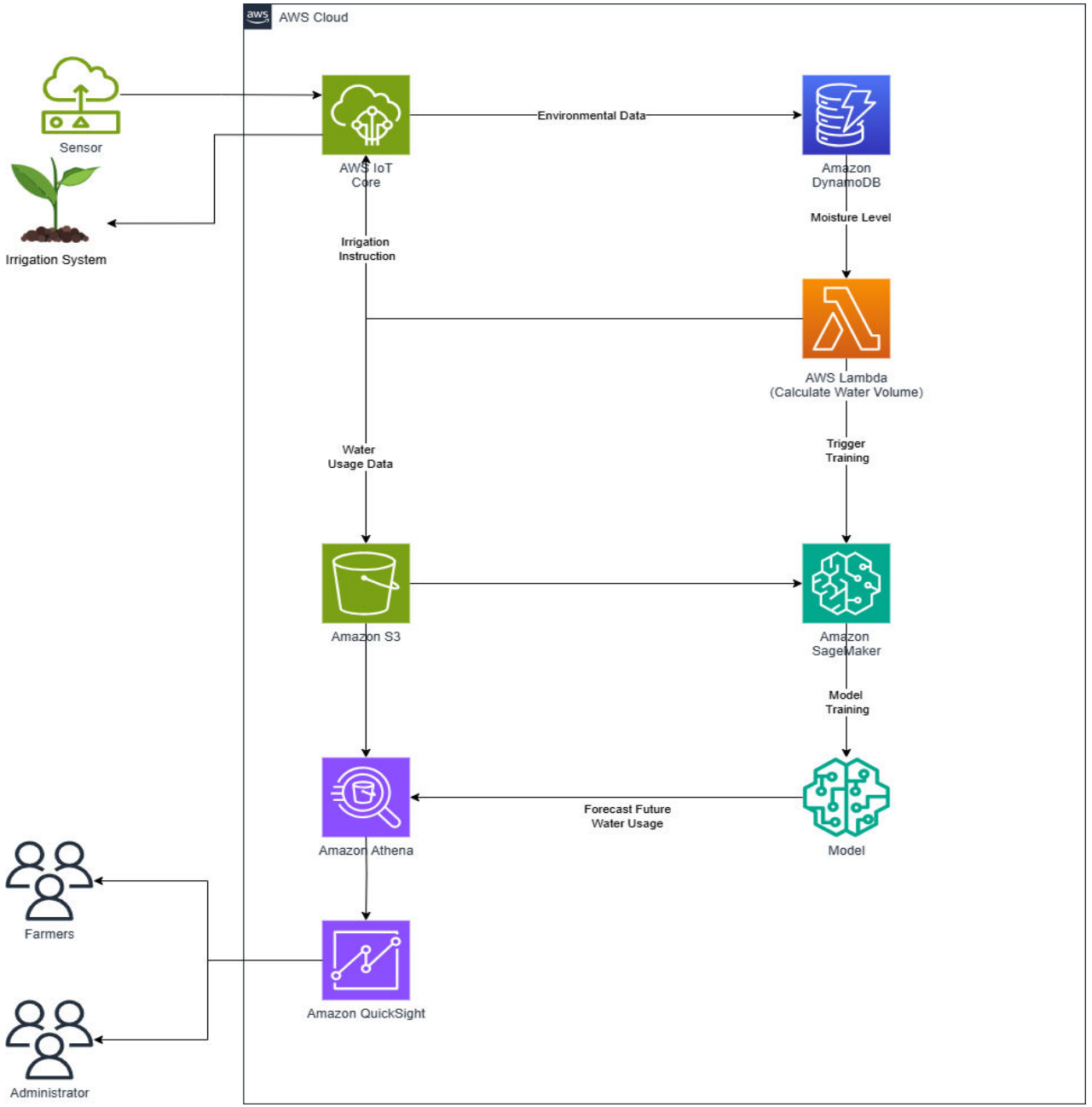
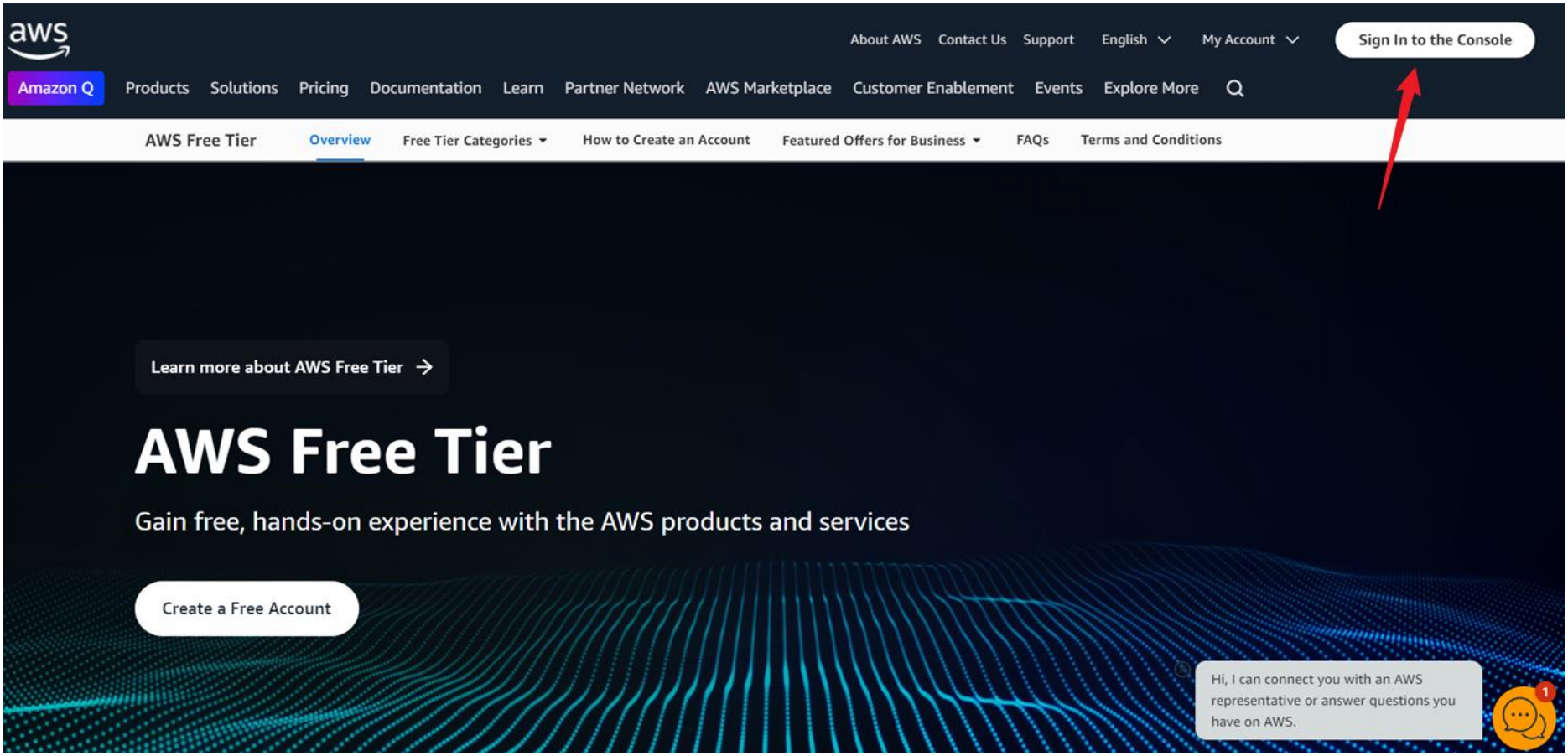
Architecture Diagram

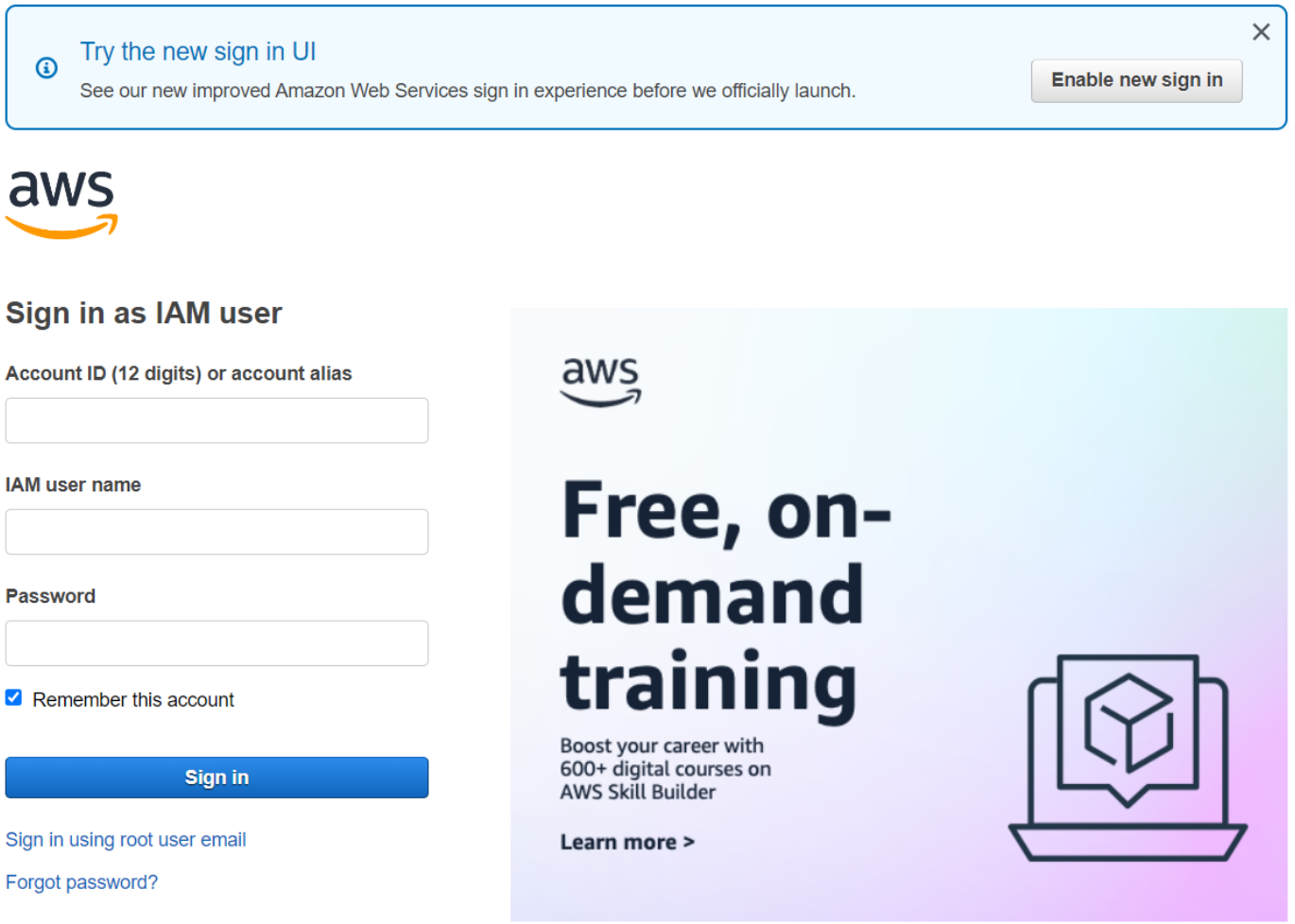


AWS Management Console

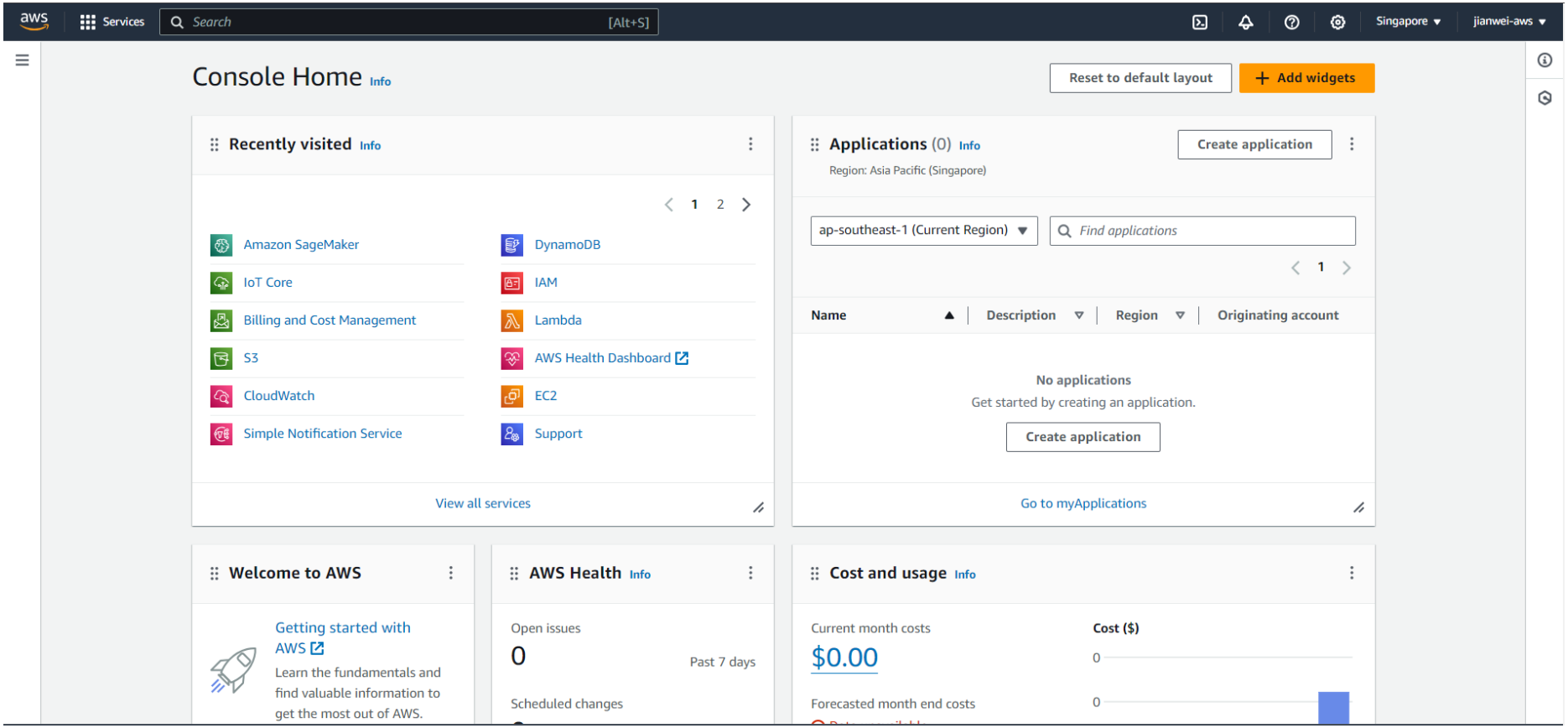
Sign in to the AWS Console



Sign in as IAM user.

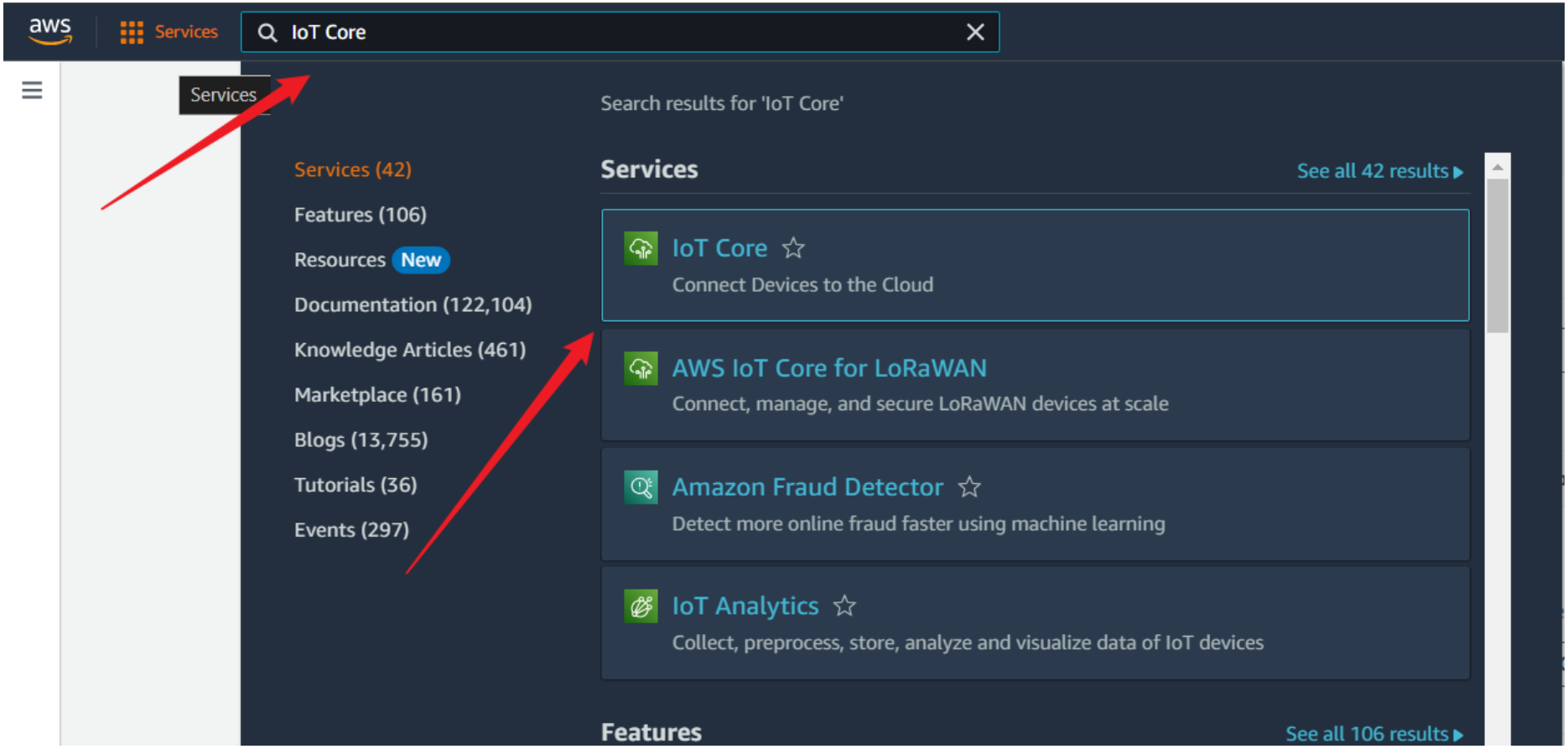


AWS Management Console

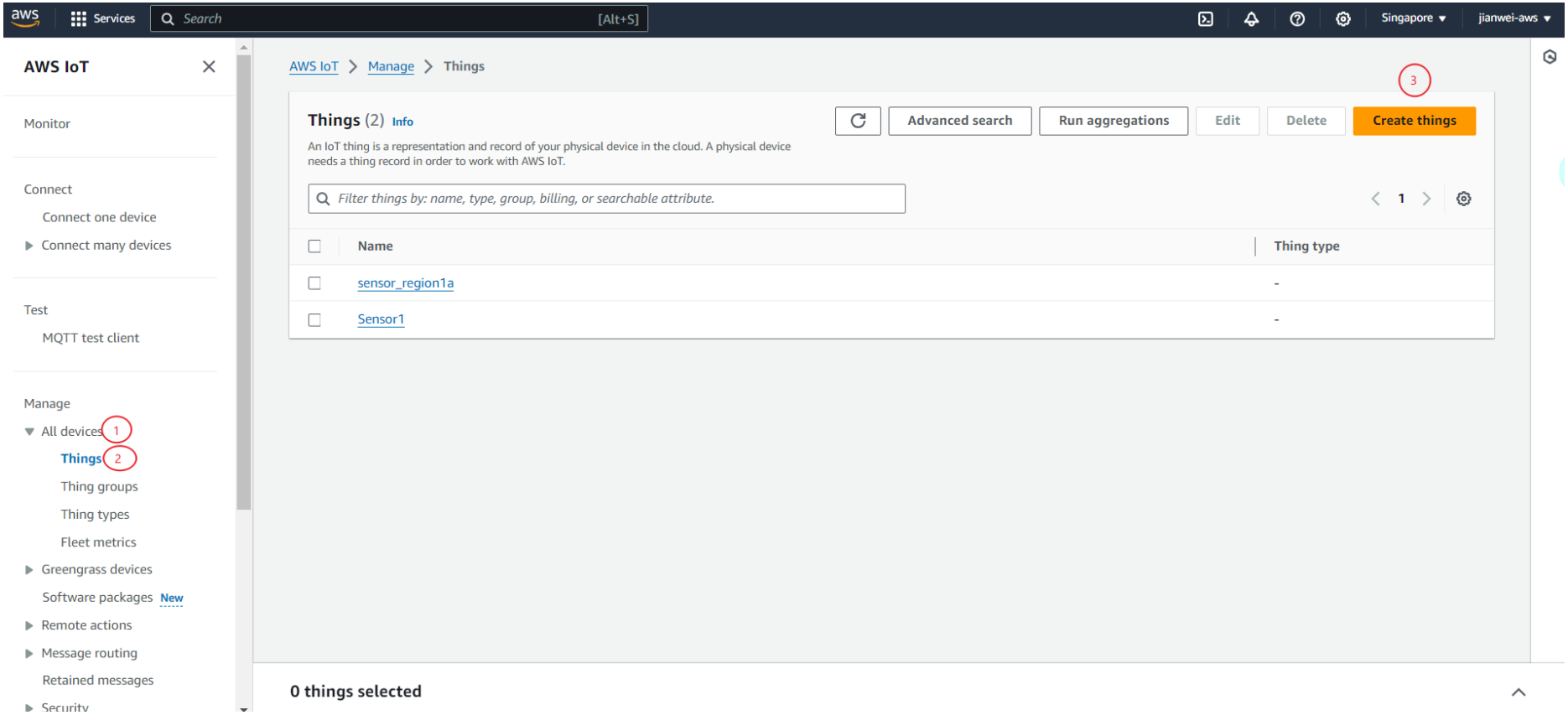


AWS IoT Core

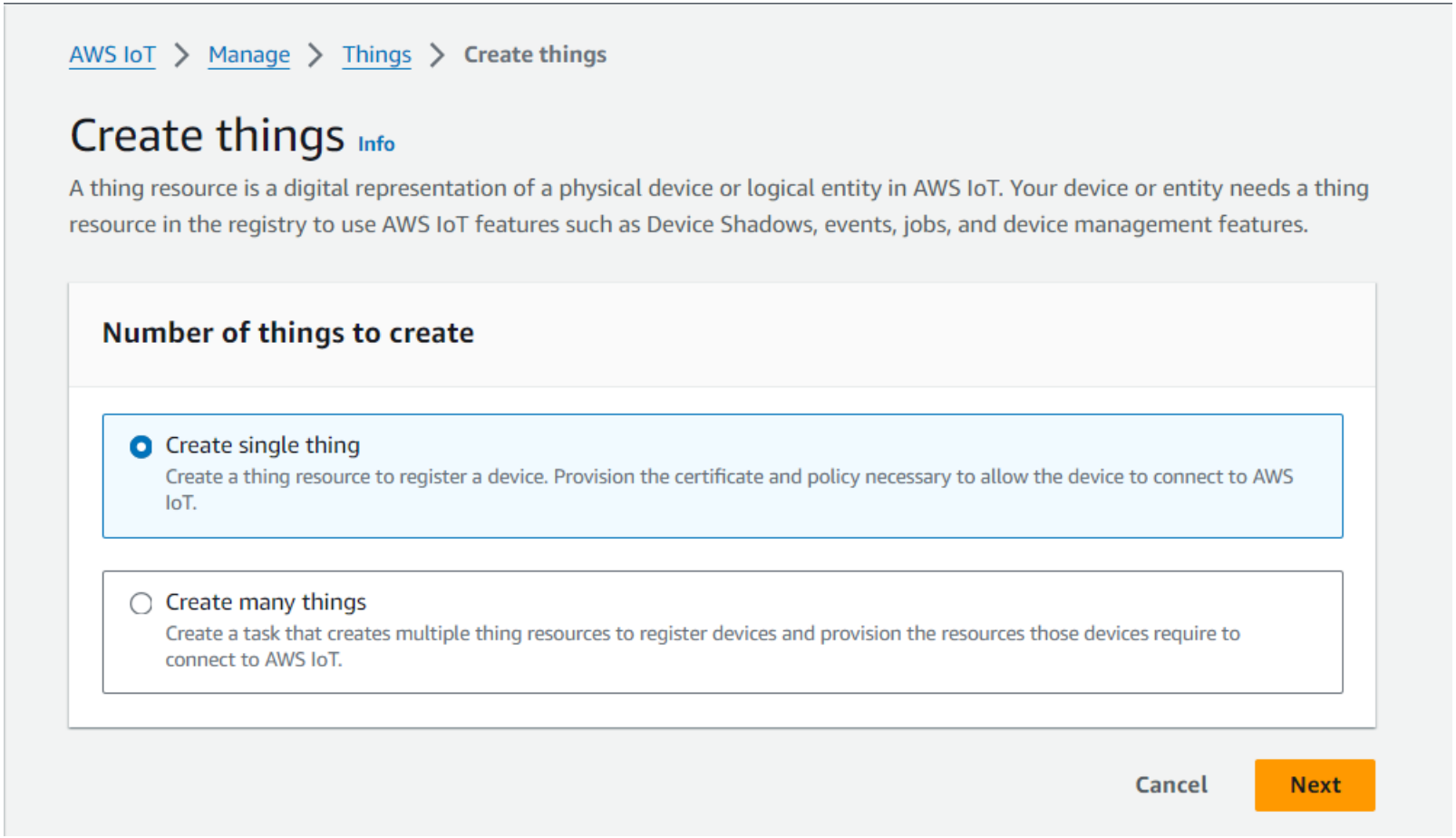
Search IoT Core service from the search bar



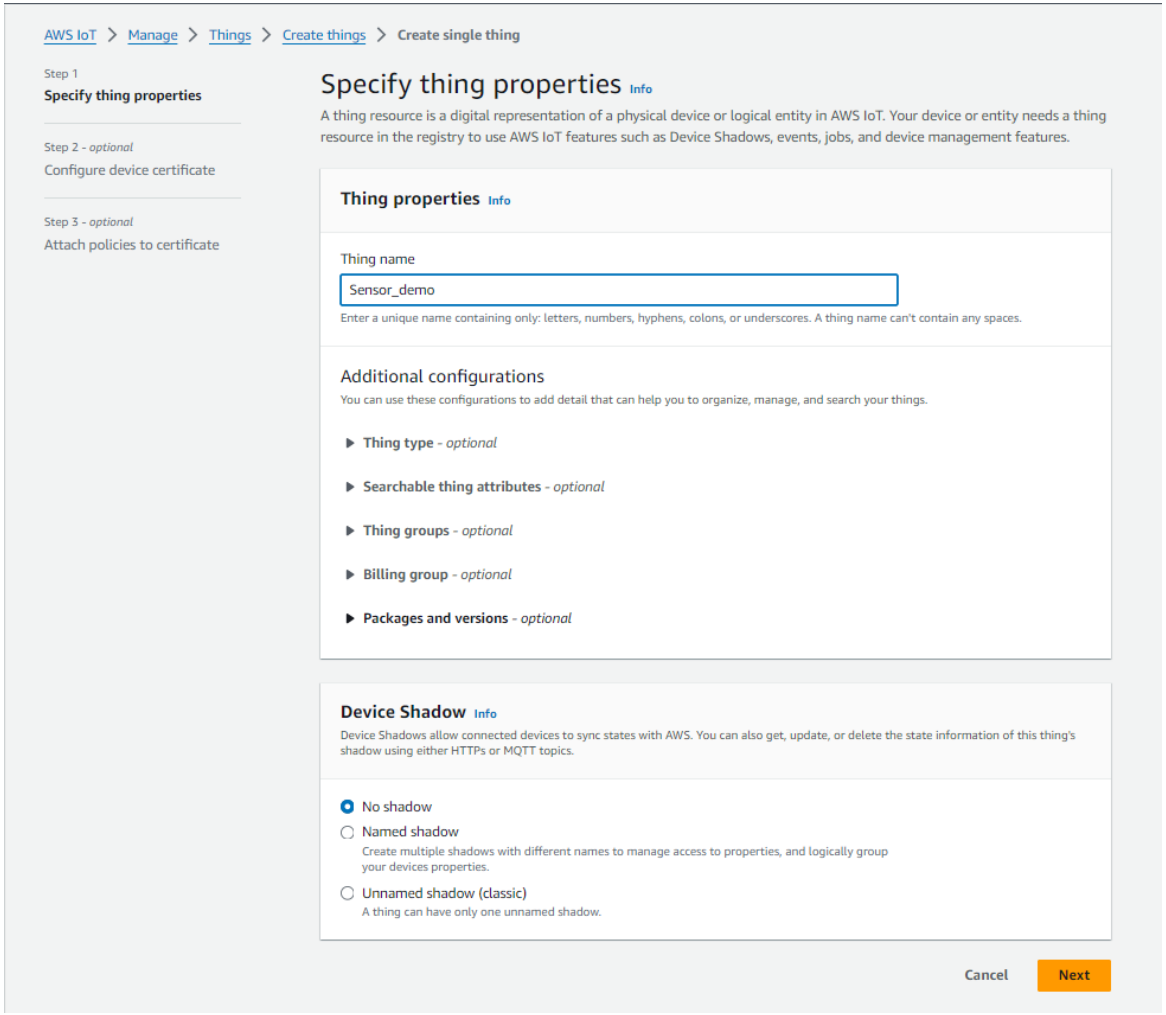
In the left sidebar, click on 'Things' under 'All devices,' then click the 'Create thing' button to create a new IoT thing.



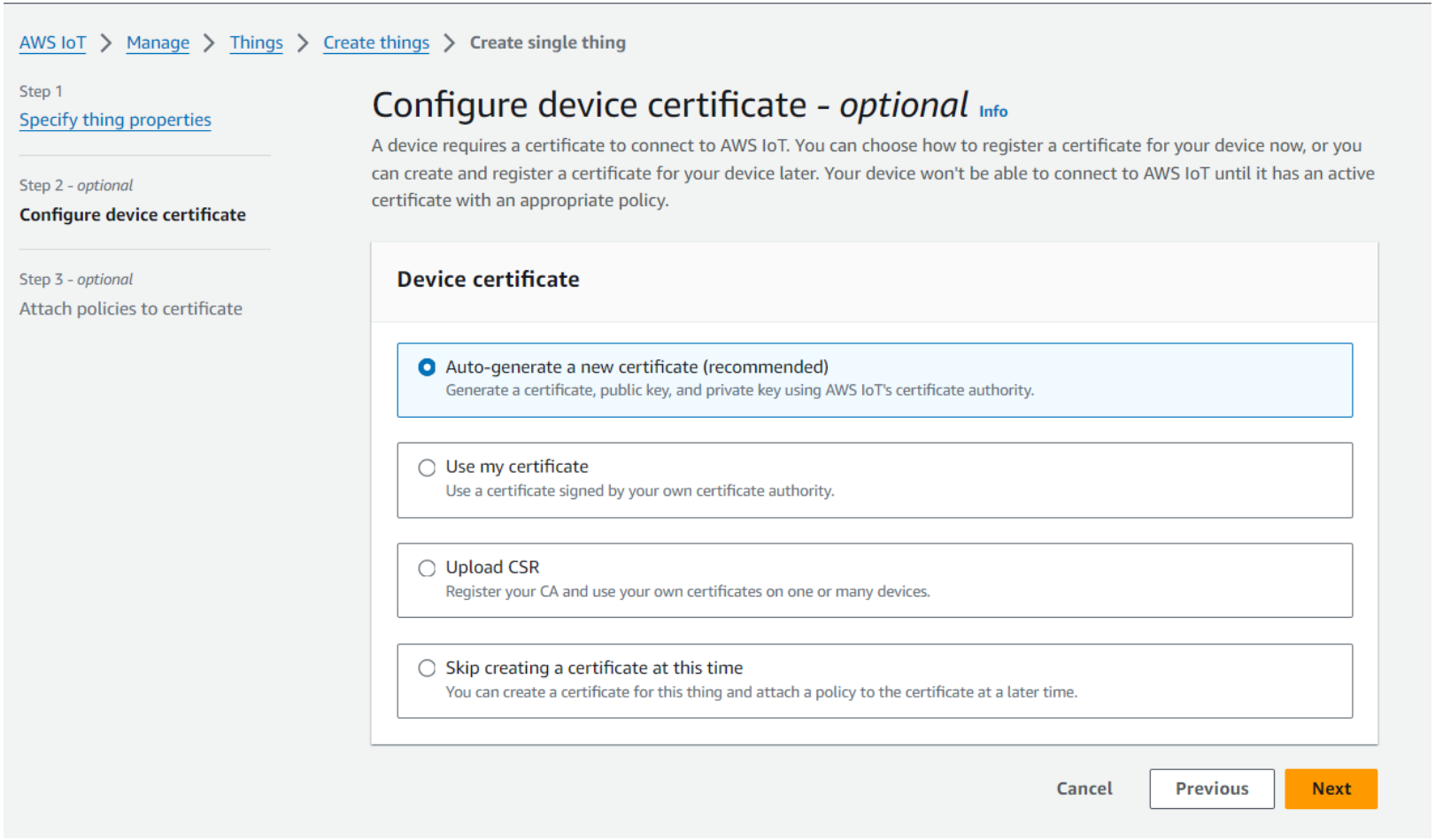
You can choose to create single thing or many things according to your need.



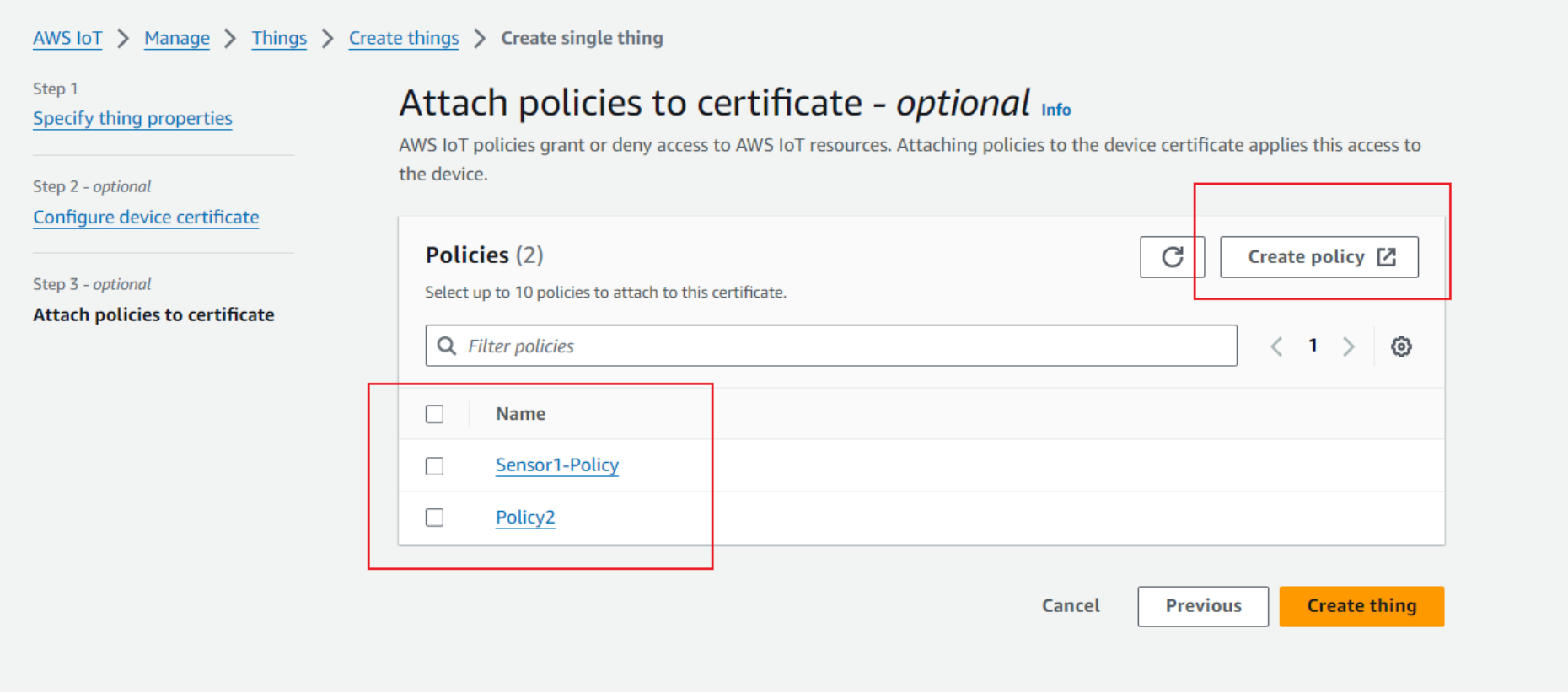
Enter a unique name for your device and click ‘next’.



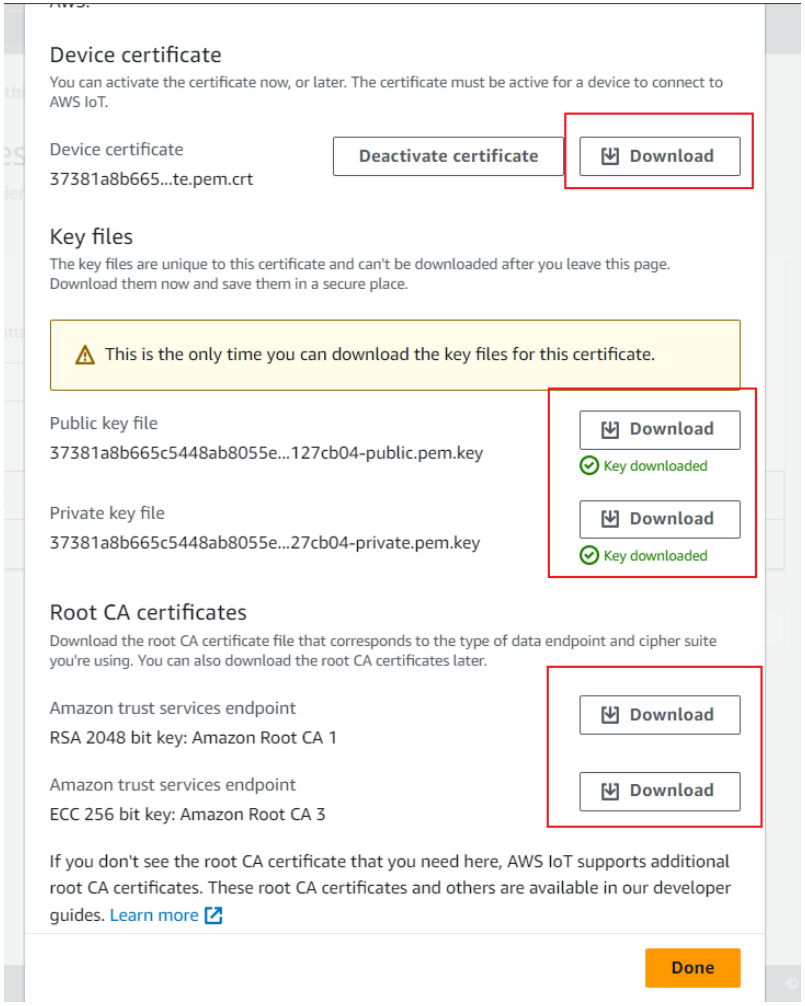
We need to generate a certificate for the thing. Just keep the default settings and click 'Next'.



