EE5111 Selected Topics in Industrial Control & Instrumentation

Continuous Assessment:

Implement a simple IoT pipeline with AWS Cloud platform and visualise the data

Submitted by

Chua Kang Loong, A0125958R

Teng Kok Hua, A0179701B

Department of Electrical & Computer Engineering

Table of Contents

Introduction

In a real-time embedded system, the architecture is subdivided to different layers and each of the layer can be replaced without affecting the functionality of overall IoT system. Sensor is sensing and gathering information about environment. Low power embedded processor process data and interface with sensor and wireless transceiver that transfer data to internet gateway. Internet gateway connects to internet and allows data transfer to server. The application is running on server to deliver application specific service to user.

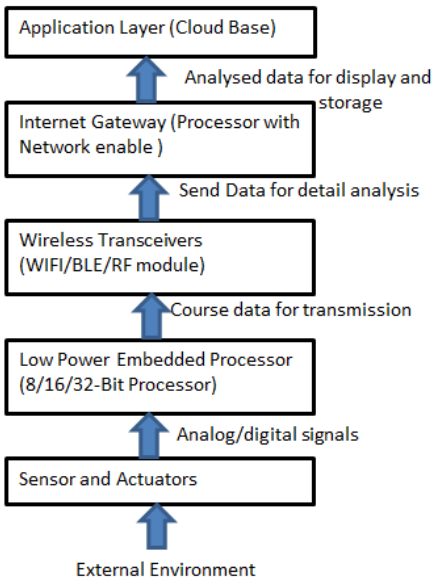

Figure 1: 5 Layers of a simple IOT Architecture

This project is to implement a simple Internet of Things (IoT) pipeline with AWS Cloud platform and visualise the data. AWS IoT provides secure, bi-directional communication between Internet-connected devices such as sensors, actuators, embedded microcontrollers e.g. RaspBerry Pi, AWS SageMaker, DeepAR (Machine Learning) and the AWS Cloud. This enables user to collect telemetry data from multiple devices, and store and analyze the data. We will simulate two small IoT setups through Python that record and push data from two jet engines as shown in Figure 2 to AWS DynamoDB. A step by step guideline is written on how to set up the entire pipeline process.

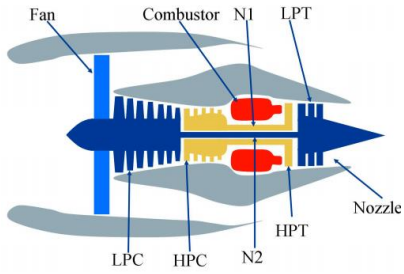

Figure 2 - Jet Engine

In this task, we obtained four jet engines data namely FD001, FD002, FD003 and FD004 in text file format from given, each text file contains a jet engine data operating settings and sensors from Commercial Modular Aero-Propulsion System. The requirement is to publish 2 pre-defined FD001 and FD002 jet engine data as shown in Figure 3 as IoT Things to AWS. In this arrangement of data, there are a sum of 26 arrangements of data for every line, which is given in the following configurations: id, cycle, os1, os2, os3, sensor1, … , sensor21 which relate to the data collected on the jet engine at each time instance.

train_FD001 - Notepad

File Edit Format View Help

```
1 1 -0.0007 -0.0004 100.0 518.67 6
14.62 21.61 554.36 2388.06 9046.19
2388.02 8138.62 8.4195 0.03 392 23
 1 2 0.0019 -0.0003 100.0 518.67 6
14.62 21.61 553.75 2388.04 9044.07
2388.07 8131.49 8.4318 0.03 392 23
 1 3 -0.0043 0.0003 100.0 518.67 6
14.62 21.61 554.26 2388.08 9052.94
2388.03 8133.23 8.4178 0.03 390 23
 1 4 0.0007 0.0000 100.0 518.67 64
14.62 21.61 554.45 2388.11 9049.48
2388.08 8133.83 8.3682 0.03 392 23
 1 5 -0.0019 -0.0002 100.0 518.67
14.62 21.61 554.00 2388.06 9055.15
```

train_FD002 - Notepad

File Edit Format View Help

```
1 1 34.9983 0.8400 100.0 449.44 555.32 1358.61 1137.23
5.48 8.00 194.64 2222.65 8341.91 1.02 42.02 183.06
2387.72 8048.56 9.3461 0.02 334 2223 100.00 14.73 8.8071
1 2 41.9982 0.8408 100.0 445.00 549.90 1353.22 1125.78
3.91 5.71 138.51 2211.57 8303.96 1.02 42.20 130.42
2387.66 8072.30 9.3774 0.02 330 2212 100.00 10.41 6.2665
1 3 24.9988 0.6218 60.0 462.54 537.31 1256.76 1047.45
7.05 9.02 175.71 1915.11 8001.42 0.94 36.69 164.22
2028.03 7864.87 10.8941 0.02 309 1915 84.93 14.08 8.6723
1 4 42.0077 0.8416 100.0 445.00 549.51 1354.03 1126.38
3.91 5.71 138.46 2211.58 8303.96 1.02 41.96 130.72
2387.61 8068.66 9.3528 0.02 329 2212 100.00 10.59 6.4701
1 5 25.0005 0.6203 60.0 462.54 537.07 1257.71 1047.93
7.05 9.03 175.05 1915.10 7993.23 0.94 36.89 164.31
```

Figure 3 - FD001 and FD002 Jet Engines Data

Introduction to Message Queuing Telemetry Transport (MQTT): In this assignment, we will be using MQTT as a Client Server to publish/subscribe IoT Things data, MQTT is a messaging transport protocol that is lightweight, open, simple, and designed so as to be easy to implement.

These characteristics make it ideal for use in constrained environments such as for communication in Machine to Machine (M2M) and IoT contexts where a small code footprint is required and/or network bandwidth is at a premium. Figure 4 shows the full AWS IoT communication diagram for this assignment to publish data from two jet engines to client server, and AWS DynamoDB would subscribe from it.

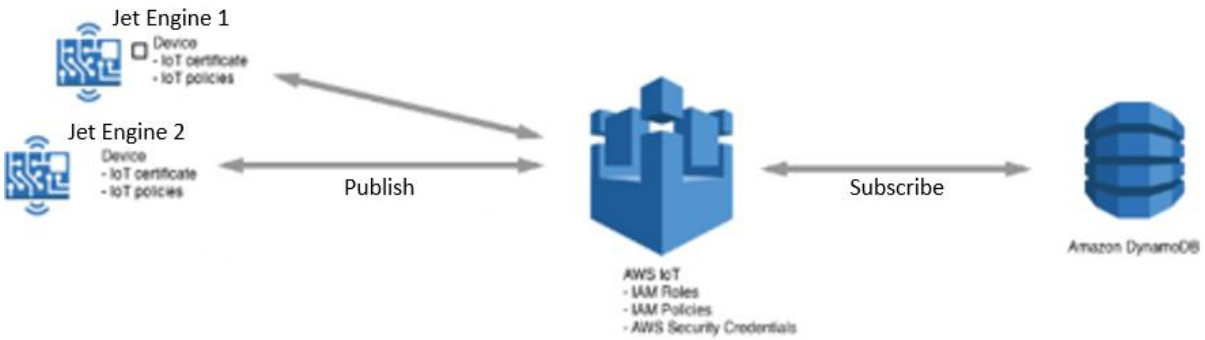

Figure 4 - AWS IoT Communication Diagram

Setting Up AWS IoT and Sending Data with Development Computer: The setting process requires registration of an AWS account, an IAM administrator user in the AWS account and a desktop or laptop development computer to work with the AWS IoT console from a web browser, and to push jet engines data into AWS IoT. This computer can be running a Windows, macOS, Linux, or Unix operating system.

Create Thing1 for FD001

Step 1: Create the AWS IoT Policy for FD001 Thing 1

In this step, to allow desktop/laptop as a substitute simulator, to perform AWS IoT operations by creating an AWS IoT policy. X.509 certificates are used to authenticate devices with AWS IoT. AWS IoT policies are used to authorize devices to perform AWS IoT operations, such as subscribing or publishing to MQTT topics. Under IoT Core console, select ‘Secure’ then

'Policies' to 'Create' a policy for FD001 Thing 1 as shown in Figure 5. Follow the red boxes guide and Enter a 'Name', 'iot:*', '*' and check 'Allow' checkbox respectively.

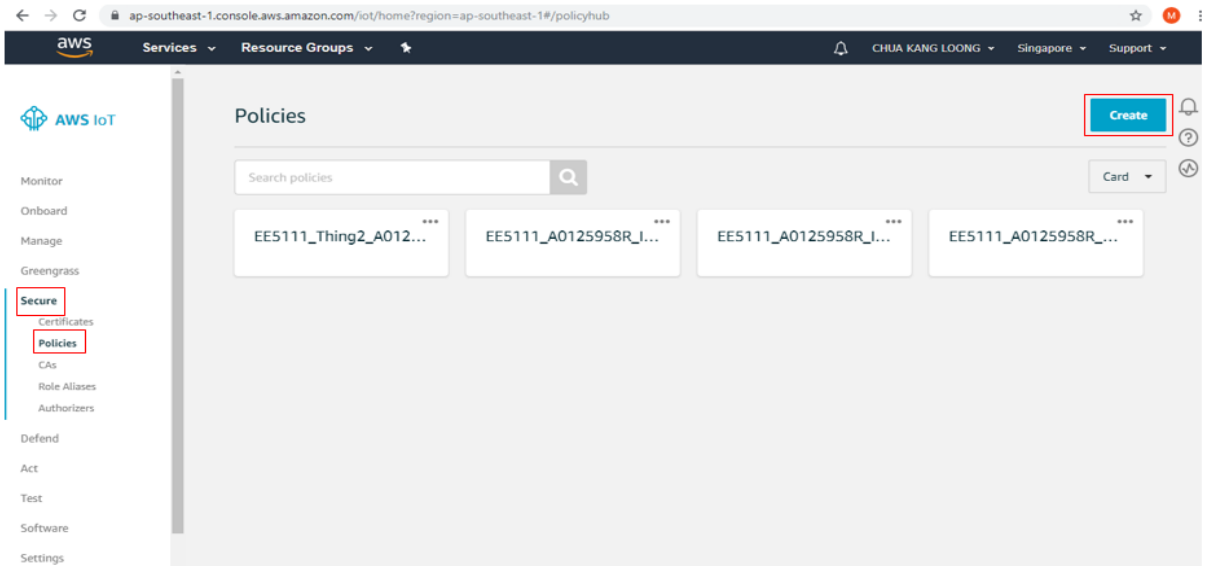

Figure 5 - Setting up Policy part 1

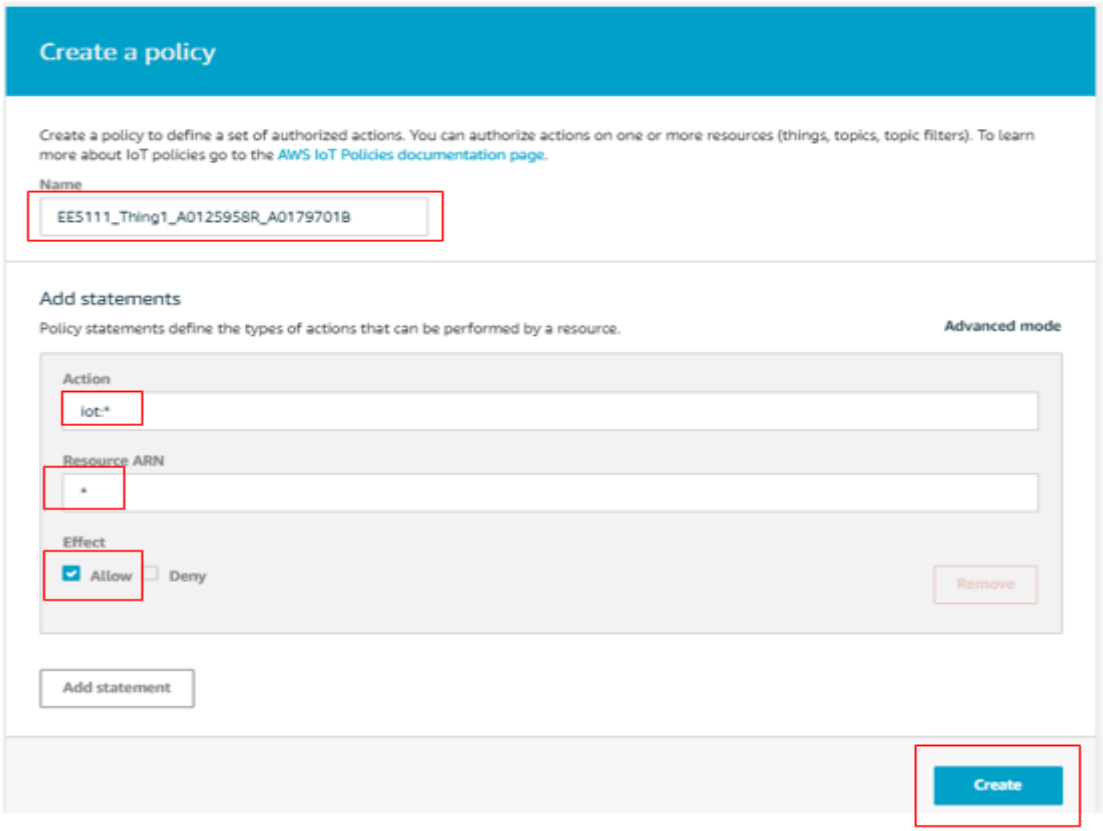

Figure 6 - Setting up Policy part 2

Step 2: Create Thing for FD001

In this step, create a thing in AWS IoT to represent desktop/laptop as a device simulator. Devices connected to AWS IoT are represented by things in the AWS IoT registry. The registry keeps a record of all of the devices that are connected to the AWS account in AWS IoT. Under IoT Core console, select ‘Manage’ then ‘Things’ to ‘Create’ a Thing for FD001 as shown in Figure 7 and 8. Follow the red boxes guide and Enter a ‘Name’ and select ‘next’ as shown in Figure 9.

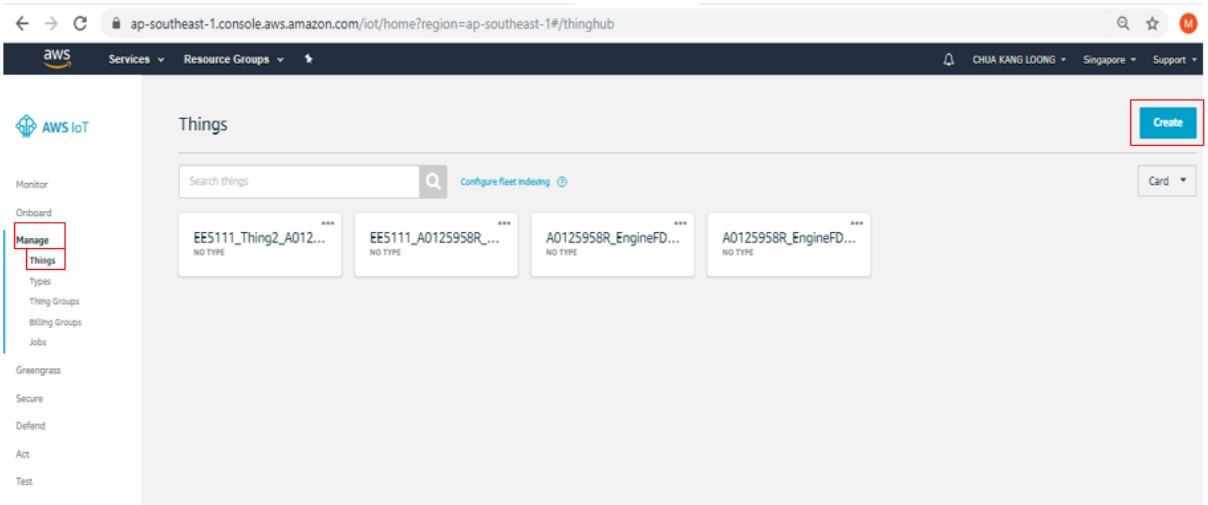

Figure 7 - Setting up Thing part 1

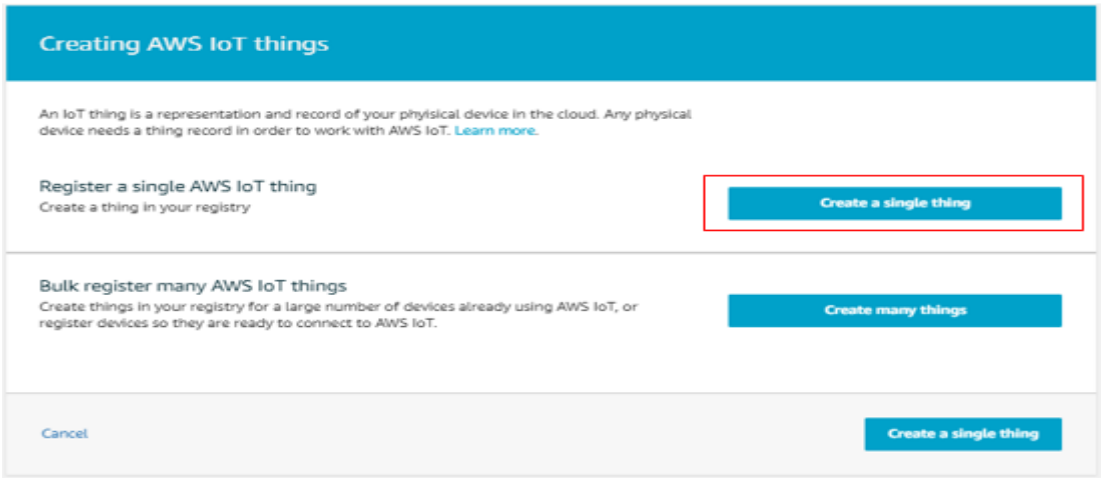

Figure 8 - Setting up Thing part 2

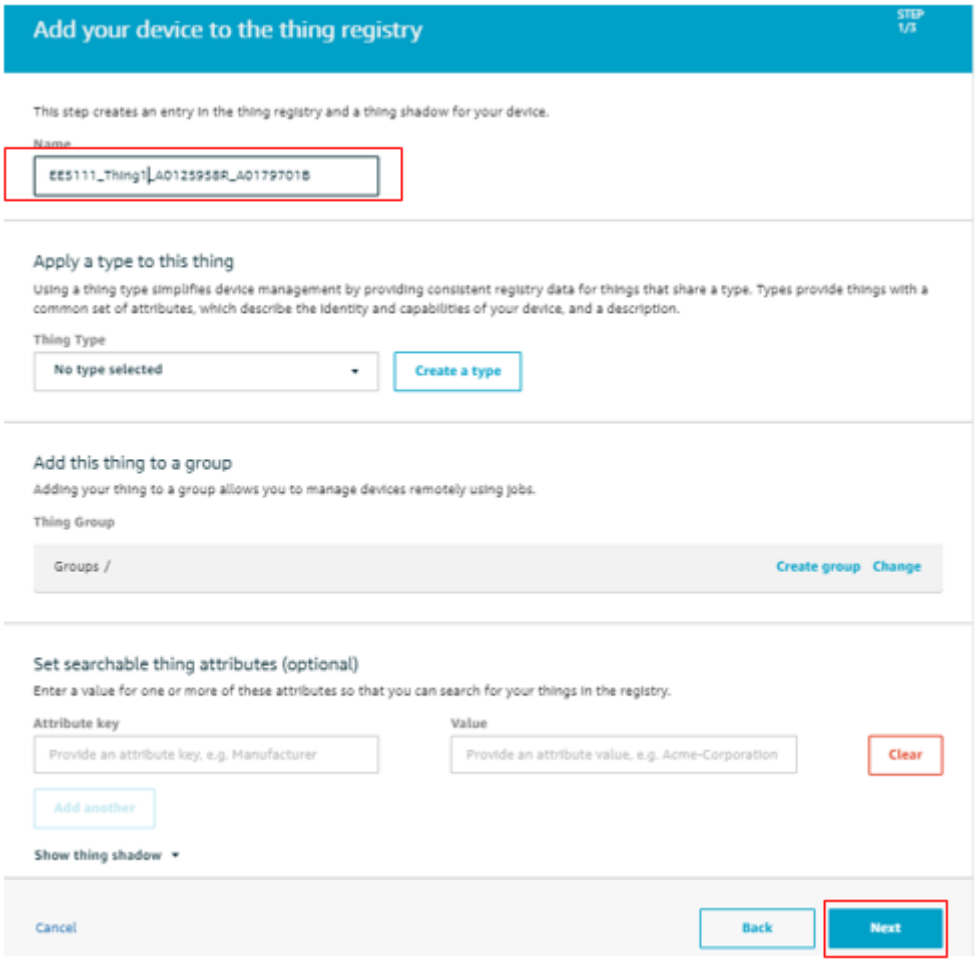

Figure 9 - Setting up Thing part 3

Step 3: Create Certificate for FD001 Thing

Select ‘Create certificate’ for FD001 Thing to gain authentication to publish jet engine data to MQTT server for as shown in Figure 10. Download ‘A certificate for this thing’, ‘A public key’ and ‘A private key’ and select ‘Activate’ as shown in Figure 11. Put the 3 downloaded files with the Python file into the same folder. Click on the checkbox of the thing ‘Name” to add policy to it as shown in Figure 12.

