CSS532(Final Project Configuration Steps)

Name – Sahithi Chimakurthi

Student ID – 2303017

1. Connect the Ultrasonic Sensor to the Raspberry Pi GPIO Pins:

The ultrasonic sensor measures the distance by sending sound waves and calculating the time it takes for the echo to return. For proper operation, connected the sensor as follows:

VCC to the 5V pin on the Raspberry Pi.

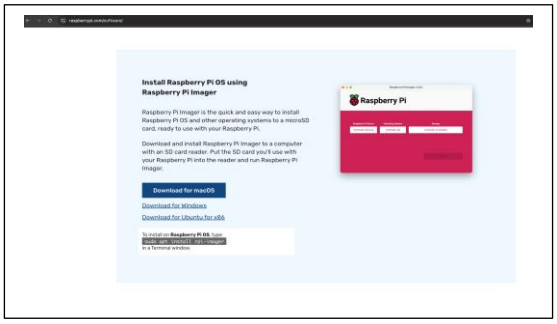
GND to the GND pin on the Raspberry Pi.

TRIG to GPIO Pin 23 on the Raspberry Pi.

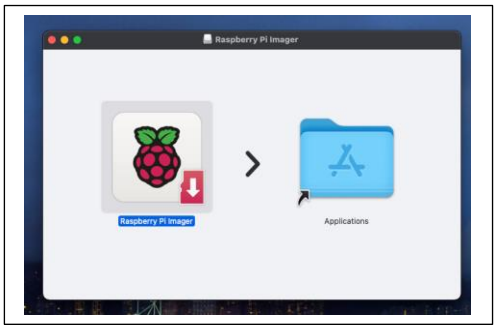
ECHO to GPIO Pin 24 on the Raspberry Pi.

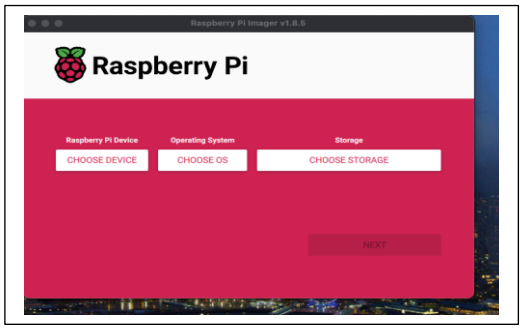
The Raspberry Pi GPIO pins operate at 3.3V, but the sensor operates at 5V. To protect the GPIO pins, I used a 1kΩ resistor and a 2kΩ resistor as a voltage divider for the ECHO pin to step the voltage down to a safe 3.3V before connecting it to the Raspberry Pi

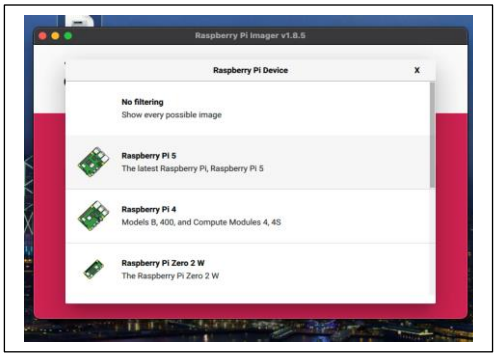
a.) Raspberry Pi Setup: Downloaded and installed the necessary software for the Raspberry Pi.

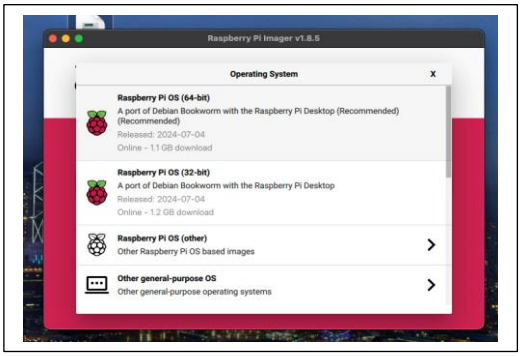


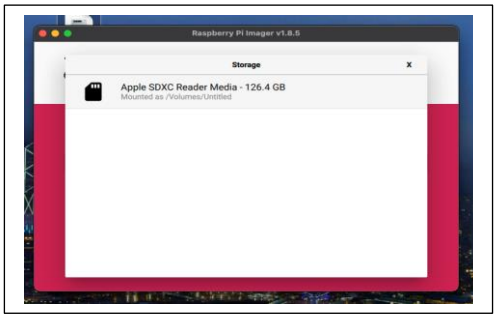
b.) Ran the Raspberry Pi Imager to install the OS onto an external SD card.

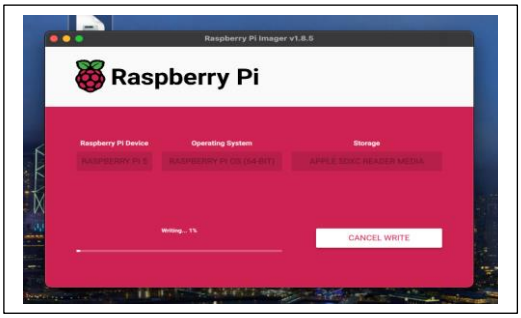


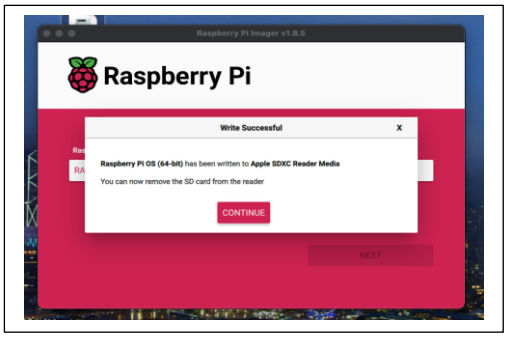




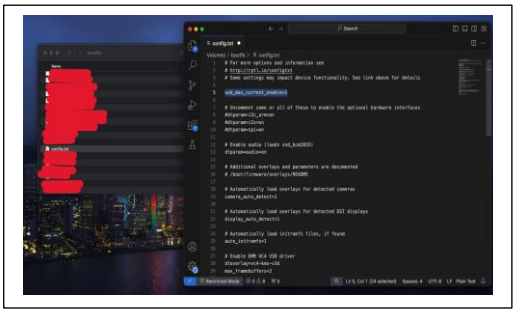




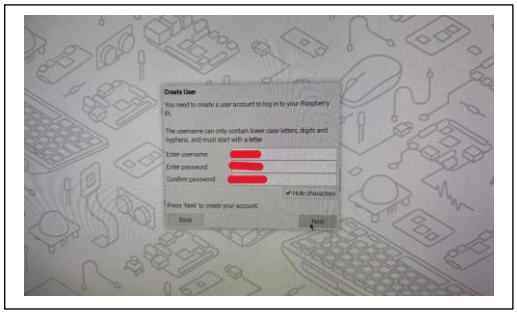


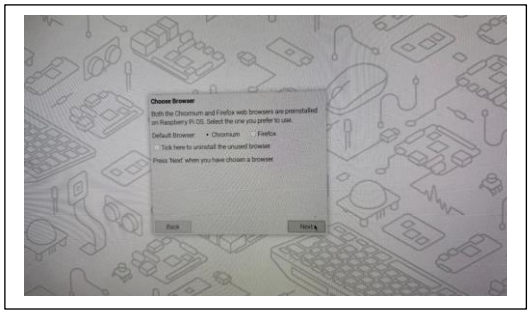


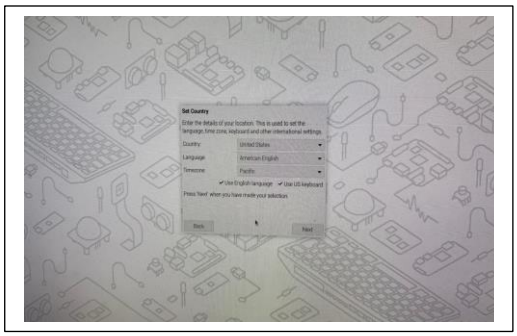
c.) Updated config.txt to enable the Pi to boot from the SD card.

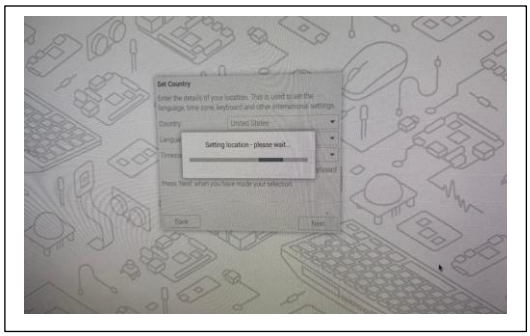


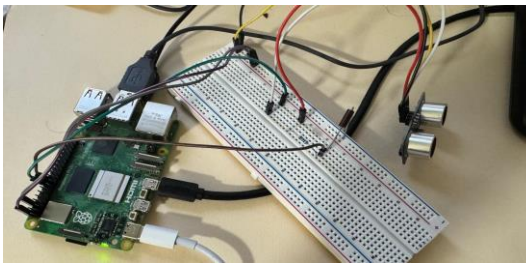
d.) Hardware Setup: Set up the hardware by connecting the Raspberry Pi with the ultrasonic sensor using a breadboard.



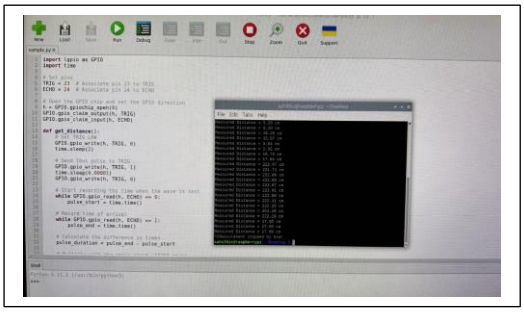


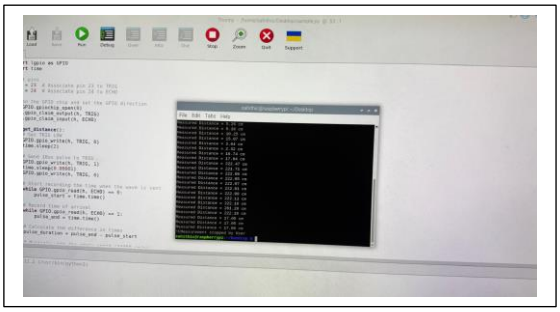




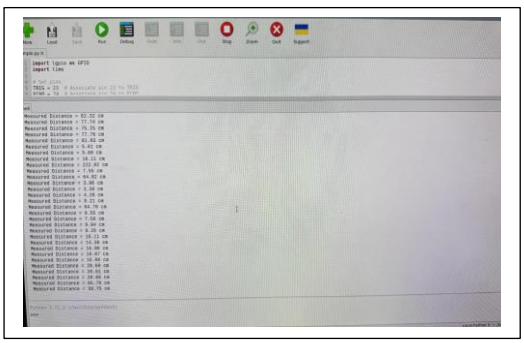


e.) Basic Sensor Functionality: Developed and ran a sample Python script to measure the distance between the sensor and an object for every 2 seconds.





f.) Verified that the distance data outputs correctly in the terminal, confirming the sensor's functionality.



2. Set Up Python Environment and Install Necessary Packages

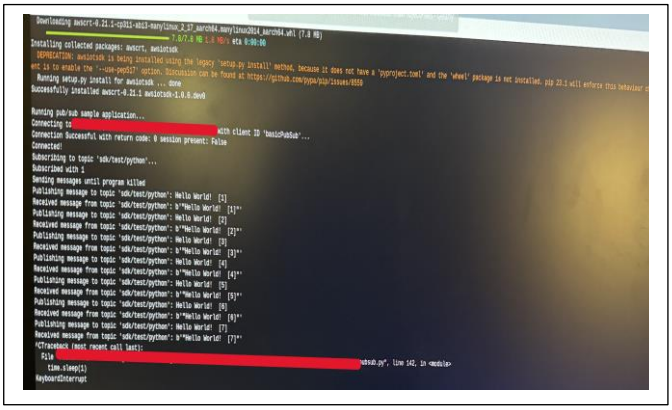
Installed necessary packages like,

paho-mqtt: Library for connecting and publishing messages to MQTT (used in AWS IoT Core).

boto3: AWS SDK for Python to enable Lambda and other AWS interactions.

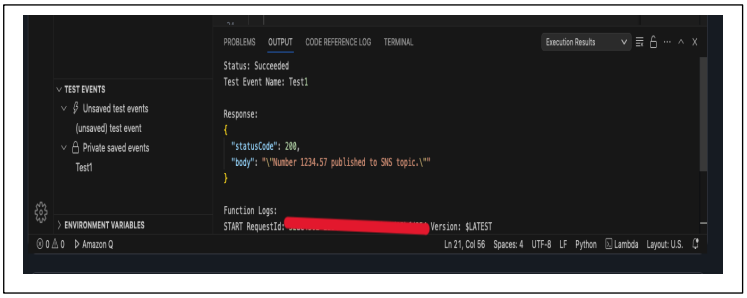
awsiotsdk: AWS IoT SDK to manage device communication.

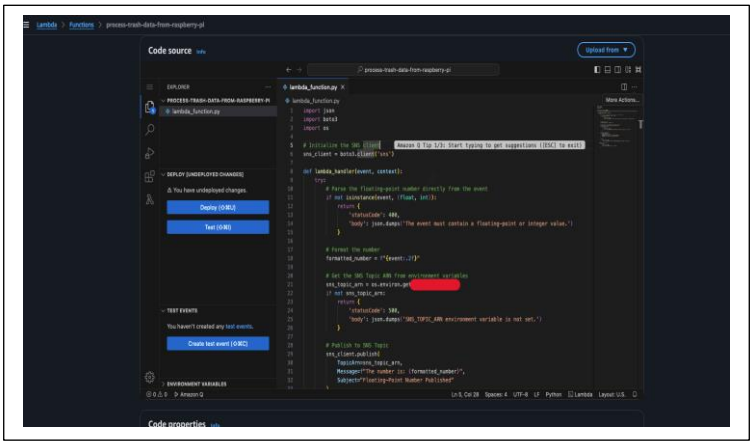
Since newer Raspberry Pi models may restricted direct package installations, I create a new Python environment, activated it and ran a sample script provided by AWS (named start.sh). This script automatically installed all necessary packages and dependencies.



3. Created a Python Script for Measuring Distance

Wrote a sample Python script to send random floating-point numbers to test the MQTT connection and message publishing to AWS IoT Core. This ensured the basic setup works correctly.





After setting Up the AWS IoT Device SDK by configuring Credentials and Connecting to AWS IoT Core, I obtained the necessary credentials from AWS IoT Core like,

a. Certificate to authenticate the device to AWS IoT Core.

b. Private Key which Secures the device's connection.

c. Root CA which Validates the AWS IoT Core server.

I attached an appropriate IoT policy to the device (the "thing") to grant permissions for publishing and subscribing to the MQTT topic (my-Topic).

Then, I configured the device to connect securely to AWS IoT Core using these credentials.

4. Writing the Sensor Distance Script

Then, I created a Python script named sensor_distance.py to measure the distance detected by the ultrasonic sensor. This script Initializes the GPIO pins,

a. To configure the GPIO chip and set the appropriate pin directions for the TRIG and ECHO pins.

b. Sends a pulse to the sensor

c. Sets the TRIG pin to LOW initially

d. Then, sends a precise 10-microsecond pulse to the TRIG pin to trigger the ultrasonic wave

e. Then, records the times by,

Starting the recording time when the wave is sent from the TRIG pin. then, records the time of arrival when the wave is received by the ECHO pin).

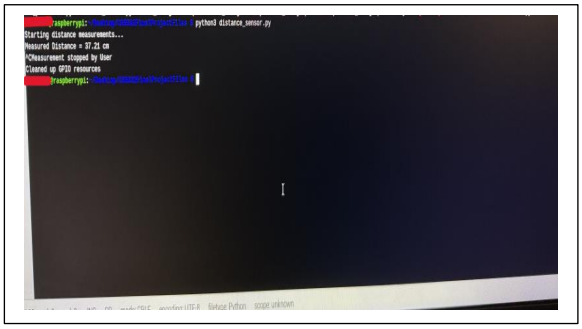
Now, Calculate the distance, by computing the time difference between sending and receiving the wave.

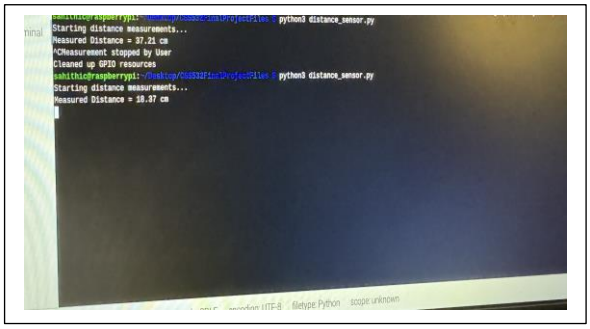
Formula used:

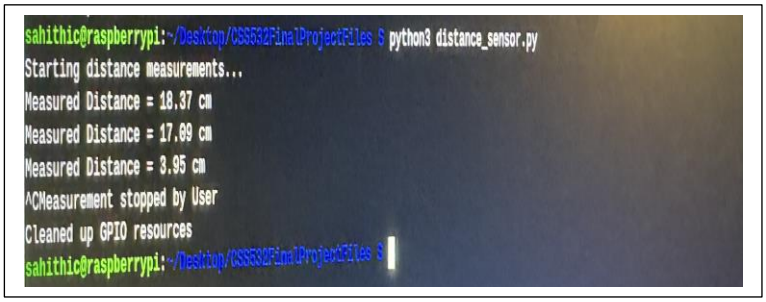
Distance (cm) = (Time Difference × 34300) / 2

(**Note -** The division by 2 accounts for the round trip (to the object and back)).

Then, I stored the calculated distance and the current timestamp in a JSON file, for subsequent processing. Then, repeated this process periodically by measuring the distance at regular intervals (every 5 minutes).



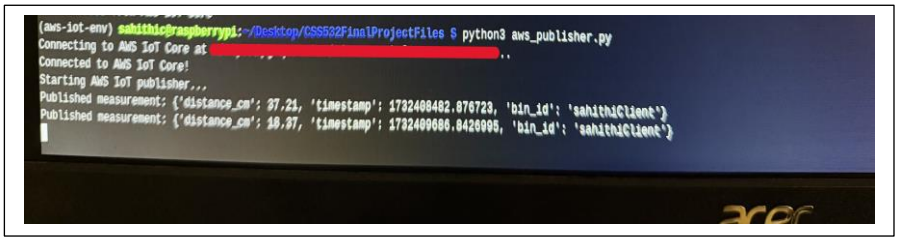




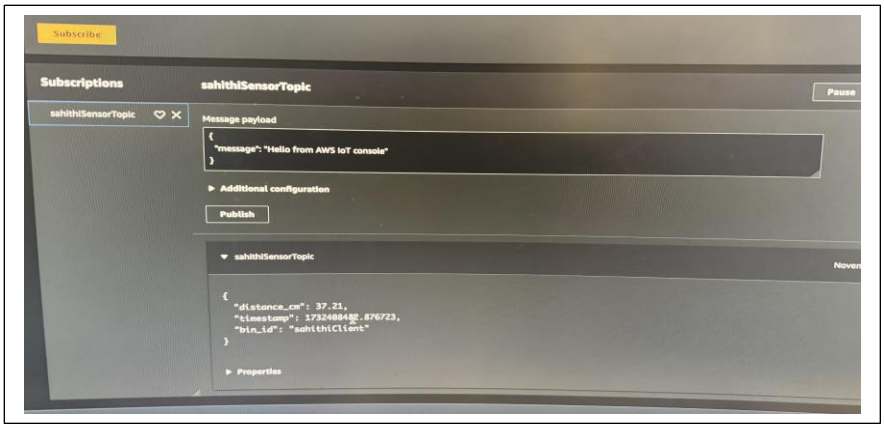
5. Writing the Publishing Script

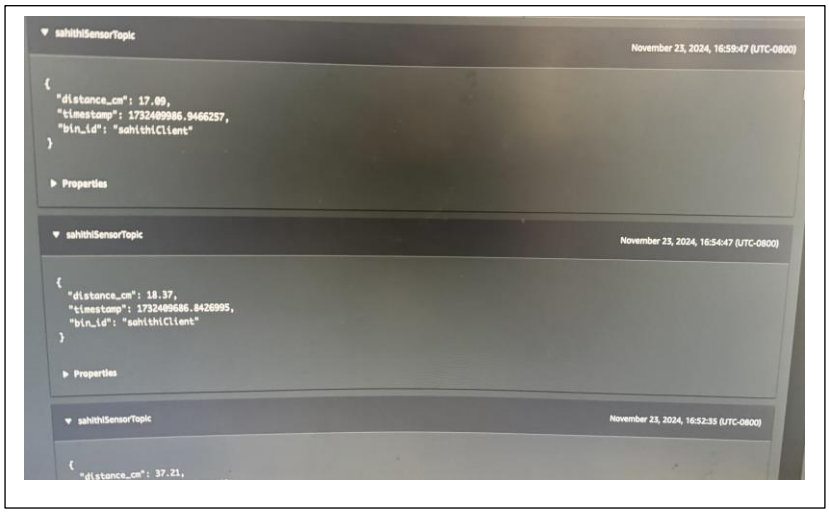
Now, I create a separate Python script named aws_publisher.py to handle data transmission to AWS IoT Core. This script reads the JSON file and

extracts the most recent distance and timestamp data and publishes it to AWS IoT Core. This data is sent to AWS lambda for further processing.