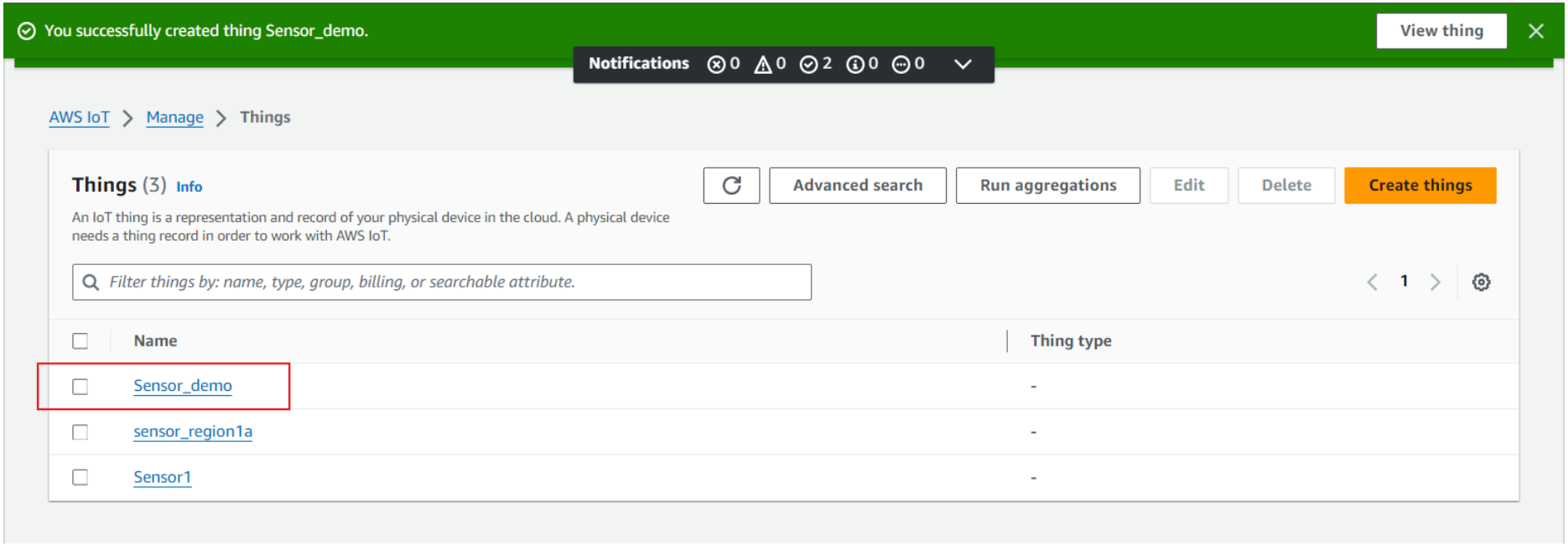
You can create a new policy or attach existing policies to the device certificate for access control. Then, click the ‘Create thing’ button.



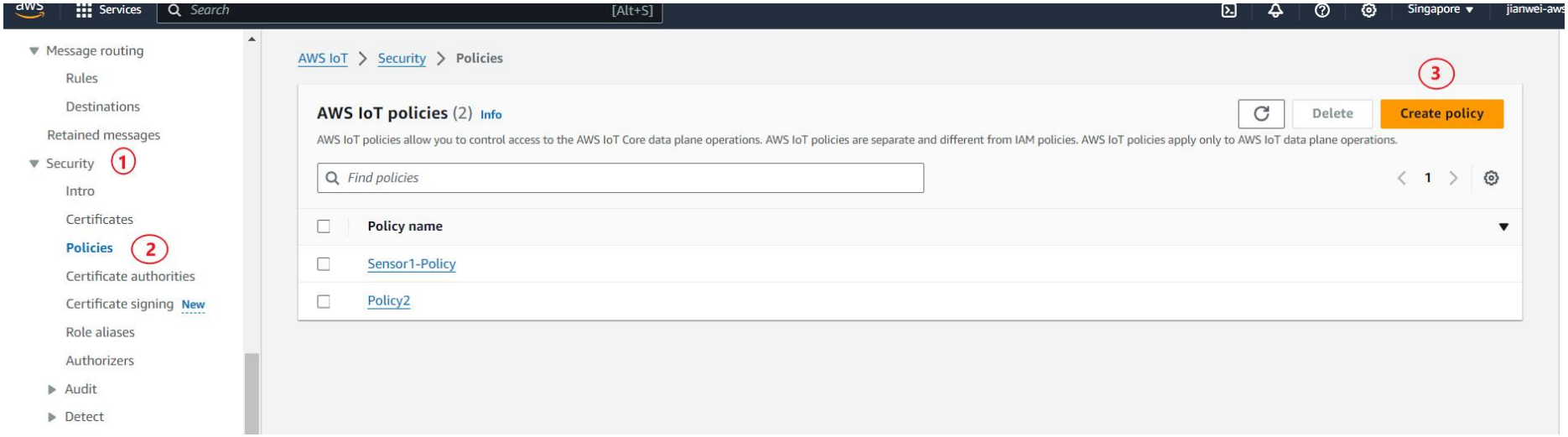
Download all the certificate and key files to install on your device. You will need them to connect to AWS later.



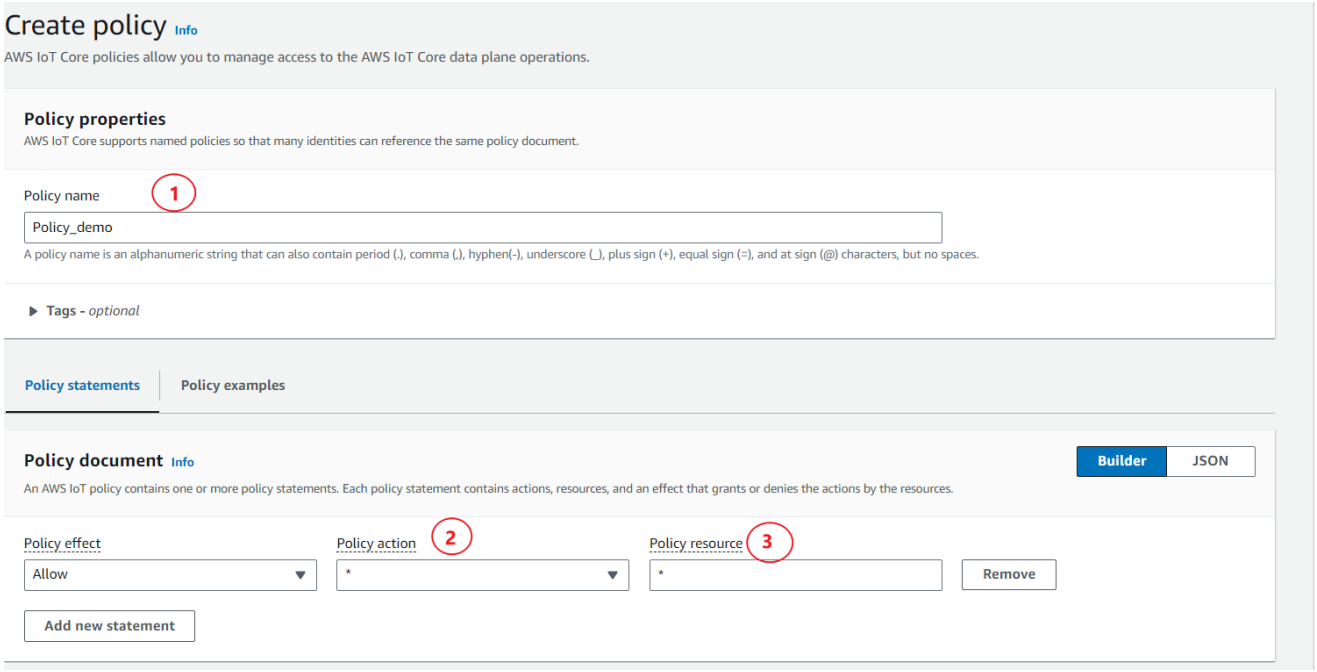
A new thing has been successfully created.



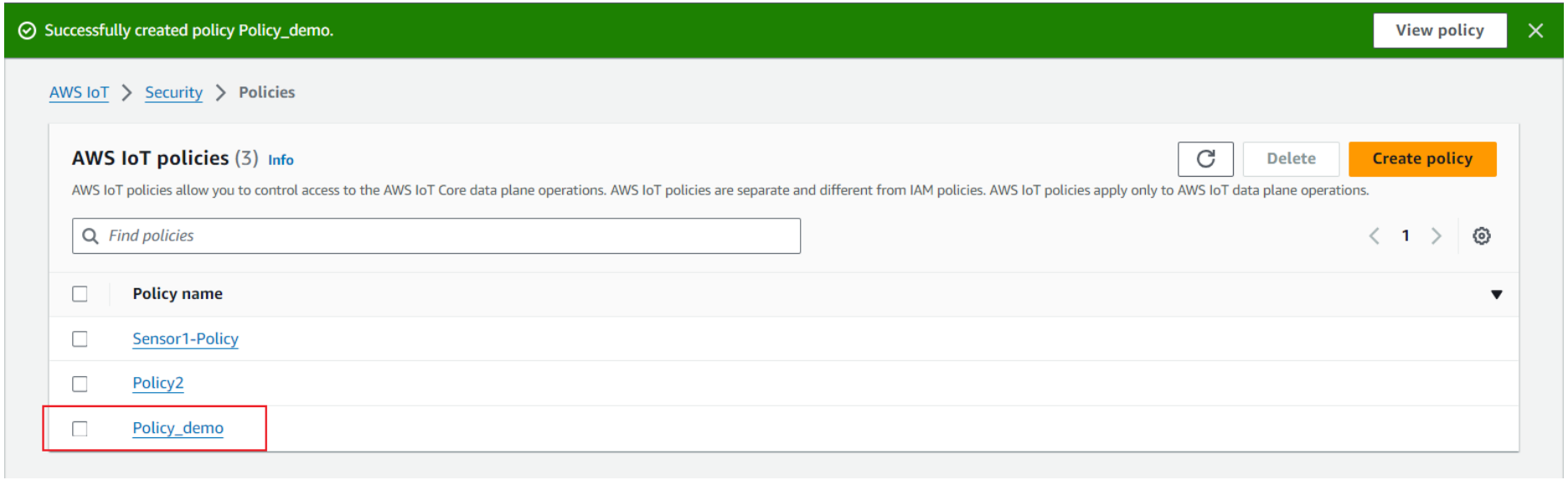
To attach a new policy to the created thing, navigate to 'Policies' under the 'Security' section, then click the 'Create policy' button.



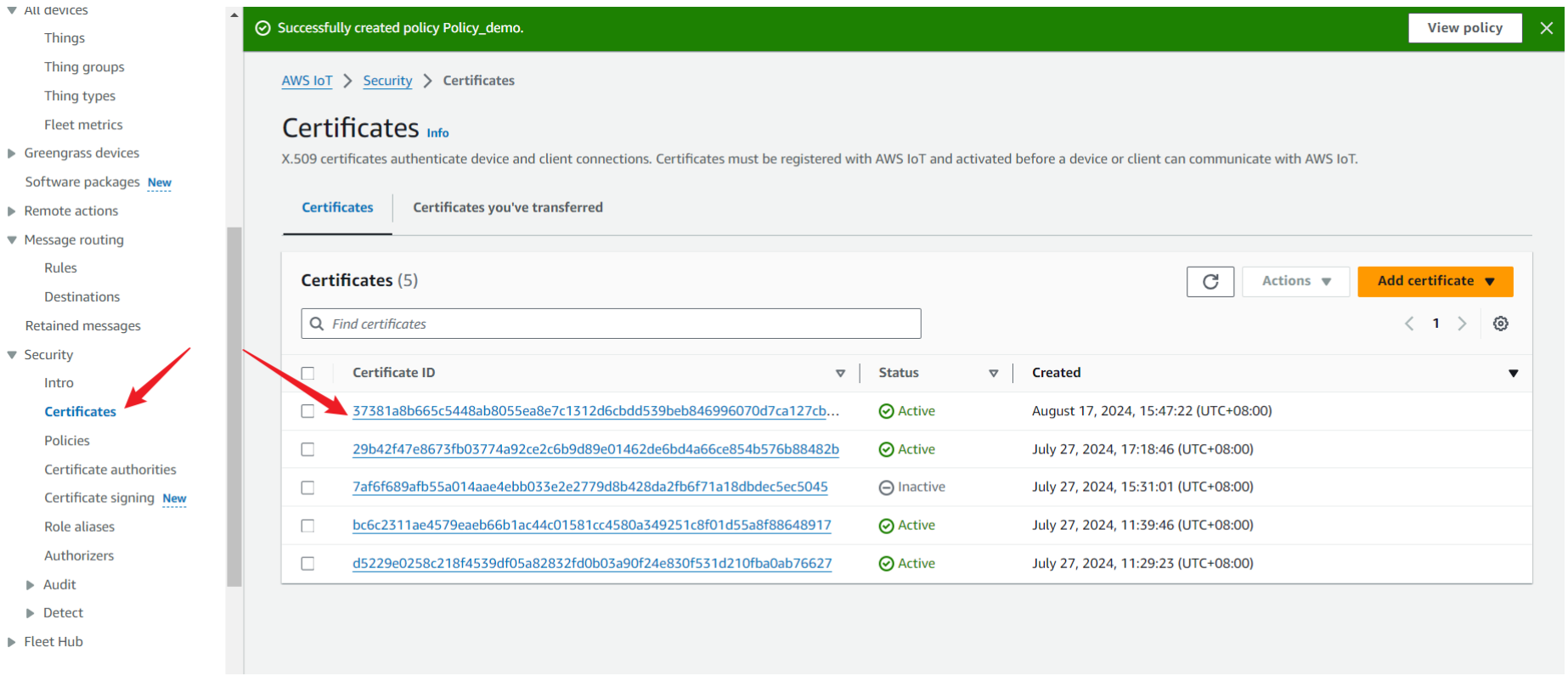
1. Enter your policy name.
2. Choose the desired policy actions.
3. Specify your AWS IoT resources, or use '*' to apply the policy to all resources.



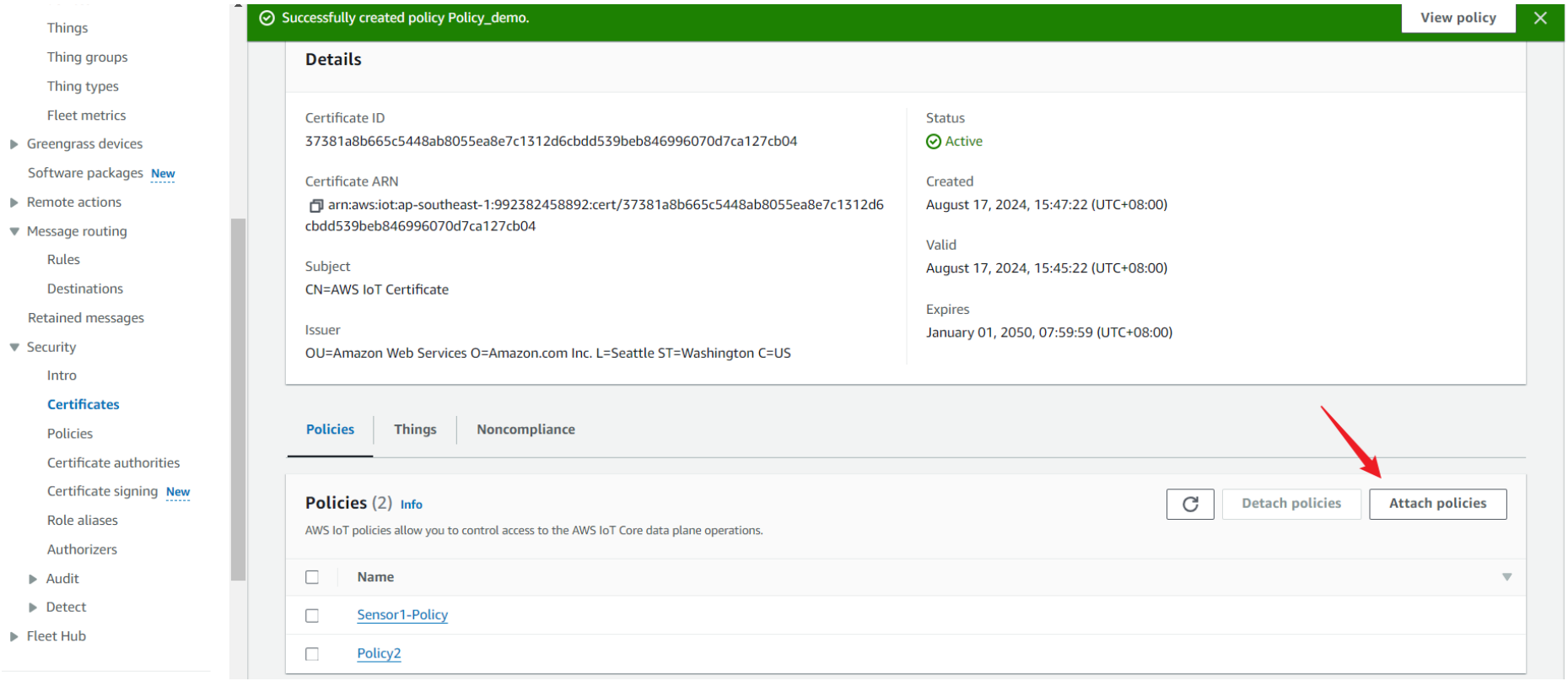
A new policy has been successfully created.



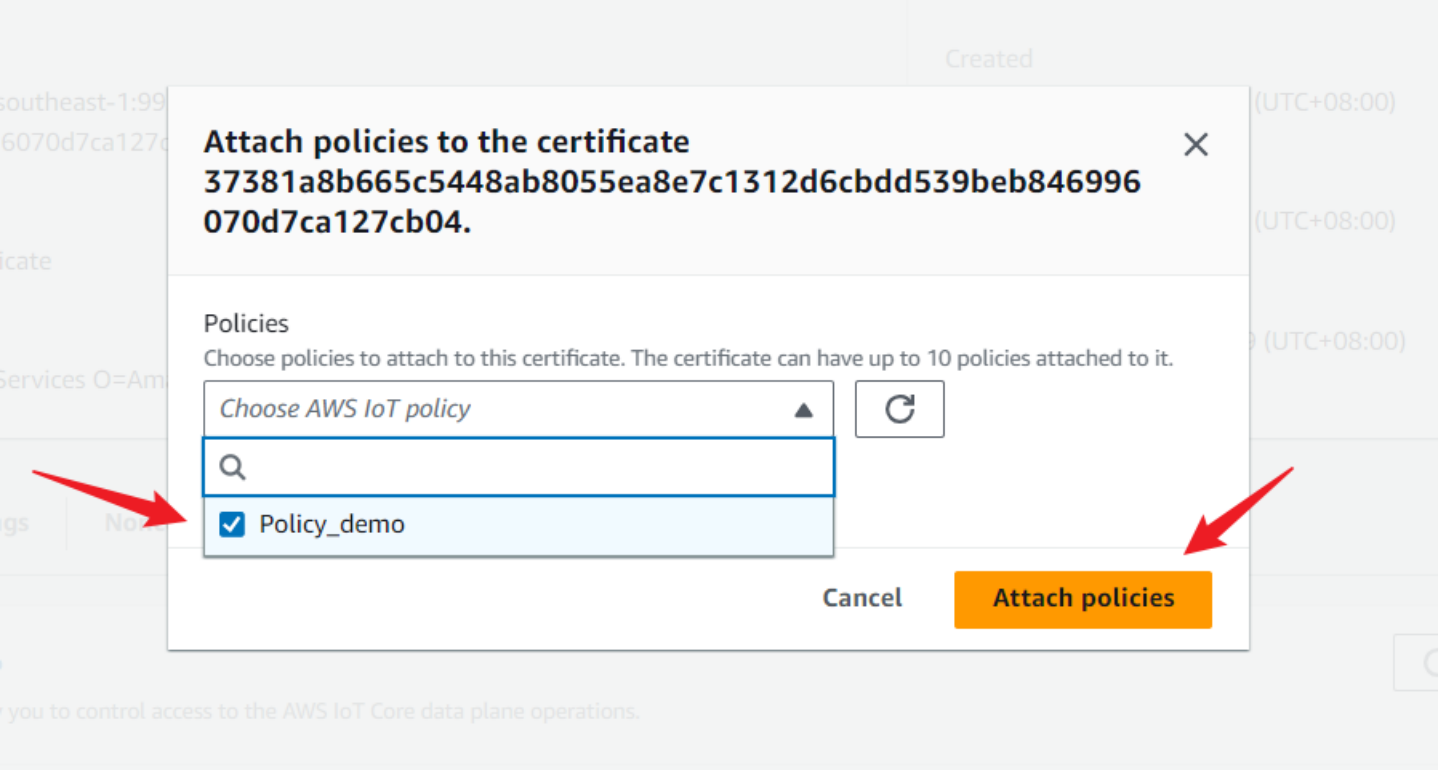
Now, navigate to the ‘Certificate’ under ‘Security’ section and click on the certificate that you wish to attach the new policy.



Click ‘Attach policies’.



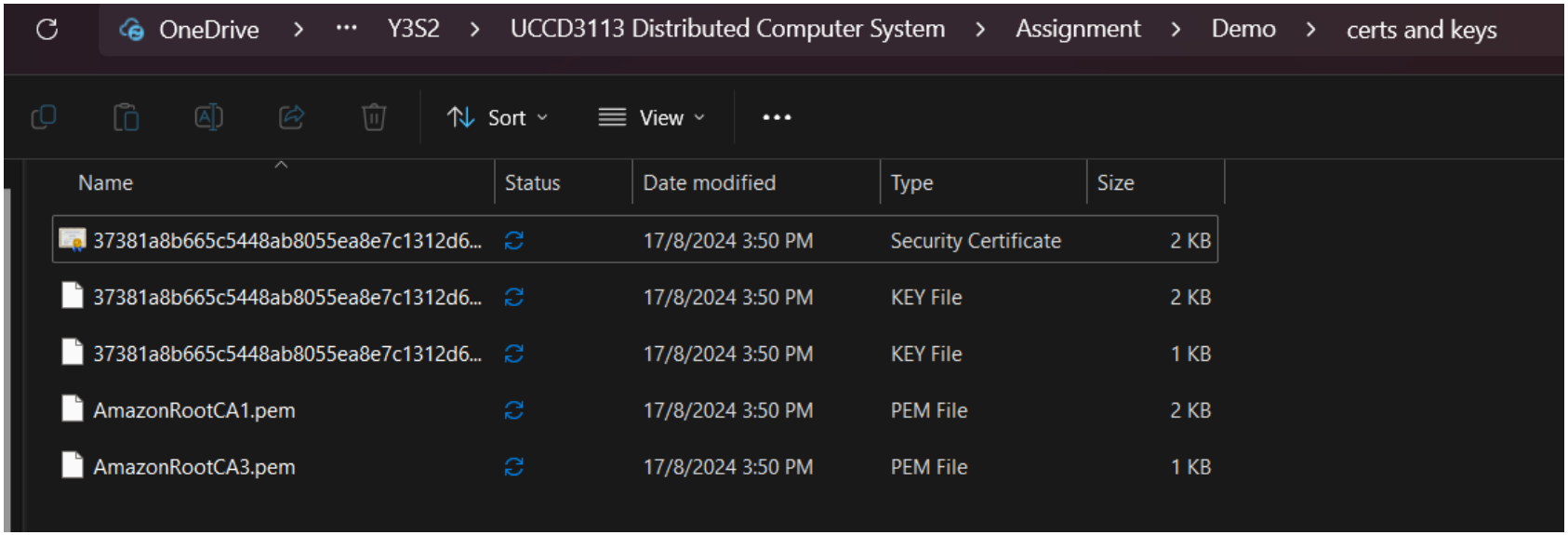
Choose the policy that you created and click ‘Attach policies’.



AWS IoT SDK

The AWS IoT SDK is a comprehensive toolkit designed to streamline the development, management, and scaling of IoT solutions. It offers features such as device management, data collection, real-time processing, and secure communication through various protocols like MQTT. The SDK supports multiple programming languages, including Python, Java, C++, and JavaScript (Node.js). For more details, refer to this link: https://docs.aws.amazon.com/iot/latest/developerguide/iot-sdks.html. In this manual, we will use Python as our primary language. You can access the device SDK from this link: https://github.com/aws/aws-iot-device-sdk-python-v2.

Put all the certificates and keys that downloaded in previous step in one folder.



- Imports necessary modules from the AWS IoT SDK and sets up the connection parameters using TLS (certificate and key files).
- Connects to an AWS IoT Core endpoint.

```
from awscrt import io, mqtt, auth, http

from awsiot import mqtt_connection_builder

import time as t

import json

import random

import threading as th

import uuid
```

```
ENDPOINT = "a2dfazm7rcnmac-ats.iot.ap-southeast-1.amazonaws.com"
PATH_TO_CERTIFICATE =
"Demo/37381a8b665c5448ab8055ea8e7c1312d6cbdd539beb846996070d7ca127cb04-certificate.pem.crt"
PATH_TO_PRIVATE_KEY =
"Demo/37381a8b665c5448ab8055ea8e7c1312d6cbdd539beb846996070d7ca127cb04-private.pem.key"
PATH_TO_AMAZON_ROOT_CA_1 = "Demo/AmazonRootCA1.pem"
CLIENT_ID = str(uuid.uuid4())
TOPIC_SUB = "env/volume"
RANGE = 1

received_count = 0

received_all = th.Event()

event_loop_group = io.EventLoopGroup(1)

host_resolver = io.DefaultHostResolver(event_loop_group)

client_bootstrap = io.ClientBootstrap(event_loop_group, host_resolver)

mqtt_connection = mqtt_connection_builder.mtls_from_path(

  endpoint=ENDPOINT, cert_filepath=PATH_TO_CERTIFICATE,pri_key_filepath=PATH_TO_PRIVATE_KEY,

  client_bootstrap=client_bootstrap,ca_filepath=PATH_TO_AMAZON_ROOT_CA_1, client_id=CLIENT_ID,

  clean_session=True, keep_alive_secs=6

)

print(f"Connecting to {ENDPOINT} with client ID '{CLIENT_ID}'...")

connect_future = mqtt_connection.connect()

connect_future.result()

print("Connected!")
```

Subscribes to the MQTT topic env/volume. When a message is received on this topic, a callback function (on_message_received) is executed, which prints the message and counts the number of messages received.

```
# Callback when the subscribed topic receives a message
def on_message_received(topic, payload, dup, qos, retain, **kwargs):
    print(f"Message received from topic '{topic}': {payload}")
    global received_count
    received_count += 1
    if received_count == RANGE:
        received_all.set()

print(f"Subscribing to topic '{TOPIC_SUB}'...")
subscribe_future, packet_id = mqtt_connection.subscribe(
    topic=TOPIC_SUB,
    qos=mqtt.QoS.AT_LEAST_ONCE,
    callback=on_message_received
)

subscribe_result = subscribe_future.result()
print(f"Subscribed with QoS {subscribe_result['qos']}")
```

```
# Callback when the subscribed topic receives a message
def on_message_received(topic, payload, dup, qos, retain, **kwargs):
    print(f"Message received from topic '{topic}': {payload}")
    global received_count
    received_count += 1
    if received_count == RANGE:
        received_all.set()

print(f"Subscribing to topic '{TOPIC_SUB}'...")
subscribe_future, packet_id = mqtt_connection.subscribe(
    topic=TOPIC_SUB,
    qos=mqtt.QoS.AT_LEAST_ONCE,
    callback=on_message_received
)

subscribe_result = subscribe_future.result()
print(f"Subscribed with QoS {subscribe_result['qos']}")
```

Generates random sensor data (soil moisture, temperature, humidity, wind speed) along with a timestamp and a location ID. This data is formatted into a JSON object and published to different MQTT topics (device/1/data, device/2/data, etc.).

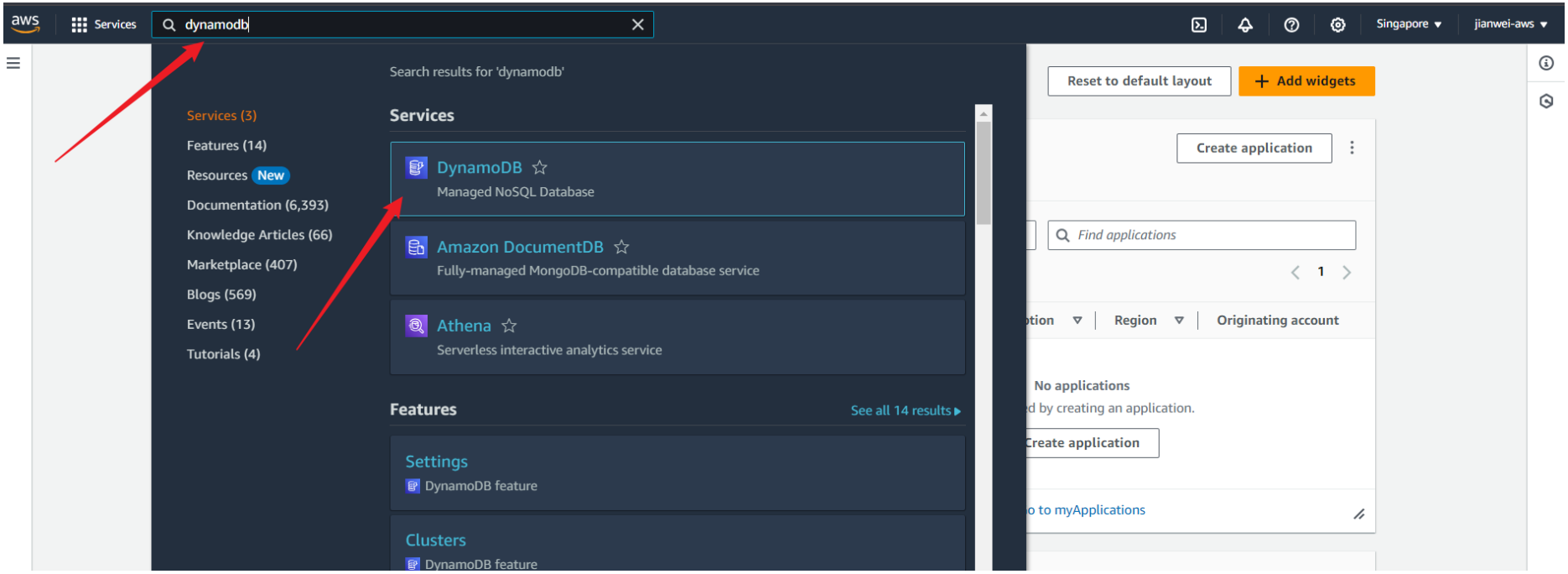
```
for i in range(RANGE):
    current_time = t.strftime("%d-%m-%Y %H:%M:%S", t.localtime())
    soil_moisture_level = random.randint(0, 100)
    temperature = random.randint(0, 40)
    humidity = random.randint(0, 100)
    wind_speed = random.randint(0, 12)
    location_id = f"region_{i + 1}"
    TOPIC_PUB = f"device/{i + 1}/data"
    message_pub = {
        "location_id": location_id,
        "soil_moisture_level": soil_moisture_level,
        "temperature": temperature,
        "humidity": humidity,
        "wind_speed": wind_speed,
        "date_time": current_time
    }
    mqtt_connection.publish(
        topic=TOPIC_PUB,
        payload=json.dumps(message_pub),
        qos=mqtt.QoS.AT_LEAST_ONCE
    )
    print(f"Message published to topic '{TOPIC_PUB}':
{json.dumps(message_pub)}")
    t.sleep(1)
```

After receiving all replies, it disconnects from the MQTT broker and ends the program.

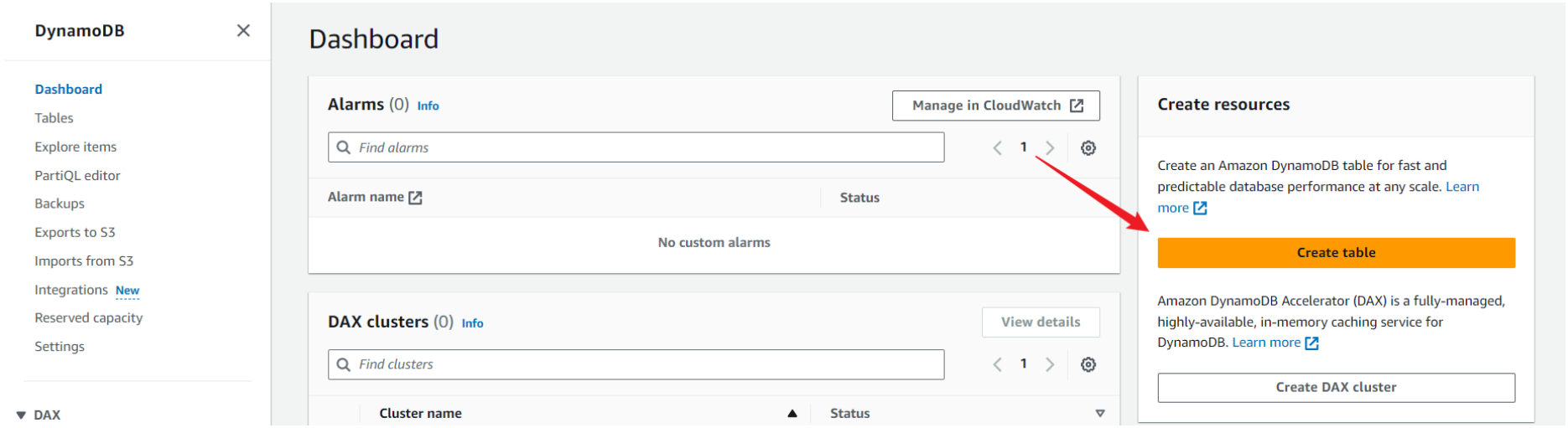
```
if not received_all.is_set():
    print("Awaiting reply...")
received_all.wait()
print("All replies received")


print("Disconnecting...")
disconnect_future = mqtt_connection.disconnect()
disconnect_future.result()
print("Disconnected. Program complete.")
```

Amazon DynamoDB

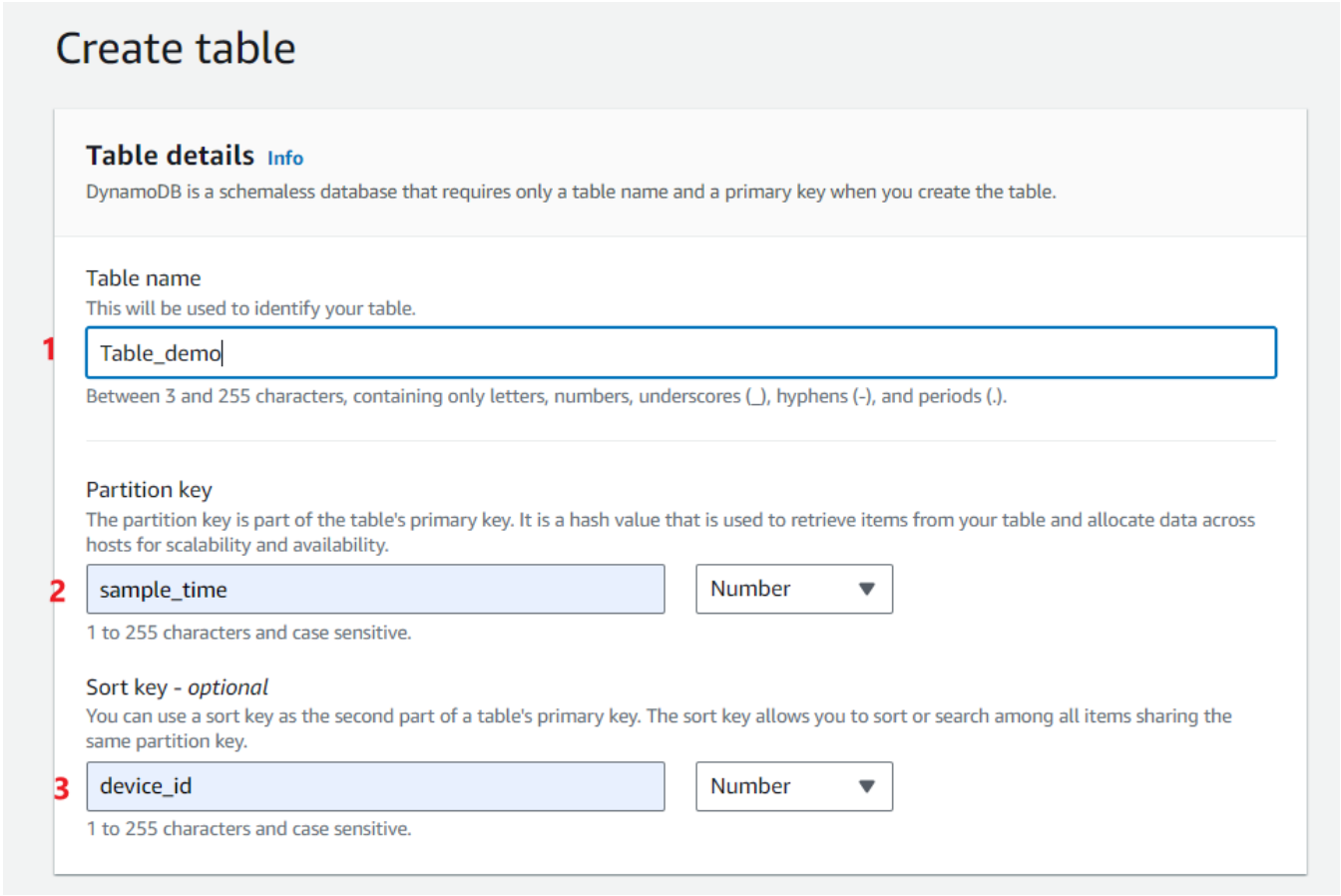
Type “dynamodb” in the search bar and click on the “DynamoDB” service.



