However, it could not be deployed due to the following constraints:

- Large model size and loading time.
- TensorFlow container incompatibilities with SageMaker endpoints.
- Inability to meet the deployment timeout and memory limits of standard Lambda or SageMaker configurations.

Code for Preprocessing and Model Training:

The code shown below is used to train the Random Forest regression model to estimate the Time-Till-Failure for our vehicles.

```
# 1. Mount Google Drive
from google.colab import drive
drive.mount('/content/drive')

# 2. Imports
import pandas as pd
import numpy as np
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import MinMaxScaler
from sklearn.ensemble import RandomForestRegressor, RandomForestClassifier
from sklearn.metrics import mean_absolute_error, accuracy_score,
classification_report
import joblib

# 3. Load the CSV
file_path = '/content/drive/MyDrive/daily_user.csv'
df = pd.read_csv(file_path)

# 4. Drop unnecessary columns
df = df.drop(columns=['Unnamed: 0', 'DTC'])  # Remove timestamp and non-
numeric

# 5. Add synthetic TTF column (simulate time to failure in hours)
np.random.seed(42)
df['TTF'] = np.random.randint(10, 120, size=len(df))  # Simulate between
10h and 120h

# 6. Separate features and target for regression
target_column = 'TTF'
features = df.drop(columns=[target_column])
labels = df[target_column]

# 7. Normalize features
scaler = MinMaxScaler()
X_scaled = scaler.fit_transform(features)

# 8. Split for regression
X_train, X_test, y_train, y_test = train_test_split(X_scaled, labels,
test_size=0.2, random_state=42)

# 9. Train regression model
regressor = RandomForestRegressor(n_estimators=100, random_state=42)
regressor.fit(X_train, y_train)
```

```
# 10. Evaluate regression
y_pred_reg = regressor.predict(X_test)
print("Mean Absolute Error (Regression):", mean_absolute_error(y_test,
y_pred_reg))

# 11. Create binary label: FailureSoon = 1 if TTF < 48 hours
df['FailureSoon'] = (df['TTF'] < 48).astype(int)

# 12. Classification setup
features_clf = df.drop(columns=['TTF', 'FailureSoon'])
labels_clf = df['FailureSoon']
X_scaled_clf = scaler.transform(features_clf)

X_train_c, X_test_c, y_train_c, y_test_c = train_test_split(X_scaled_clf,
labels_clf, test_size=0.2, random_state=42)

# 13. Train classification model
classifier = RandomForestClassifier(n_estimators=100, random_state=42,
class_weight='balanced')
classifier.fit(X_train_c, y_train_c)

# 14. Evaluate classification
y_pred_clf = classifier.predict(X_test_c)
print("Accuracy (Classification):", accuracy_score(y_test_c, y_pred_clf))
print(classification_report(y_test_c, y_pred_clf))

# 15. Save models + scaler to Drive
joblib.dump(regressor, '/content/drive/MyDrive/rf_regressor_model.pkl')
joblib.dump(classifier, '/content/drive/MyDrive/rf_classifier_model.pkl')
joblib.dump(scaler, '/content/drive/MyDrive/minmax_scaler.pkl')
```

Model Development in Google Colab

The initial prototype was developed in Google Colab, where we:

- Loaded the electric vehicle dataset from Google Drive.
- Dropped irrelevant columns (Unnamed: 0, DTC).
- Simulated a synthetic TTF column (10–120 hours) to represent the time remaining until vehicle failure.
- Applied MinMaxScaler to normalize the sensor features.
- Trained a Random Forest Regressor to predict TTF.
- Converted the regression output into a binary class: FailureSoon = 1 if TTF < 48 hours, else 0.
- Trained a Random Forest Classifier for this binary classification task.
- Saved the trained models and scaler using joblib for later reuse.

Colab was suitable for development, but limitations in library versions compatibility with the SageMaker deployment environments led to migration.

Amazon SageMaker
ML Model Deployment

Migration to Amazon SageMaker for Training & Deployment

Since AWS Lambda and SageMaker endpoints required specific versions and packaging formats, we restructured the training pipeline to run fully in Amazon SageMaker using a script-based mode:

train_script.py (Model Training)

- Performs the same preprocessing and training as the Colab prototype.
- Saves the trained Random Forest Regressor and MinMaxScaler into the /opt/ml/model directory, as required by SageMaker.
- Uses SageMaker's default input/output folder conventions to access training data and persist artifacts.

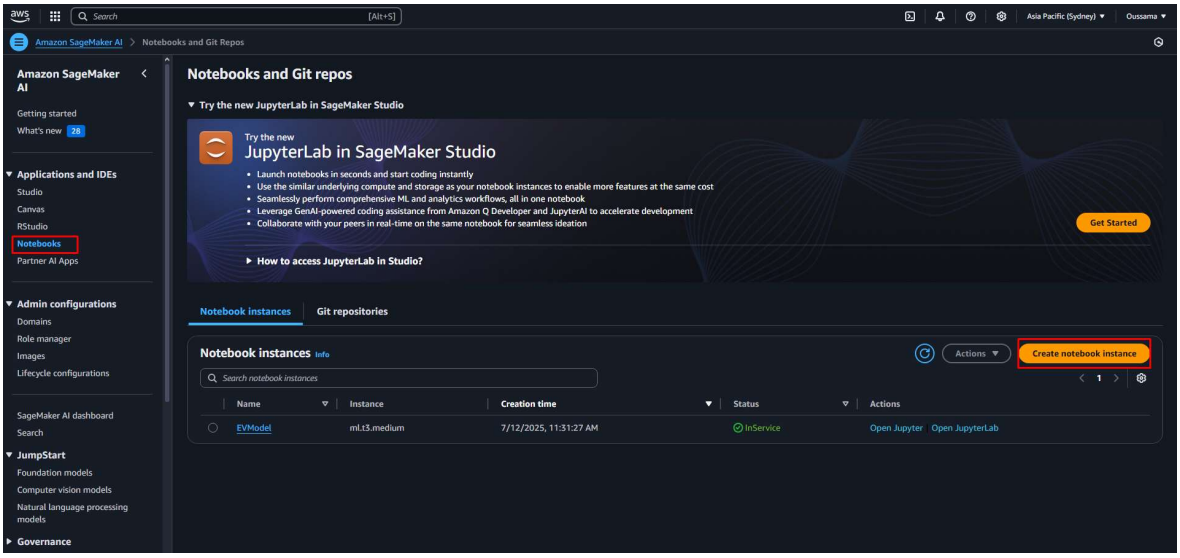
inference.py (Model Inference)

- Loads both the Random Forest model and the scaler.
- Accepts raw input data, normalizes it with the stored scaler, and performs predictions.
- Designed to be deployed as a SageMaker endpoint that can serve real-time TTF predictions.

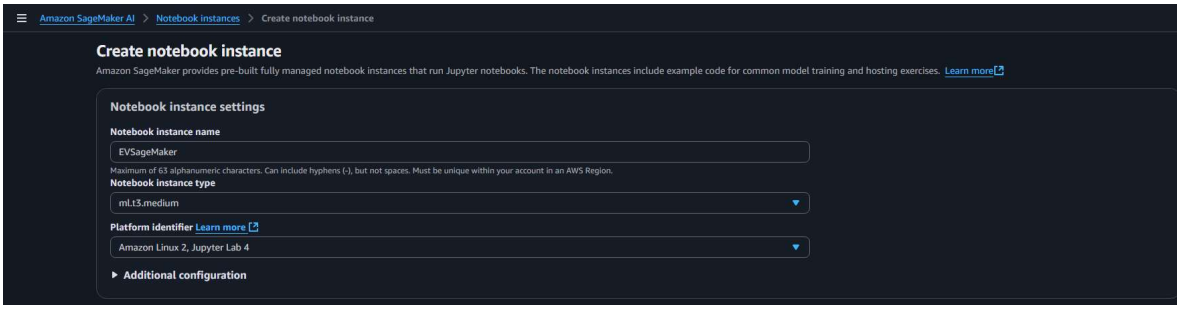
This approach allowed us to integrate the model seamlessly into the AWS pipeline, including Lambda functions and SiteWise data flow.

Setting Up SageMaker

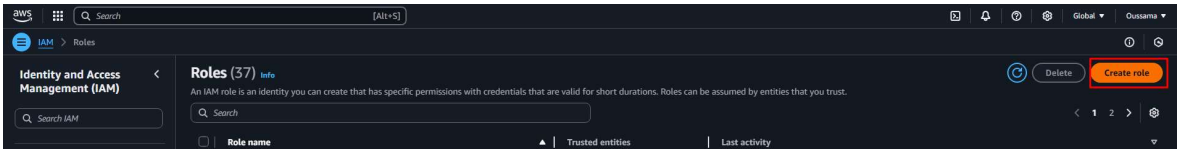
1. Go to Amazon SageMaker AI
2. Click on **Notebooks** under Applications and IDEs
3. Click on **Create Notebook Instance**



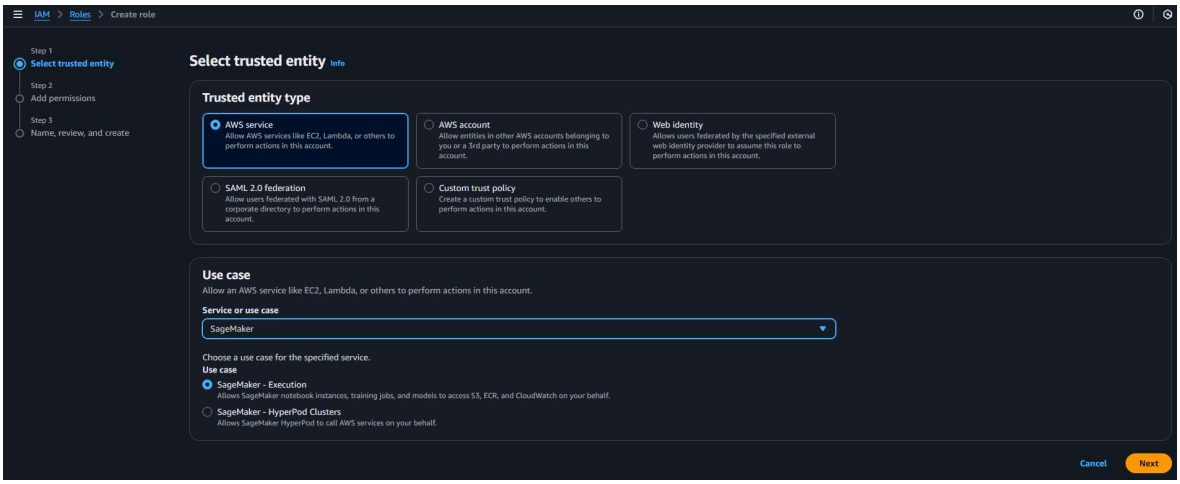
4. Enter the notebook instance name

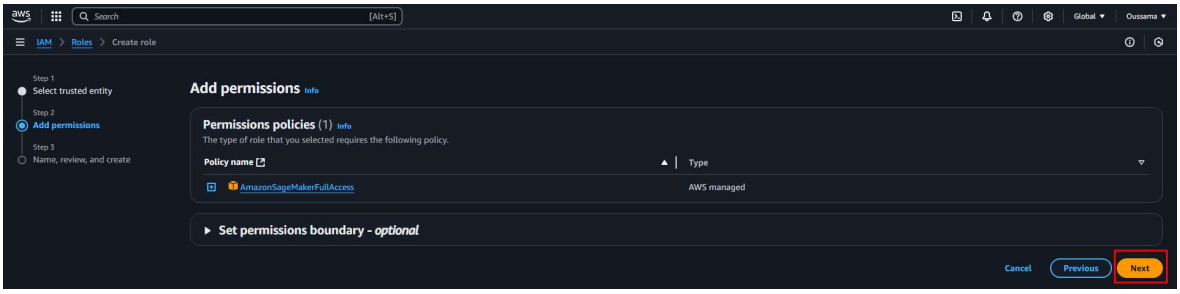


5. Go to IAM -> **Create Role**

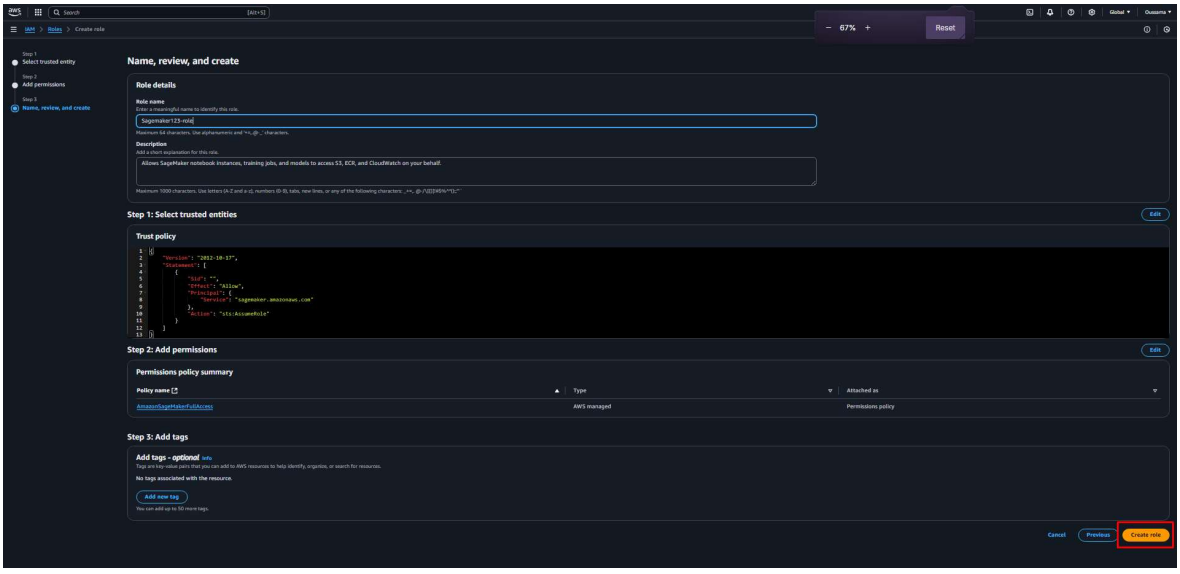


6. Choose SageMaker as the use case of our role, keep AWS service selected by default

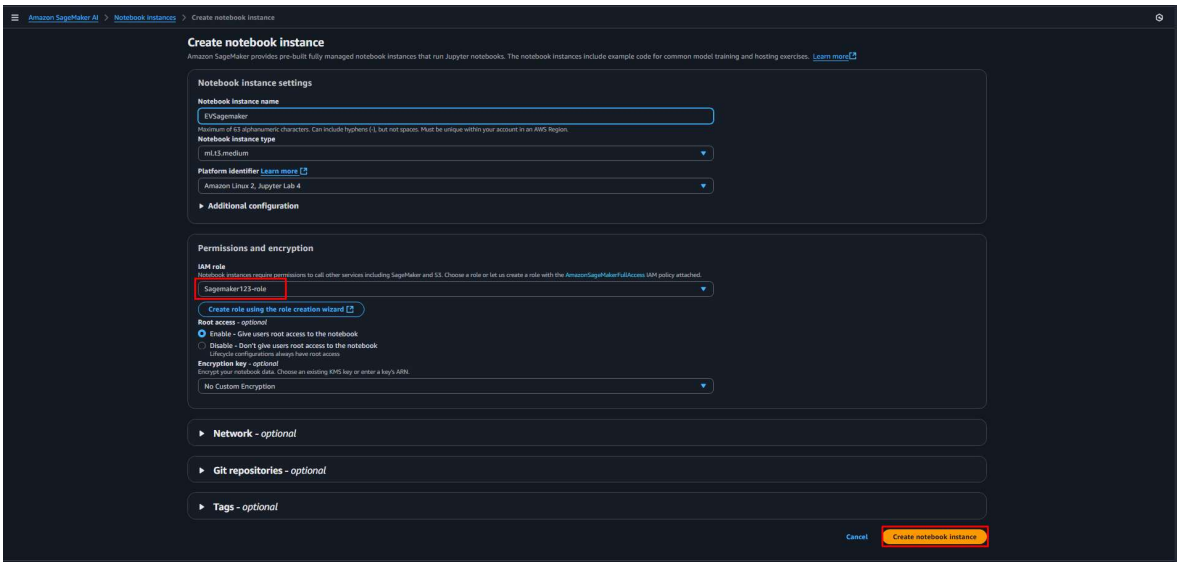




7. Press on **Create Role**



8. Go back to the page where we were creating the Notebook
9. Make sure you select the role we just created
10. Click on **Create notebook instance**

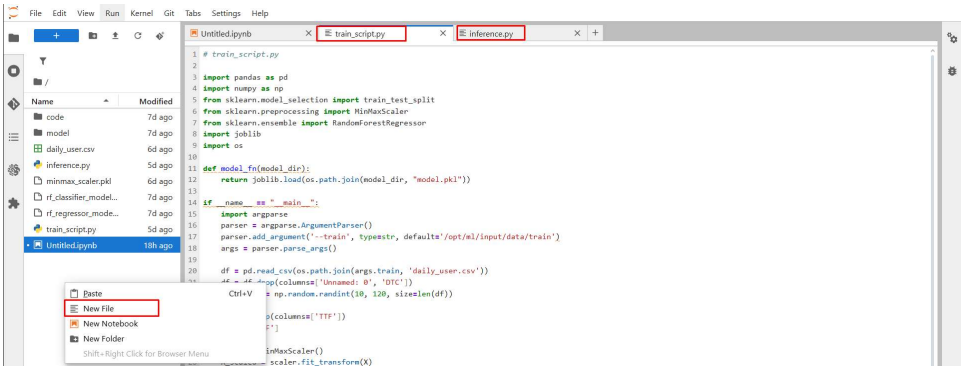


Model Deployment Process

1. To create the python script files, navigate to your SageMaker Notebook

2. Right click on the empty space, click **New File**

3. Name the file train_script.py and paste it in as shown below, do the same for inference.py



Code for train_script.py

```
# train_script.py

import pandas as pd
import numpy as np
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import MinMaxScaler
from sklearn.ensemble import RandomForestRegressor
import joblib
import os

def model_fn(model_dir):
    return joblib.load(os.path.join(model_dir, "model.pkl"))

if __name__ == "__main__":
    import argparse
    parser = argparse.ArgumentParser()
    parser.add_argument('--train', type=str,
default='/opt/ml/input/data/train')
    args = parser.parse_args()

    df = pd.read_csv(os.path.join(args.train, 'daily_user.csv'))
    df = df.drop(columns=['Unnamed: 0', 'DTC'])
    df['TTF'] = np.random.randint(10, 120, size=len(df))

    X = df.drop(columns=['TTF'])
    y = df['TTF']

    scaler = MinMaxScaler()
    X_scaled = scaler.fit_transform(X)

    X_train, X_test, y_train, y_test = train_test_split(X_scaled, y,
test_size=0.2, random_state=42)

    model = RandomForestRegressor(n_estimators=100, random_state=42)
    model.fit(X_train, y_train)

    # Save model and scaler
    joblib.dump(model, os.path.join('/opt/ml/model', 'model.pkl'))
    joblib.dump(scaler, os.path.join('/opt/ml/model', 'scaler.pkl'))
```

Code for inference.py

```
import joblib
import os

def model_fn(model_dir):
    # Load model
    model = joblib.load(os.path.join(model_dir, 'model.pkl'))

    # Load scaler
    scaler = joblib.load(os.path.join(model_dir, 'scaler.pkl'))

    # Return both
    return (model, scaler)

def predict_fn(input_data, model_and_scaler):
    model, scaler = model_and_scaler

    # Scale input
    scaled_input = scaler.transform(input_data)

    # Predict
    return model.predict(scaled_input)
```

After creating the script files, create a folder in your S3 bucket and name it "**trained-models**" (Refer to Amazon S3 section), then drag and drop your train_script in it (this process can also be automatically done through SageMaker using a python code. After that, we have to train the model on SageMaker to prepare it for deployment, to do that we will use our train_script.py which is stored now in S3

Paste in the code below in your SageMaker notebook

```
import sagemaker
from sagemaker.sklearn.estimator import SKLearn

role = sagemaker.get_execution_role()
bucket = 'ev-iot-dataset'
prefix = 'trained-models'  # e.g., 'ev-iot-model'

# CSV already in S3
input_data = f's3://{bucket}/daily_user.csv'

sklearn_estimator = SKLearn(
    entry_point='train_script.py',  # (see below)
    role=role,
    instance_count=1,
    instance_type='ml.m5.large',
    framework_version='1.2-1',
    py_version='py3',
    base_job_name='rf-training-job',
    output_path=f's3://{bucket}/{prefix}/output'
)

sklearn_estimator.fit({'train': input_data})
```

After running it, you should see something like this, it should say Training job completed.

```
2025-07-17 10:58:51 Starting - Starting the training job...
2025-07-17 10:59:13 Starting - Preparing the instances for training...
2025-07-17 10:59:56 Downloading - Downloading the training image.........
2025-07-17 11:01:07 Training - Training image download completed. Training in progress.2025-07-17 11:01:08,543 sagemaker-containers INFO     Imported f
ramework sagemaker_sklearn_container.training
2025-07-17 11:01:08,546 sagemaker-training-toolkit INFO     No GPUs detected (normal if no gpus installed)
2025-07-17 11:01:08,548 sagemaker-training-toolkit INFO     No Neurons detected (normal if no neurons installed)
2025-07-17 11:01:08,563 sagemaker_sklearn_container.training INFO     Invoking user training script.
2025-07-17 11:01:08,753 sagemaker-training-toolkit INFO     No GPUs detected (normal if no gpus installed)
2025-07-17 11:01:08,757 sagemaker-training-toolkit INFO     No Neurons detected (normal if no neurons installed)


2025-07-17 11:01:08,845 sagemaker-training-toolkit INFO     Exceptions not imported for SageMaker Debugger as it is not installed.
2025-07-17 11:01:08,846 sagemaker-training-toolkit INFO     Exceptions not imported for SageMaker TF as Tensorflow is not installed.
2025-07-17 11:01:52,278 sagemaker-containers INFO     Reporting training SUCCESS

2025-07-17 11:02:15 Uploading - Uploading generated training model
2025-07-17 11:02:15 Completed - Training job completed
Training seconds: 164
Billable seconds: 164
```

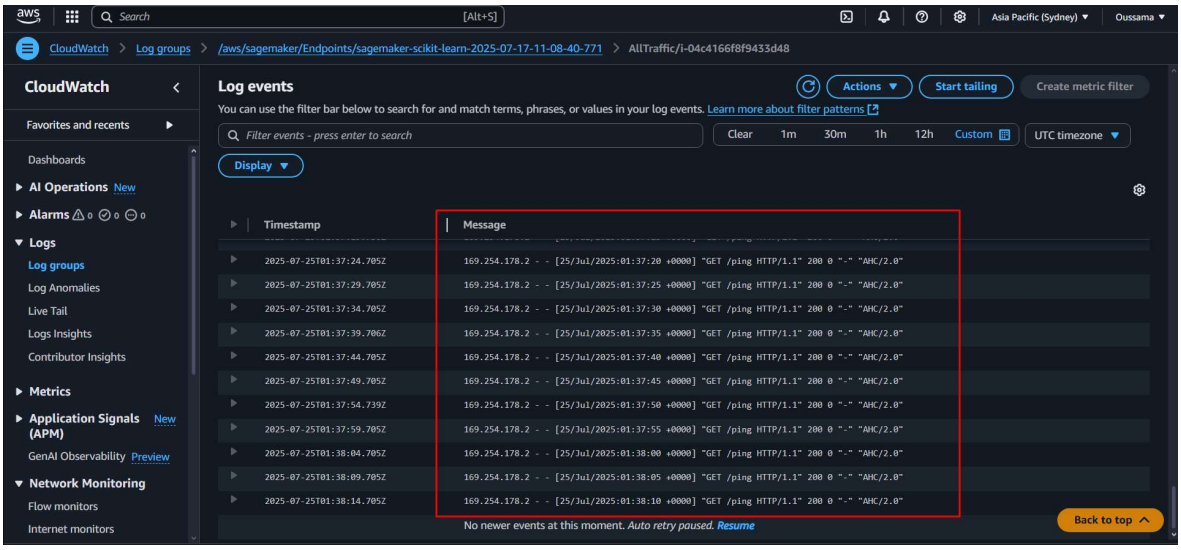
After successfully training, we can now deploy the model using the inference.py script, which you also have to paste in the Notebook (same directory as script) then run it.

```python
from sagemaker.sklearn.model import SKLearnModel

model_artifact = sklearn_estimator.model_data

sklearn_model = SKLearnModel(
    model_data=model_artifact,
    role=role,
    entry_point='inference.py',  # you define this next
    framework_version='1.2-1',
    py_version='py3'
)

predictor = sklearn_model.deploy(
    initial_instance_count=1,
    instance_type='ml.m5.large'
)
```

After it’s successful, you will only see something like this in the console

```
------!
```

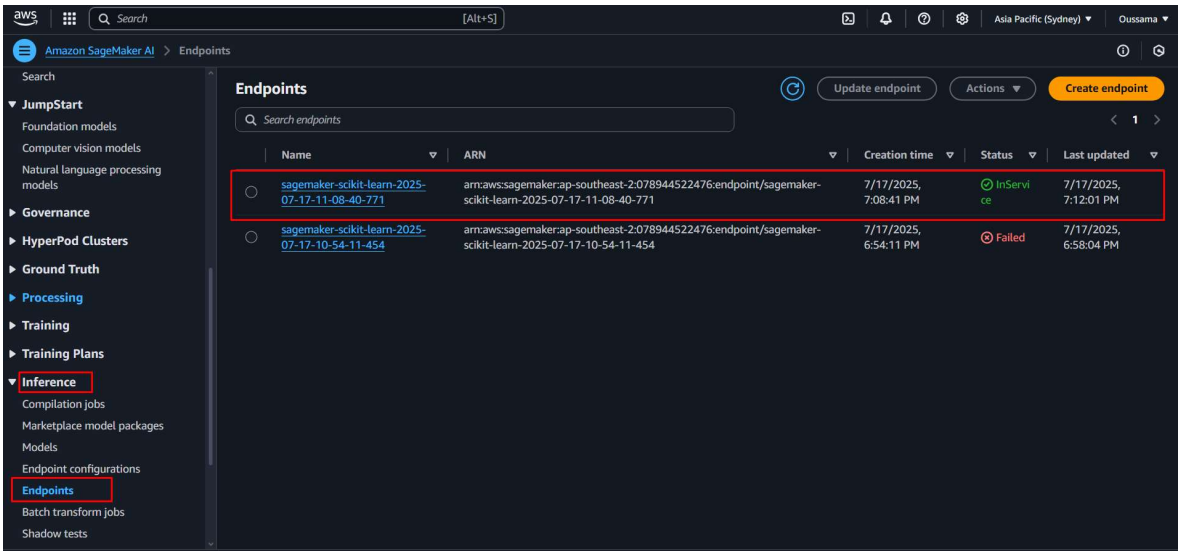
Since this won't be the best indicator, you can check CloudWatch Logs



If you see repetitive messages similar to this, it means your endpoint is working.