I confirmed that the data was received in the IoT console under the sensor-Topic, and appropriate alerts were triggered.





All notifications were successfully delivered to the email subscribers, ensuring seamless integration and functionality across the system.

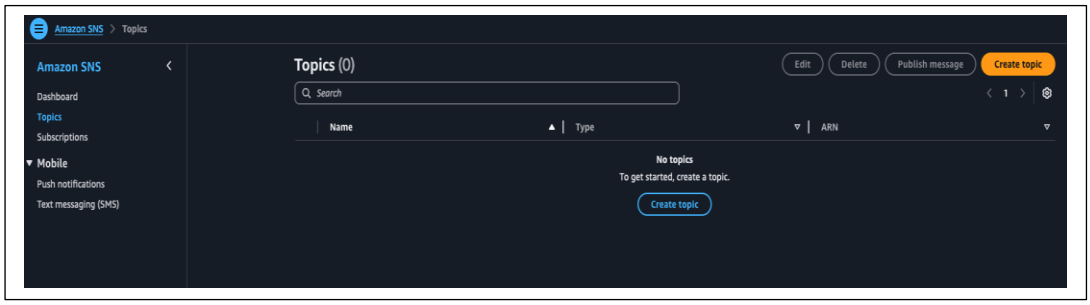
6. Setting up AWS IOT core: (already done during HW1 for CSS532)

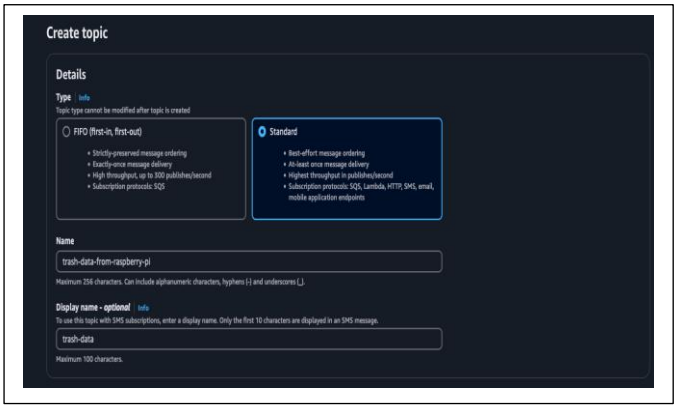
(Note: Already done creating AWS account, setting up thing for HW1)

I created a new policy specifically for subscribing to a topic named “mySensorTopic”. Using the same Python script for HW1, I successfully subscribed to the topic by executing the script through the terminal. After configuring the policy, I used the AWS IoT console to subscribe to the same topic and verify that my device could successfully publish messages to it. This validated the functionality of both publishing and subscribing, ensuring seamless communication between my device and AWS IoT.

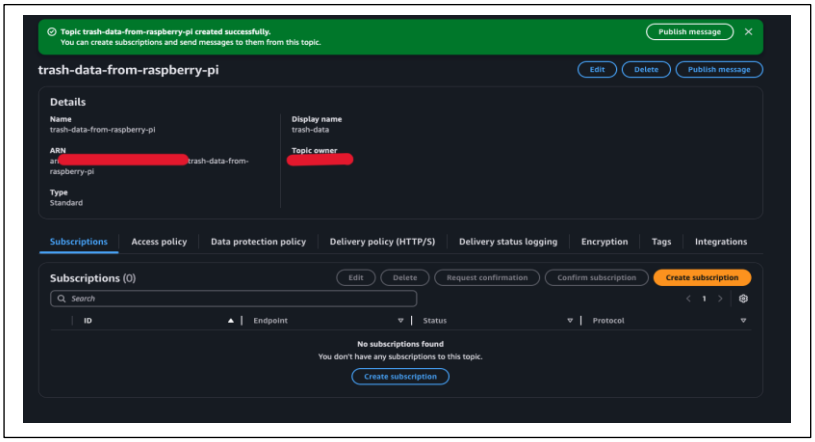
7. Setting Up Amazon SNS (Simple Notification Service):

I navigated to the SNS Console by opening the AWS Management Console and accessing the SNS service. Once there, I created a topic by selecting "Create Topic" and filling in the necessary details, such as the topic name and type (e.g., Standard).

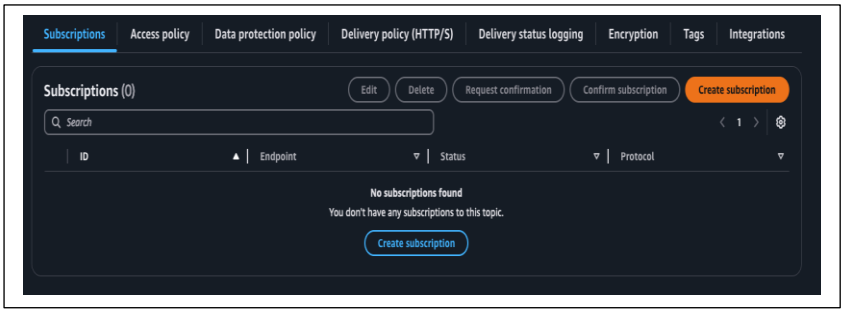


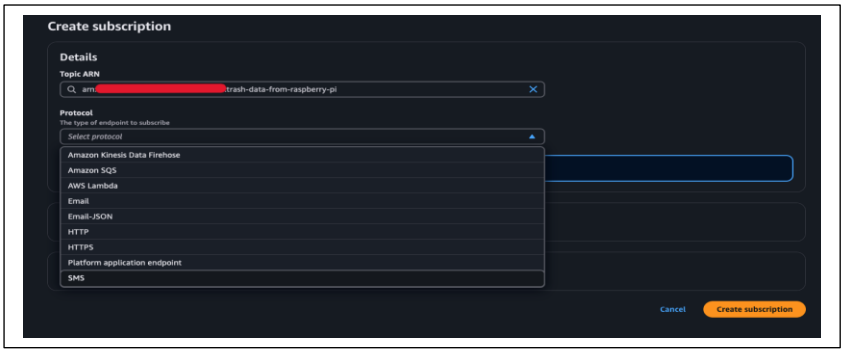


I assigned a unique name to the topic, such as trash-data-from-raspberry-pi, which reflects its purpose, and completed the topic creation process. The newly created topic was displayed in the list of available topics.

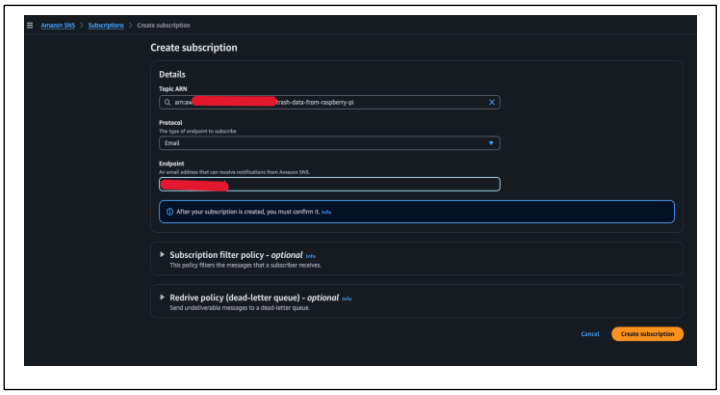


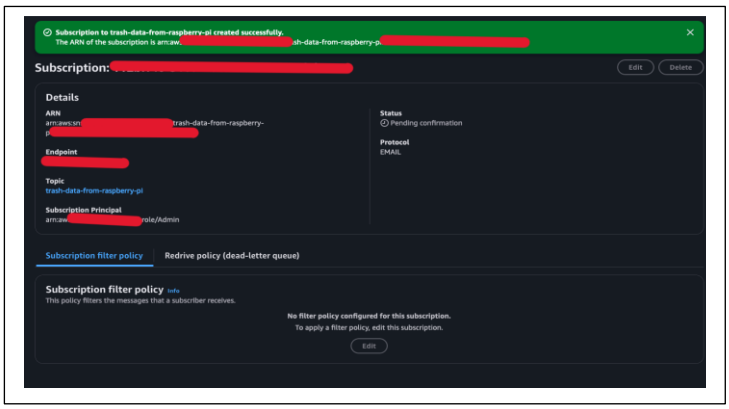
Next, I went to the Subscriptions tab under the created topic to set up a new subscription.



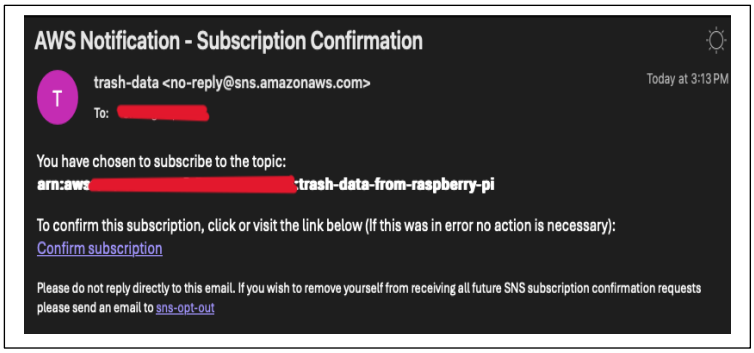


I selected the Email protocol as the method of notification. In the Endpoint field, I entered the recipient's email address and confirmed the creation of the subscription.





An email was automatically sent to the provided address for confirmation. Upon opening the email, I clicked the confirmation link, which validated the subscription and updated its status to Confirmed in the SNS console.



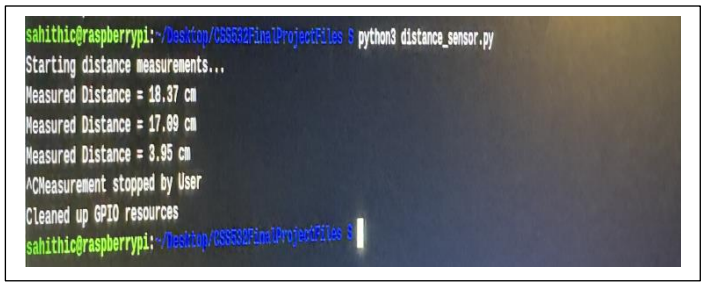
8. Test the sensor’s accuracy:

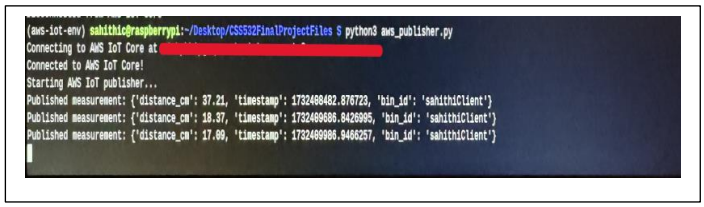
Now, to test the sensor’s accuracy, I ran the sensor script (sensor_distance.py), to verify that the distance is accurately measured and recorded in the JSON file. Also, cross-check the calculated distance with a known measurement to ensure reliability.

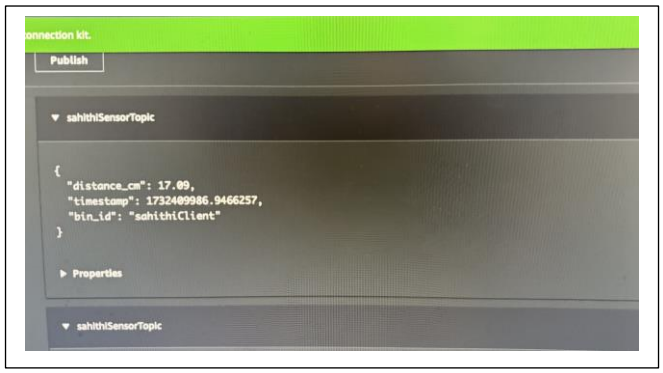
Then, I ran the publishing script (aws_publisher.py) to verify that the data from the JSON file is correctly transmitted to AWS IoT Core.

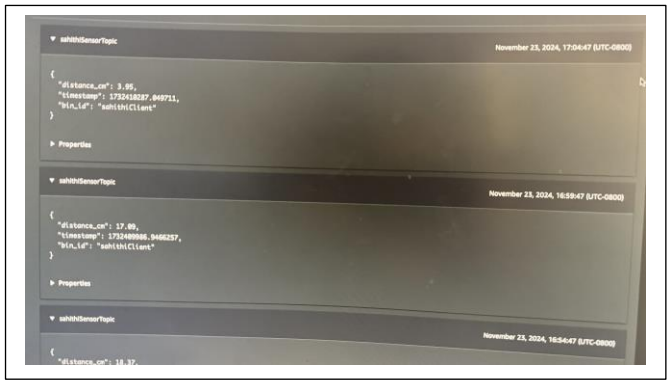
I used the MQTT Test Client in the AWS IoT console to subscribe to myTopic and confirmed the receipt of messages.

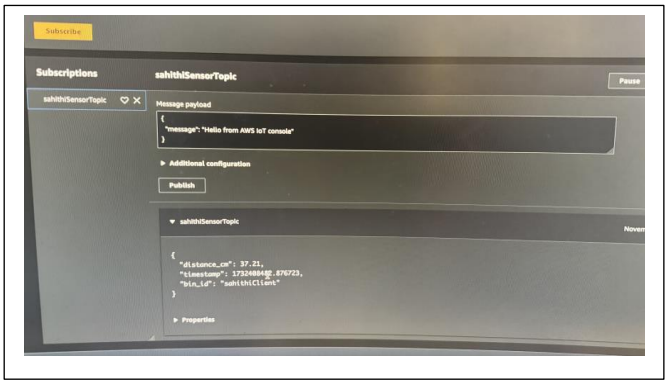
Then, I Continued testing, to observe the consistency and accuracy of distance measurements over multiple cycles. This, ensured that messages appear in AWS IoT Core at 5-minute intervals as expected.





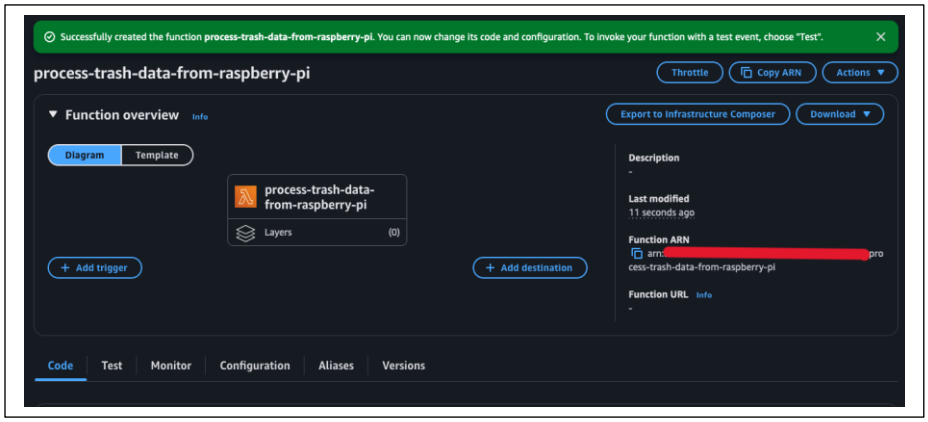




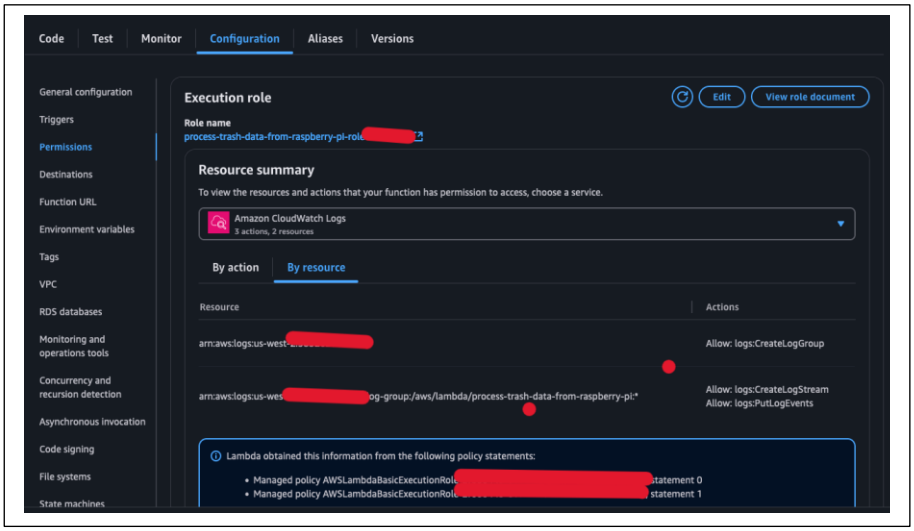


9. Setting Up the AWS Lambda Function:

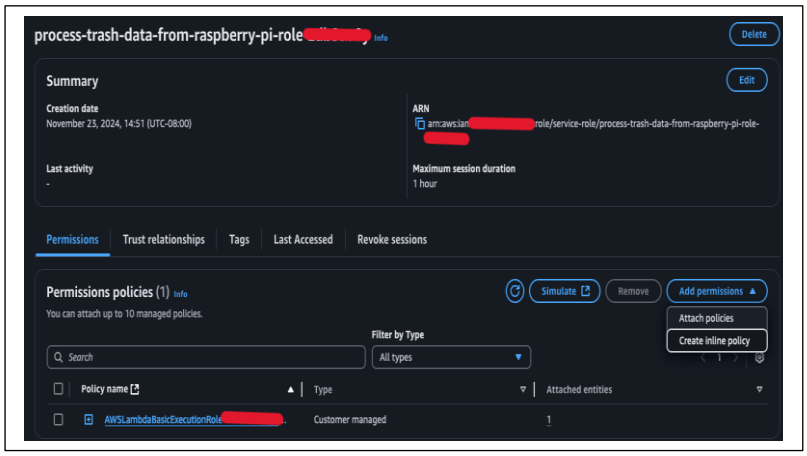
Following this, I navigated to the AWS Lambda Console to create a new Lambda function. I chose "Create Function", selected Author from Scratch, and provided a descriptive name for the function, such as process-trash-data-from-raspberry-pi. I specified the runtime environment as Python 3.13 and assigned an appropriate IAM role with basic execution permissions to the function.



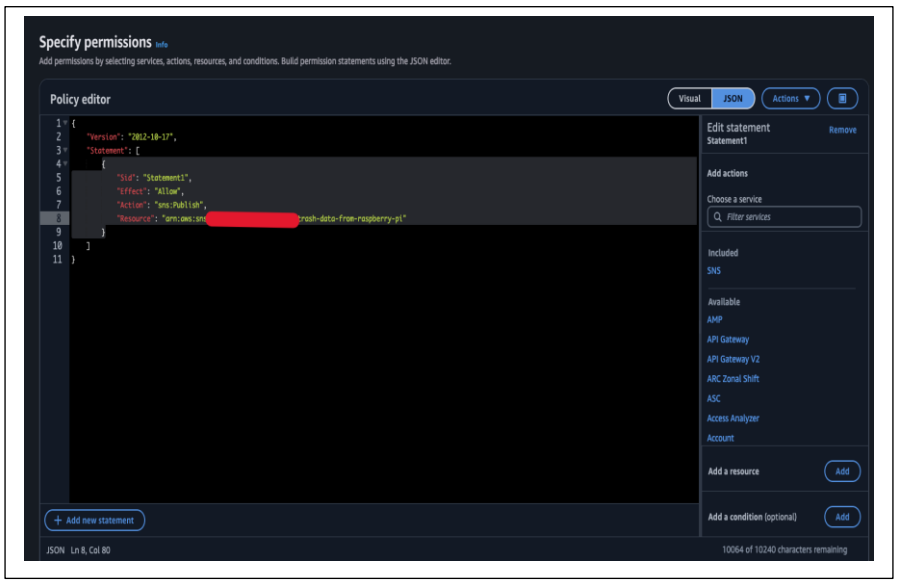
After creating the function, I updated its permissions by navigating to the Configuration tab in the Lambda Console and selecting the IAM role linked to the function.