Click “Create table”.



1. Enter 'Table_demo' as the table name.
2. Enter 'sample_time' as the Partition key and select **Number** as the data type.
3. Enter 'device_id' as the Sort key and select **Number** as the data type.
4. At the bottom of the page, click **Create**.

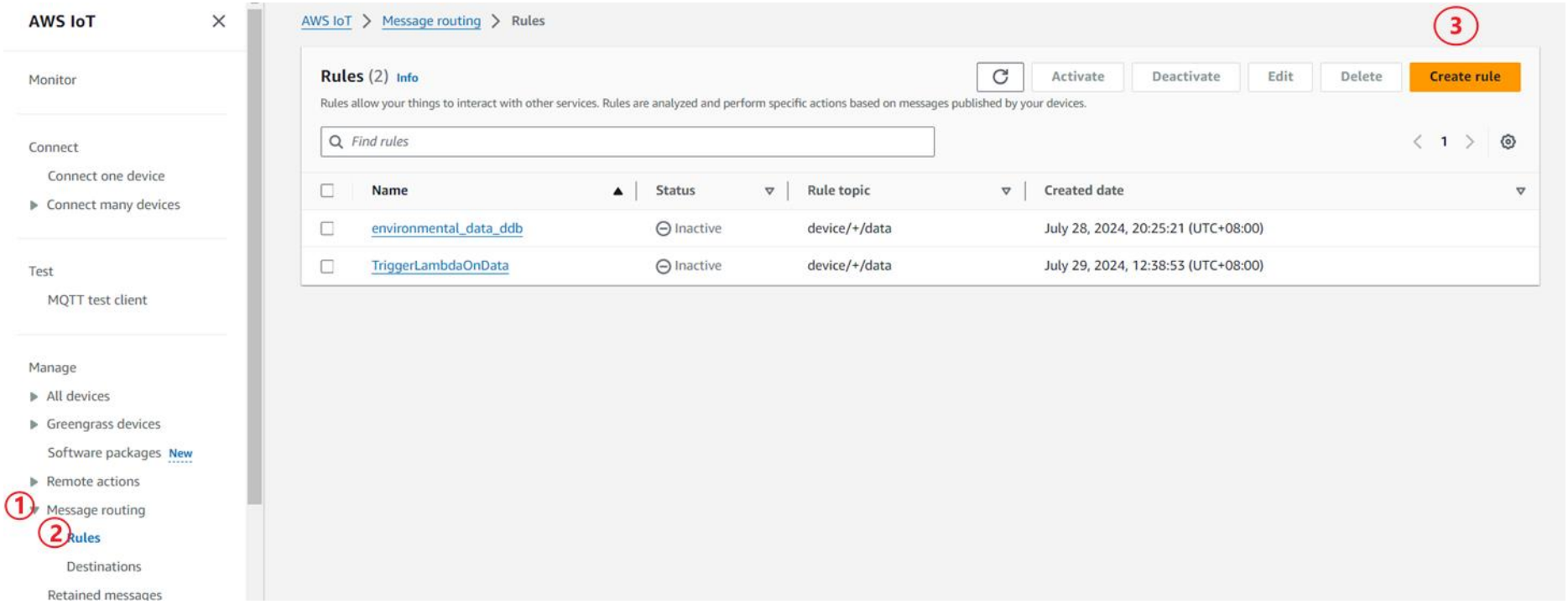


- ***sample_time*** serves as the primary key, representing the timestamp when the sample was recorded.
- ***device_id*** functions as the sort key, identifying the device that provided the sample.
- ***device_data*** contains the data received from the device, formatted by the rule query statement. It will be defined later when configuring the DynamoDB rule action.

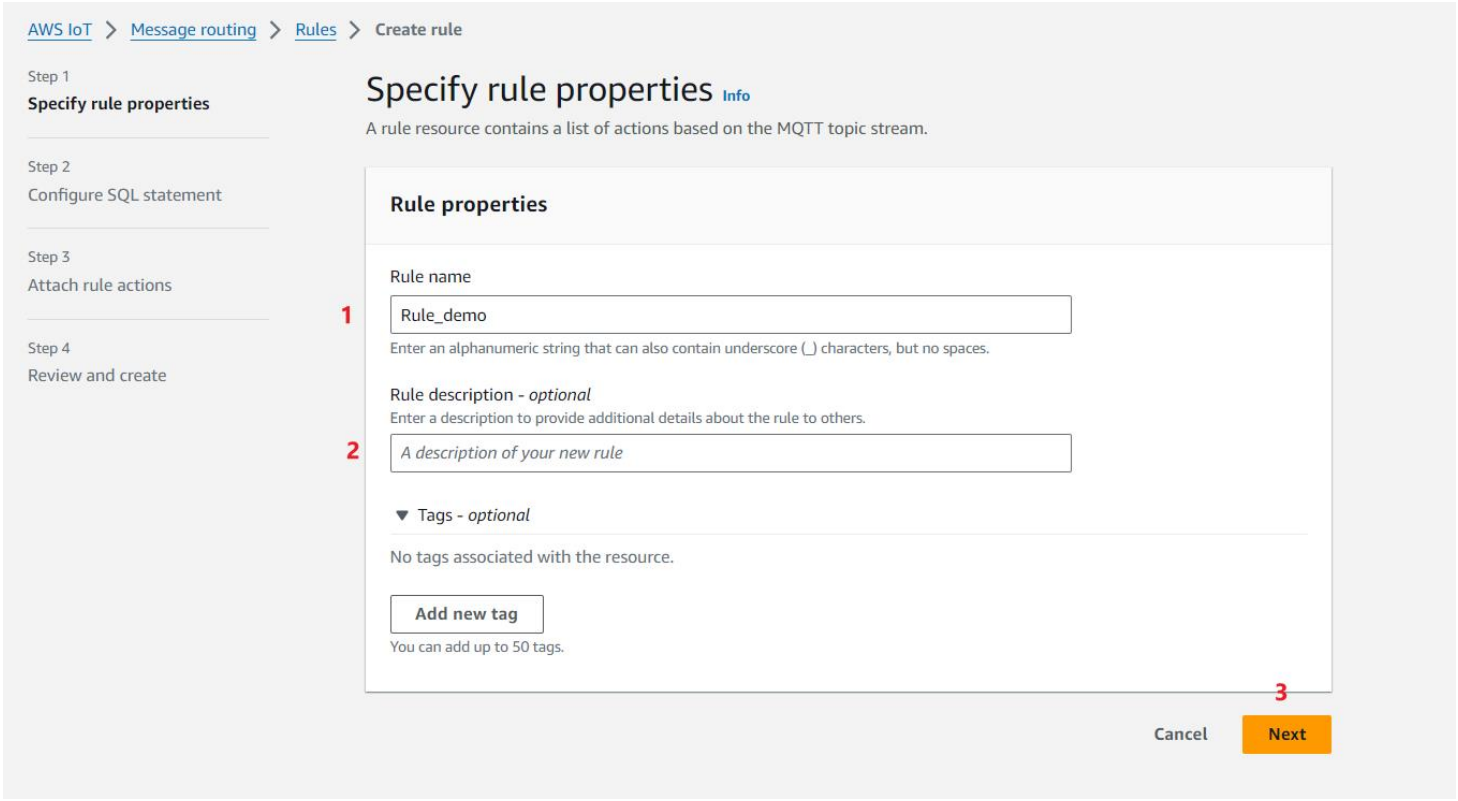
Now, we need to create an AWS IoT rule to send data to the DynamoDB table.

AWS IoT Rule

AWS IoT Rules enable devices to interact with AWS services by processing data sent from IoT devices. These rules listen for specific MQTT messages, apply SQL-based query statements to filter and transform the data, and then route the processed data to various AWS services, such as DynamoDB in our case. First, navigate to the IoT Core service. Then, in the left navigation bar, click on "Rules" under the "Message routing" section, and select "Create rule."

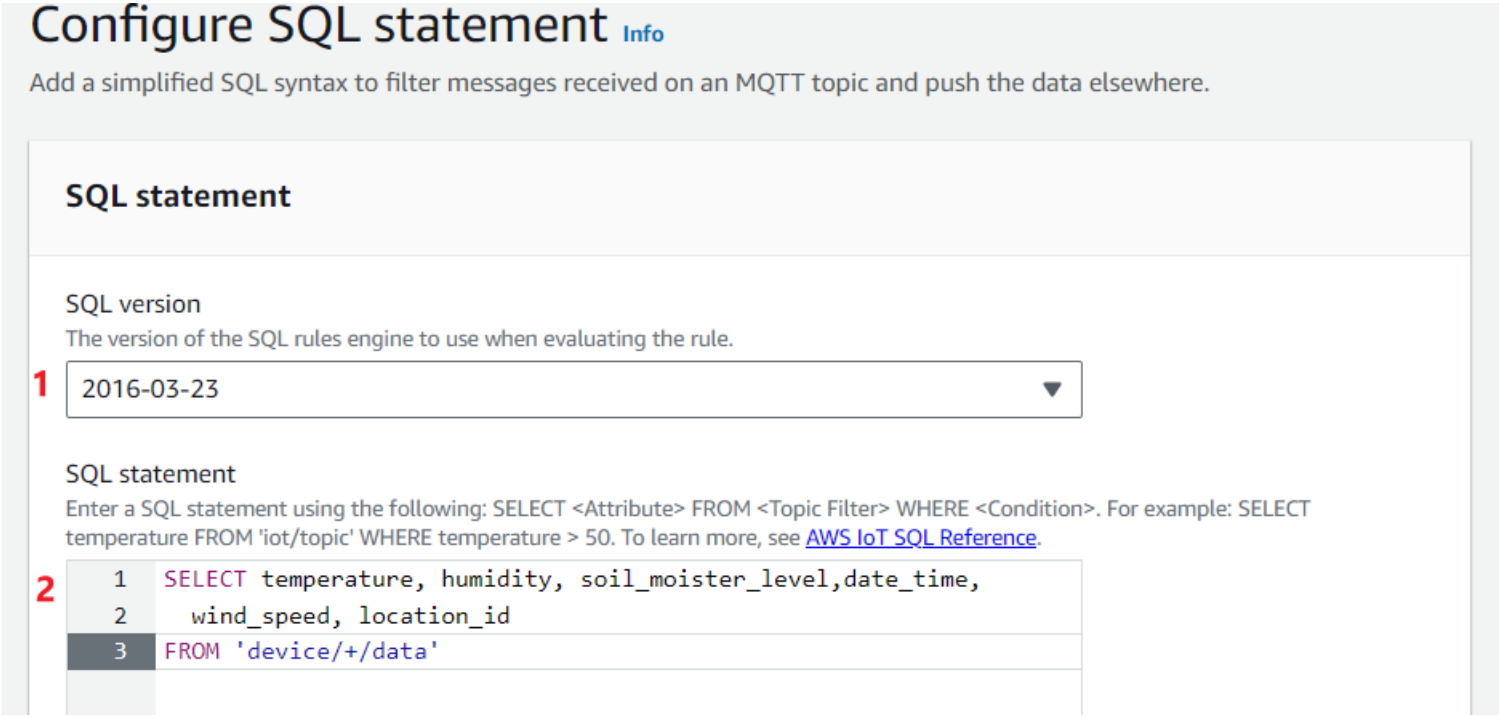


1. In the Rule name field, enter 'Rule_demo'.
2. In the Rule description field, provide a detailed description of the rule. A meaningful description will help you easily recall the purpose and function of this rule in the future.
3. Click Next to proceed.



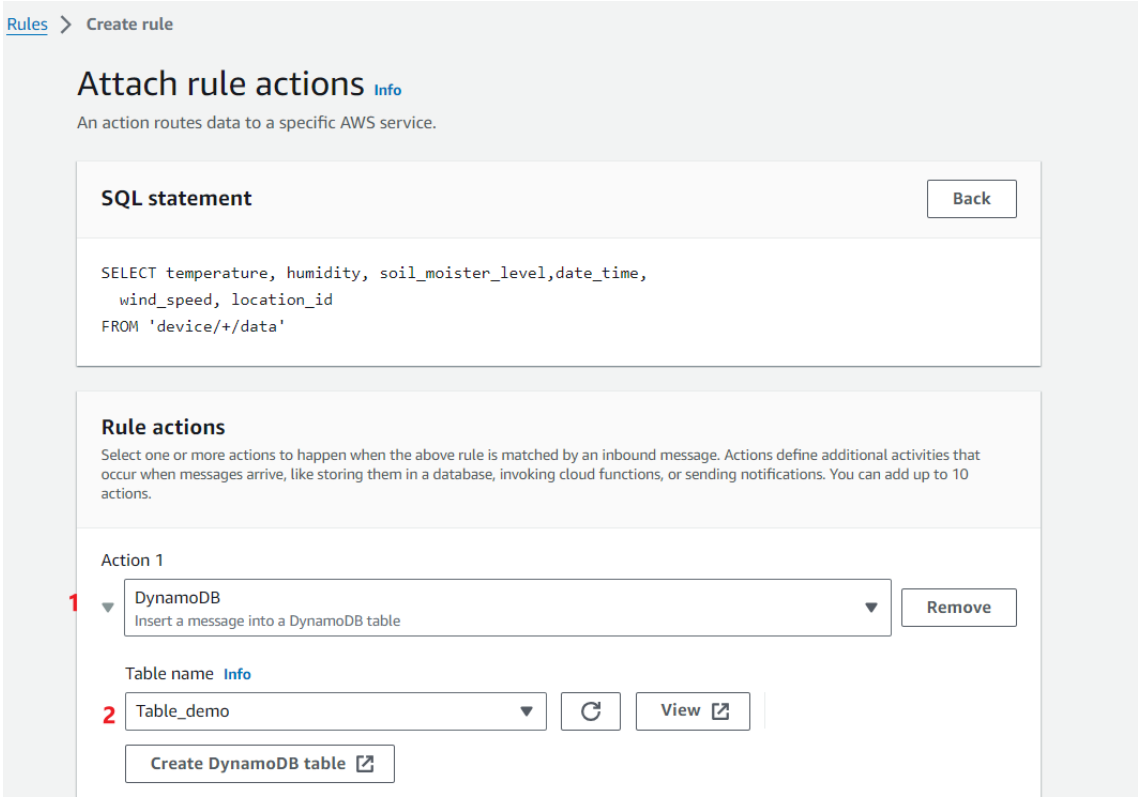
1. Select ***2016-03-23*** in SQL version.
2. Enter the statement in the SQL statement edit box:

```
SELECT temperature, humidity,
soil_moister_level,date_time,
 wind_speed, location_id
FROM 'device/+/data'
```



The SQL statement listens for MQTT messages that match the device/+/data topic filter and passes all the attributes unchanged.

1. In Rule actions, choose ***DynamoDB*** in Action1.
2. In Table name, choose the name of the DynamoDB table you created in a previous step: ***Table_demo***.



1. For the Partition key, enter sample_time.
2. For the Partition key value, use ${timestamp()}. This will substitute the value returned by the timestamp function instead of using a value from the message payload.
3. For the Sort key, enter device_id.
4. For the Sort key value, use ${cast(topic(2) AS DECIMAL)}. This substitution template will take the second element from the topic name (the device's ID), cast it to a DECIMAL value, and insert it to match the numeric format of the key.
5. In the "Write message data to this column" field, enter device_data. This will create a device_data column in the DynamoDB table.
6. Leave the Operation field blank.

Partition key

The partition key (also called hash key) must match the partition key of the DynamoDB table that you created.

1 sample_time

Partition key type

The partition key (also called hash key) type can be STRING or NUMBER. The default value is STRING.

NUMBER

Partition key value

The partition key (also called hash key) value supports substitution templates that provide data at runtime.

2 ${timestamp()}

Sort key - *optional*

The sort key (also called range key) must match the sort key of the DynamoDB table that you created.

3 device_id

Sort key type

The sort key (also called range key) type can be STRING or NUMBER. The default value is STRING.

NUMBER

Sort key value

The sort key (also called range key) value supports substitution templates that provide data at runtime.

4 ${cast(topic(2) AS DECIMAL)}

Write message data to this column - *optional*

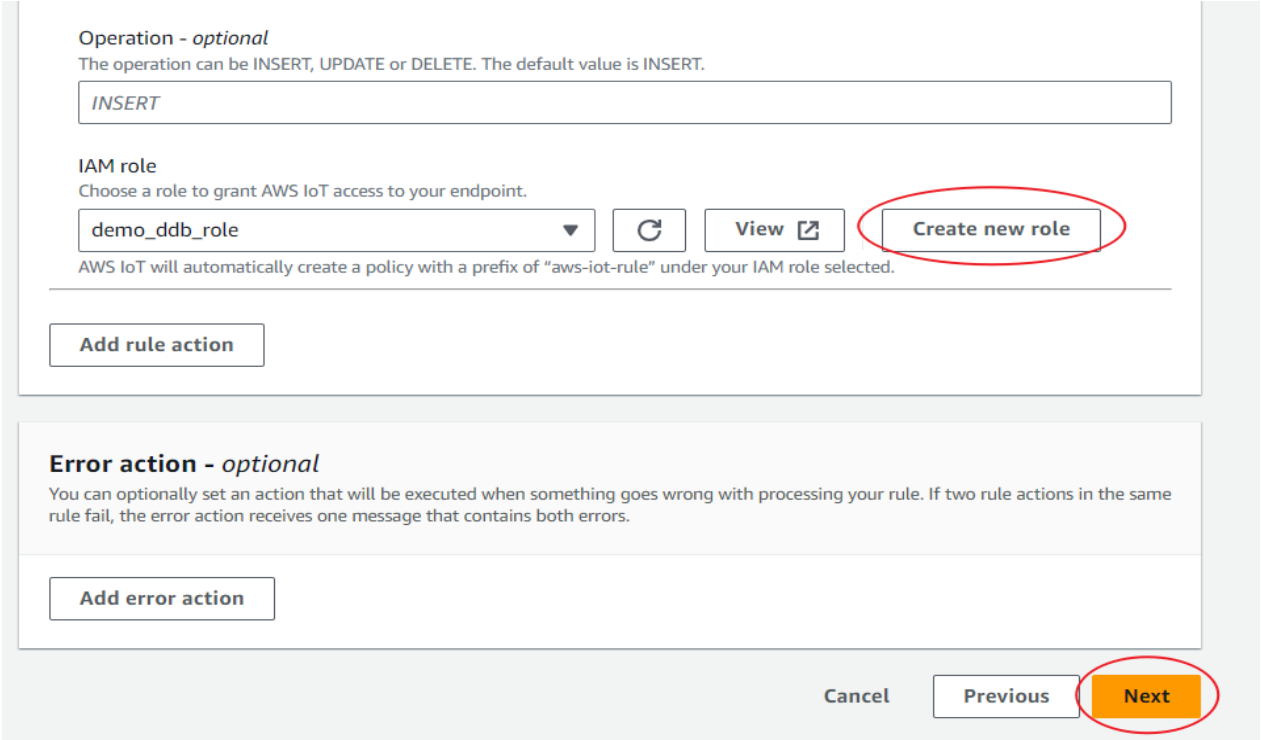
5 device_data

Operation - *optional*

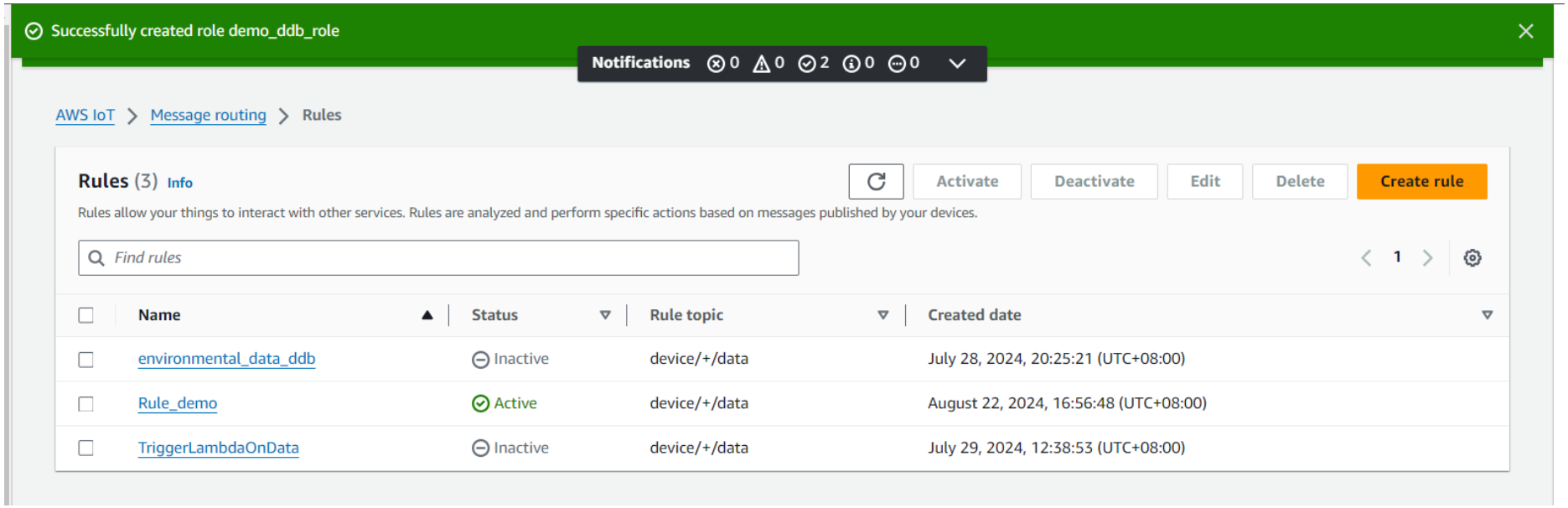
The operation can be INSERT, UPDATE or DELETE. The default value is INSERT.

6 INSERT

1. In IAM role, choose Create new role.
2. In the Create role dialog box, for Role name, enter ***demo_ddb_role***. This new role will automatically contain a policy with a prefix of "aws-iot-rule" that will allow the ***Rule_demo*** rule to send data to the ***Table_demo*** DynamoDB table you created.
3. In IAM role, choose wx_ddb_role.
4. At the bottom of the page, choose Next.

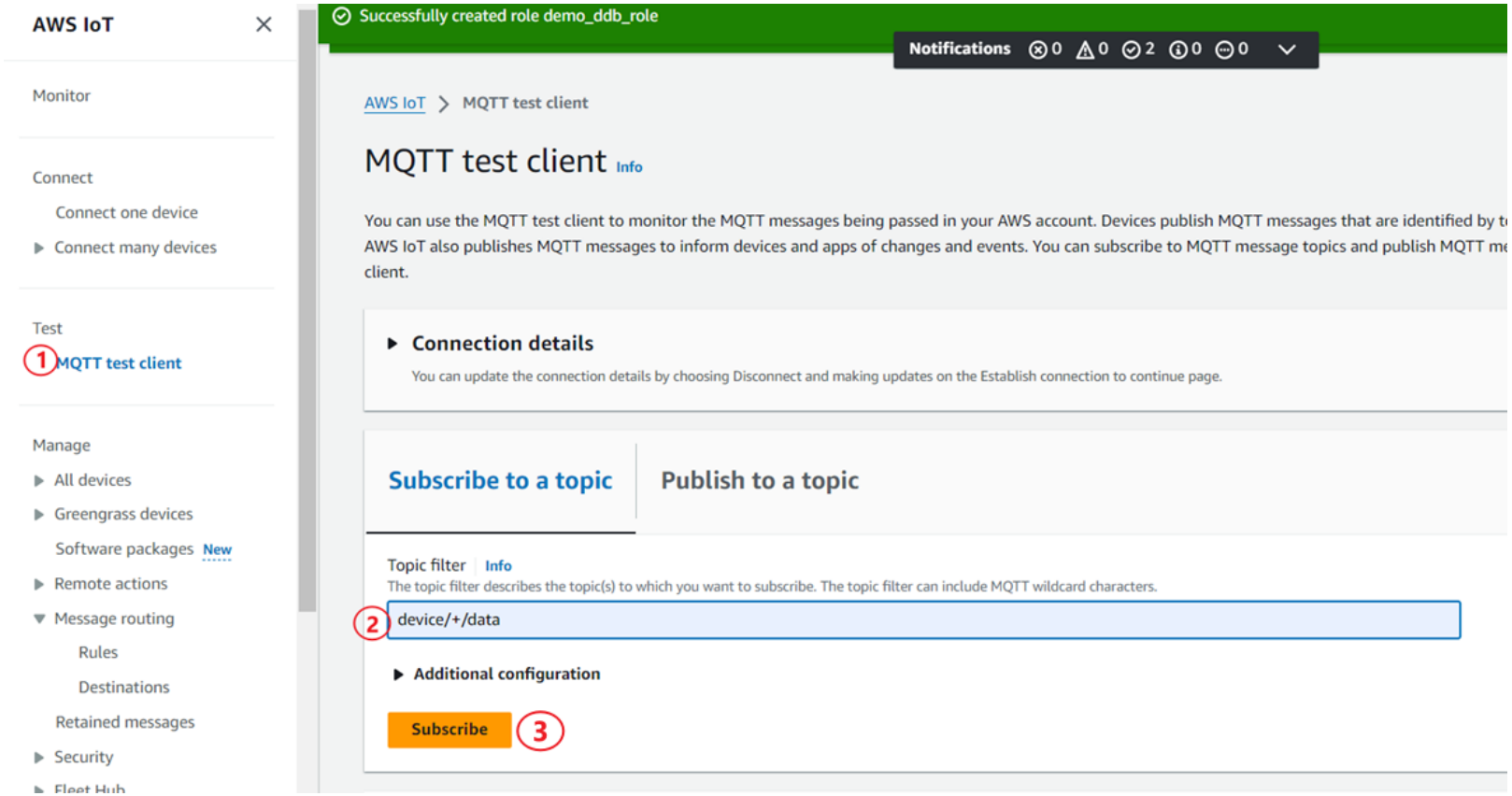


At the bottom of the "Review and create" page, click "Create" to finalize and create the rule. You will see that a new ***Rule_demo*** has been successfully created.



Next, we will use the MQTT client to publish and subscribe to the MQTT messages for testing the new rule.

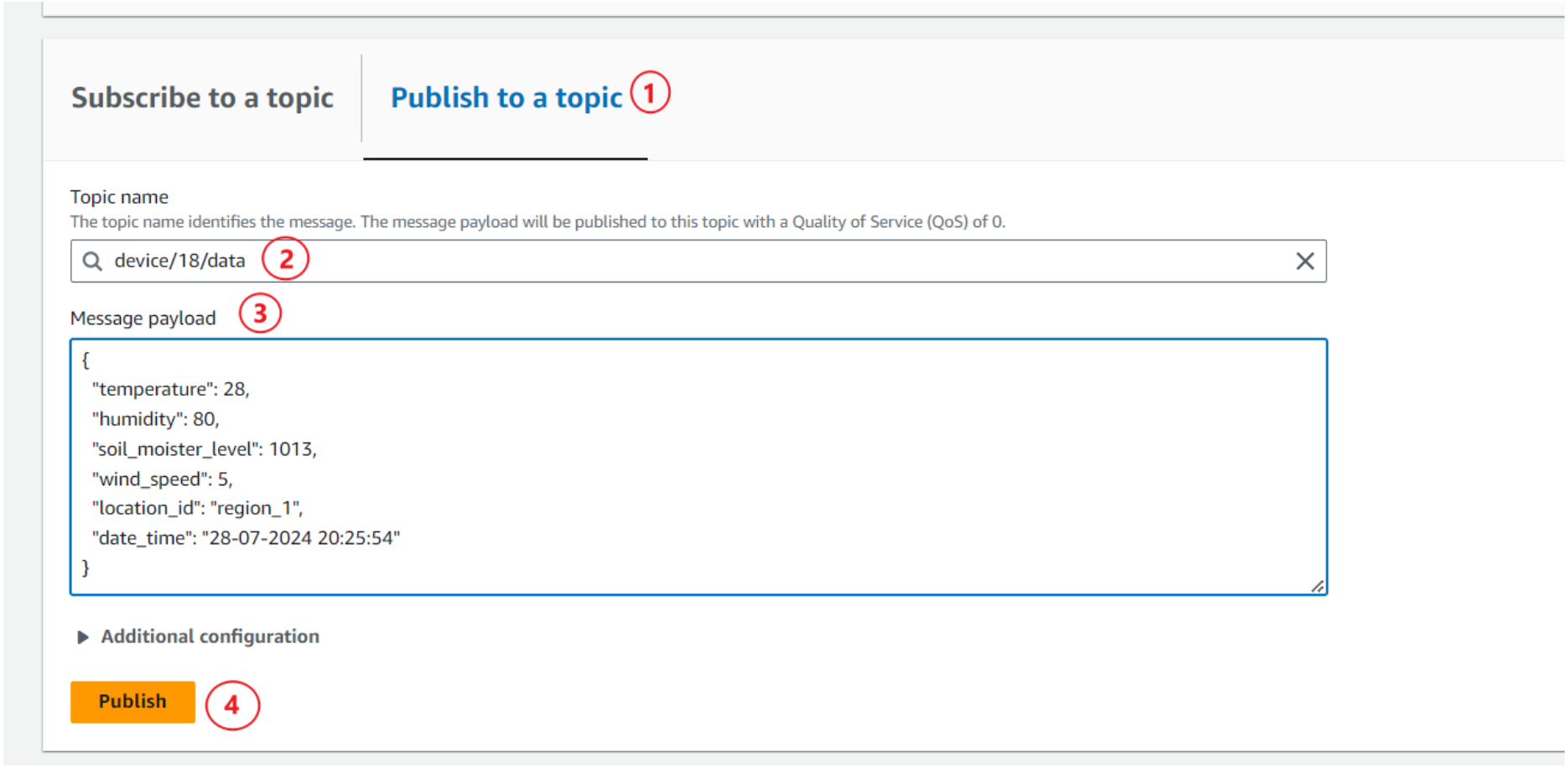
1. Click on the MQTT test client in the navigation bar.
2. Enter ***device/+/data*** as the input topic filter.
3. Click ‘Subscribe’.



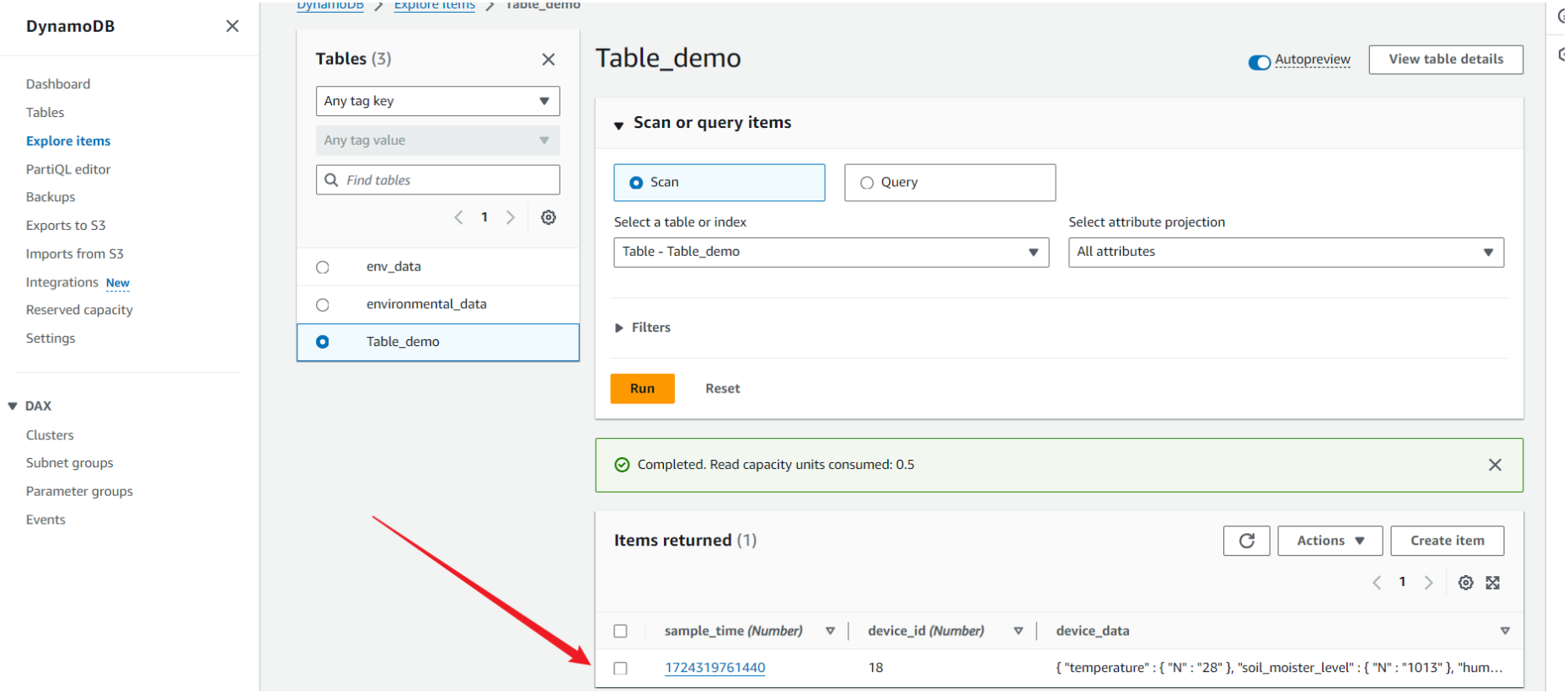
1. Choose ***Publish to a topic***.
2. Enter the input topic with a specific device ID, ***device/18/data***.
3. Enter the following sample data as ***Message payload.***

```
{
  "temperature": 28,
  "humidity": 80,
  "soil_moister_level": 1013,
  "wind_speed": 5,
  "location_id": "region_1",
  "date_time": "28-07-2024 20:25:54"
}
```

4. Choose ***Publish***


Subscribe to a topic
Publish to a topic
1
Topic name
The topic name identifies the message. The message payload will be published to this topic with a Quality of Service (QoS) of 0.
device/18/data
2
Message payload
3
{
"temperature": 28,
"humidity": 80,
"soil_moister_level": 1013,
"wind_speed": 5,
"location_id": "region_1",
"date_time": "28-07-2024 20:25:54"
}
Additional configuration
Publish
4

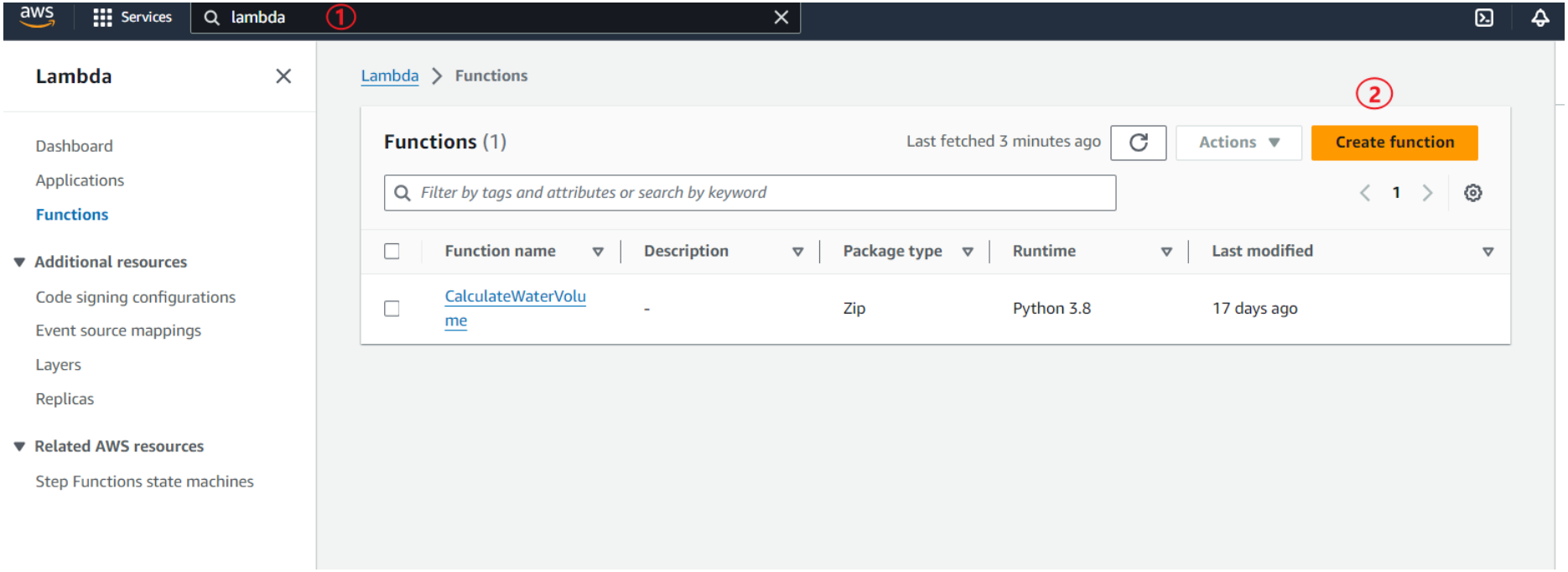
You will see that a new record has been inserted into DynamoDB successfully.