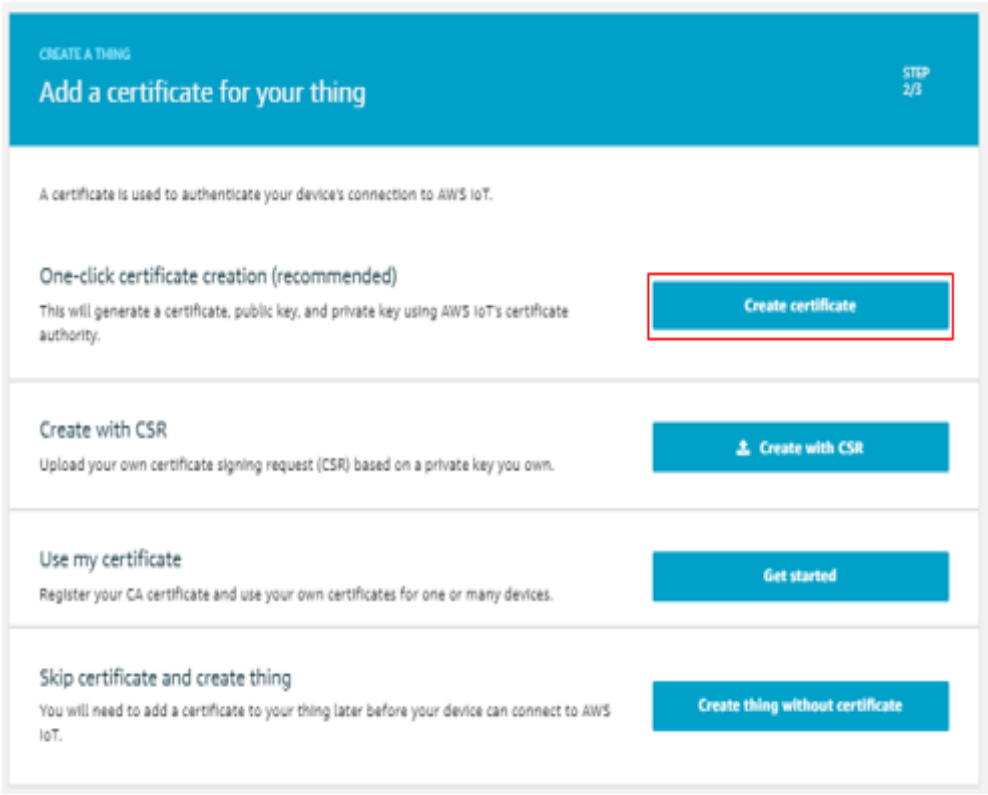

Figure 10 - Setting up Thing part 4

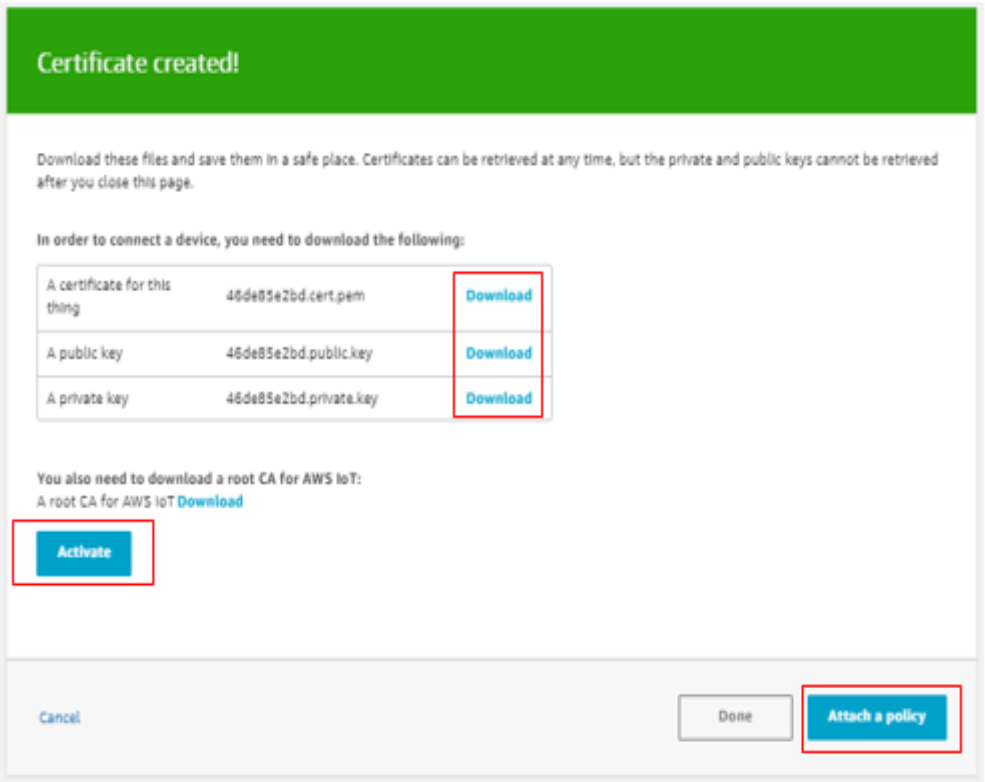

Figure 11 - Setting up Thing part 5

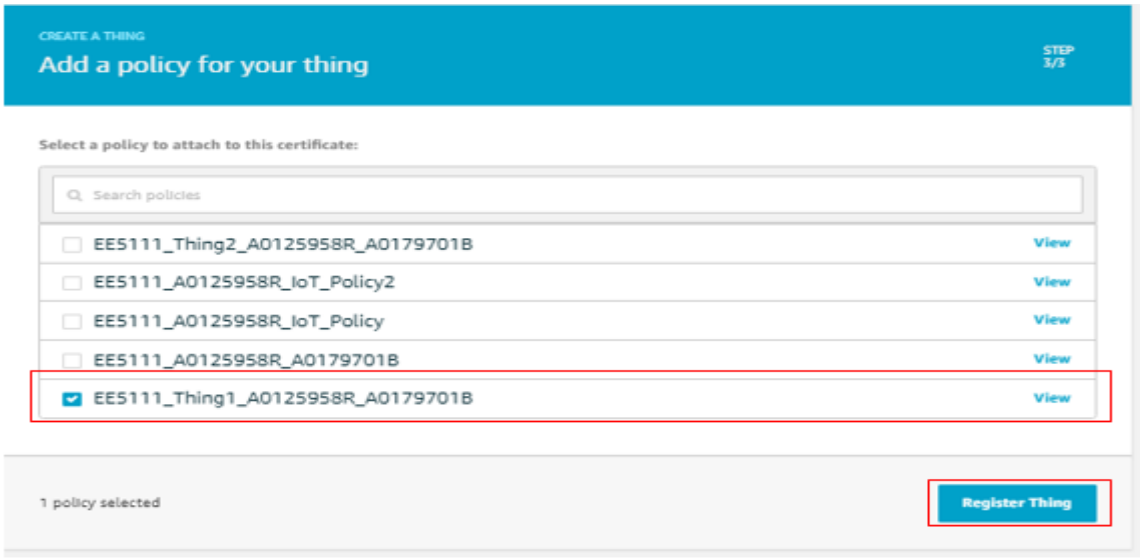

Figure 12 - Setting up Thing part 6

Step 4: Send and Receive Test Data for the Thing

In this step, we can send jet engines data to the thing shadow for the desktop/laptop as a device simulator. A thing's shadow is a JSON document, stored in AWS IoT , that AWS IoT uses to save and retrieve current state information for a device. The Device Shadow Service for AWS IoT maintains a shadow for each device connected to AWS IoT. Go to ‘Things’ and select ‘Interact’ as shown in Figure 13.

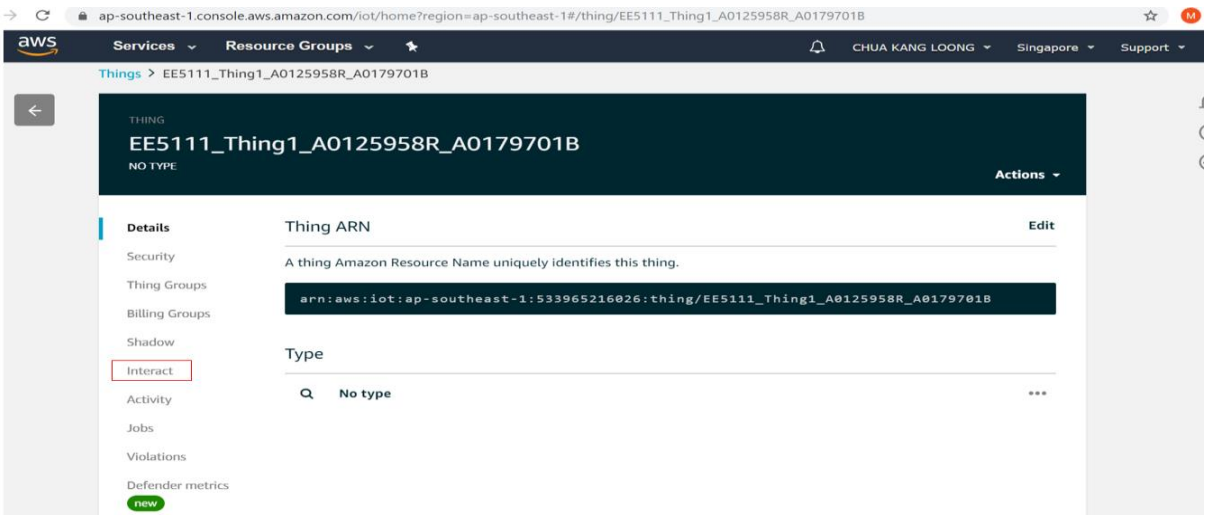

Figure 13 - Thing’s shadow send and receive data part 1

For MQTT, make a note of the value for each of the following MQTT topics, which enable you to set and get updates to the shadow as shown in Figure 14:

- Update to this thing shadow (for example, $aws/things/EE5111_Thing1_A125958R_A0179701B/shadow/update)
- Get this thing shadow (for example, $aws/things/EE5111_Thing1_A125958R_A0179701B/shadow/get)
- Get this thing shadow accepted (for example, $aws/things/EE5111_Thing1_A125958R_A0179701B/shadow/get/accepted)

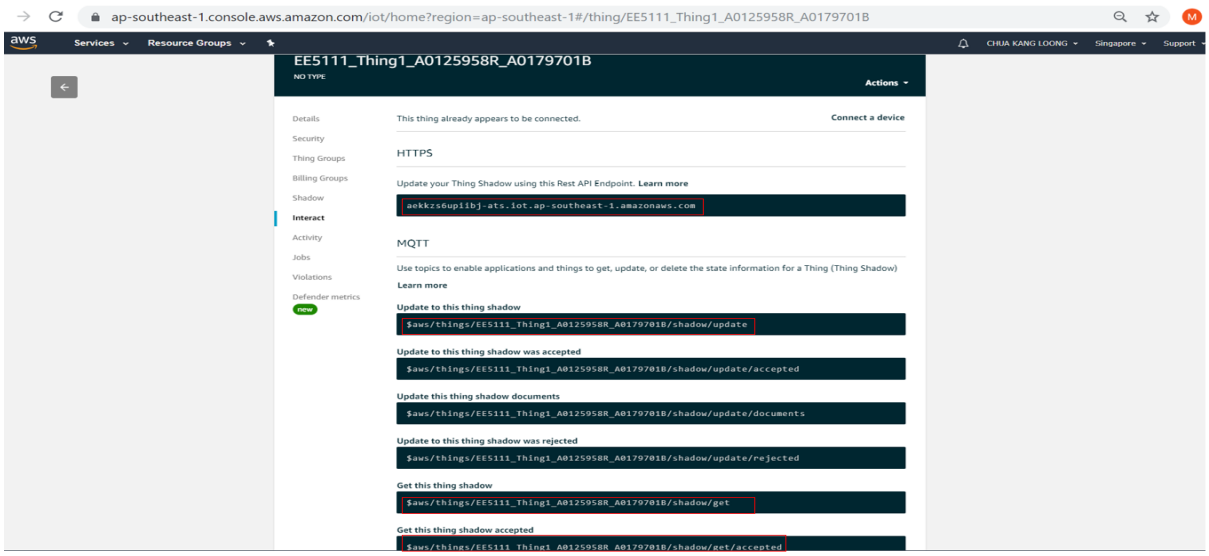

Figure 14 - Things's shadow send and receive data part 2

For Subscription topic, enter the MQTT topic value in Figure 13 of this procedure for Update to thing shadow (for example, $aws/things/EE5111_Thing1_A125958R_A0179701B/shadow/update), and then choose Subscribe to topic. Repeat the same procedure for the MQTT topic values for (for example, $aws/things/EE5111_Thing1_A125958R_A0179701B/shadow/get) and (for example, $aws/things/EE5111_Thing1_A125958R_A0179701B/shadow/get/accepted) as shown in Figure 15.

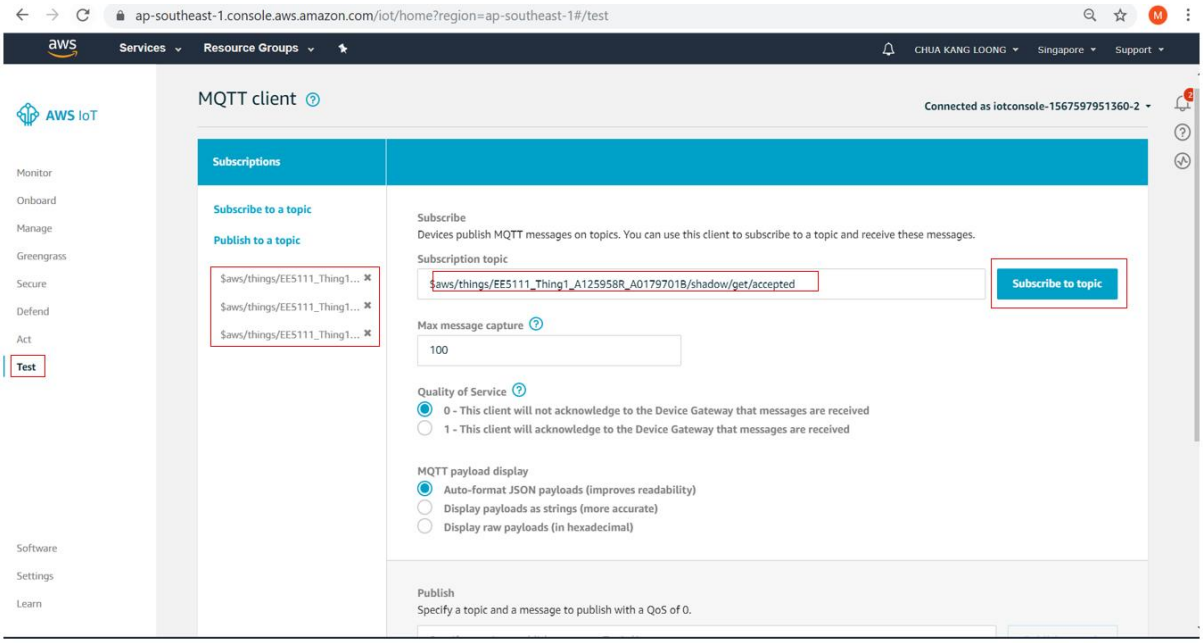

Figure 15 - Things's shadow send and receive data part 3

To get that data from the shadow, choose the MQTT topic value for Get this thing shadow (for example, $aws/things/EE5111_Thing1_A125958R_A0179701B/shadow/get). In the message payload area, replace the current payload with the following payload and 'Publish to topic' as shown in Figure 16 and 17.

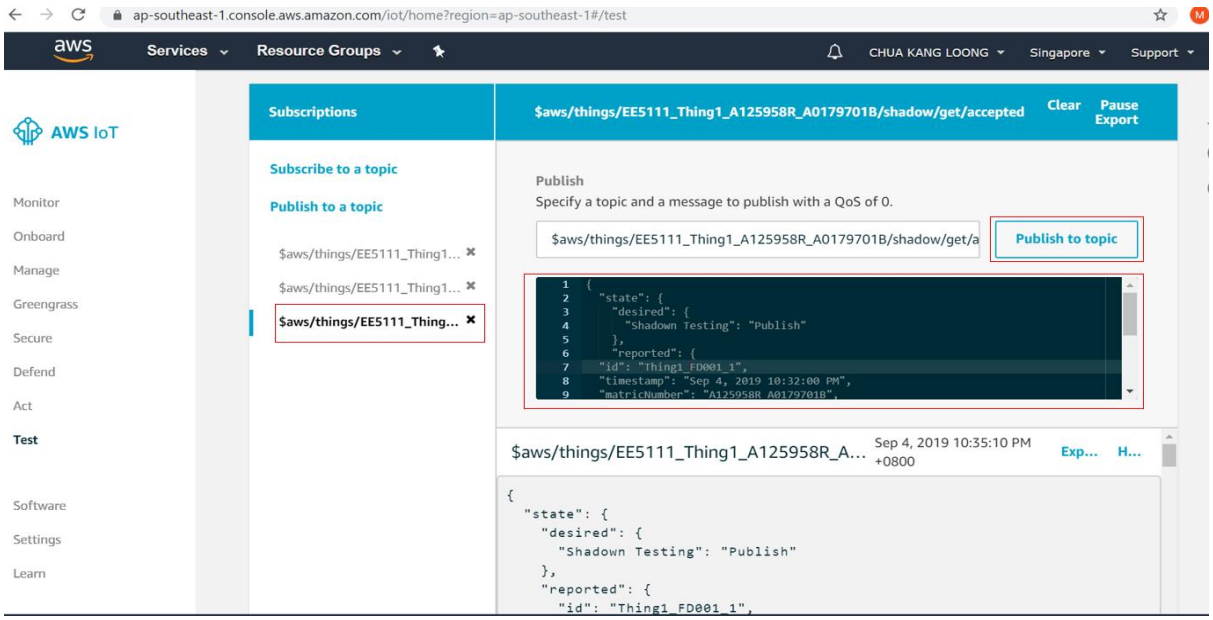

Figure 16 - Things's shadow send and receive data part 4

Figure 17 - JSON document

Step 5: Creating an Amazon DynamoDB Rule

DynamoDB rules allow users to take information from an incoming MQTT message (When running the Python script) and write it to a DynamoDB table. In the AWS IoT console, in the navigation panel, choose ‘Act’ and select ‘Create’ as shown in Figure 18.

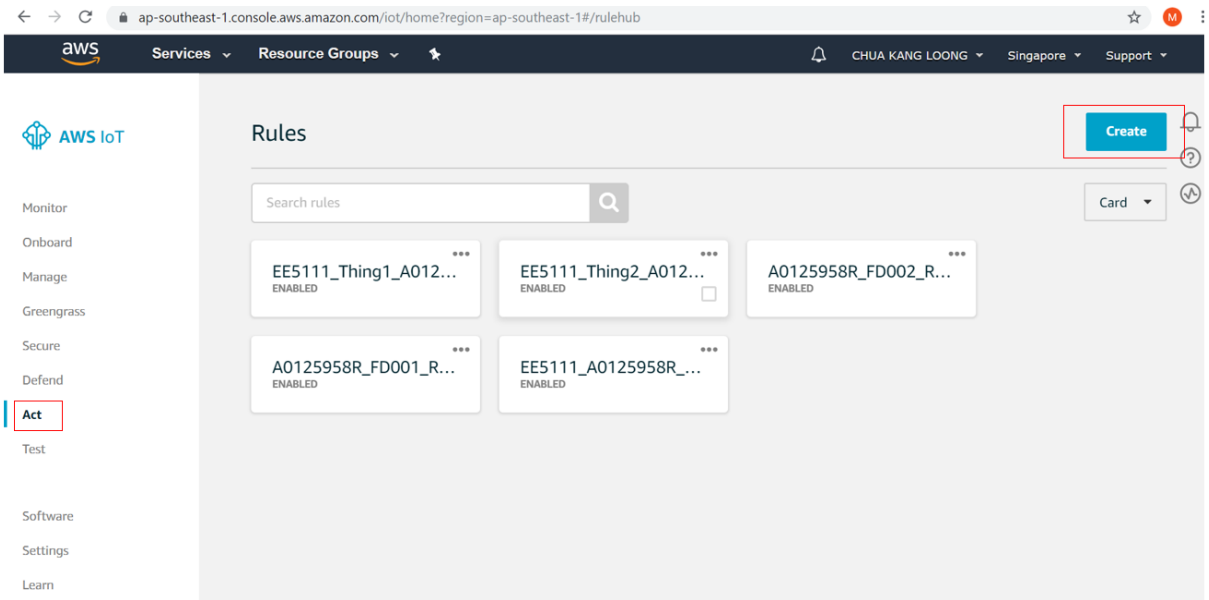

Figure 18 - Amazon DynamoDB part 1

On the Create a rule page, enter a name,description for your rule and select ‘Add action’ to choose ‘Split message into multiple columns of a DynamoDB table (DynamoDBv2)’, and then choose ‘Configure action’ as shown in Figure 19, 20, 21 and 22.