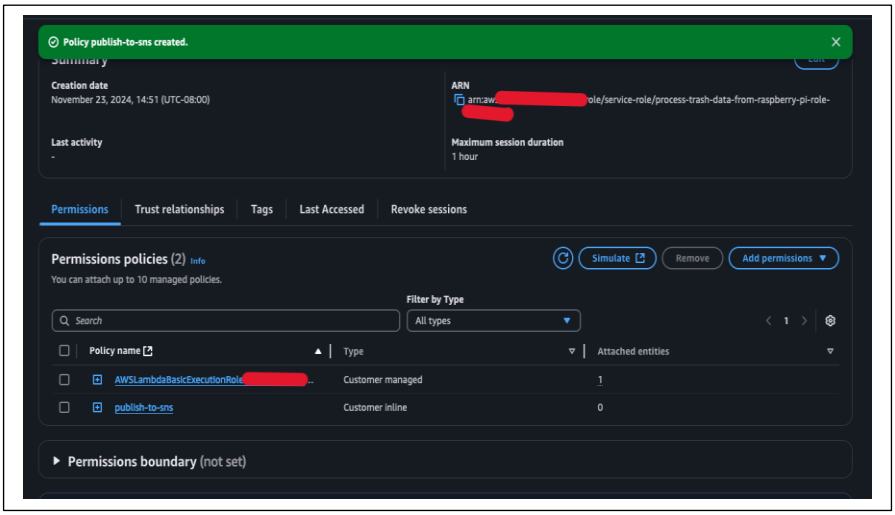


Within the IAM policy editor, I added an inline policy to grant the Lambda function the necessary permissions to publish messages to the SNS topic.

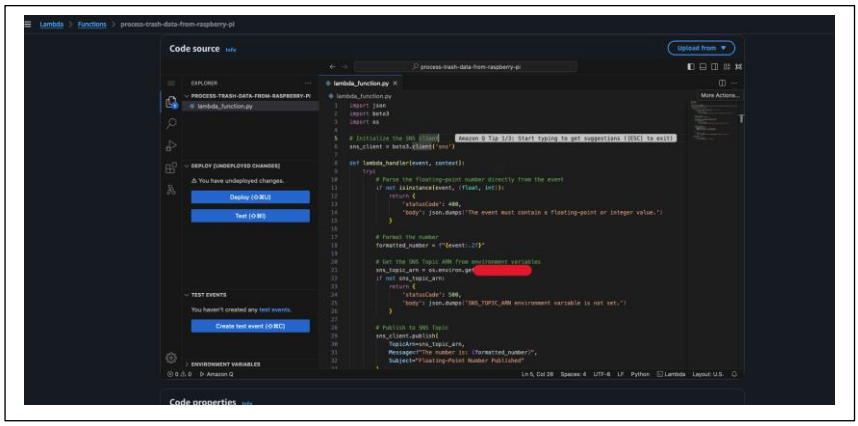


This policy included the sns:Publish action and specified the ARN of the SNS topic created earlier. Once the policy was saved, it was successfully attached to the Lambda function.

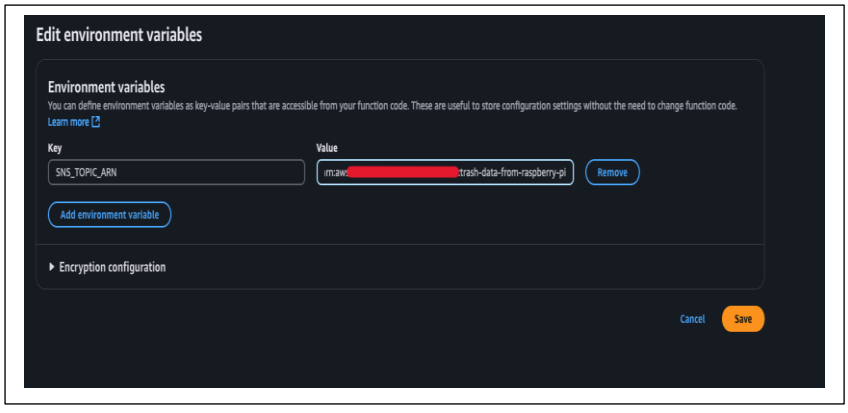




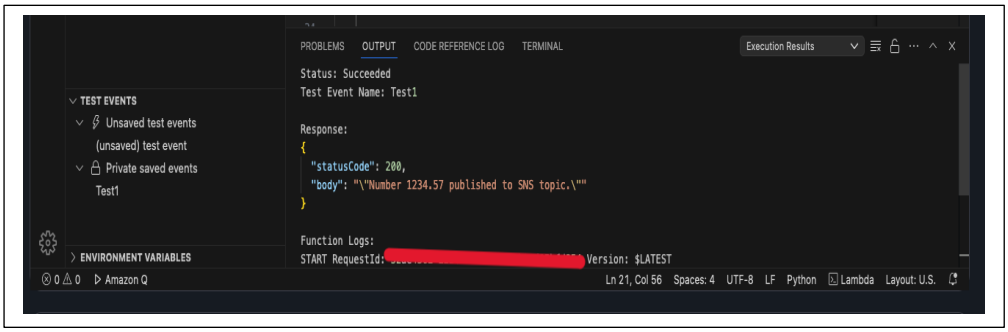
I then wrote the Lambda function's code, which included logic to publish alert messages to the SNS topic.



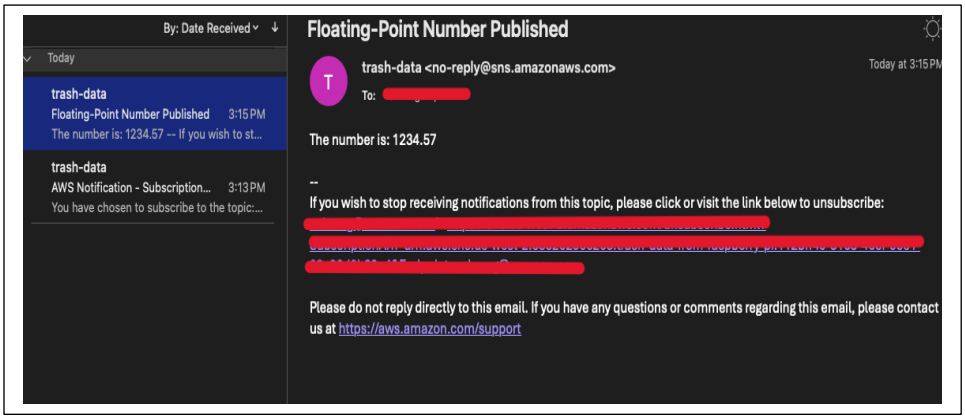
To make the function dynamic and maintainable, I used environment variables to store the SNS topic ARN.



After deploying the code, I tested the Lambda function using the Test option available in the Lambda console. The function executed successfully, sending messages to the SNS topic.

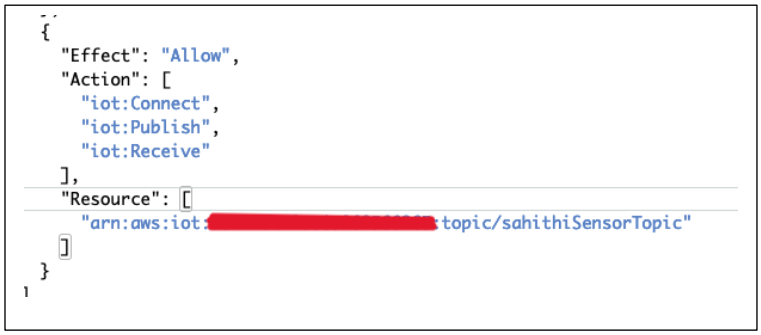


I verified the test results by checking that an email notification was received at the subscribed address.

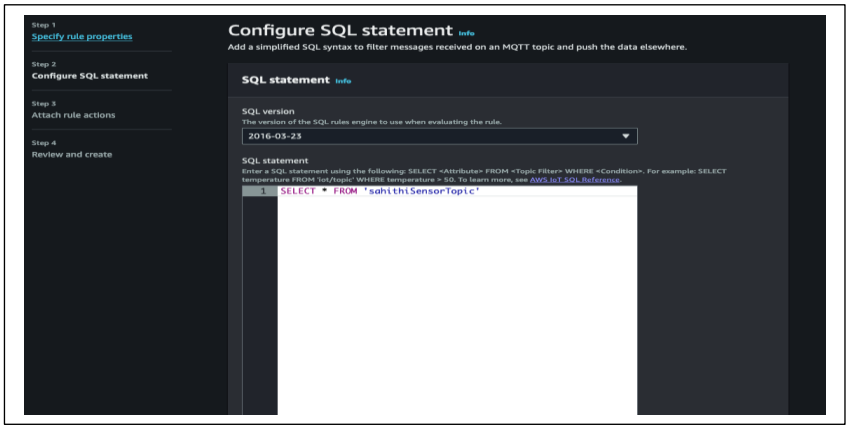


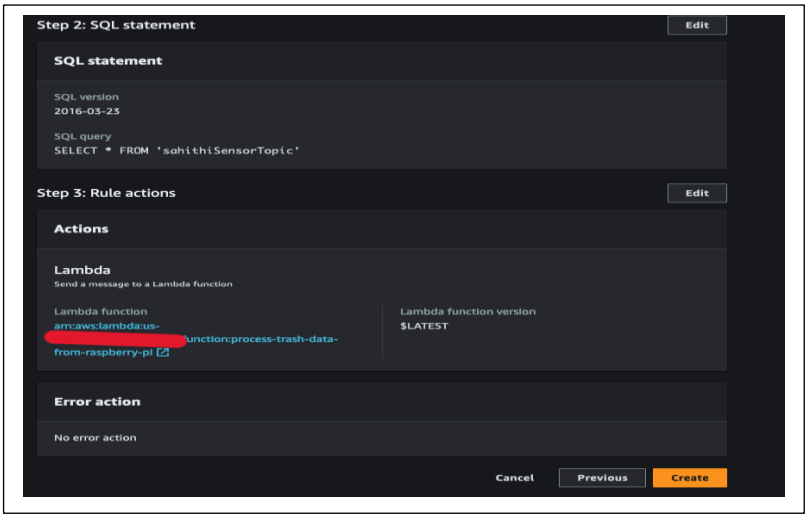
10. Setting Up AWS IoT Core Rules:

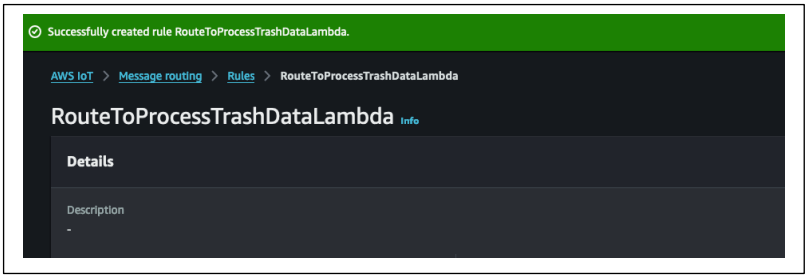
With SNS and Lambda configured, I proceeded to the AWS IoT Core Console to set up a rule for routing messages from the IoT topic to the Lambda function.



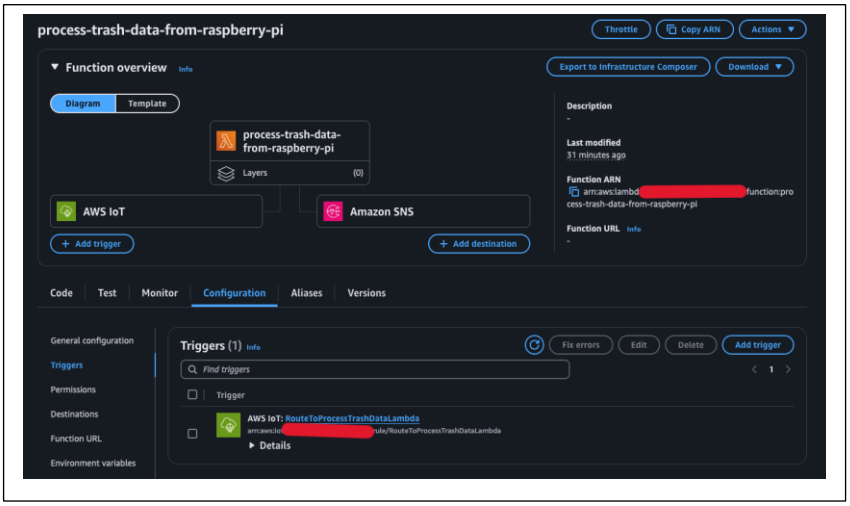
I defined a SQL statement within the IoT Core rules engine to filter and route data. For instance, I wrote a statement to select all data from the “mysensorTopic”.



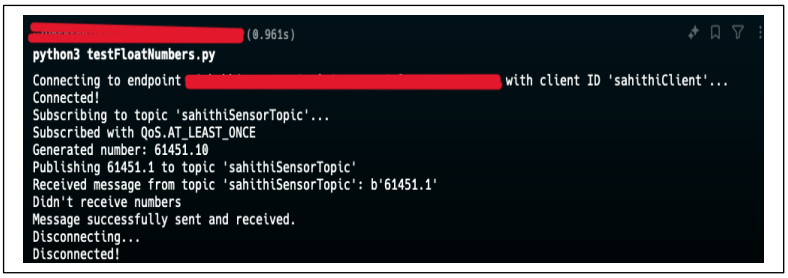




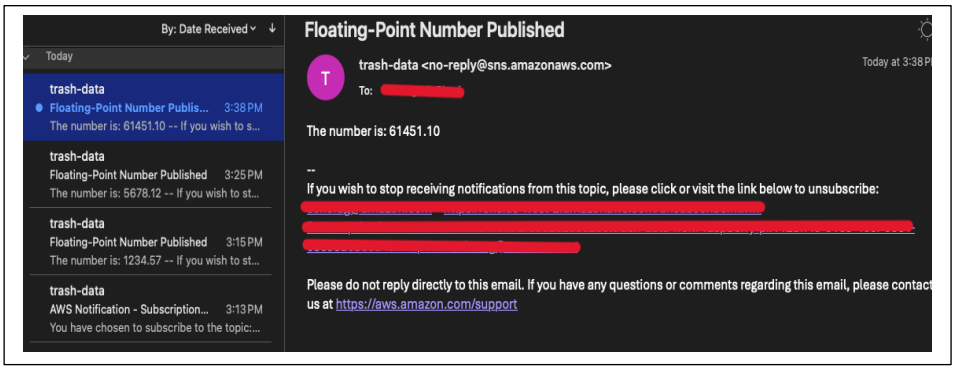
I created a new IoT rule, assigned it a descriptive name “RouteToProcessTrashDataLambda” and linked the Lambda function as the action to execute when the rule was triggered.



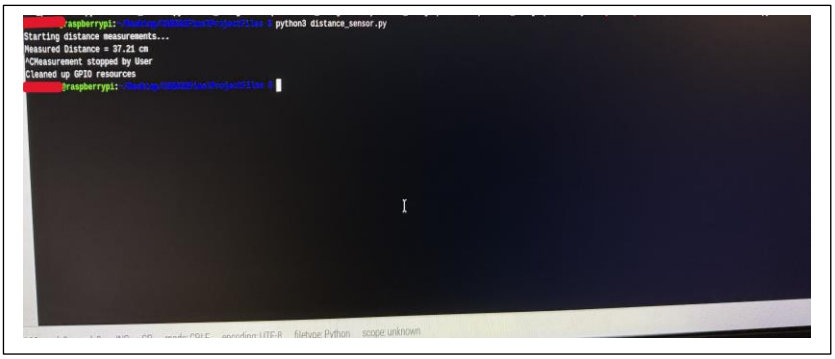
After successfully creating the IoT rule, I tested the setup by publishing test data to the Topic using a Python script.



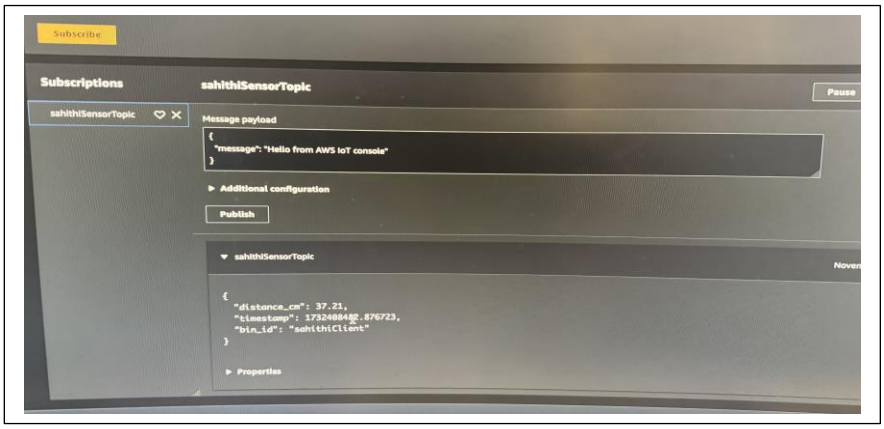
I verified that the IoT Core rule routed the data to the Lambda function and triggered an SNS notification. Additionally, I validated that all components, including email notifications, worked as intended by monitoring their responses in real time.

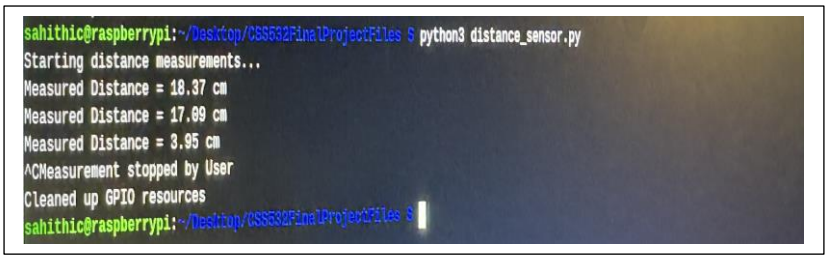


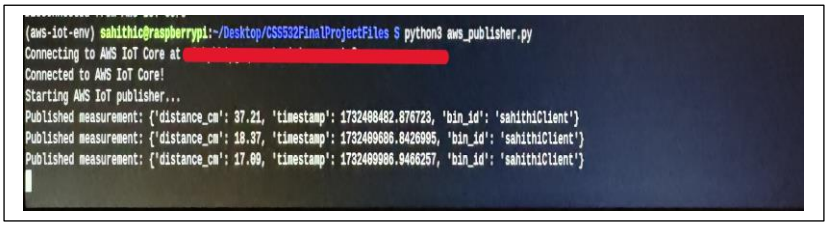
Finally, I ran the original python script, starting with publishing real sensor data to AWS IoT Core from a Python script.

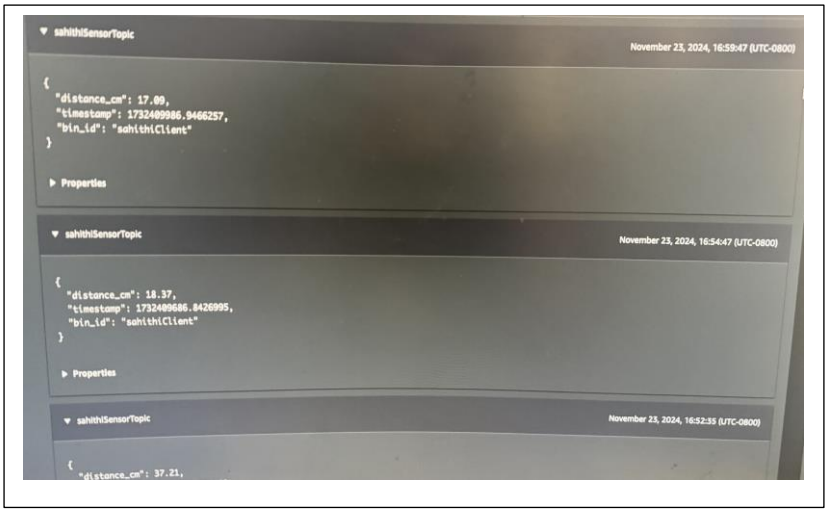


I confirmed that the data was received in the IoT console under the sensor-Topic, and appropriate alerts were triggered.