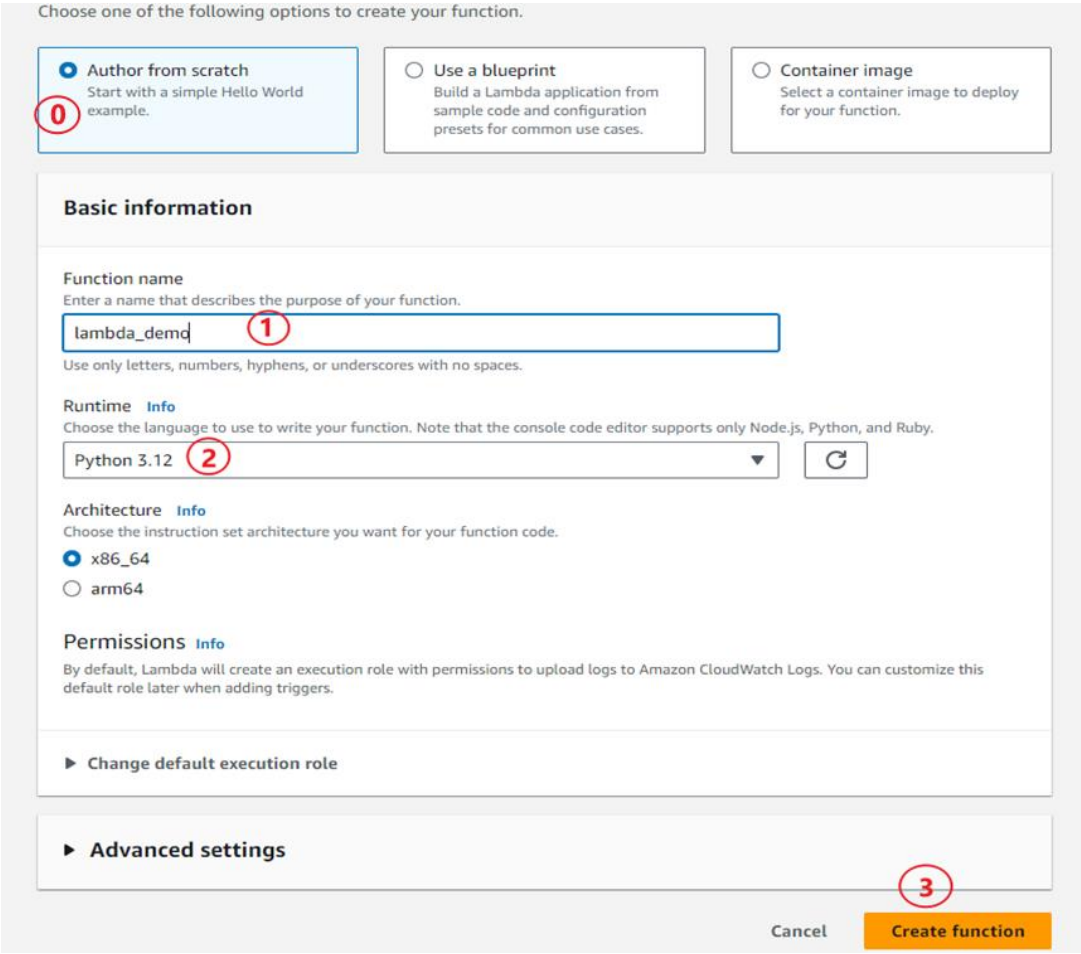
Next, when a new record is inserted into DynamoDB, we need to calculate the water volume based on the newly inserted environmental data and send the result back to the IoT device to trigger the irrigation instruction. This can be achieved using AWS Lambda, a serverless compute service that runs code without the need to provision or manage servers.

AWS Lambda

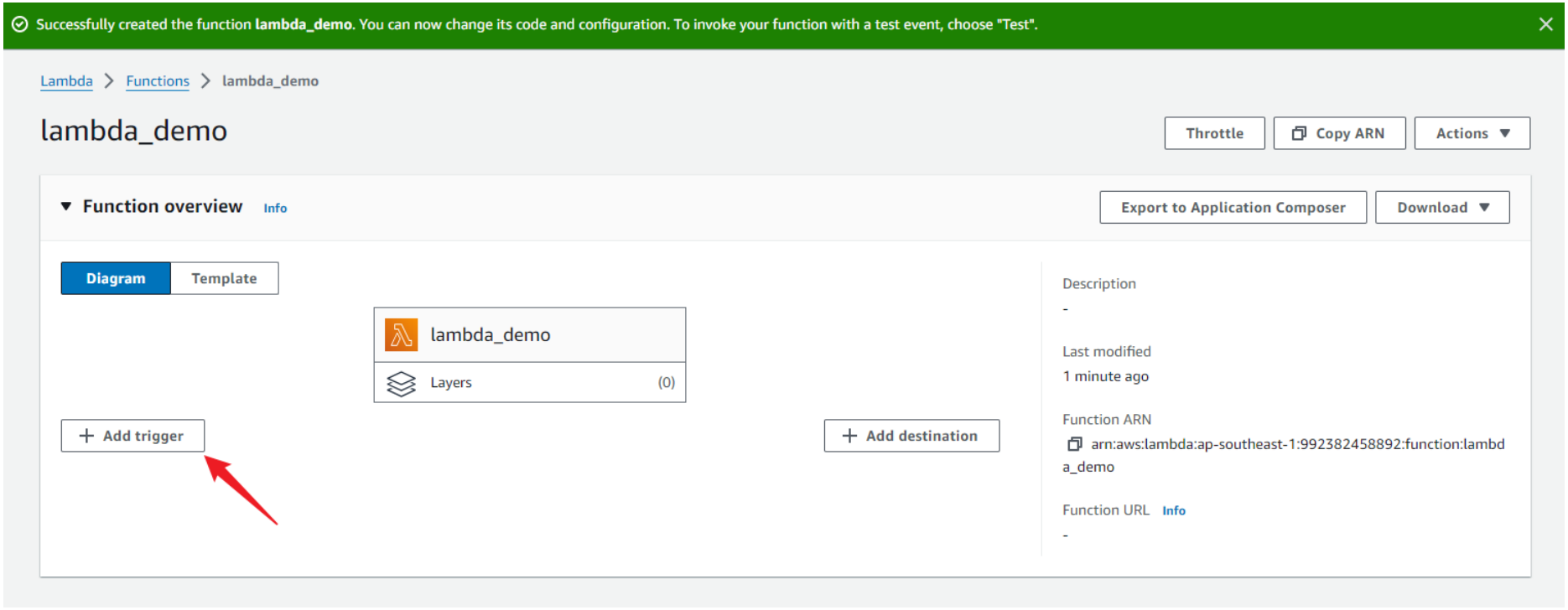
1. Enter ***Lambda*** in search bar and choose the ***Lambda*** service.
2. Click ***Create function***.



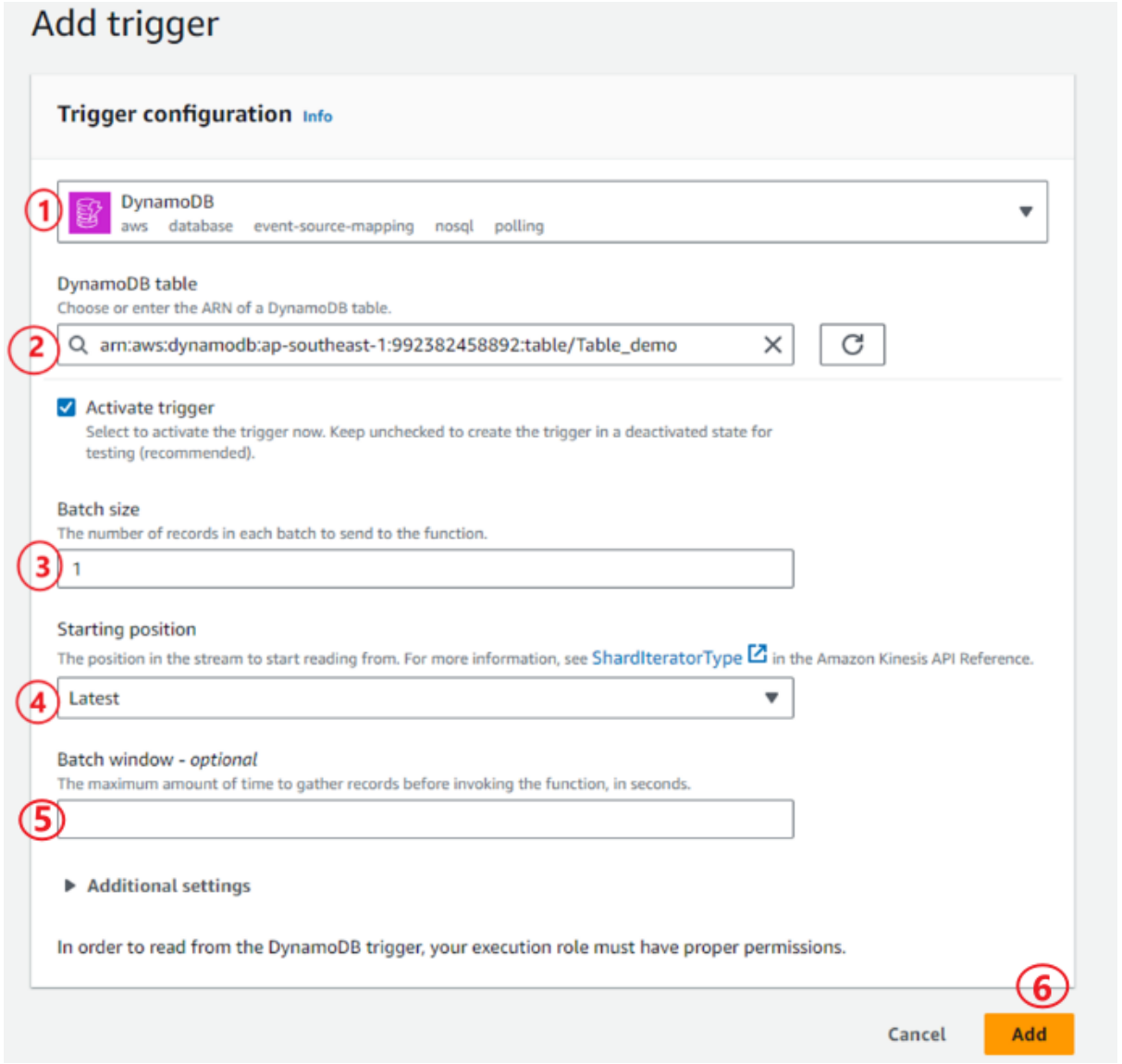
0. Choose ***Author from scratch***.
1. Enter ***lambda_demo*** as the function name.
2. Select ***Python 3.12*** for the Runtime.



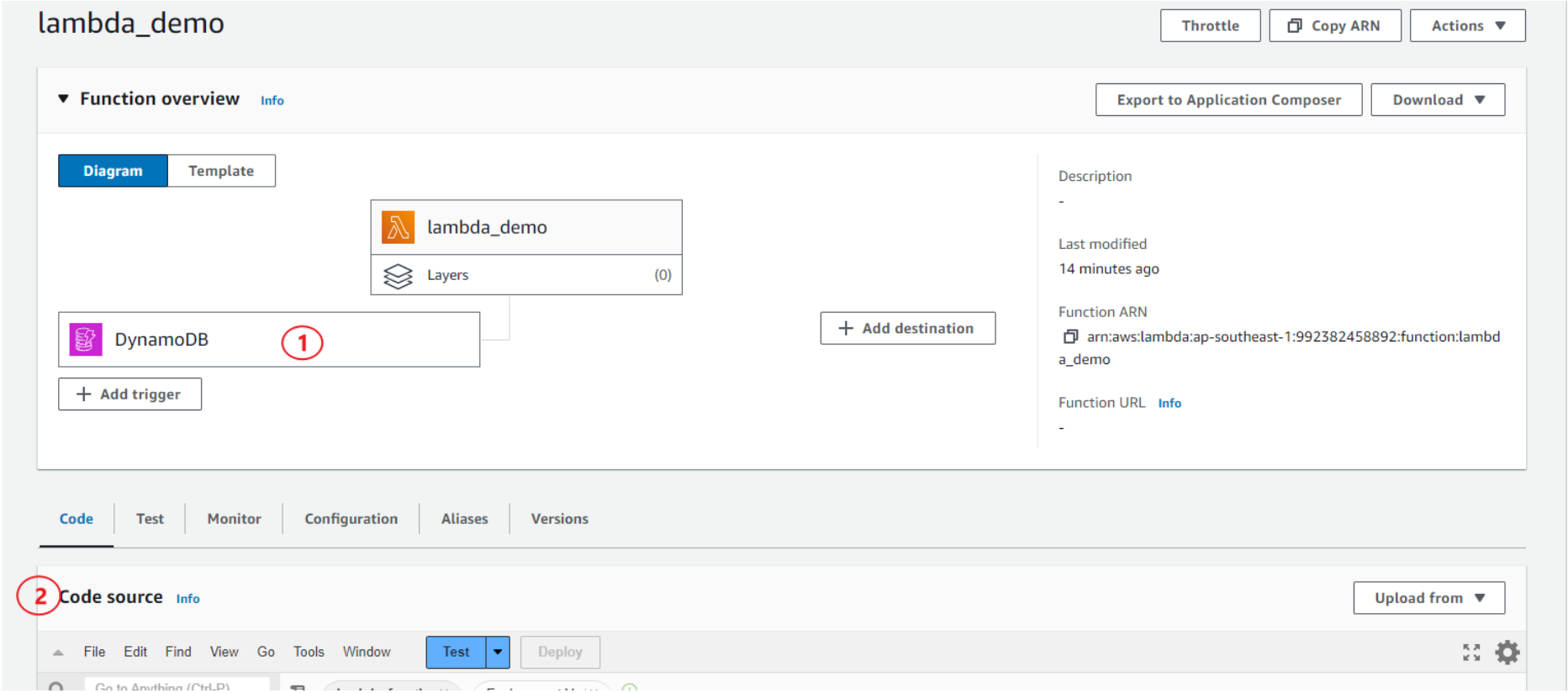
After the function has been successfully created, click on ‘Add trigger’.



1. Select ***DynamoDB*** as the source.
2. Choose the ***Table_demo*** that you previously created.
3. Enter ***1*** for the batch size, meaning that the Lambda function will be triggered every time a single new record is inserted.
4. Select ***Latest*** for the starting position. This means that the Lambda function will process only the records added after the function is created.
5. Leave the Batch window blank.
6. Click ***Add***


Add trigger
Trigger configuration Info
1
DynamoDB
aws database event-source-mapping nosql polling
DynamoDB table
Choose or enter the ARN of a DynamoDB table.
2
arn:aws:dynamodb:ap-southeast-1:992382458892:table/Table_demo
Activate trigger
Select to activate the trigger now. Keep unchecked to create the trigger in a deactivated state for testing (recommended).
Batch size
The number of records in each batch to send to the function.
3
1
Starting position
The position in the stream to start reading from. For more information, see ShardIteratorType in the Amazon Kinesis API Reference.
4
Latest
Batch window - optional
The maximum amount of time to gather records before invoking the function, in seconds.
5
Additional settings
In order to read from the DynamoDB trigger, your execution role must have proper permissions.
6
Cancel
Add

1. The trigger ***Table_demo*** was successfully added to function ***lambda_demo***.
2. Scroll down to the ***code source*** section to edit the code.



Enter the following code into the lambda_function.py

```
import json
import boto3
import logging
import csv
import io

# Initialize logger
logger = logging.getLogger()
logger.setLevel(logging.INFO)

# Initialize AWS IoT Data Client
iot_client = boto3.client('iot-data', region_name='ap-southeast-1')

# Initialize S3 Client
s3_client = boto3.client('s3')
bucket_name = ‘yourbucketname'
csv_key = 'data/water_usage.csv'
```

```
def lambda_handler(event, context):
    logger.info("Received event: %s", json.dumps(event))

    # Check if 'Records' key is in the event
    if 'Records' not in event:
        logger.error("Invalid event structure: 'Records' key missing")
        return {
            'statusCode': 400,
            'body': json.dumps('Invalid event structure')
        }

    for record in event['Records']:
        logger.info("Processing record: %s", json.dumps(record))
```

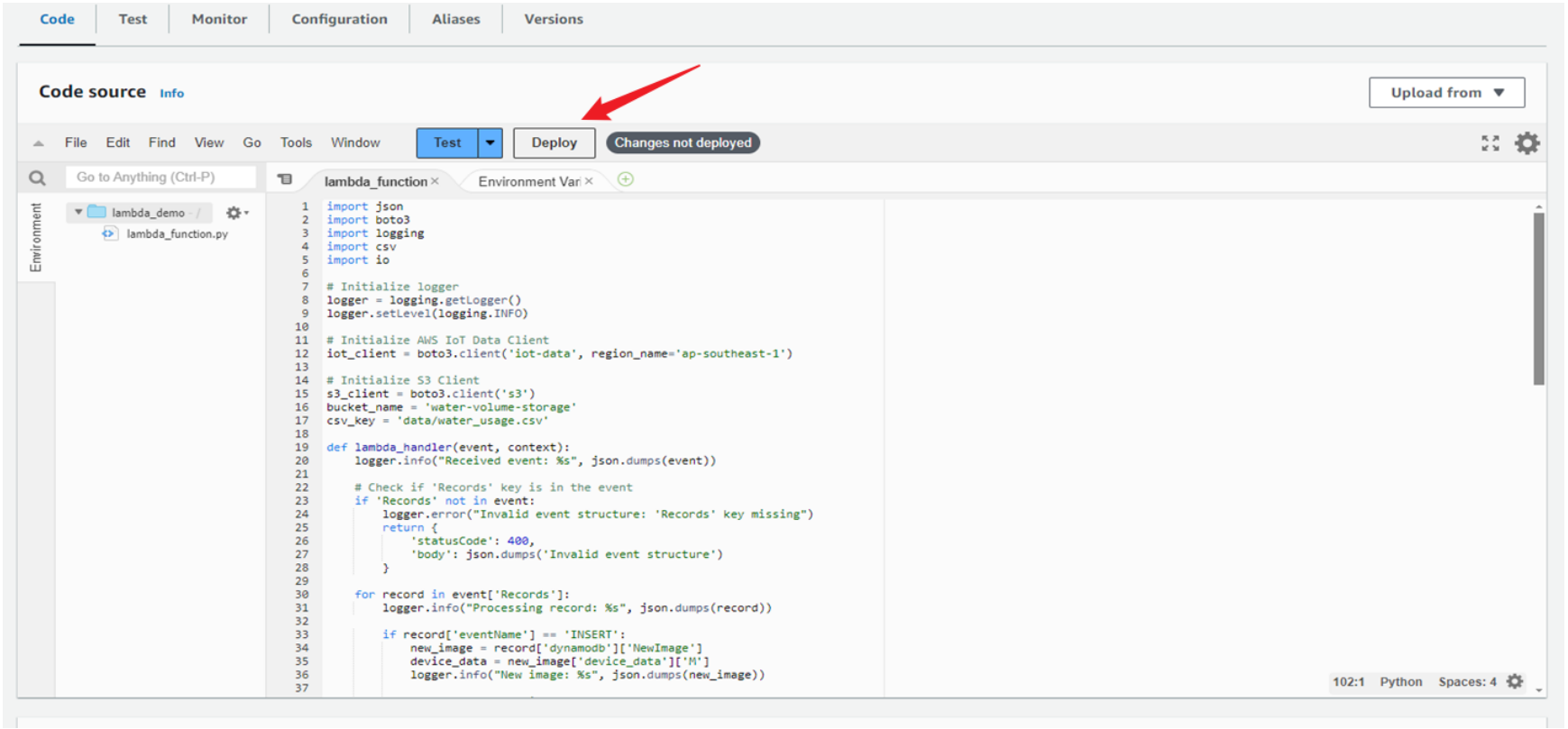
```
    if record['eventName'] == 'INSERT':
        new_image = record['dynamodb']['NewImage']
        device_data = new_image['device_data']['M']
        logger.info("New image: %s", json.dumps(new_image))

        # Extract the environmental data
        humidity = float(device_data['humidity']['N'])
        temperature = float(device_data['temperature']['N'])
        wind_speed = float(device_data['wind_speed']['N'])
        soil_moisture = float(device_data['soil_moister_level']['N'])
        location_id = device_data['location_id']['S']
        date_time = device_data['date_time']['S']
        # Calculate the water volume (example calculation)
        water_volume = calculate_water_volume(humidity,
temperature, wind_speed, soil_moisture)
        logger.info("Calculated water volume: %s", water_volume)
```

```
        # Prepare the new row to append to the CSV
        new_row = [humidity, temperature,
wind_speed, soil_moisture, location_id, date_time,
water_volume]

        # Read the existing CSV from S3
        try:
            csv_obj =
s3_client.get_object(Bucket=bucket_name,
Key=csv_key)
            csv_body =
csv_obj['Body'].read().decode('utf-8')
            csv_reader =
csv.reader(io.StringIO(csv_body))
            csv_data = list(csv_reader)
        except s3_client.exceptions.NoSuchKey:
```

```
        # If the CSV file doesn't exist, create a new one with headers
        csv_data = [["humidity", "temperature", "wind_speed", "soil_moisture", "location_id", "date_time", "water_volume"]]

    # Append the new row to the CSV data
    csv_data.append(new_row)

    # Convert the CSV data back to string
    csv_buffer = io.StringIO()
    csv_writer = csv.writer(csv_buffer)
    csv_writer.writerows(csv_data)
    csv_str = csv_buffer.getvalue()

    # Store the updated CSV back to S3
    s3_client.put_object(
      Bucket=bucket_name,
      Key=csv_key,
      Body=csv_str
    )
    logger.info("Updated CSV stored in S3: %s", csv_key)

    # Publish the water volume to the IoT topic
    topic = 'env/volume'
    message = {
      'water_volume': water_volume,
      'location_id': location_id,
      'date_time': date_time
    }
    response = iot_client.publish(
      topic=topic,
      qos=1,
      payload=json.dumps(message)
    )
    logger.info("Publish response: %s", response)

  return {
    'message': 'Success'
  }
```

```
def calculate_water_volume(humidity, temperature, wind_speed, soil_moisture):
    # Example calculation logic
    water_volume = (100 - soil_moisture) * (1 + humidity / 100) * (1 - wind_speed / 10) * temperature / 30
    return water_volume
```

This code is a Lambda function that processes environmental data inserted into a DynamoDB table, calculates water volume, updates a CSV file in an S3 bucket, and publishes the result to an AWS IoT topic. Click the ***Deploy*** button to save changes and deploy the code.

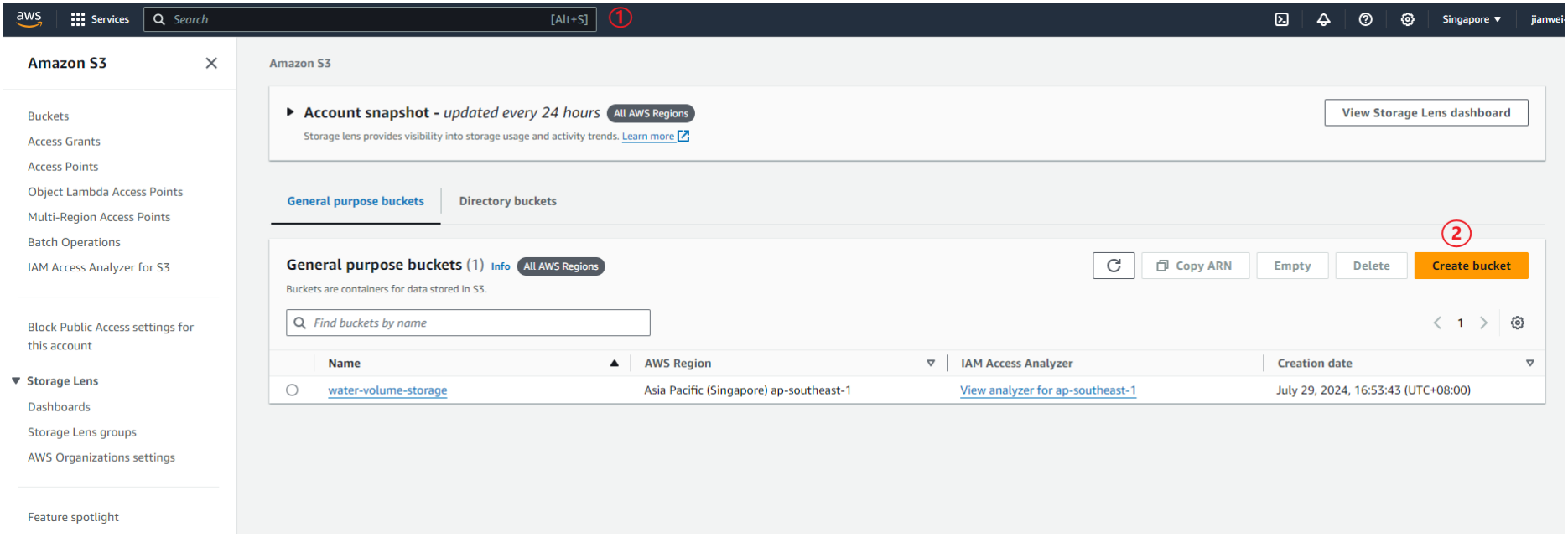


At this point, we can test the SDK program created in the previous step. If, upon running the program, you receive a reply with the water volume, it means all the services are working correctly. Congratulations!

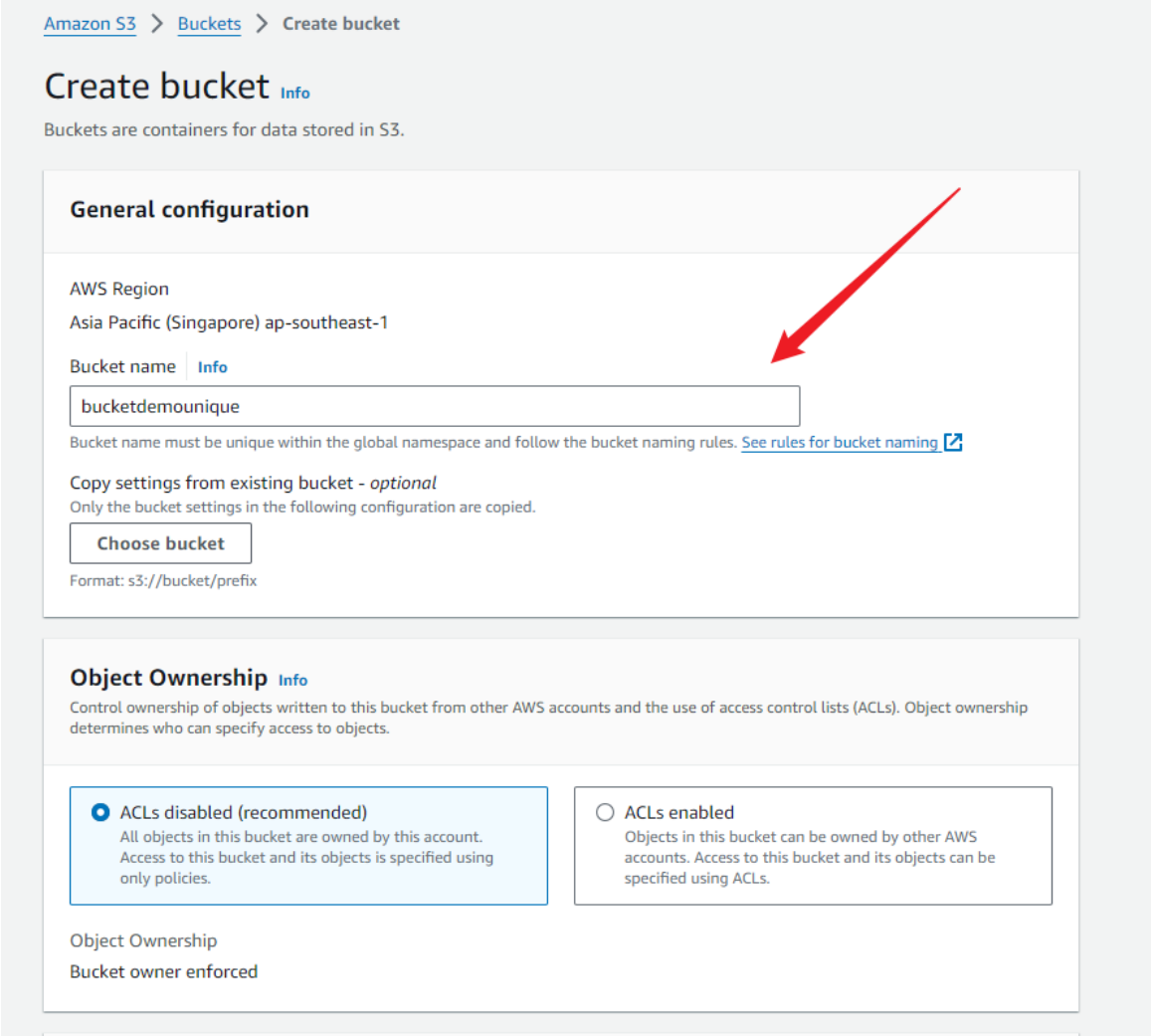
```
C:\Users\Oct23\OneDrive - Universiti Tunku Abdul Rahman\Degree\Y3S2\UCCD3113 Distributed Computer System\Assignment\cert>test.py
Connecting to a2dfazm7rcnmac-ats.iot.ap-southeast-1.amazonaws.com with client ID 'b60cbaa7-b545-4b18-b52d-1c74d0b1bc7e'...
Connected!
Subscribing to topic 'env/volume'...
Subscribed with QoS 1
Published message to topic 'device/1/data': {"location_id": "region_1", "soil_moister_level": 65, "temperature": 15, "humidity": 80, "wind_speed": 3, "date_
time": "28-08-2024 21:16:41"}
Received message from topic 'env/volume': b'{"water_volume": 22.049999999999997, "location_id": "region_1", "date_time": "28-08-2024 21:16:41"}'
All replies received
Disconnecting...
Disconnected! Program complete.
```

Amazon S3

Now, we need to create a bucket to store the water usage volume. First, navigate to the S3 service by typing '***S3***' in the search bar. Then, click on ***Create bucket***.



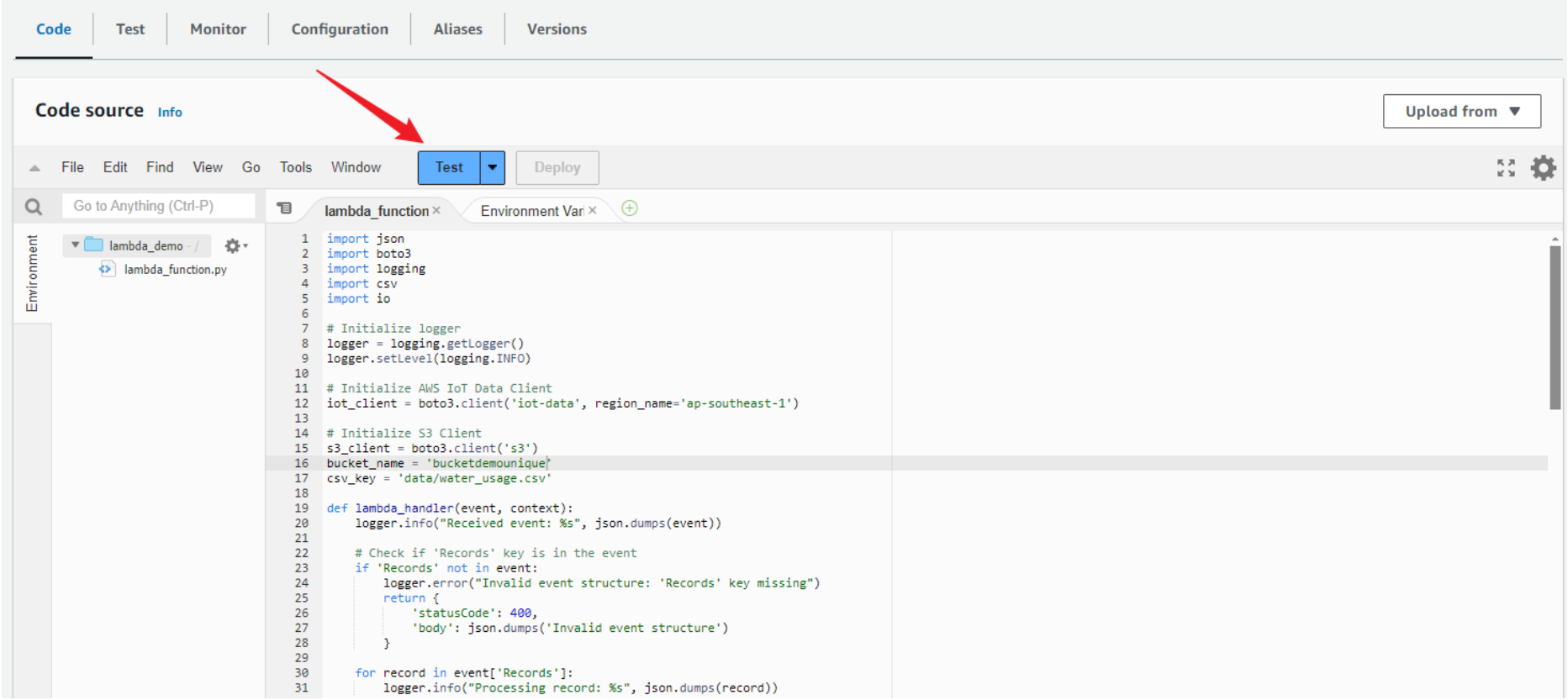
Enter ***bucketdemounique*** as the bucket name, leave the remaining settings as default, then scroll down to the bottom of the page and click ***Create bucket***.



After creating the bucket, you can now upload files to it. Return to the Lambda function you previously created and replace the bucket name with your newly created ***'bucketdemounique'*** bucket.

```
# Initialize S3 Client

s3_client = boto3.client('s3')

bucket_name = '***bucketdemounique'***

csv_key = 'data/water_usage.csv'
```

To make sure everything is working correctly, go back to the lambda function and click on ***Test*** button.