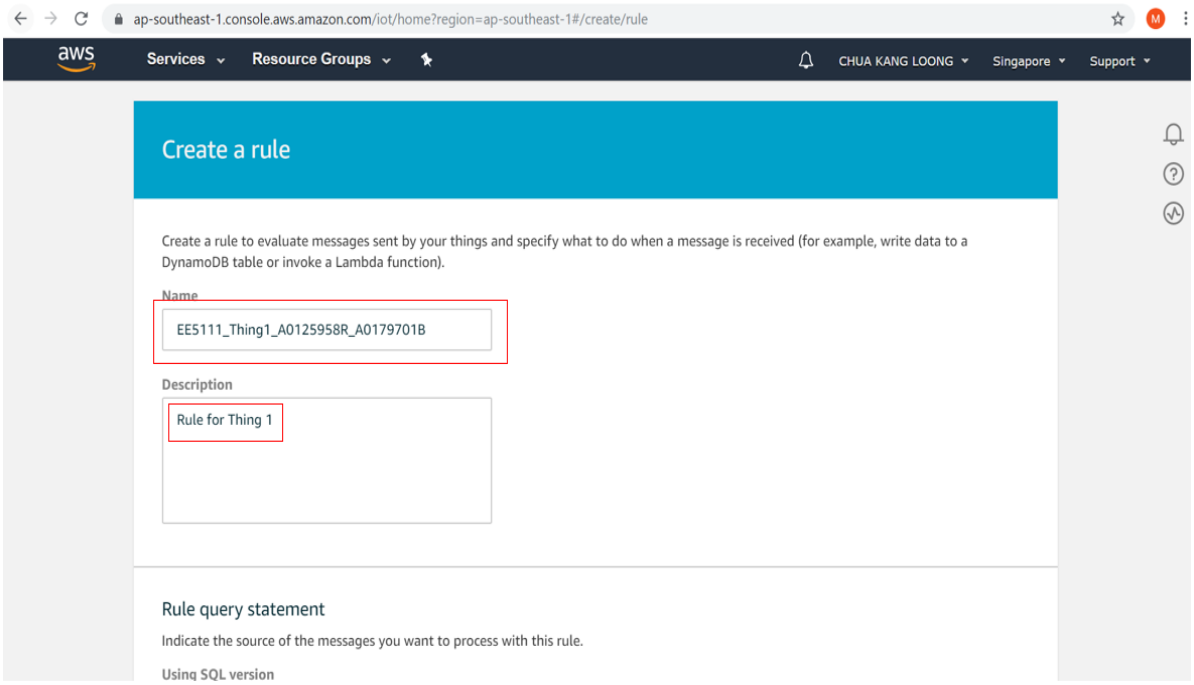

Figure 19 - Amazon DynamoDB part 2

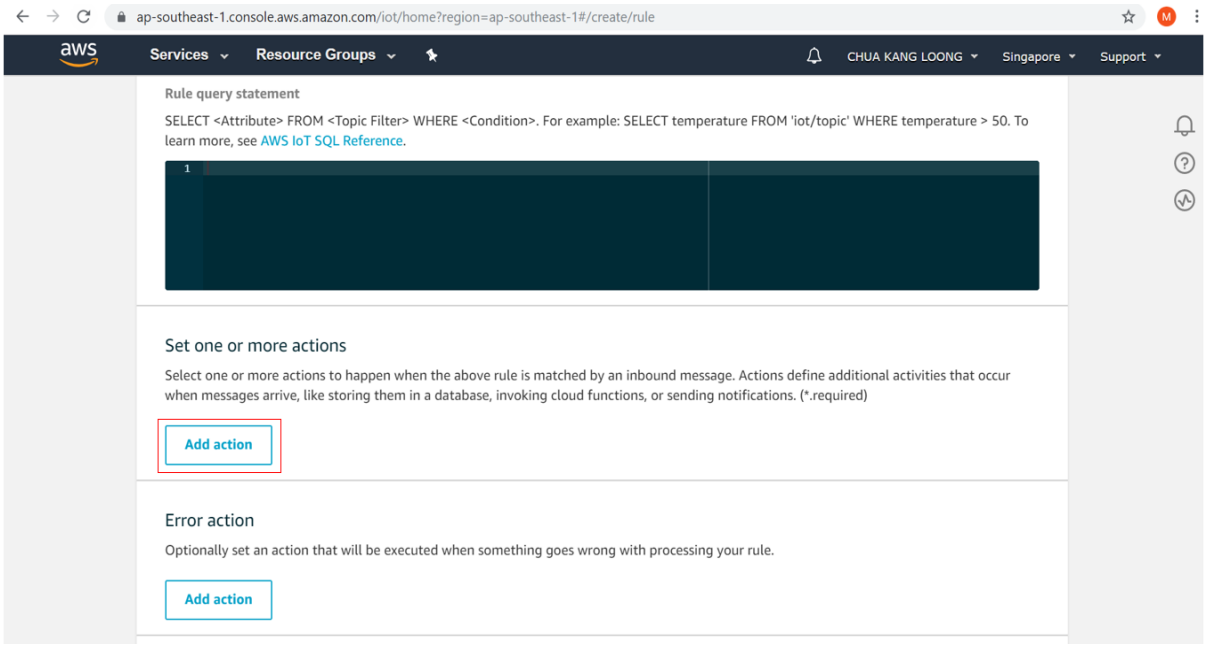

Figure 20 - Amazon DynamoDB part 3

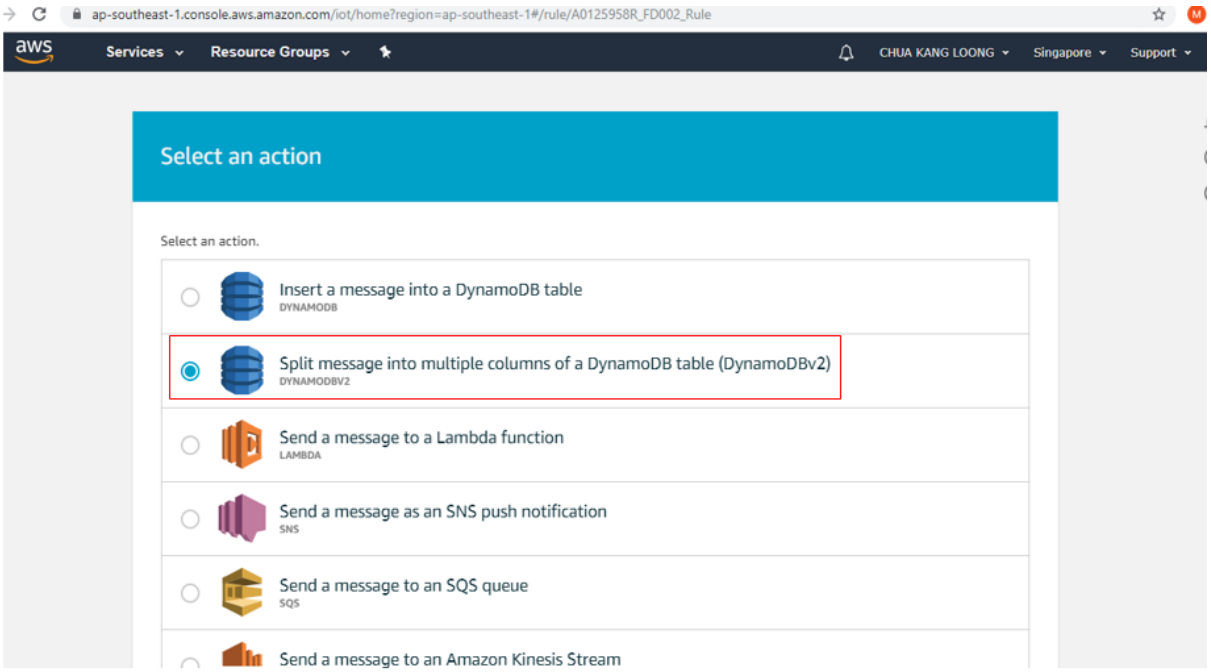

Figure 21 - Amazon DynamoDB part 4

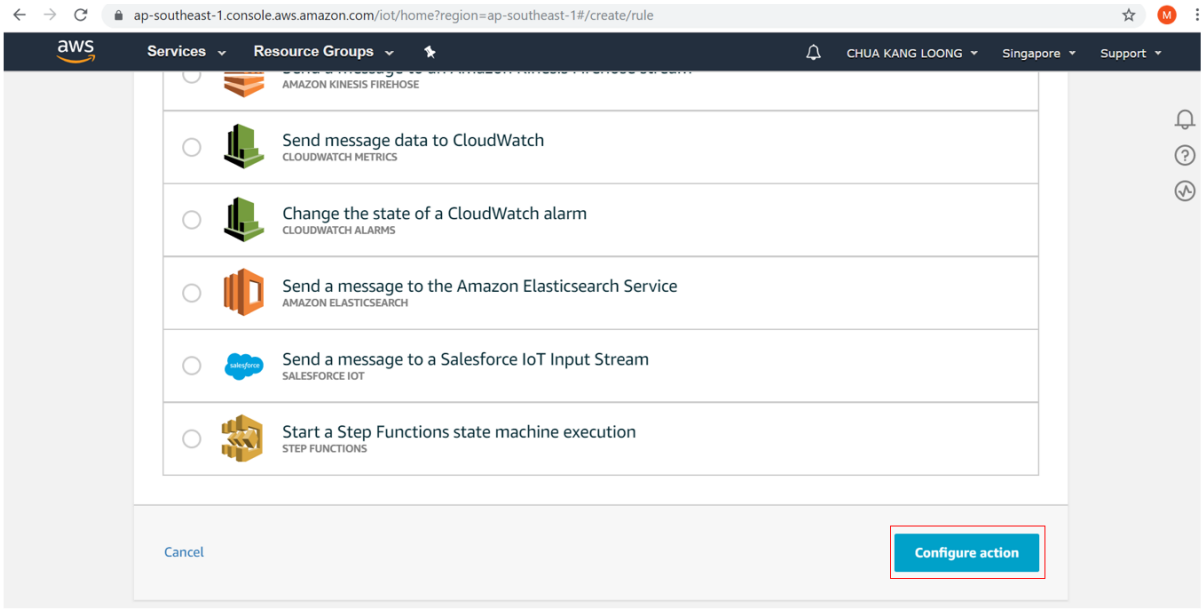

Figure 22 - Amazon DynamoDB part 5

On the Configure action page, choose ‘Create a new resource’. This action will bring you DynamoDB table as shown in Figure 23.

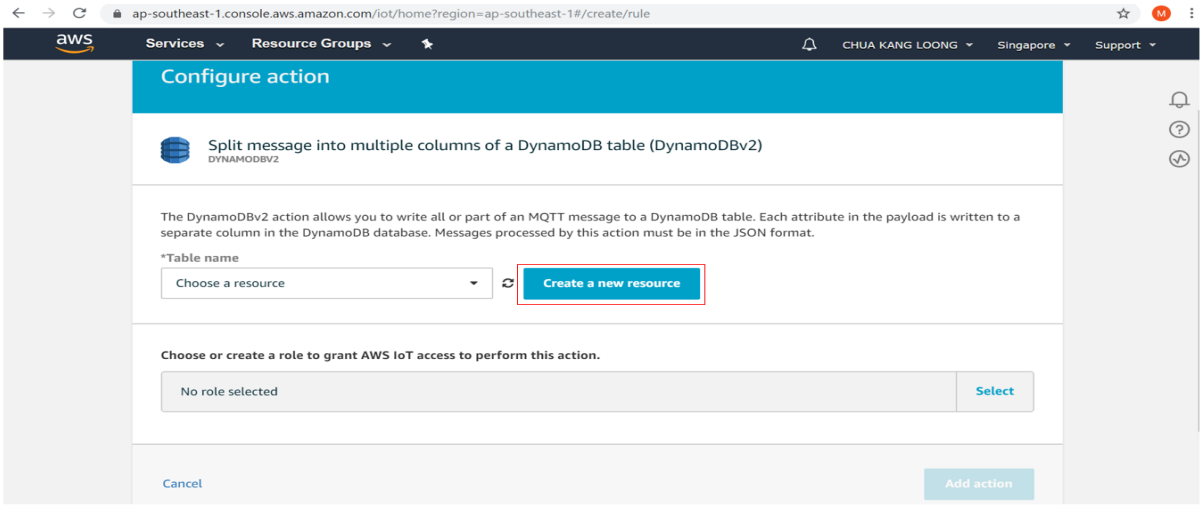

Figure 23 - Amazon DynamoDB part 7

On the Amazon DynamoDB page, choose ‘Create table’ as shown in Figure 24.

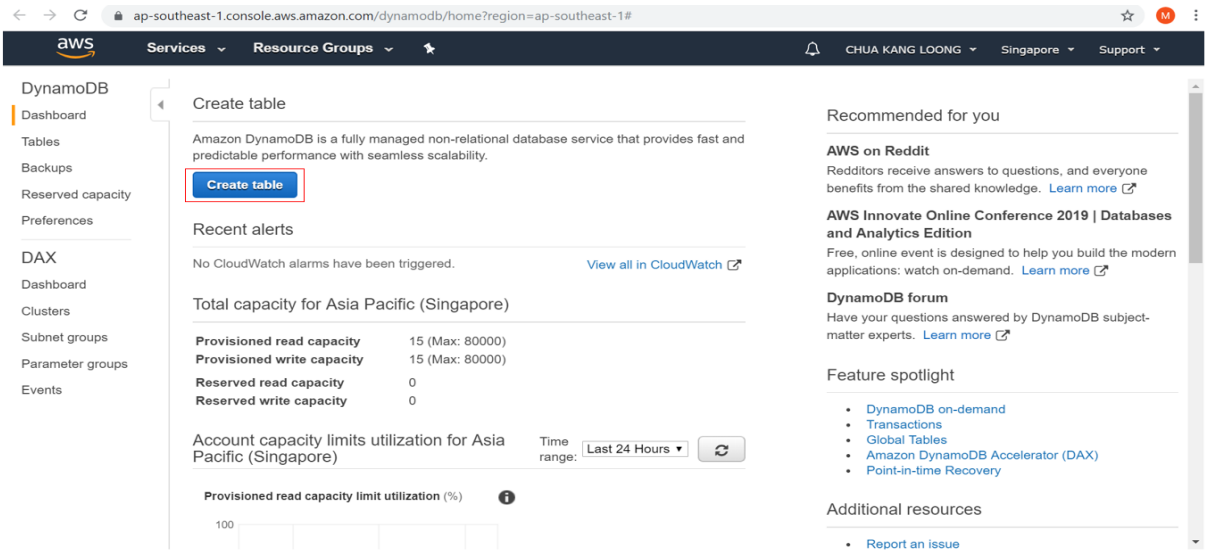

Figure 24 - Amazon DynamoDB part 8

On the Create DynamoDB table page, enter a ‘Name’ in Table name. In Partition key, enter ‘id’. Select Add sort key, and then enter ‘timestamp’ in the Sort key field. Row represents an id of the sensors for jet engine. Timestamp represents the time in UTC (e.g. UTC 2019-01-28 14:41:15.237). Choose String for both the partition and sort keys, and then choose Create. It takes a few seconds to create your DynamoDB table. Close the browser tab where the Amazon

DynamoDB console is open. If you don't close the tab, your DynamoDB table is not displayed in the Table name list on the Configure action page of the AWS IoT console.

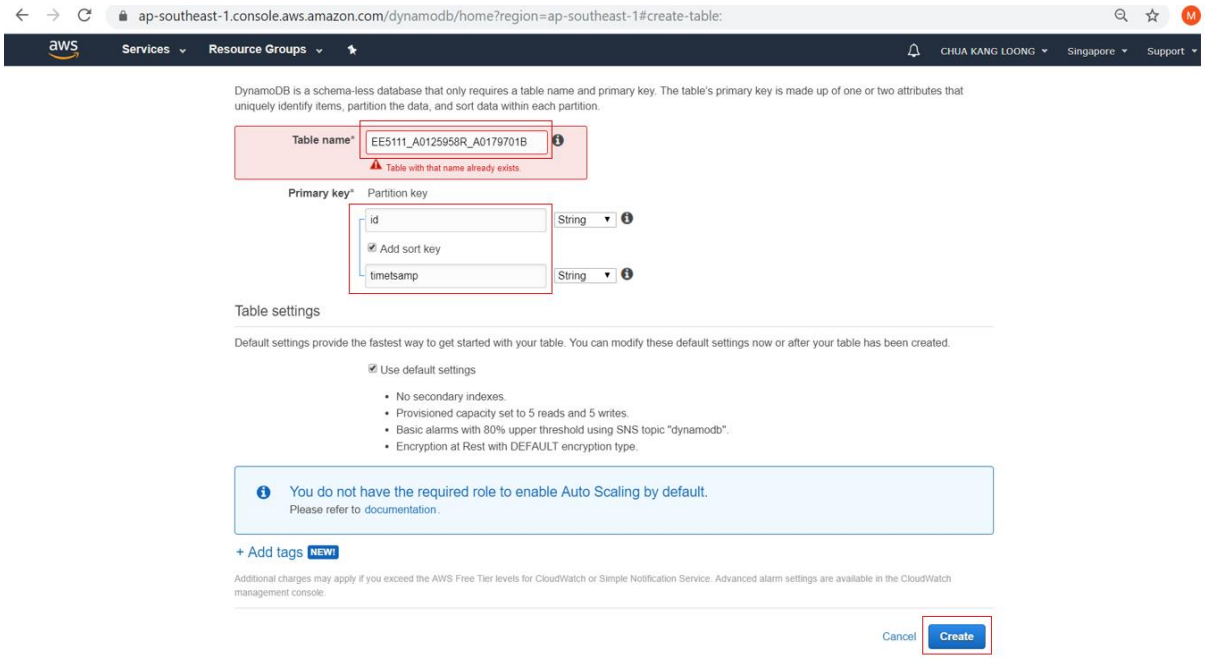

Figure 25 - Amazon DynamoDB part 9

On the Amazon DynamoDB page, choose ‘Manage Stream’ and select ‘Enable’ for DynamoDB table to receive data as shown in Figure 26 and 27.

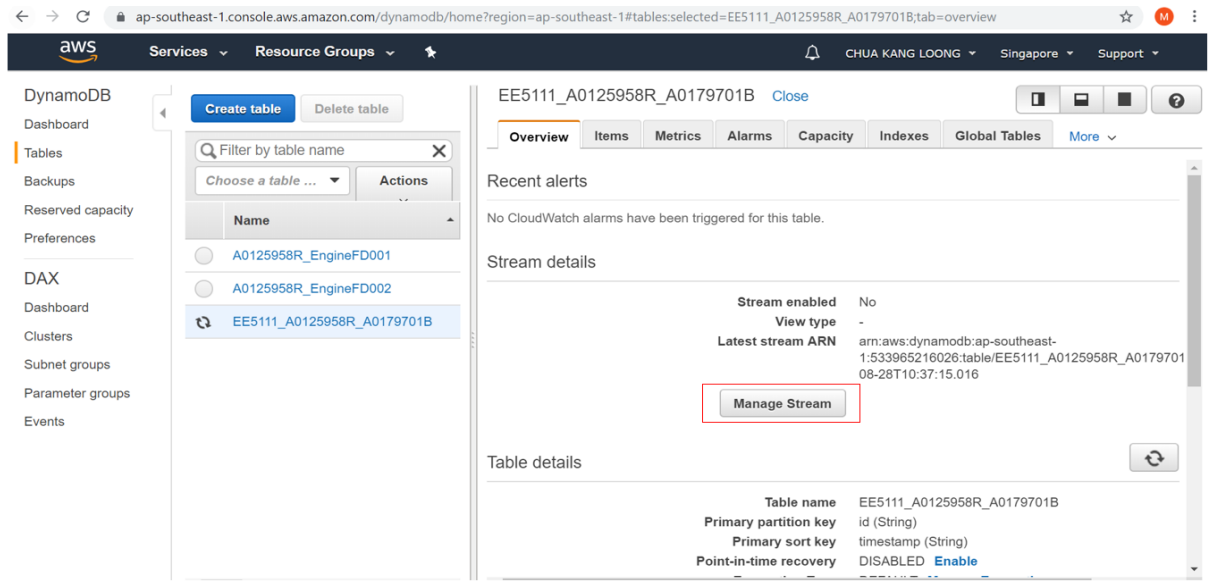

Figure 26 - Amazon DynamoDB part 10

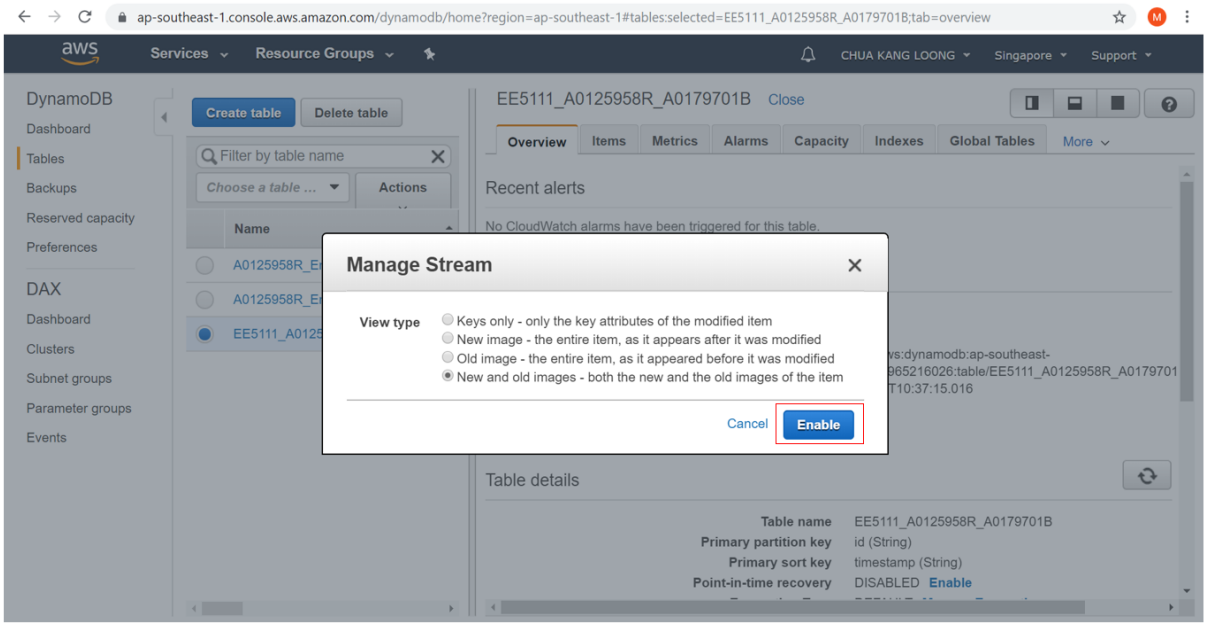

Figure 27 - Amazon DynamoDB part 11

Back on the Configure action page, choose the table created earlier from the Table name list.

Select ‘Create Role’ and attach Policy for the table as shown in Figure 28 and 29.

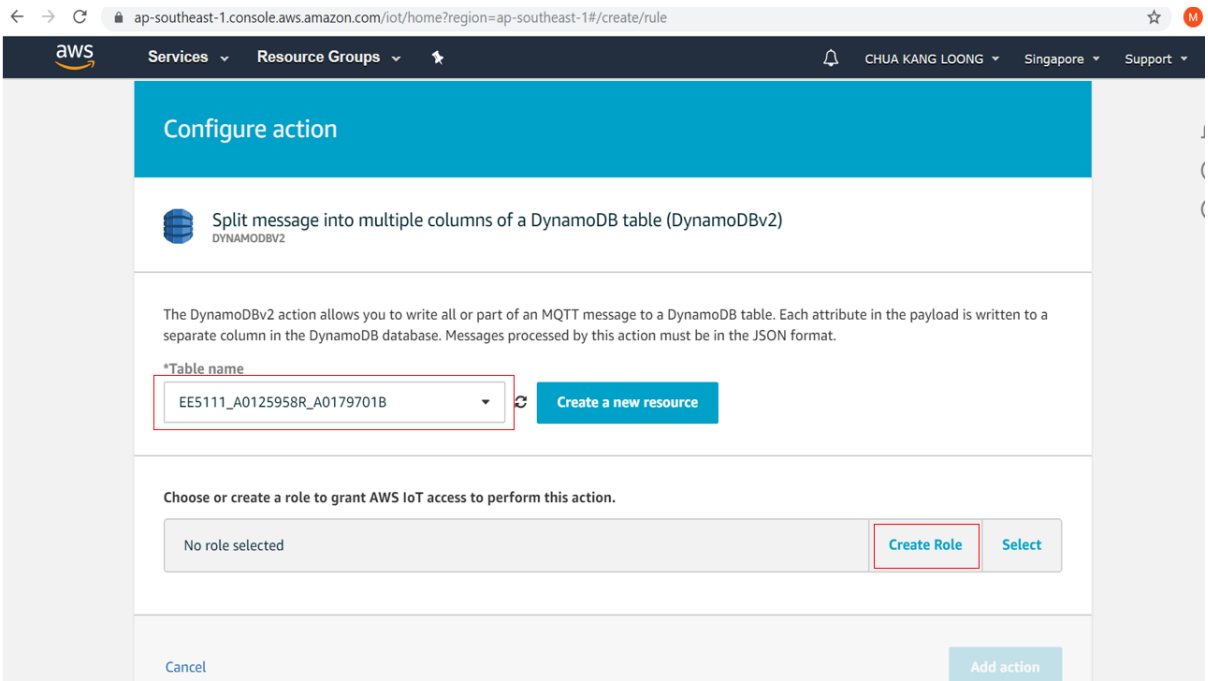

Figure 28 - Amazon DynamoDB part 12

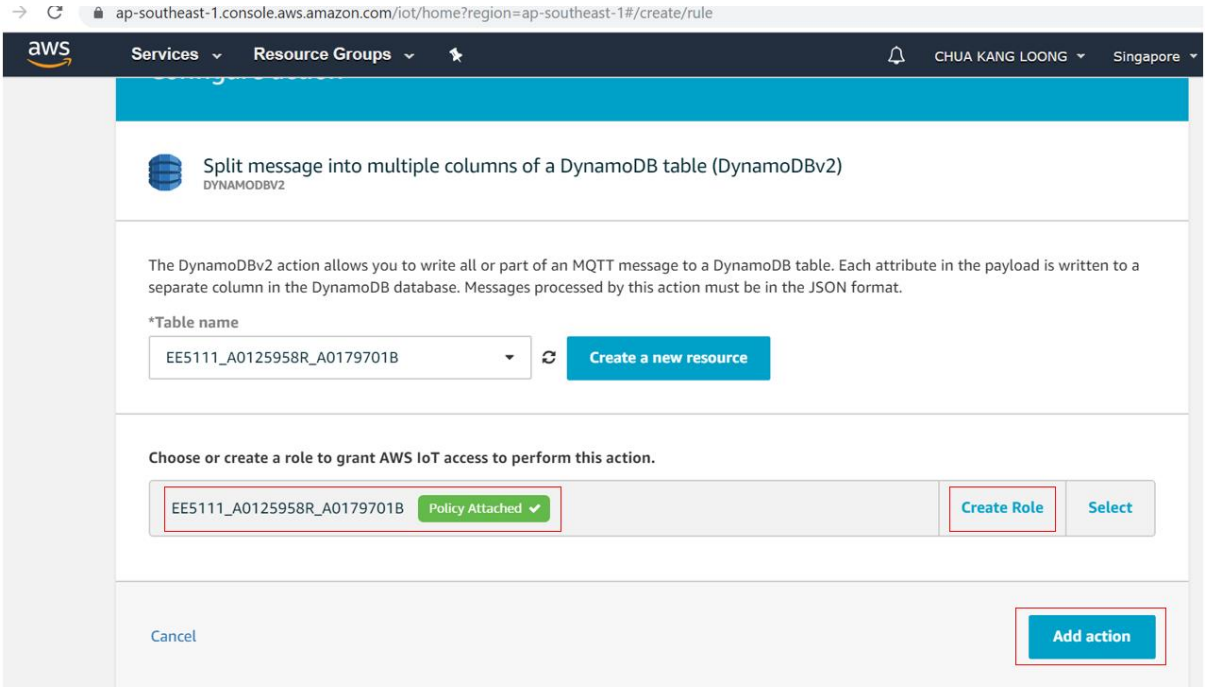

Figure 29 - Amazon DynamoDB part 13

On the Configure action page, enter the JSON message: SELECT state.reported.* FROM '$aws/things/A125958R_A0179701B/shadow/update/accepted' for rule query statement in the window to retrieve data then select ‘Create Rule’ as shown in Figure 30 and 31.