You can also navigate to Amazon SageMaker AI -> Inference -> **Endpoints**

It should show your endpoint is **InService**.



Now, once the model is successfully deployed, we can create our **Lambda 2** Function which will call the endpoint to perform the prediction on the sensor data that was sent to SiteWise.

```
import boto3
import pandas as pd
import numpy as np
import json
from io import BytesIO
from datetime import datetime, timezone

# === AWS clients ===
s3 = boto3.client('s3')
runtime = boto3.client('sagemaker-runtime')
sitewise = boto3.client('iotsitewise')

# === Constants ===
S3_BUCKET = 'ev-iot-dataset'
CSV_KEY = 'daily_user.csv'
ENDPOINT_NAME = 'sagemaker-scikit-learn-2025-07-17-11-08-40-771'

# === SiteWise property IDs ===
PREDICTED_TTF_ID = 'a6a6497f-6438-481c-ae68-36d680fe8be4'
FAILURE_SOON_ID = 'f95f4634-6a64-4a01-ad7e-be37bb8e746a'
TRIGGER_FLAG_ID = '0923535f-47d0-466b-bdb3-6389823d2756'
ASSET_ID = '148d1ebe-1bb6-4da4-a23e-006f3f046791'

# === Feature columns ===
FEATURE_COLUMNS = ['SOC', 'SOH', 'Charging_Cycles', 'Battery_Temp', 'Motor_RPM',
                   'Motor_Torque', 'Motor_Temp', 'Brake_Pad_Wear', 'Tire_Pressure', 'DTC']

def load_csv_from_s3():
    response = s3.get_object(Bucket=S3_BUCKET, Key=CSV_KEY)
    df = pd.read_csv(BytesIO(response['Body'].read()))
    return df

def get_latest_sitewise_soc():
    response = sitewise.get_asset_property_value(
        assetId=ASSET_ID,
        propertyId='f13b2a63-c016-4dbf-87c9-98fc0de7fdd1'  # SOC
    )
    return float(response['propertyValue']['value']['doubleValue'])

def get_matching_row(df, latest_soc):
    df_sorted = df.sort_values('SOC').reset_index(drop=True)
    closest_idx = (df_sorted['SOC'] - latest_soc).abs().idxmin()
    return df_sorted.iloc[closest_idx][FEATURE_COLUMNS]

def call_sagemaker(input_row):
    input_data = input_row.astype(float).values.reshape(1, -1)  # shape: (1, 10)
    response = runtime.invoke_endpoint(
        EndpointName=ENDPOINT_NAME,
        ContentType='application/json',
        Body=json.dumps(input_data.tolist())
    )
    result = json.loads(response['Body'].read().decode())

    if isinstance(result, dict) and 'predictions' in result:
        ttf = result['predictions'][0]
    elif isinstance(result, list):
        ttf = result[0]
    else:
        raise ValueError("Unexpected SageMaker response format")

    return float(ttf)
```

```python
def push_to_sitewise(ttf_value, failure_soon):
    now_ms = int(datetime.now(timezone.utc).timestamp() * 1000)
    entries = [
        {
            'entryId': '1',
            'assetId': ASSET_ID,
            'propertyId': PREDICTED_TTF_ID,
            'propertyValues': [{
                'value': {'doubleValue': ttf_value},
                'timestamp': {'timeInSeconds': now_ms // 1000, 'offsetInNanos': 0},
                'quality': 'GOOD'
            }]
        },
        {
            'entryId': '2',
            'assetId': ASSET_ID,
            'propertyId': FAILURE_SOON_ID,
            'propertyValues': [{
                'value': {'booleanValue': failure_soon},
                'timestamp': {'timeInSeconds': now_ms // 1000, 'offsetInNanos': 0},
                'quality': 'GOOD'
            }]
        },
        {
            'entryId': '3',
            'assetId': ASSET_ID,
            'propertyId': TRIGGER_FLAG_ID,
            'propertyValues': [{
                'value': {'booleanValue': False},
                'timestamp': {'timeInSeconds': now_ms // 1000, 'offsetInNanos': 0},
                'quality': 'GOOD'
            }]
        }
    ]
    sitewise.batch_put_asset_property_value(entries=entries)

# === Lambda Handler ===
def lambda_handler(event, context):
    try:
        df = load_csv_from_s3()
        latest_soc = get_latest_sitewise_soc()
        input_row = get_matching_row(df, latest_soc)
        ttf = call_sagemaker(input_row)
        failure_soon = ttf < 48
        push_to_sitewise(ttf, failure_soon)

        return {
            'statusCode': 200,
            'body': json.dumps({
                'PredictedTTF': ttf,
                'FailureSoon': failure_soon
            })
        }

    except Exception as e:
        return {
            'statusCode': 500,
            'body': json.dumps({'error': str(e)})
        }
```

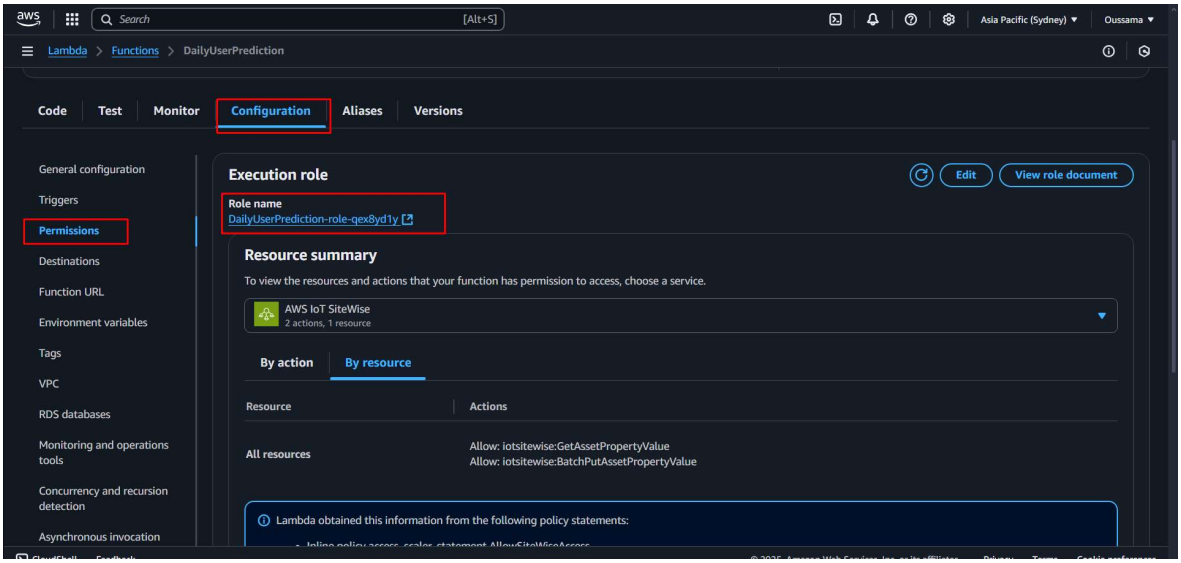
This Lambda 2 function is triggered whenever a new row of sensor data is pushed into SiteWise by Lambda 1 as mentioned in the IoT Sitewise section.

Lambda 2:

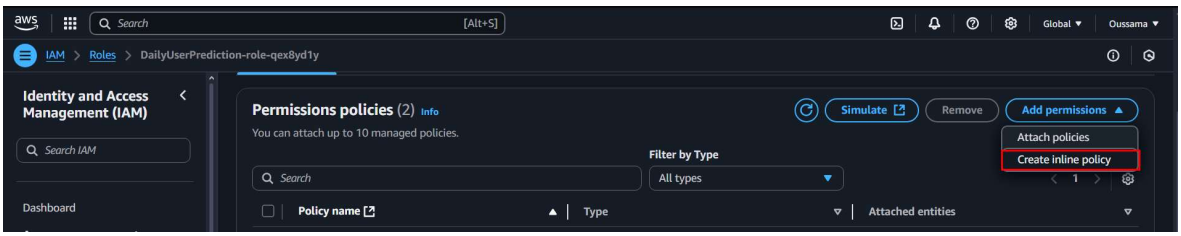
1. Identifies the matching sensor data row from a historical CSV in S3 based on latest SOC (State of Charge) received by SiteWise.
2. Uses that row as input to a trained SageMaker model endpoint (Random Forest).
3. Predicts TTF (Time to Failure) and determines whether the vehicle is likely to fail soon (if TTF < 48 hours).
4. Sends the prediction results (PredictedTTF and FailureSoon) back to SiteWise.
5. Resets the trigger flag (TriggerPrediction) in SiteWise to False so the process doesn't repeat until explicitly re-triggered.

You need to make sure the Lambda function has the necessary permissions.

Go to Lambda 2 -> Configuration -> Permissions -> **Execution Role**



Add permissions -> Create inline policy



Paste in this policy and save

```
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Sid": "AllowS3ReadAccess",
      "Effect": "Allow",
      "Action": [
        "s3:GetObject"
      ],
      "Resource": [
        "arn:aws:s3:::ev-iot-dataset/daily_user.csv"
      ]
    },
    {
      "Sid": "AllowSiteWiseAccess",
      "Effect": "Allow",
      "Action": [
        "iotsitewise:GetAssetPropertyValue",
        "iotsitewise:BatchPutAssetPropertyValue"
      ],
      "Resource": "*"
    },
    {
      "Sid": "AllowSageMakerInvoke",
      "Effect": "Allow",
      "Action": [
        "sagemaker:InvokeEndpoint"
      ],
      "Resource": "arn:aws:sagemaker:ap-southeast-2:078944522476:endpoint/sagemaker-scikit-learn-2025-07-17-11-08-40-771"
    },
    {
      "Sid": "AllowLogs",
      "Effect": "Allow",
      "Action": [
        "logs:CreateLogGroup",
        "logs:CreateLogStream",
        "logs:PutLogEvents"
      ],
      "Resource": "*"
    }
  ]
}
```

This ensures that the Lambda has the necessary permissions to perform actions on S3, IoT SiteWise, SageMaker endpoint, and CloudWatch Logs.

Amazon S3

Amazon Simple Storage Service (Amazon S3) is a scalable, high-availability object storage service provided by AWS. It is designed to store and retrieve any amount of data from anywhere on the web. S3 is commonly used for storing application assets, backups, data lakes, datasets, logs, and even machine learning models. For our project case, Amazon S3 plays a central role as the data and ML model storage layer.

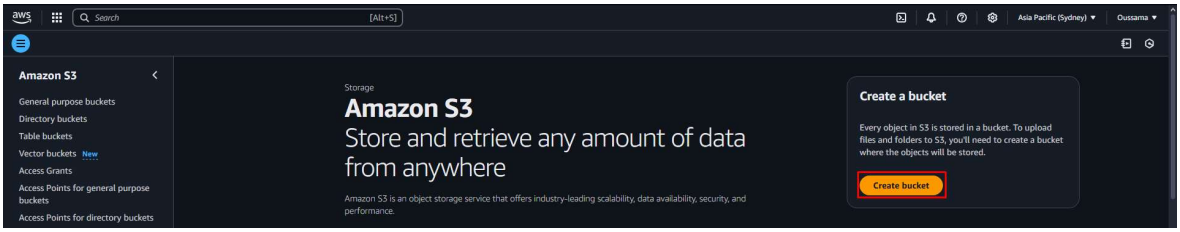
The project uses a CSV file (daily_user.csv) stored in S3 that contains real electric vehicle sensor data. This file includes time-series readings for various parameters such as state of charge (SOC), state of health (SOH), battery and motor temperature, RPM, brake pad wear, and more. A Lambda function is triggered every 15 minutes to simulate real-time sensor updates by reading one row at a time from this CSV file in S3 and pushing it to AWS IoT SiteWise.

In addition to the dataset, the training script (train_script.py) is also stored in S3. This allows Amazon SageMaker to access and execute the script directly during the model training phase. Hosting the script in S3 ensures version control, reusability, and easy updates without modifying the training infrastructure.

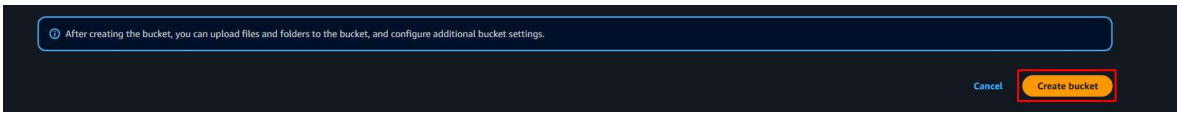
This approach uses S3 as a central, serverless storage solution that is accessible by multiple AWS services. It allows the system to remain stateless and scalable, while also making it easy to update the dataset or model training logic when improvements or changes are required.

Setting up Amazon S3

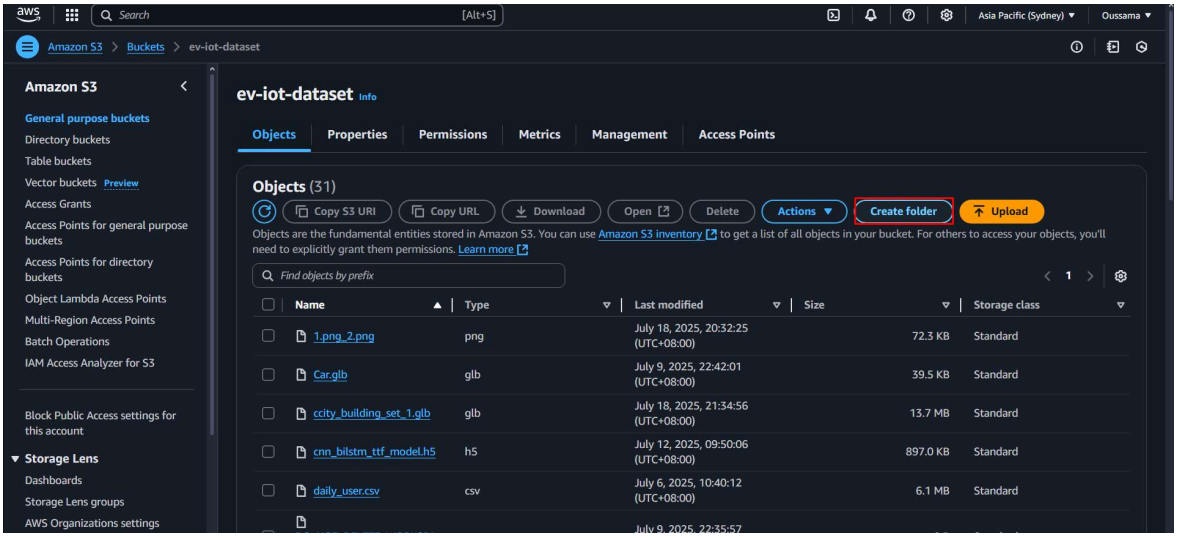
1. Go to Amazon S3 and click on **Create bucket**



2. Enter a name for the bucket and click Create bucket



3. Create a folder to use for our SageMaker



AWS IoT TwinMaker

AWS IoT TwinMaker is a service that enables the creation of digital twins, or in other words, the virtual representations of physical systems that combine real-time data, historical data, 3D models, and metadata into a single, interactive environment. It allows organizations to visualize and analyze complex systems such as factories, vehicles, and buildings, helping them better understand system behavior, detect anomalies, and make informed decisions.

In this project, AWS IoT TwinMaker is used to build a digital twin of electrical vehicles which brings together sensor data, machine learning predictions, and a 3D visual interface for real-time monitoring and diagnostics.

How It's Used in This Project:

1. **Digital Representation of the EV**
   TwinMaker provides a 3D scene that visually represents components of the electric vehicle, which includes the battery system, motor, tires, and brakes. This scene is defined using a 3D asset model and metadata, allowing each component to be linked to actual data streams coming from AWS IoT SiteWise.

2. **Integration with AWS IoT SiteWise**
   The core of the TwinMaker environment relies on SiteWise to provide real-time and historical telemetry data. As sensor values are sent to SiteWise every 15 minutes by a Lambda function, TwinMaker accesses these values to update the 3D visualizations accordingly. For example, a user can see changes in battery temperature, tire pressure, or brake pad wear over time.

3. **Visualization of Predictions**
   The prediction Lambda function processes the most recent sensor data and returns two values to SiteWise: the **Predicted Time To Failure (TTF)** and a **FailureSoon classification (True/False)**. These prediction values are also shown in the TwinMaker interface, allowing users to not only view the current state of the vehicle but also see insights about potential upcoming failures. For instance, the visual dashboard can highlight a warning near the battery if a failure is likely within the next 48 hours.

4. **User Interaction and Monitoring**
   TwinMaker allows engineers or maintenance teams to interact with the digital twin of the EV. Users can zoom into specific components, track sensor trends over time, and understand the current health and risk status of each subsystem, and all of that could be done in a unified 3D dashboard.

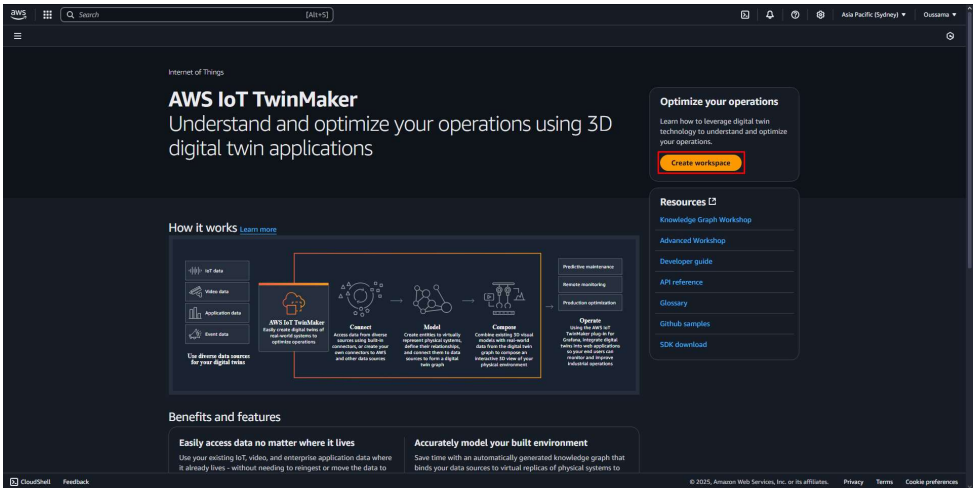
5. **TwinMaker is Passive (Simulation Needs External Control)**
   It's important to note that AWS IoT TwinMaker does not actively request or trigger new data updates. It is designed to visualize data only, not to control when or how data is collected or processed. Because of this, we implement Amazon EventBridge to trigger the first Lambda function every 15 minutes, simulating real-time data input from a vehicle. To keep track of which row of the dataset has been sent, we use AWS Systems Manager Parameter Store as a counter. Additionally, once new sensor data is pushed, a flag (TriggerPrediction = true) is set in SiteWise to activate the second Lambda function, which runs the machine learning model. This architecture ensures that TwinMaker continuously receives fresh and meaningful data, despite not having the ability to initiate data flow on its own.

Summary

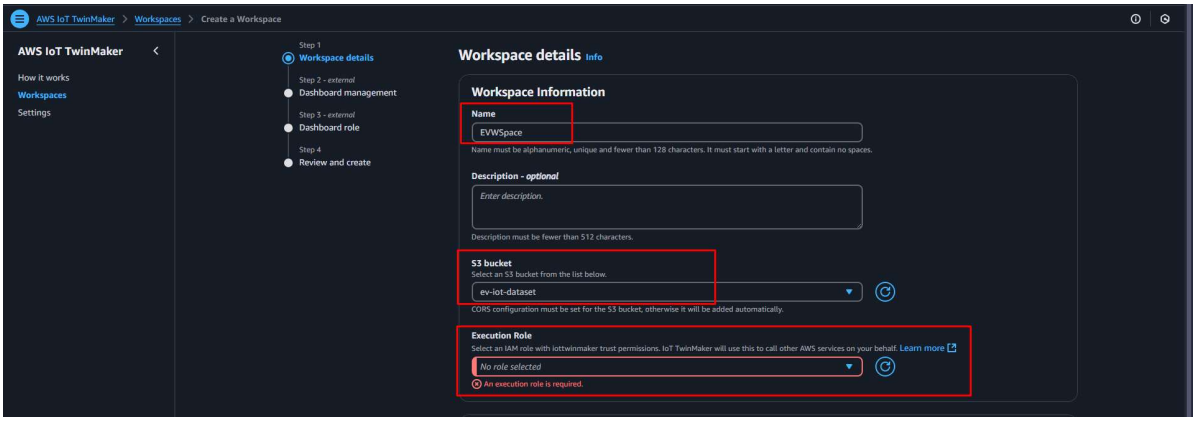
By integrating AWS IoT TwinMaker into the project, we bridge the gap between raw sensor data and user-friendly, intuitive visualization. Instead of viewing data in isolated tables or logs, users can now interact with a realistic digital twin of the EV that reflects both live telemetry and predictive insights. However, since TwinMaker does not control the data ingestion process, we simulate real-time behavior using a combination of EventBridge for scheduling, Systems Manager for tracking progress, and a two-step Lambda workflow to manage data sending and ML prediction. This setup provides a complete and functional simulation of how a smart vehicle monitoring system would work in a real-world IoT deployment.