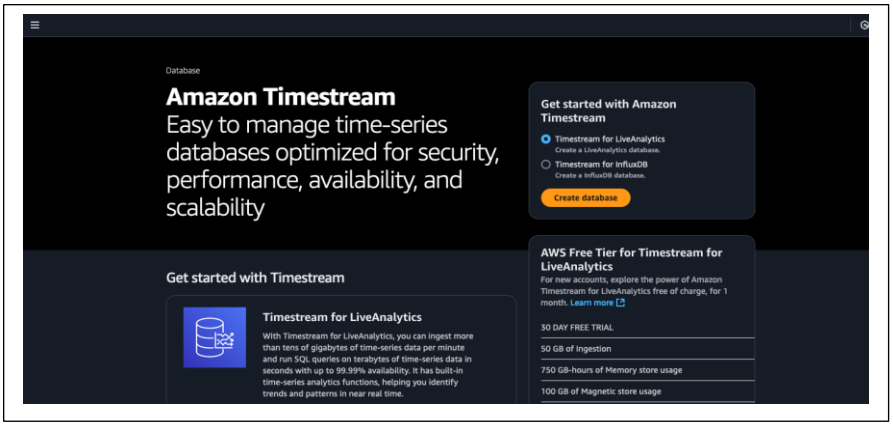




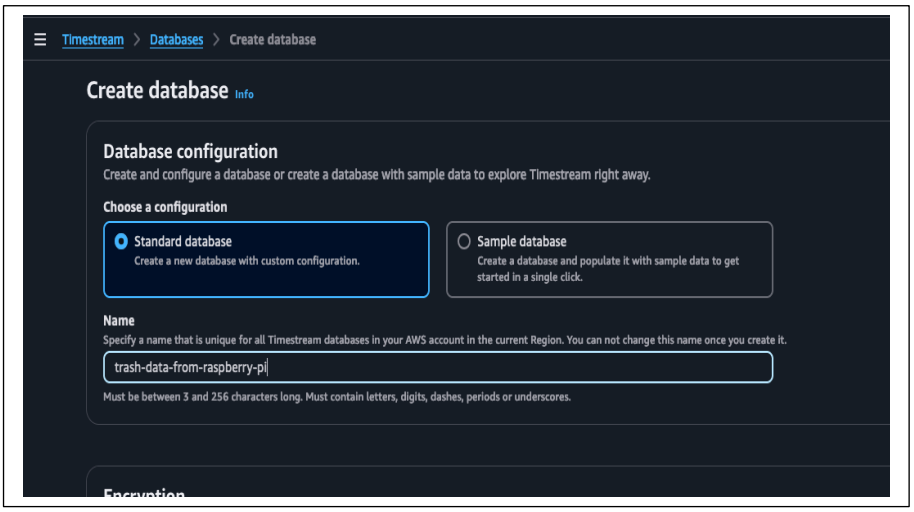
All notifications were successfully delivered to the email subscribers, ensuring seamless integration and functionality across the system.

11. Setting up TimeStream database:

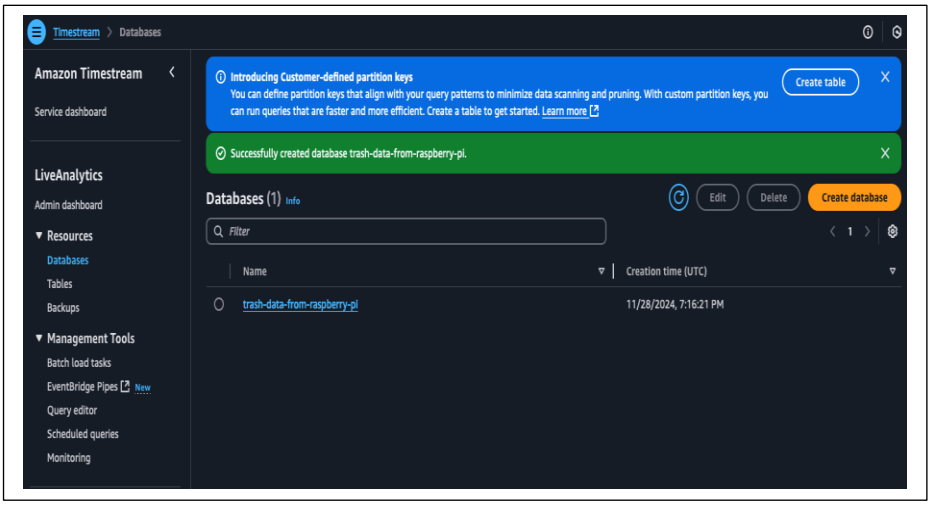
To set up and utilize Amazon Timestream for real-time analytics, I began by navigating to the Timestream Console on the AWS Management Console.



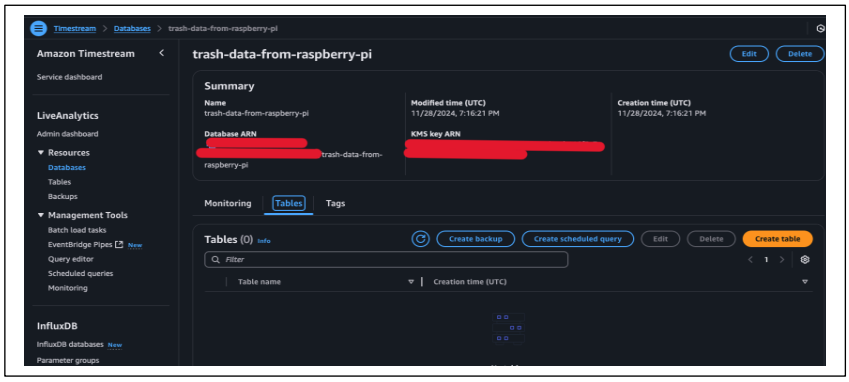
I selected the option to create a new database, opting for the "Standard database" type to suit the project's needs. I named it as "trash-data-from-raspberry-pi".



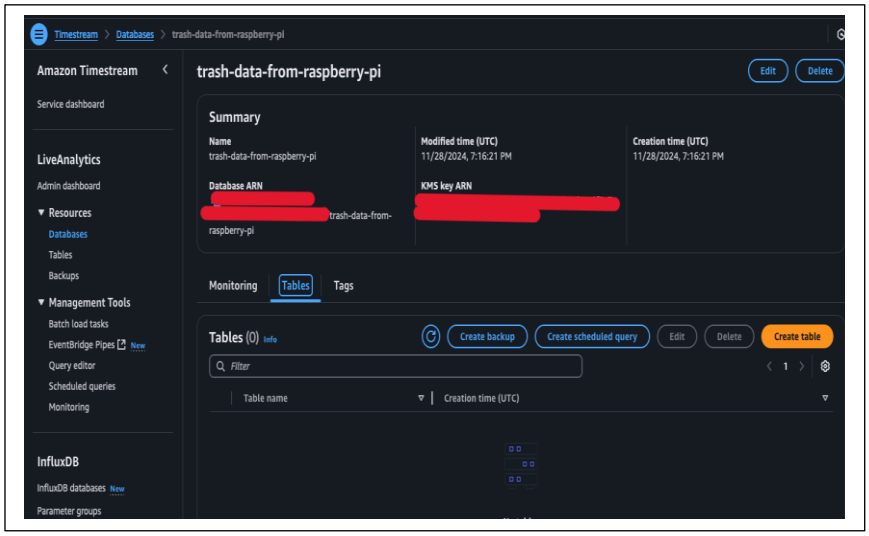
Upon entering the necessary details, I finalized the setup, receiving a confirmation message indicating the successful creation of the database.

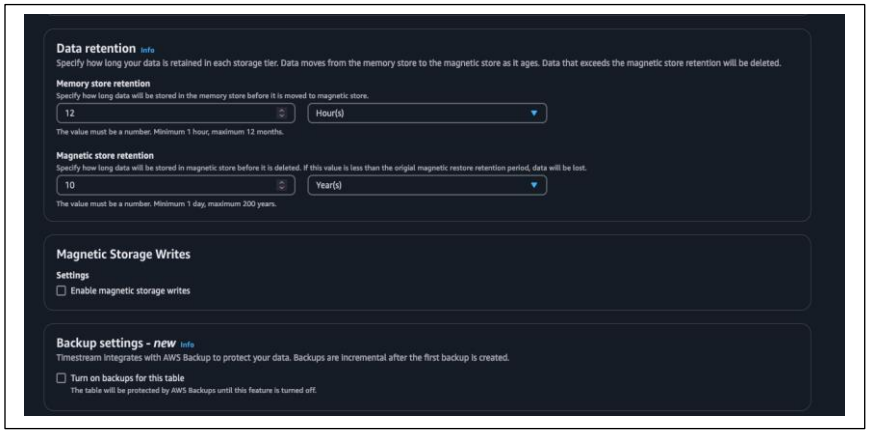


After the database creation, I navigated to the database's page to view its details. In the Tables tab, I initially observed that no tables had been created yet, confirming the need for further configuration.

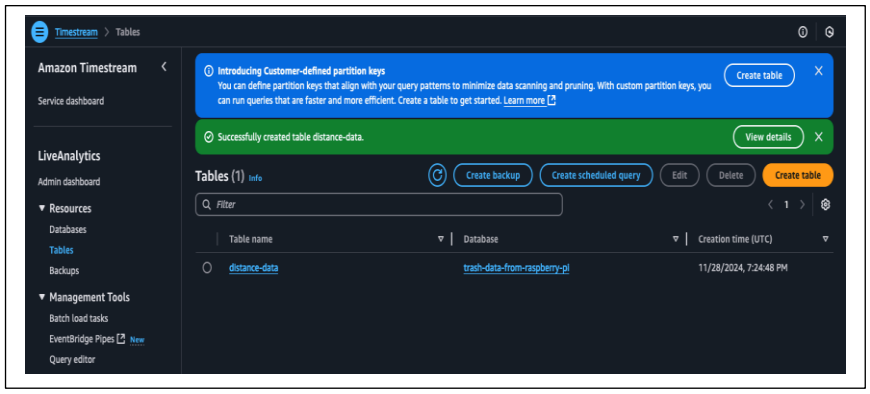


To proceed, I created a new table by selecting "Create table," and named it as "distance-data" to clearly represent the data it would store. I chose the default settings, including default partitioning, as they are well-suited for the typical use cases of this project.

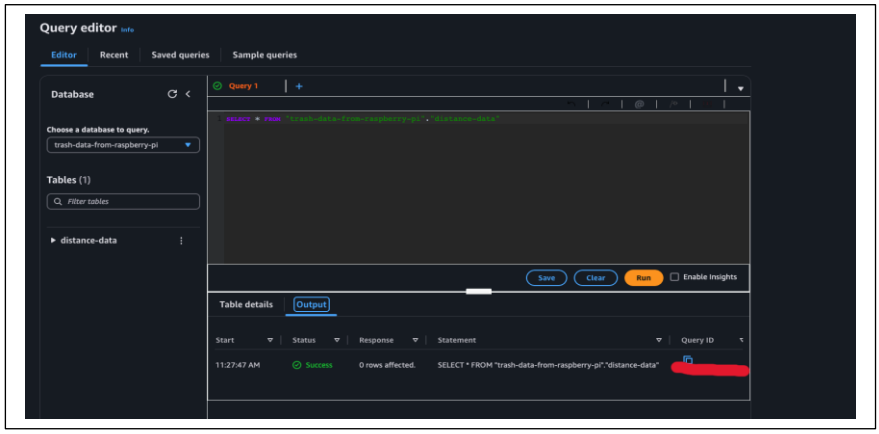


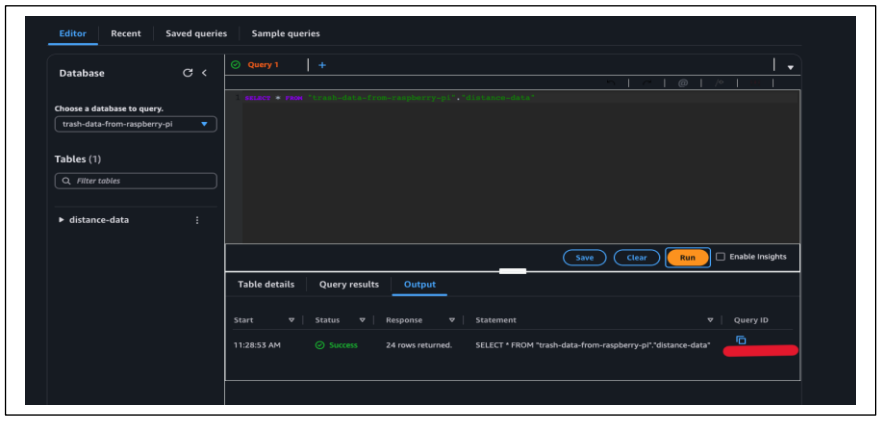


The table creation process was completed, and the system confirmed the successful creation of the table.

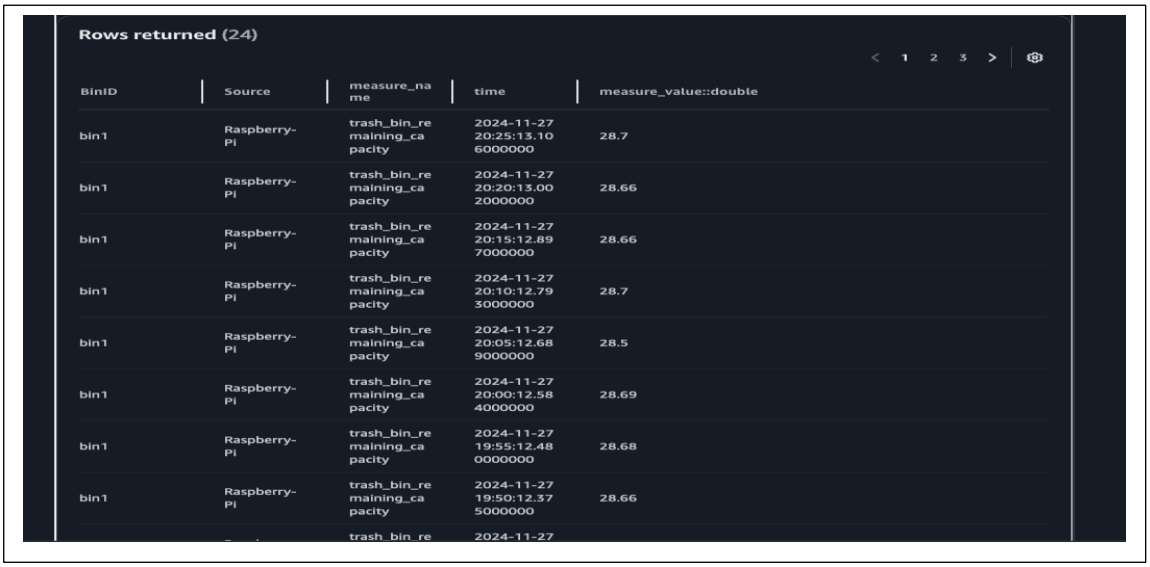


Once the Raspberry Pi started collecting waste level data using the ultrasonic sensor and publishing it to AWS IoT Core, this data was stored in the Timestream database. I wrote a SQL query and returned to the Query Editor and ran the query, before and after collecting the data from raspberry-pi.





The results displayed records with details such as BinID, source, measure_name, timestamp and measure_value(distance_cm), verifying that the data was being collected and stored accurately.



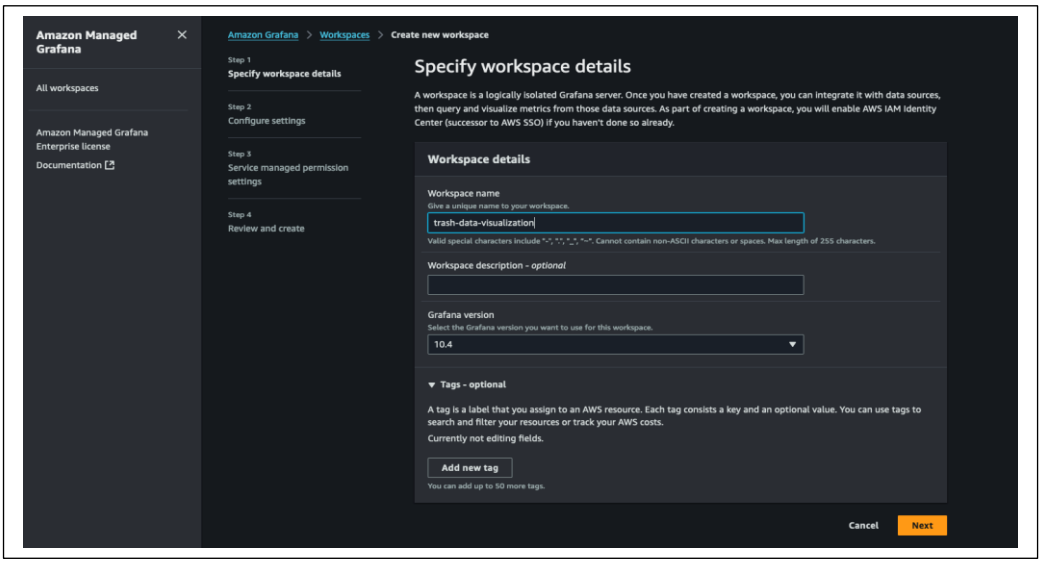
Rows returned (24)

BinID	Source	measure_name	time	measure_value::double
bin1	Raspberry-Pi	trash_bin_remaining_capacity	2024-11-27 20:25:13.106000000	28.7
bin1	Raspberry-Pi	trash_bin_remaining_capacity	2024-11-27 20:20:13.002000000	28.66
bin1	Raspberry-Pi	trash_bin_remaining_capacity	2024-11-27 20:15:12.897000000	28.66
bin1	Raspberry-Pi	trash_bin_remaining_capacity	2024-11-27 20:10:12.793000000	28.7
bin1	Raspberry-Pi	trash_bin_remaining_capacity	2024-11-27 20:05:12.689000000	28.5
bin1	Raspberry-Pi	trash_bin_remaining_capacity	2024-11-27 20:00:12.584000000	28.69
bin1	Raspberry-Pi	trash_bin_remaining_capacity	2024-11-27 19:55:12.480000000	28.68
bin1	Raspberry-Pi	trash_bin_remaining_capacity	2024-11-27 19:50:12.375000000	28.66
		trash_bin_re	2024-11-27	

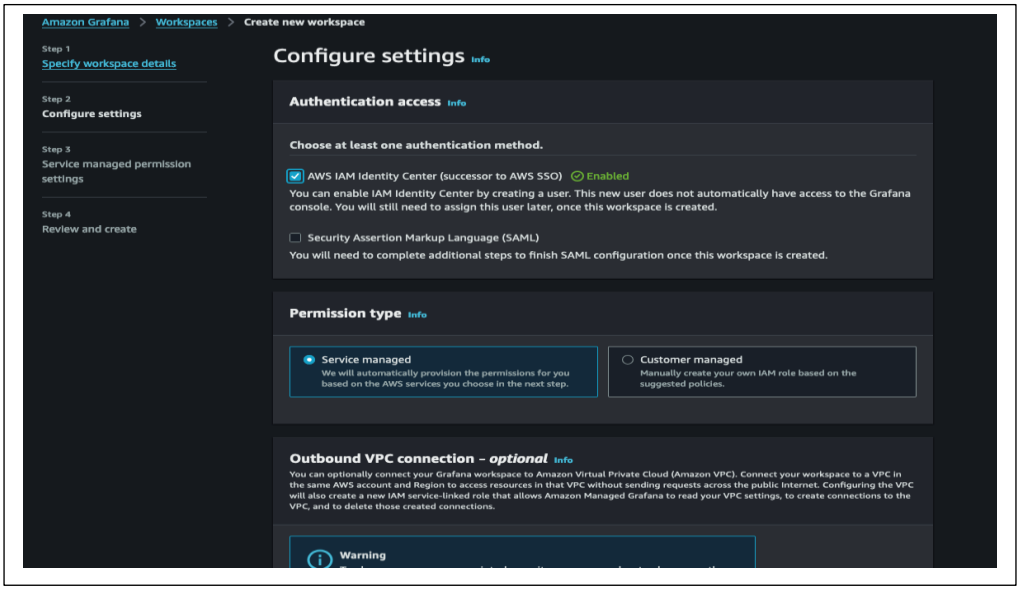
This process validated the successful setup and integration of Amazon Timestream for live analytics, enabling real-time monitoring and analysis of waste bin levels.