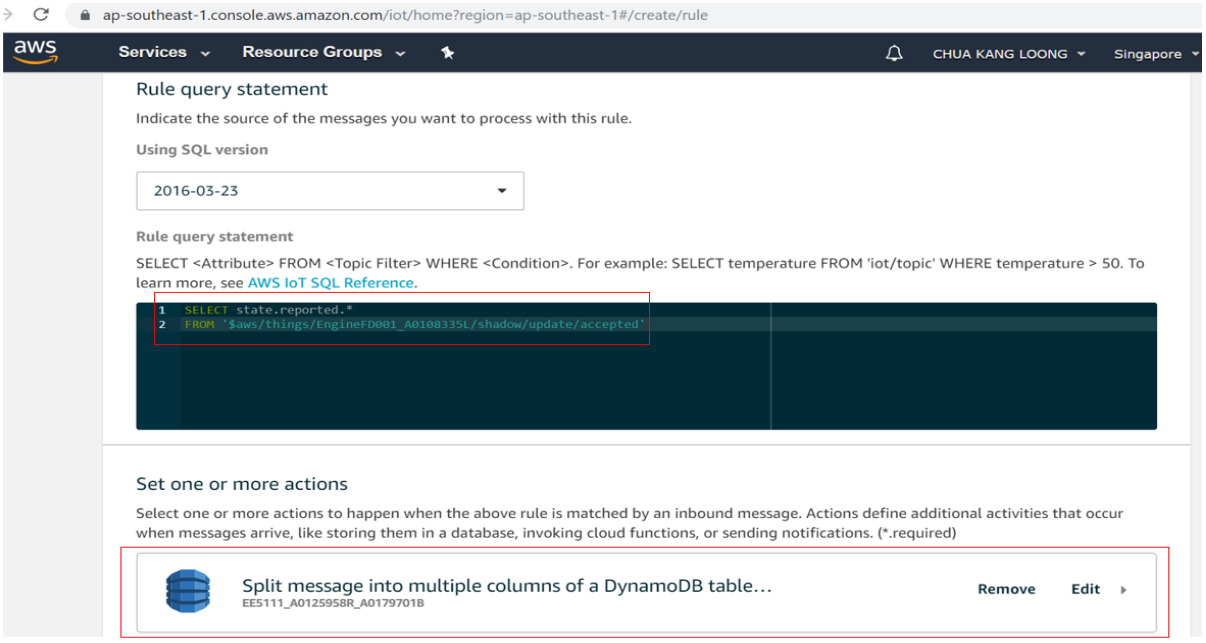

Figure 30 - Amazon DynamoDB part 14

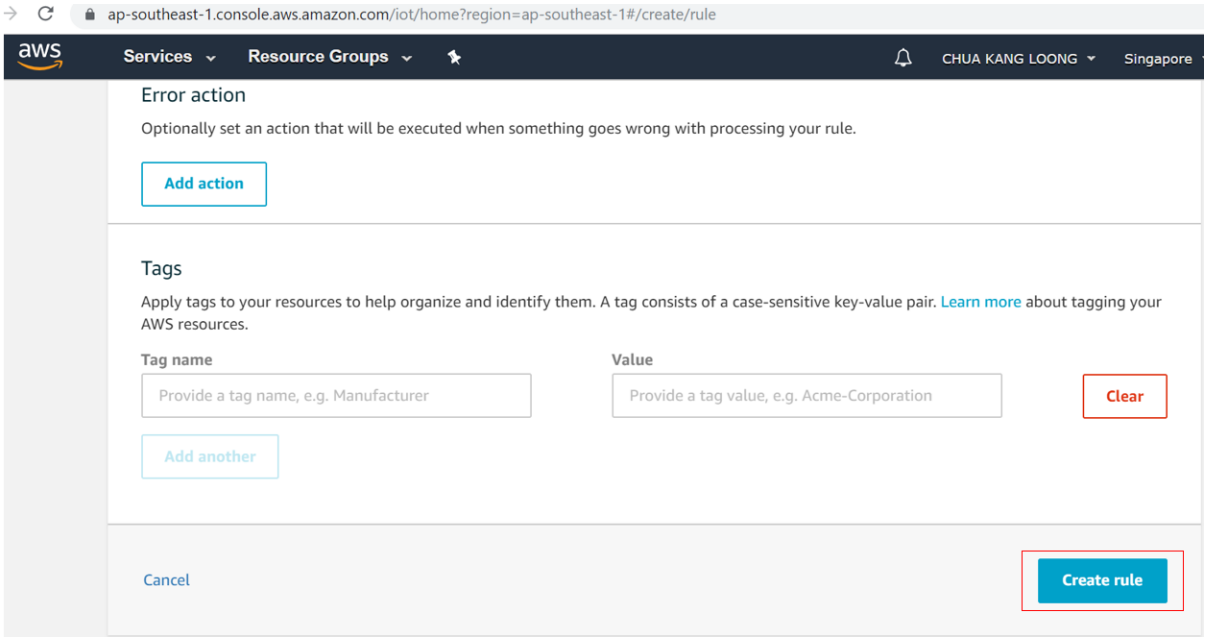

Figure 31 - Amazon DynamoDB part 15

Step 6: Creating Python codes

From the given AWS given skeleton codes, modify the following for Thing1.

#Obtained from the Thing1 ‘Name’

SHADOW_CLIENT = "EE5111_Thing1_A0125958R_A0179701B"

#Obtained from IoT Core Console -> Things -> Interact -> HTTPS

HOST_NAME = "aekkzs6upiibj-ats.iot.ap-southeast-1.amazonaws.com"

#Download from https://github.com/aws-samples/aws-iot-jitp-sample-scripts/blob/master/certs/AmazonRootCA1.pem and put it together with the Python codes in a folder

ROOT_CA = "AmazonRootCA1.pem"

#Obtained from certificate when creating Thing1

PRIVATE_KEY = "46de85e2bd-private.pem.key"

#Obtained from certificate when creating Thing1

CERT_FILE = "46de85e2bd-certificate.pem.crt"

#Obtained from certificate when creating Thing1

SHADOW_HANDLER = "EE5111_Thing1_A0125958R_A0179701B"

The codes are able to read and publish data from trainFD001.txt to your thing under AWS IoT platform at the rate of 10 seconds per row. Overwrite column 'id' of the engine as 'FD001' + id (e.g. FD001_12), add one more columns 'timestamp' as timestamps in UTC (e.g. UTC 2019-01-28 14:41:15.237), add one more column that containing Matric number, concatenate data into JSON string and send it to Thing1 shadow handler as shown with comments in Figure 32, 33 and 34. Full codes of FD001 Thing1 and FD002 Thing2 are in the appendix of this report.

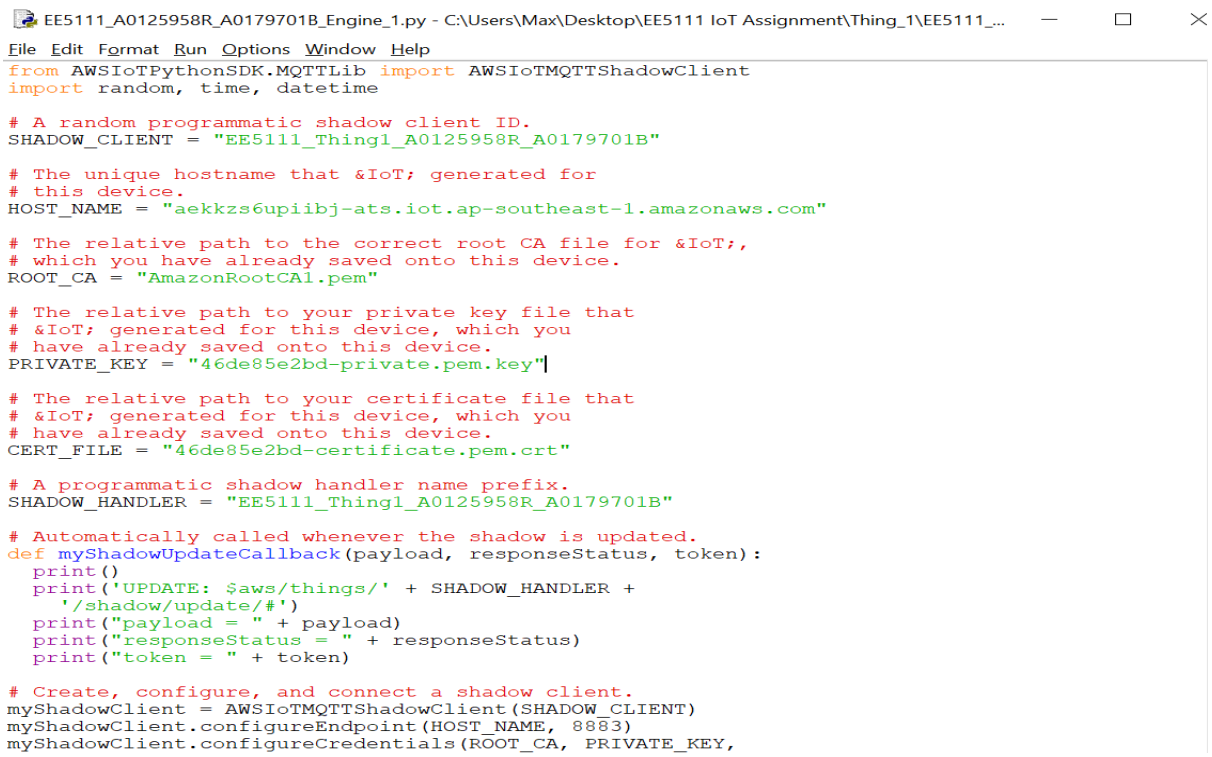

Figure 32- Python code part 1

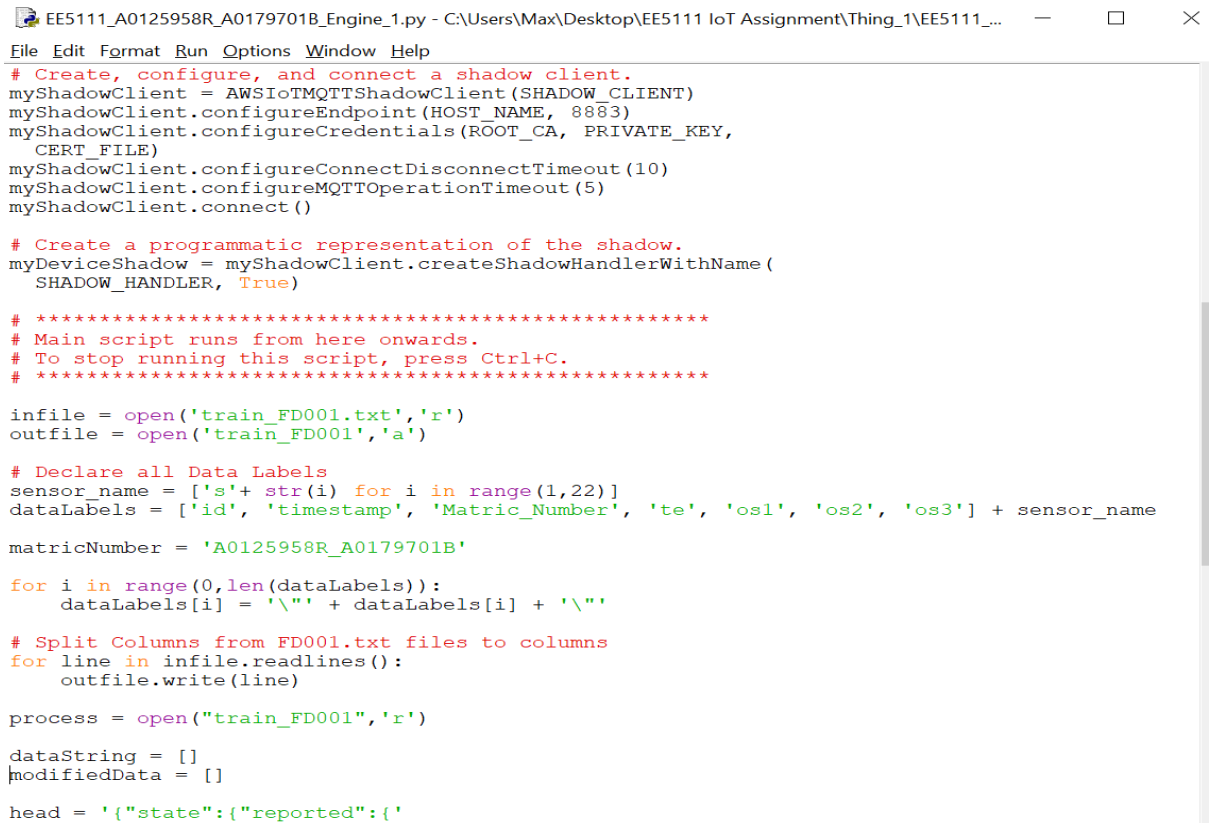

Figure 33- Python code part 2

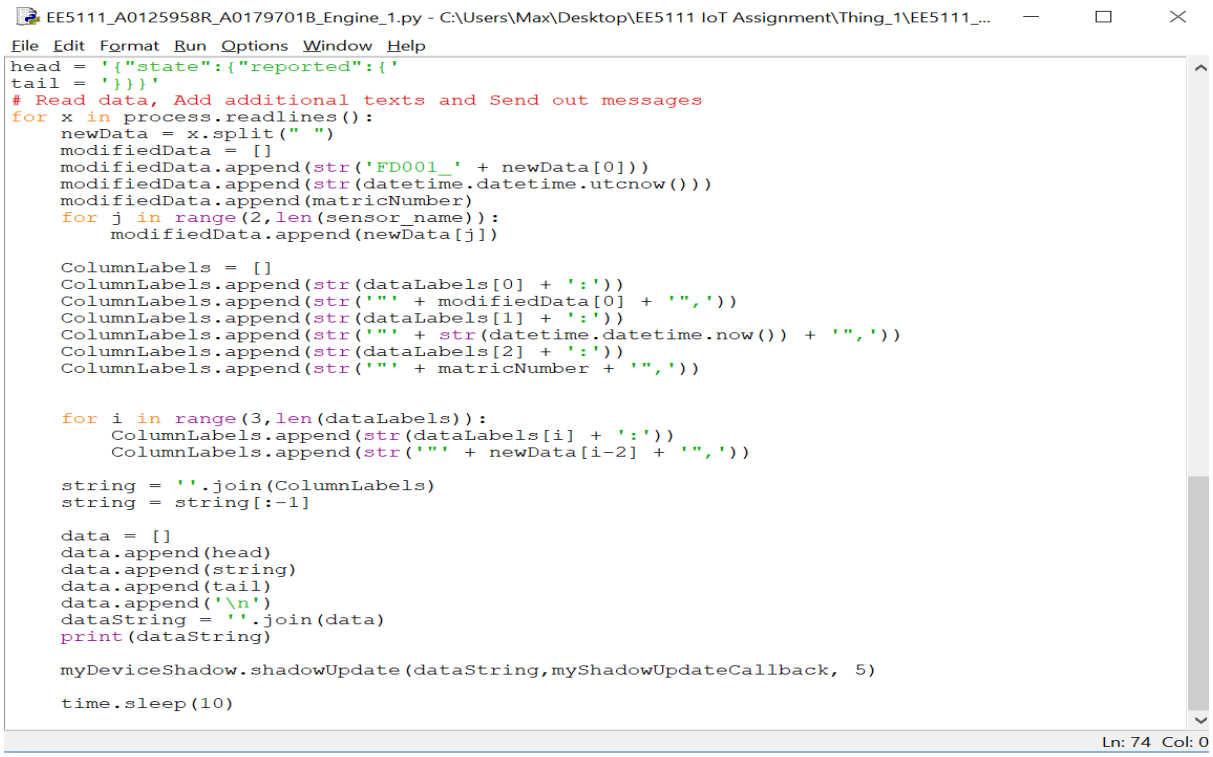

Figure 34 - Python code part 3

Step 7: Executing Python codes and streaming data to AWS DynamoDB table

Run the Python codes and shown in Figure 35 and an update will appear for every 10 seconds interval on Python Shell upon successful streaming of the FD001 engine data to AWS DynamoDB table. To view the published data, go to AWS DynamoDB table and select ‘Items’ as shown in Figure 36.

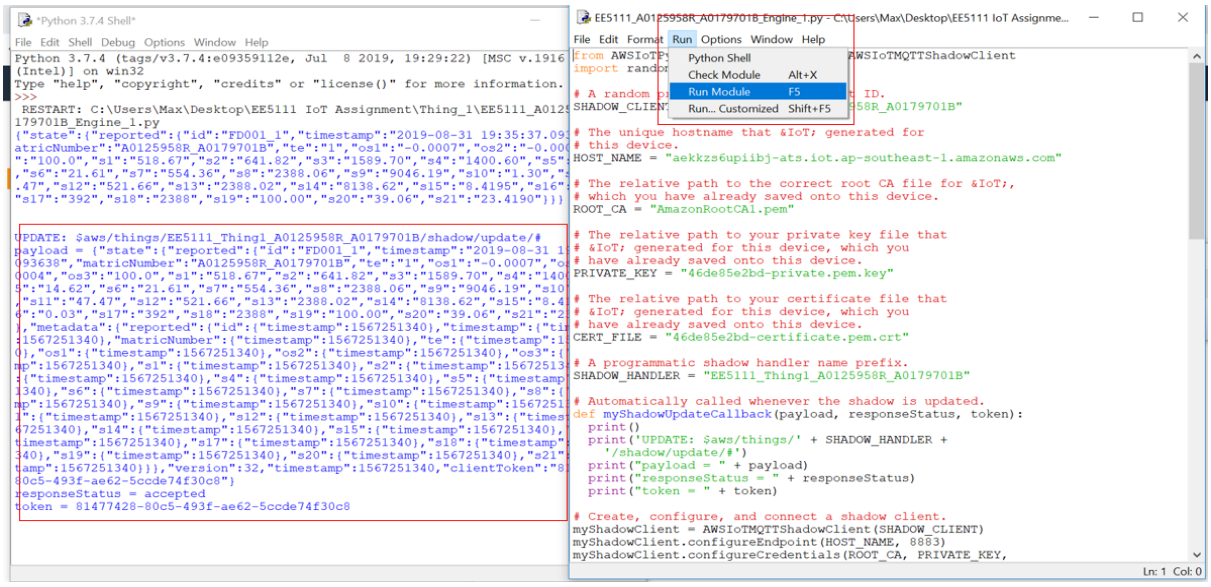

Figure 35 - Executing Python for Thing1

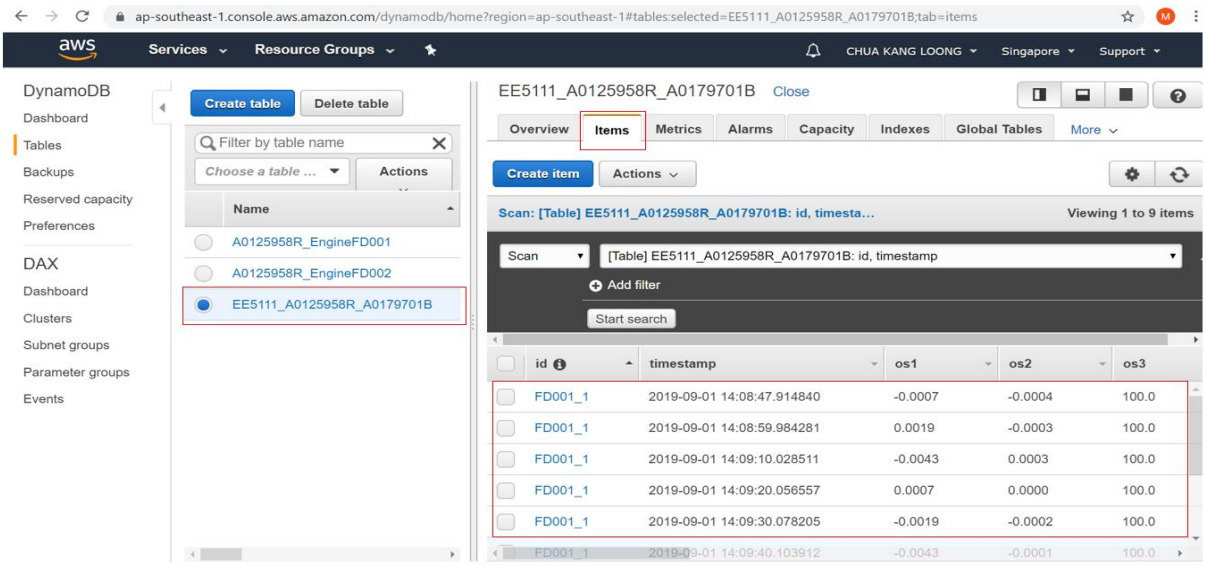

Figure 36 - Python code part 3

Create Thing2 for FD002

To simulate two IoT things now, add one more thing under AWS IoT platform and one more certificate, create a copy of the Jupyter notebook above, renaming the client name, certificate, data source train_FD002.txt, modify the rule under AWS IoT platform to be triggered by '$aws/things/+/shadow/update/accepted' as shown in Thing1 for FD001 step 1 - 6. Now, this rule will push data of both engines from both things. Let the two Jupyter notebooks run at the same time as shown in Figure 37, simulating the two "things" to run in parallel to publish data with a sampling rate of one record per 10 seconds in each thing.