Setting up IoT TwinMaker

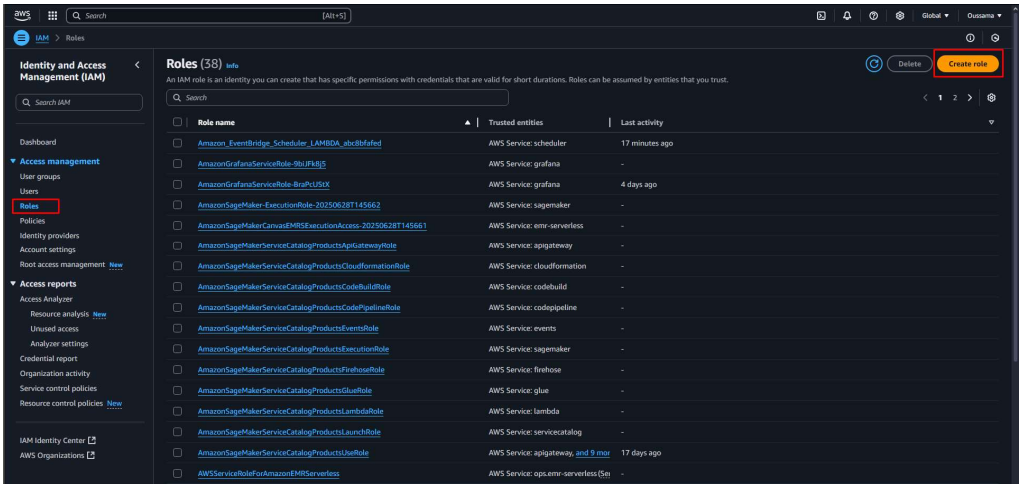
1. Go to AWS IoT TwinMaker and click on **Create workspace**



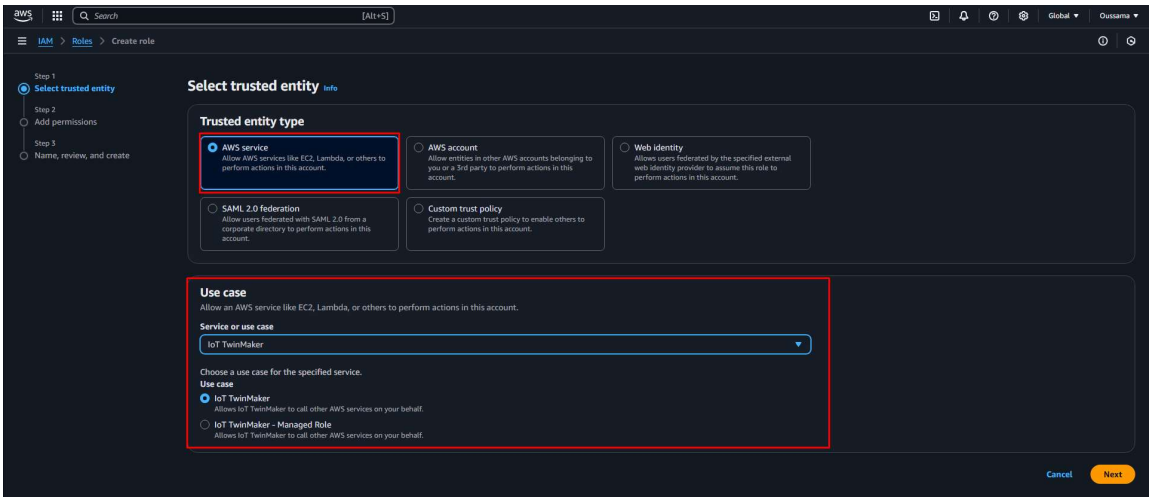
2. Enter a name for the workspace
3. Select the S3 bucket you created earlier



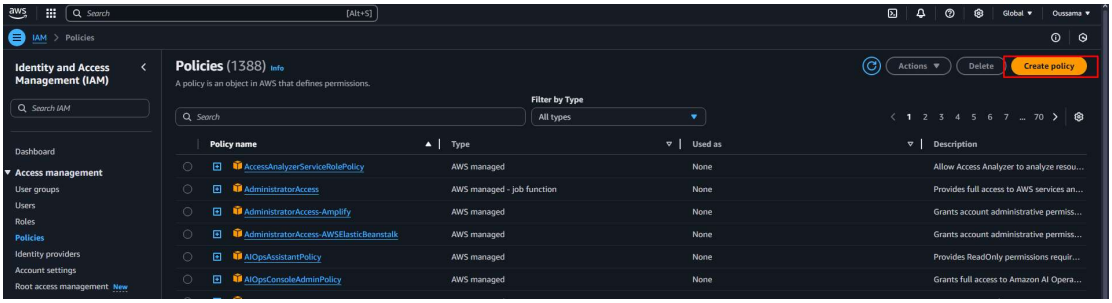
4. To create an Execution Role, go to IAM -> Roles -> **Create role**



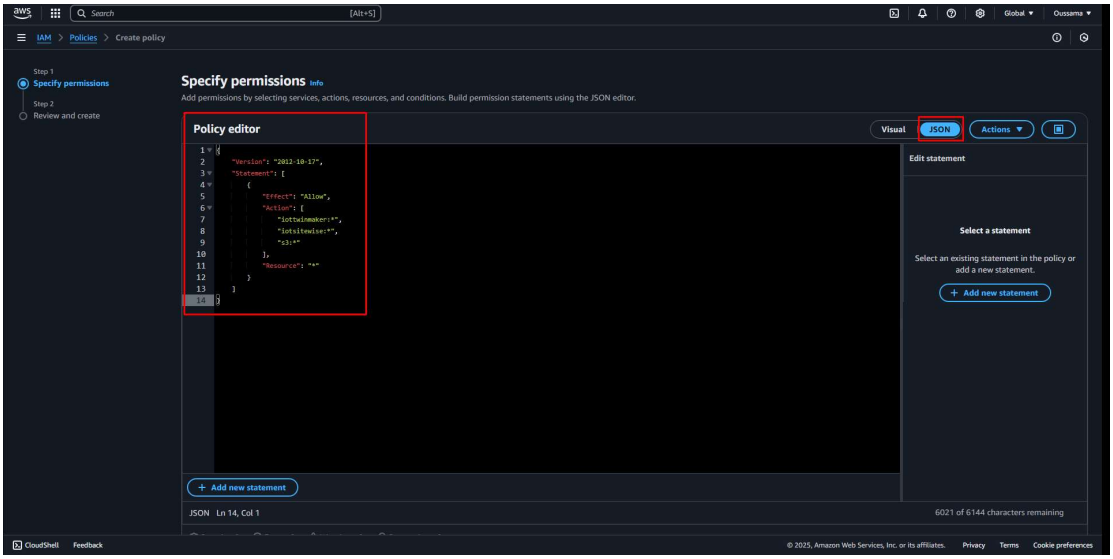
5. Under Trusted entity type, select **AWS service**
6. Under Use case, select **IoT TwinMaker**, for use case click “**IoT TwinMaker**”



7. After you click Next, to create a Policy open another IAM tab, go to Policies -> **Create policy**

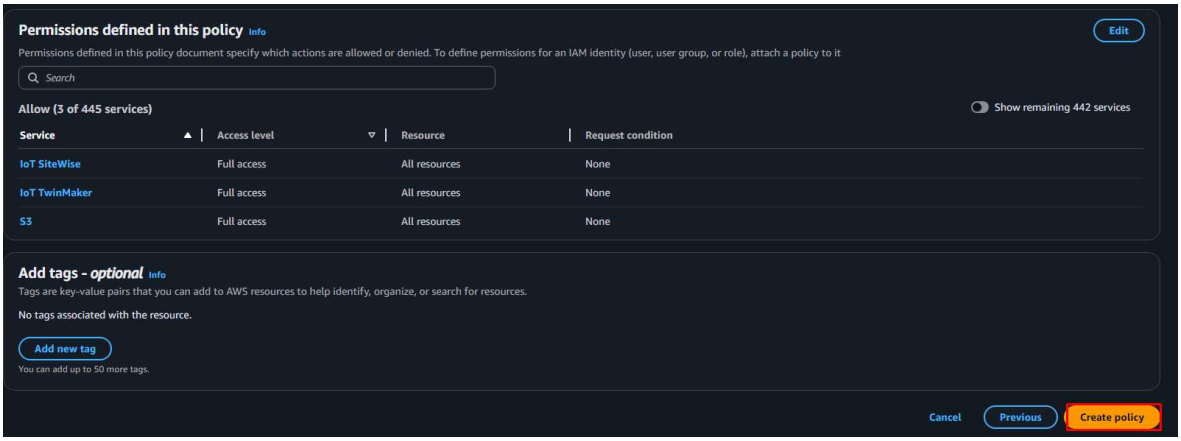


8. Click on JSON and paste in this policy

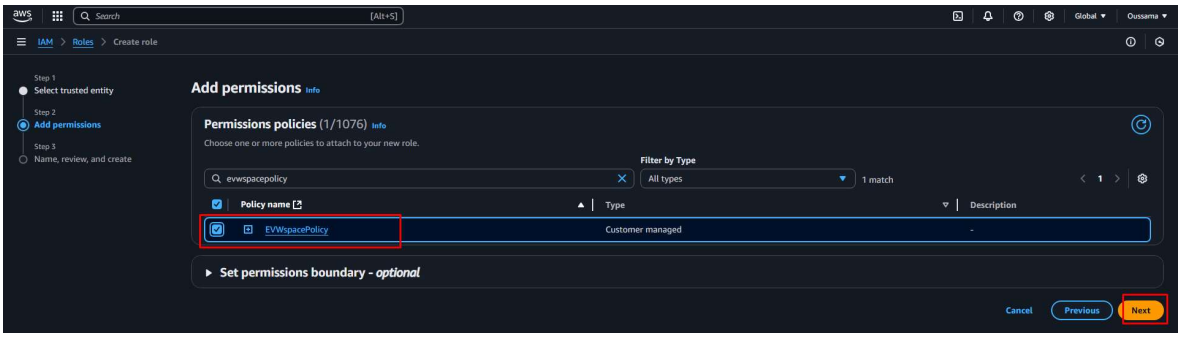


9. After clicking Next, enter a name for the policy.

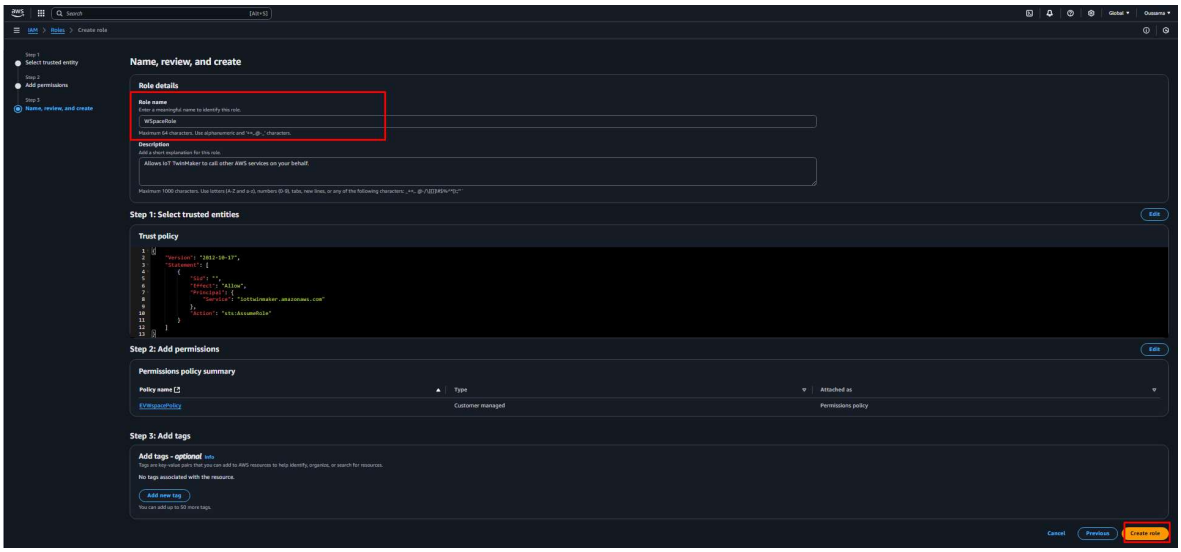
10. After confirming the permissions you set, click **Create policy**



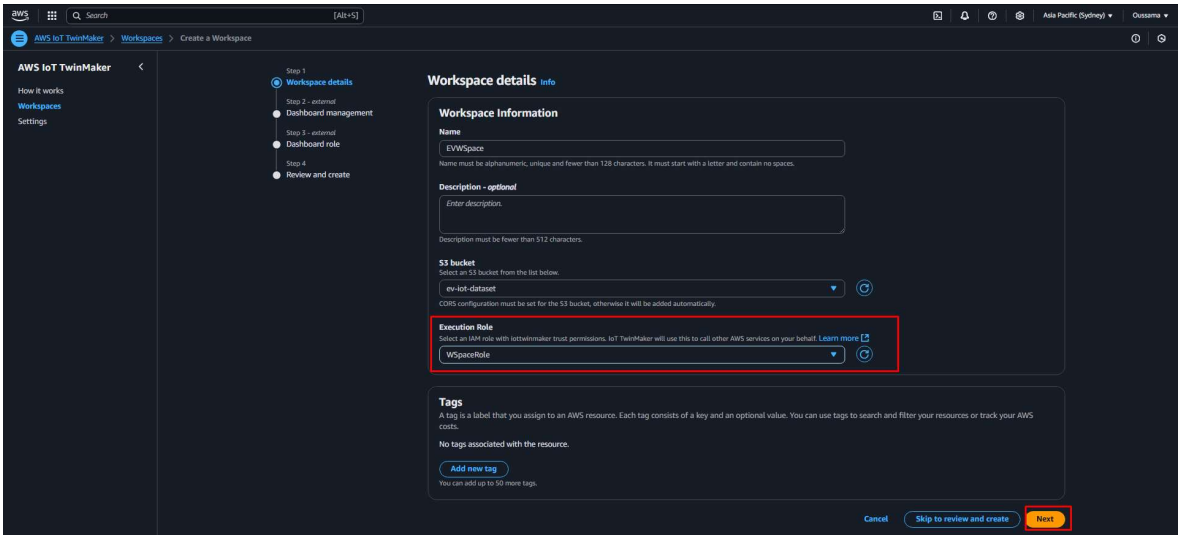
11. Now go back to the Execution Role we were creating, and select the policy we just created



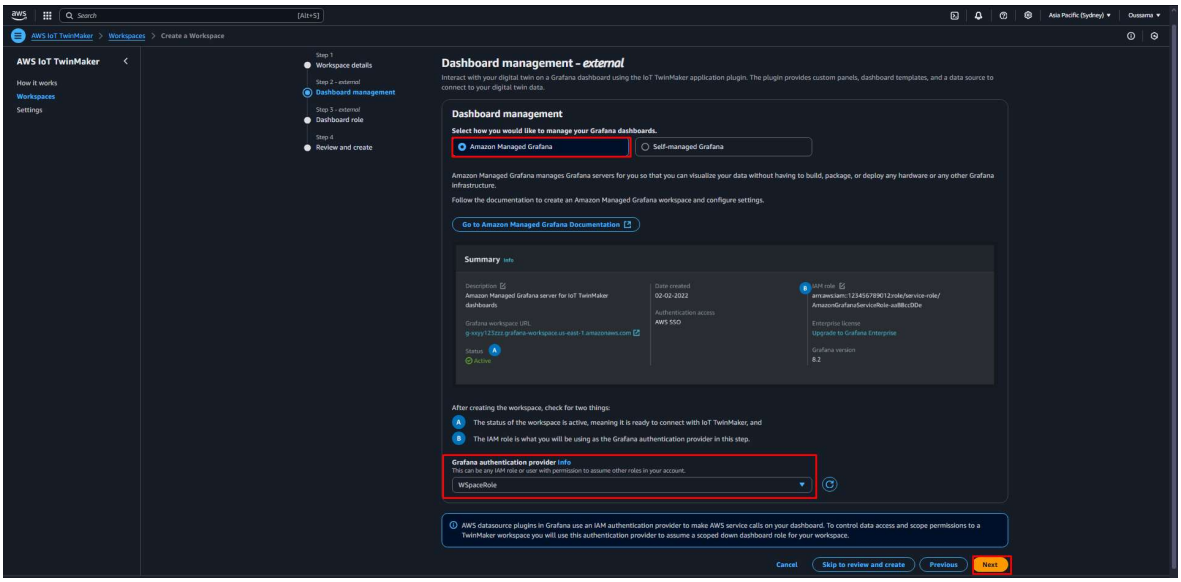
12. Give the role a name now and click Create role



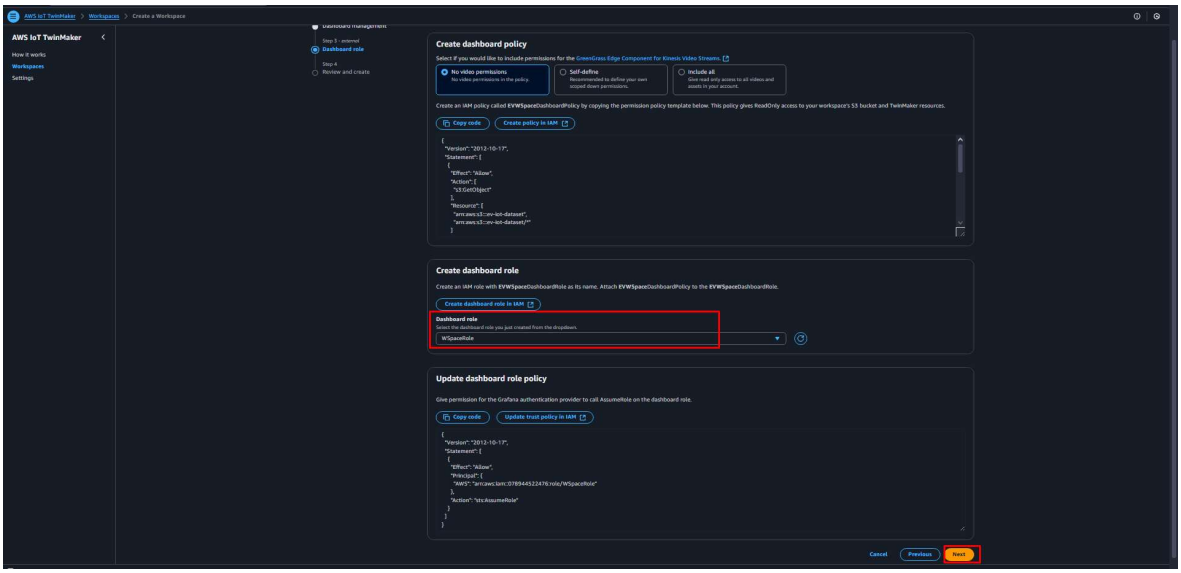
13. Now go back to TwinMaker and select the role we just created



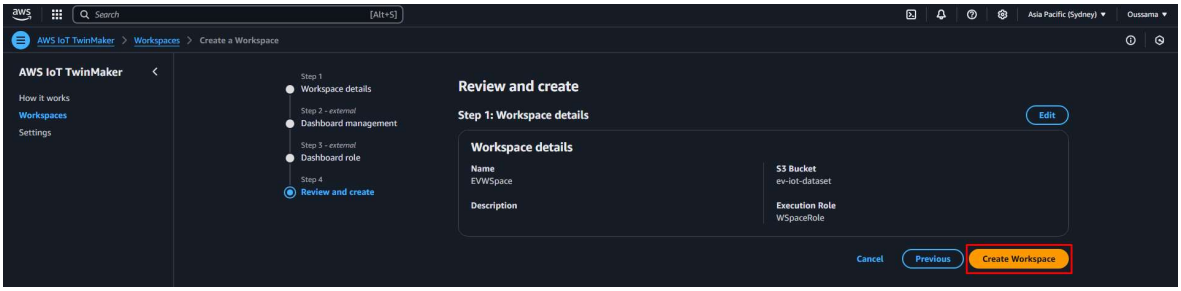
14. Under Dashboard management, select **Amazon Managed Grafana**
15. For **Grafana authentication provider**, select the same role we created earlier



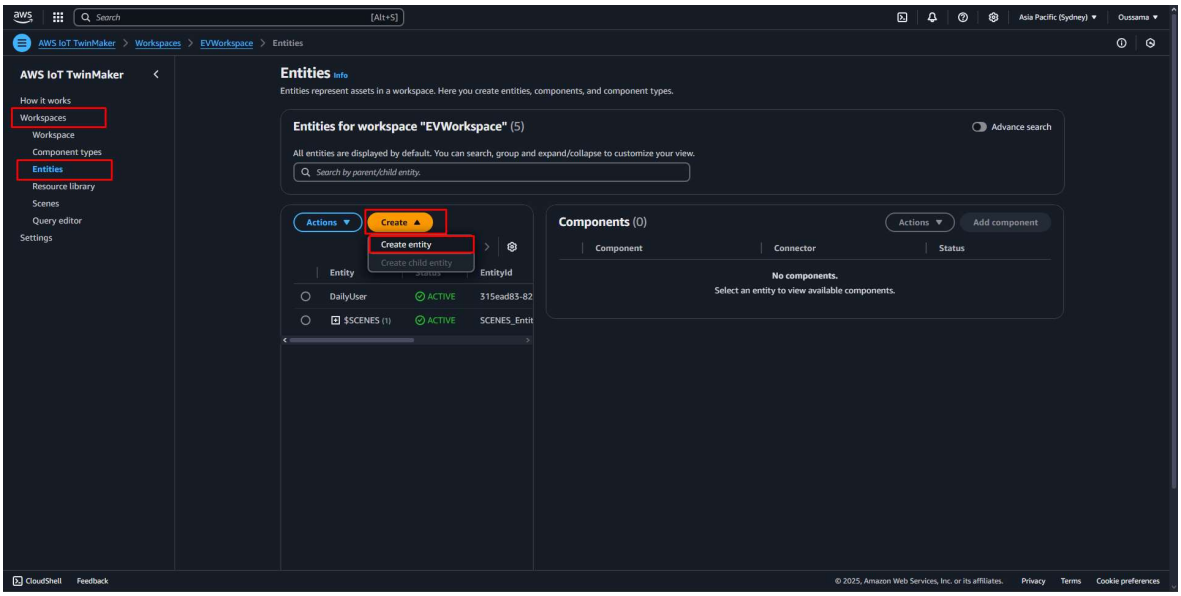
16. Under Dashboard role, select the same one we created earlier



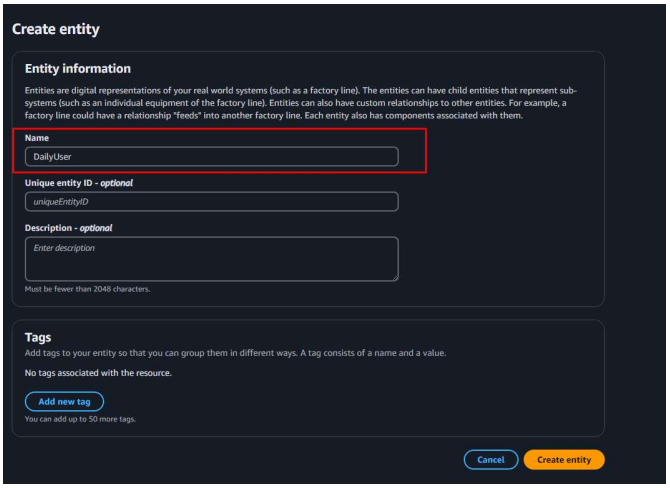
17. Click on **Create Workspace**



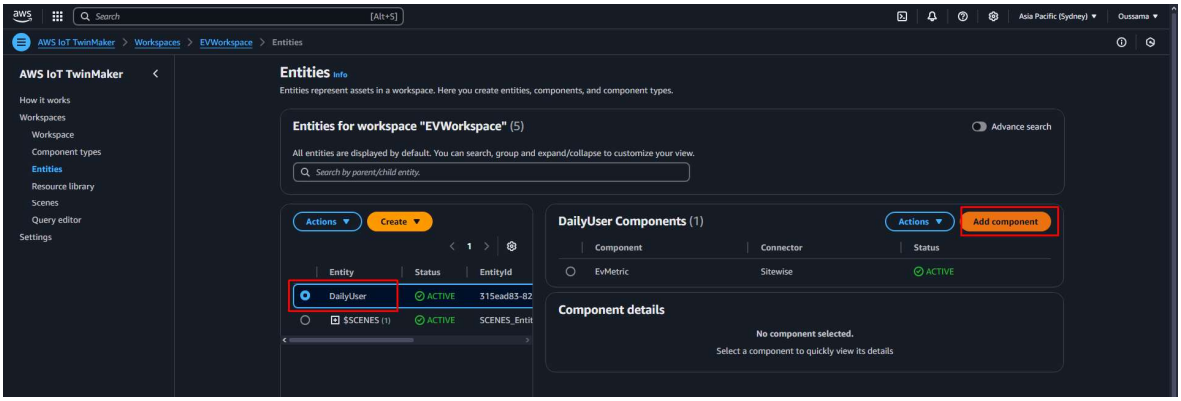
18. Go to your Workspace -> Entities -> Create -> **Create entity**