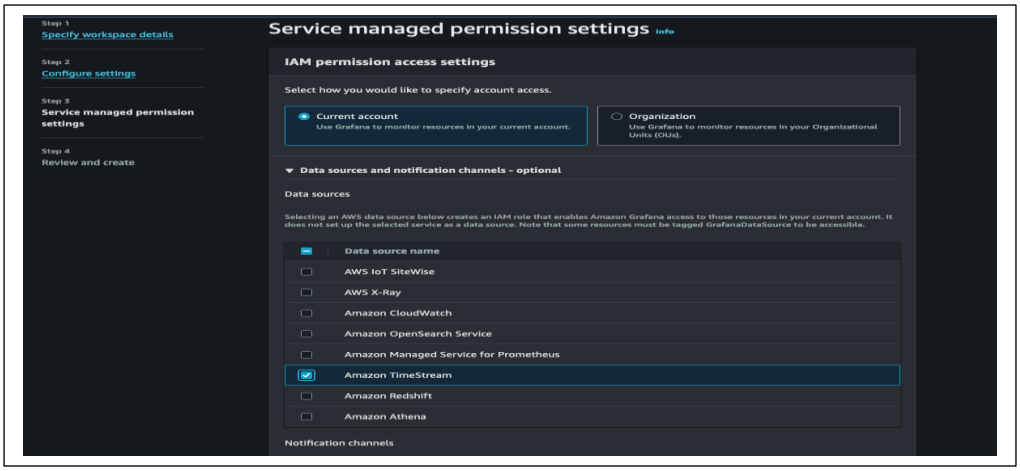
12. Setting up Graphana dashboard:

To set up and utilize Grafana for visualizing data stored in Amazon Timestream, I started by creating a new Grafana workspace in the AWS Management Console.

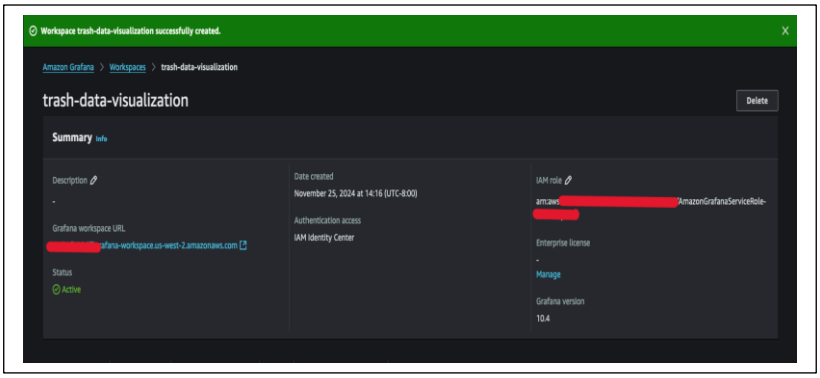


After navigating to the Grafana service, I selected the option to create a workspace, configuring the necessary workspace settings to align with the project's requirements and selected time-Stream as data source for Graphana dashboard.

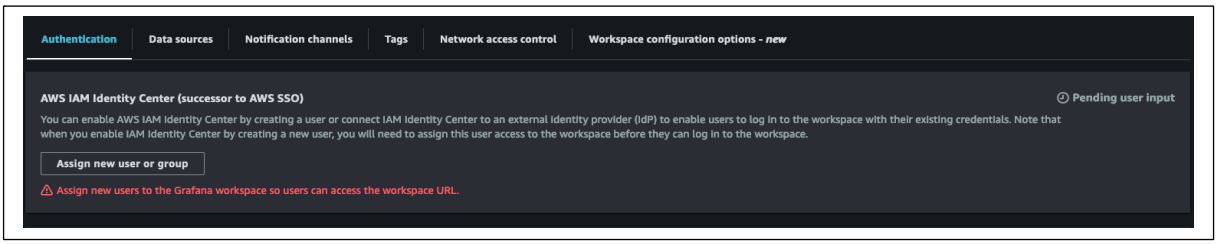




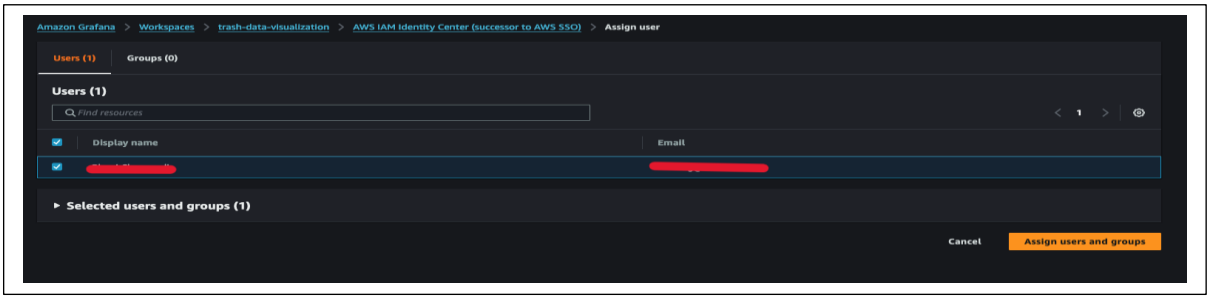
Once the settings were finalized, the workspace creation was completed successfully, providing a dedicated URL for accessing the Grafana dashboard.

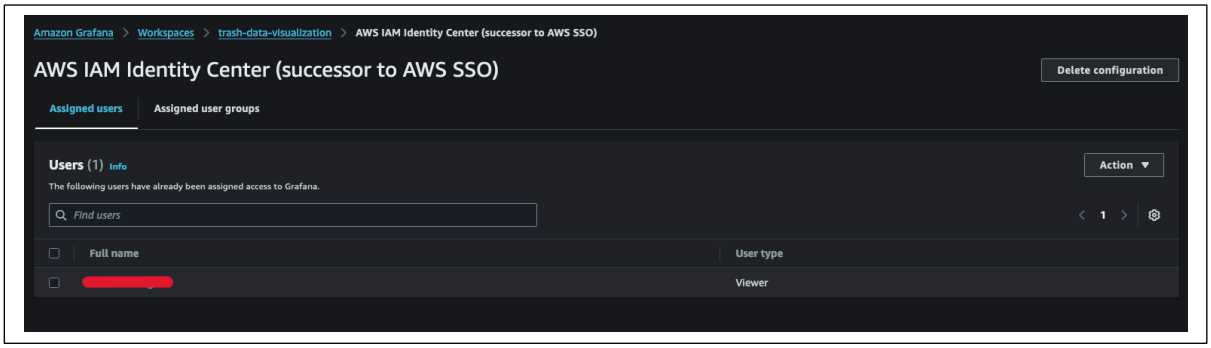


Initially, no users were added to the workspace.

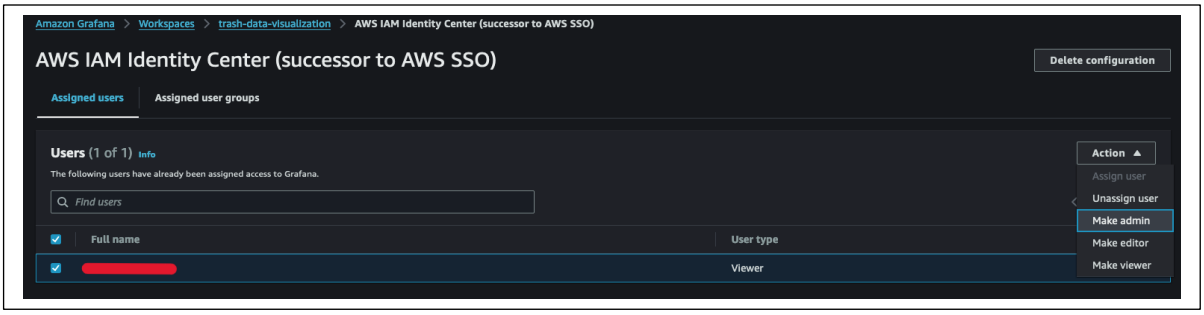


I navigated to the user management section and added a new user with the Viewer role.

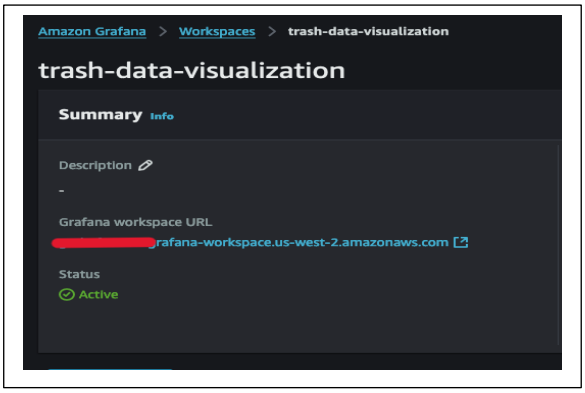


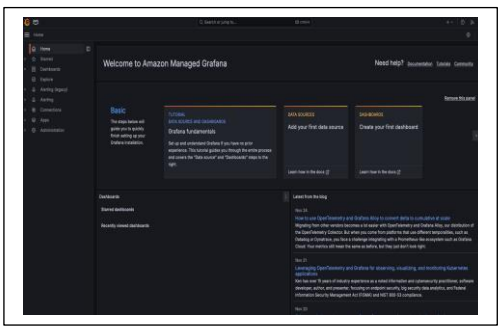


After verifying the user's addition, I updated the role to Admin, which enabled me to make changes to the workspace. This update was reflected in the user management interface, confirming the role change.

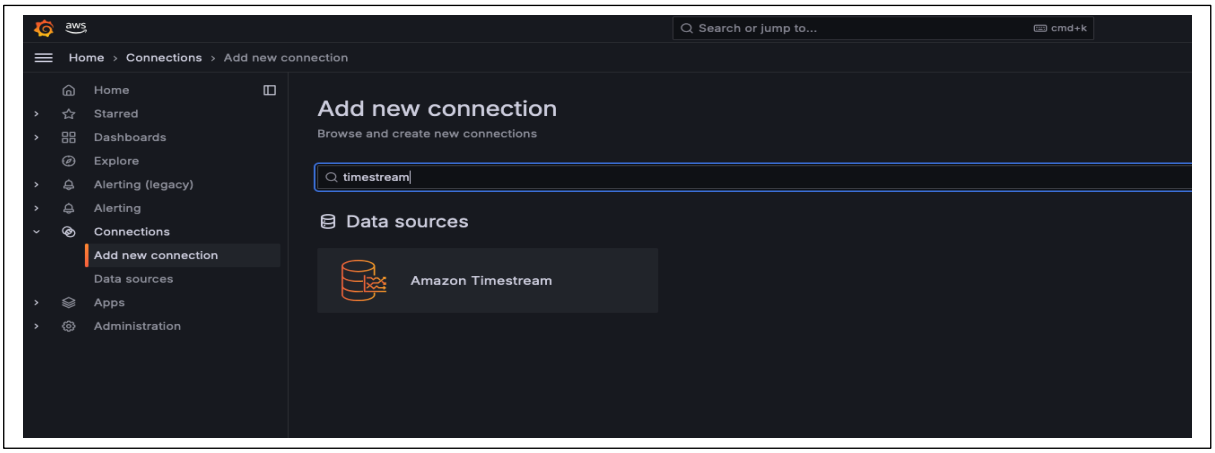


Next, I accessed the Grafana workspace URL and signed in using the designated username and password. After a successful login, the Grafana homepage became visible, marking the completion of the setup phase.

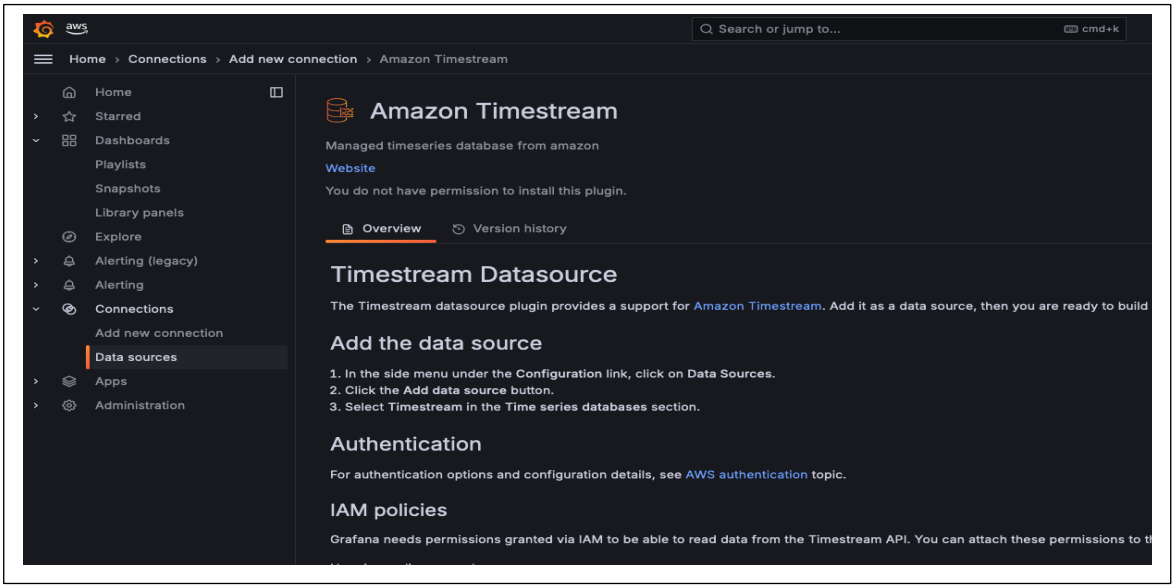


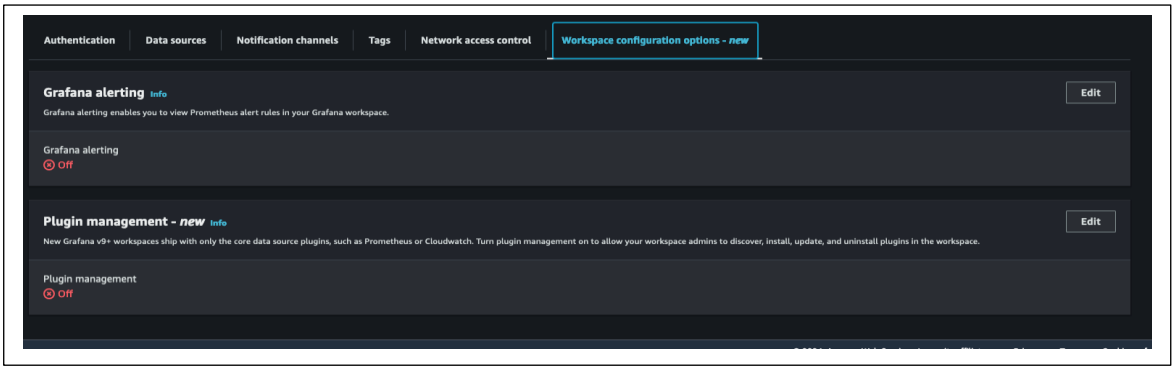


I proceeded to add Amazon Timestream as the data source for the workspace.

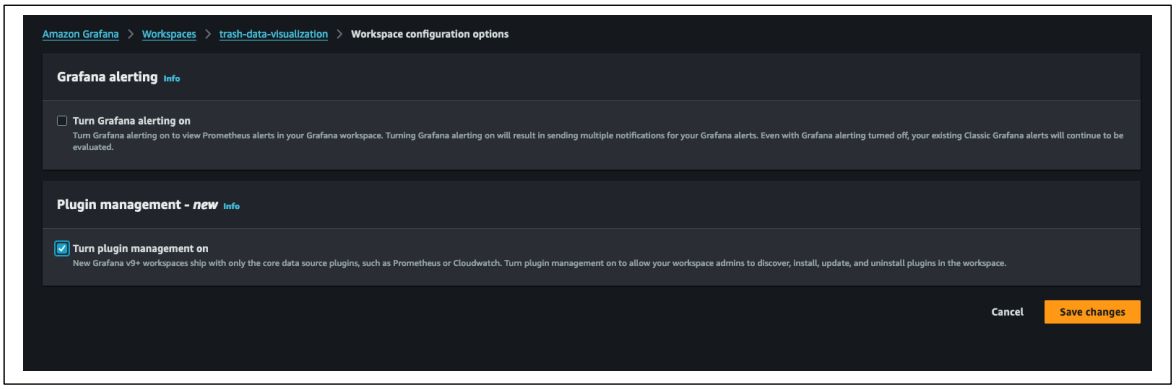


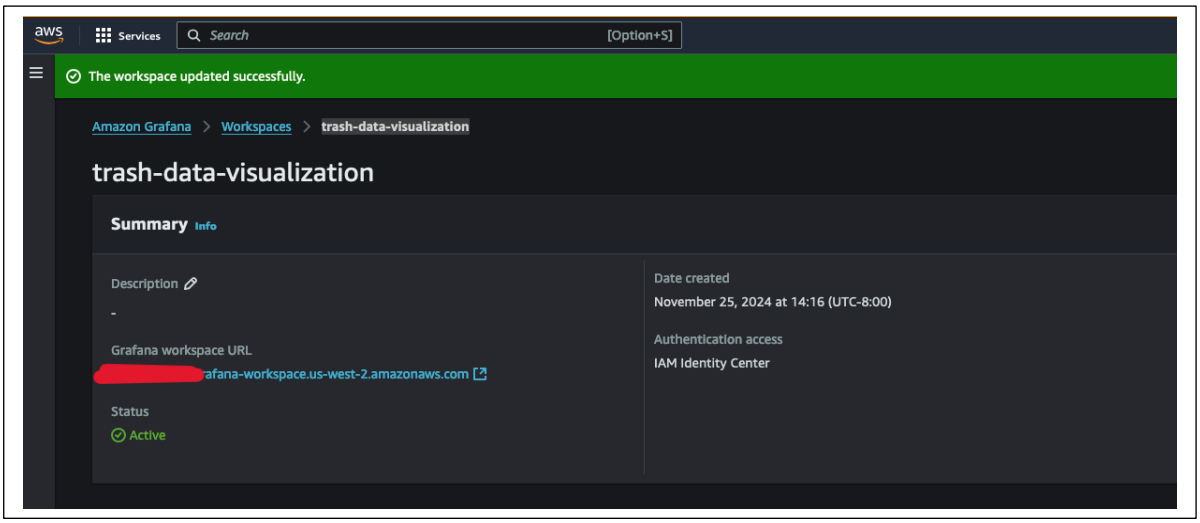
However, I encountered a restriction—by default, the workspace lacked permission to manage plugins.



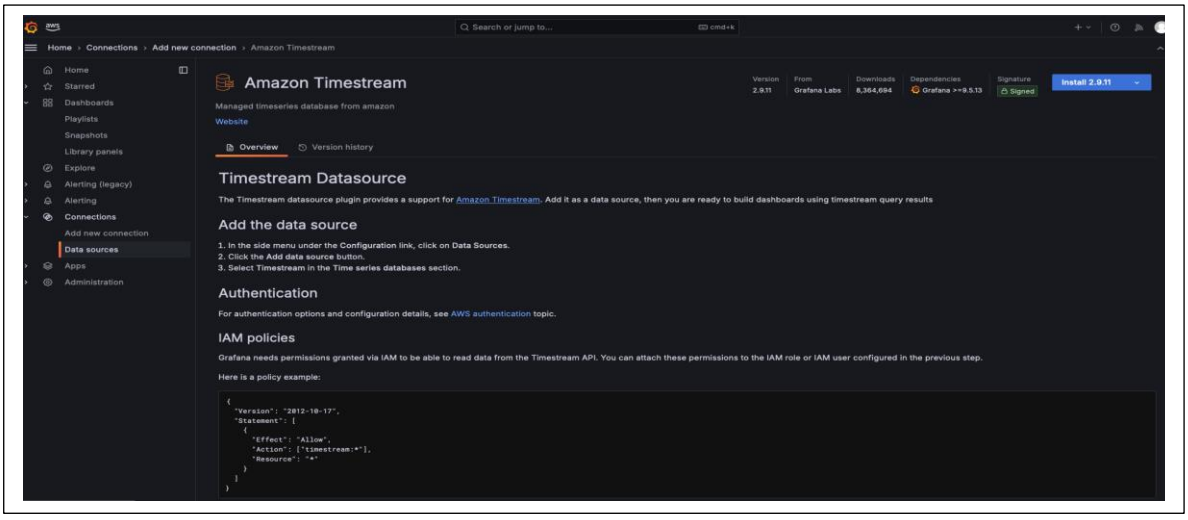


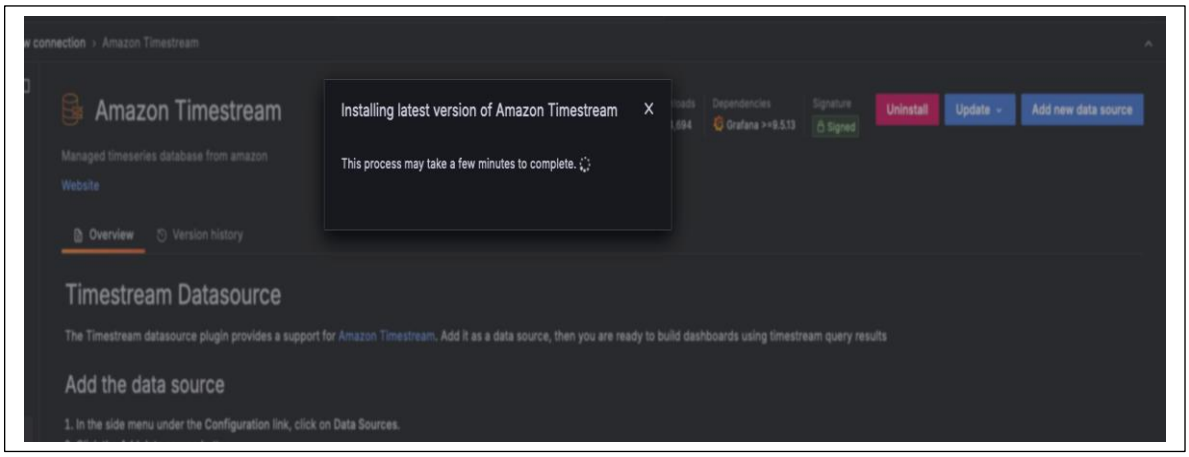
To address this, I enabled the Plugin Management feature within the workspace settings.