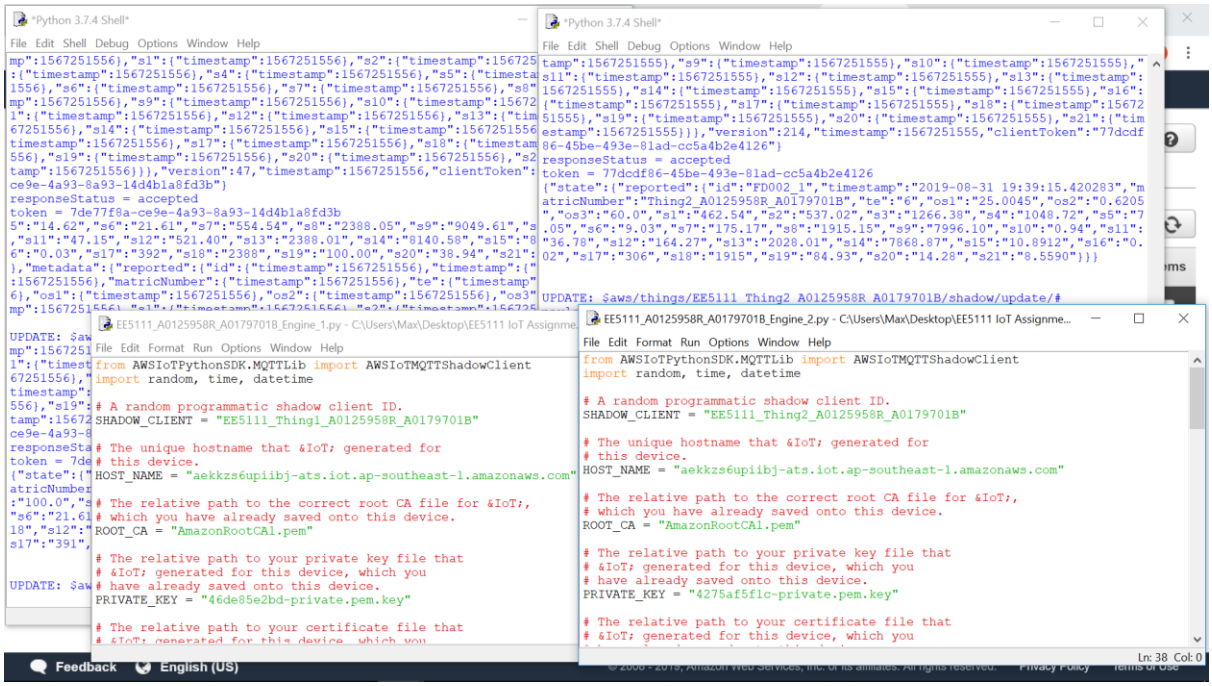

Figure 37 - Executing Python for Thing1 and Thing2

To view both jet engines data for Thing1 and Thing2, go toAmazon DynamoDB page select ‘Items’, click on ‘Refresh’ and click on ‘timestamp’ to view FD001 and FD002 data as shown in Figure 38.

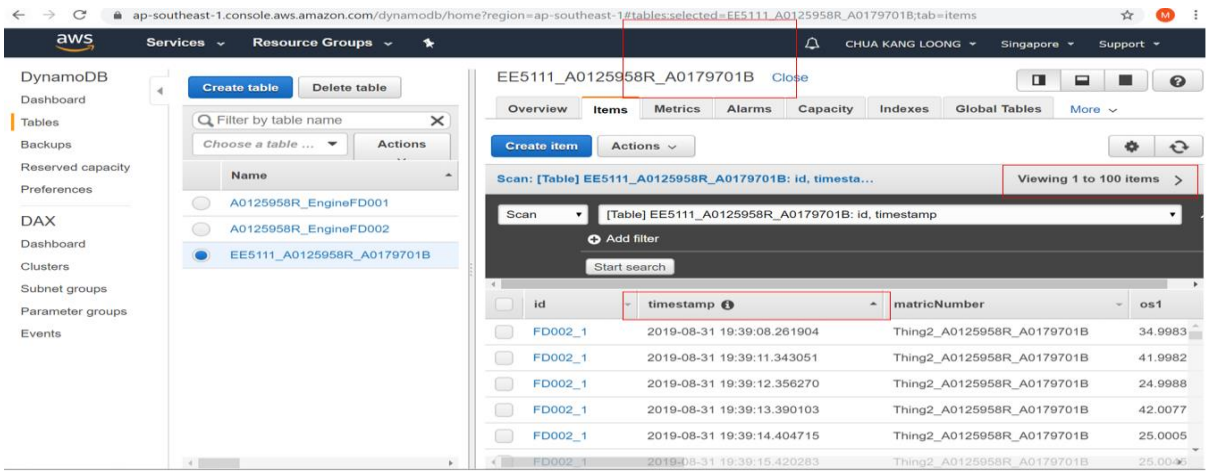

Figure 38 - AWS DynamoDB Table for Thing1 and Thing2

Data visualization on Redash for both Thing1 and Thing2

IAM Policies:

Using Identity-Based Policies (IAM Policies) for Amazon DynamoDB allows an account administrator can attach permissions policies to IAM identities (that is, users, groups, and roles) and thereby grant permissions to perform operations on Amazon DynamoDB resources. From IoT console interface, search for ‘IAM’, select ‘Users’ and click on ‘Add user’ as shown in Figure 39.

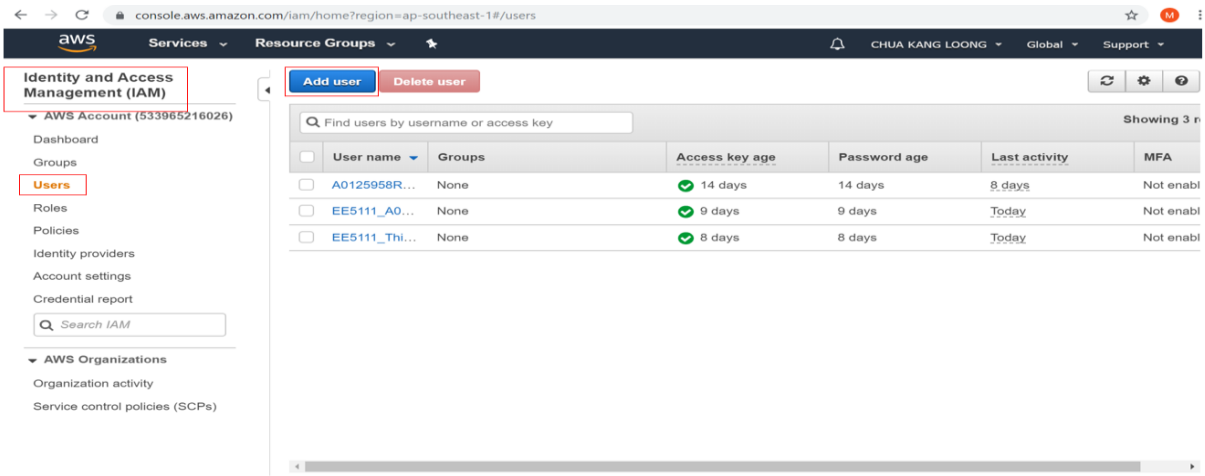

Figure 39 - AWS IAM Table for Things part 1

Enter ‘User name’,check the checkboxes for ‘Programmatic access’ and ‘AWS Management Console access’ and select ‘Next Permission’ as shown in Figure 40.

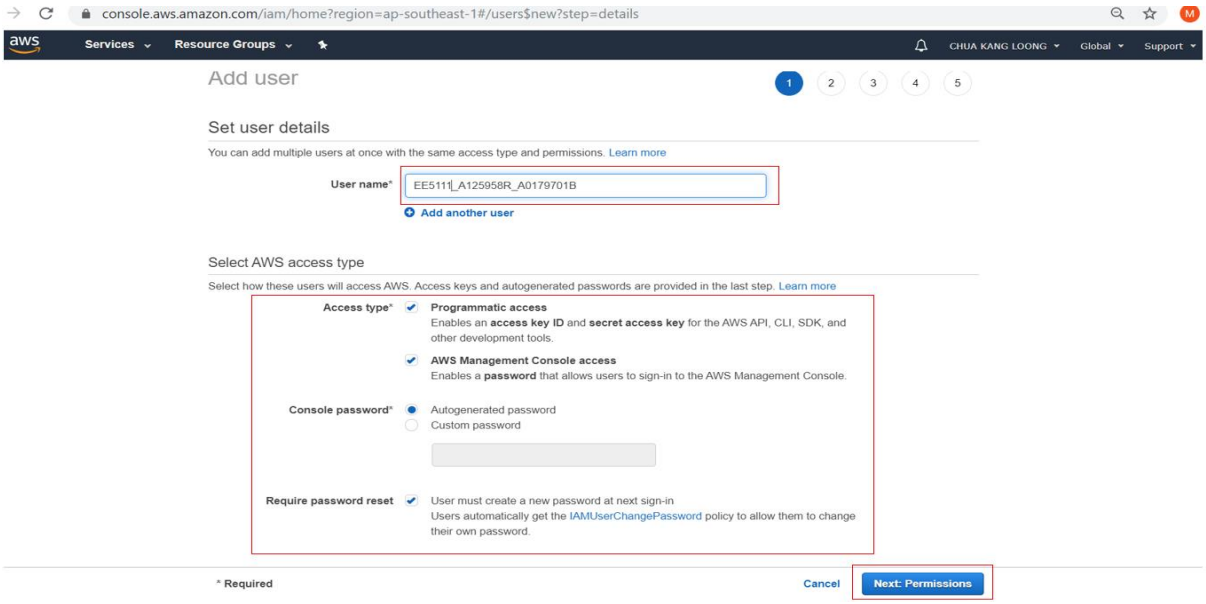

Figure 40 - AWS IAM Table for Things part 2

Select ‘Attach existing policies directly’ and search for ‘AmazonDynamo Db’ under Filter policies. Check the 3 checkboxes, click on ‘Next Tags’, ‘Next Review’ and ‘Create user’ as shown in Figure 41, 42 and 43.

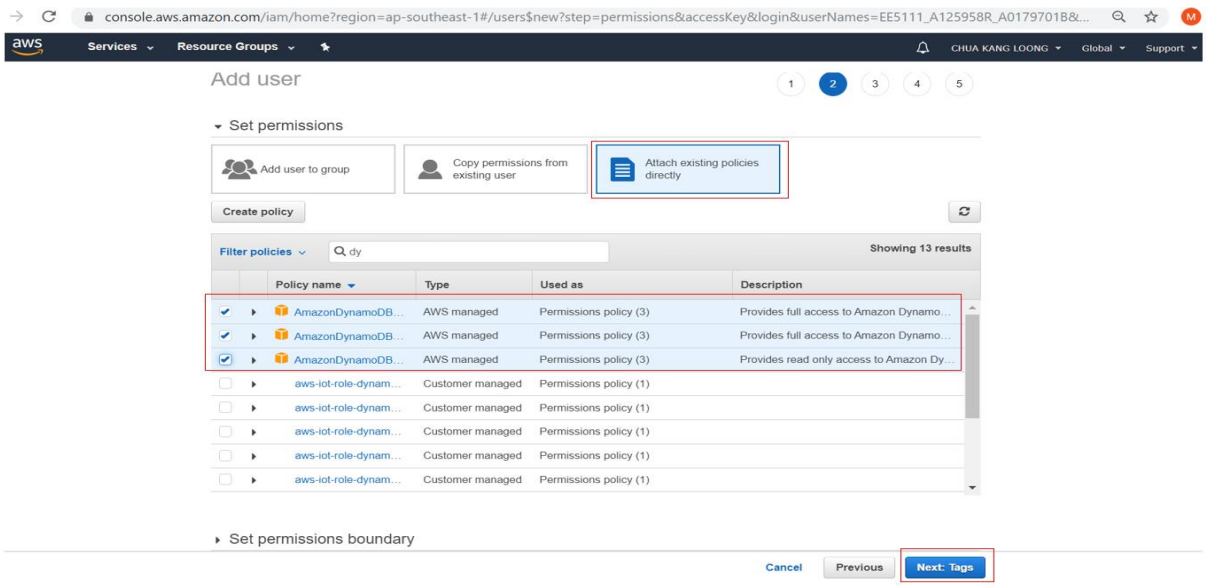

Figure 41 - AWS IAM Table for Things part 3

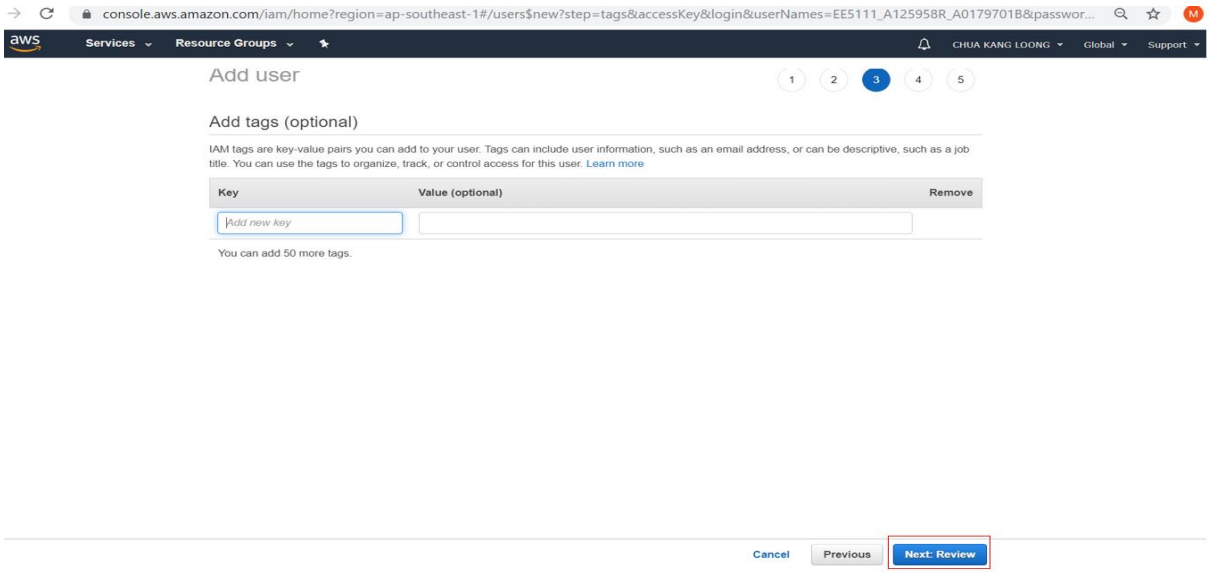

Figure 42 - AWS IAM Table for Things part 4

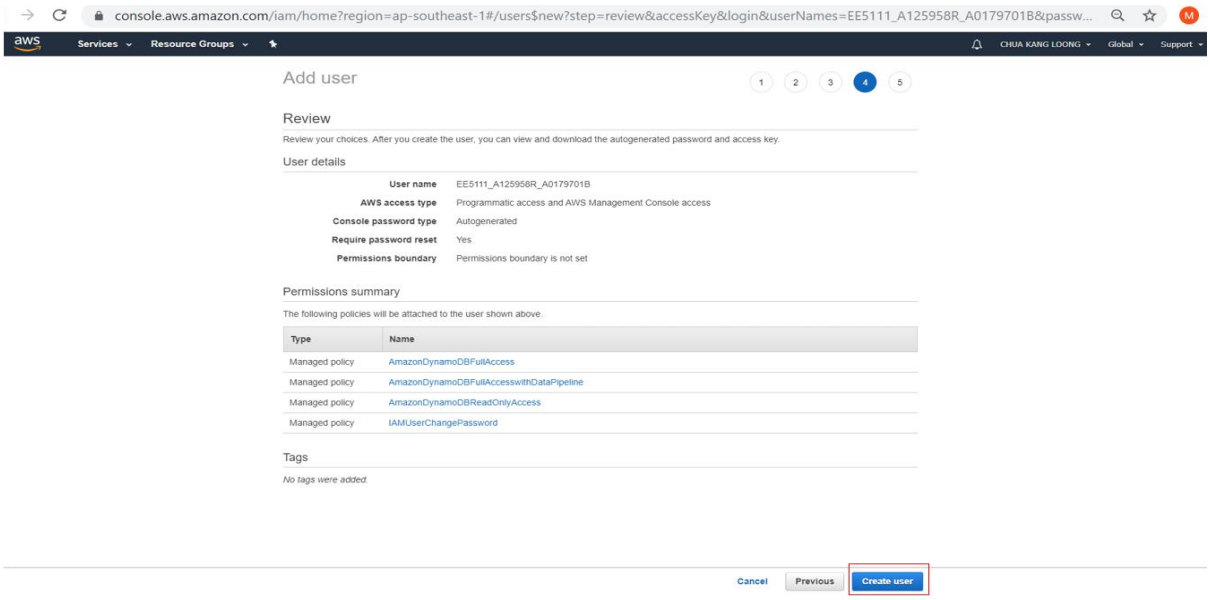

Figure 43 - AWS IAM Table for Things part 5

Download the ‘credential’ file containing User name, Password, Access key ID, Secret access key and Console login link by clicking on ‘Download .csv’ as shown in Figure 44. The authentication access information will be used later on for data visualization on Redash platform.

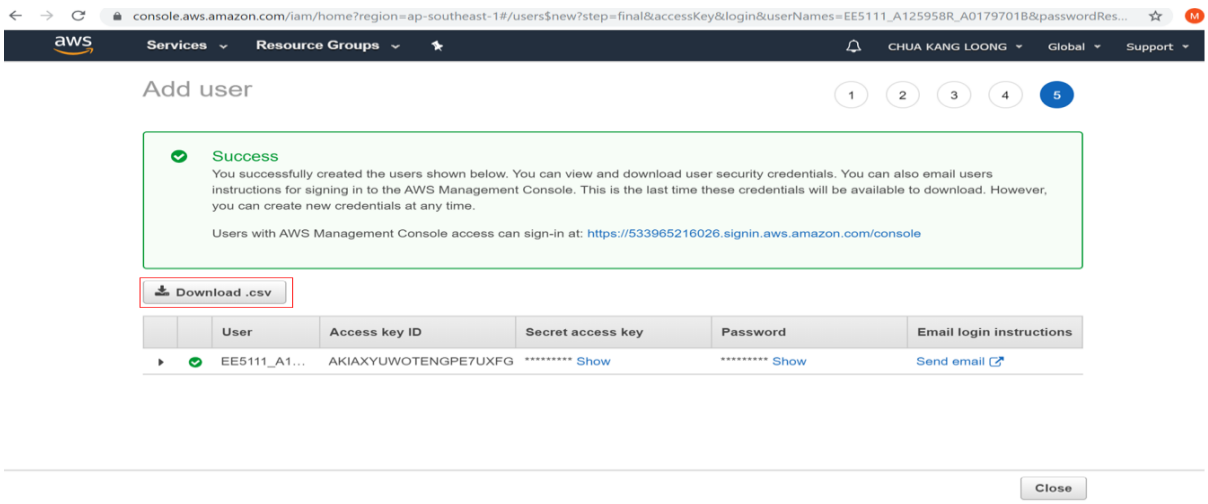

Figure 44 - AWS IAM Table for Things part 6

Redash an open source tool used for Thing1 and Thing2 in this assignment by connecting and querying data sources from AWS DynamoDB, it can also build dashboards to visualize data and share them with the public through URLs. Redash is quick to setup and works with any data source to begin, sign up with Redash and 'Create Your Account' as shown in Figure 45.

Redash data visualization

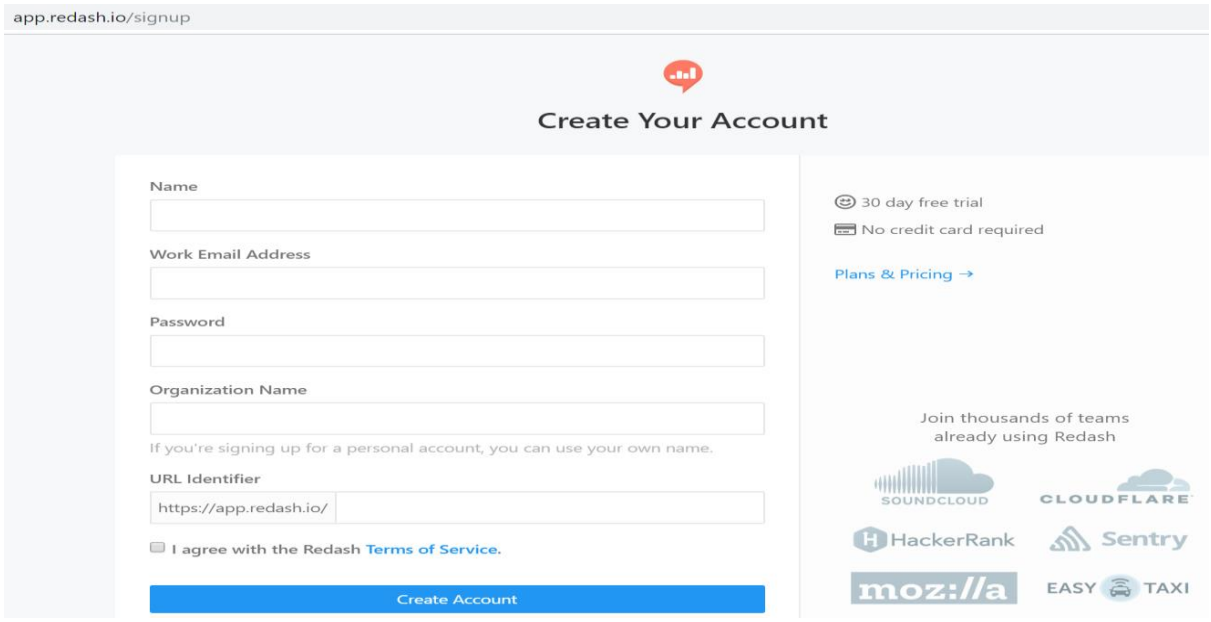

Figure 45 - Redash data visualization part 1

After creating an account, a new query can be created at Redash main page as shown in Figure 46.

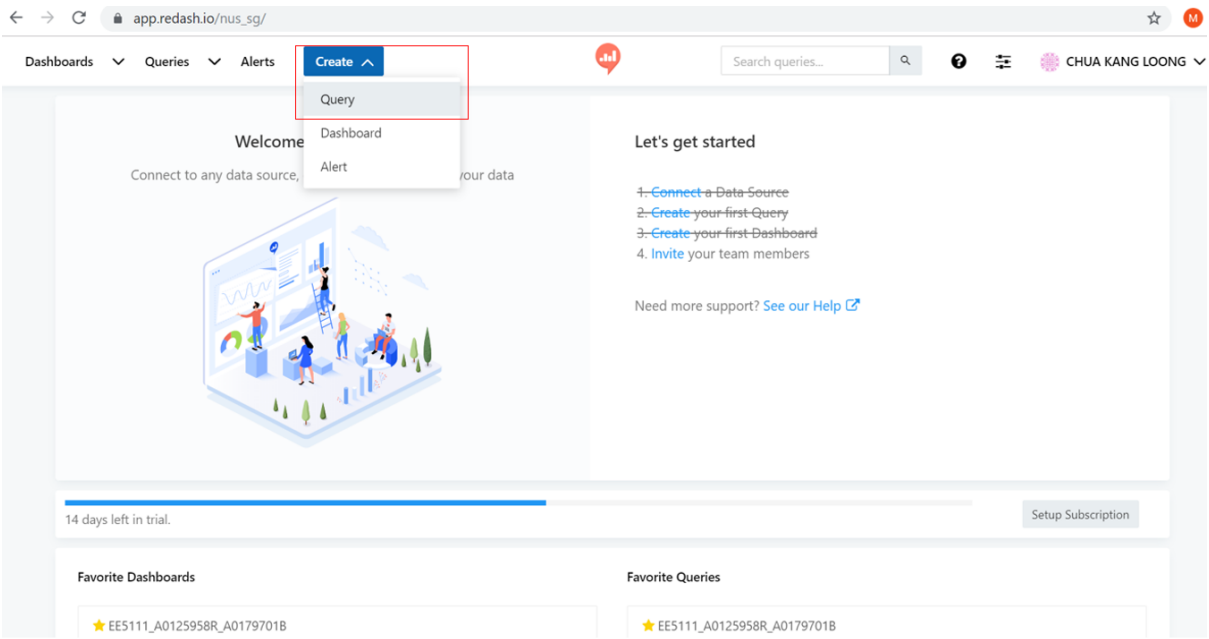

Figure 46 - Redash data visualization part 2

To link Redash query to AWS DynamoDB table, go to ‘Data Sources’ and select ‘New Data Source’ under settings as shown in Figure 47.