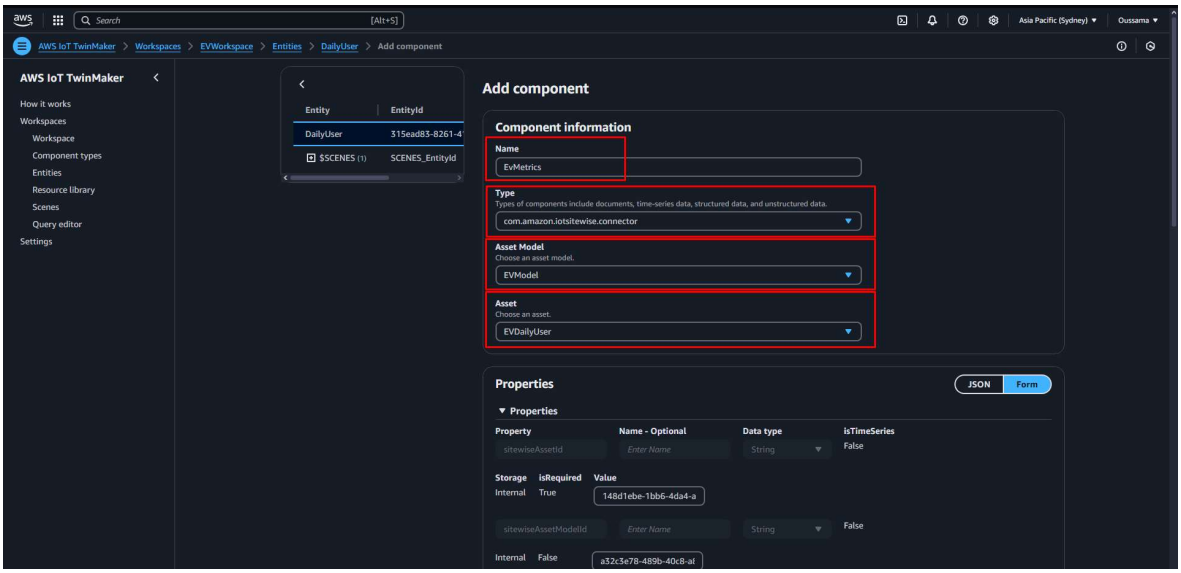
19. Give your entity a name and click **Create entity**



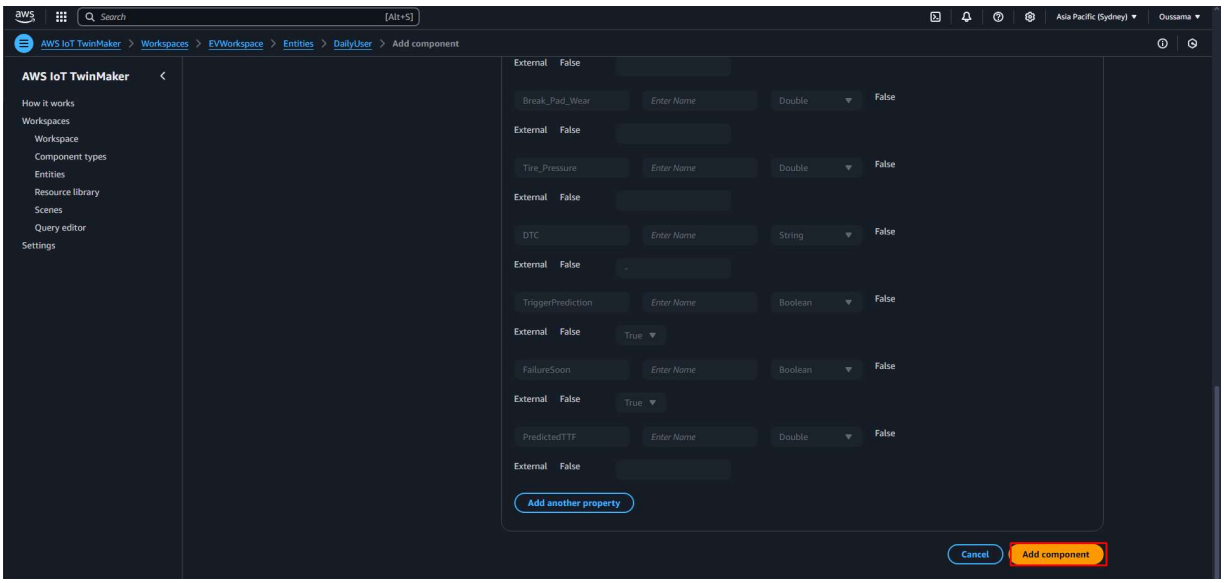
20. Go to the entity you just created and click on **Add component**



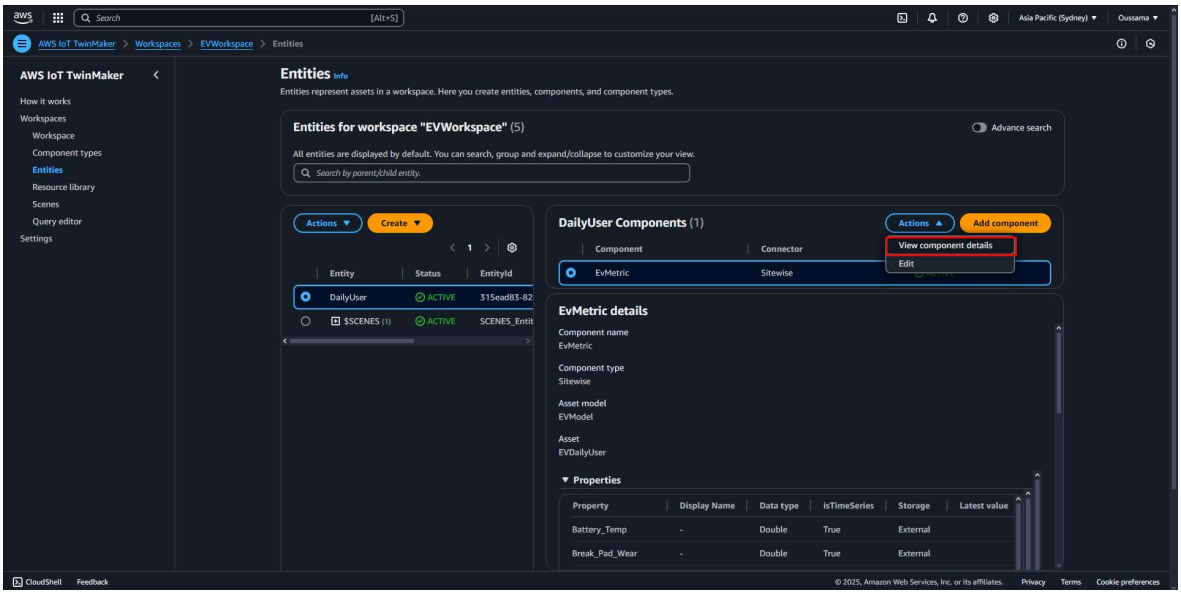
21. Enter a name for your component
22. Under Type, select **com.amazon.iotsitewise.connector**
23. Under Asset Model, pick the one we created in the IoT Sitewise step
24. Under Asset, choose the one we also created in IoT Sitewise



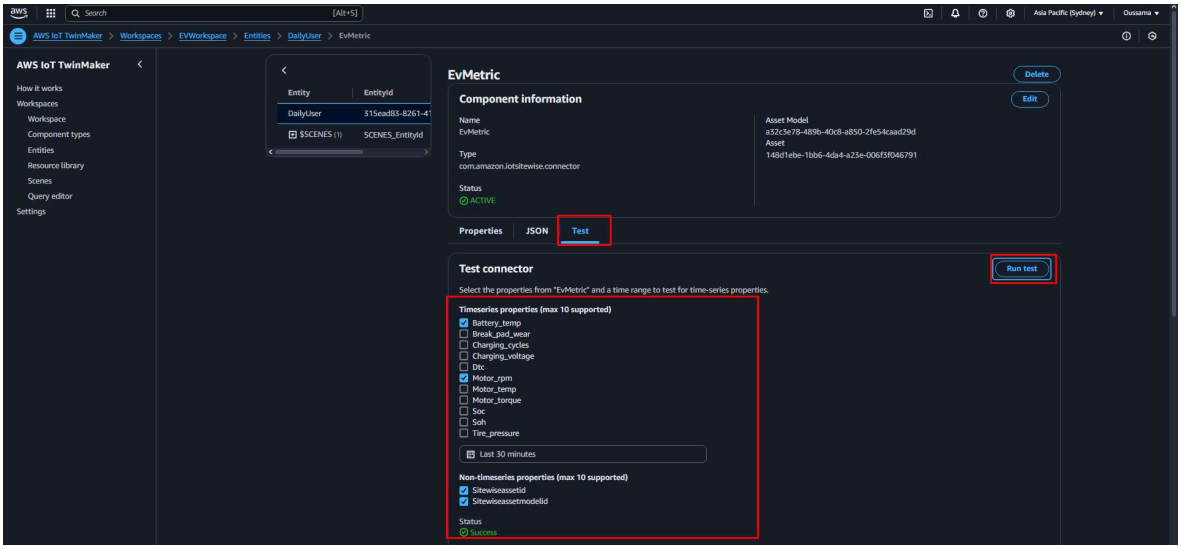
25. Once you confirm all your properties are showing, click Add component



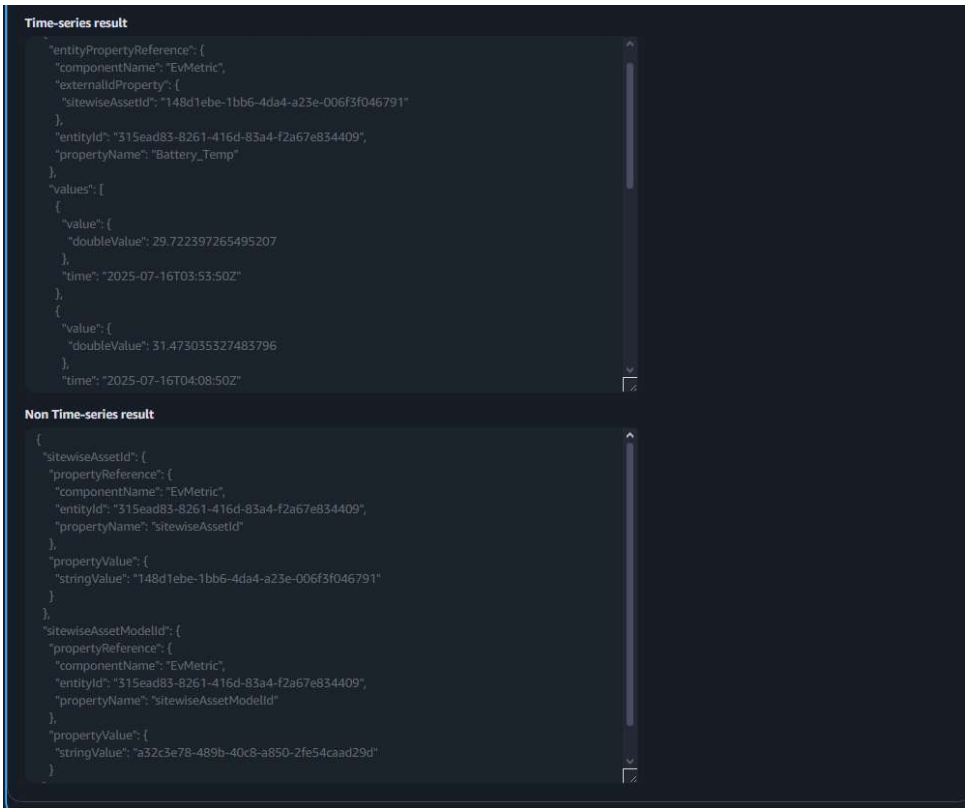
26. Now to test that we are receiving both time-series and non-time series properties data streams from IoT SiteWise, select your entity then your component, then go to Actions -> **View component details**



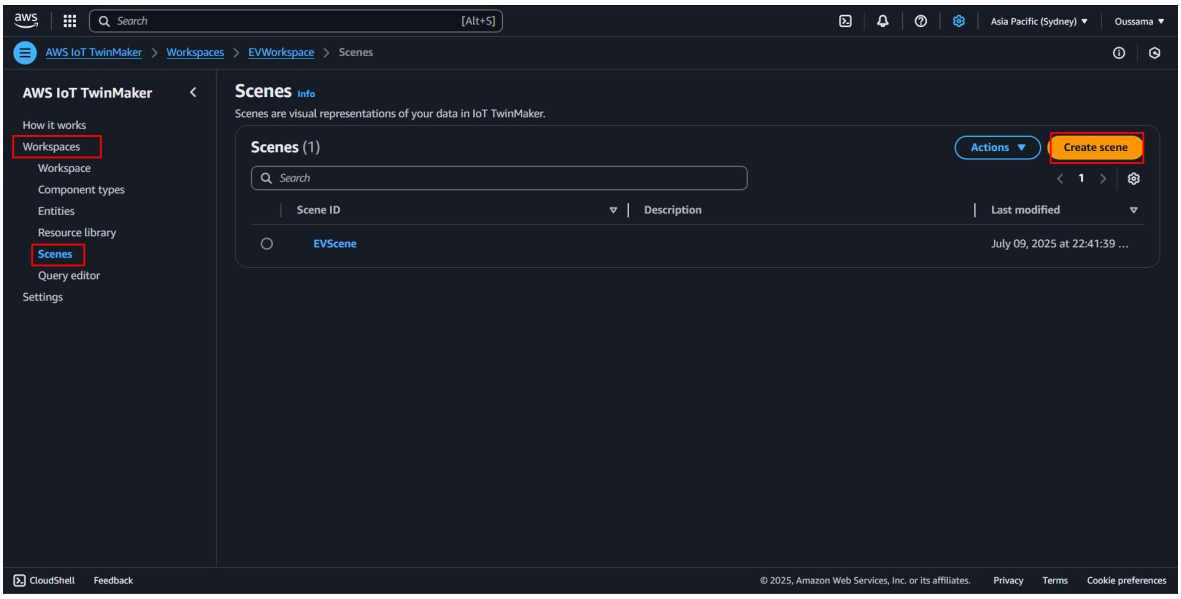
27. Click on Test, then under Timeseries properties, select any properties you want to test, choose the time frame, and under Non-timeseries properties, it will be our Asset ID and Model ID from SiteWise



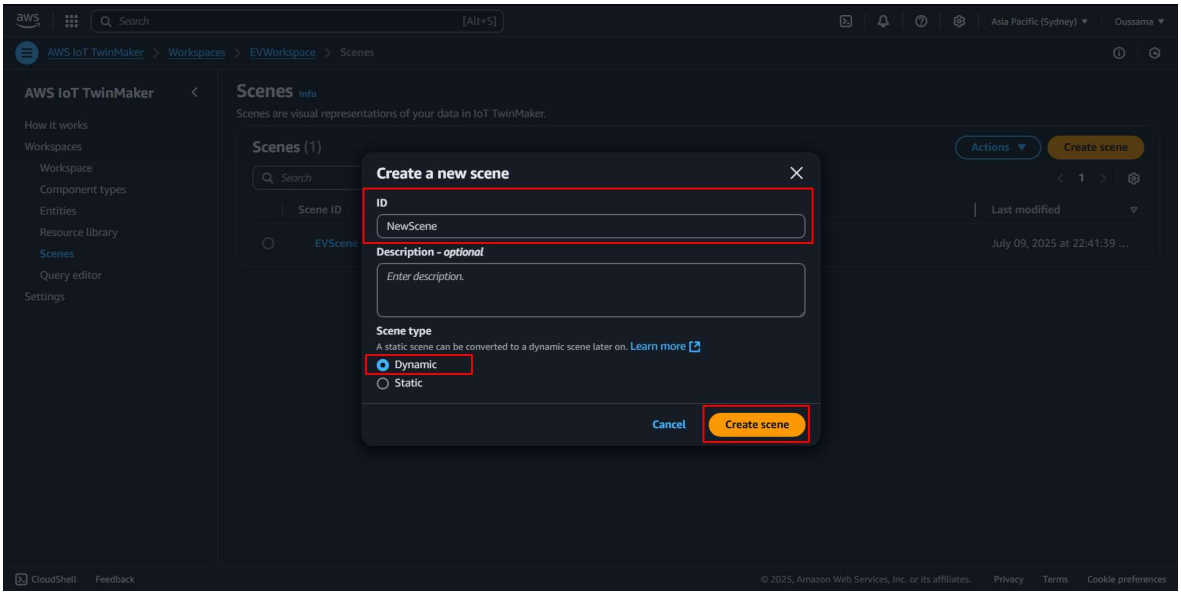
28. You should be able to see the outcome results for both time-series and non-timeseries



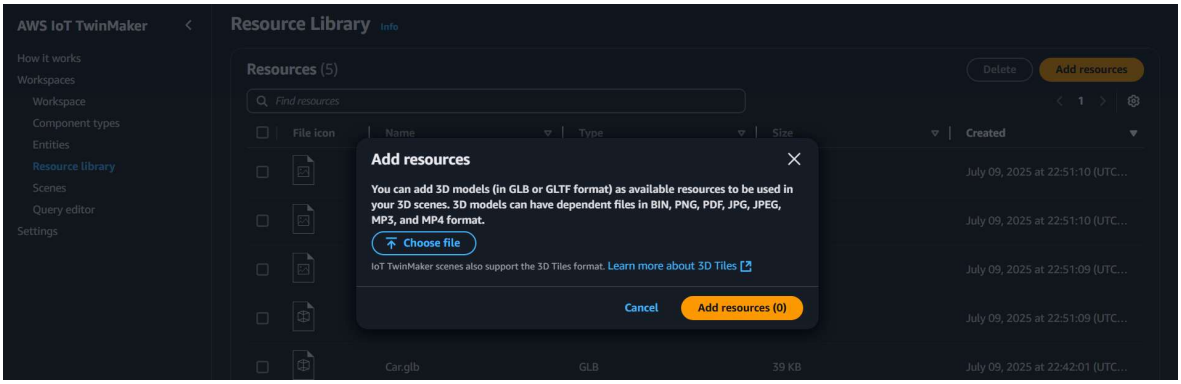
29. Now go to Workspaces -> Scenes -> **Create scene**



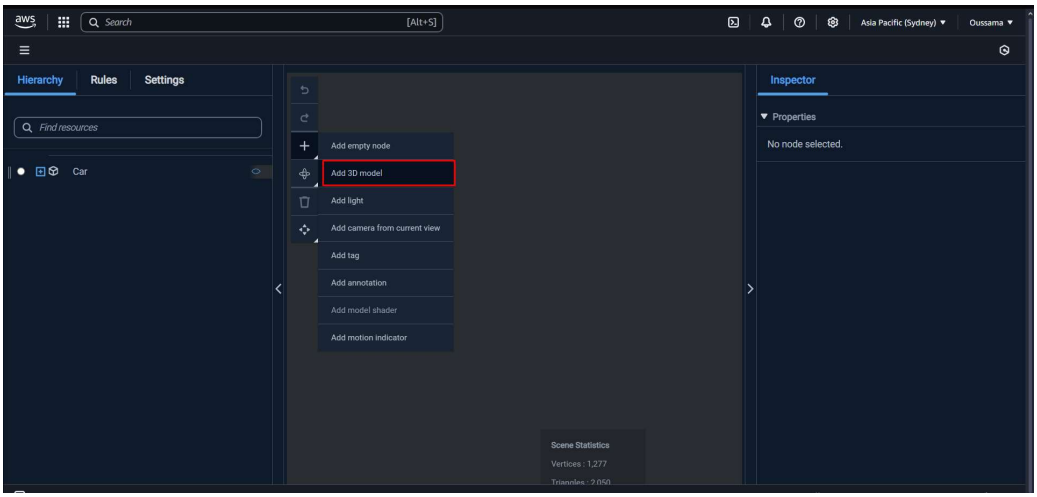
30. Enter a name for your scene
31. Make sure Scene type is **Dynamic**
32. Click **Create scene**

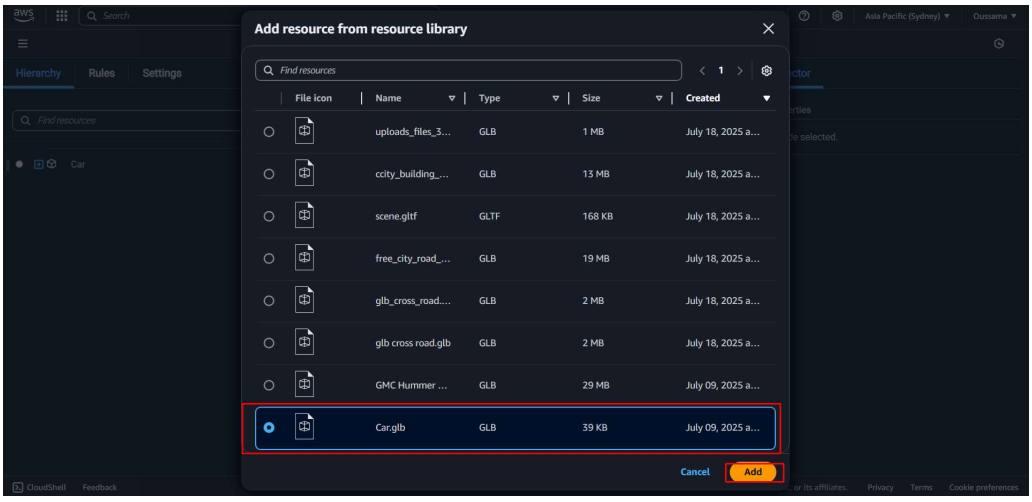


33. Go to Workspaces -> Resource library -> **Add resources**
34. Attach the resources you will need when setting up your scene