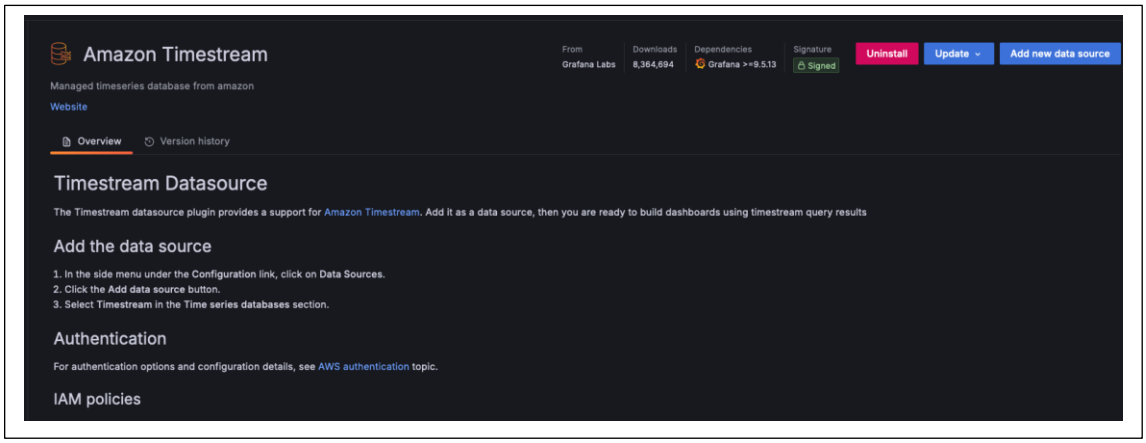




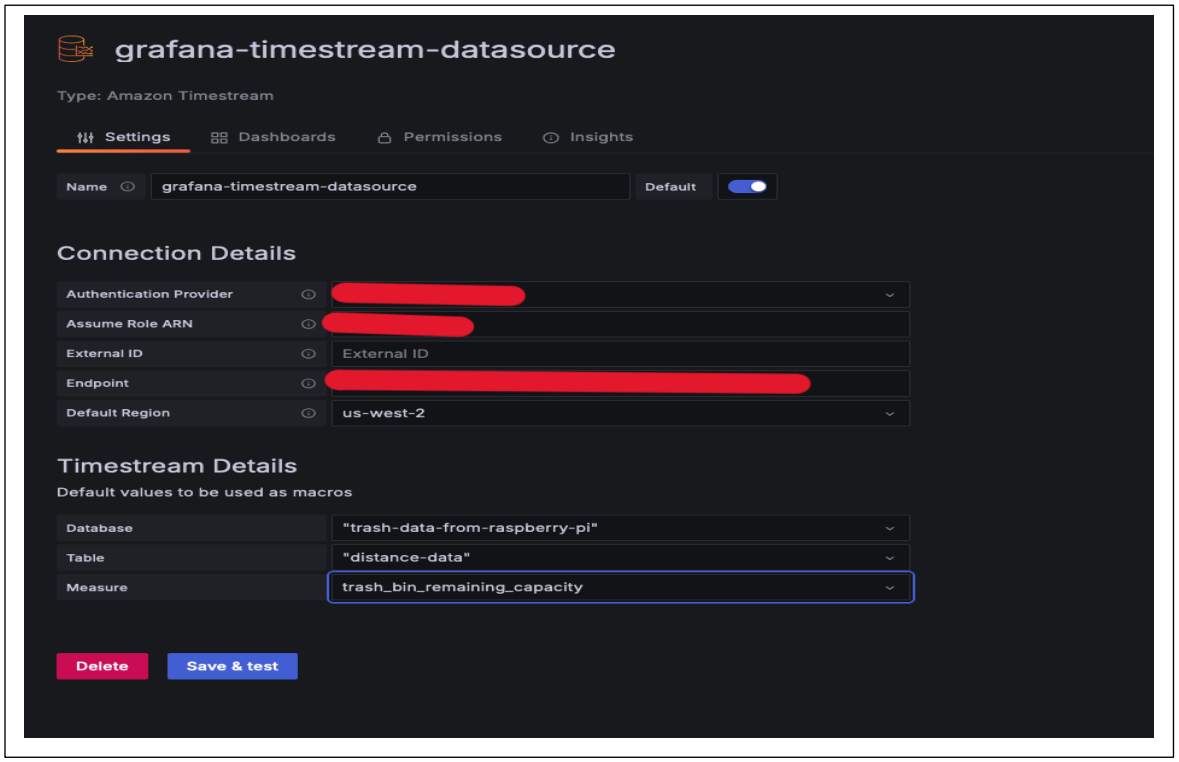
Once this feature was activated, I returned to the console, where an option to install the Amazon Timestream plugin became available.

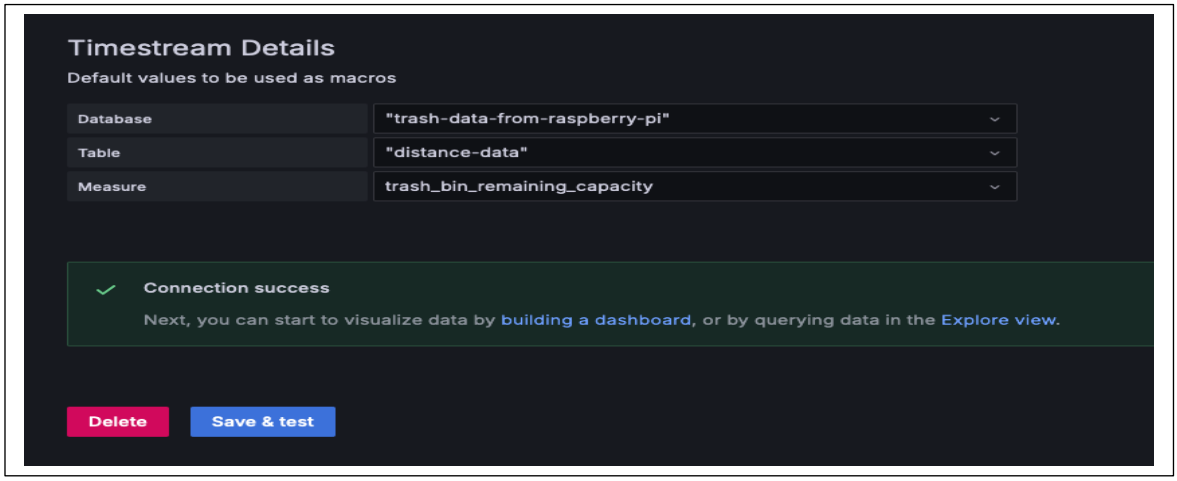




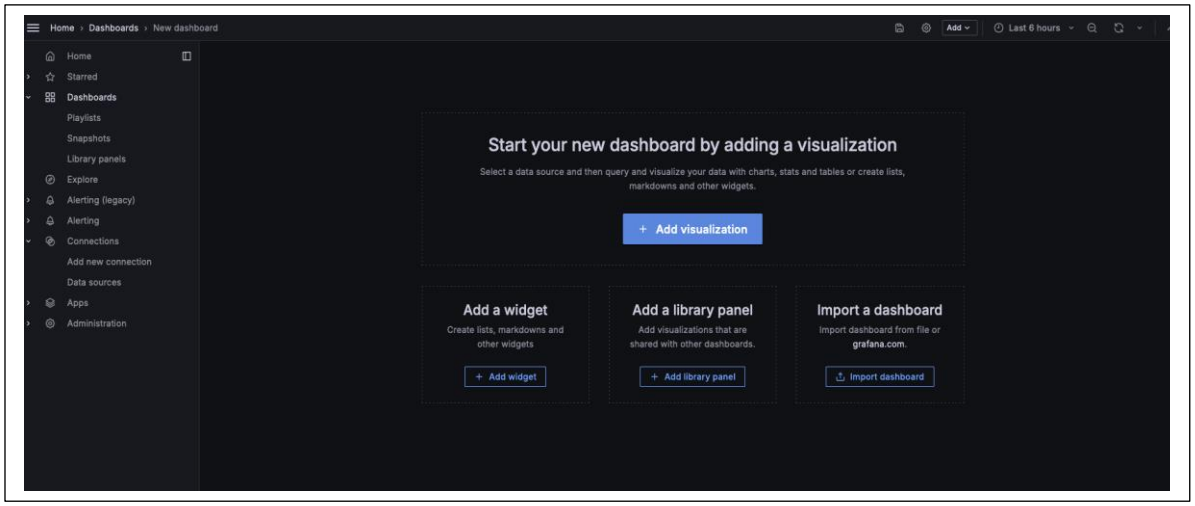


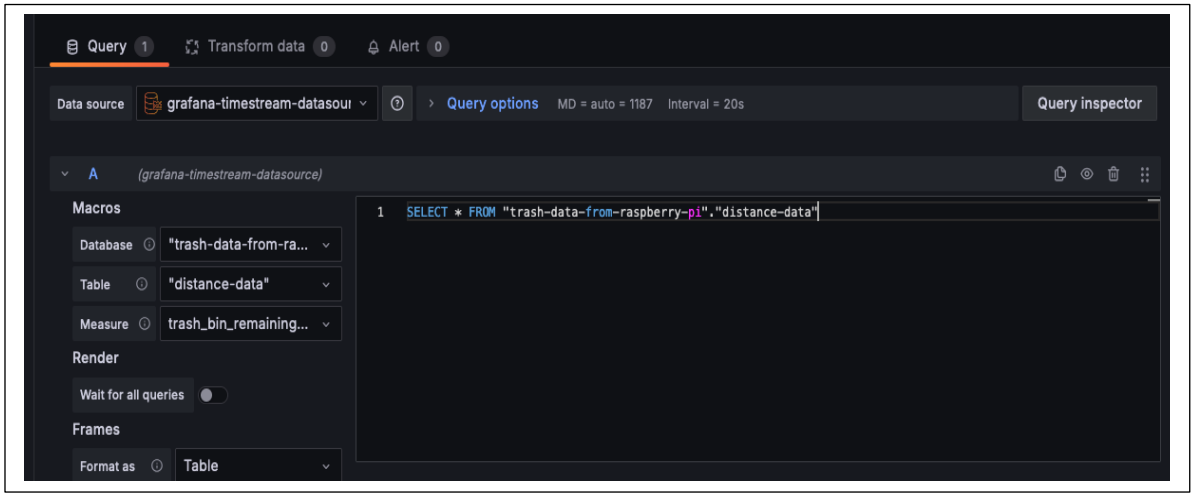
After installing the plugin successfully, I added the Timestream database as a data source and tested the connection. The test confirmed that the database was integrated with Grafana.



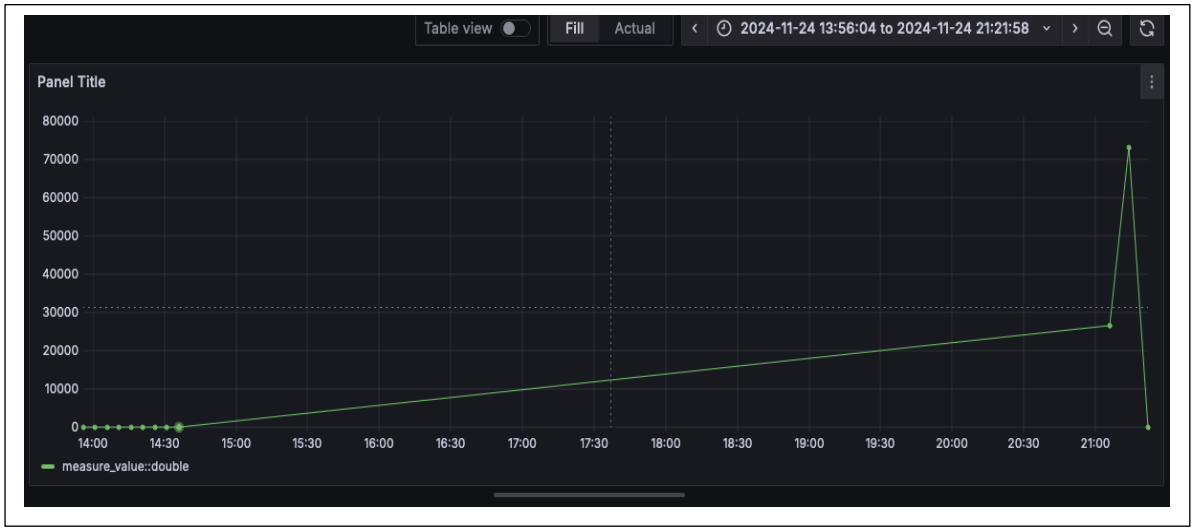


With the connection in place, I created a new dashboard, inputting the database details and a custom query to fetch relevant data.





The dashboard successfully displayed the data in both graphical and table views, providing an interactive and informative visualization.



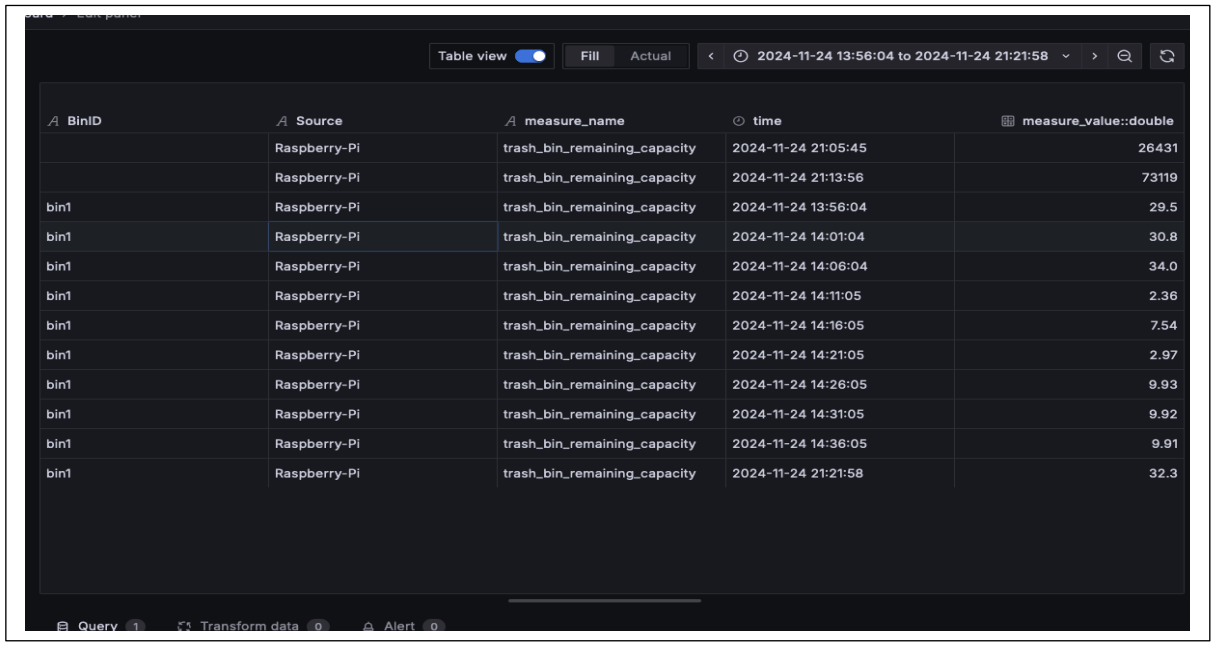
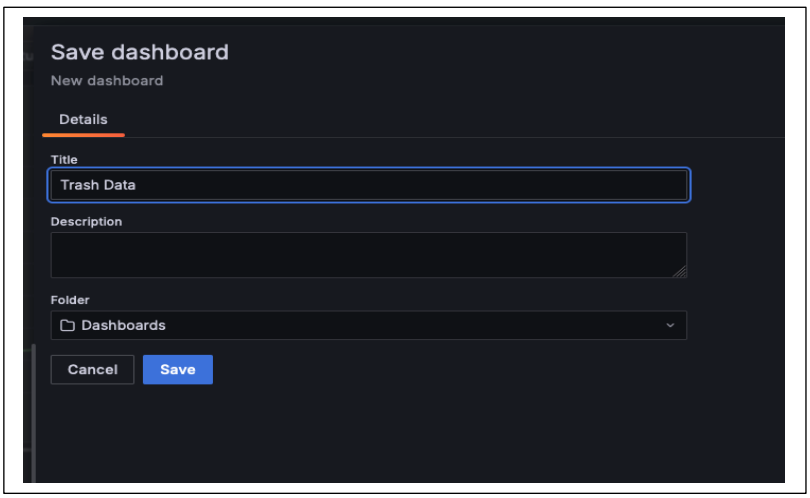


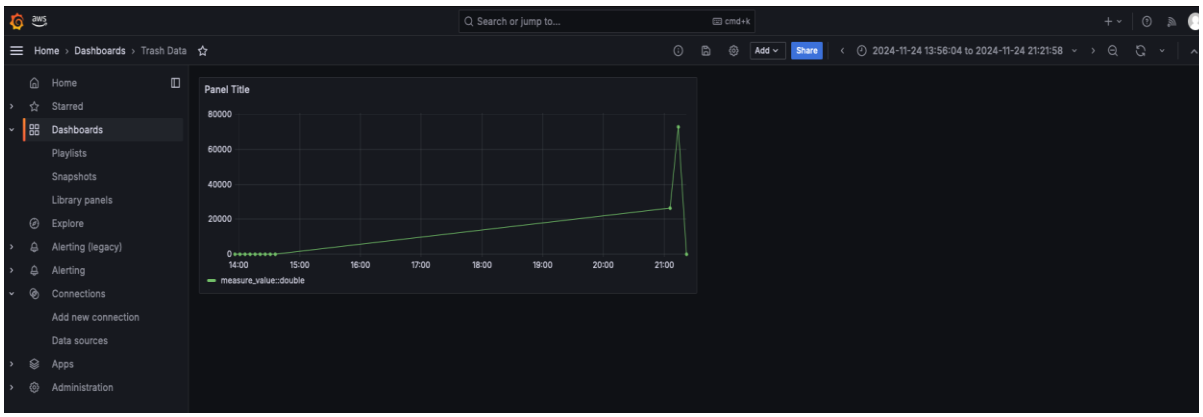
Table view | Fill | Actual | 2024-11-24 13:56:04 to 2024-11-24 21:21:58

BinID	Source	measure_name	time	measure_value::double
	Raspberry-Pi	trash_bin_remaining_capacity	2024-11-24 21:05:45	26431
	Raspberry-Pi	trash_bin_remaining_capacity	2024-11-24 21:13:56	73119
bin1	Raspberry-Pi	trash_bin_remaining_capacity	2024-11-24 13:56:04	29.5
bin1	Raspberry-Pi	trash_bin_remaining_capacity	2024-11-24 14:01:04	30.8
bin1	Raspberry-Pi	trash_bin_remaining_capacity	2024-11-24 14:06:04	34.0
bin1	Raspberry-Pi	trash_bin_remaining_capacity	2024-11-24 14:11:05	2.36
bin1	Raspberry-Pi	trash_bin_remaining_capacity	2024-11-24 14:16:05	7.54
bin1	Raspberry-Pi	trash_bin_remaining_capacity	2024-11-24 14:21:05	2.97
bin1	Raspberry-Pi	trash_bin_remaining_capacity	2024-11-24 14:26:05	9.93
bin1	Raspberry-Pi	trash_bin_remaining_capacity	2024-11-24 14:31:05	9.92
bin1	Raspberry-Pi	trash_bin_remaining_capacity	2024-11-24 14:36:05	9.91
bin1	Raspberry-Pi	trash_bin_remaining_capacity	2024-11-24 21:21:58	32.3

Query 1 | Transform data 0 | Alert 0

Finally, I saved the newly created dashboard, ensuring the configuration was preserved for future use. This process established a seamless pipeline for monitoring and analysing the data collected by the smart waste management system using Grafana and Amazon Timestream.