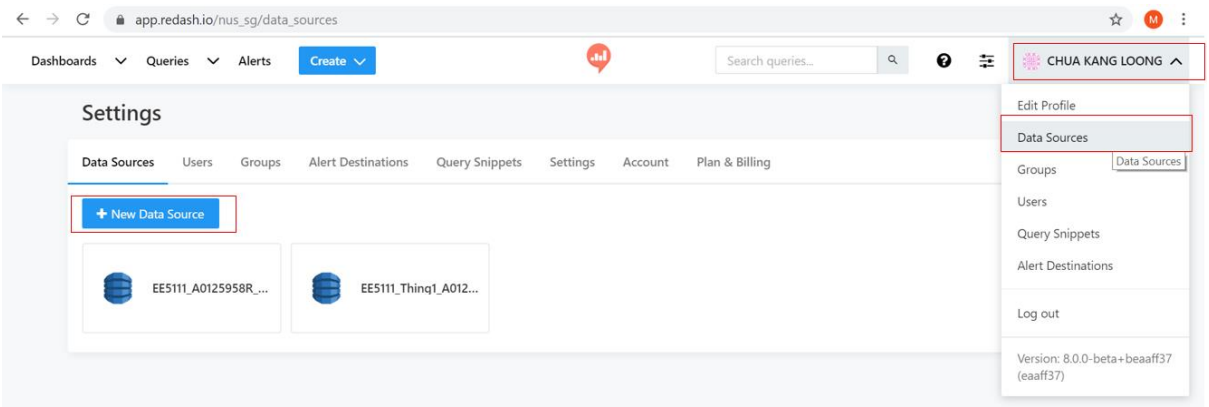

Figure 47 - Redash data visualization part 3

Under ‘Create a New Data Source’ search for ‘DynamoDB (with DQL)’ and select ‘Create’ as shown in Figure 48.

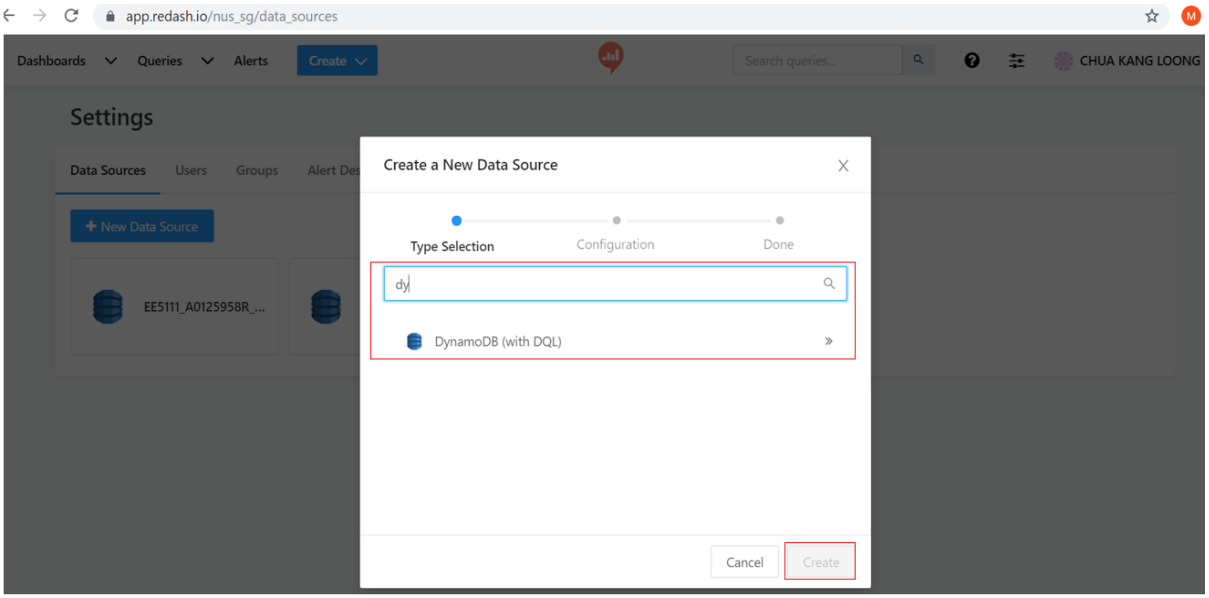

Figure 48 - Redash data visualization part 4

Enter all 4 rows namely ‘Name’, ‘Access Key, ‘Region’ and ‘Secret Key’ which can be obtained from ‘credential’ csv downloaded earlier and select ‘Create’ as shown in Figure 49 and 50.

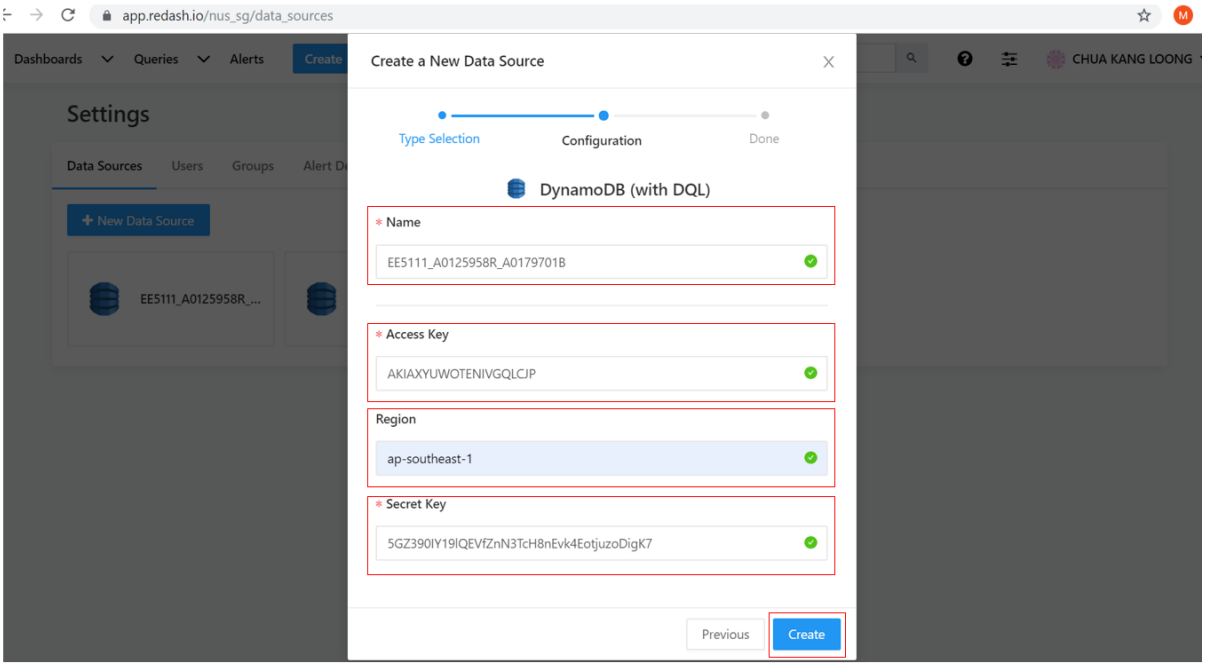

Figure 49 - Redash data visualization part 5

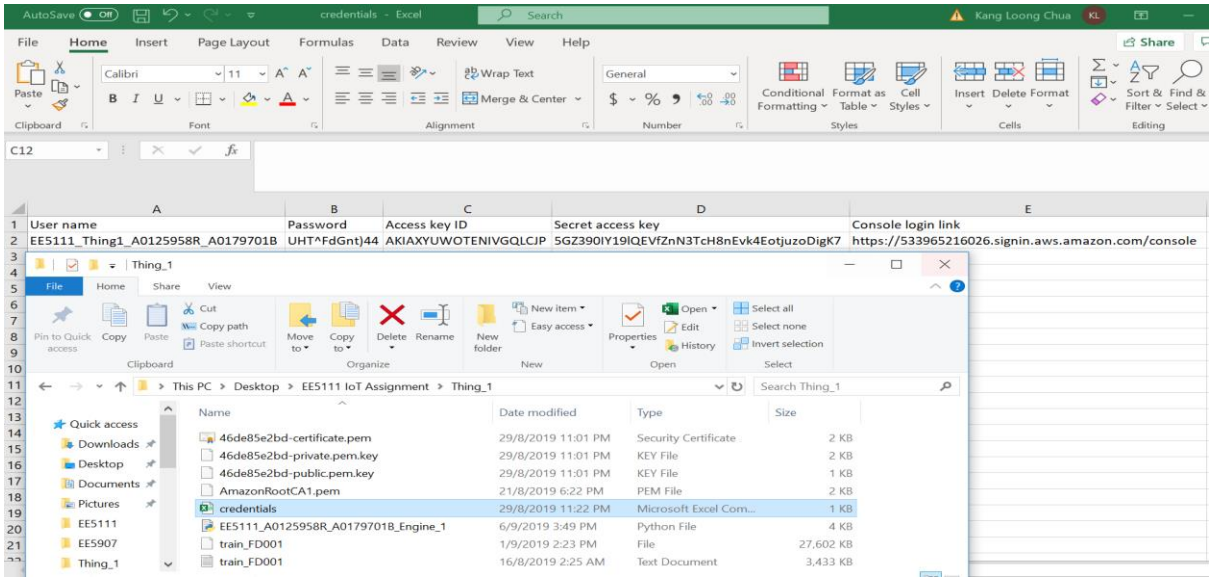

Figure 50 - Redash data visualization part 6

In Redash query interface, input the query ‘Name’, enter the SQL query codes for all the sensors from jet engine, ‘Save’ and select ‘Execute’ as shown in Figure 51. Loading of the data from AWS DynamoDB to Redash maybe take some while depending on the amount of data to be transferred. SQL query codes can be found in the appendix.

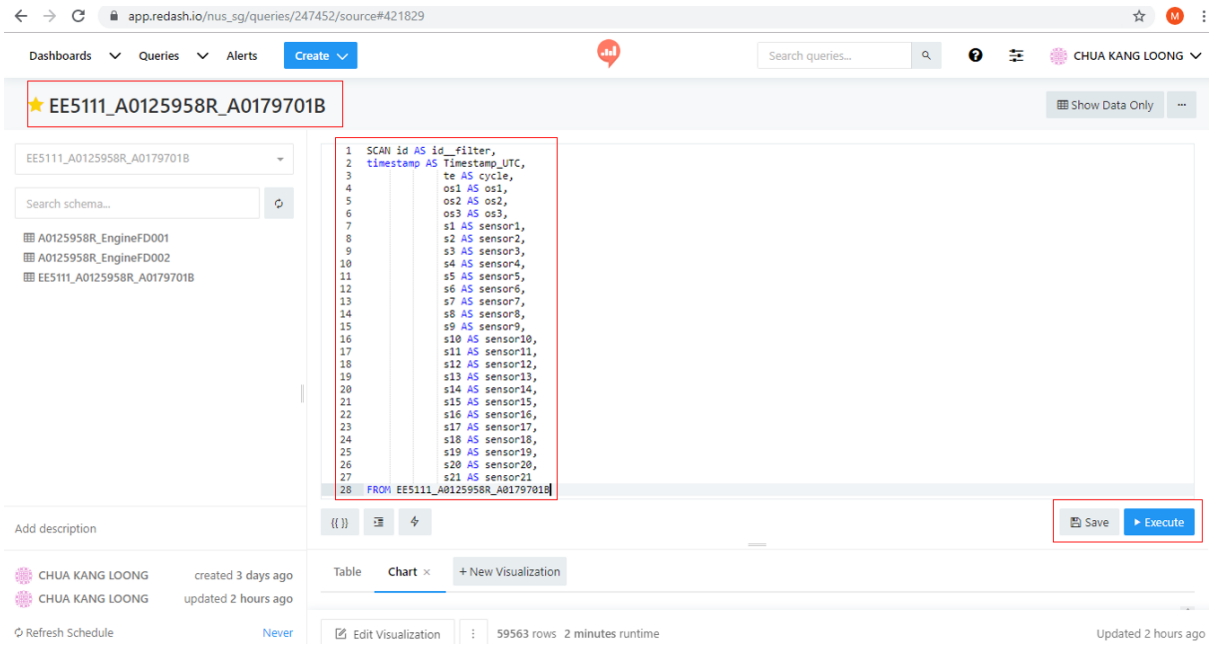

Figure 51 - Redash data visualization part 7

Once all data have been export over, it will be displayed on the interface. Select ‘New Visualization’ to create data visualization for the Things as shown in Figure 52.

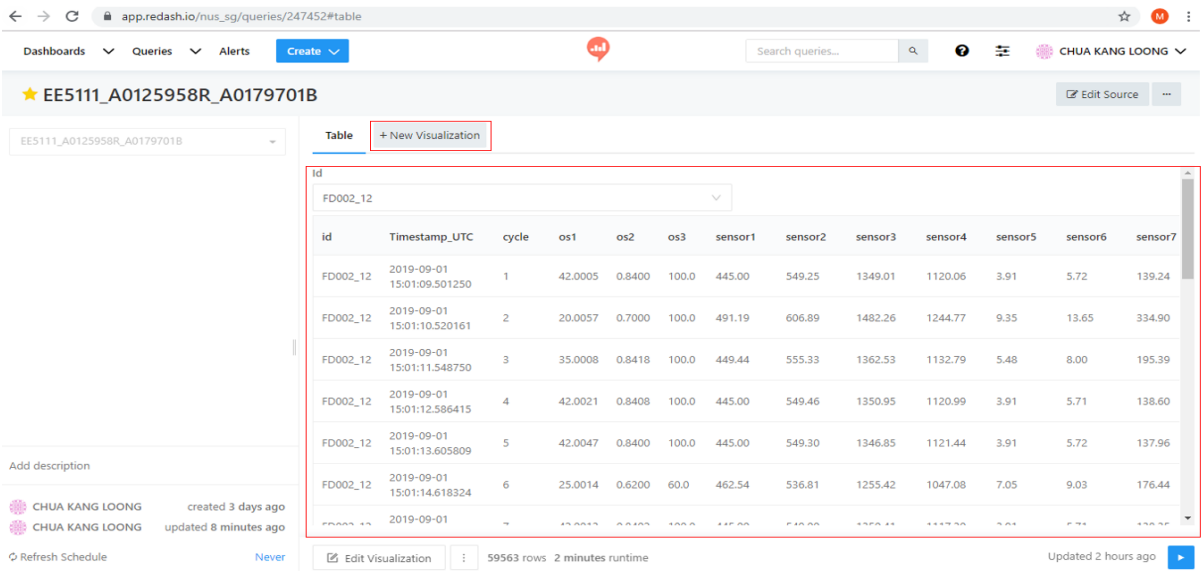

Figure 52 - Redash data visualization part 8

Under visualization editor, enter a ‘Visualization Name’ for the Thing1, choose chart type to be ‘Line’ graph, X column to be ‘Timestamp_UTC’,Y column to be ‘sensorX’, id to be the ‘cycle’ of interest and select ‘Save’ as shown in Figure 53.

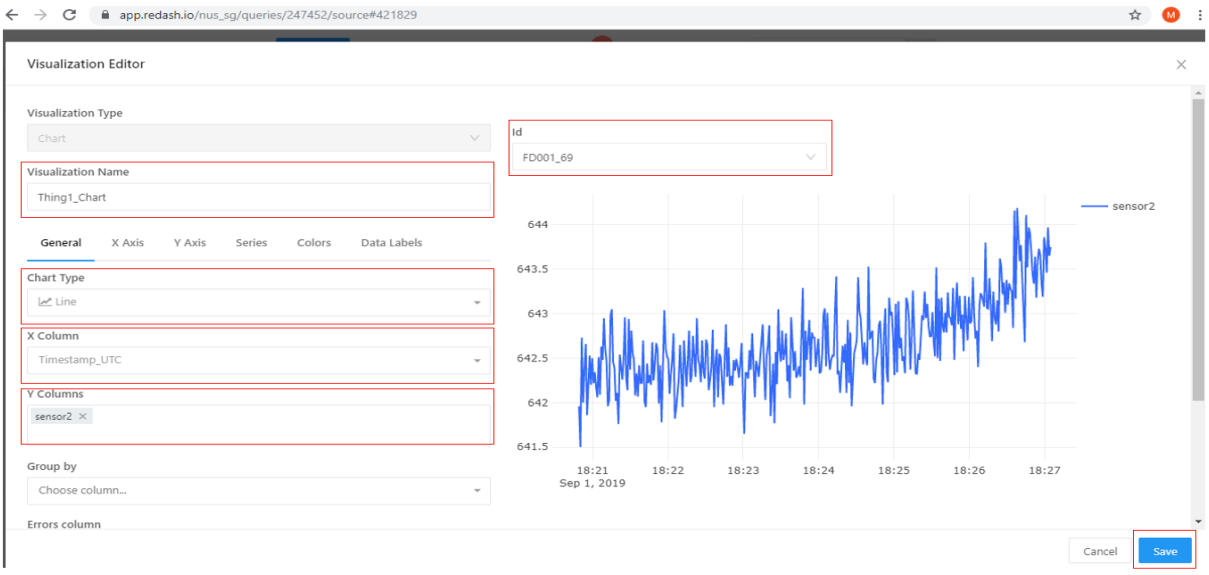

Figure 53 - Redash data visualization part 9

Repeat the same procedure above for Thing2 for both Things to be visualized in dashboard later as shown in Figure 54.

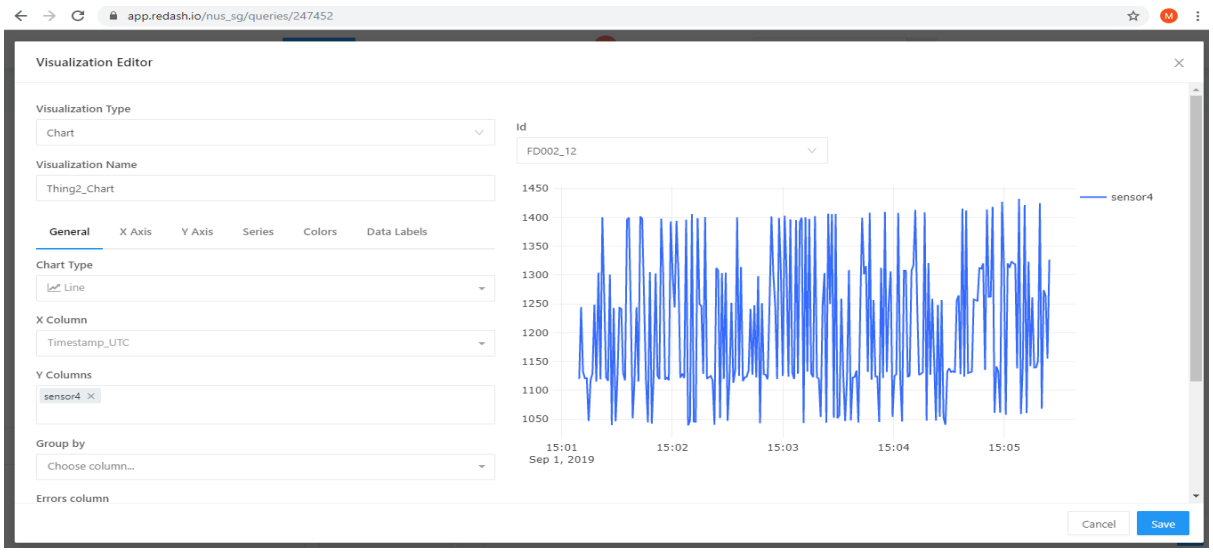

Figure 54 - Redash data visualization part 10

Redash dashboard allows combine visualizations and text boxes that provide context with the data.On Redash interface, select ‘Create’ new dashboard and enter a ‘Name’ for the dashboard as shown in Figure 55.

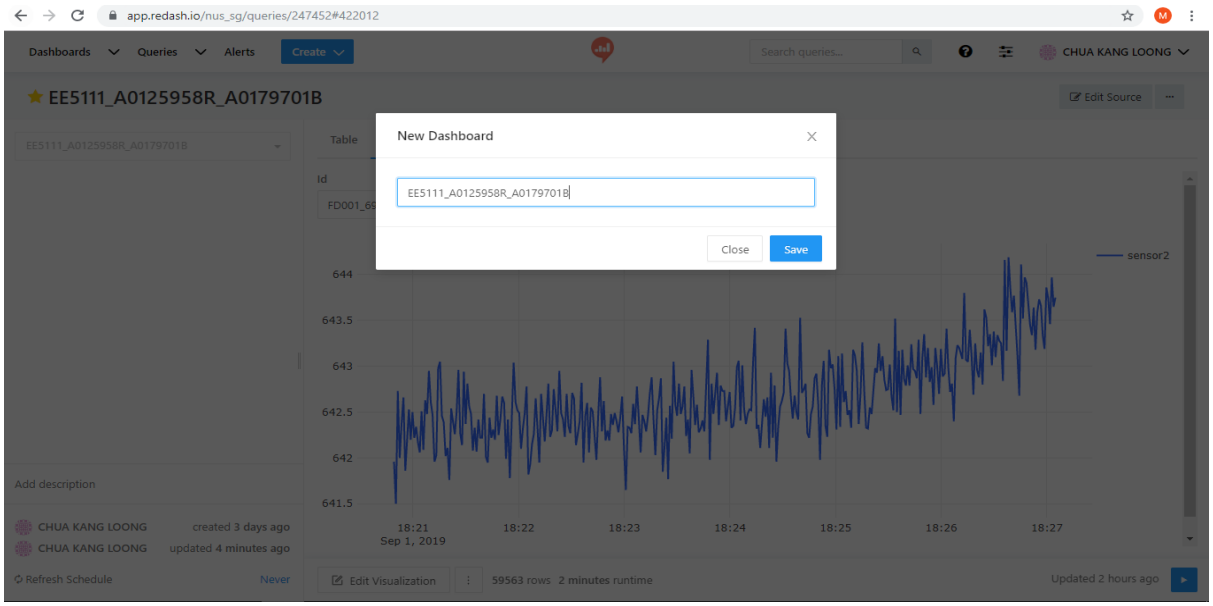

Figure 55 - Redash data visualization part 11

After creating the dashboard, go back to the query created and select ‘Add to Dashboard’ as shown in Figure 56.