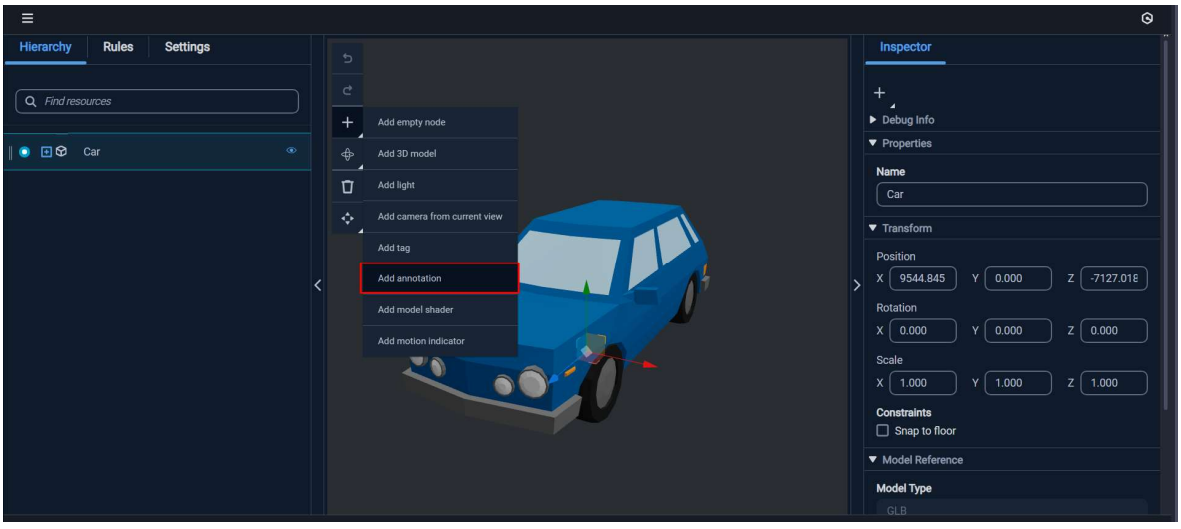


35. Go to the Scene you just created, click on Add 3D Model, choose resources you want

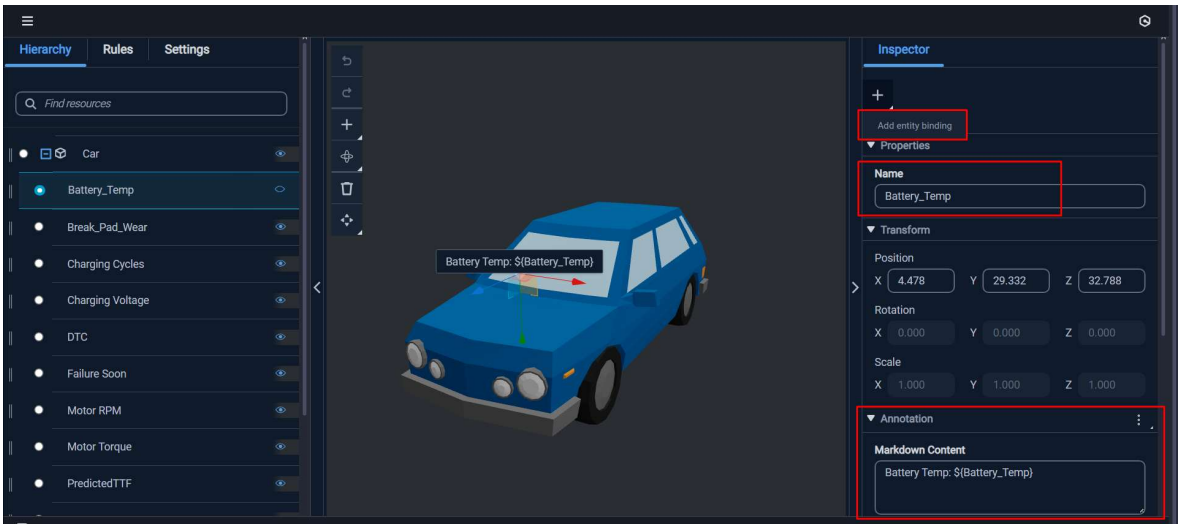




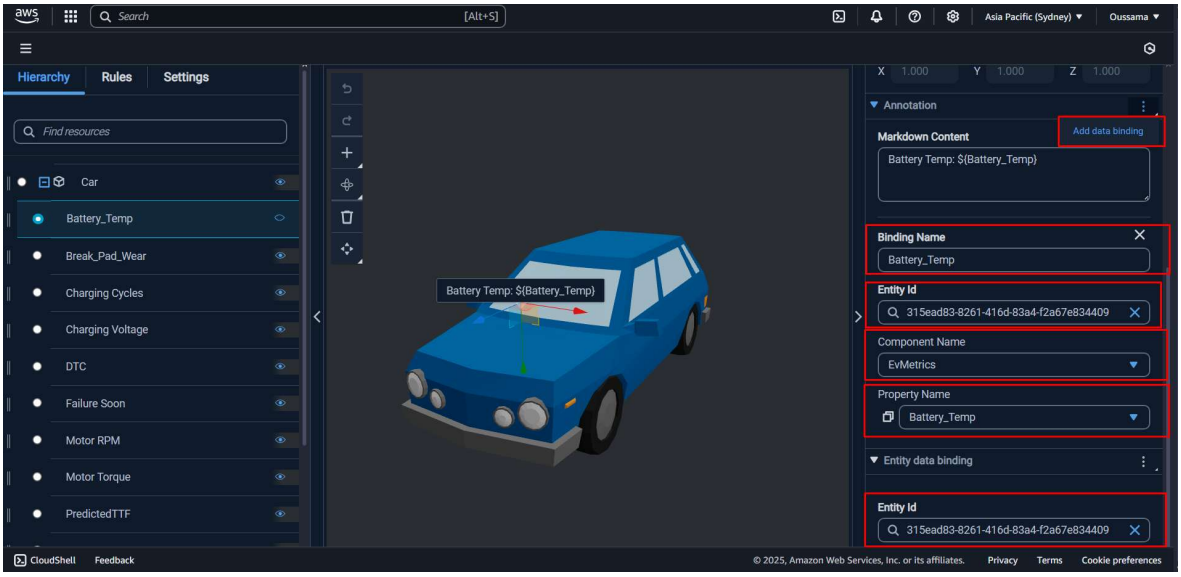
36. Click on Add Annotation



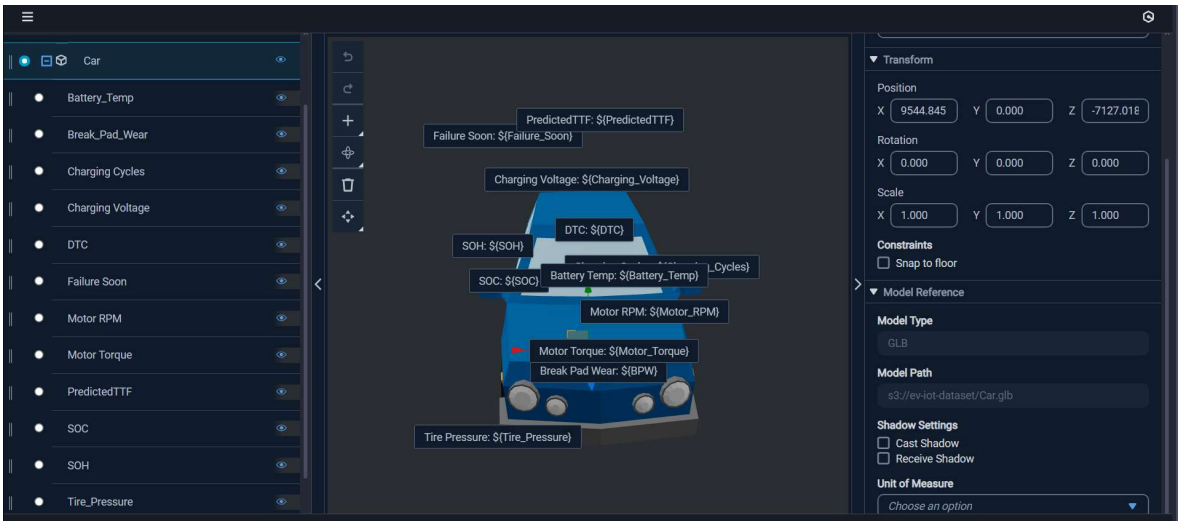
37. Add entity binding, name, and markdown content



38. Add data binding, enter the data binding name placeholder *Battery_Temp*
39. Select your entity ID for both the Annotation and Entity data binding sections
40. Select your Component Name and Property Name

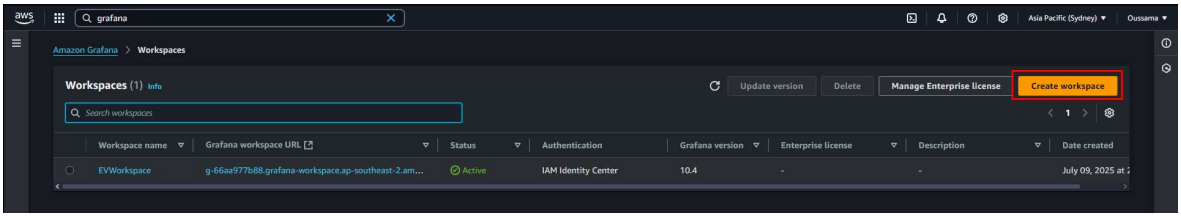


41. Repeat the same steps for all the properties you need

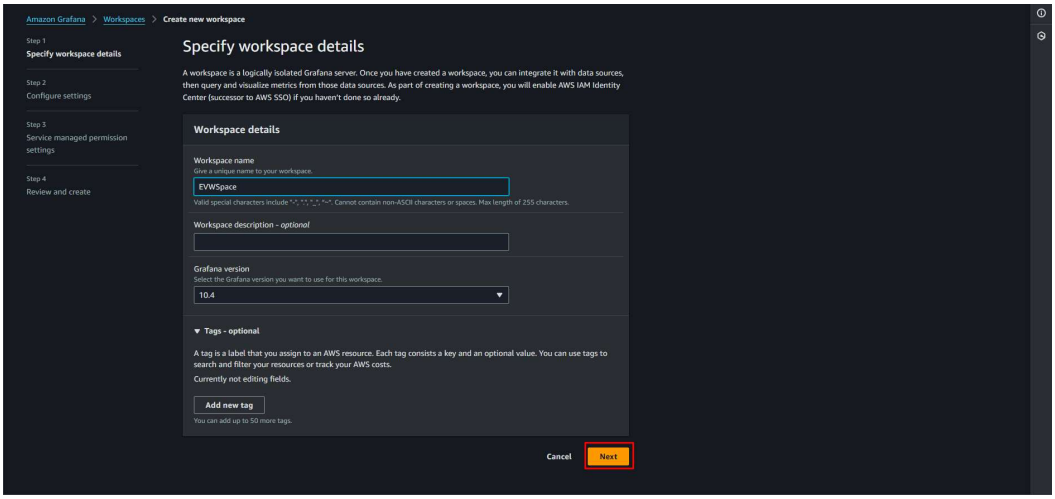


Amazon Managed Grafana

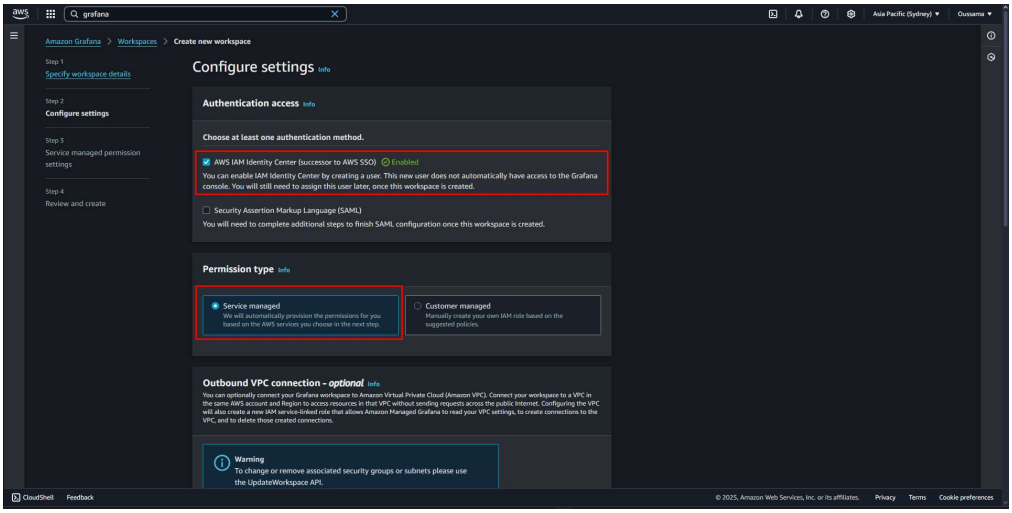
1. Go to Amazon Grafana -> Create Workspace



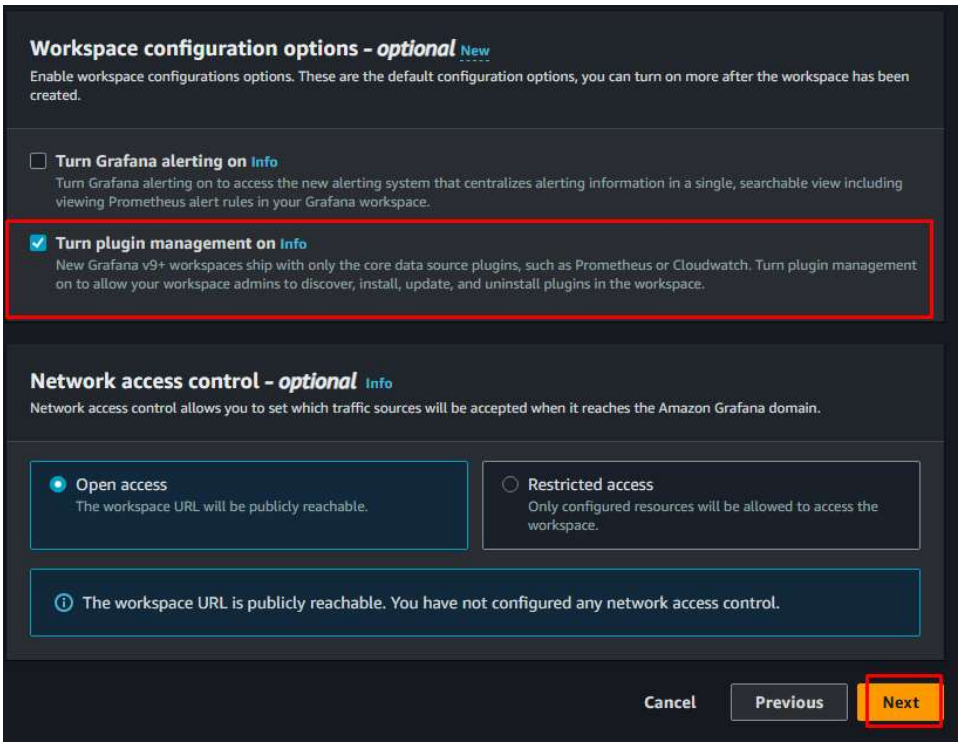
2. Enter Workspace Name and click Next



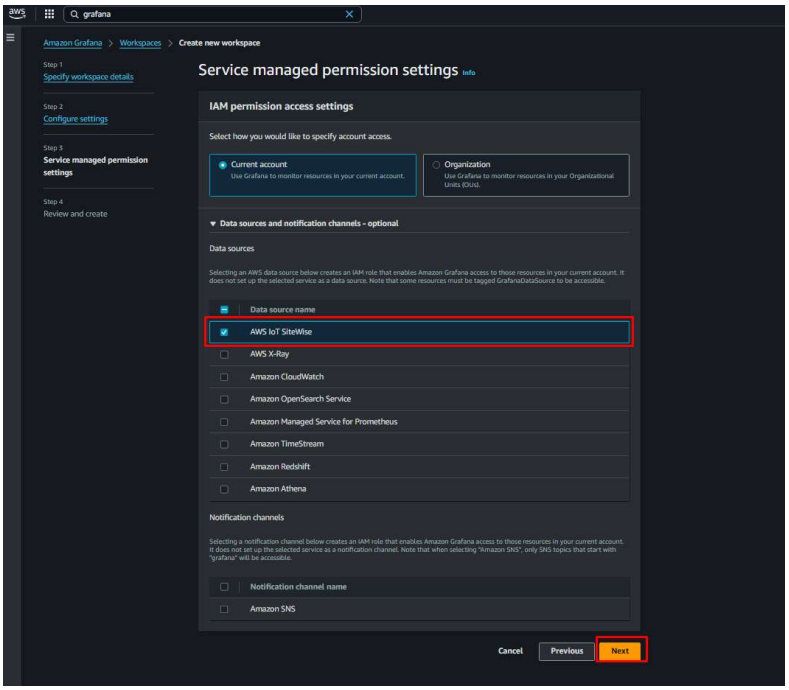
3. Under Authentication access, click AWS IAM Identity Center
4. Under Permission type, click Service managed



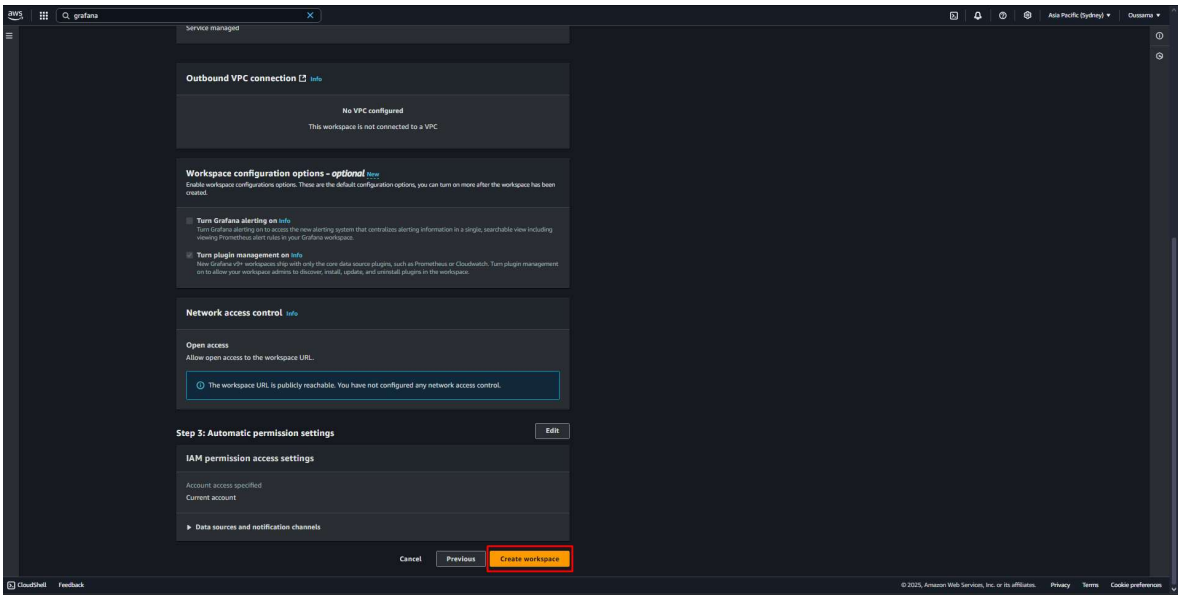
5. Scroll down, enable Plugin Management



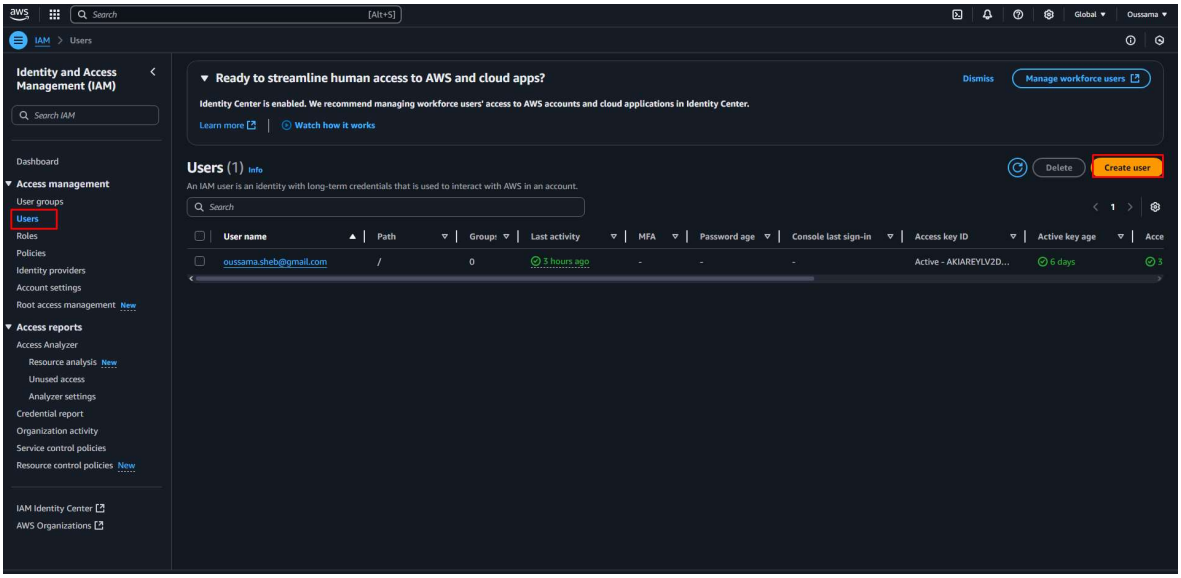
6. Click Current Account
7. Under Data sources, select AWS IoT SiteWise



8. Review your selections and click Create Workspace

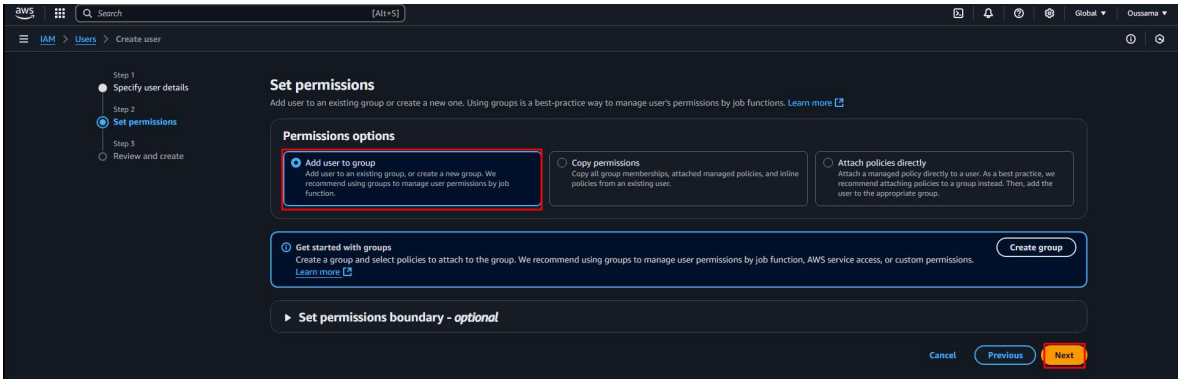


9. Now we have to create a User to be able to access our Grafana workspace
   » Go to IAM -> Users -> Create user

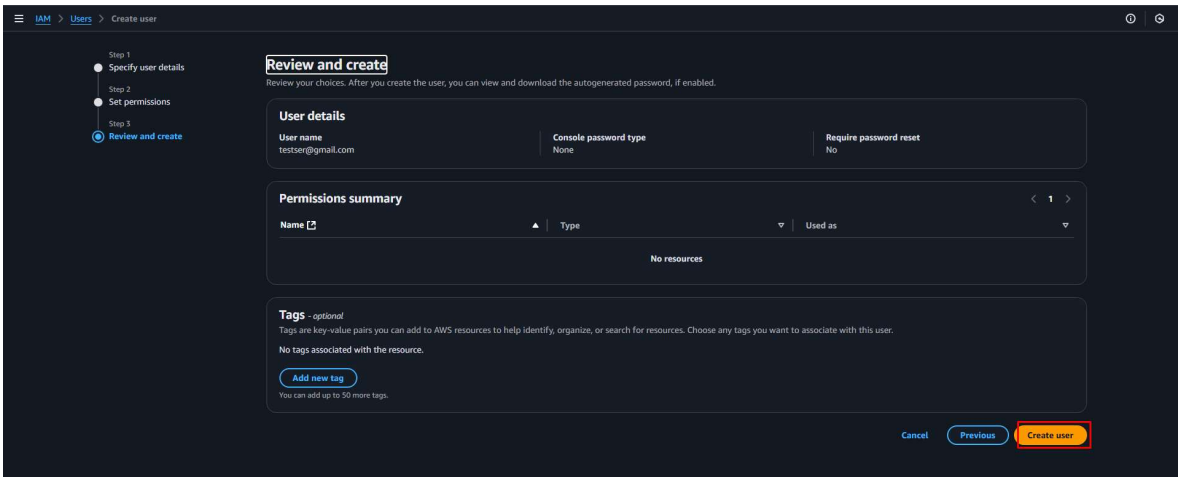


10. Enter a username and click Next

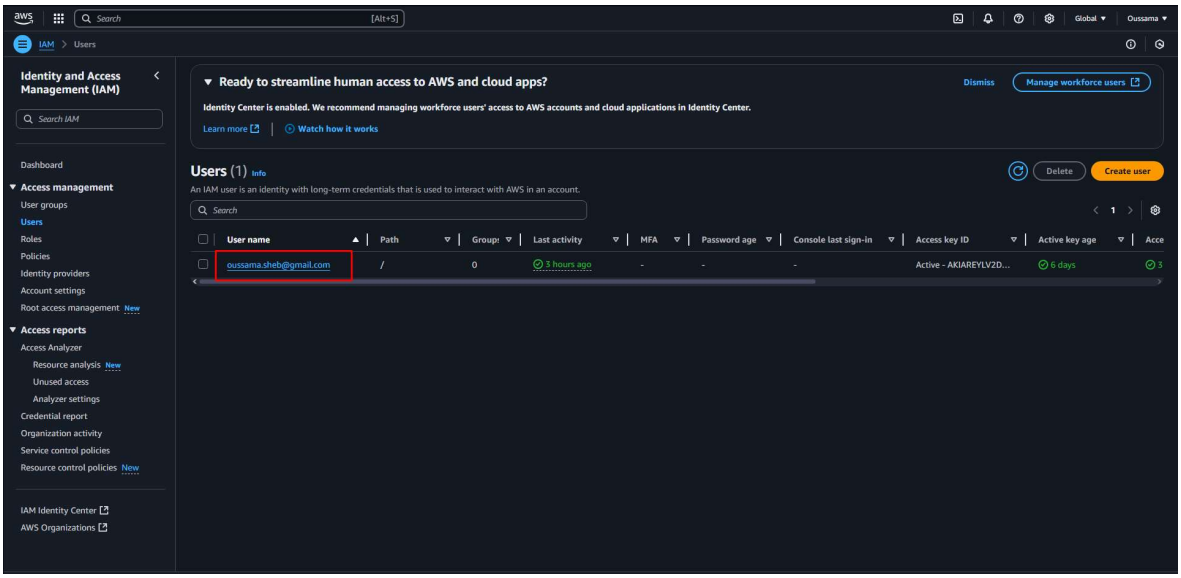
11. Click on Attach policies directly -> **Add user to group**