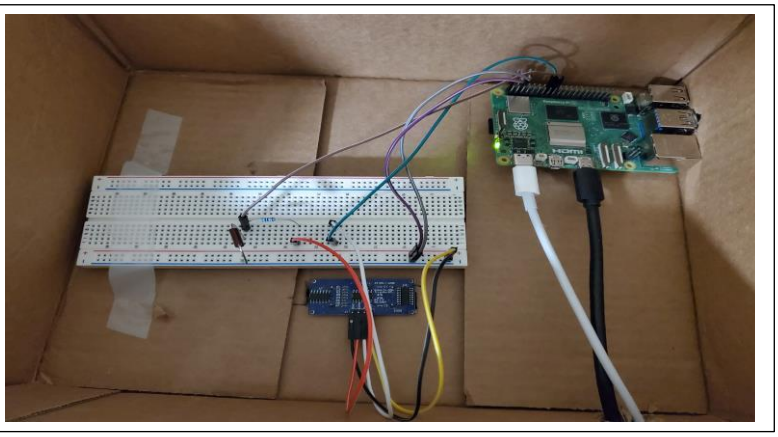
13. Setting Up Data Validation and Logging:

To ensure data accuracy and traceability, I added a validation step in the aws_publisher.py script. Before publishing to AWS IoT Core, the script checks if the calculated distance is within an expected range (e.g., 0 to the height of the bin in centimetres). Any anomalous data points (e.g., negative values or excessively large distances) are logged for review and excluded from publication.

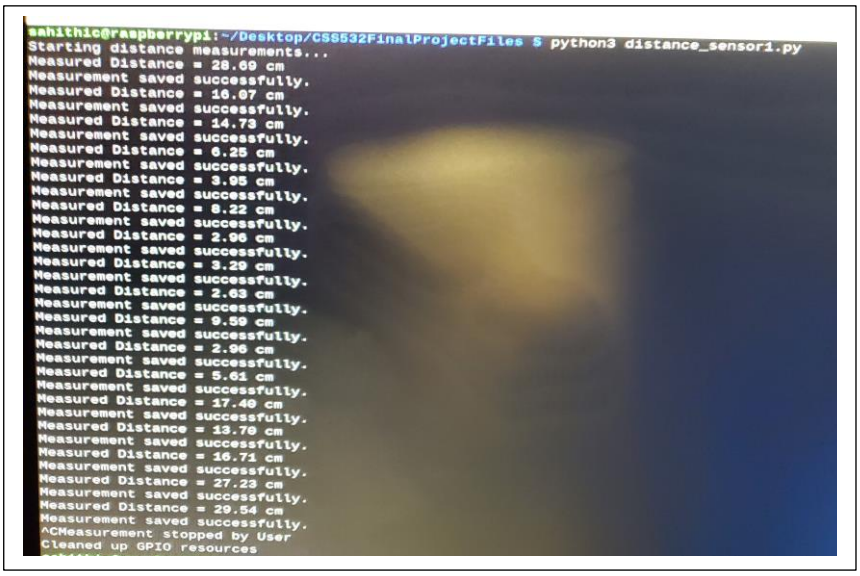
I implemented a logging mechanism using Python's logging library, which captures the details of each data point sent to AWS IoT Core. Logs include a timestamp, the distance measured, and the status of the MQTT message (success/failure). This ensures that issues can be diagnosed efficiently during testing and operation.

14. End-to-End Testing of the Smart Waste Management System:

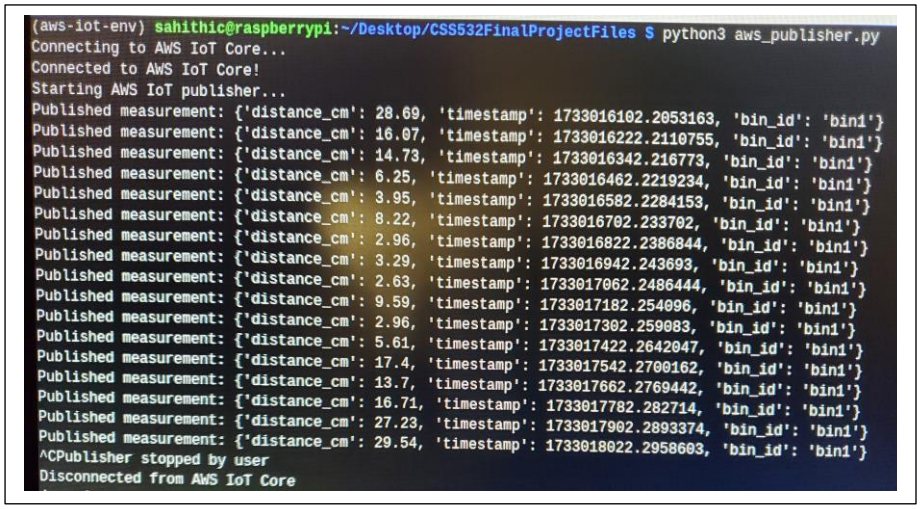
After completing the setup of all components in the smart waste management system, including AWS IoT Core, AWS SNS, AWS Lambda, Amazon Timestream, and the Grafana dashboard, comprehensive end-to-end testing was conducted to validate the system's performance and functionality.

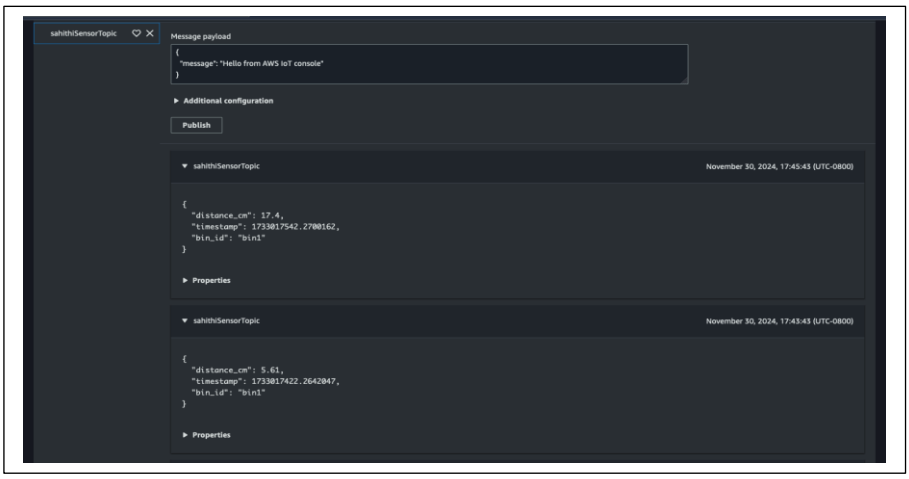


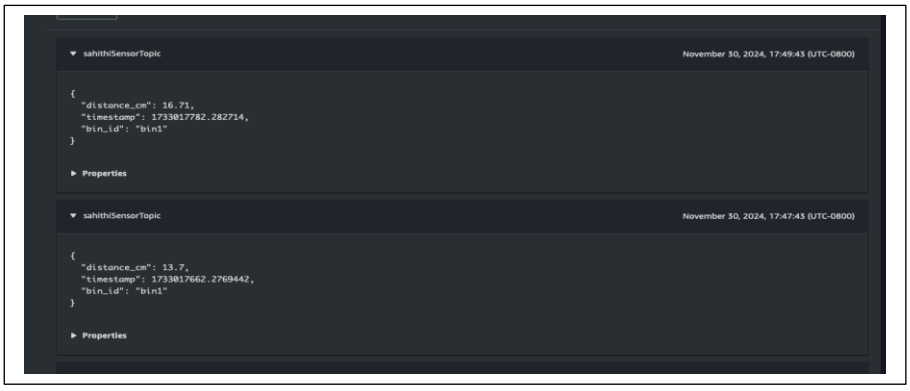
The testing involved running two Python scripts on the Raspberry Pi. The first script interfaced with the ultrasonic sensor, measured the distance to the waste level, and stored the data in a shared JSON file.

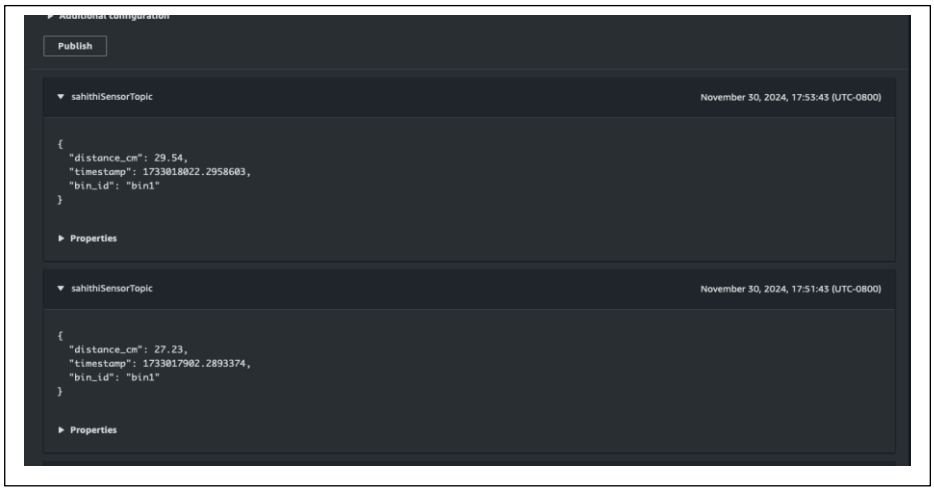


The second script monitored the JSON file continuously for new data entries. When new data was detected, the script published it to AWS IoT Core.









(**Note:** I was unable to capture all the messages published to AWS IoT Core as the system logged me out every 15 minutes.)

AWS IoT Core redirected the data to AWS Lambda, which processed the information to compute the bin's fill level, stored it in Amazon Timestream along with a timestamp, and triggered alerts based on defined thresholds.

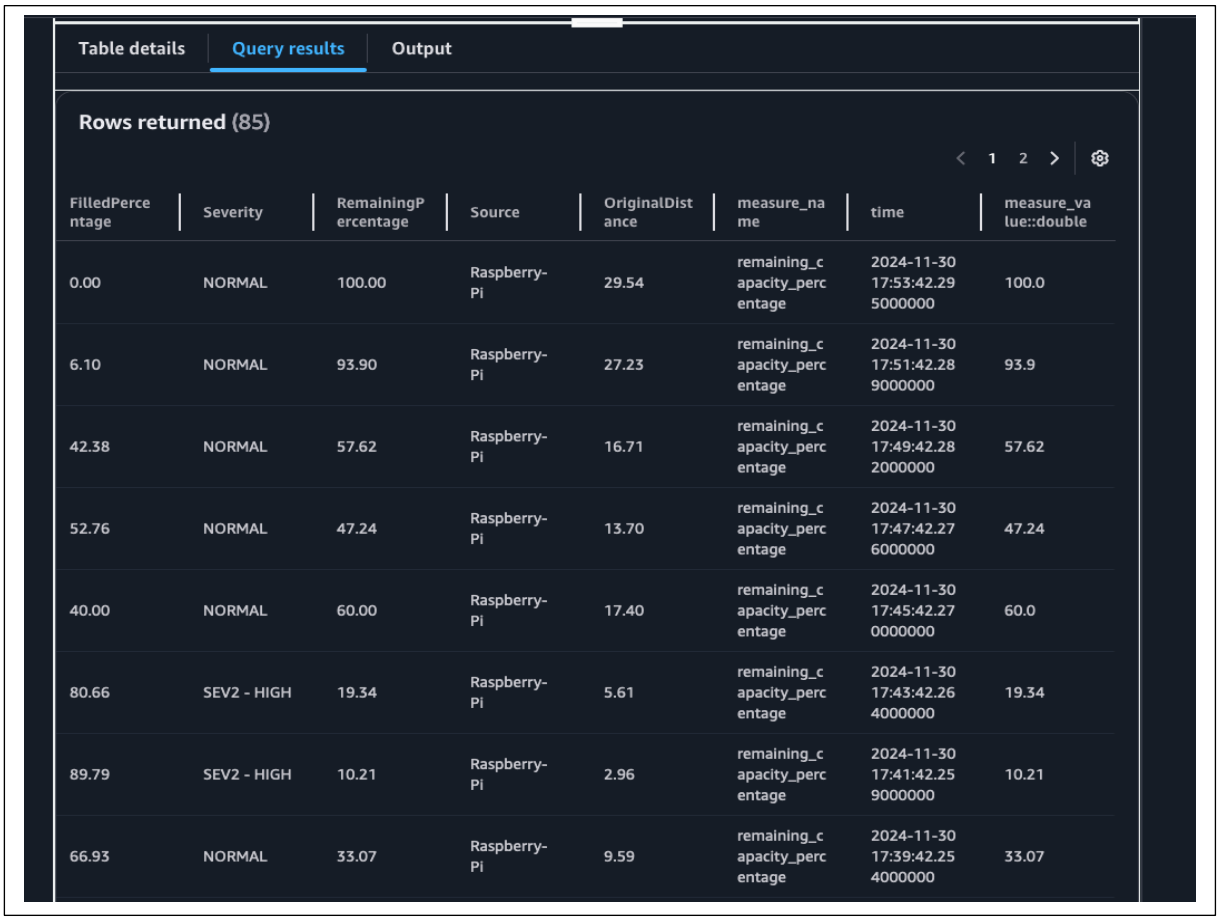
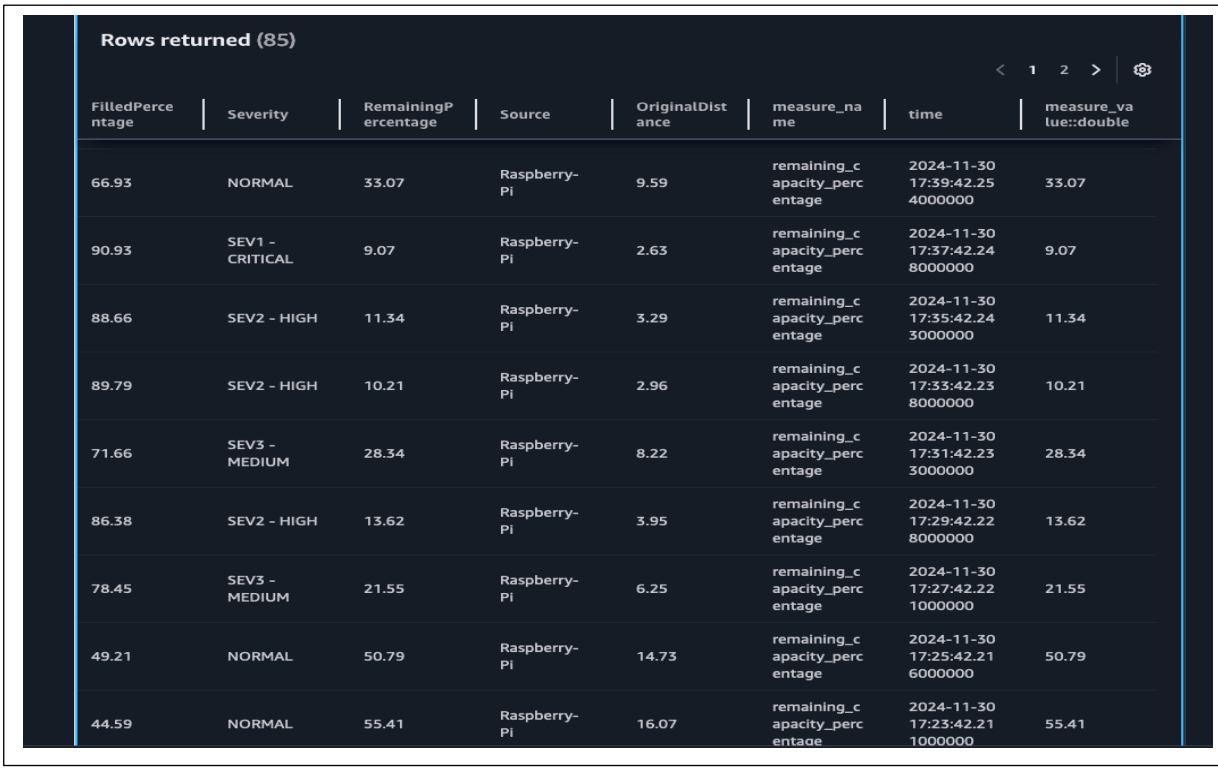


Table details | Query results | Output

Rows returned (85)

1 2

FilledPercentage	Severity	RemainingPercentage	Source	OriginalDistance	measure_name	time	measure_value::double
0.00	NORMAL	100.00	Raspberry-Pi	29.54	remaining_capacity_percentage	2024-11-30 17:53:42.295000000	100.0
6.10	NORMAL	93.90	Raspberry-Pi	27.23	remaining_capacity_percentage	2024-11-30 17:51:42.289000000	93.9
42.38	NORMAL	57.62	Raspberry-Pi	16.71	remaining_capacity_percentage	2024-11-30 17:49:42.282000000	57.62
52.76	NORMAL	47.24	Raspberry-Pi	13.70	remaining_capacity_percentage	2024-11-30 17:47:42.276000000	47.24
40.00	NORMAL	60.00	Raspberry-Pi	17.40	remaining_capacity_percentage	2024-11-30 17:45:42.270000000	60.0
80.66	SEV2 - HIGH	19.34	Raspberry-Pi	5.61	remaining_capacity_percentage	2024-11-30 17:43:42.264000000	19.34
89.79	SEV2 - HIGH	10.21	Raspberry-Pi	2.96	remaining_capacity_percentage	2024-11-30 17:41:42.259000000	10.21
66.93	NORMAL	33.07	Raspberry-Pi	9.59	remaining_capacity_percentage	2024-11-30 17:39:42.254000000	33.07



Rows returned (85)