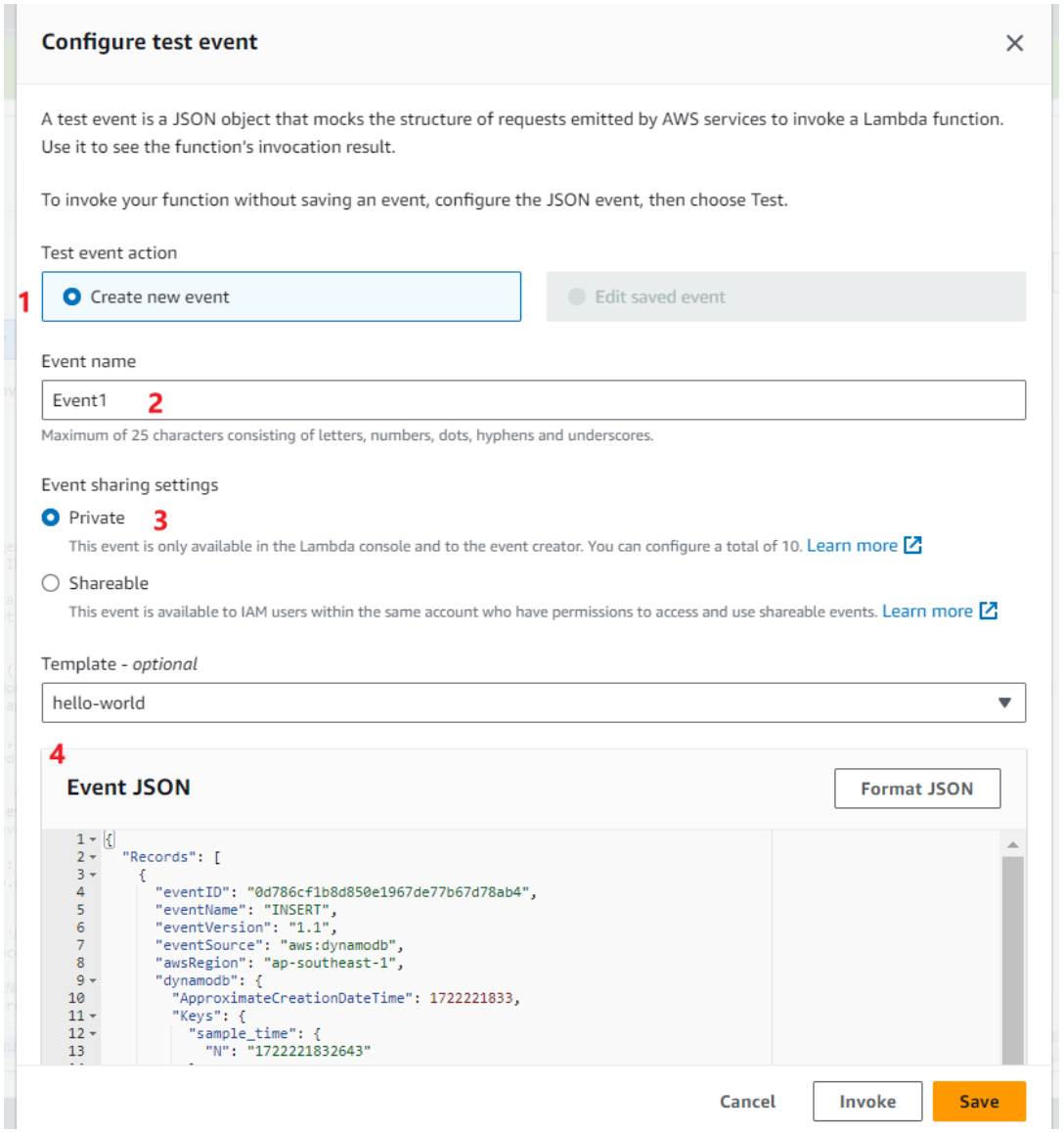


1. Select ‘Create new event’.
2. Enter ‘Event1’ as the event name.
3. Select ‘Private’ for the event sharing settings.
4. Enter the following event JSON:


```
{
 "Records": [
  {
   "eventID": "0d786cf1b8d850e1967de77b67d78ab4",
   "eventName": "INSERT",
   "eventVersion": "1.1",
   "eventSource": "aws:dynamodb",
   "awsRegion": "ap-southeast-1",
   "dynamodb": {
    "ApproximateCreationDateTime": 1722221833,
    "Keys": {
     "sample_time": {
      "N": "1722221832643"
      "NewImage": {
       "sample_time": {
        "N": "1722221832643"
       },
       "device_data": {
        "M": {
         "soil_moister_level": {
          "N": "34"
         },
         "date_time": {
          "S": "29-07-2024 10:57:10"
         },
```

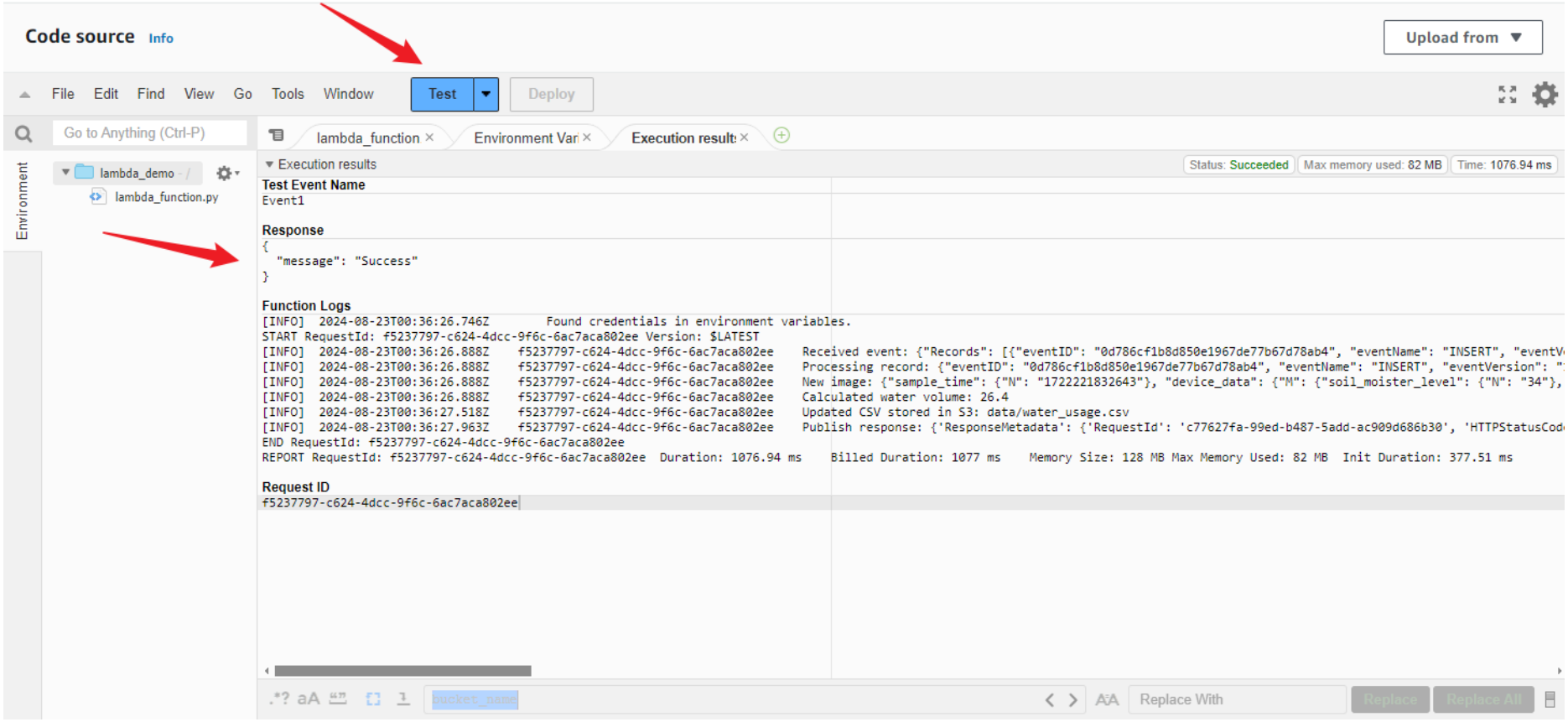
```
      "temperature": {
       "N": "12"
      },
      "location_id": {
       "S": "region_2"
      }
     }
    },
    "device_id": {
     "N": "1"
    }
   },
      "humidity": {
       "N": "100"
      },
      "wind_speed": {
       "N": "5"
      },
```

```
      "SequenceNumber": "841320000000000036341113576",
      "SizeBytes": 182,
      "StreamViewType": "NEW_AND_OLD_IMAGES"
     },
     "eventSourceARN": "arn:aws:dynamodb:ap-southeast-1:992382458892:table/environmental_data/stream/2024-07-28T09:02:46.192"
    }
  ]
}
```

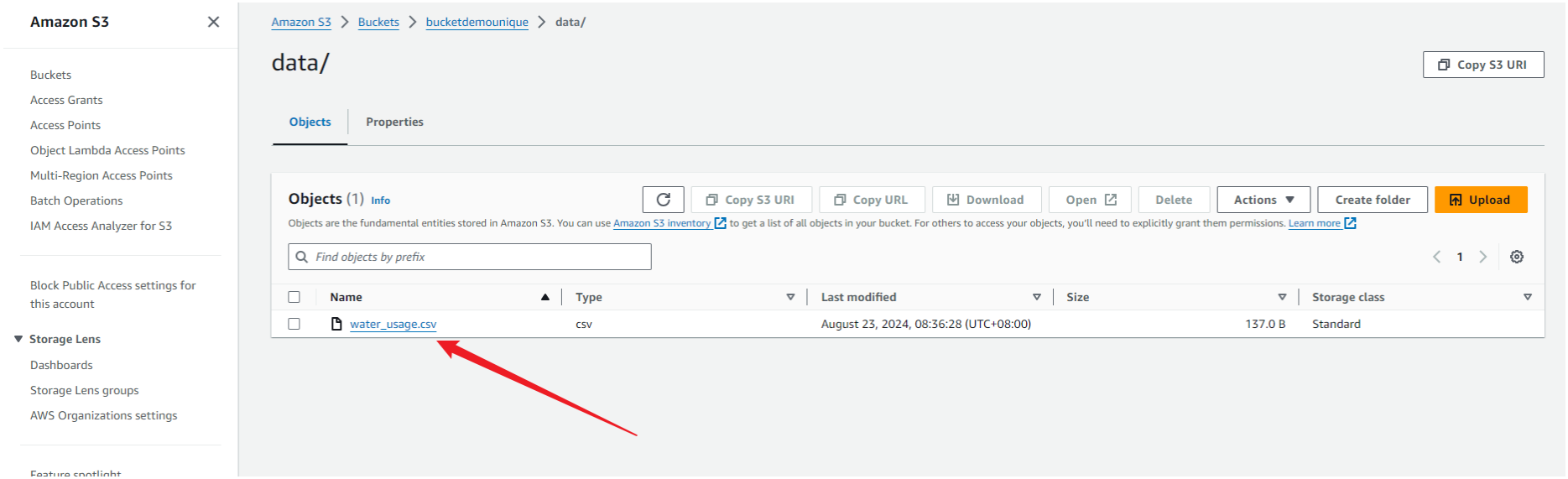


Click on ***Save*** button.

Click on Test button and you should see a response with “Success” message.

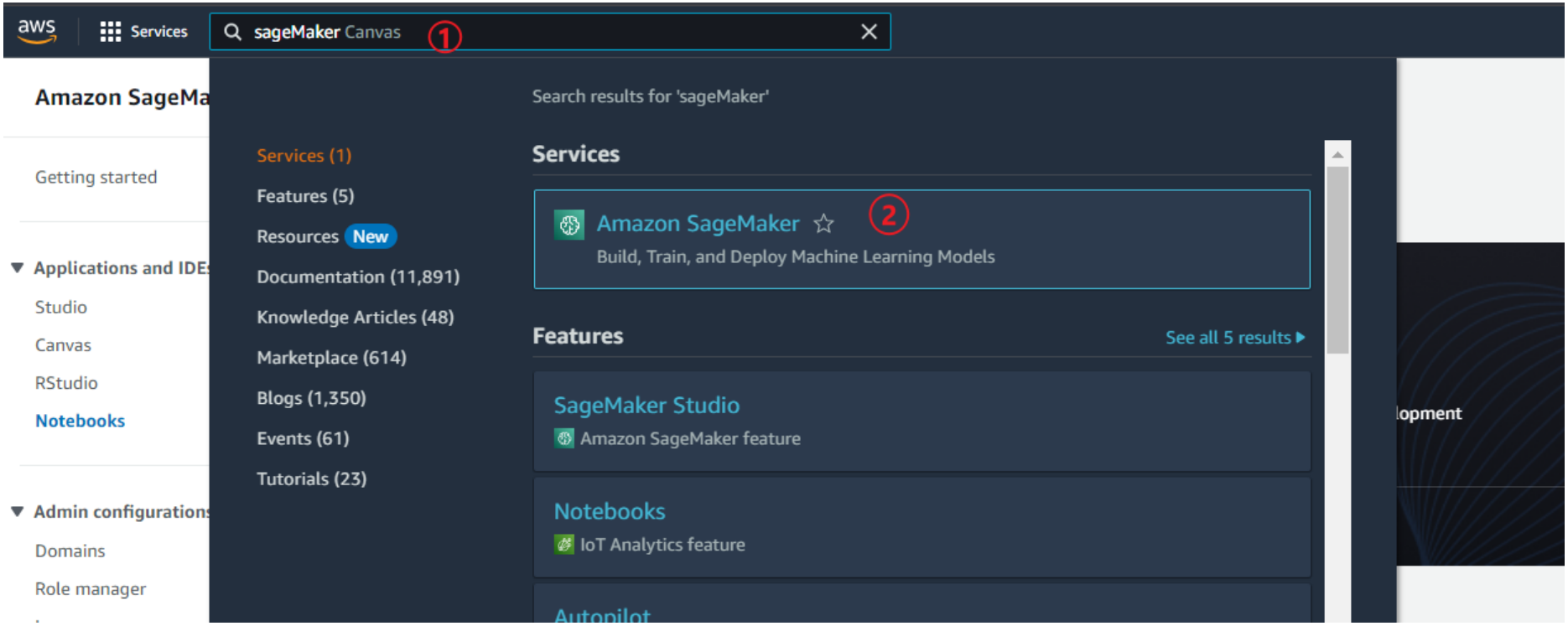


Go back to your ***bucketdemounique*** bucket. You should see that a new ***water_usage.csv*** file has been inserted into the bucket.

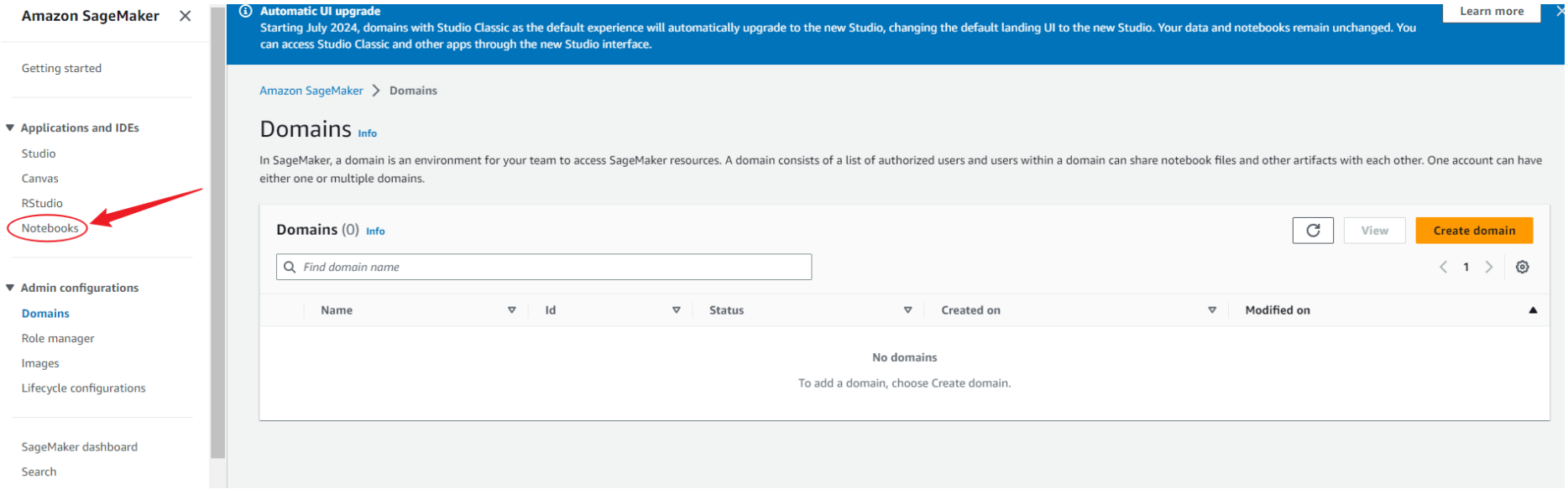


Amazon SageMaker

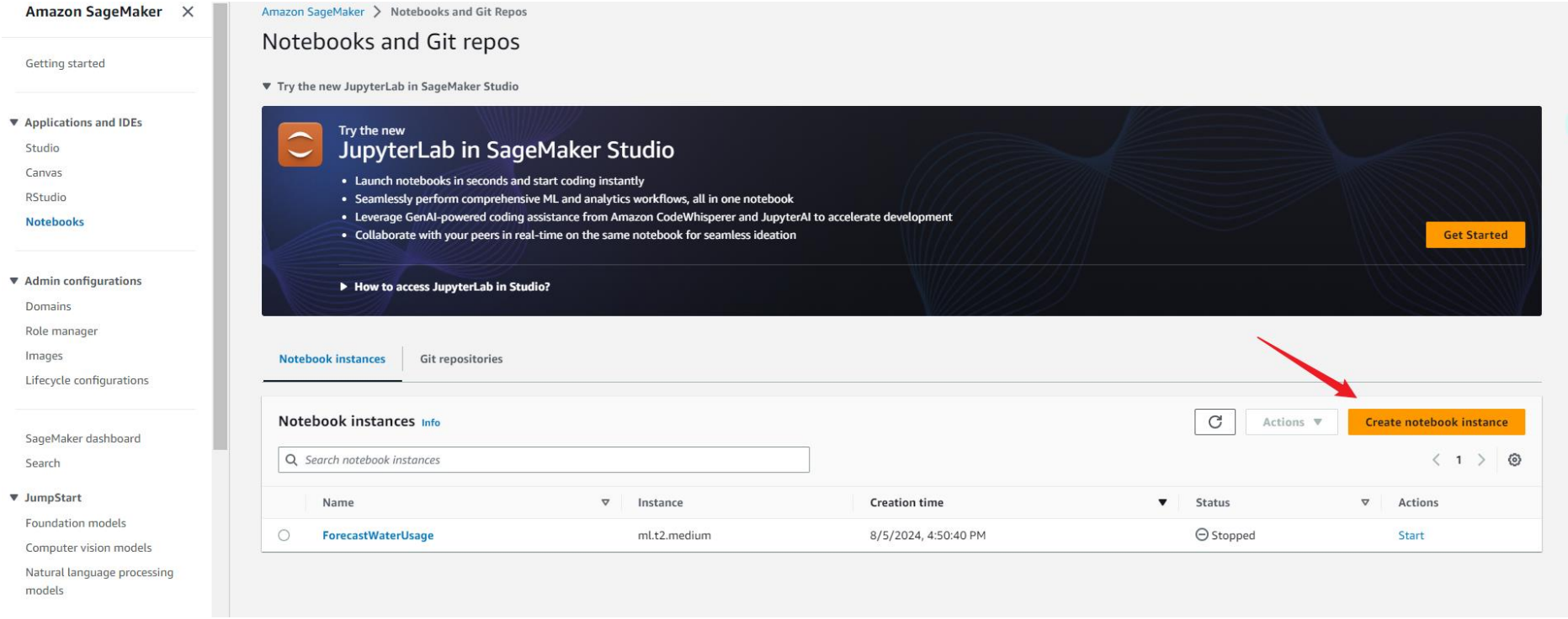
With all the data stored in the S3 bucket, we can use it to forecast future water usage demand by using Amazon SageMaker, a fully managed machine learning (ML) service where you can build, train, and deploy ML models at scale. To use the service, type ***SageMaker*** in search bar and click the ***Amazon SageMaker*** service.



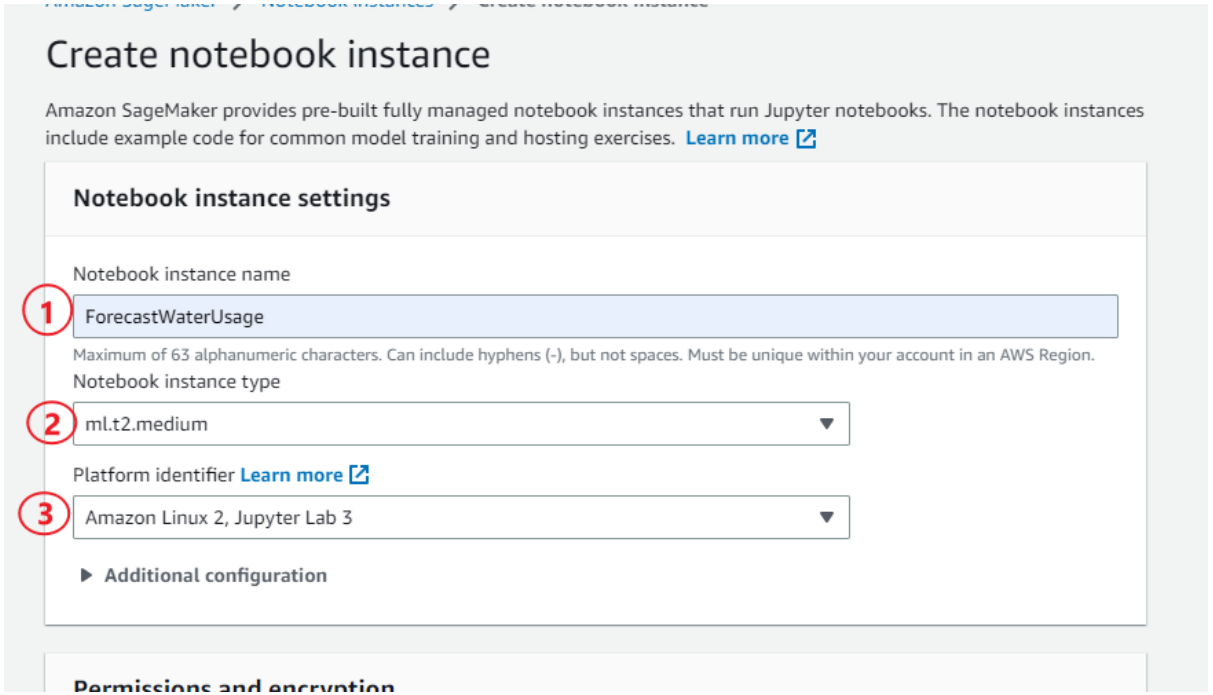
Click ***notebooks*** from the navigation bar.



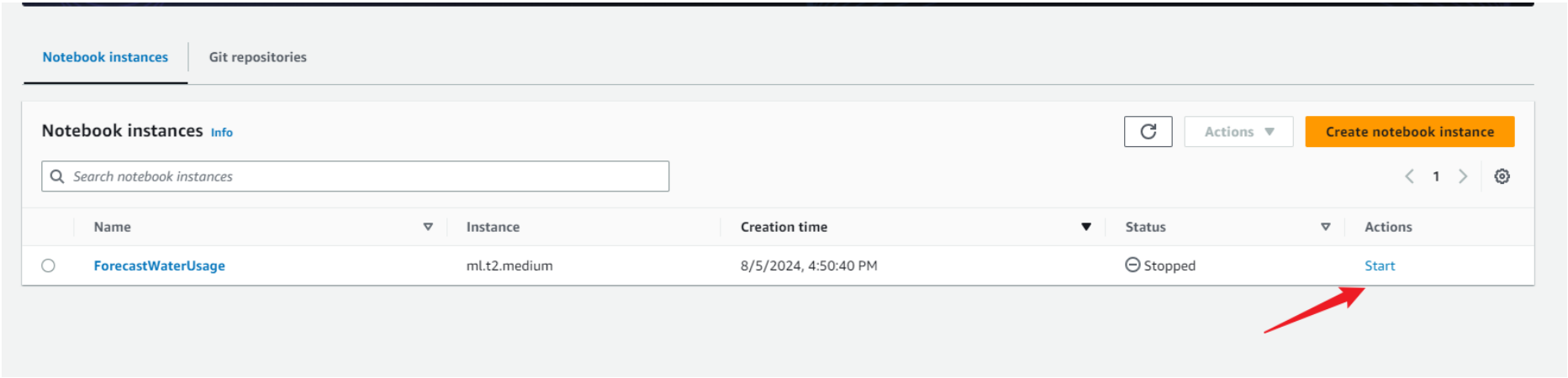
Click ***Create notebook instance.***



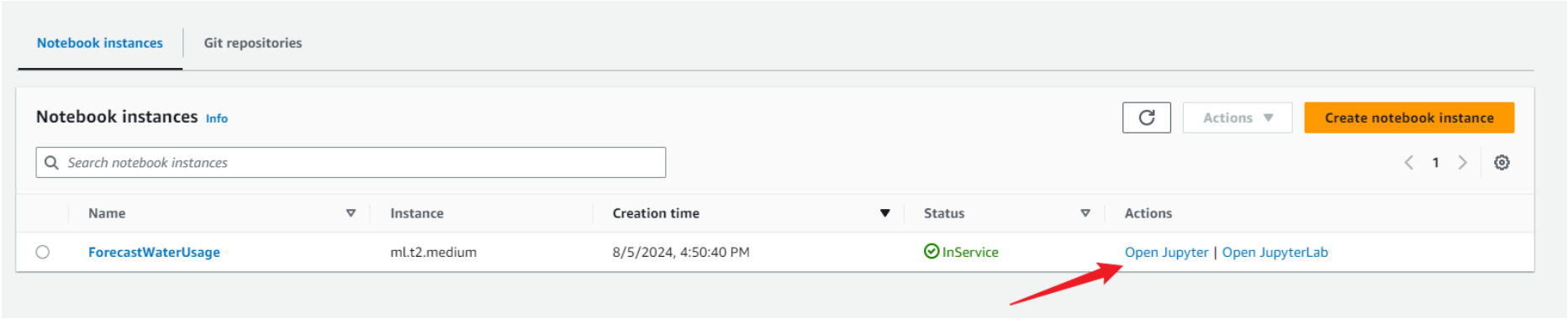
1. Enter ***'ForecastWaterUsage'*** as notebook instance name.
2. Select ***'ml.t2.medium'*** as notebook instance type.
3. Select ***'Amazon Linux 2, Jupyter Lab 3'*** as platform identifier.
4. Leave other settings as default, scroll down to the bottom of the page, and click ***Create notebook instance***.



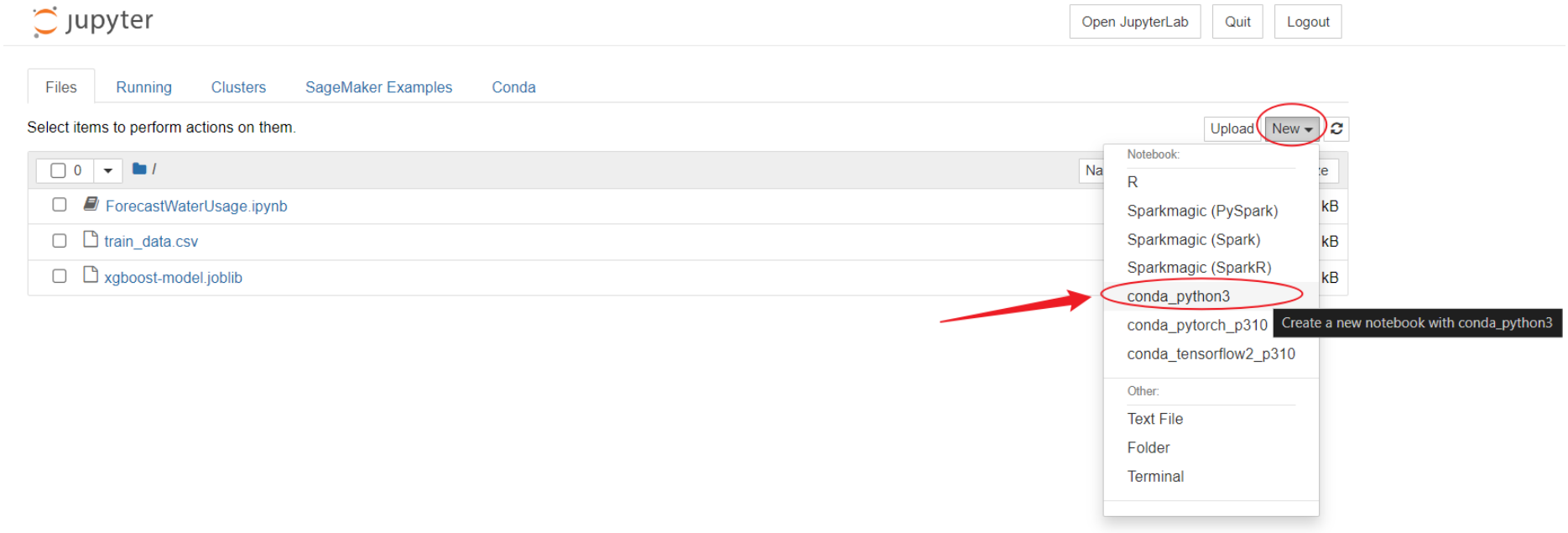
Once your notebook has been created, click on ***Start***.



Click ***Open Jupyter.***



You will be redirected to a new Jupyter page. Click on ***New*** and select ***conda_python3.***



Now, we want to write a script to reads historical water usage data from S3, trains an ARIMA model to predict future usage, generates a forecast CSV, uploads it to S3, and visualizes the results. Enter the following code into your notebook:

Import all the necessary libraries.

```
import pandas as pd
from statsmodels.tsa.arima.model import ARIMA
import matplotlib.pyplot as plt
import boto3
import io
```

Specifies the S3 bucket and the location of the historical water usage data (water_usage.csv) and reads the csv data from S3 into a pandas DataFrame.

```
bucket = 'water-volume-storage'

data_key = 'data/water_usage.csv'

data_location = f's3://{bucket}/{data_key}'


# Load the data

df = pd.read_csv(data_location, parse_dates=['date_time'], index_col='date_time', dayfirst=True)


# Ensure water_volume is numeric

df['water_volume'] = pd.to_numeric(df['water_volume'], errors='coerce')


# Convert index to DatetimeIndex, allowing for mixed formats

if not isinstance(df.index, pd.DatetimeIndex):

   df.index = pd.to_datetime(df.index, format='mixed', dayfirst=True)


# Aggregate data (assuming daily water volume usage)

df = df.resample('D').sum().interpolate(method='linear')
```

Trains the ARIMA model on the historical water_volume data.

```
# Train the ARIMA model
model = ARIMA(df['water_volume'], order=(5, 1, 0))  # Example order, can be optimized
model_fit = model.fit()
# Forecast future water usage
forecast_steps = 30  # Predict for the next 30 days
forecast = model_fit.forecast(steps=forecast_steps)
# Prepare the forecast data for CSV
start_date = df.index[-1].strftime('%Y-%m-%d')
end_date = pd.date_range(df.index[-1], periods=forecast_steps + 1, freq='D')[-1].strftime('%Y-%m-%d')
forecast_filename = f'forecast_total_water_usage_{start_date}_to_{end_date}.csv'
# Create a DataFrame for the forecast
forecast_dates = pd.date_range(df.index[-1], periods=forecast_steps + 1, freq='D')[1:]
forecast_df = pd.DataFrame({'date': forecast_dates, 'forecasted_water_volume': forecast})
total_forecasted_usage = forecast.sum()
```

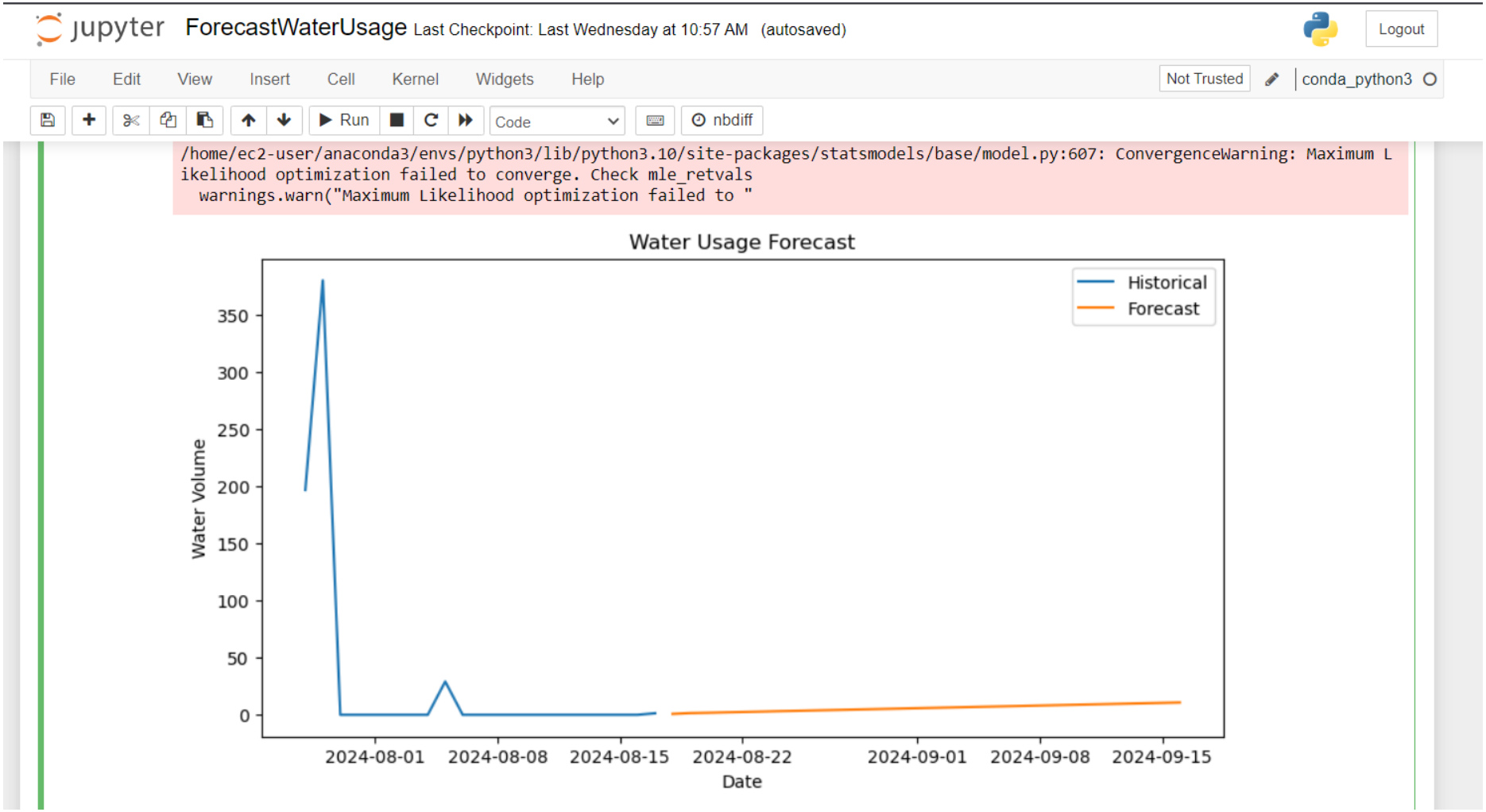
Uploads the forecast CSV to the S3 bucket in a forecast directory.

```
# Add total forecasted usage as a final row

total_row = pd.DataFrame({'date': ['Total'], 'forecasted_water_volume':
[total_forecasted_usage]})

forecast_df = pd.concat([forecast_df, total_row])


# Convert DataFrame to CSV format in memory

csv_buffer = io.StringIO()

forecast_df.to_csv(csv_buffer, index=False)


# Upload the CSV file to S3

s3_client = boto3.client('s3')

s3_client.put_object(Body=csv_buffer.getvalue(), Bucket=bucket,
Key=f'forecast/{forecast_filename}')
```

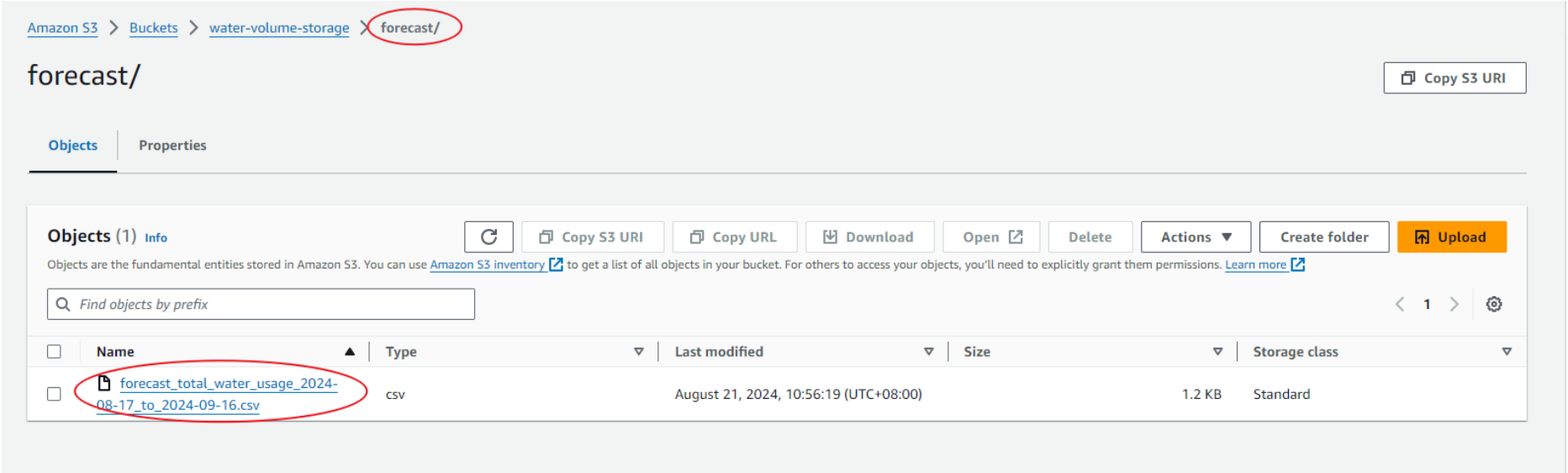
Plots the historical water usage and the forecasted values on a graph.

```
# Plot the results
plt.figure(figsize=(10, 5))
plt.plot(df.index, df['water_volume'], label='Historical')
plt.plot(forecast_dates, forecast, label='Forecast')
plt.xlabel('Date')
plt.ylabel('Water Volume')
plt.title('Water Usage Forecast')
plt.legend()
plt.show()
```

After running the code in the notebook, you should see the forecasted graph.



Go to your S3 service, and you should see a new file named ***'forecast_total_water_usage_2024-08-17_to_2024-09-16.csv'*** under the '***forecast***' directory in your bucket. The forecasted water usage demand is stored in this file.



Here is an example of forecasted water usage data.