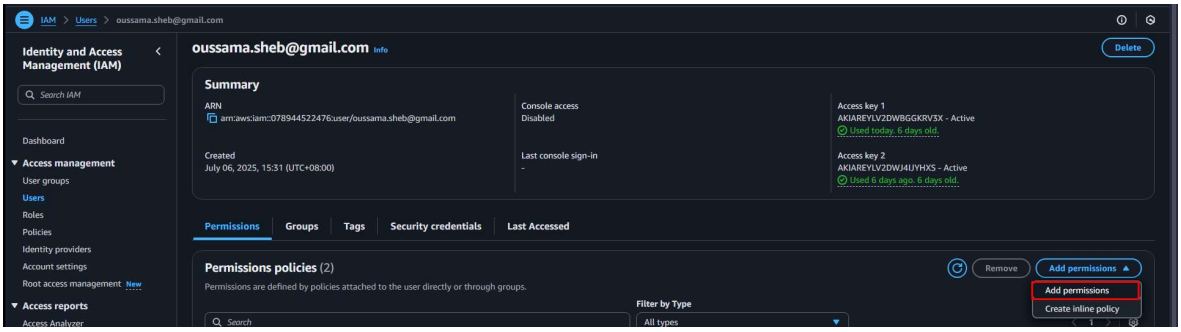
12. Review your details then click **Create user**



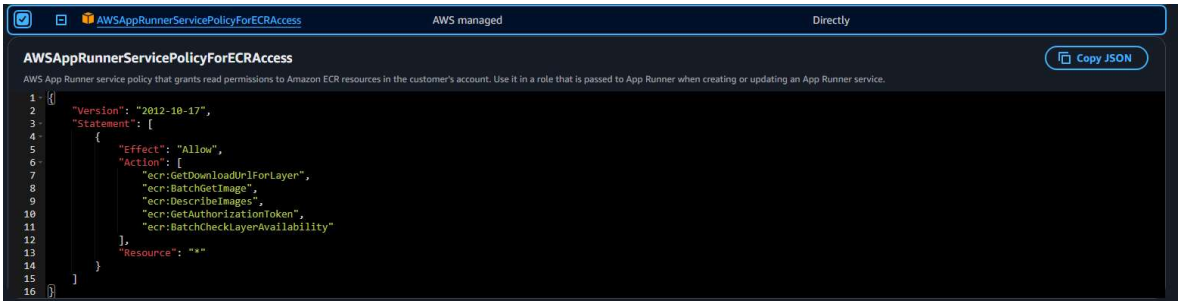
13. Navigate to the user you have created, click on it



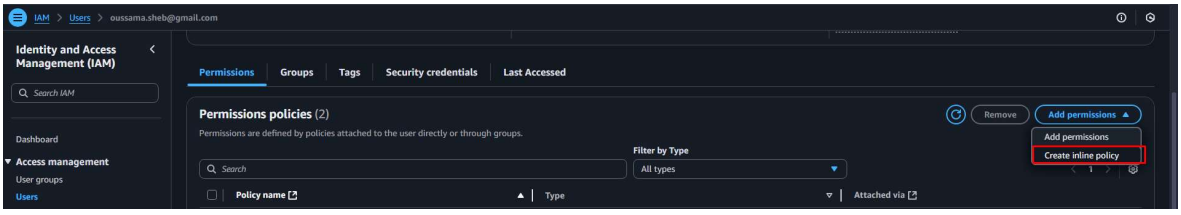
14. Under Permission policies, go to Add permissions

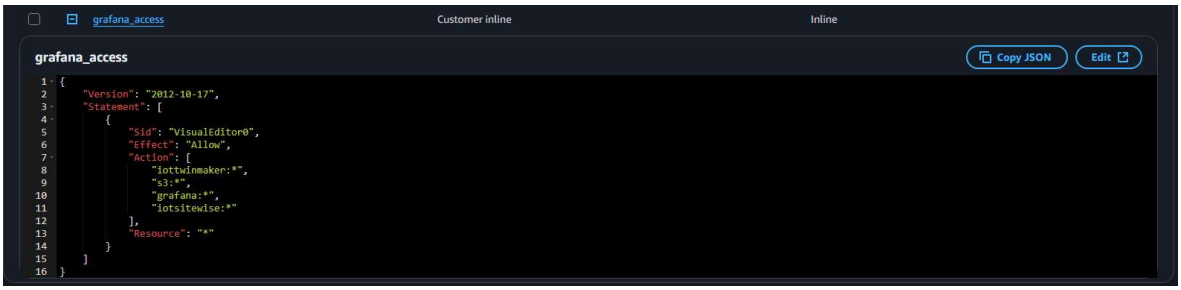


15. Add this ECR permission

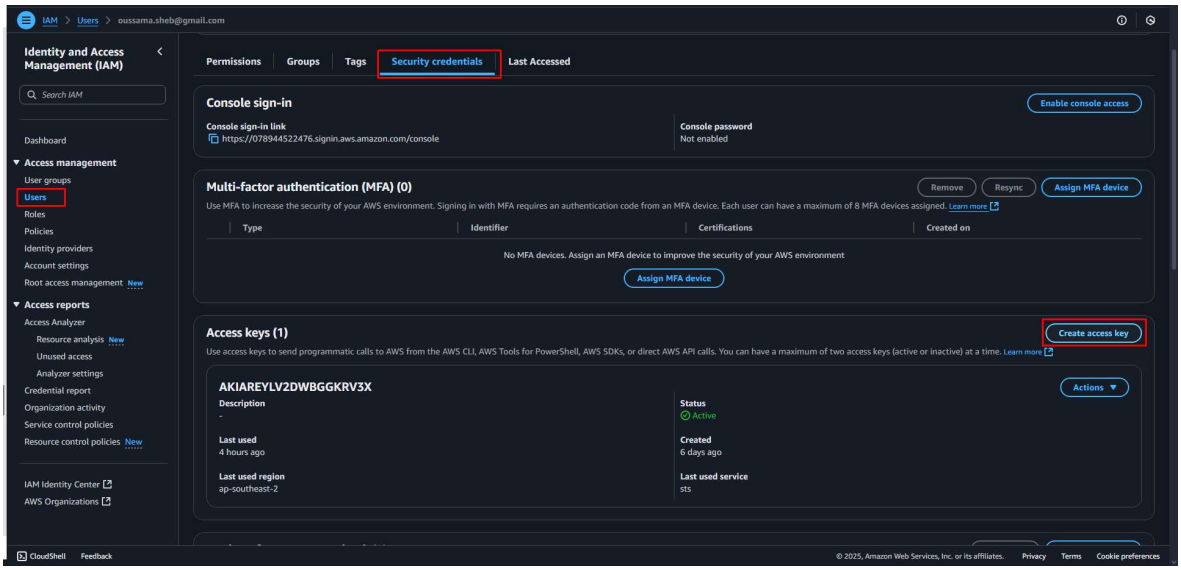


16. After that, create an Inline policy and paste this JSON

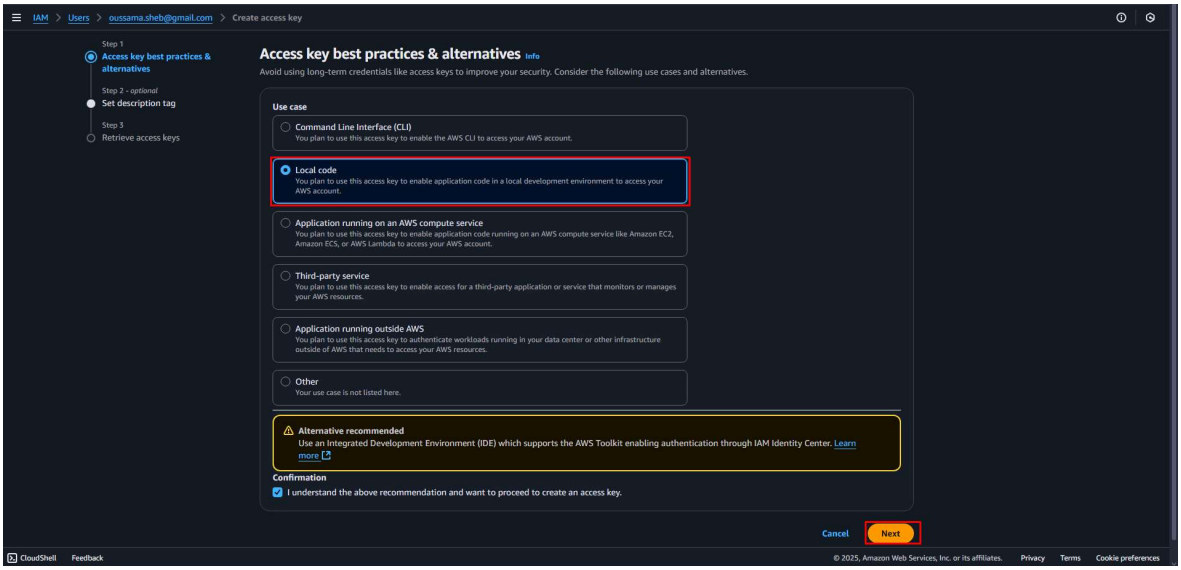


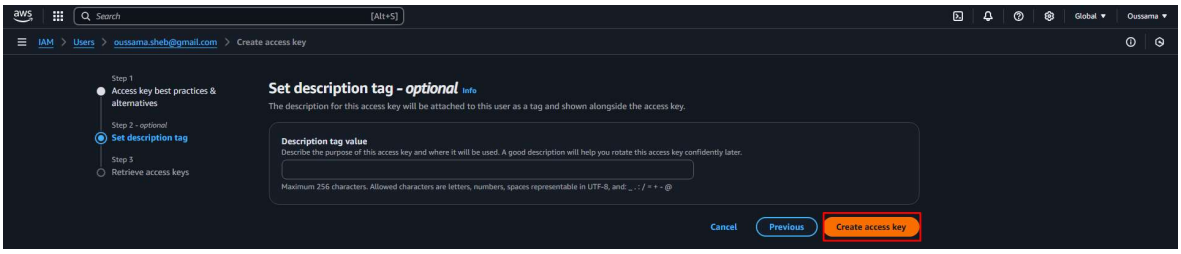


17. Now go to Security credentials -> Create access key

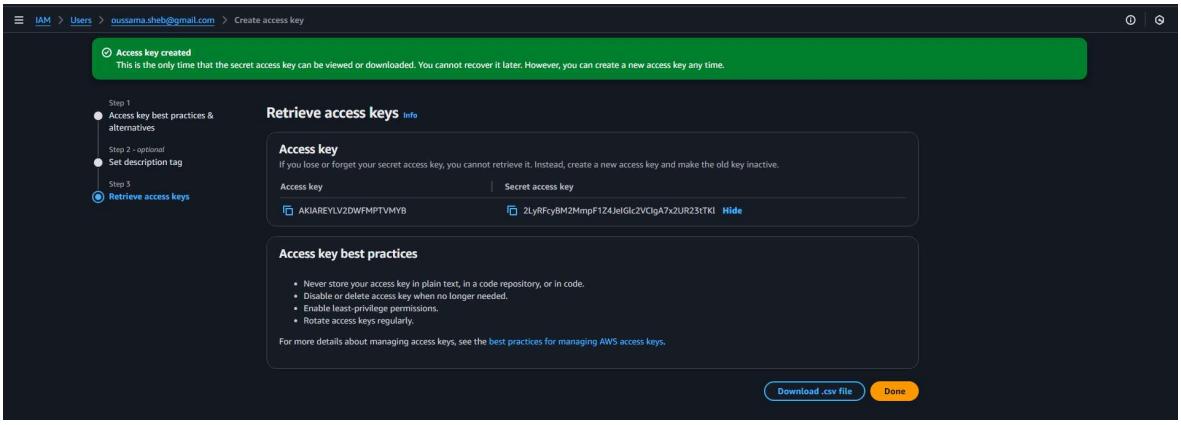


18. Under Use case click Local code, tick the Confirmation, then Next

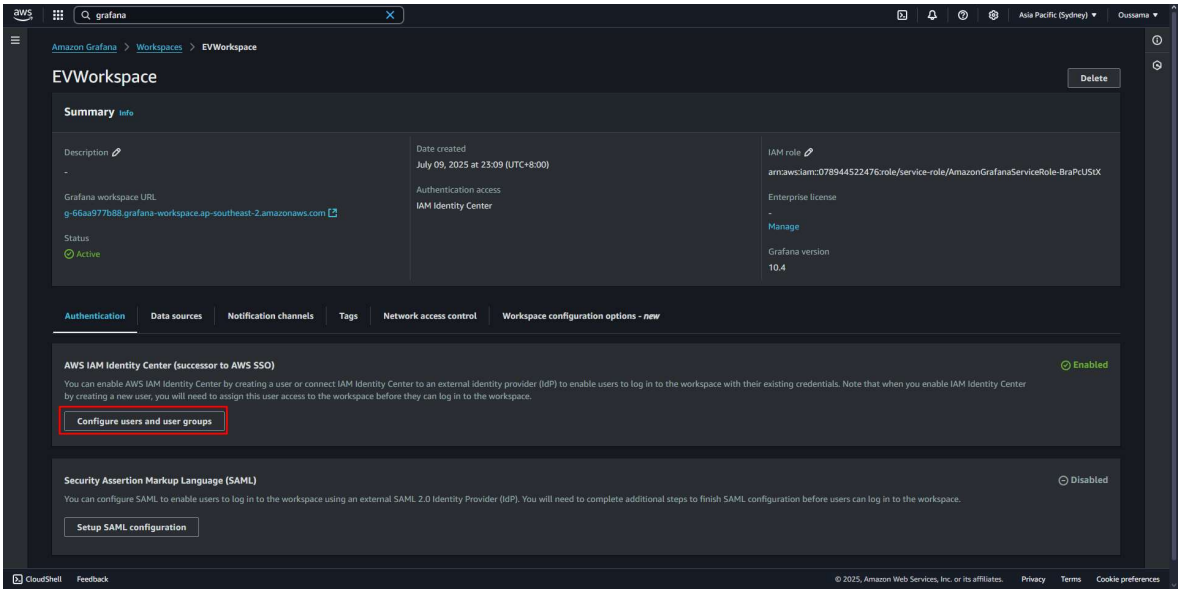




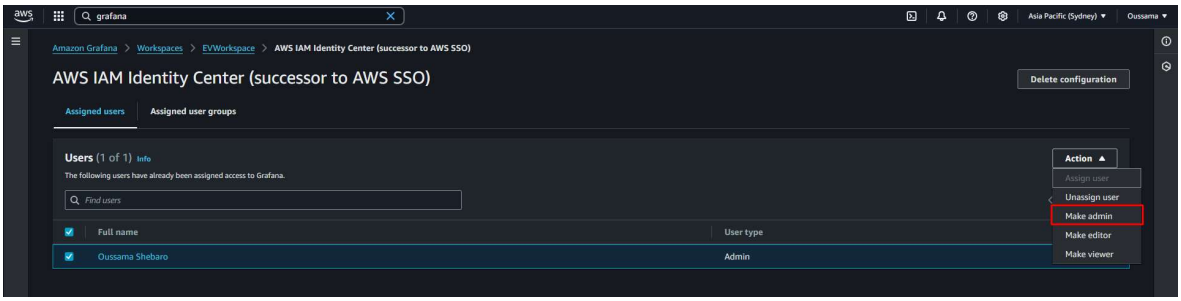
19. Now download and save your Access key and Secret Access key as the secret one cannot be seen again, you would have to delete and create a new one if you lost it



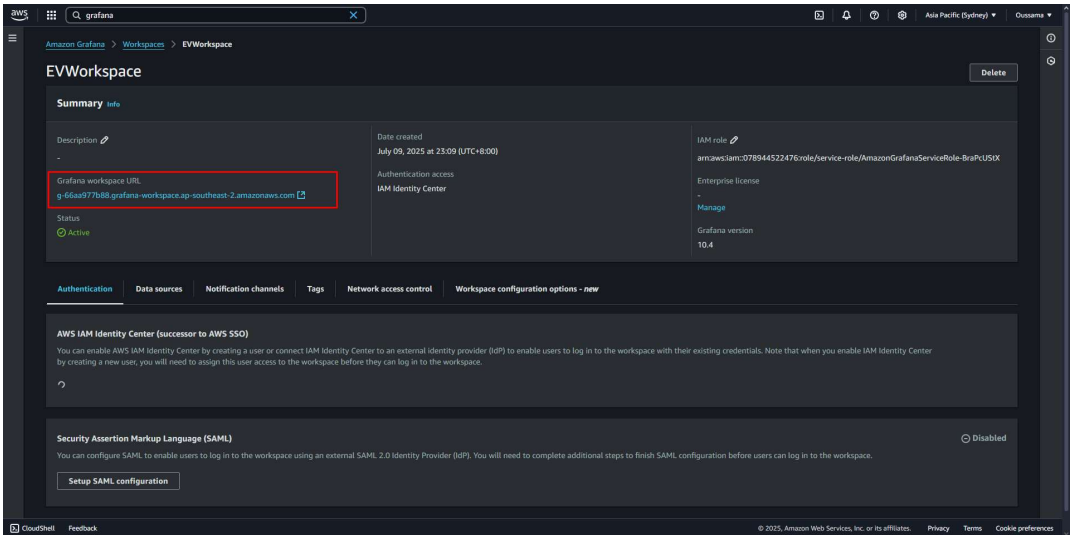
20. Go back to Grafana, click on Configure users and user groups



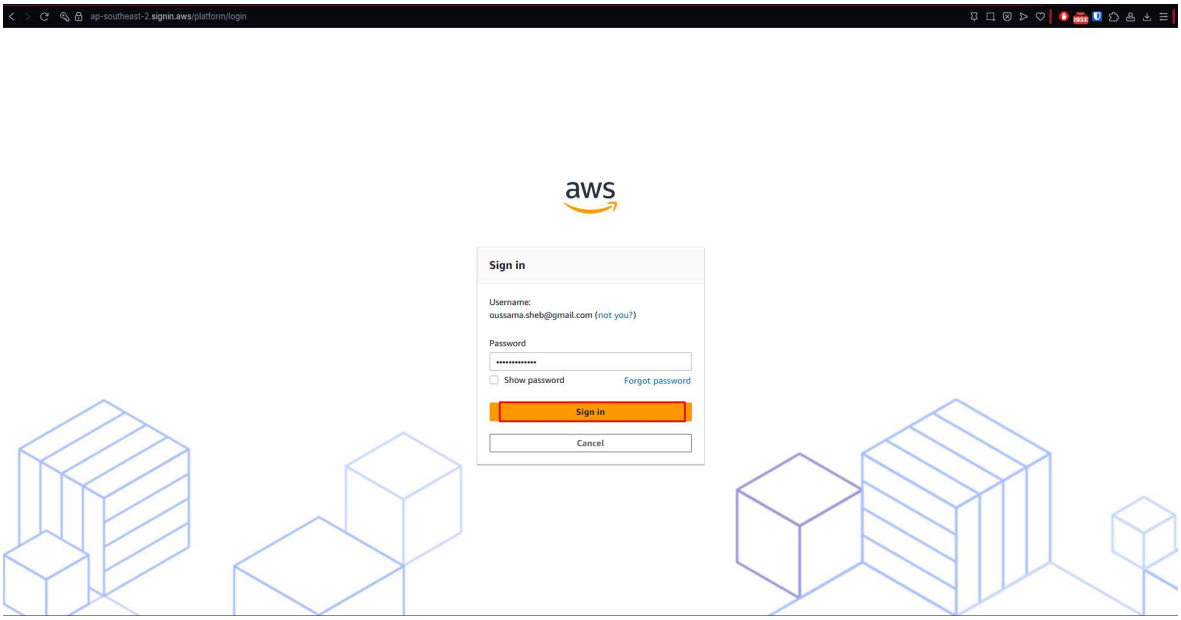
21. Give your user Admin permissions



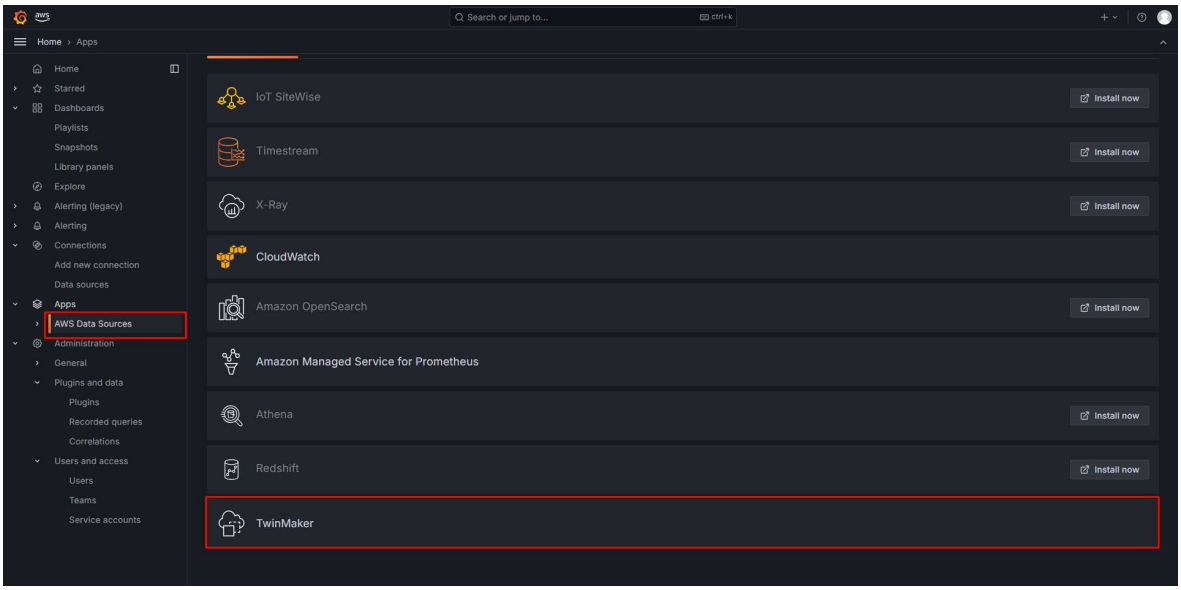
22. Once you have your User ready, go to your workspace



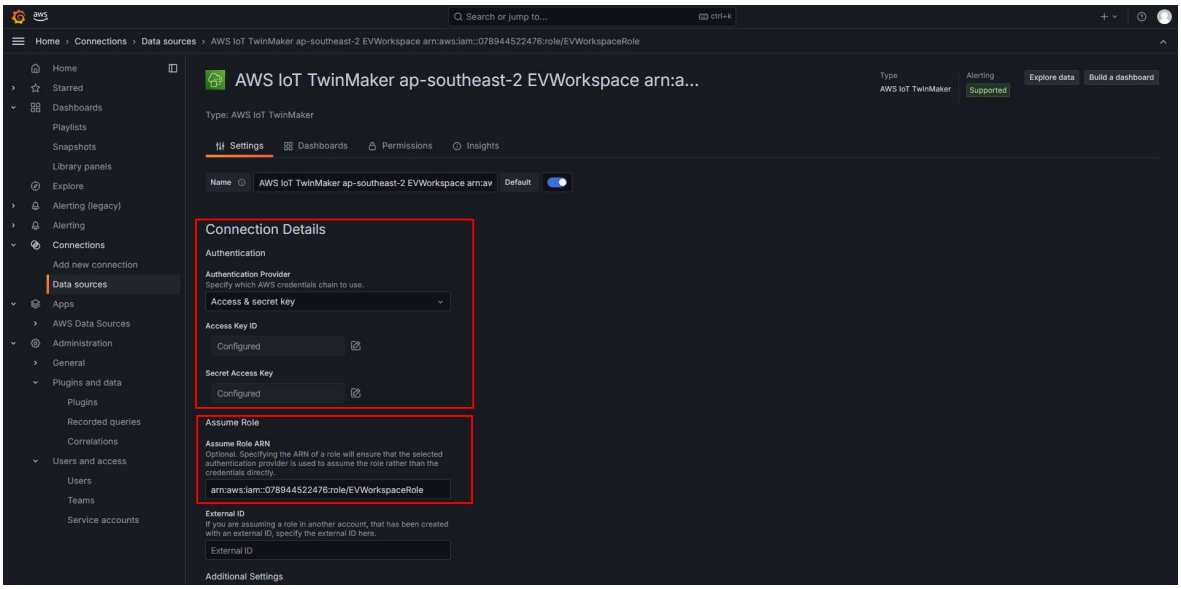
23. Enter the login details we just created