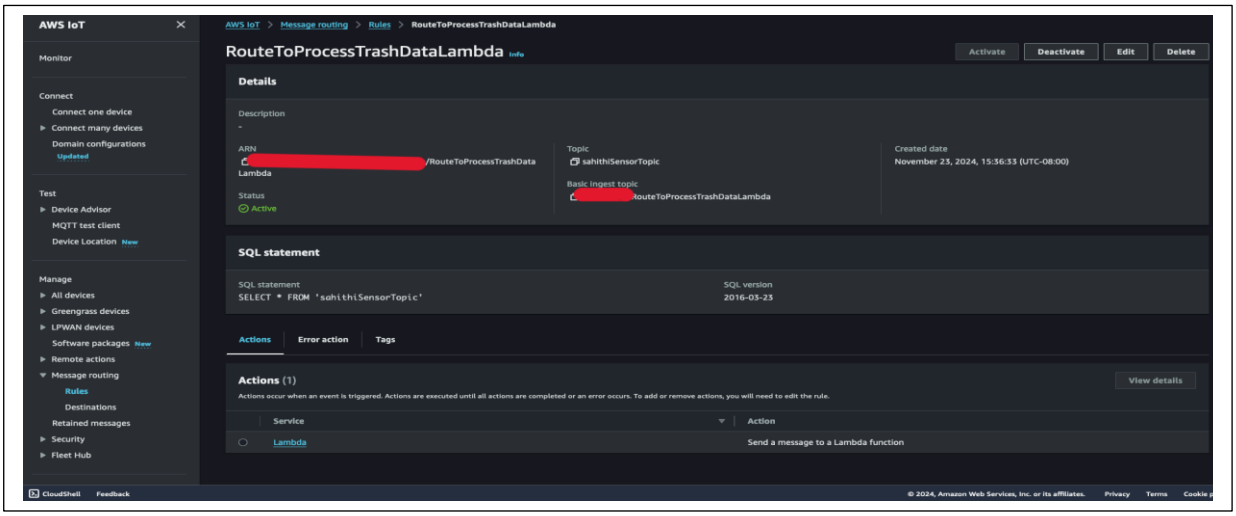
1 2

FilledPercentage	Severity	RemainingPercentage	Source	OriginalDistance	measure_name	time	measure_value::double
66.93	NORMAL	33.07	Raspberry-Pi	9.59	remaining_capacity_percentage	2024-11-30 17:39:42.254000000	33.07
90.93	SEV1 - CRITICAL	9.07	Raspberry-Pi	2.63	remaining_capacity_percentage	2024-11-30 17:37:42.248000000	9.07
88.66	SEV2 - HIGH	11.34	Raspberry-Pi	3.29	remaining_capacity_percentage	2024-11-30 17:35:42.243000000	11.34
89.79	SEV2 - HIGH	10.21	Raspberry-Pi	2.96	remaining_capacity_percentage	2024-11-30 17:33:42.238000000	10.21
71.66	SEV3 - MEDIUM	28.34	Raspberry-Pi	8.22	remaining_capacity_percentage	2024-11-30 17:31:42.233000000	28.34
86.38	SEV2 - HIGH	13.62	Raspberry-Pi	3.95	remaining_capacity_percentage	2024-11-30 17:29:42.228000000	13.62
78.45	SEV3 - MEDIUM	21.55	Raspberry-Pi	6.25	remaining_capacity_percentage	2024-11-30 17:27:42.221000000	21.55
49.21	NORMAL	50.79	Raspberry-Pi	14.73	remaining_capacity_percentage	2024-11-30 17:25:42.216000000	50.79
44.59	NORMAL	55.41	Raspberry-Pi	16.07	remaining_capacity_percentage	2024-11-30 17:23:42.211000000	55.41

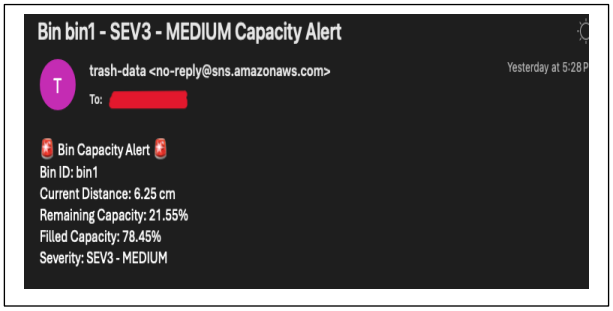
Rows returned (85)

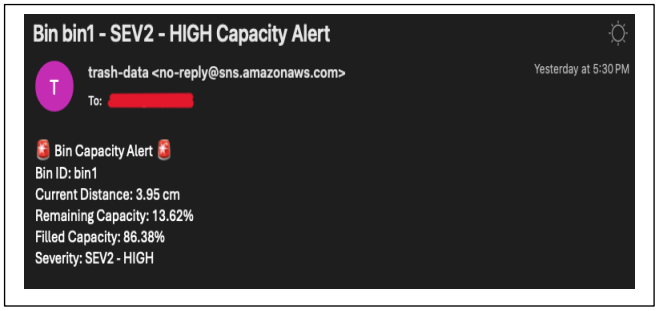
1 2

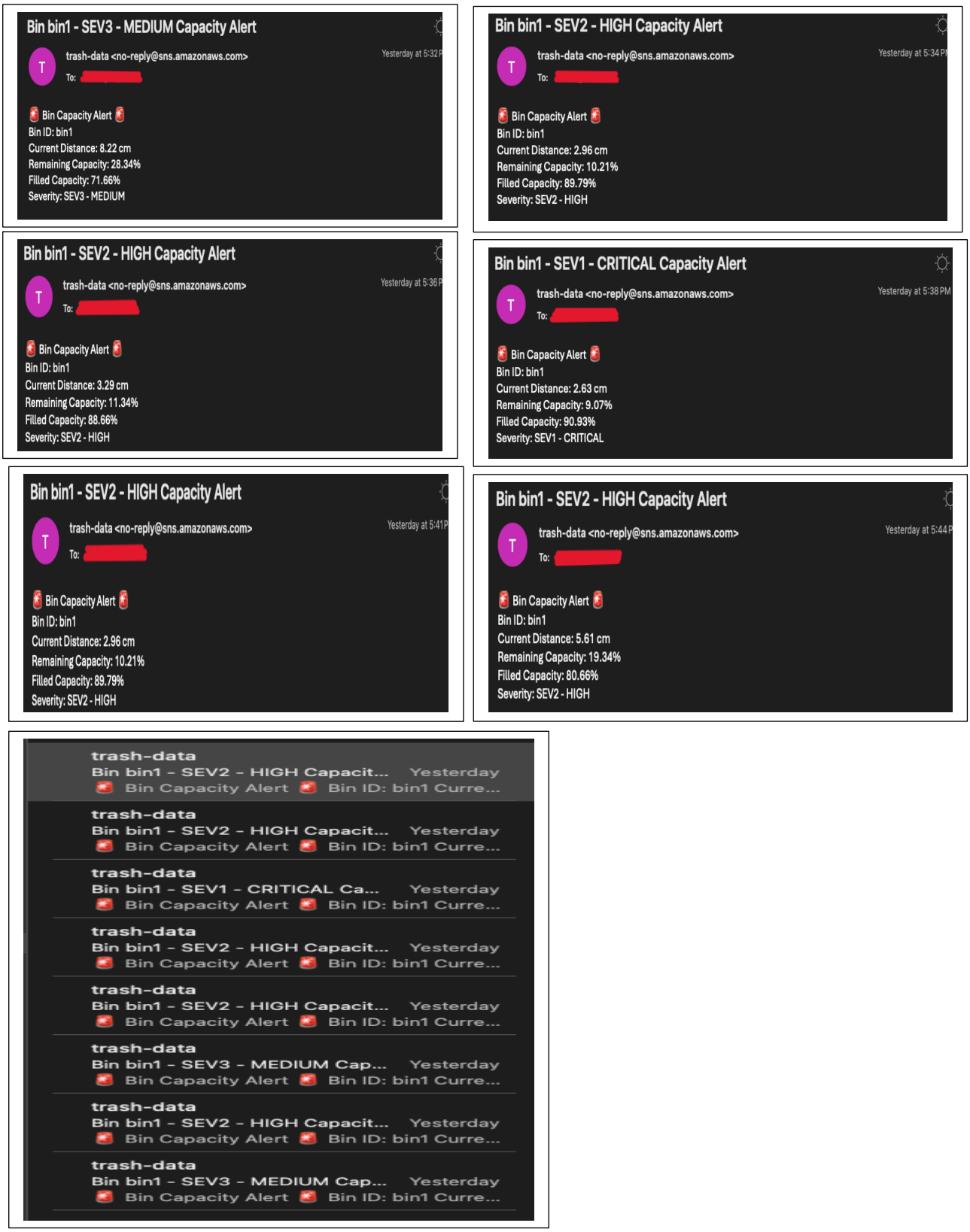
FilledPercentage	Severity	RemainingPercentage	Source	OriginalDistance	measure_name	time	measure_value::double
44.59	NORMAL	55.41	Raspberry-Pi	16.07	remaining_capacity_percentage	2024-11-30 17:23:42.211000000	55.41
1.07	NORMAL	98.93	Raspberry-Pi	28.69	remaining_capacity_percentage	2024-11-30 17:21:42.205000000	98.93
1.10	NORMAL	98.90	Raspberry-Pi	28.68	remaining_capacity_percentage	2024-11-30 17:19:54.982000000	98.9
1.17	NORMAL	98.83	Raspberry-Pi	28.66	remaining_capacity_percentage	2024-11-30 17:18:58.354000000	98.83



The Grafana dashboard was configured to visualize the data, with the x-axis representing time and the y-axis showing the percentage of remaining bin space. Testing began with an empty bin, and trash was added and removed randomly to simulate dynamic waste accumulation. The system accurately measured and updated the bin's fill level throughout the test. Threshold-based alerts were generated correctly: at 70% (Sev3), 80% (Sev2), and 90% (Sev1) fill levels, demonstrating the efficacy of the alerting mechanism.

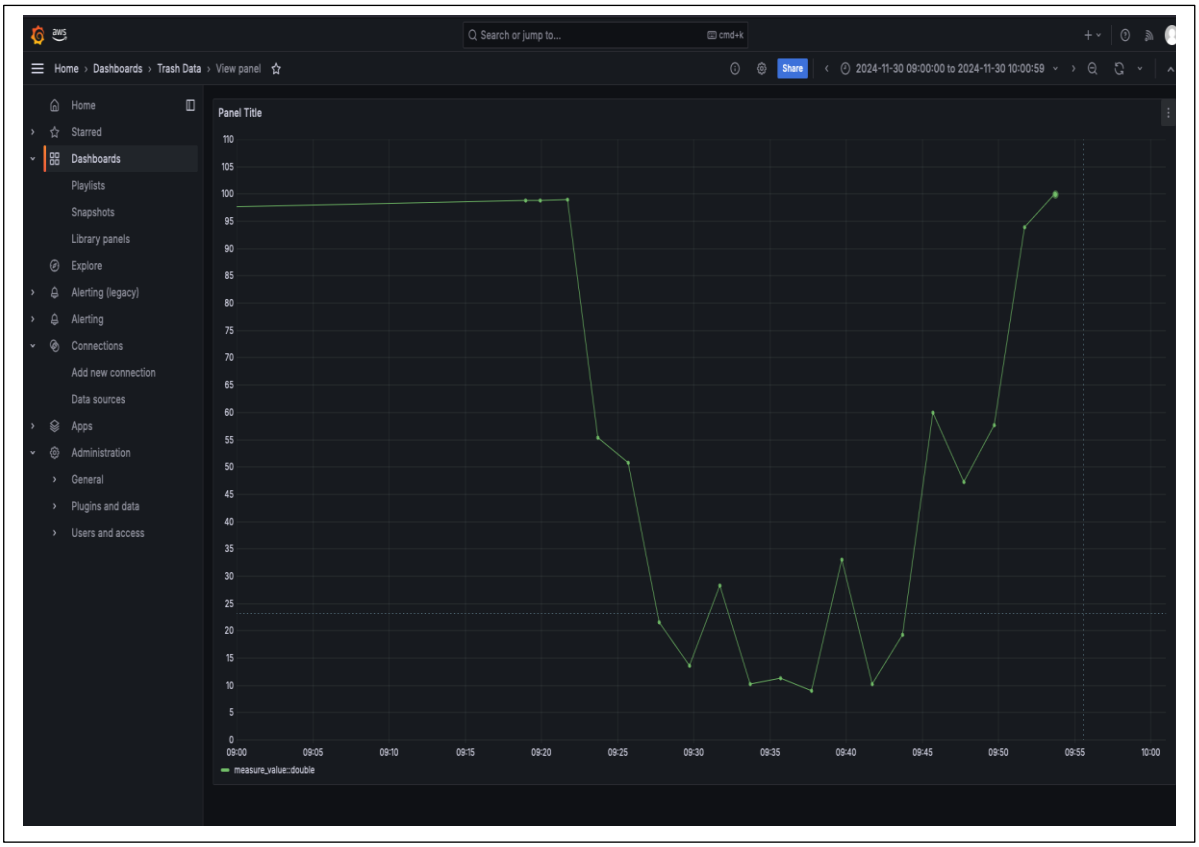


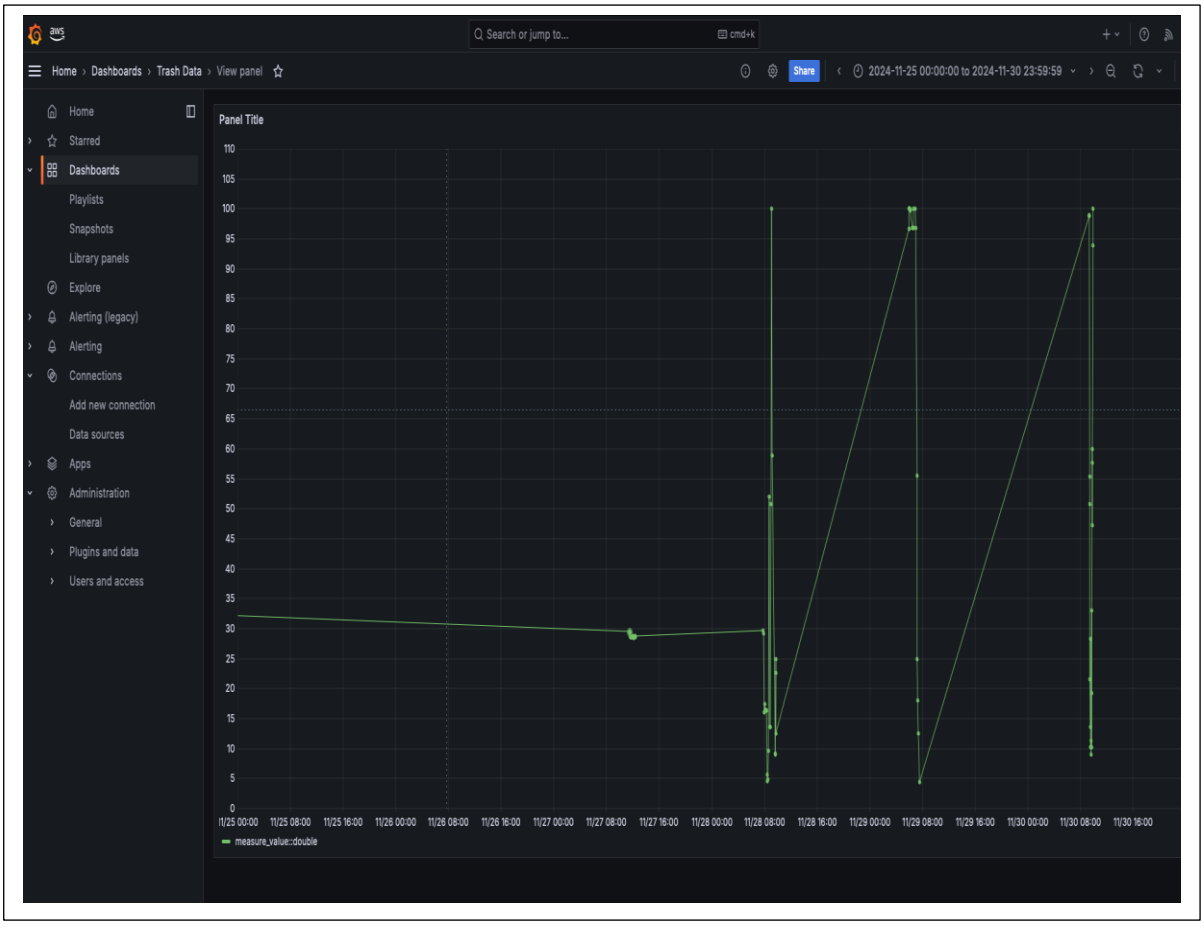


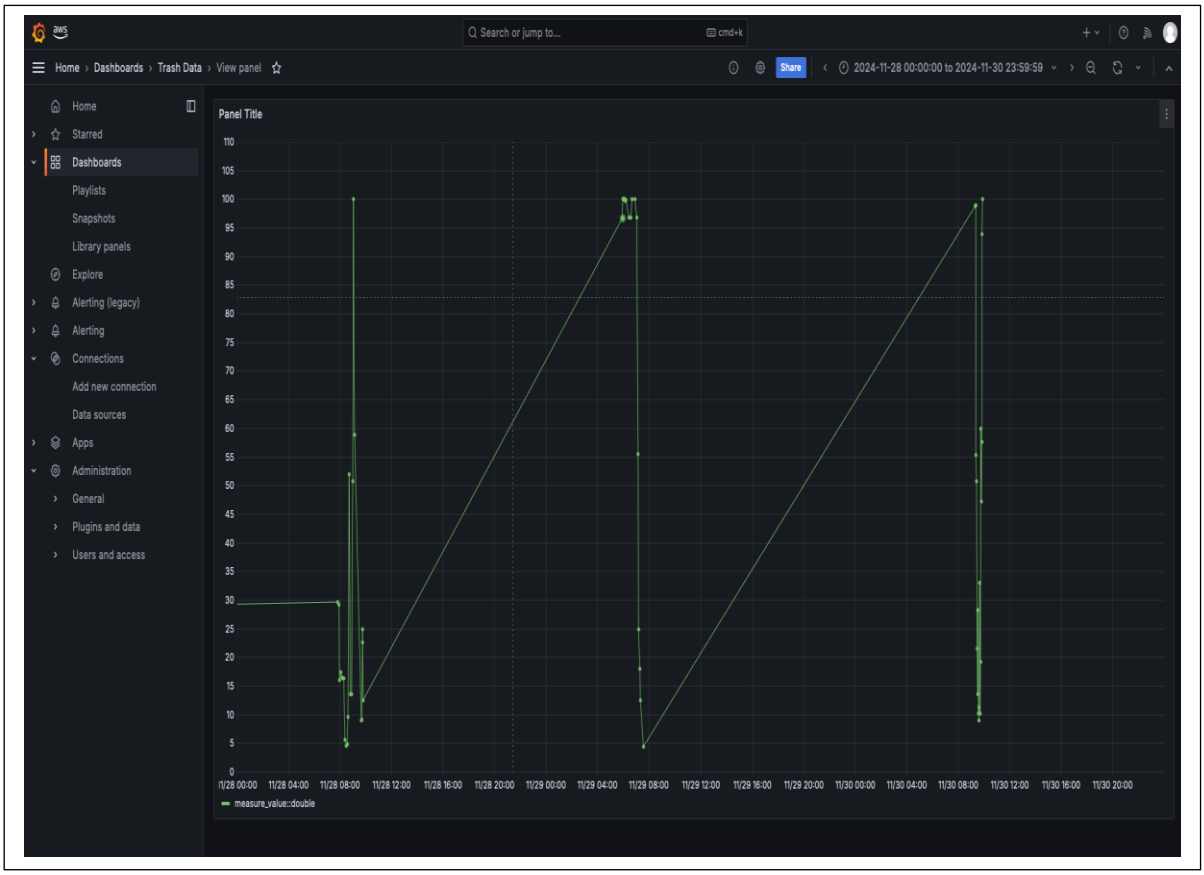


The data flow across the system was monitored at each stage. Measurements were successfully transmitted to AWS IoT Core, processed by AWS Lambda, stored in Amazon Timestream, and displayed visually in Grafana in real-time. The system

consistently demonstrated accurate distance measurement, timely alert generation, and efficient data visualization, validating its reliability and suitability for practical waste management applications.







Encountered challenges:

The development of the device code and integration with cloud services presented multiple challenges that required iterative troubleshooting and adaptations to resolve, some of them are

1. Sensor Calibration: The ultrasonic sensor initially produced inconsistent readings due to variations in sensitivity across different surfaces. This issue was resolved by conducting tests under diverse conditions and recalibrating the sensor to ensure accurate measurements in real-world scenarios.

2. Raspberry Pi Booting Issues: While setting up the Raspberry Pi, the device failed to boot from the external SD card. This was addressed by carefully re-flashing the SD card and making necessary adjustments to the config.txt file to enable proper booting.

3. Grafana Integration: During the setup of the Grafana dashboard, I encountered permission restrictions while attempting to add Amazon Timestream as the data source. The workspace lacked the default Plugin Management feature. After identifying this issue, I enabled Plugin Management within the workspace settings, which allowed the integration to proceed.

4. Additionally, the timestamps in Grafana were initially displayed in UTC. To address this, I adjusted the settings to display the time in PST, ensuring that the data aligned with local time zones.

5. AWS SNS Alert Configuration: I initially planned to use both email and SMS notifications for alerts. However, due to predefined limitations on the number of SMS messages allowed by AWS, I was unable to increase the quota. Despite reaching out to AWS customer service, I received no response. After consulting with my professor, I decided to proceed with email notifications exclusively for the alerting mechanism.

6. AWS IoT Core Connection Timeout: While testing the data publishing functionality, I faced periodic disconnections from AWS IoT Core due to session timeouts, which occurred approximately every 15 minutes. This required frequent re-authentication, which posed a challenge for continuous monitoring.

7. Debugging Lambda Functions: AWS Lambda functions initially failed to process data correctly due to improperly defined event triggers. Debugging these functions involved analysing CloudWatch logs.

Time spent to work on the project:

I spent nearly ~ 160 hours on this project for the past 1 month.