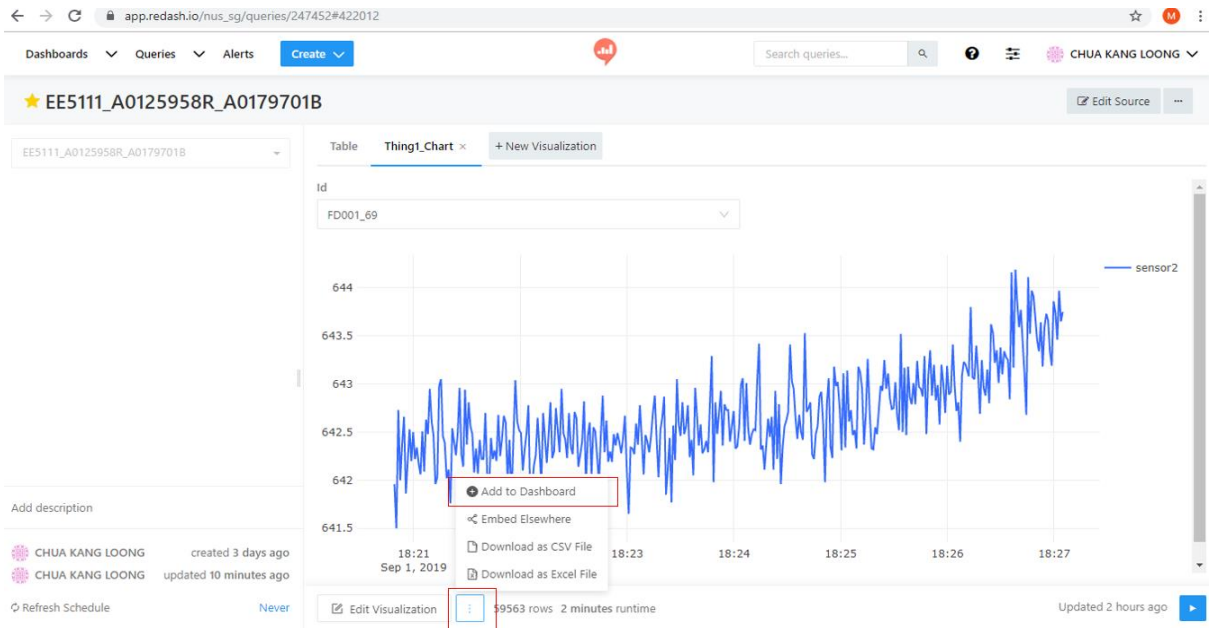

Figure 56 - Redash data visualization part 12

Under ‘Add to Dashboard’, select the dashboard name created earlier. Repeat the same procedure for both Thing1 and Thing2 as shown in Figure 57.

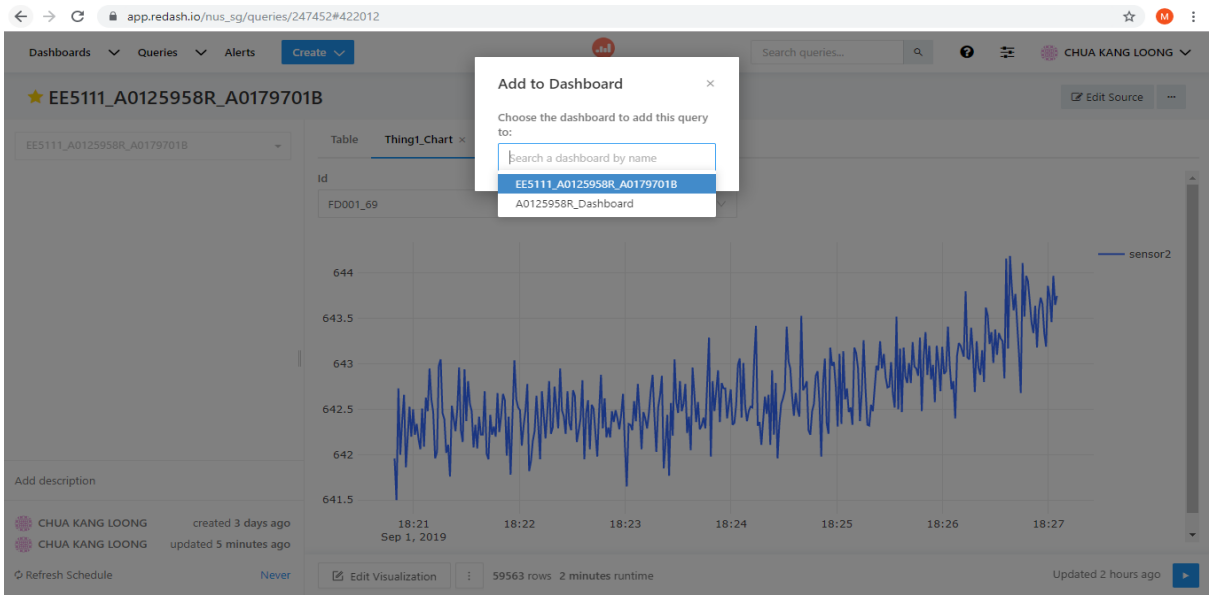

Figure 57 - Redash data visualization part 13

On Redash interface, choose Dashboard to view the queries chosen to be displayed. More sensors data can be added in its query interface or different cycle of the things can be selected by changing the ‘id’. Public ULR of the dashboard can also be set public as clicking the ‘Share’ icon as shown in Figure 58.

https://app.redash.io/nus_sg/public/dashboards/r3e1hl20Zs7PvSFy8D7o57wDvKnkKz6kfJEIY0h8

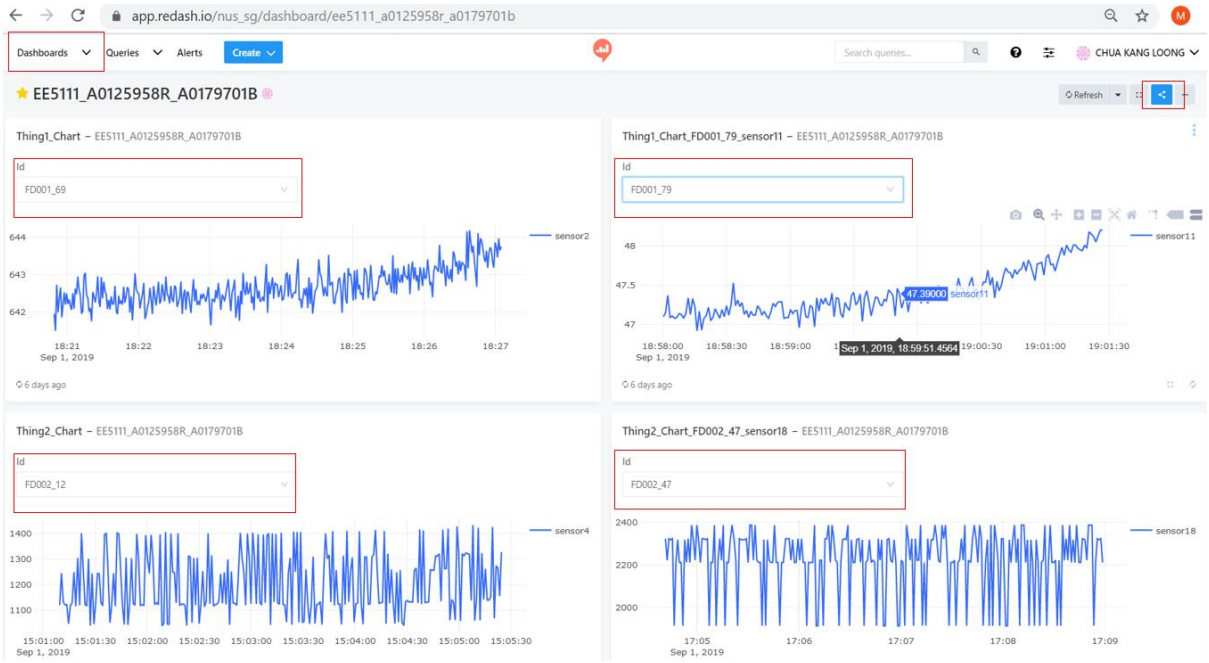

Figure 58 - Redash data visualization part 14

Real-Time Embedded System – Raspberry Pi with AWS IOT (Additional)

For the ingestion of real time data from embedded system, we are using Raspberry Pi 3 Model B with build in WIFI as IoT hardware which has low power embedded processor, wireless and internet network gateway capability. The data to be published to AWS IoT is the real-time board temperature of the Raspberry Pi. By publishing Raspberry Pi board temperature, we could monitor the health of the board via AWS. The following sections are demonstrated the AWS IoT thing creation, setup, AWS dynamoDB table creation, hardware setup, software

package installation, transfer data from Raspberry Pi 3 Model B to AWS and Data Visualization.

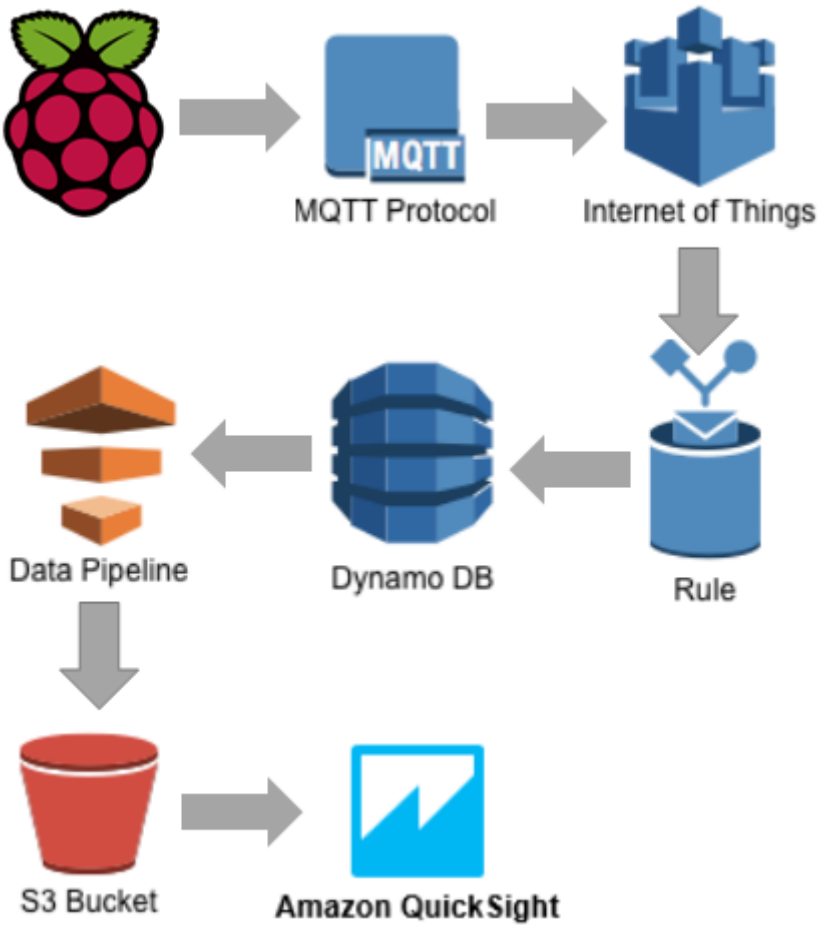

Figure 59: Real-time Embedded System Overview

AWS IoT Thing Creation and Setup

First of all, we are creating the AWS IoT Policy. The policy is to govern and authorise the IoT devices to be perform certain operations such as subscribing or publishing to MQTT topics. From the Policies tab under the Secure section, click on the “Create” button, then fill up the “Create a policy” window as per below screenshot.

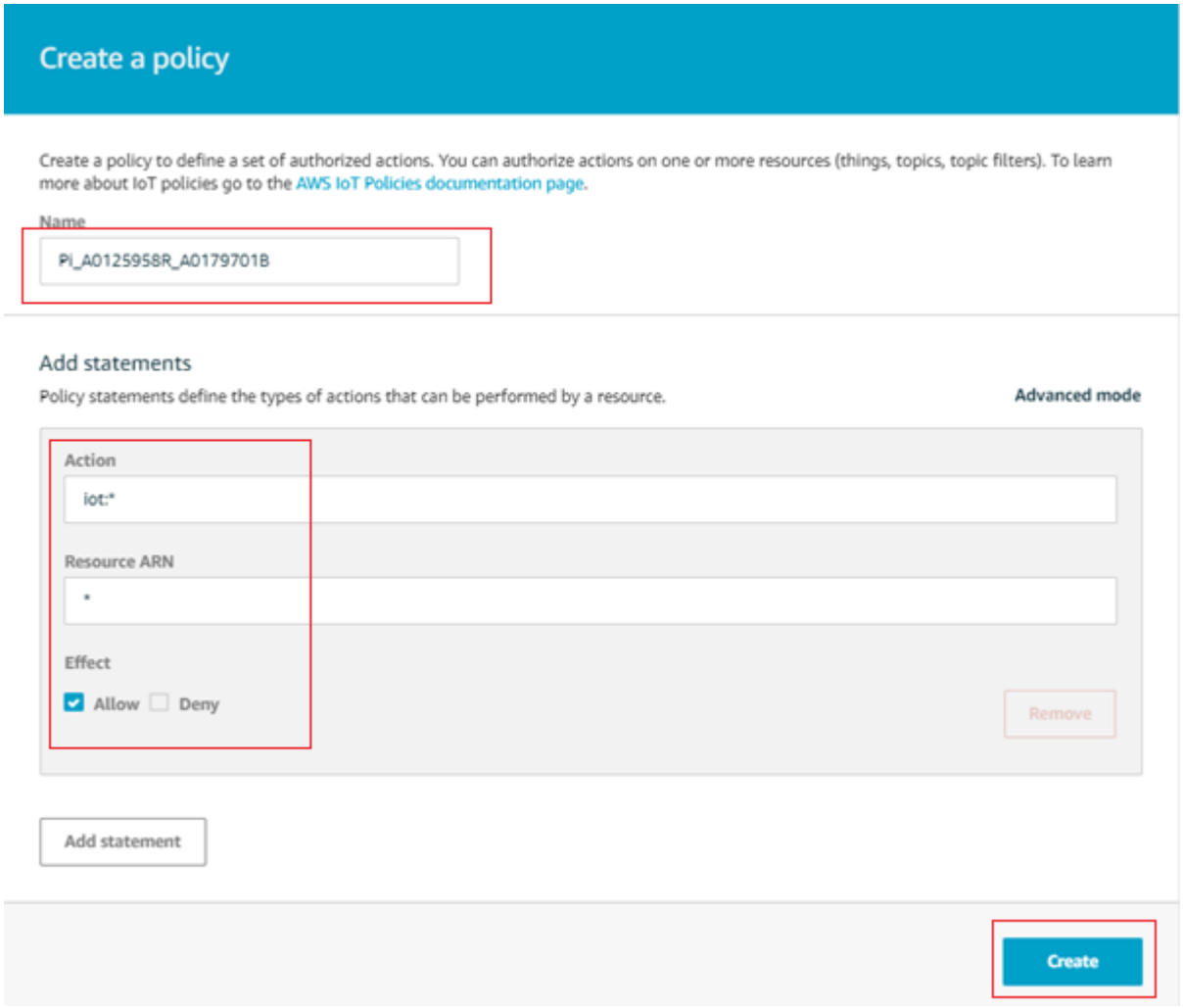

Figure 60: Fields to set up the AWS IoT Policy (highlighted in red boxes; Action: iot.* ; Resource ARN: *)

After the policy is created and set up, proceed to create an IoT thing to represent the physical Raspberry Pi by clicking on the “Create” button in the Things tab under Manage section. Fill up the “CREATE A THING” Window as shown in screenshot below,

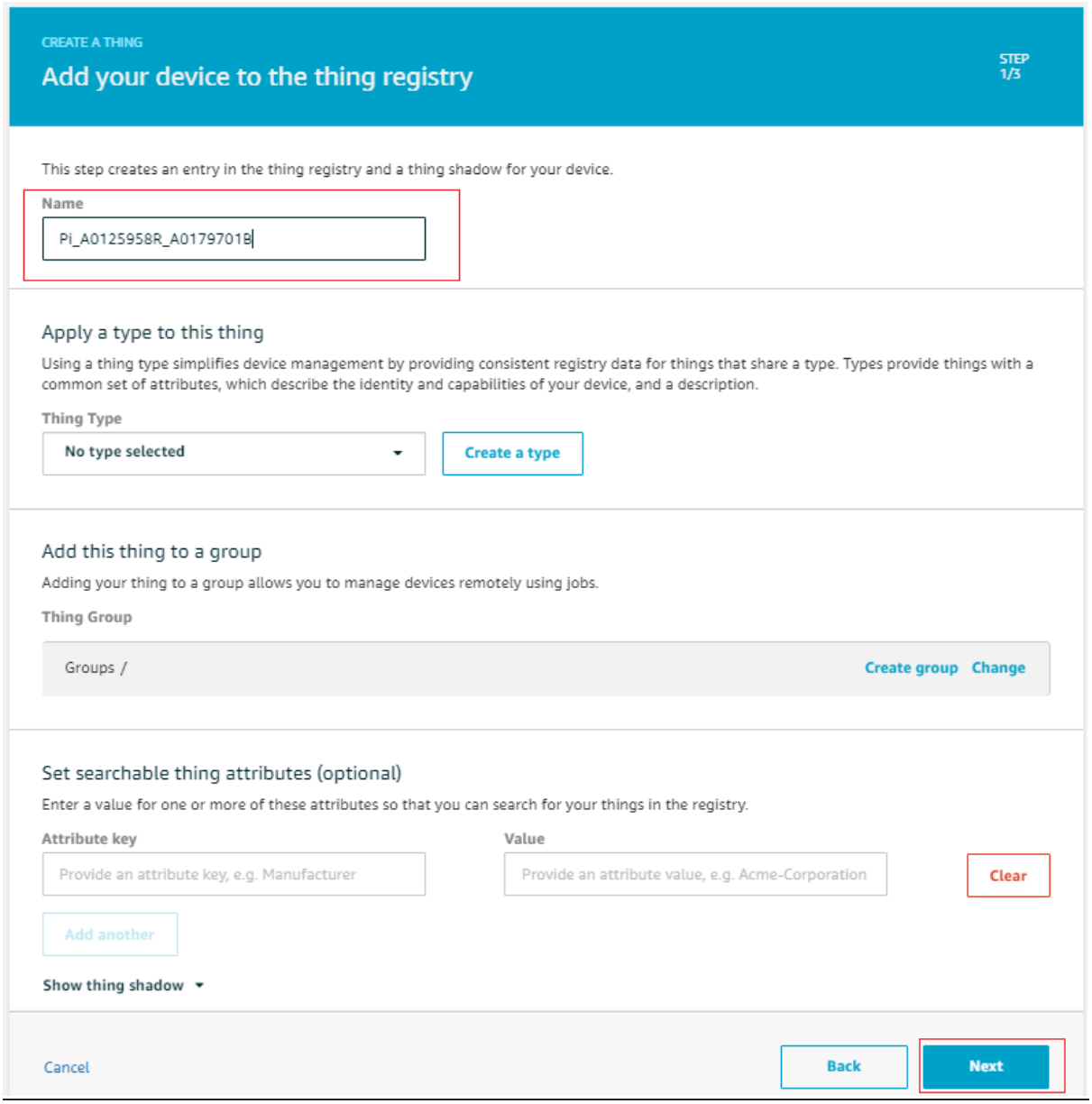

Figure 61: create a Thing

After naming the Thing, next is to add a certificate to the Thing. This authentication Certificate is required for handshaking the Raspberry Pi and the AWS IoT Thing so that it could allowed to send or receive data via the MQTT protocol. To do this, the AWS IoT Core platform will issue a unique authentication certificate (Notice the string of pre-generated numbers, e.g. in "2b35f1588c.cert.pem", see Figure 63 below) for the Thing and produce private and public keys. It is recommended to download all of them, including the root CA and put into Raspberry Pi.