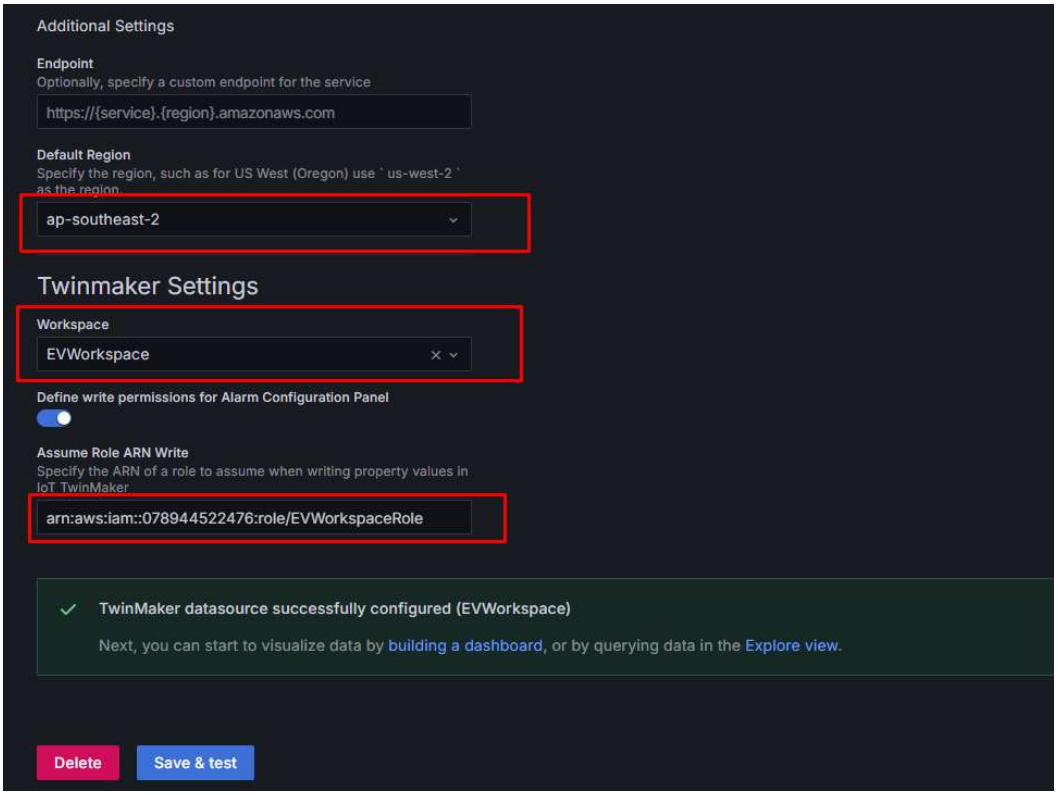
24. Once you are logged in, go to Apps -> AWS Data Sources -> TwinMaker



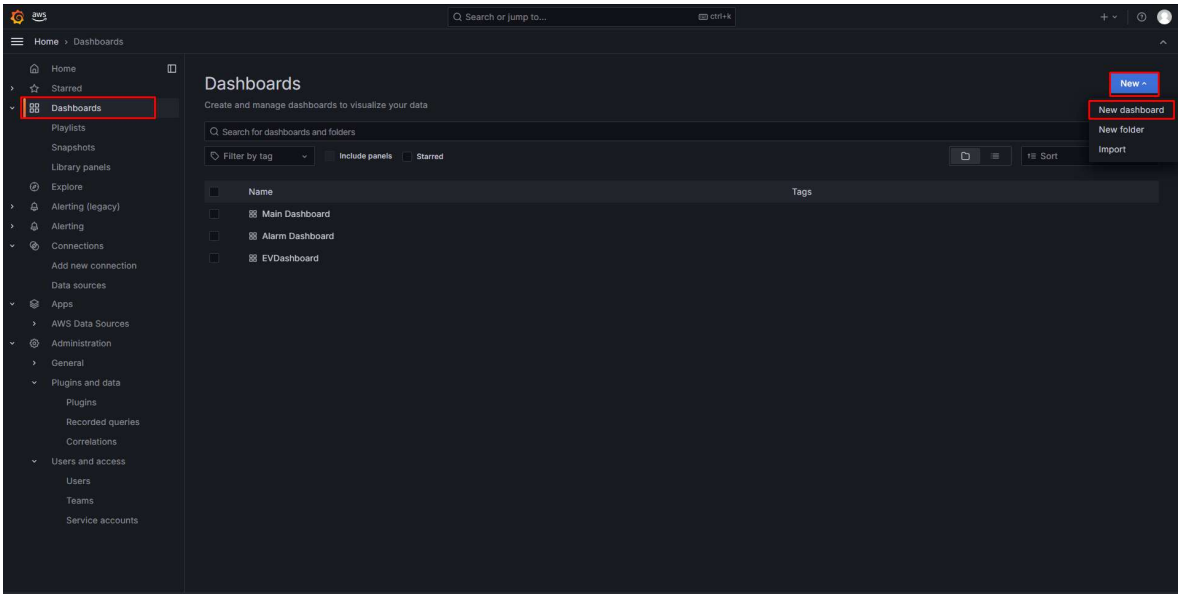
25. For the Authentication Provider, we will use the Access and secret key we saved earlier from the User we just created and used to log in
26. Under Assume Role, enter the ARN of the workspace role we created earlier in TwinMaker



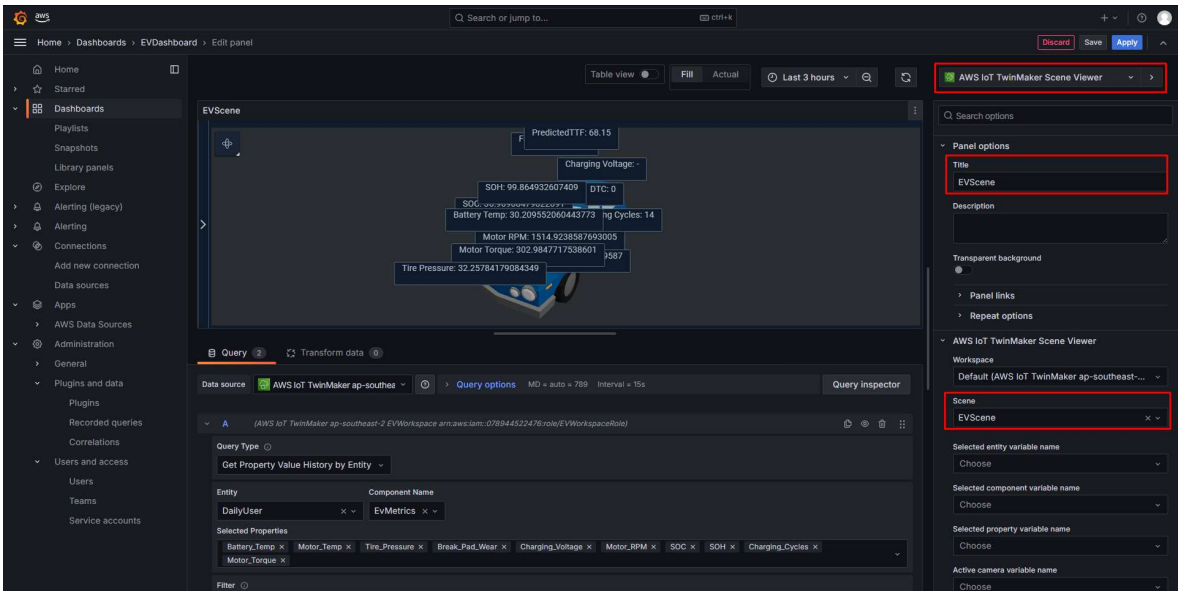
27. Default Region for our case is ap-southeast-2
28. Select your TwinMaker Workspace
29. Turn on Write permissions for Alarm Configuration Panel
30. Input the same workspace ARN we entered earlier
31. Click Save & Test, you should see a message saying it's successful



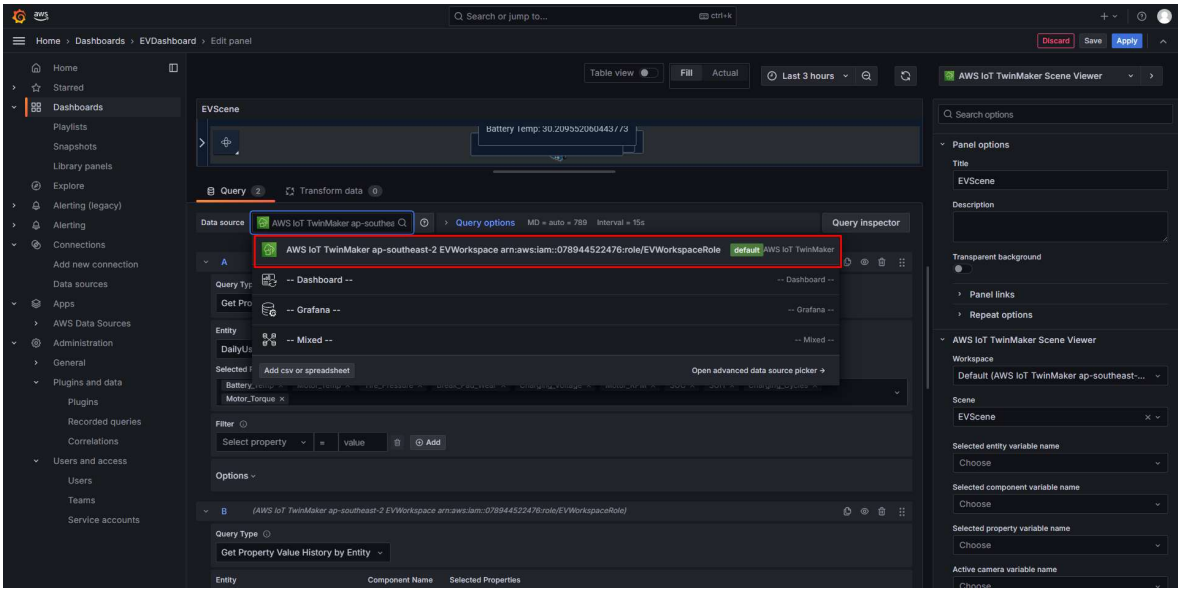
32. Go to Dashboards -> New -> New dashboard



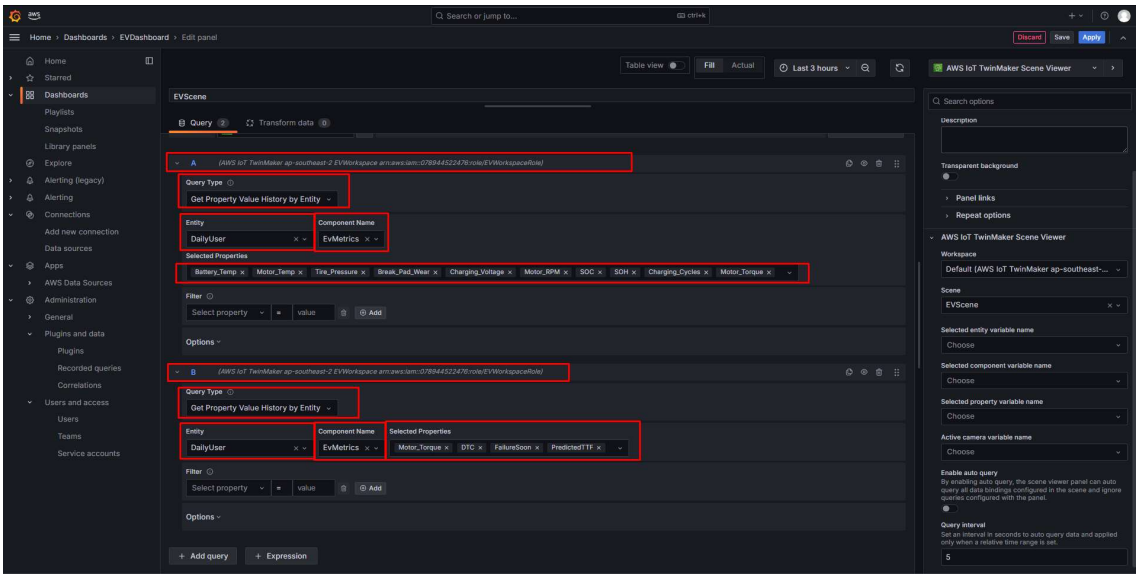
33. Select AWS IoT TwinMaker Scene Viewer
34. Enter a Title, and select the TwinMaker Scene



35. Now select TwinMaker as the datasource

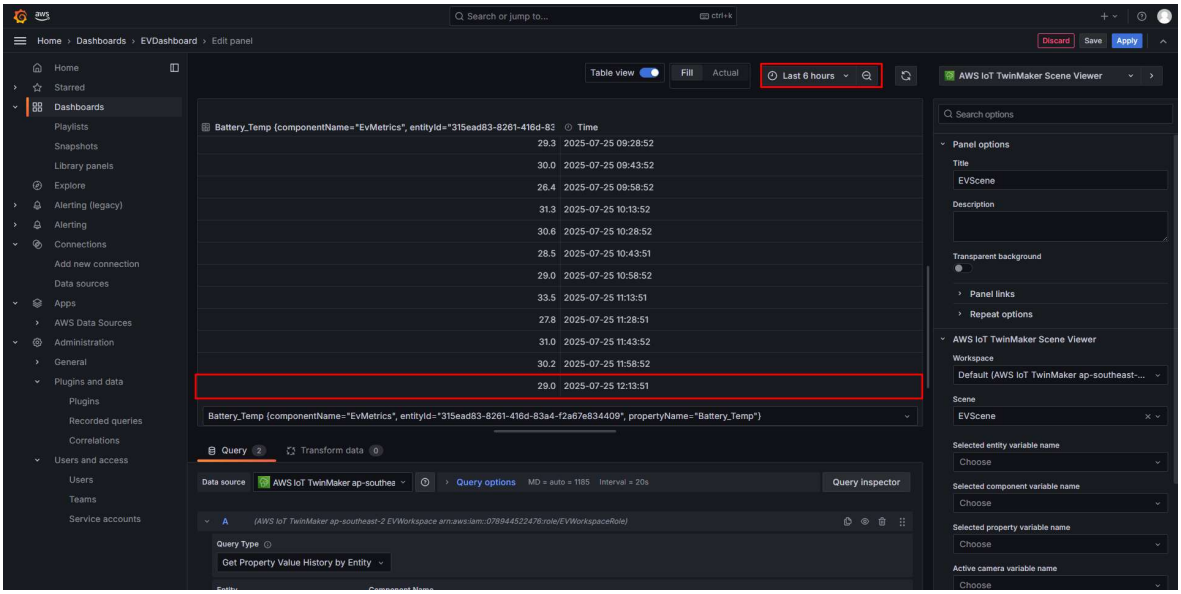


36. Under Query Type, select Get Property Value History By Entity
37. Choose your entity, for our case DailyUser
38. Select your Component Name
39. Now choose your properties, since one query allows a maximum of 10 properties and we want to show more, so we will create two queries and split the properties into both

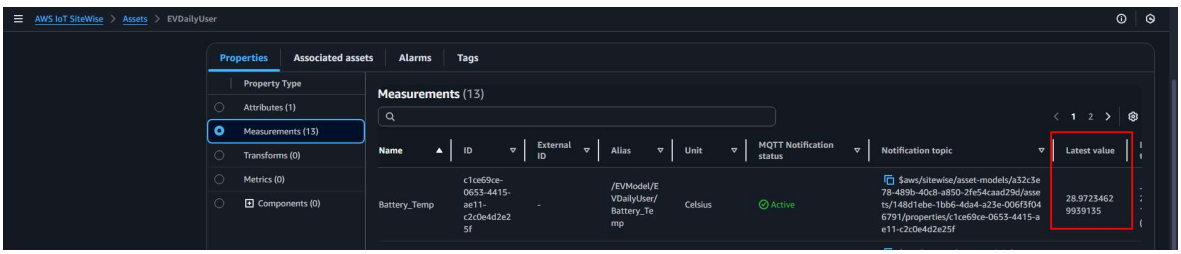


40. Now we can check our time-series data

Let’s see in the last 6 hours for Battery Temp, the trend in table format



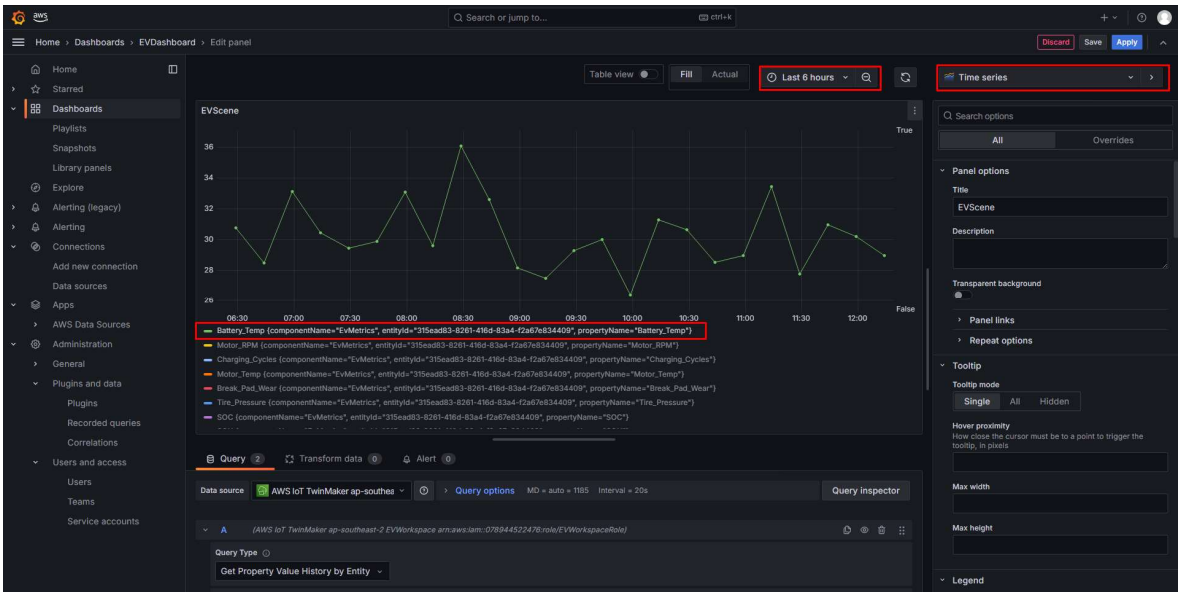
Let’s check the last data sent to IoT SiteWise



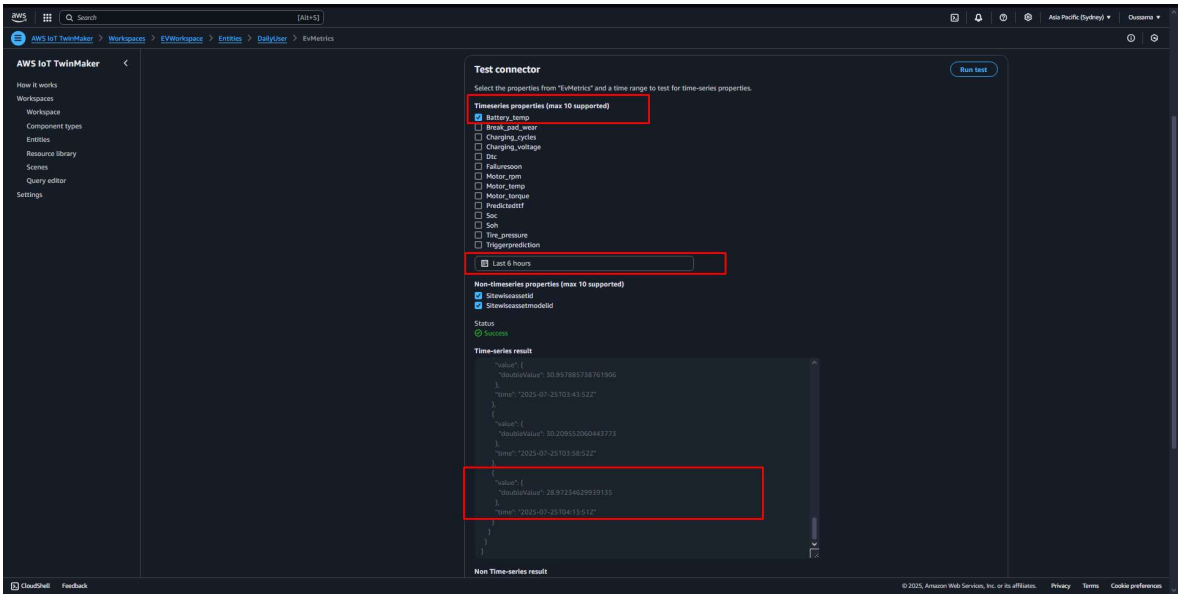
ID	External ID	Alias	Unit	MQTT Notification status	Notification topic	Latest value	Latest value timestamp
c1ce69ce-0653-4415-ae11-c2c0e4d2e25f	-	/EVModel/EVDailyUser/Battery_Temp	Celsius	Active	$aws/sitewise/asset-models/a32c3e78-489b-40c8-a850-2fe54caad29d/assets/148d1ebe-1bb6-4da4-a23e-006f3f046791/properties/c1ce69ce-0653-4415-ae11-c2c0e4d2e25f	28.97234629939135	July 25, 2025 at 12:13:51 (UTC+8:00)

As we can see, to be able to easily read it, Grafana will round up the number in our table.