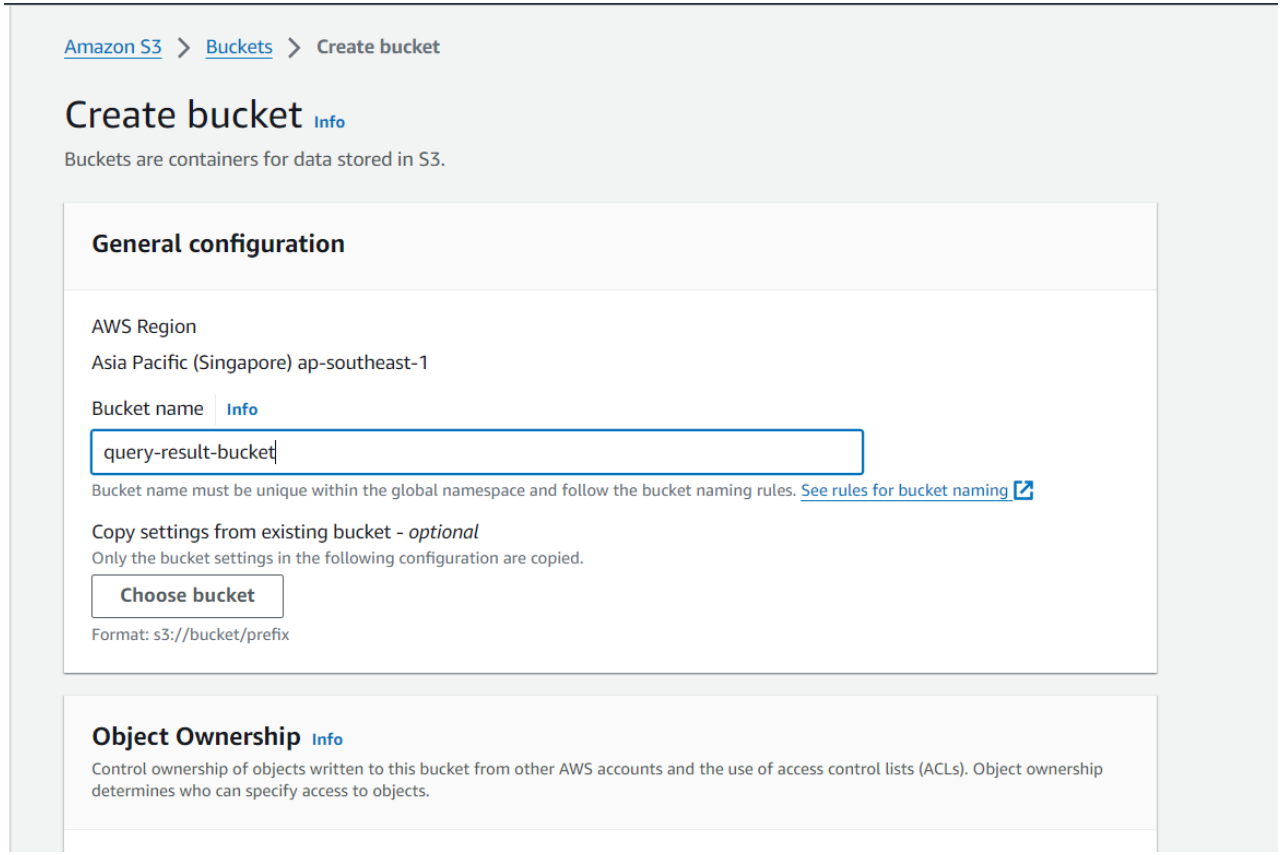
date	forecasted_water_volume
24/8/2024 0:00	3.145479899
25/8/2024 0:00	3.446850713
26/8/2024 0:00	3.793579094
27/8/2024 0:00	4.122465724
28/8/2024 0:00	4.458018899
29/8/2024 0:00	4.789107497
30/8/2024 0:00	5.122080414
31/8/2024 0:00	5.456221848
1/9/2024 0:00	5.78806716
2/9/2024 0:00	6.121499969
3/9/2024 0:00	6.45420208
4/9/2024 0:00	6.787247406
5/9/2024 0:00	7.120038847
6/9/2024 0:00	7.452929982
7/9/2024 0:00	7.785832858
8/9/2024 0:00	8.118653227
9/9/2024 0:00	8.451516843
10/9/2024 0:00	8.784342091
11/9/2024 0:00	9.117173006
12/9/2024 0:00	9.44998157
13/9/2024 0:00	9.782785045
14/9/2024 0:00	10.11557771
15/9/2024 0:00	10.44835761
16/9/2024 0:00	10.78112853
Total	178.3951097

forecast_total_water_usage_2024

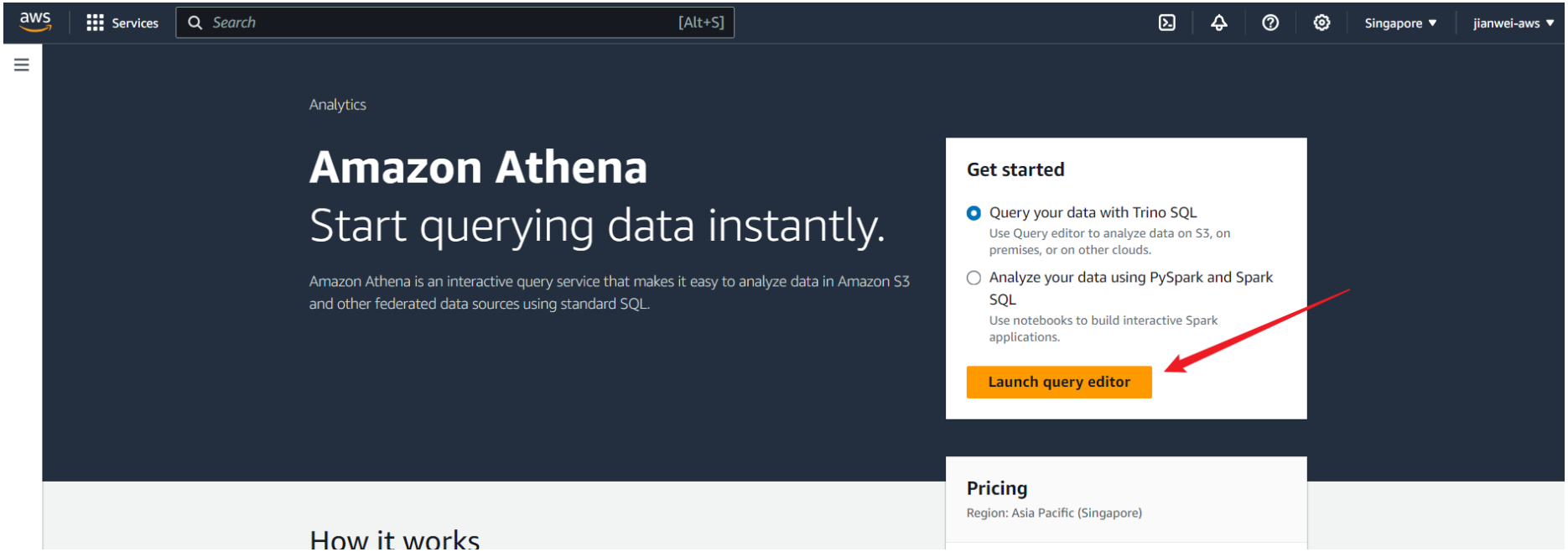
Amazon Athena

With all the data stored in S3, we can now visualize it to provide insights to stakeholders and farmers using Amazon QuickSight. Before visualizing the data, we need to preprocess it by performing SQL operations in Amazon Athena. Athena is a serverless, interactive query service that allows you to use standard SQL to query data directly from Amazon S3. Using Athena to filter or aggregate the data first helps reduce data size and complexity, resulting in faster and more efficient visualizations.

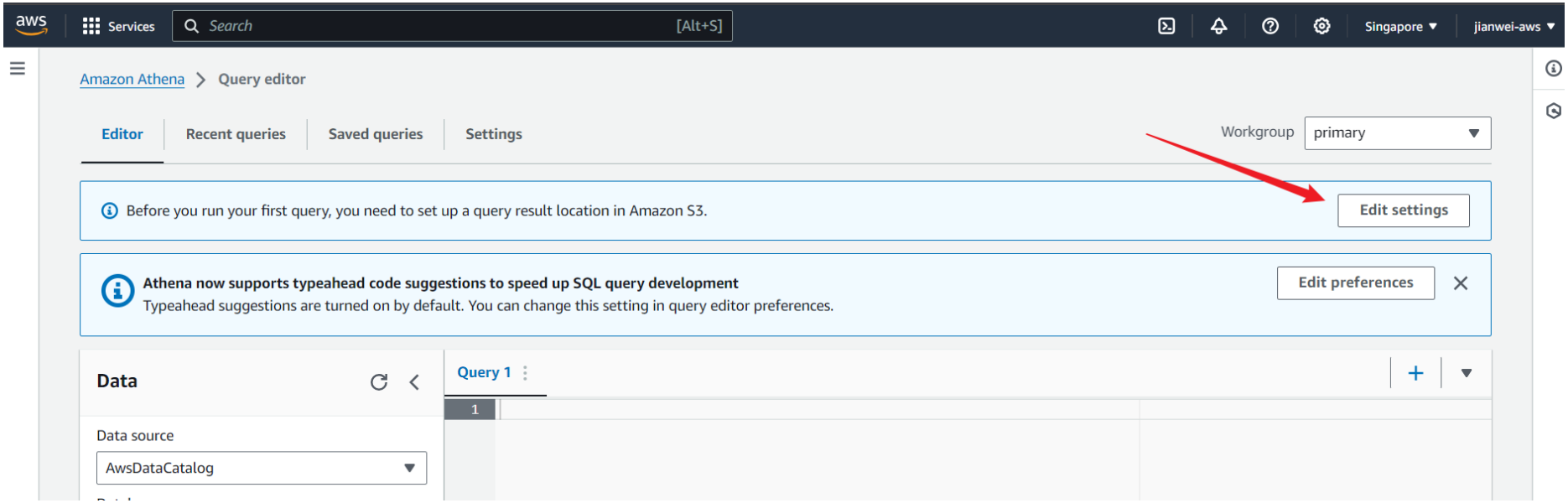
First, navigate to the S3 service and create a new bucket to store the query results. Name the bucket 'query-result-bucket' and leave the remaining settings as default.



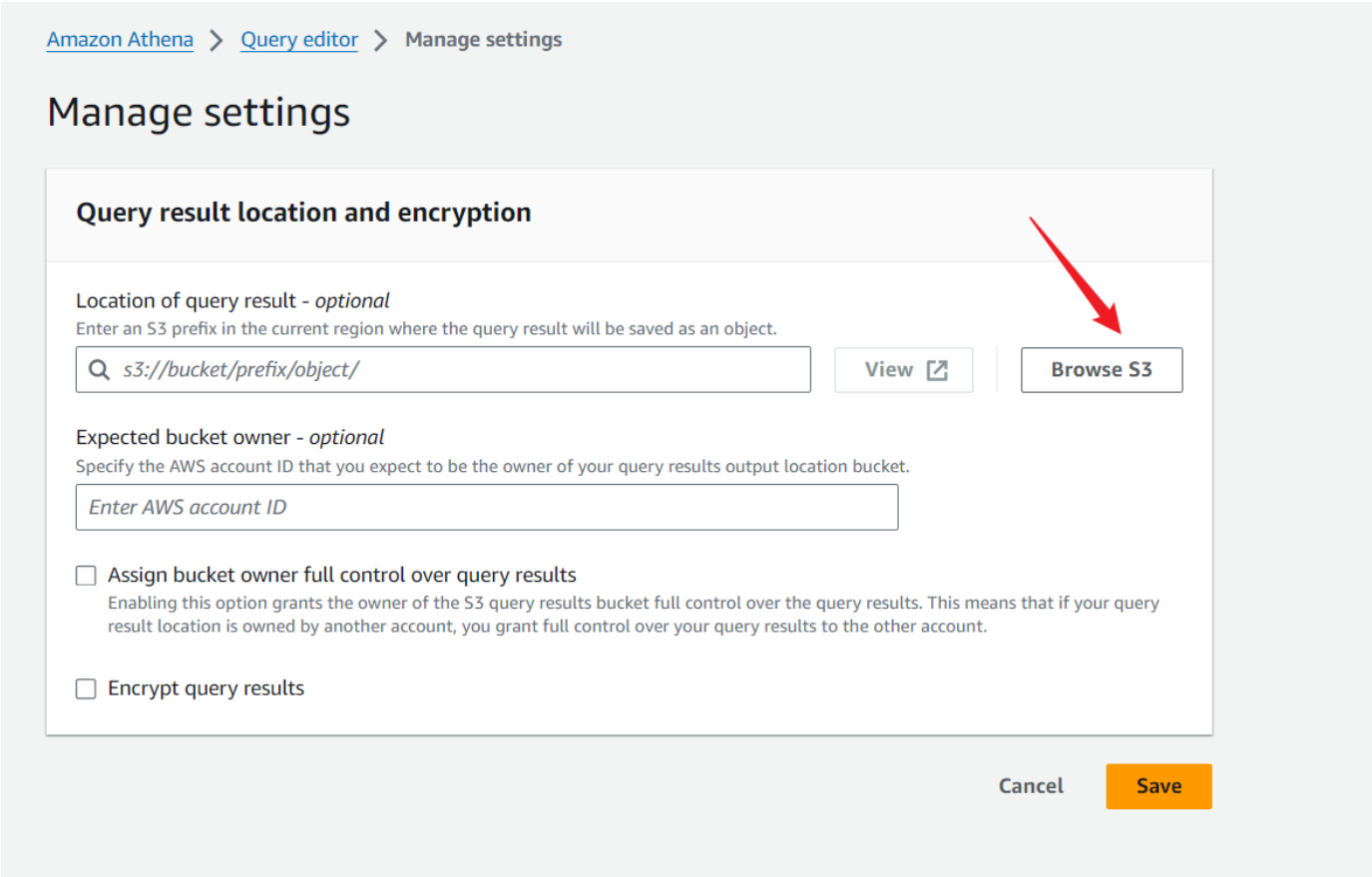
After that, type 'Athena' in the search bar and select the 'Athena' service. You will be directed to the Athena page. Choose 'Query your data in Trino SQL' since we want to analyze data in S3 and click 'Launch query editor'.



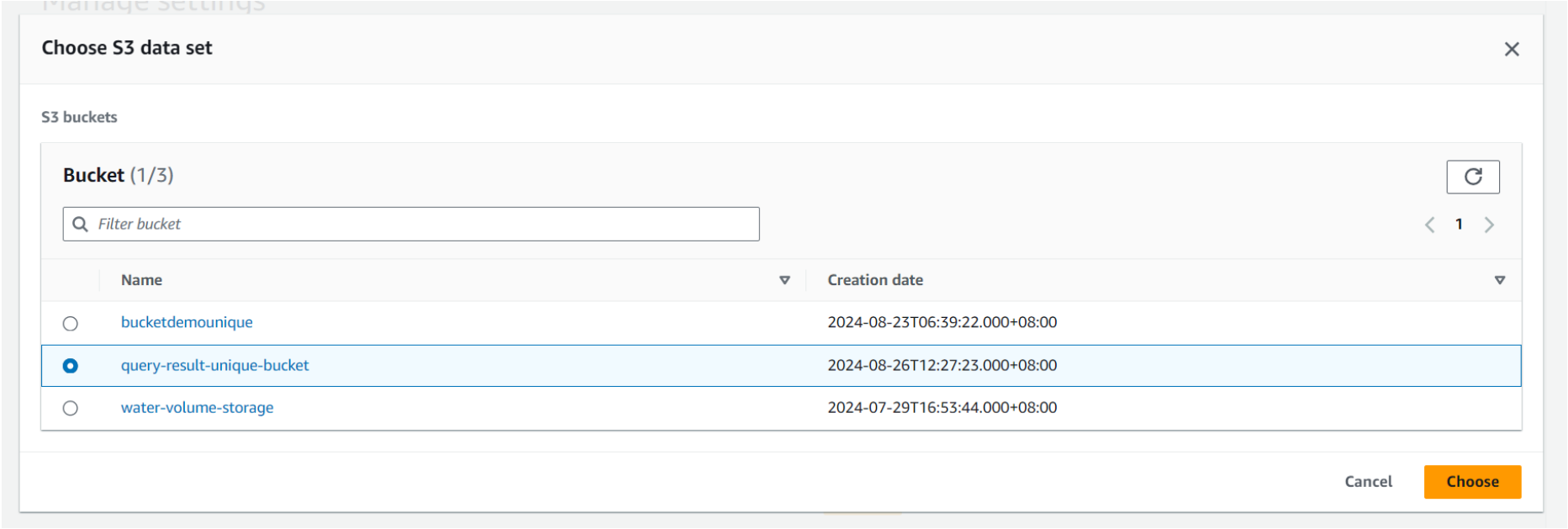
Before running your first query, you need to set up a query result location in Amazon S3. Click 'Edit settings'.



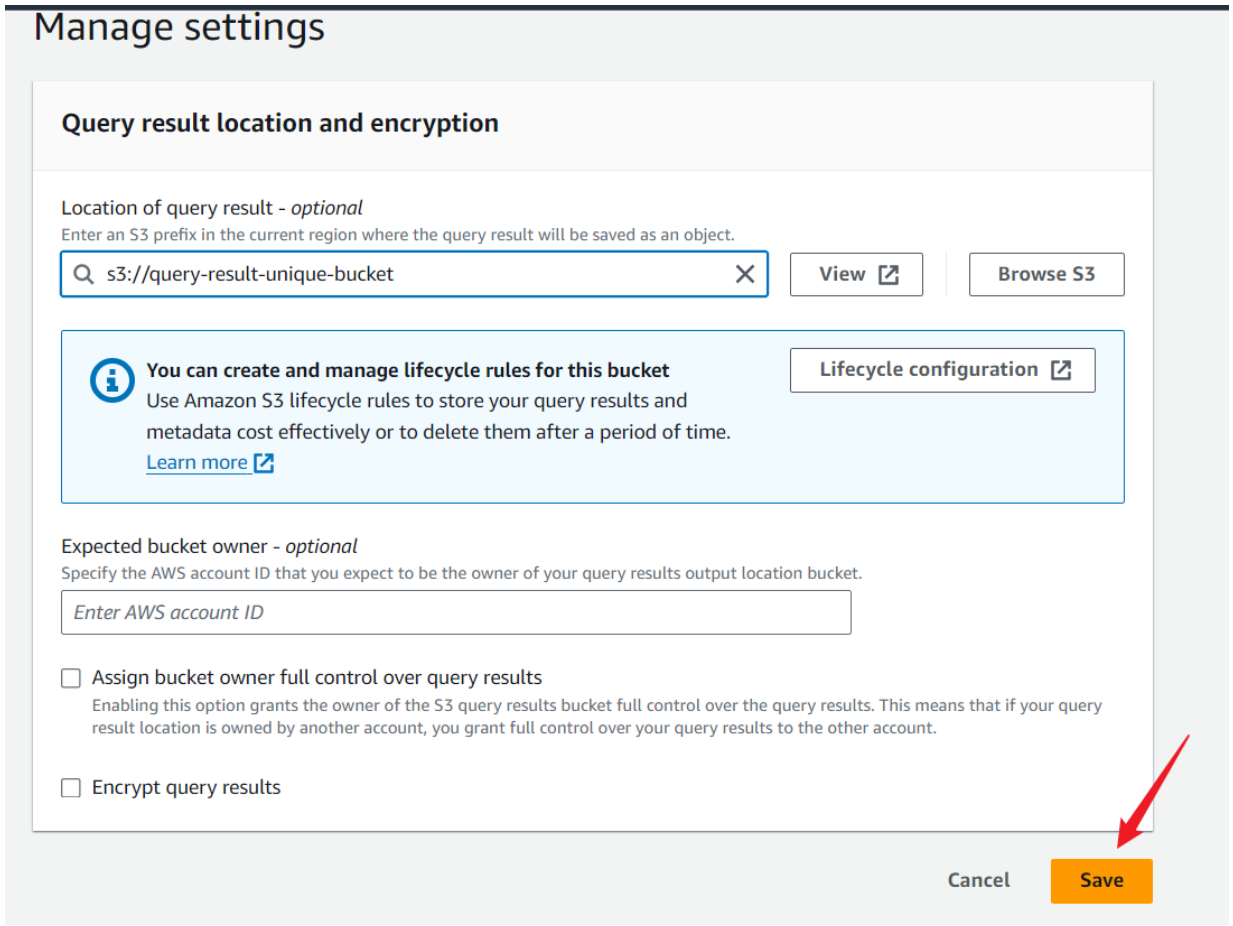
Click ‘Browse S3’ to select your bucket.



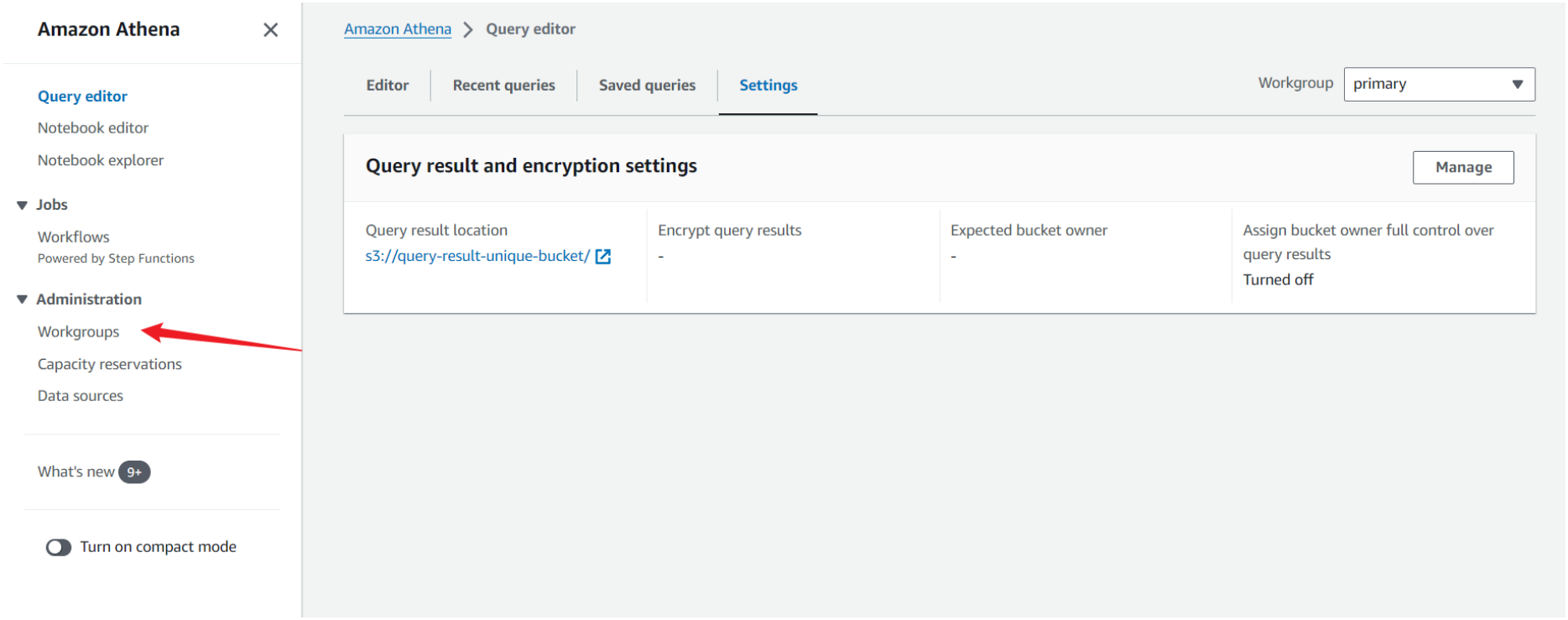
Choose the bucket that you created in previous step.



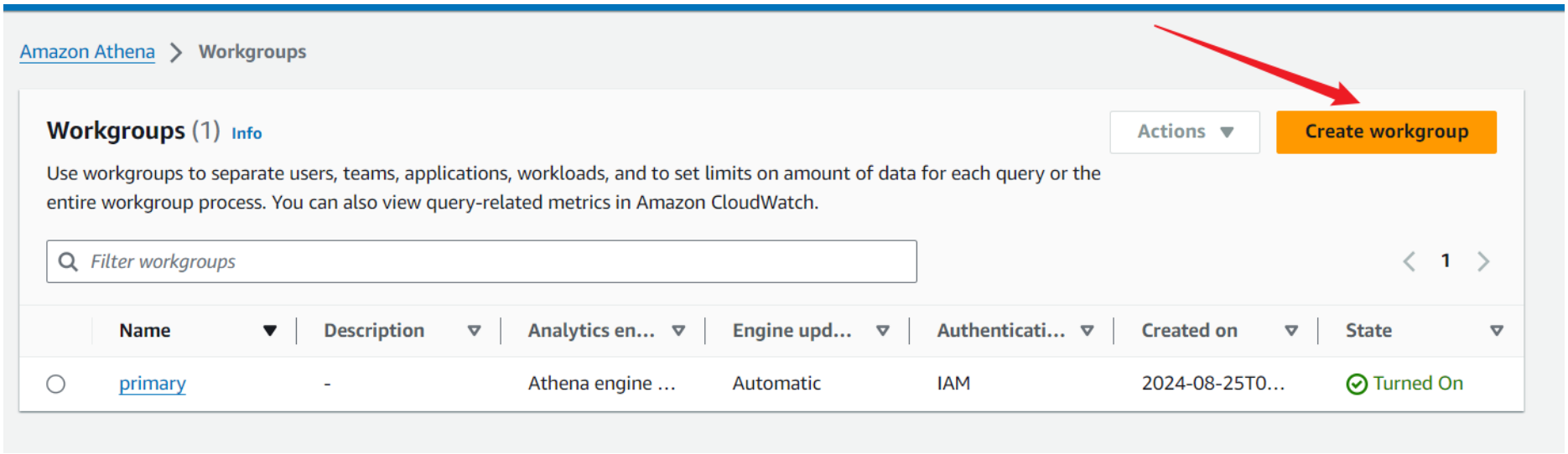
Click 'Save'. All query results will be saved to this bucket.



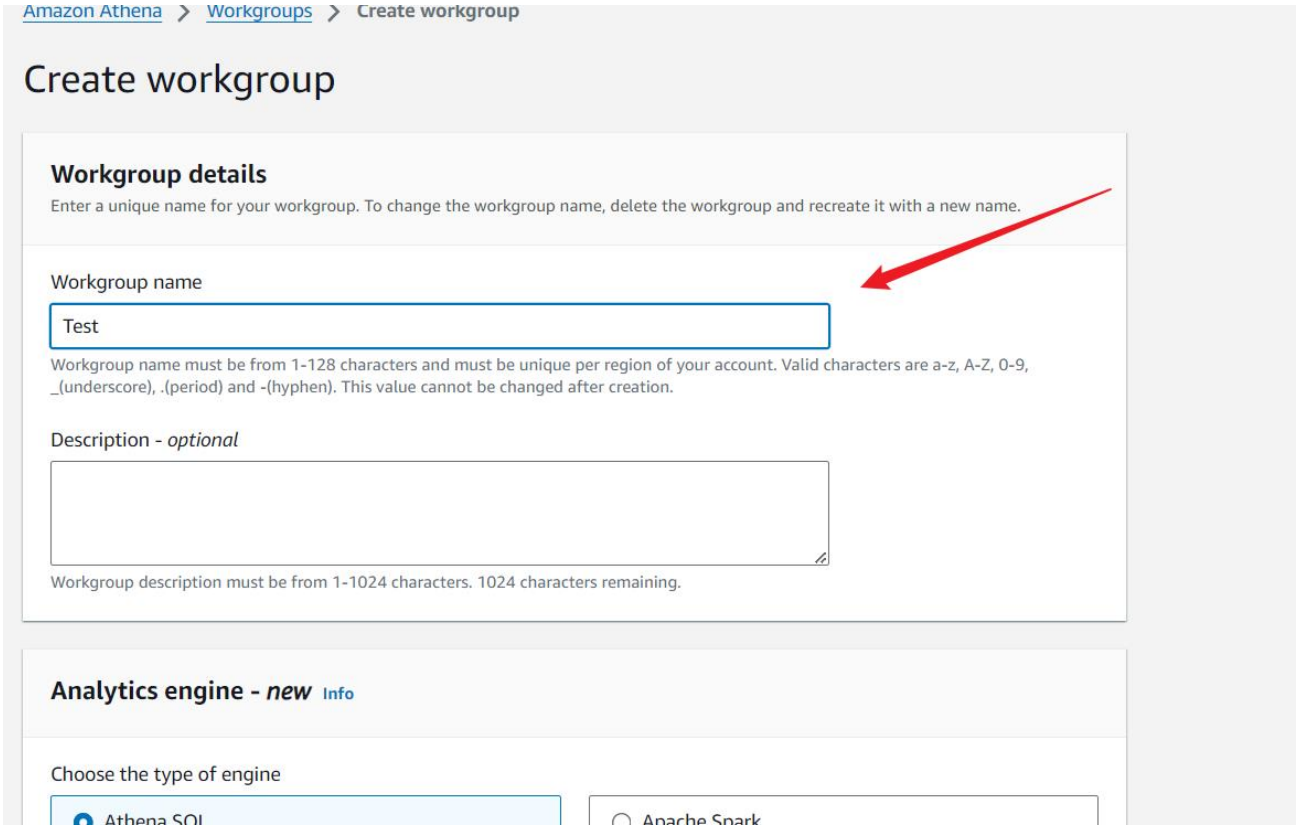
Click ‘Workgroups’ from the left navigation bar.



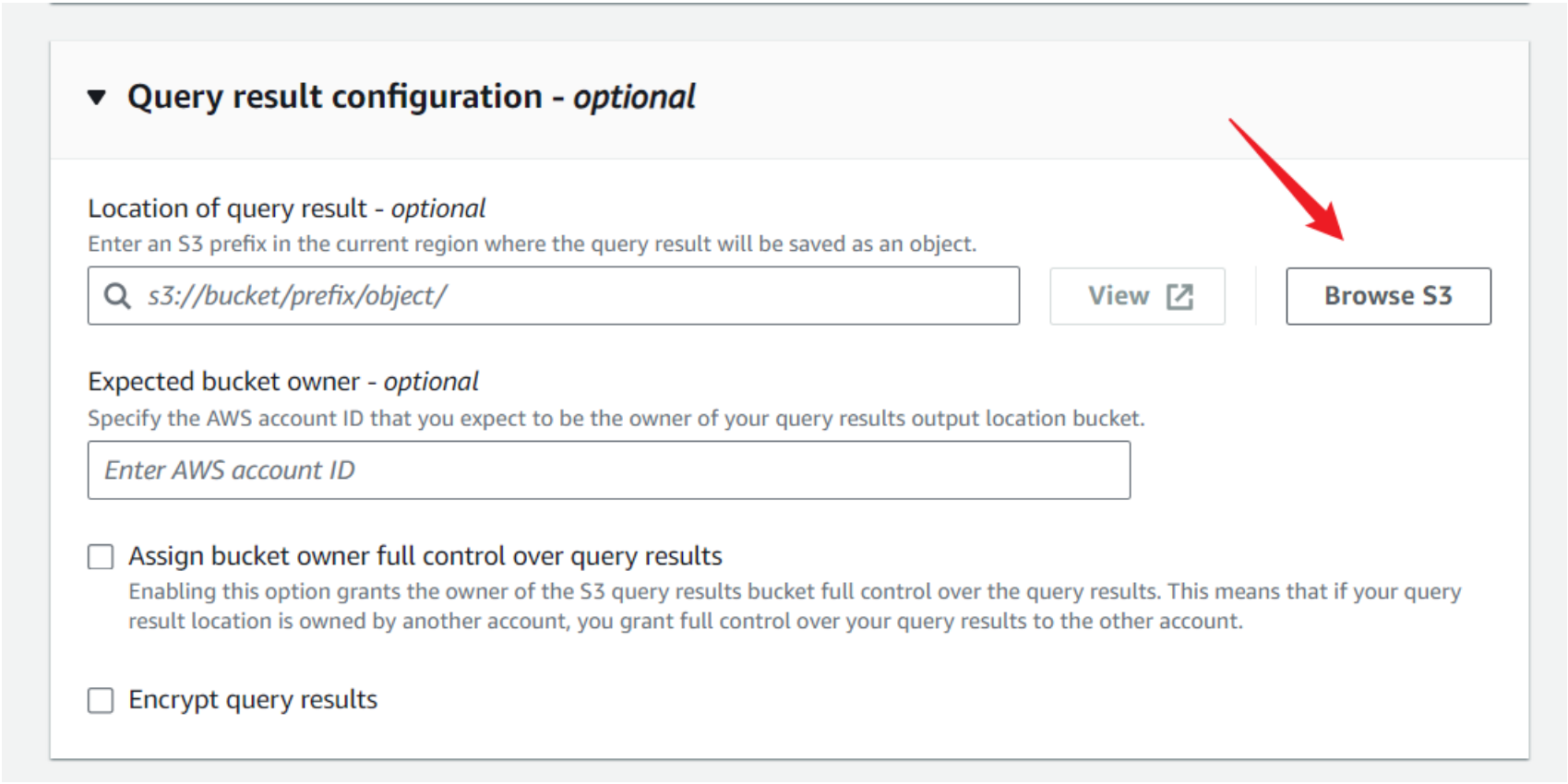
Click ‘Create workgroup’



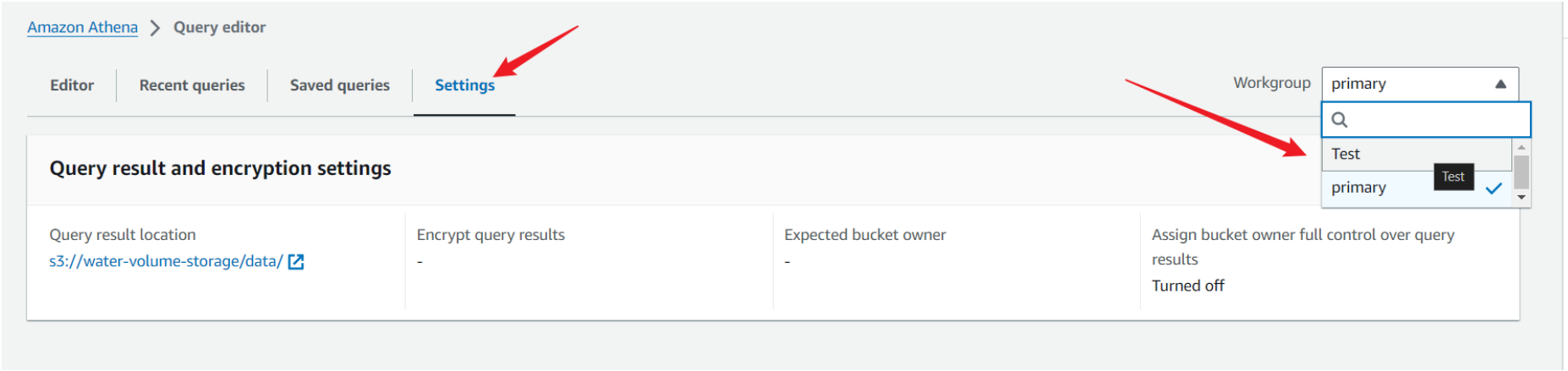
Enter your workgroup name.



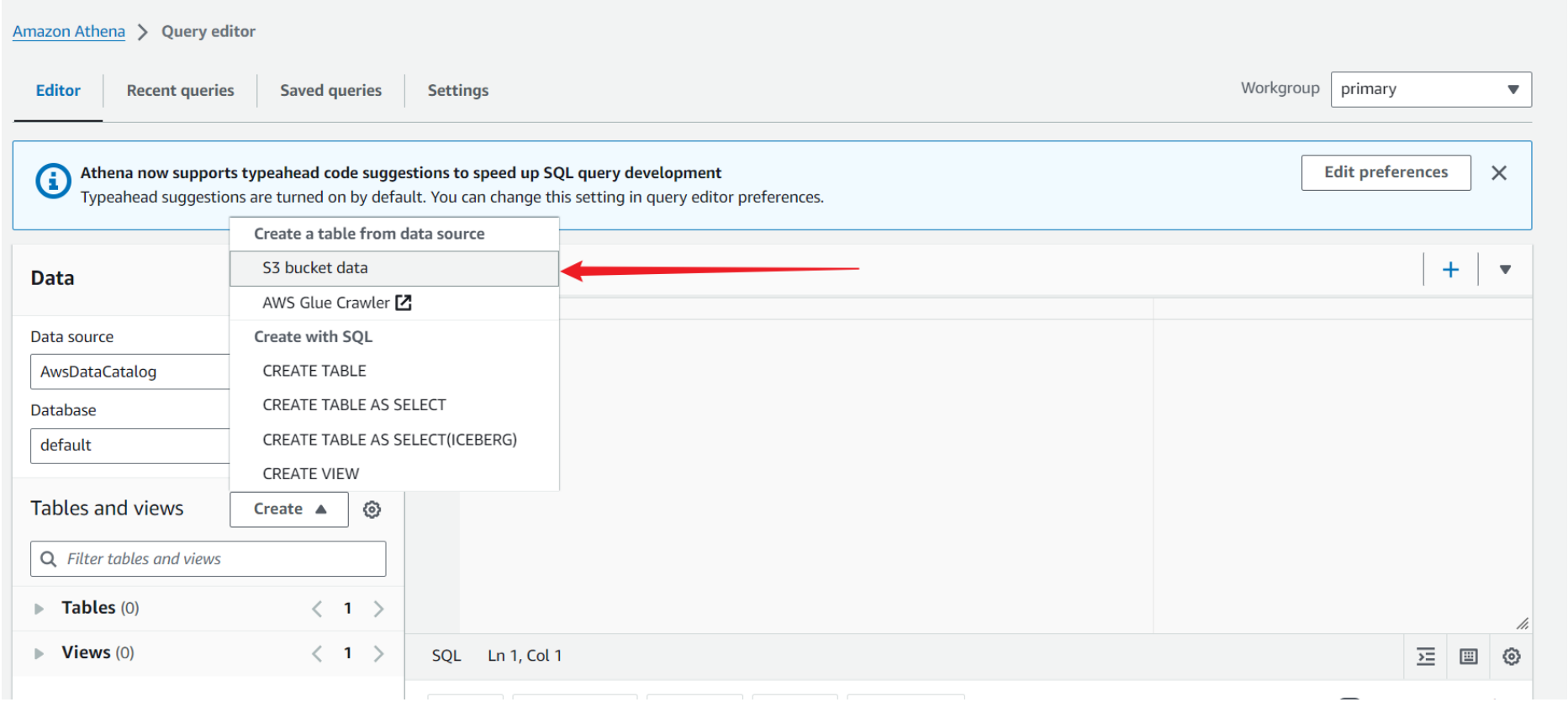
Click 'Browse S3' to select the S3 bucket you created in the previous step. Leave the other settings as default and click 'Create workgroup'.



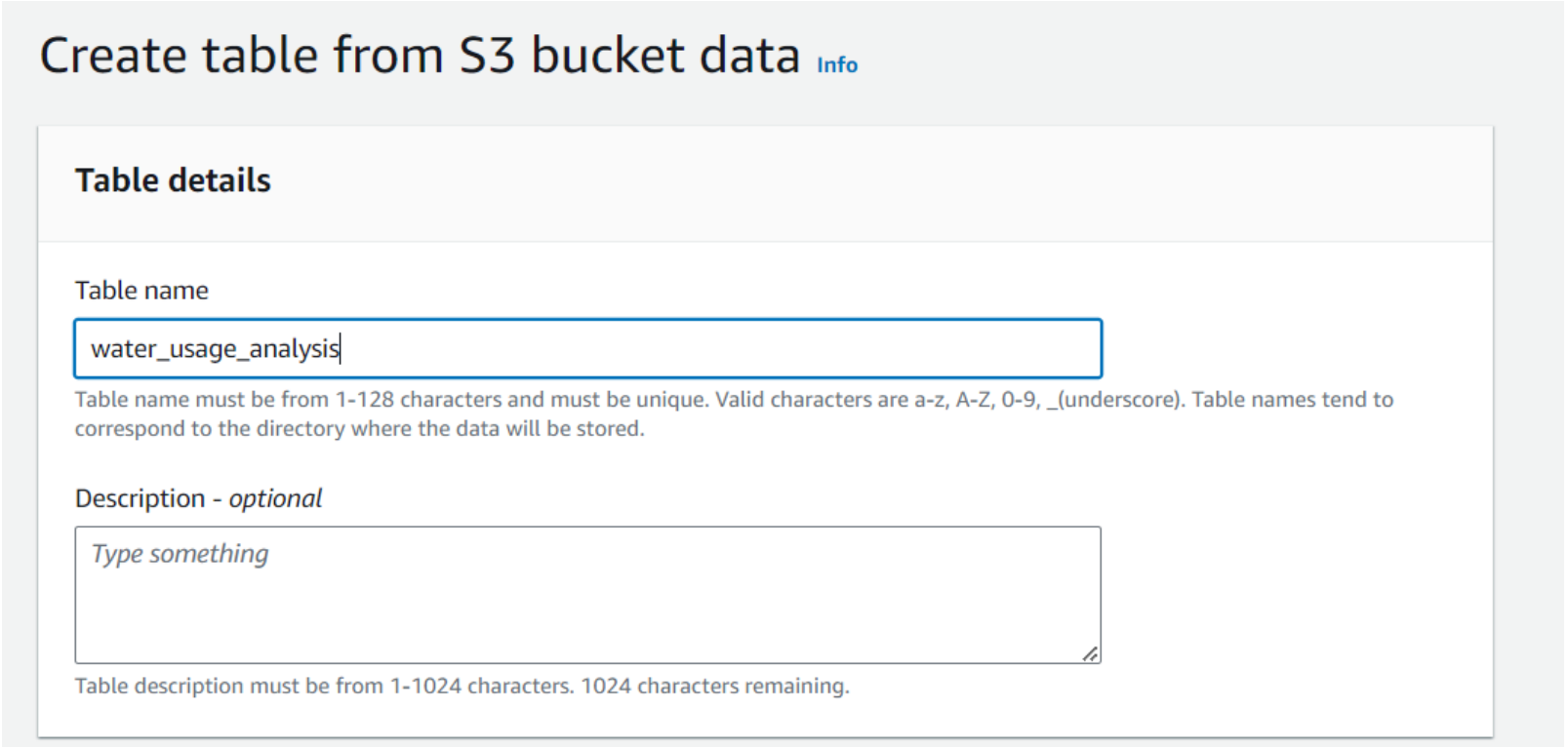
Go back to 'Settings' and switch to the workgroup you just created.