Let’s look at the Time-series graph for Battery temp and see how it’s changing over time



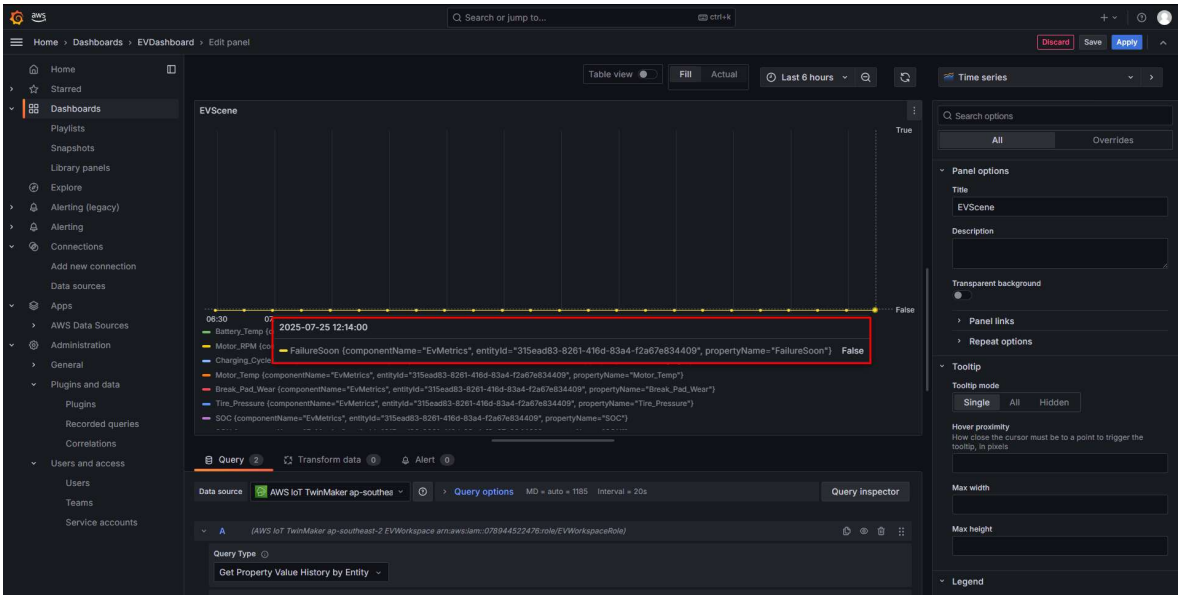
We can also verify it is going through TwinMaker from the Test connector

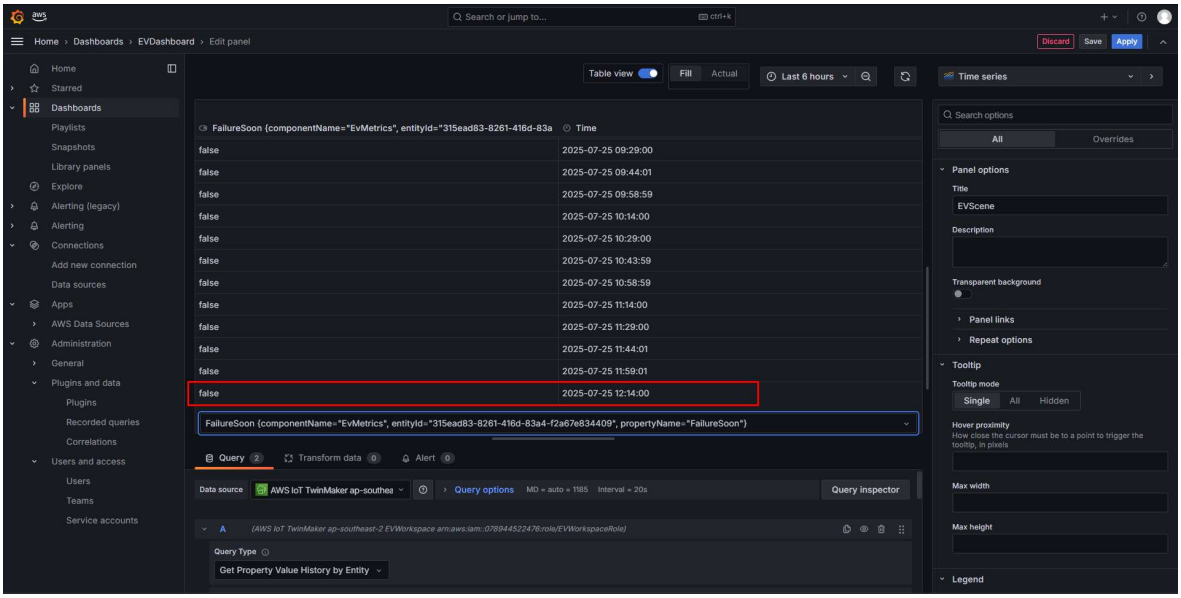


Now let us verify if the predictions are being made as they should…

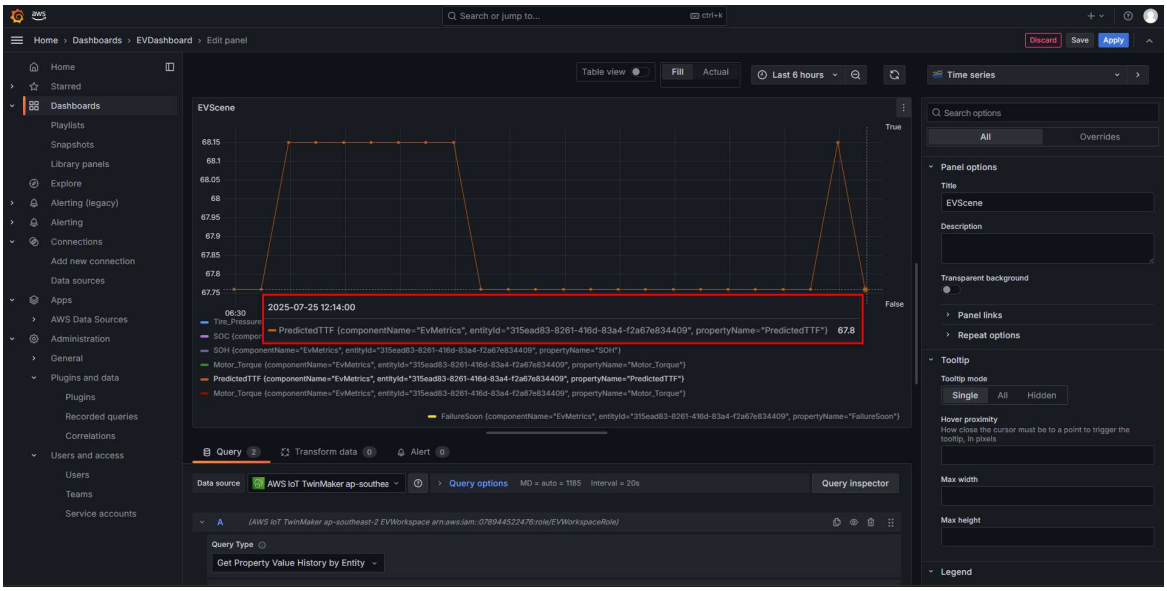
So, as you can see, the last few vehicles are marked as FailureSoon = False

We can also analyze that prediction is made after approximately **9 seconds** from when the sensor data is sent, this aligns with what we have created as Lambda 1 is setting TriggerPrediction to True first, then Lambda 2 will be triggered to do the prediction and set TriggerPrediction to False again. So, it is at approx. 12:14 PM.



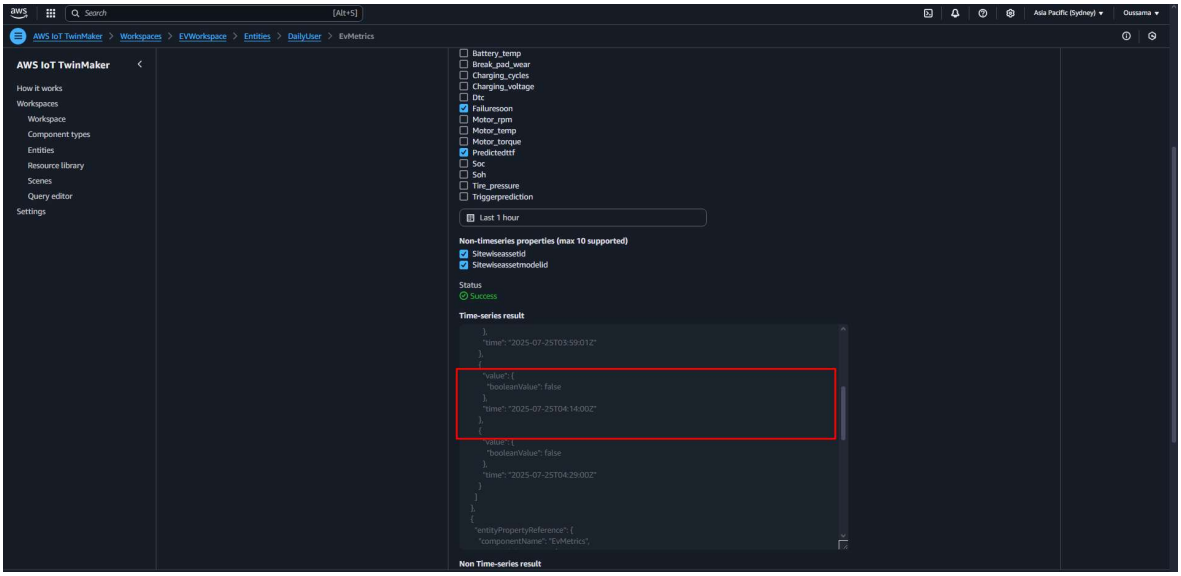


Here is a look at the PredictedTTF graph and table as well at 12:14 PM

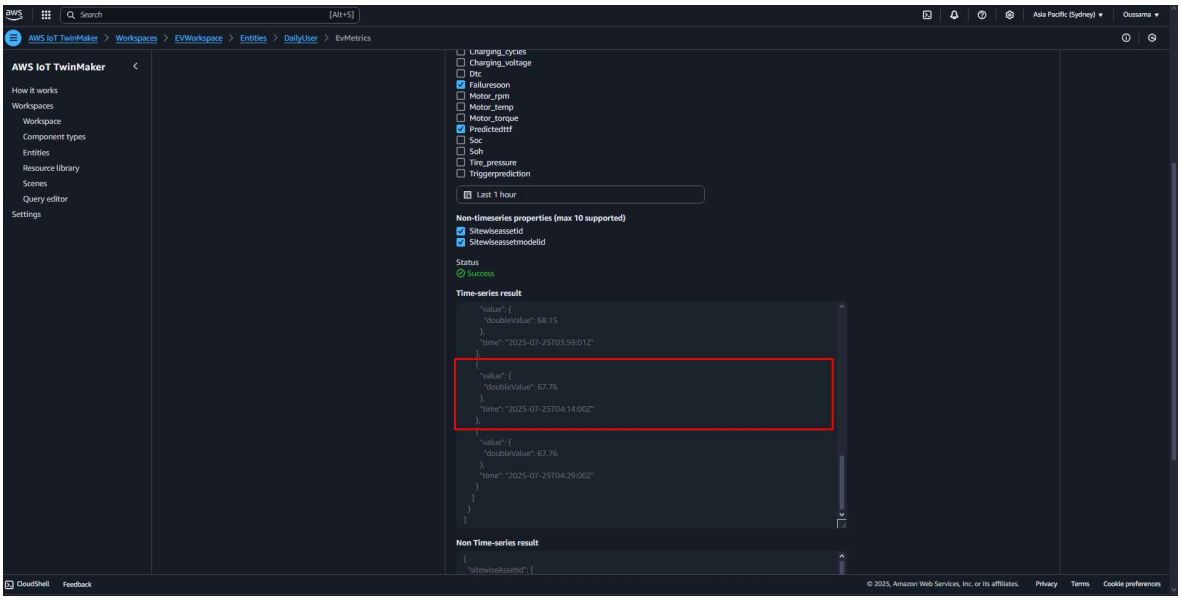


Let’s see them in TwinMaker Test Connector

Below you can see the FailureSoon Time-series result at 12:14 PM



Below you can see the PredictedTTF Time-series result at 12:14



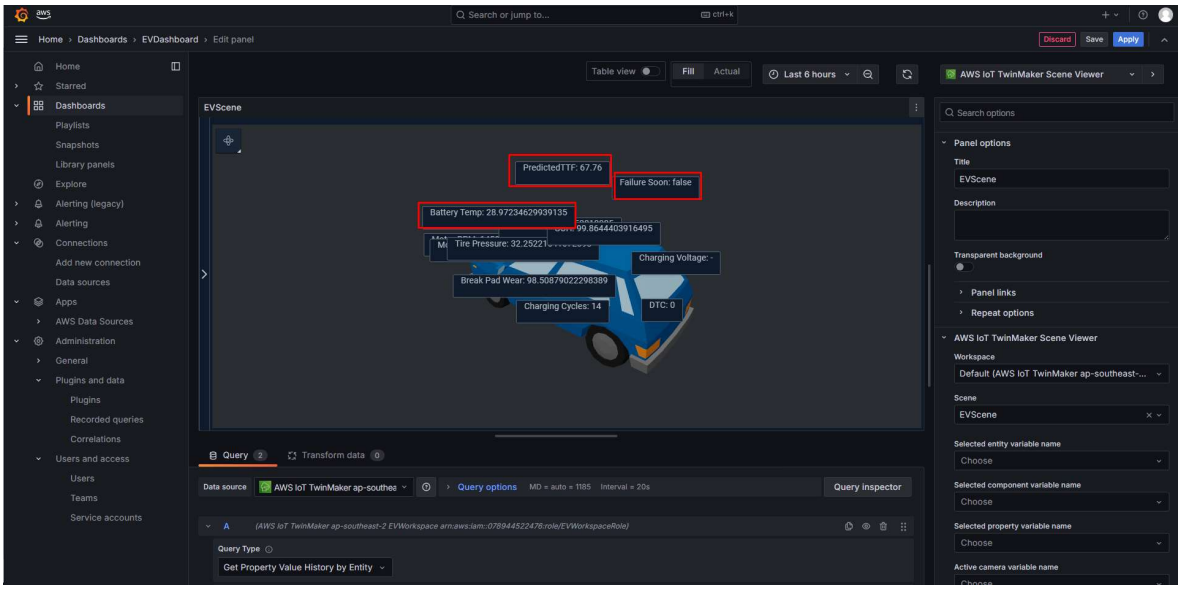
PredictedTTF {componentName="EvMetrics", entityId="315ead83-8261-416d-83	Time
67.8	2025-07-25 06:29:00
67.8	2025-07-25 06:44:00
68.2	2025-07-25 06:58:59
68.2	2025-07-25 07:14:00
68.2	2025-07-25 07:29:01
68.2	2025-07-25 07:44:01
68.2	2025-07-25 07:59:00
68.2	2025-07-25 08:14:00
68.2	2025-07-25 08:29:00
67.8	2025-07-25 08:44:01
67.8	2025-07-25 08:59:00
67.8	2025-07-25 09:14:00

PredictedTTF {componentName="EvMetrics", entityId="315ead83-8261-416d-83a4-f2a67e834409", propertyName="PredictedTTF"}

PredictedTTF {componentName="EvMetrics", entityId="315ead83-8261-416d-83	Time
67.8	2025-07-25 09:29:00
67.8	2025-07-25 09:44:01
67.8	2025-07-25 09:58:59
67.8	2025-07-25 10:14:00
67.8	2025-07-25 10:29:00
67.8	2025-07-25 10:43:59
67.8	2025-07-25 10:58:59
67.8	2025-07-25 11:14:00
67.8	2025-07-25 11:29:00
67.8	2025-07-25 11:44:01
68.2	2025-07-25 11:59:01
67.8	2025-07-25 12:14:00

PredictedTTF {componentName="EvMetrics", entityId="315ead83-8261-416d-83a4-f2a67e834409", propertyName="PredictedTTF"}

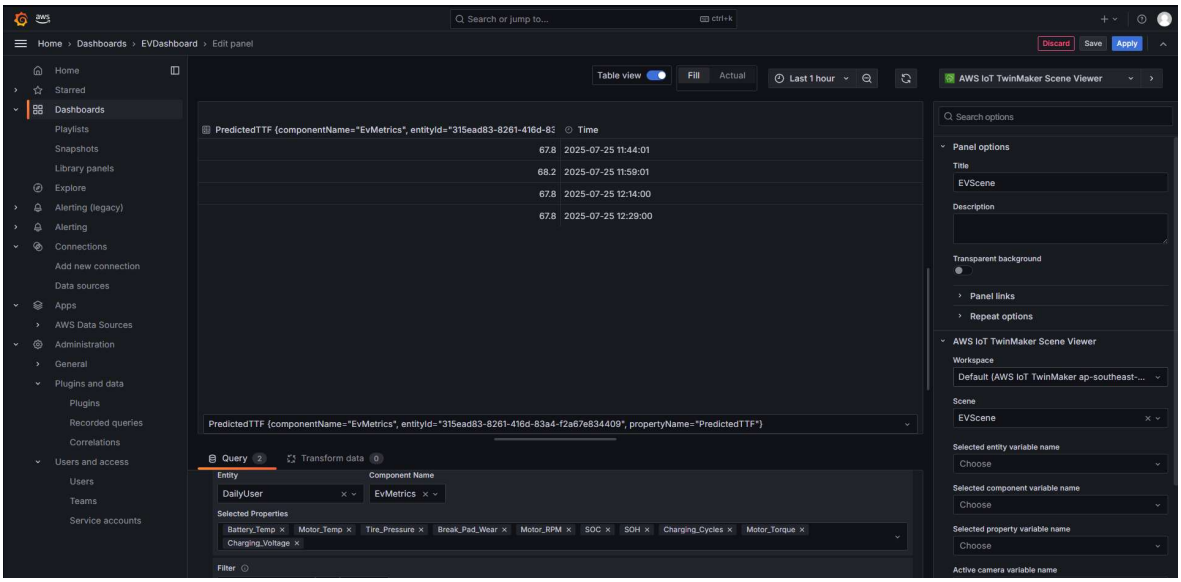
We can also visualize these values in the Scene Viewer



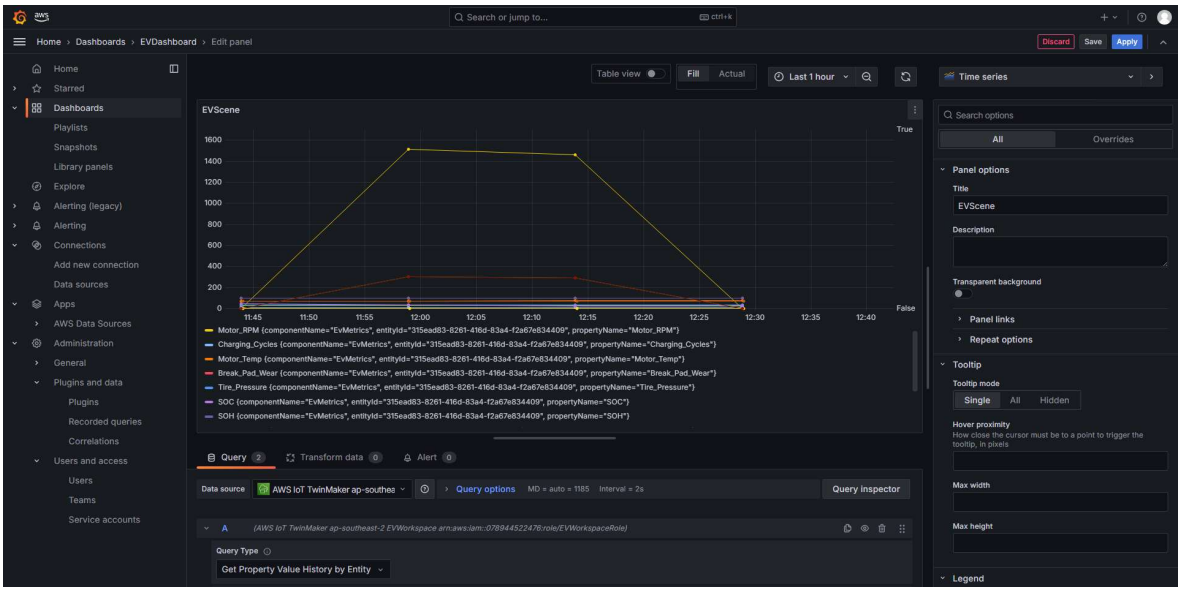
After 15 minutes, it’s 12:29 PM, we have new set of sensor data and new predictions as you can see below.

FailureSoon	f95f4634-6a64-4a01-ad7e-be37bb8e746a	-	/EVModel/EVDailyUser/FailureSoon	-	Active	$aws/sitewise/asset-models/a32c3e78-489b-40c8-a850-2fe54caad29d/assets/148d1ebe-1bb6-4da4-a23e-006f3f046791/properties/f95f4634-6a64-4a01-ad7e-be37bb8e746a	false	July 25, 2025 at 12:29:00 (UTC+8:00)
Motor_RPM	378c529e-9151-4aaa-bd57-7276b6d2f6c4	-	/EVModel/EVDailyUser/Motor_RPM	RPM	Active	$aws/sitewise/asset-models/a32c3e78-489b-40c8-a850-2fe54caad29d/assets/148d1ebe-1bb6-4da4-a23e-006f3f046791/properties/378c529e-9151-4aaa-bd57-7276b6d2f6c4	0	July 25, 2025 at 12:28:52 (UTC+8:00)
Motor_Temp	e9890ad9-6ed4-46dd-a020-a4280311dd4e	-	/EVModel/EVDailyUser/Motor_Temp	Celsius	Active	$aws/sitewise/asset-models/a32c3e78-489b-40c8-a850-2fe54caad29d/assets/148d1ebe-1bb6-4da4-a23e-006f3f046791/properties/e9890ad9-6ed4-46dd-a020-a4280311dd4e	76.14636826944593	July 25, 2025 at 12:28:52 (UTC+8:00)
Motor_Torque	364c56e6-d3e7-4786-bbb5-66cc468c8138	-	/EVModel/EVDailyUser/Motor_Torque	Nm	Active	$aws/sitewise/asset-models/a32c3e78-489b-40c8-a850-2fe54caad29d/assets/148d1ebe-1bb6-4da4-a23e-006f3f046791/properties/364c56e6-d3e7-4786-bbb5-66cc468c8138	0	July 25, 2025 at 12:28:52 (UTC+8:00)
PredictedTTF	a6a6497f-6438-481c-ae68-36d680fe8be4	-	/EVModel/EVDailyUser/PredictedTTF	-	Active	$aws/sitewise/asset-models/a32c3e78-489b-40c8-a850-2fe54caad29d/assets/148d1ebe-1bb6-4da4-a23e-006f3f046791/properties/a6a6497f-6438-481c-ae68-36d680fe8be4	67.76	July 25, 2025 at 12:29:00 (UTC+8:00)
SOC	f13b2a63-c016-4dbf-87c9-98fc0de7fdd1	-	/EVModel/EVDailyUser/SOC	%	Active	$aws/sitewise/asset-models/a32c3e78-489b-40c8-a850-2fe54caad29d/assets/148d1ebe-1bb6-4da4-a23e-006f3f046791/properties/f13b2a63-c016-4dbf-87c9-98fc0de7fdd1	25.544674953818085	July 25, 2025 at 12:28:52 (UTC+8:00)

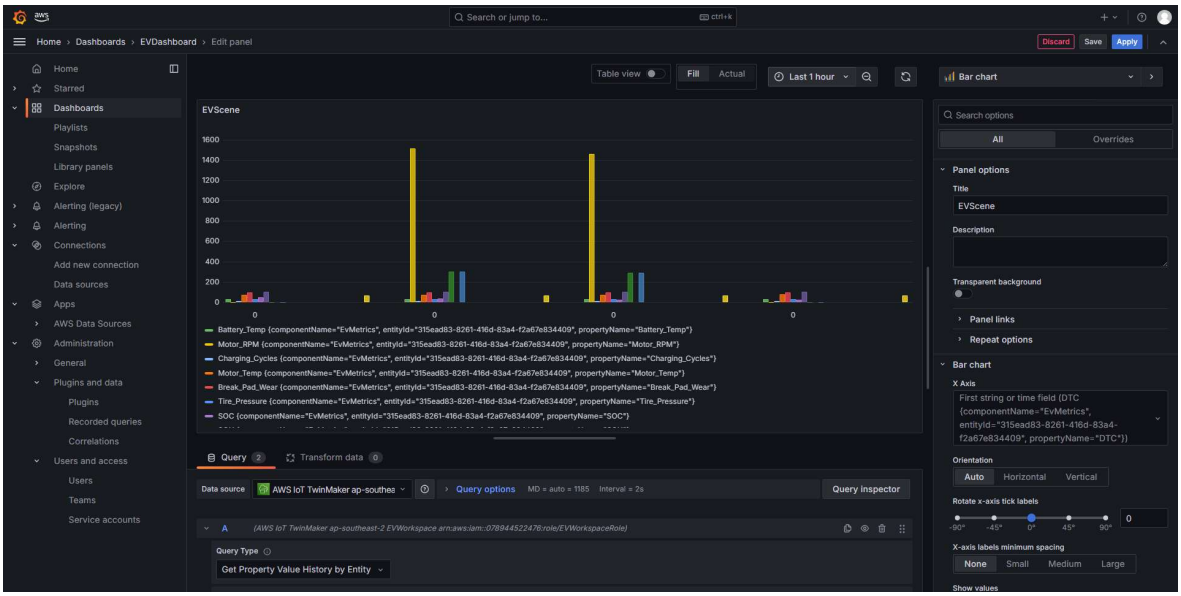
These changes are also being reflected in Grafana as we can visualize



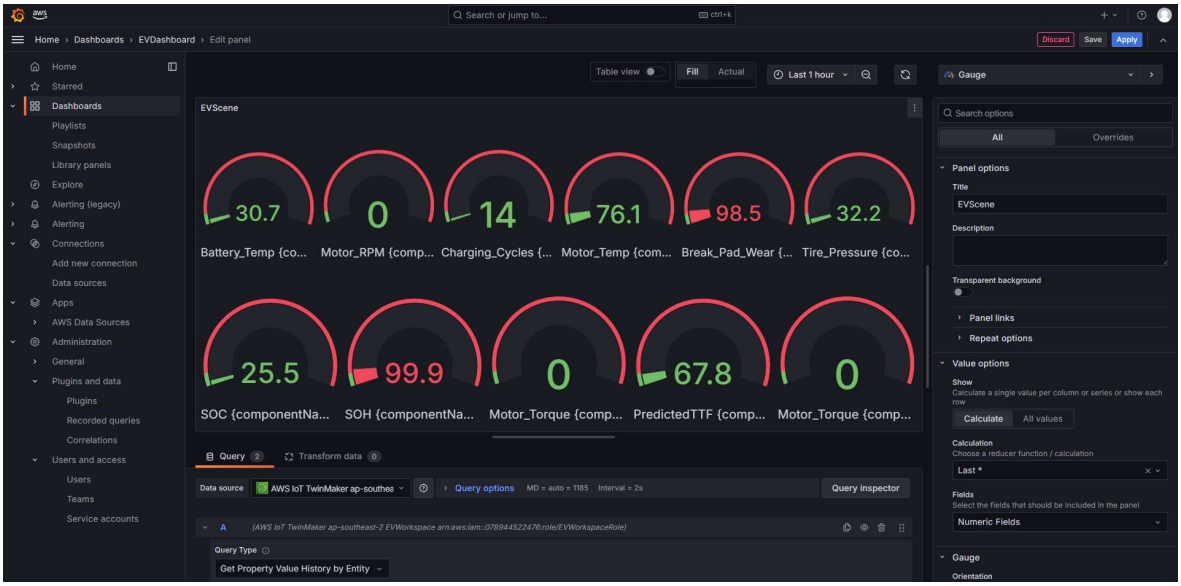
This is a time-series graph that helps you visualize all of our properties together



This is a bar chart to help visualize further



Grafana even allows you to visualize it as gauges!



Here is a link to a YouTube video that helps you visualize our data connectivity better:

https://youtu.be/QH6AbAbXqbg

Skip to 8:40 if you want to see the data changing real-time during the recording