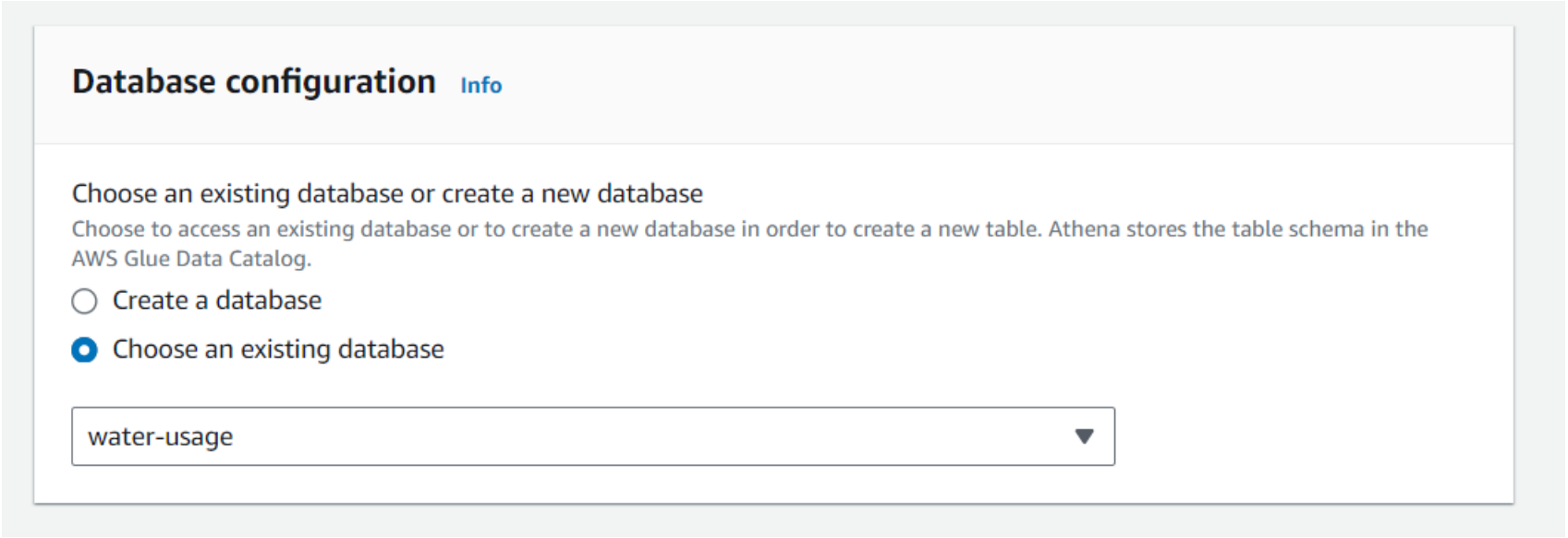
Click the 'Editor' tab, then click 'Create' and select 'S3 bucket data'. This will manually create a table directly from your S3 bucket.



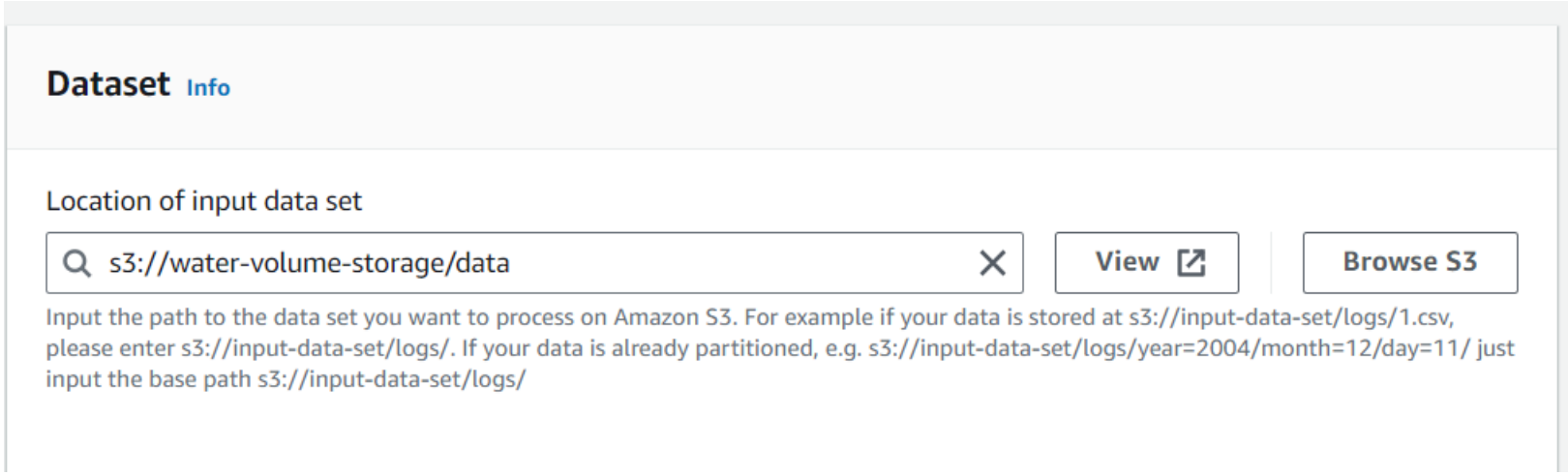
Enter ‘water_usage_analysis’ as the table name.



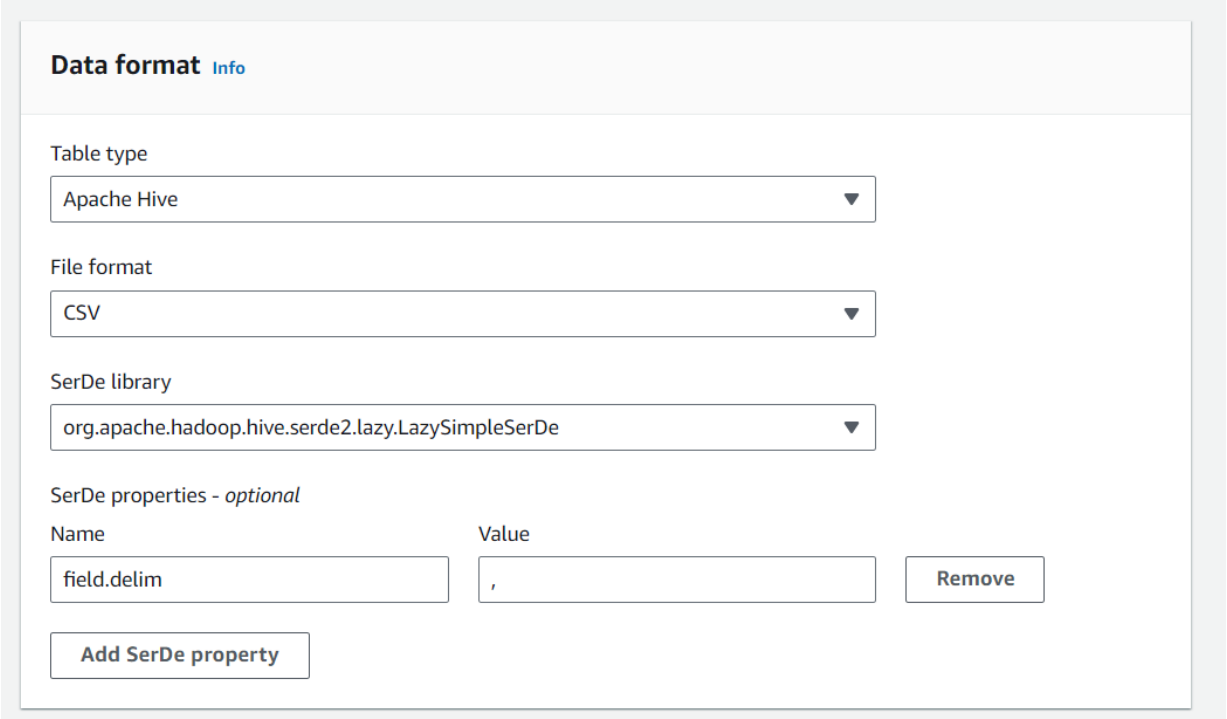
You can create a new database or choose an existing database.



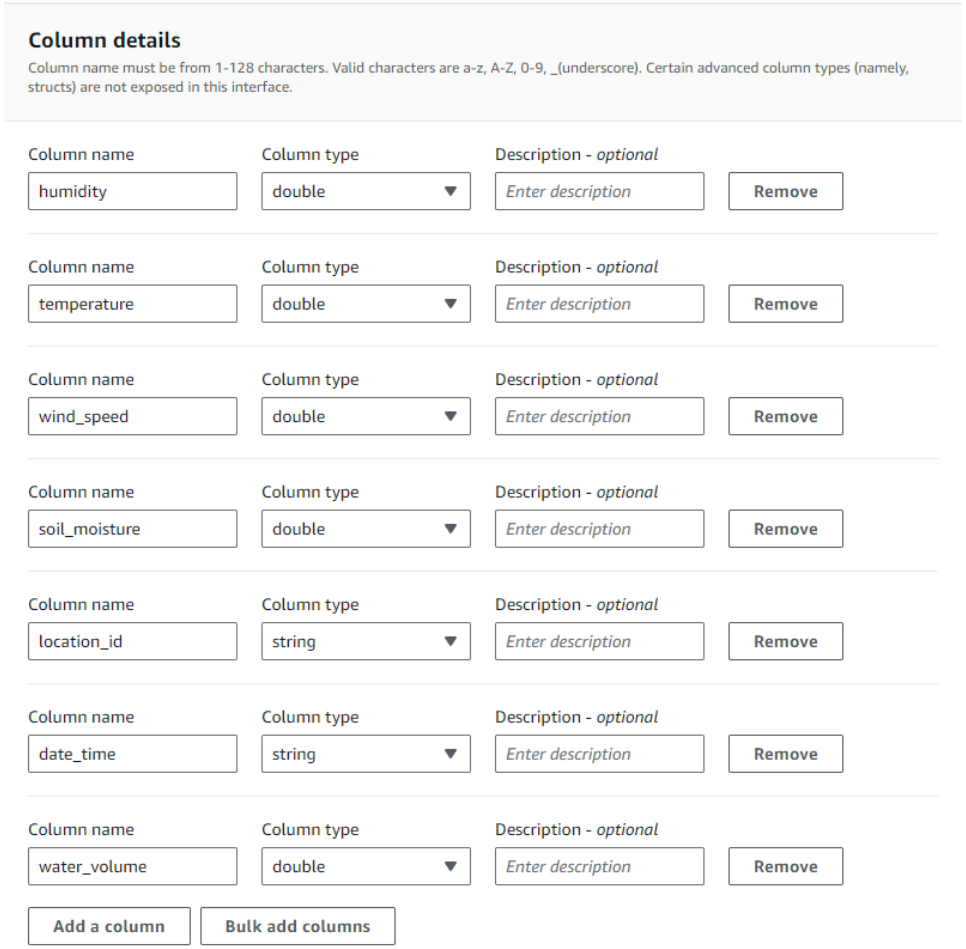
Click on ‘Browse S3’ to select the bucket where you store the data.



Select the data format as shown in the picture.



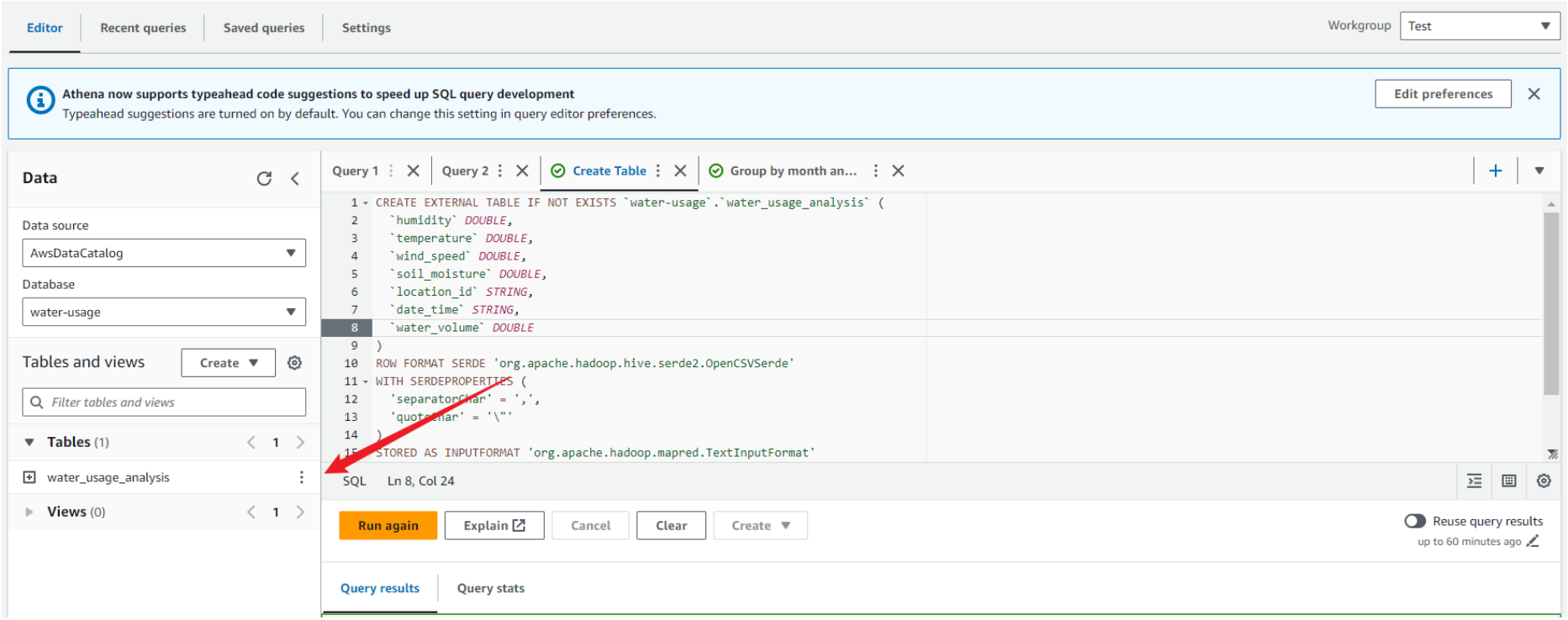
Enter the names and data types of all columns, then click 'Create table'.



Alternatively, you can run the following SQL statement to create a table.

```
CREATE EXTERNAL TABLE IF NOT EXISTS `water-
usage`.`water_usage_analysis` (
  `humidity` DOUBLE,
  `temperature` DOUBLE,
  `wind_speed` DOUBLE,
  `soil_moisture` DOUBLE,
  `location_id` STRING,
  `date_time` STRING,
  `water_volume` DOUBLE
)
ROW FORMAT SERDE 'org.apache.hadoop.hive.serde2.OpenCSVSerde'
WITH SERDEPROPERTIES (
  'separatorChar' = ',',
  'quoteChar' = '\"'
)
STORED AS INPUTFORMAT 'org.apache.hadoop.mapred.TextInputFormat'
OUTPUTFORMAT
'org.apache.hadoop.hive.ql.io.HiveIgnoreKeyTextOutputFormat'
LOCATION 's3://bucketdemounique/input/'
TBLPROPERTIES ('classification' = 'csv');
```

You will see that a new ‘water_usage_analysis’ table has been successfully created.

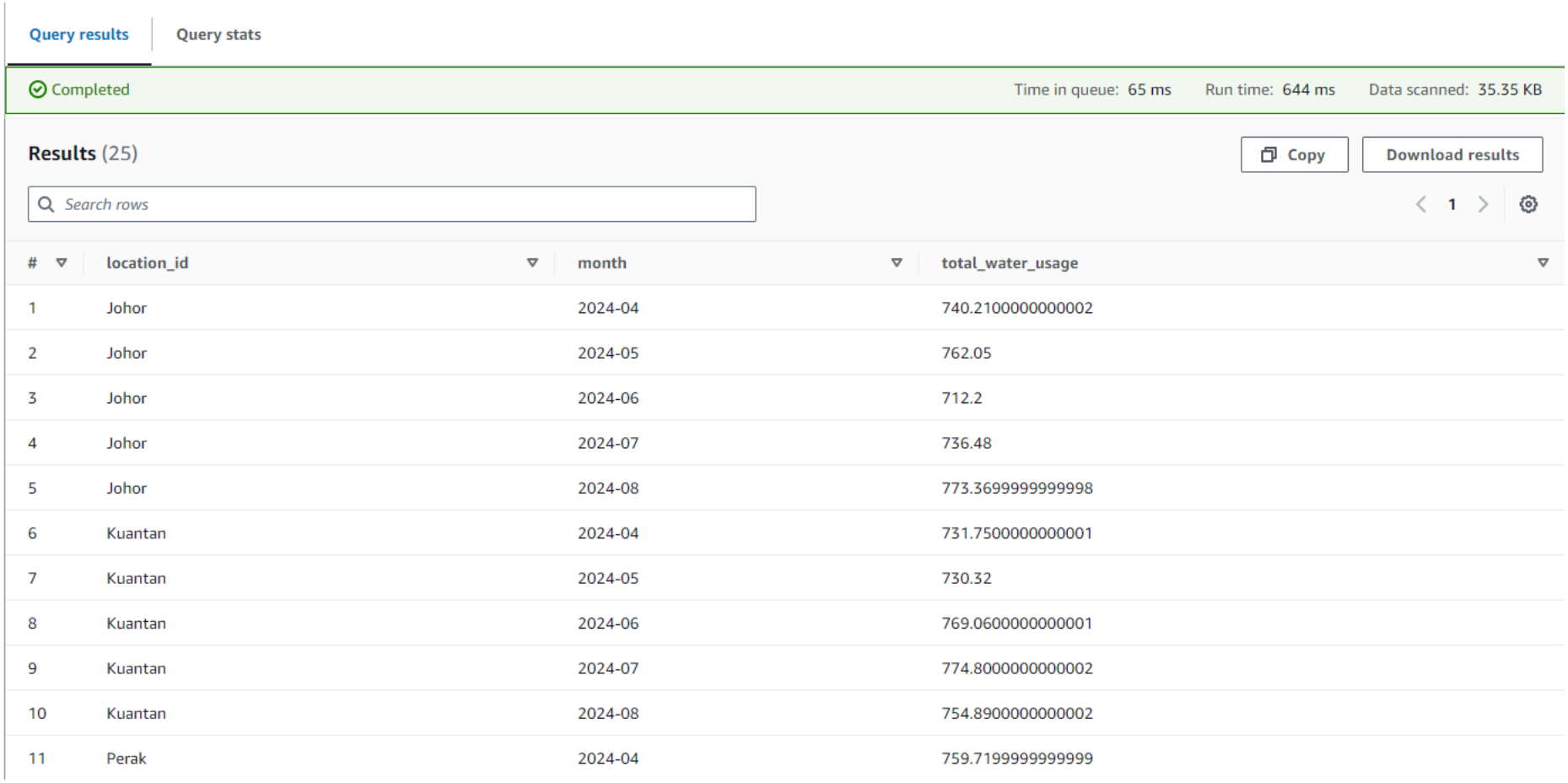


Now, we can perform an SQL query on the table to filter the data we want to visualize in QuickSight. Run the following SQL statement:

```
SELECT
  CASE
    WHEN location_id = 'region_1' THEN 'Johor'
    WHEN location_id = 'region_2' THEN 'Perak'
    WHEN location_id = 'region_3' THEN 'Selangor'
    WHEN location_id = 'region_4' THEN 'Kuantan'
    WHEN location_id = 'region_5' THEN 'Sabah'
    ELSE location_id
  END AS location_id,
  DATE_FORMAT(
    parse_datetime(date_time, 'd/M/yyyy HH:mm'),
    '%Y-%m'
  ) AS month,
  SUM(water_volume) AS total_water_usage
FROM
  "water-usage"."water_usage_analysis"
```

```
GROUP BY
  location_id,
  DATE_FORMAT(
    parse_datetime(date_time, 'd/M/yyyy HH:mm'),
    '%Y-%m'
  )

ORDER BY
  location_id,
  DATE_FORMAT(
    parse_datetime(date_time, 'd/M/yyyy HH:mm'),
    '%Y-%m'
  );
```

The query result:



Query results | Query stats

Completed — Time in queue: 65 ms — Run time: 644 ms — Data scanned: 35.35 KB

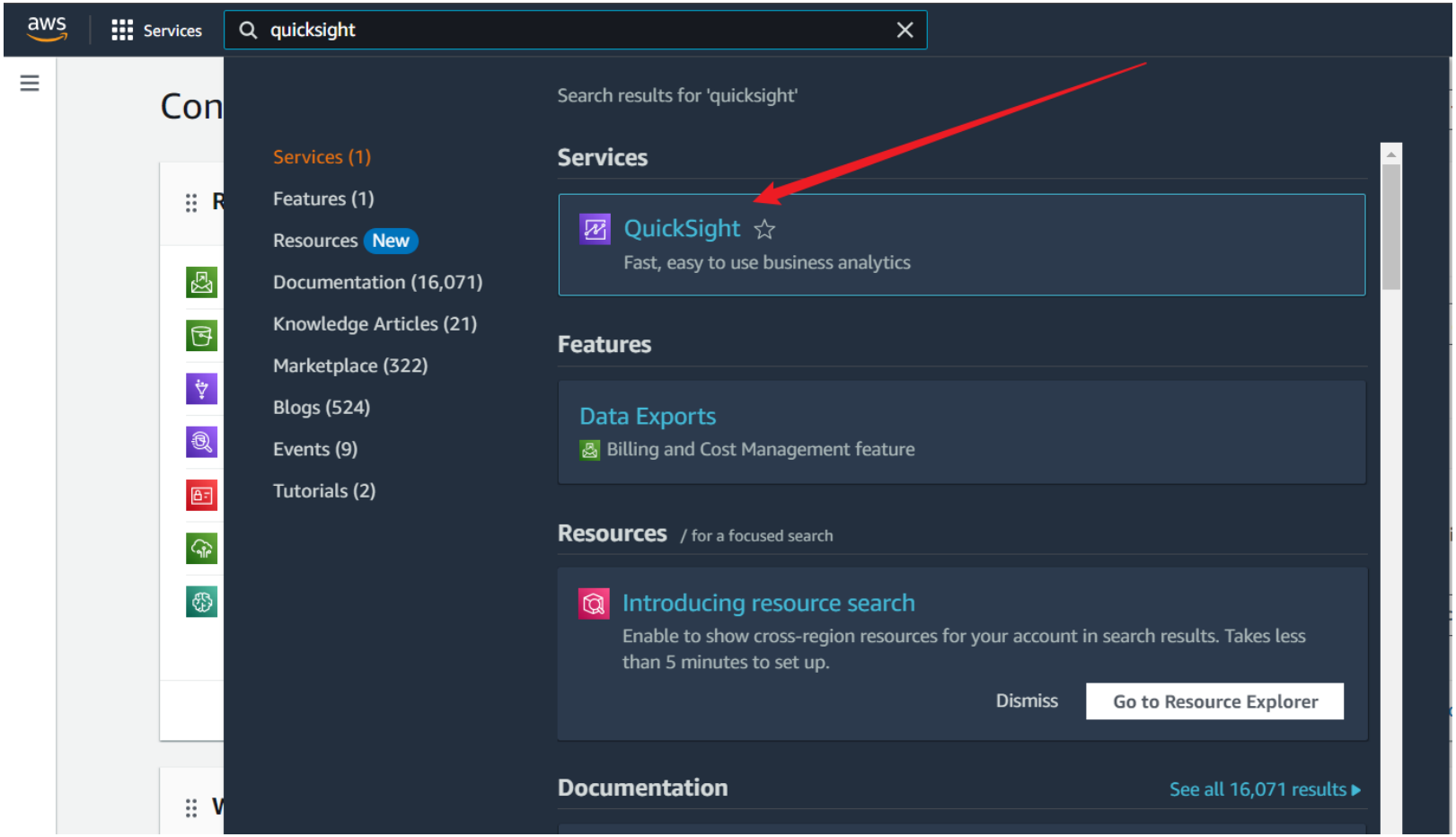
Results (25) — Copy — Download results

Search rows — 1

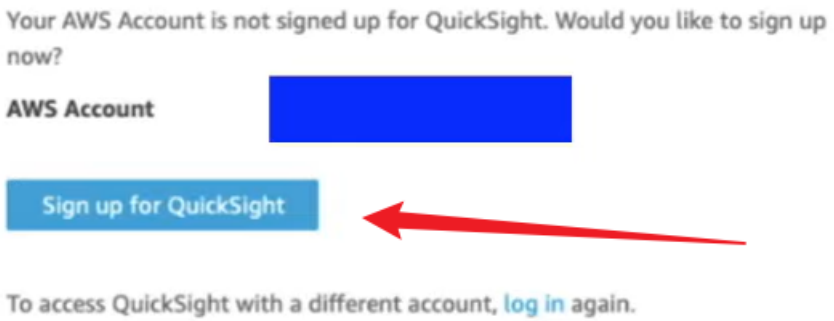
#	location_id	month	total_water_usage
1	Johor	2024-04	740.2100000000002
2	Johor	2024-05	762.05
3	Johor	2024-06	712.2
4	Johor	2024-07	736.48
5	Johor	2024-08	773.3699999999998
6	Kuantan	2024-04	731.7500000000001
7	Kuantan	2024-05	730.32
8	Kuantan	2024-06	769.0600000000001
9	Kuantan	2024-07	774.8000000000002
10	Kuantan	2024-08	754.8900000000002
11	Perak	2024-04	759.7199999999999

Amazon QuickSight

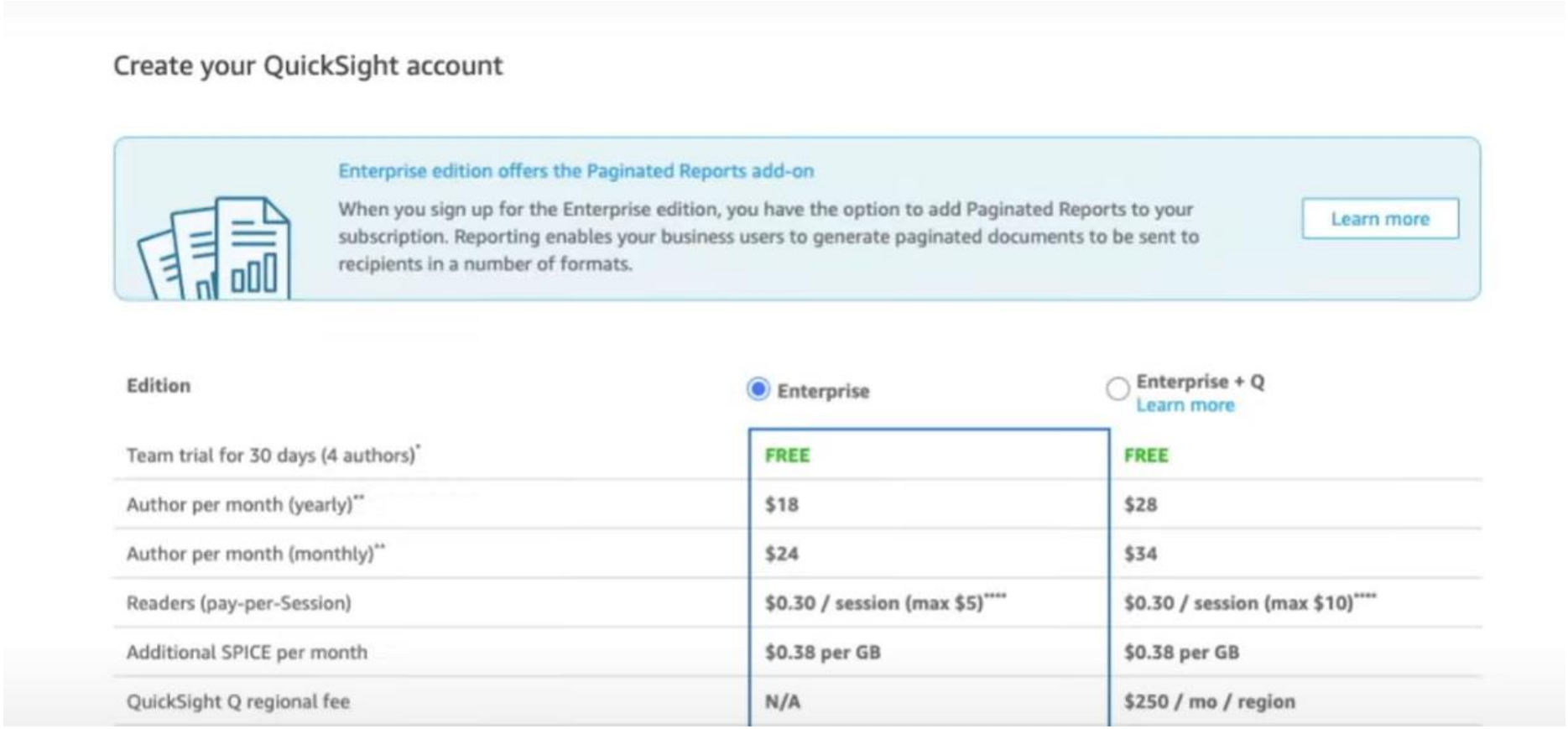
Type ‘QuickSight’ in search bar and click the ‘QuickSight’ service.



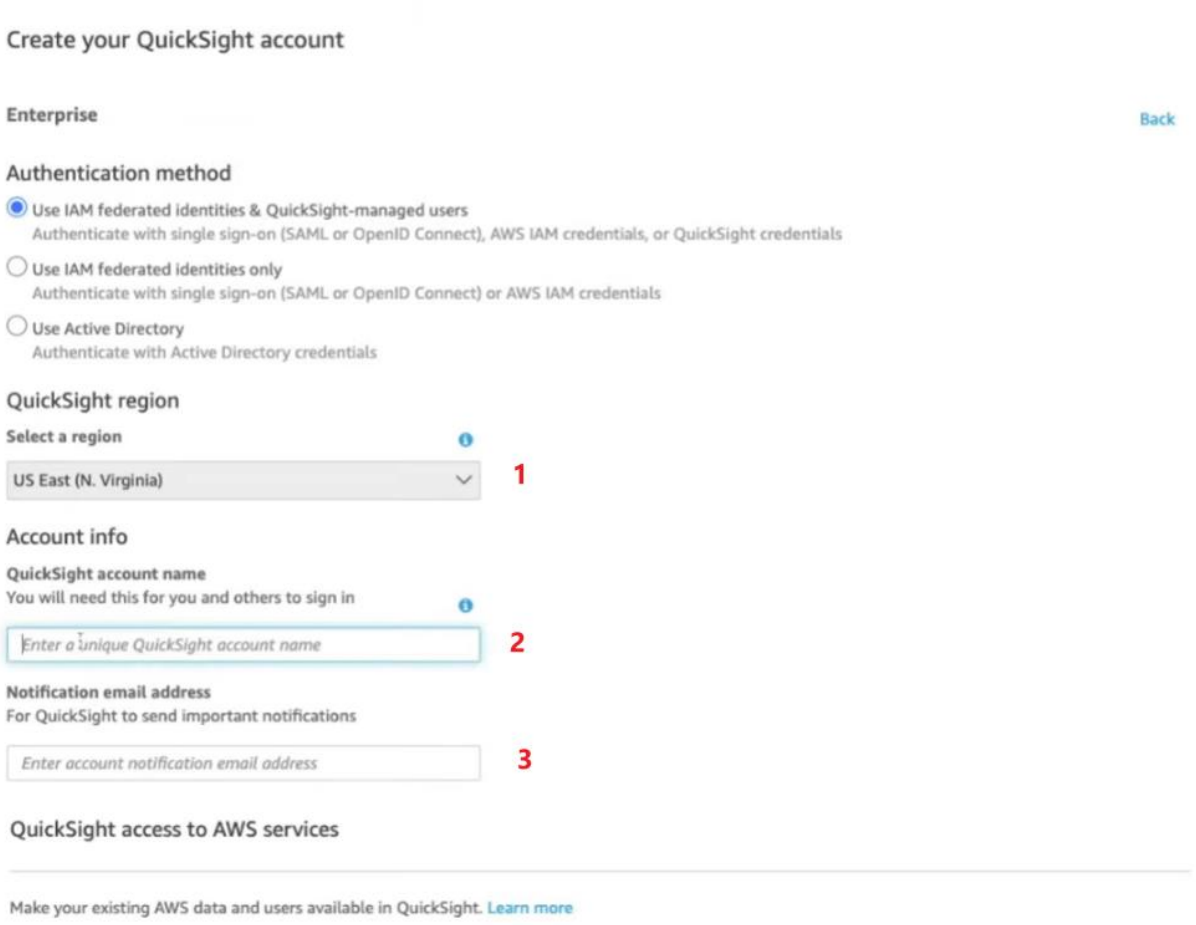
Click ‘Sign up for QuickSight’



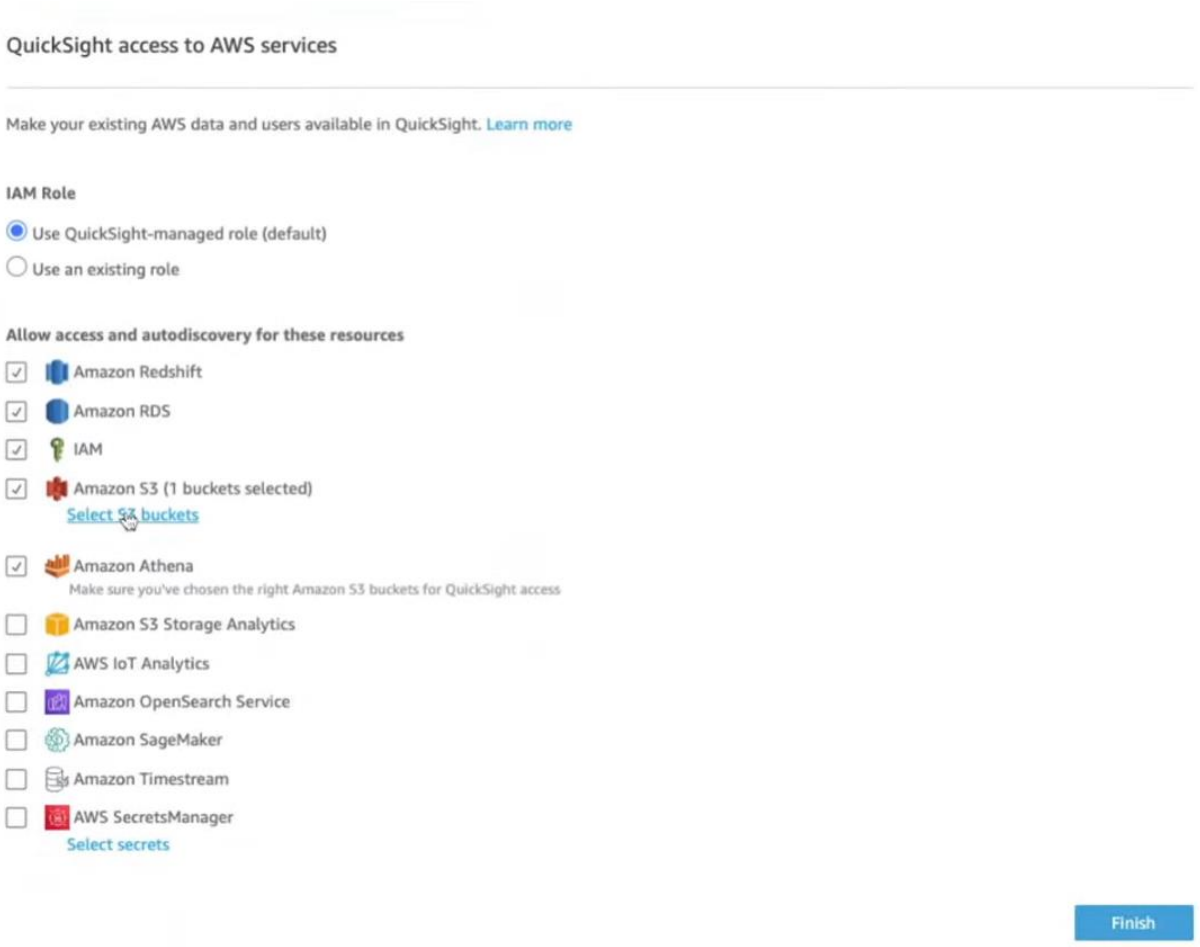
Choose the ‘Enterprise’ edition and press ‘Continue’.



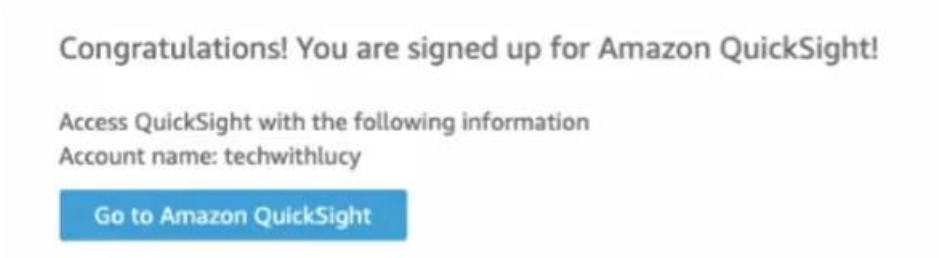
1. Select ‘Singapore’ for the region.
2. Enter your account name.
3. Enter your notification email.



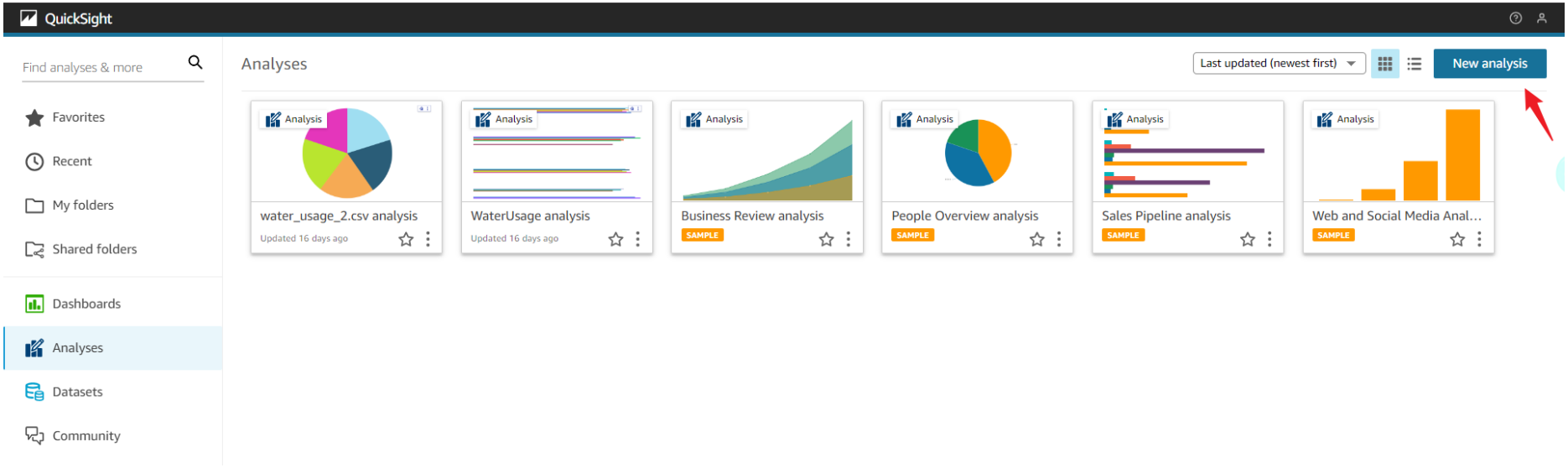
Click on 'Select S3 buckets' and choose the S3 bucket that contains the water usage data. Leave the other settings as default and click ‘Finish’.



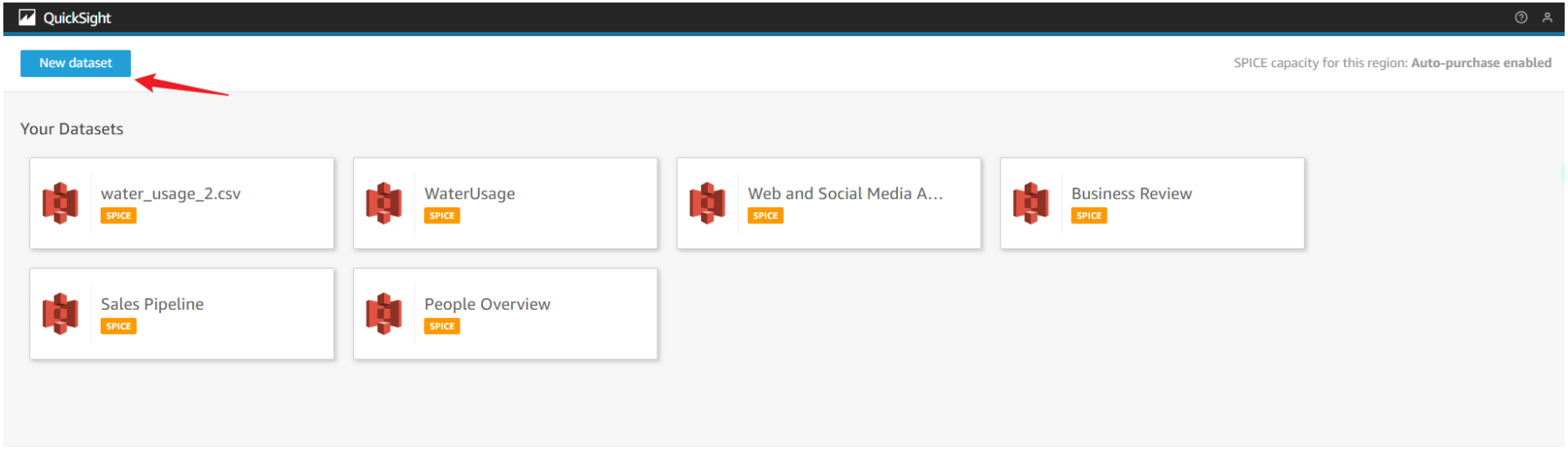
You have now successfully created a QuickSight account.



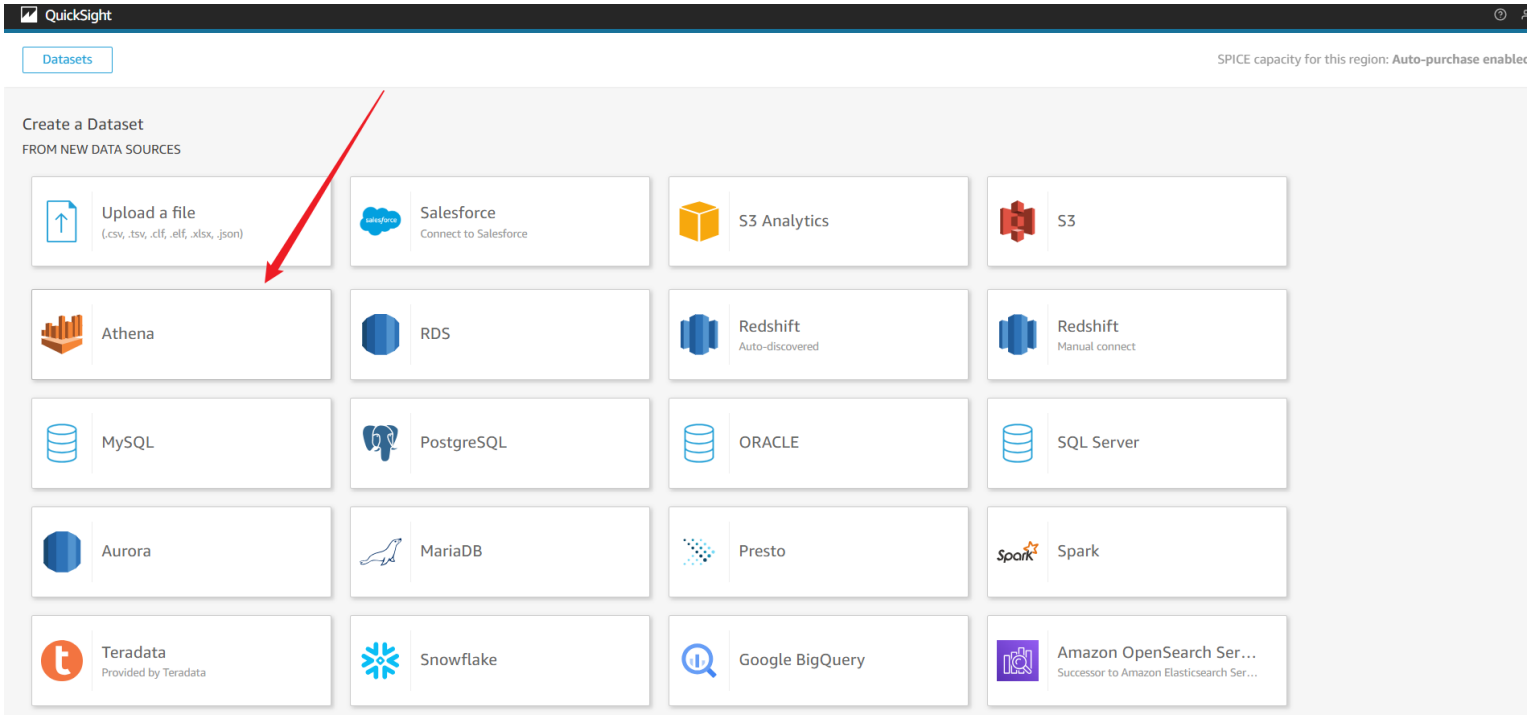
Click ‘New analysis’.



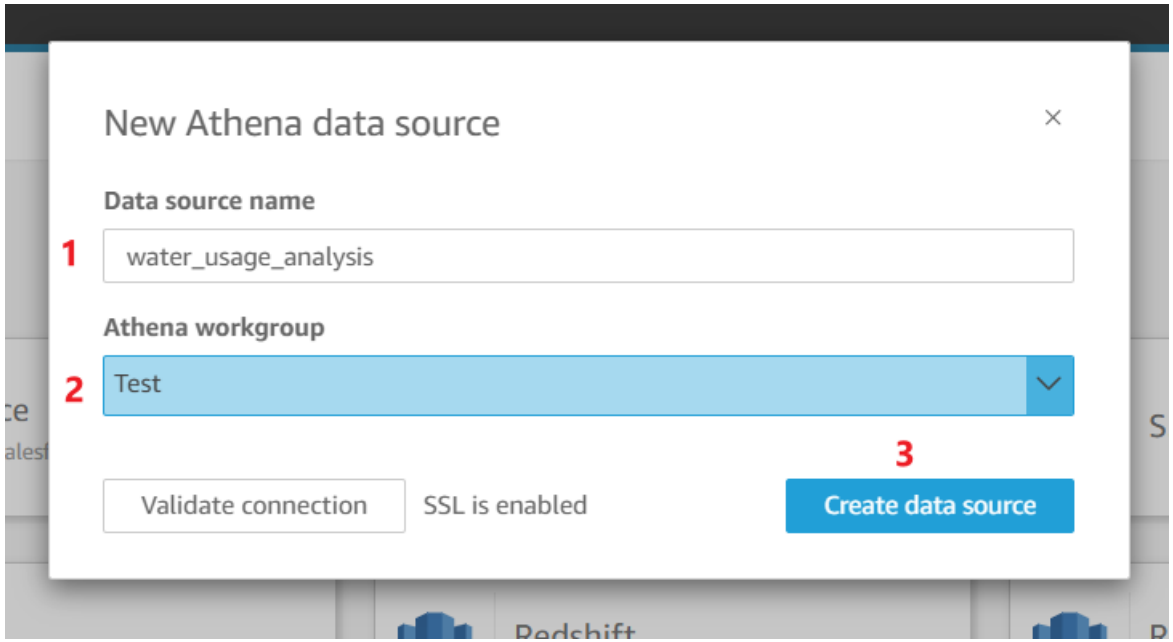
Select ‘New dataset’.



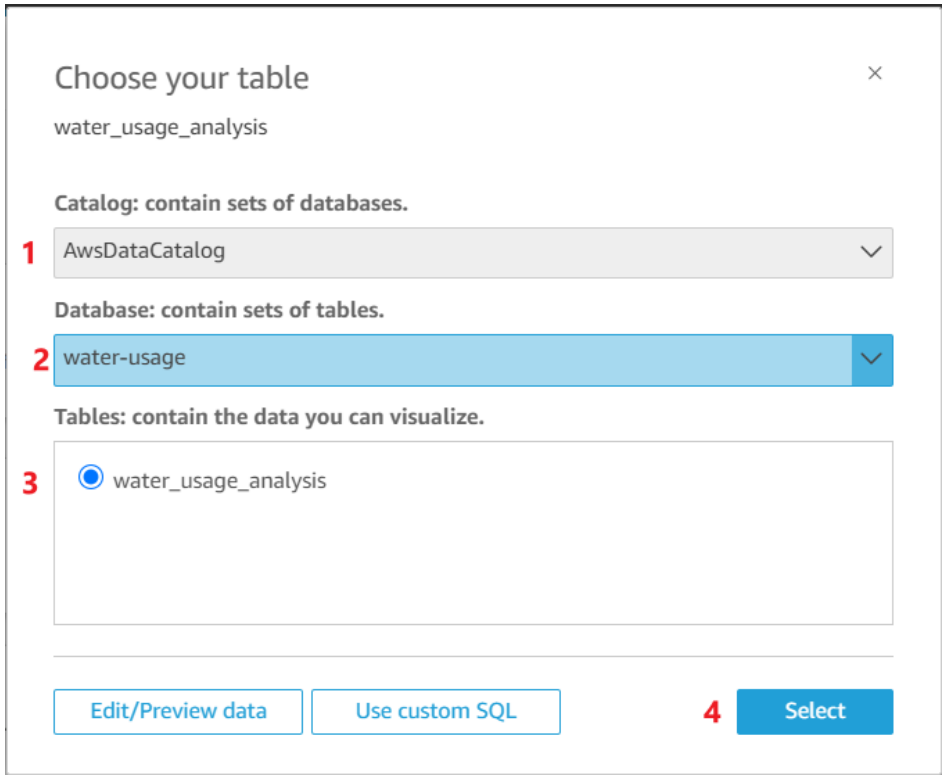
Select ‘Anthena’.



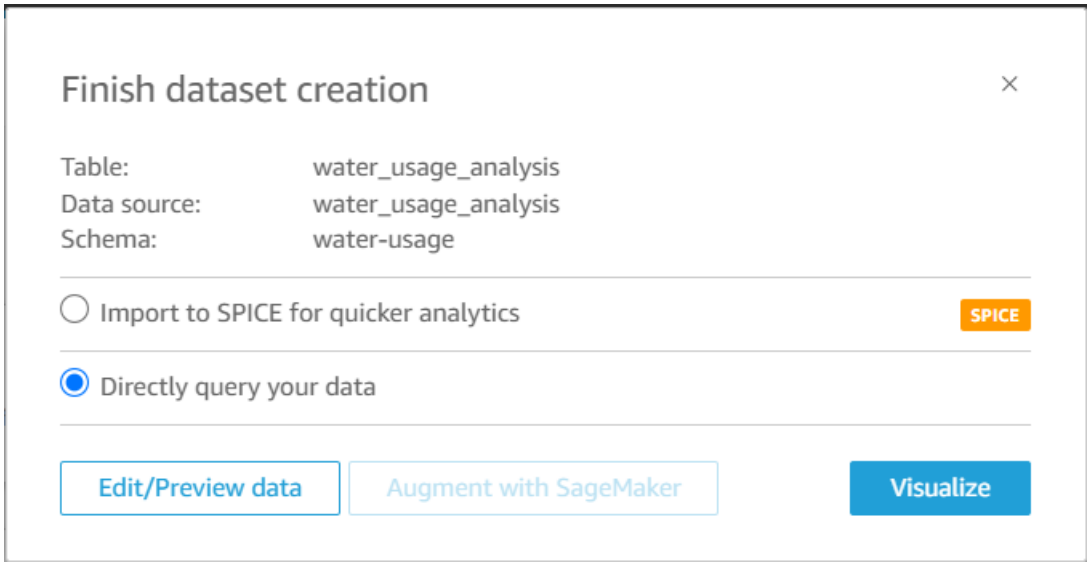
1. Enter ‘water_usage_analysis’ as data source name.
2. Select ‘Test’ as Athena workgroup.
3. Click ‘Create data source’.



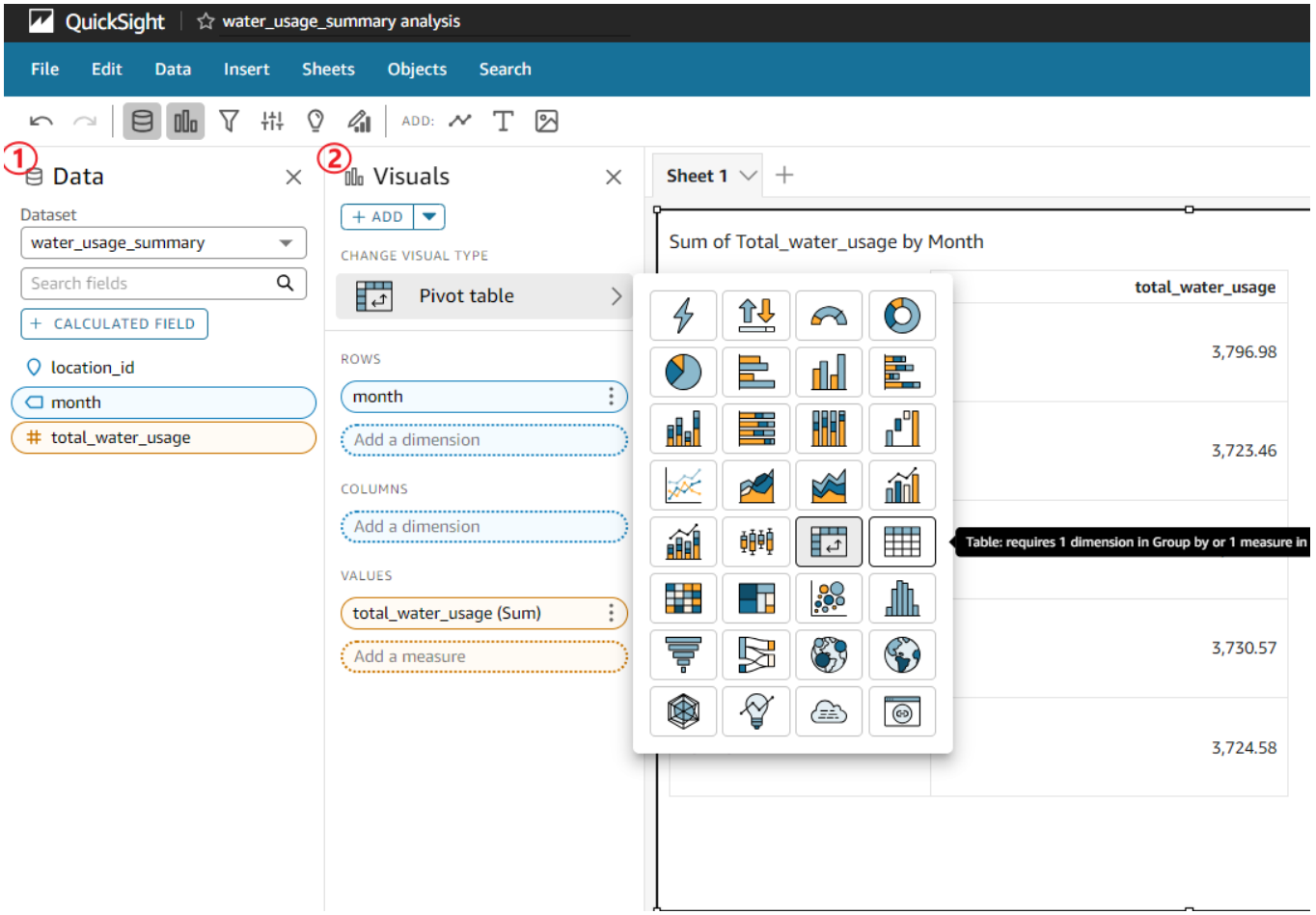
1. Select ‘AwsDataCatalog’
2. Select ‘water-usage’ database.
3. Select ‘water_usage_analysis’ table.



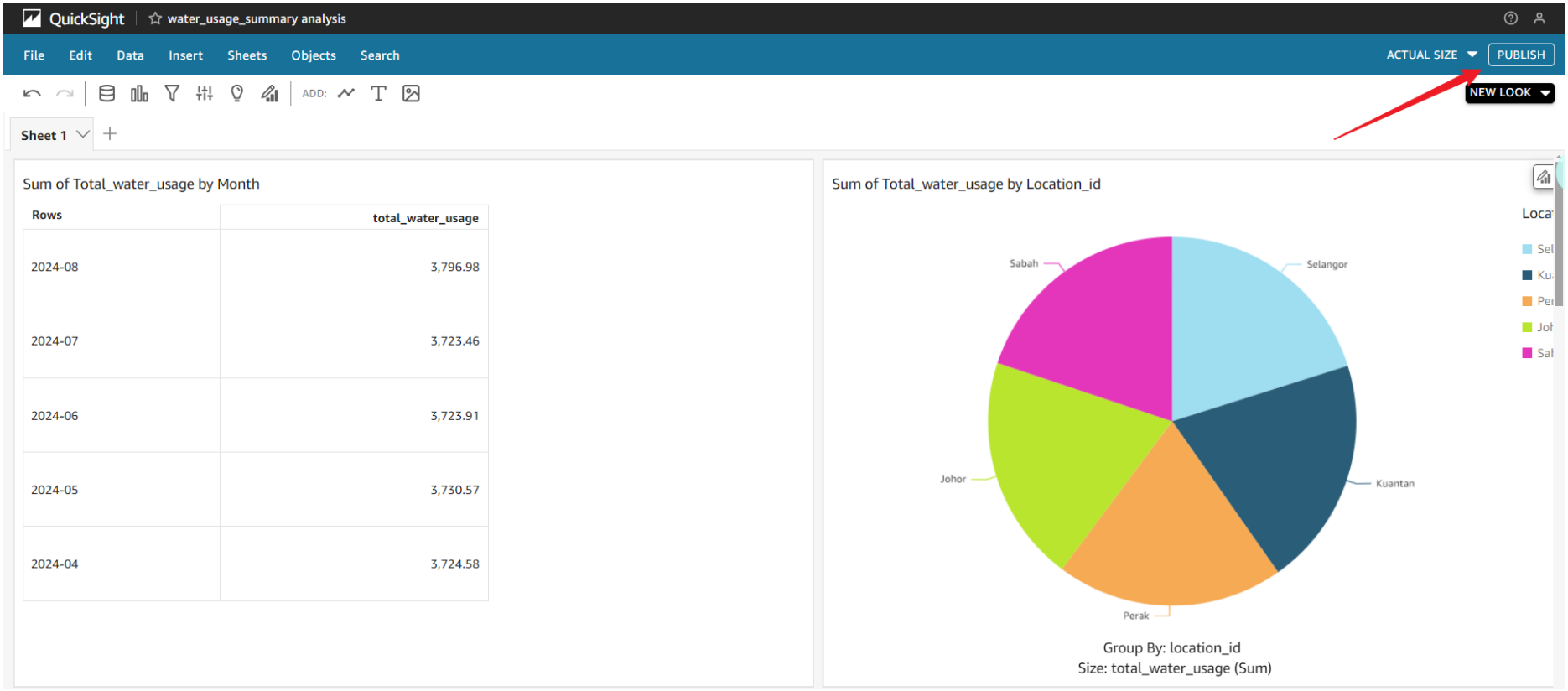
Choose ‘Directly query your data’ and click ‘Visualize’.



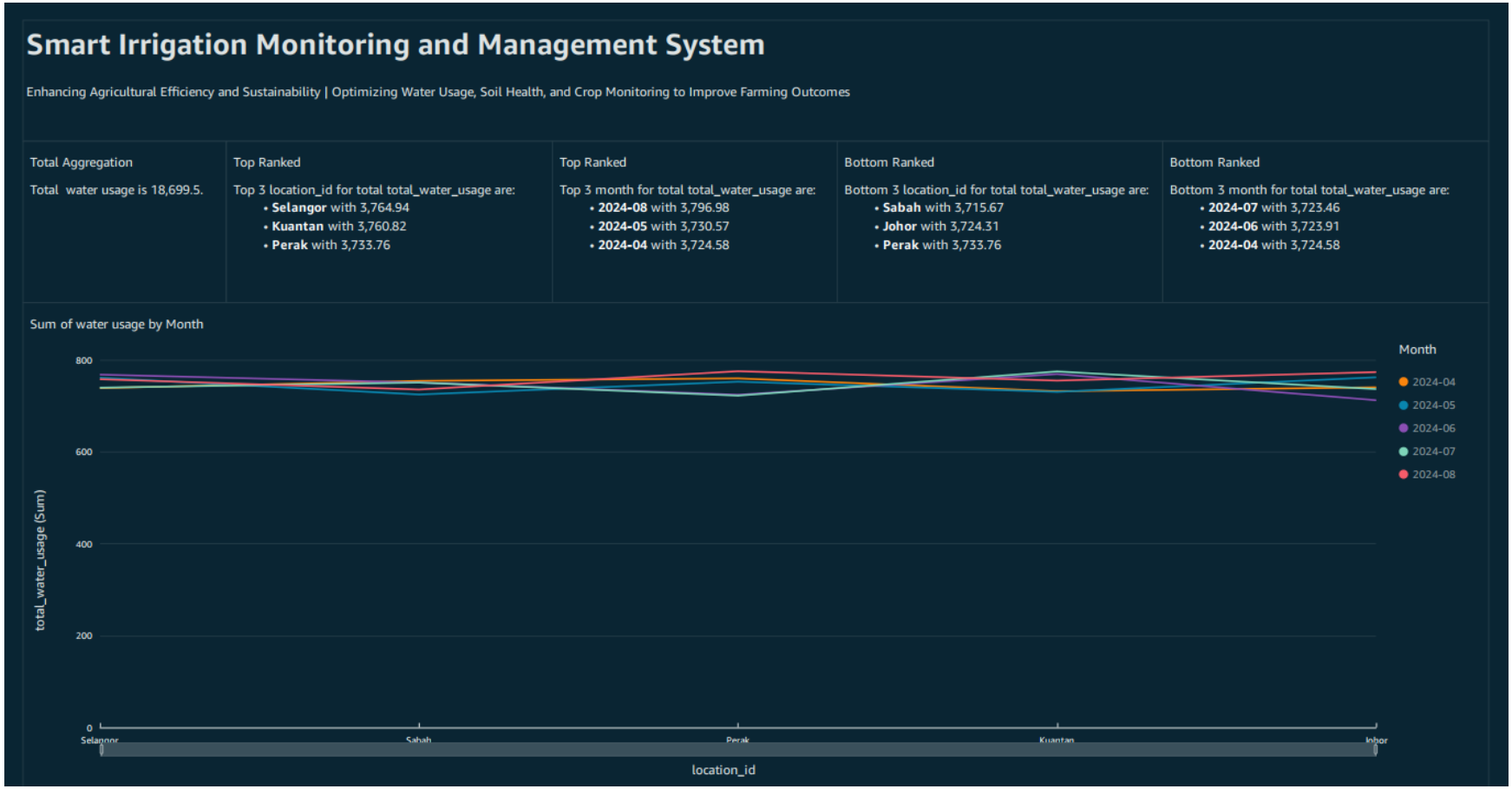
1. You can choose multiple data that want to be visualized.
2. You can choose various types of visualization such as bar chart, histogram, line chart, etc.



After you finish customizing your visualization, you can click the 'PUBLISH' button.

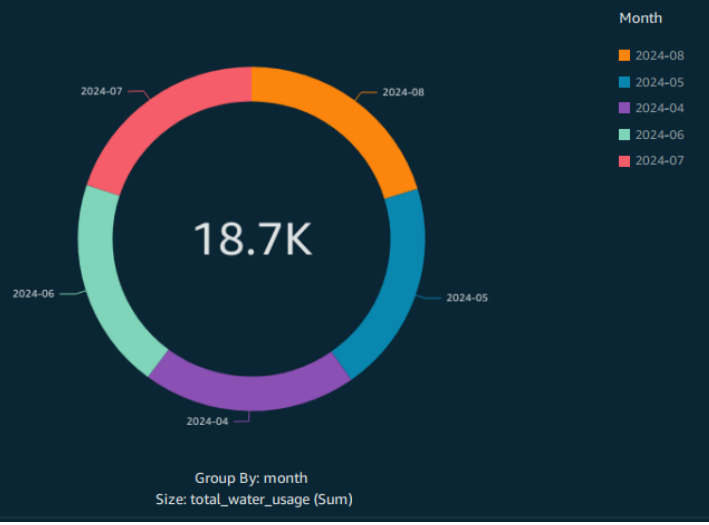


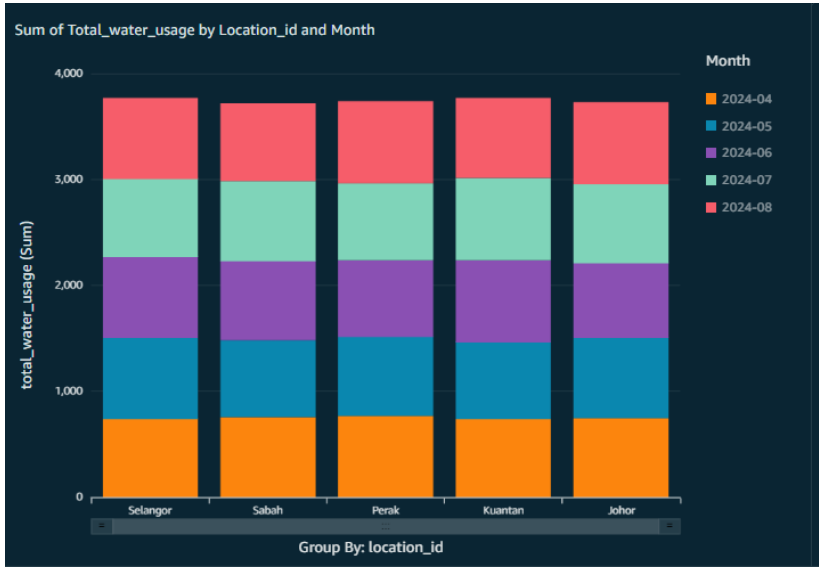
Here is the completed dashboard.

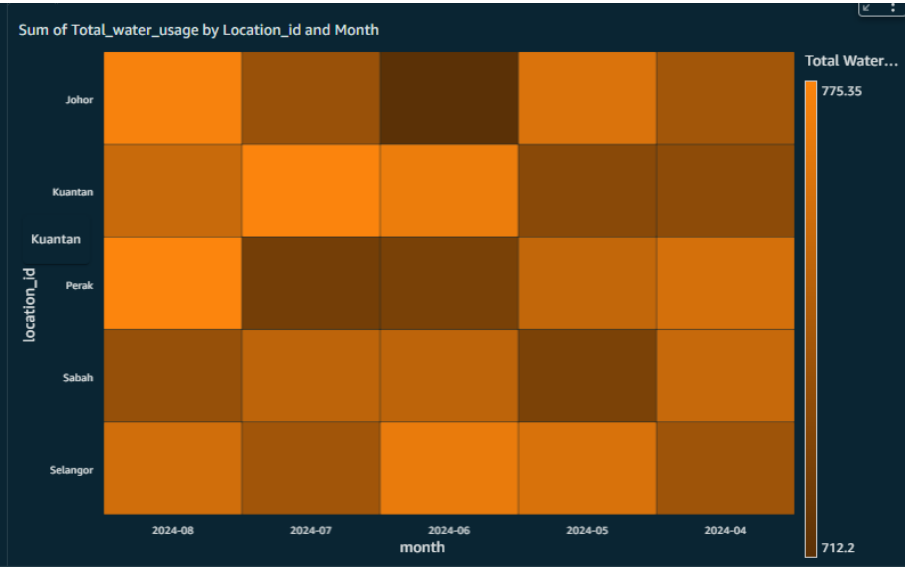


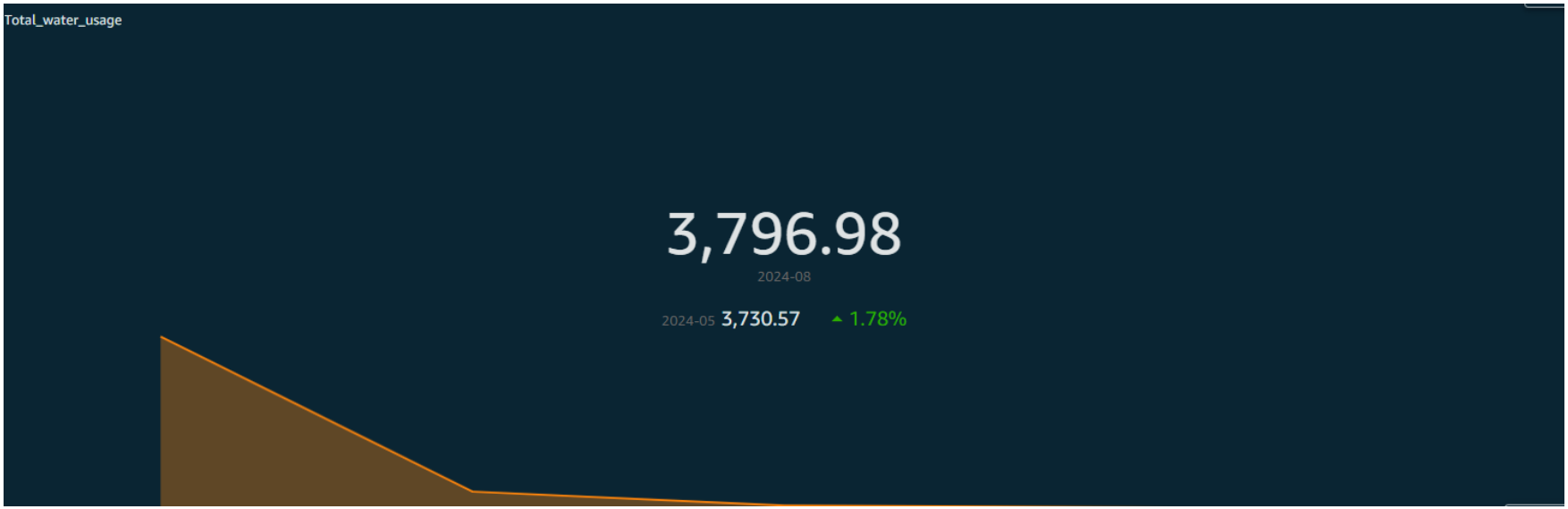
Sum of water usage by Month

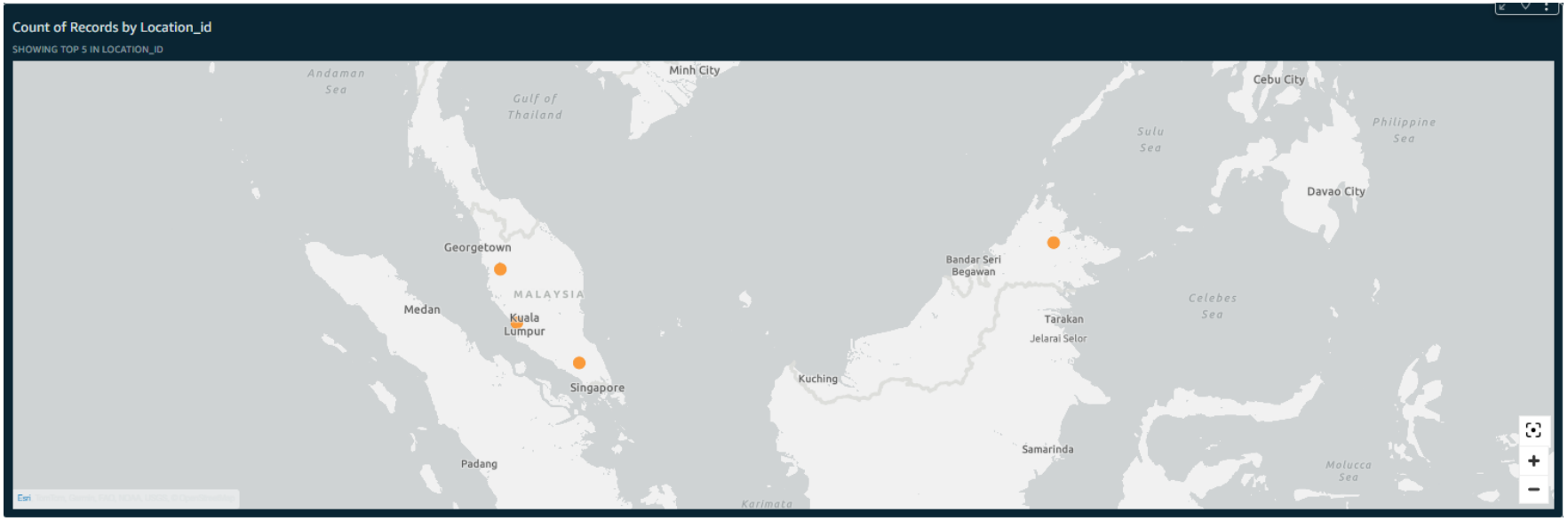
Rows	total_water...	wind_speed	temperature	soil_moisture	humidity
2024-04	3,724.58	9.64	148.63	338.81	50.1
2024-05	3,730.57	9.84	149.07	336.72	49.64
2024-06	3,723.91	10.14	151.67	337.25	50.77
2024-07	3,723.46	9.9	149.9	337.97	49.84
2024-08	3,796.98	9.93	150.24	339.56	50.42







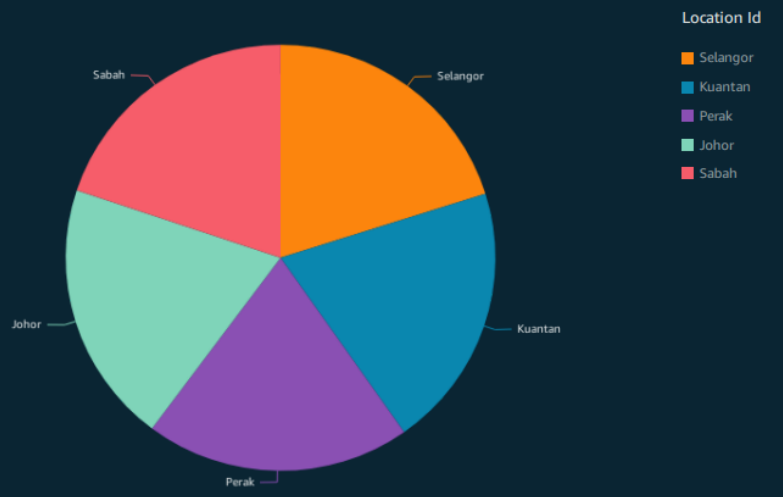

Total_water_usage
3,796.98
2024-08
2024-05 3,730.57
1.78%


Count of Records by Location_id
SHOWING TOP 5 IN LOCATION_ID
Andaman Sea
Gulf of Thailand
Minh City
Cebu City
Sulu Sea
Philippine Sea
Davao City
Georgetown
MALAYSIA
Medan
Kuala Lumpur
Singapore
Bandar Seri Begawan
Tarakan
Jelarai Selor
Celebes Sea
Kuching
Samarinda
Padang
Molucca Sea
Karimata
Esri

Sum of water usage by Location

Rows	total_water_usage	humidity	soil_moisture	temperature	wind_speed
Johor	3,724.31	50.45	337.29	149.93	9.82
Kuantan	3,760.82	50.26	340.29	150.74	10.29
Perak	3,733.76	50.09	338.27	150.63	9.68
Sabah	3,715.67	50.23	337.59	148.56	10.19
Selangor	3,764.94	49.74	336.87	149.65	9.47

Sum of water usage by Location



Group By: location_id
Size: total_water_usage (Sum)