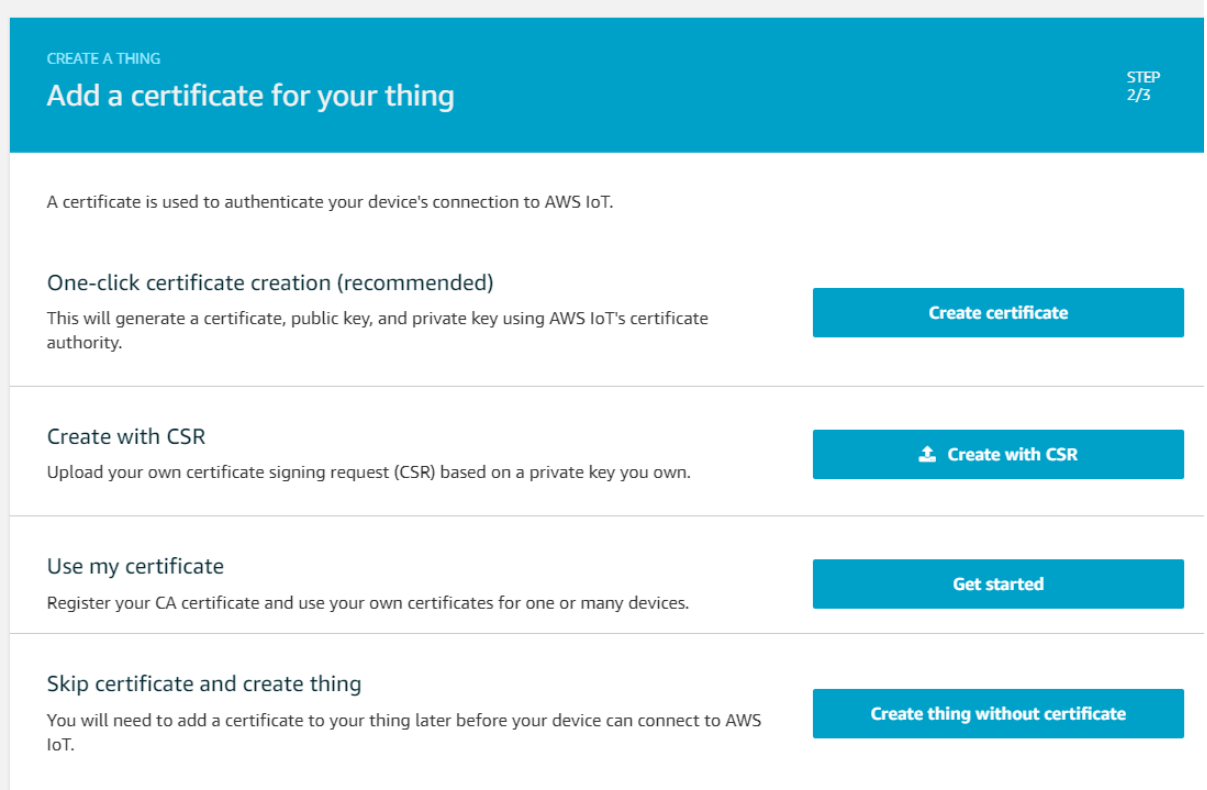

Figure 62: Create Certificate

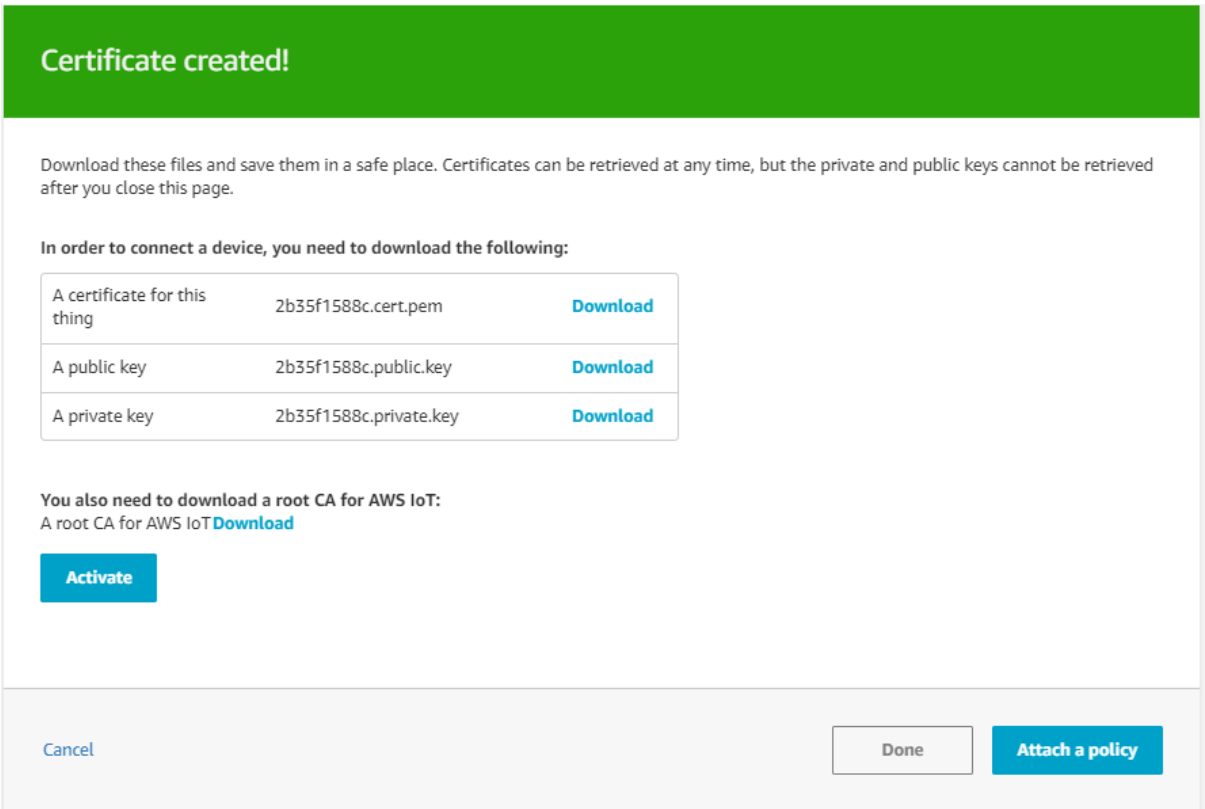

Figure 63: Download and Activate Authentication Certificate

Next is to attach policy of the thing as shown in the screenshot below,

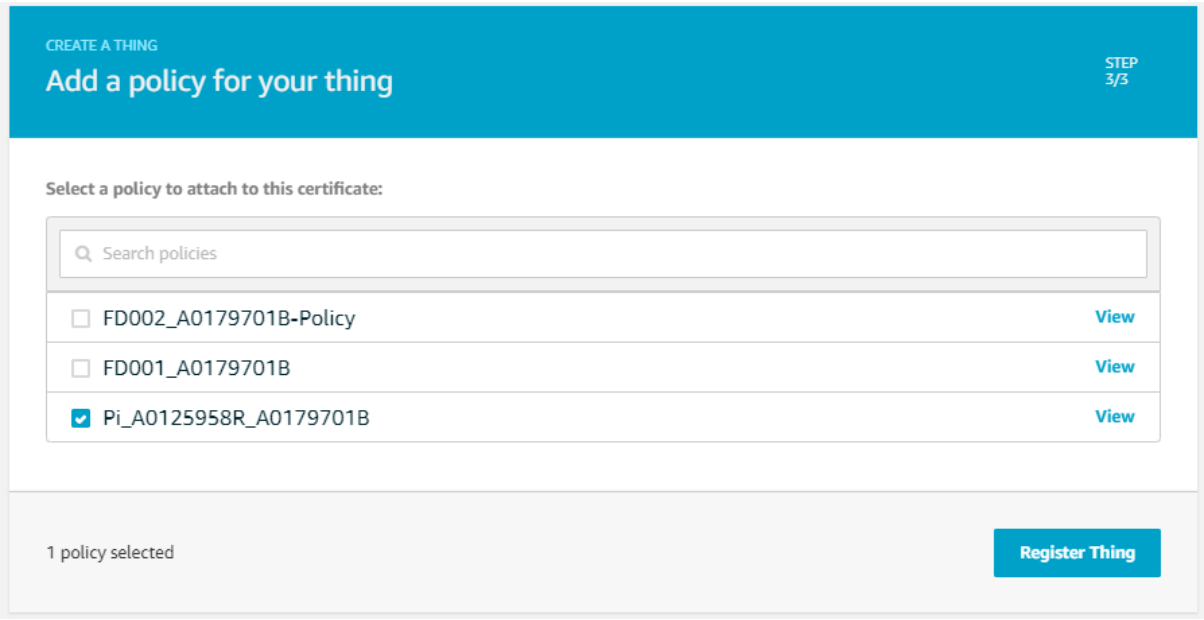

Figure 64: Attaching Policy to the Thing

We have to create a Rule In order to write the receive data from the Raspberry Pi to DynamoDB table. Click on the “Create” button under Act section then fill up the field as shown in the screenshot below,

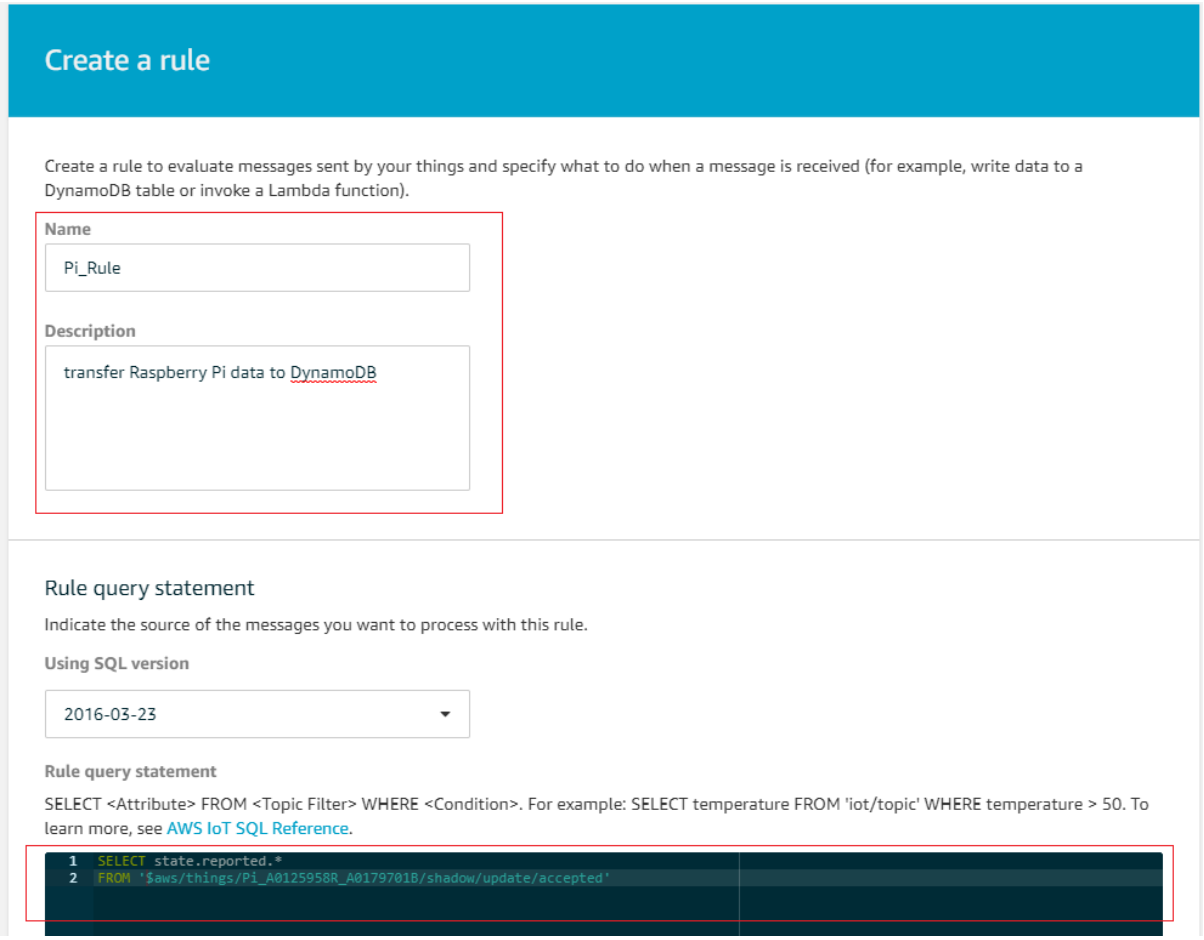

Figure 65: Create a Rule transfer Raspberry Pi Data to DynamoDB

Under the Rule query statement, type:

```
SELECT state.reported.* FROM
'$aws/things/Pi_A0125958R_A0179701B/shadow/update/accepted'
```

Then click on “Add action” and then select “Split data into multiple columns of a DynamoDB table (DynamoDBv2)”. Under Configure Action, click on the “Create a new resource” to create a table.

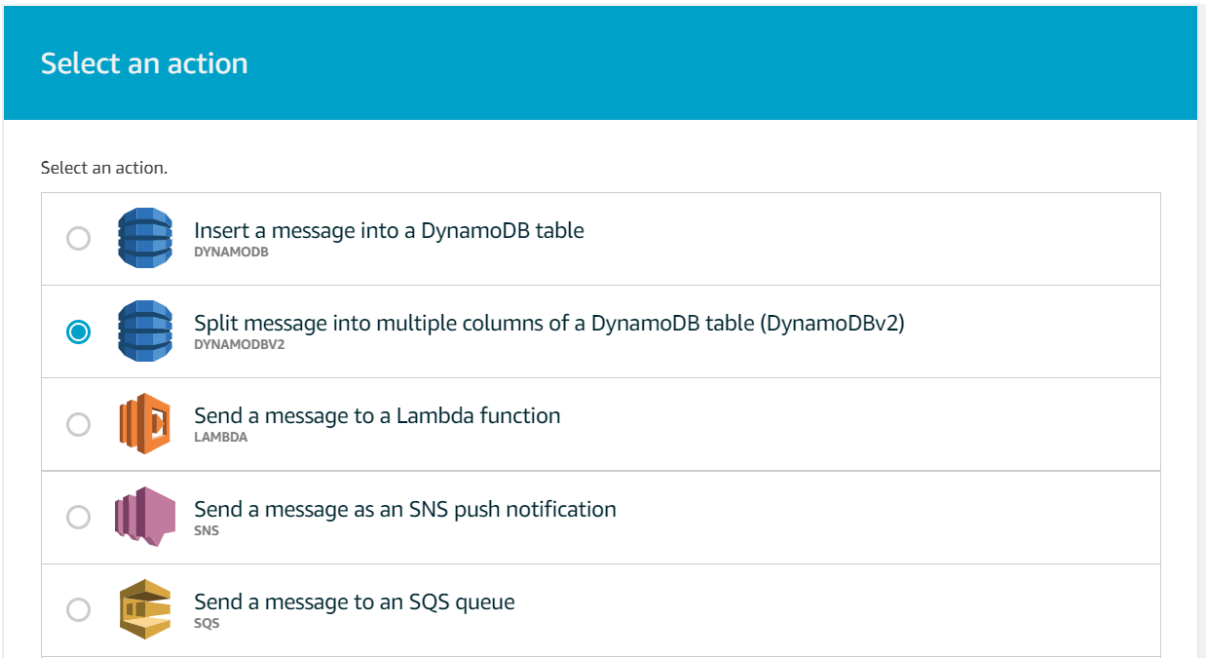

Figure 66: Add action

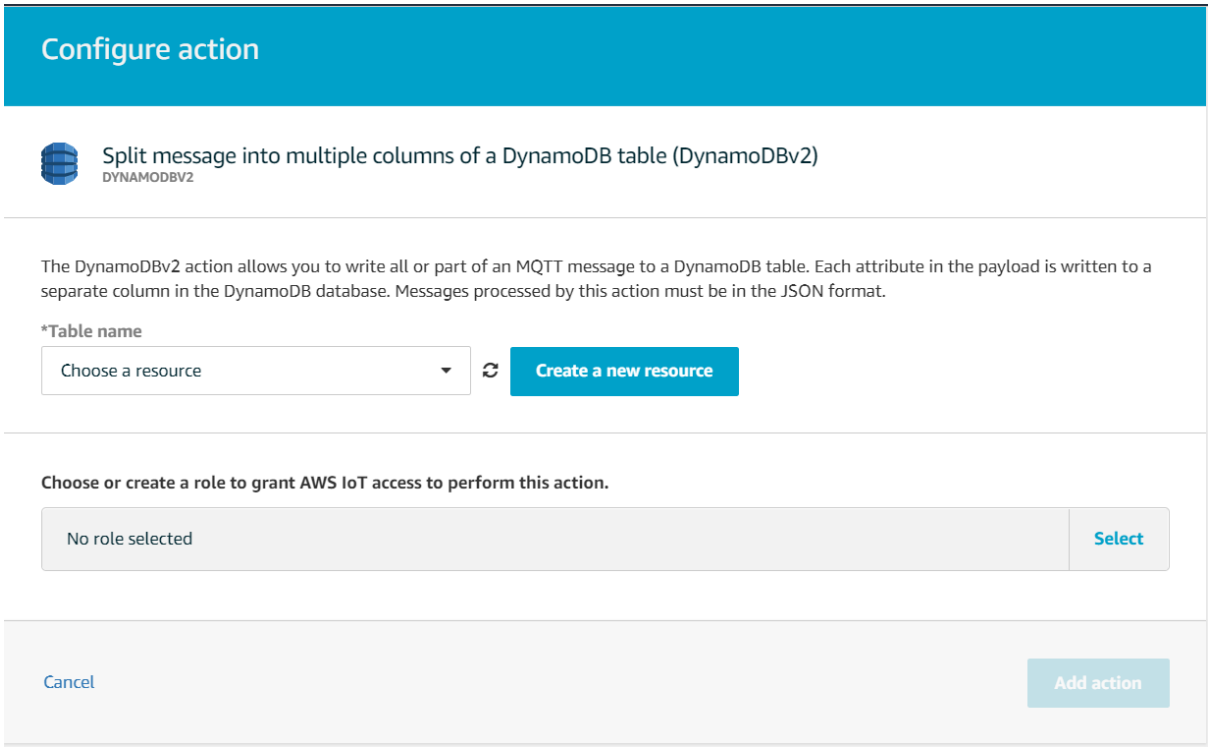

Figure 67: Configure action

Select “Tables” tab under AWS DynamoDB and click on “Create table”. Name the Table name with “Pi_A0125958R_A0179701B” and designate the partition and sort keys for which to organise the data from the JSON string being sent from Raspberry Pi to the DynamoDB table.

In this case, we select the ‘id’ and ‘timestamp’ fields to be the partition and sort keys respectively (see below).

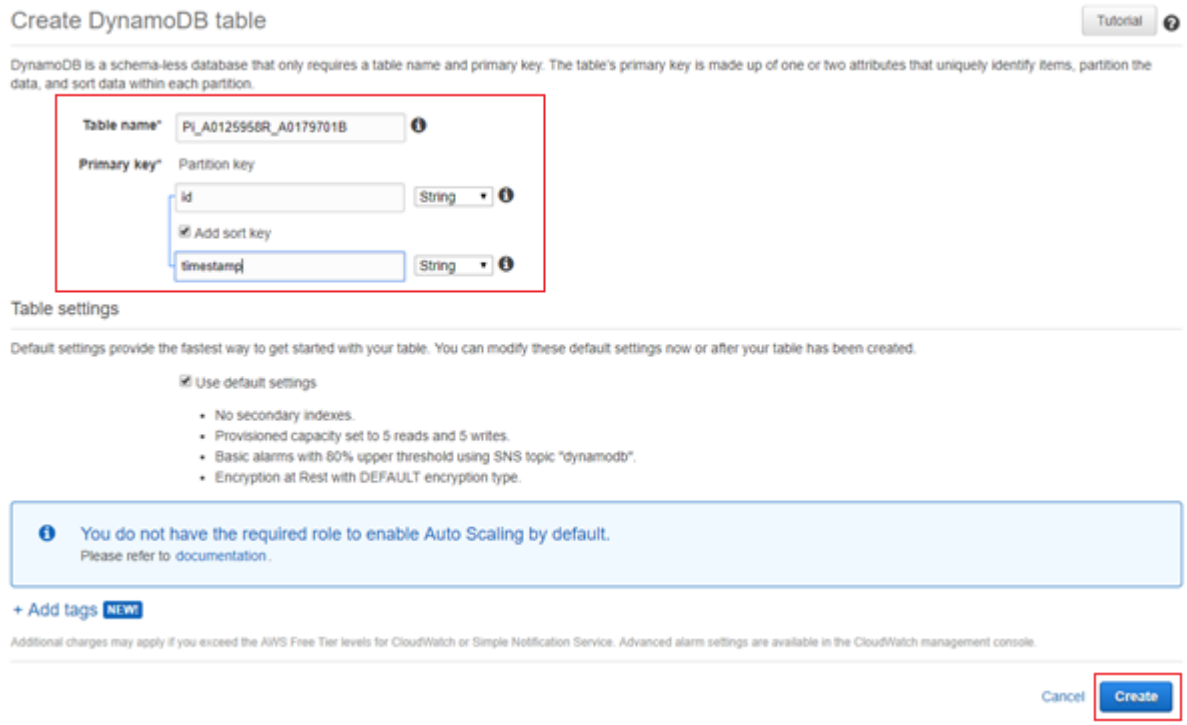

Figure 68: Create DynamoDB Table

Once the table is created, select the table “Pi_A0125958R_A0179701B” and click on “Manage Stream” and then click “Enable”. This is to allow table to receive and update new data.

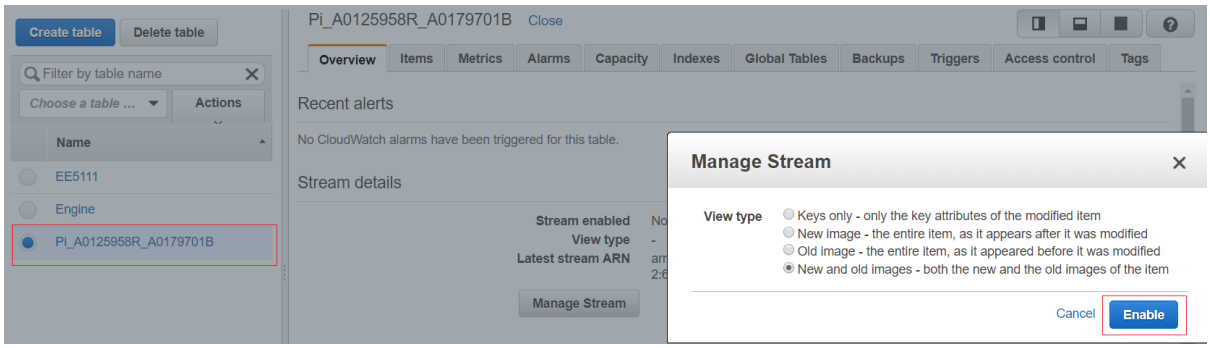

Figure 69: Enable Data Stream in DynamoDB Table

Now we have the table ready, continue to configure action for the rule by selecting the “Pi_A0125958R_A0179701B” as the Table name and select “Publisher” role. If the role is not created prior, then create one by clicking “Create Role”.

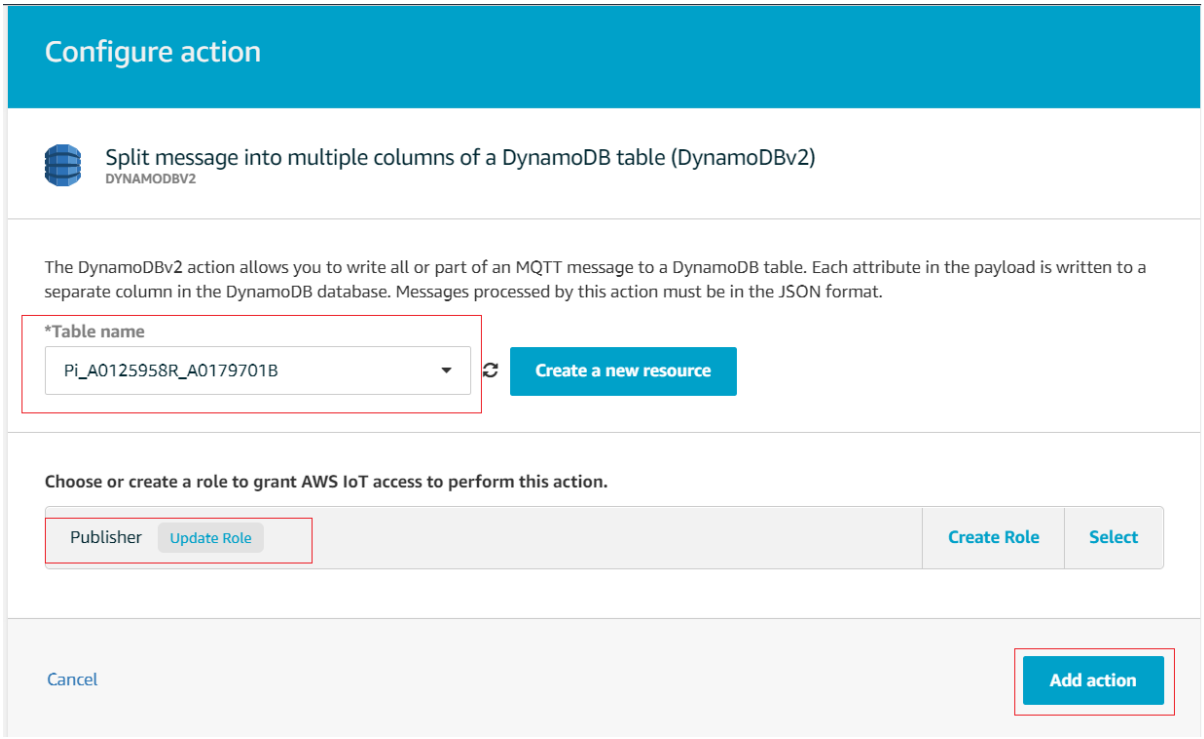

Figure 70: Configure Action (select table and role)

Raspberry Pi OS installation

The operating system is New Out Of Box Software (NOOBS) which can downloaded from https://www.raspberrypi.org/downloads/. Then format a SD card as FAT and copy and extract NOOBS file to the SD card.

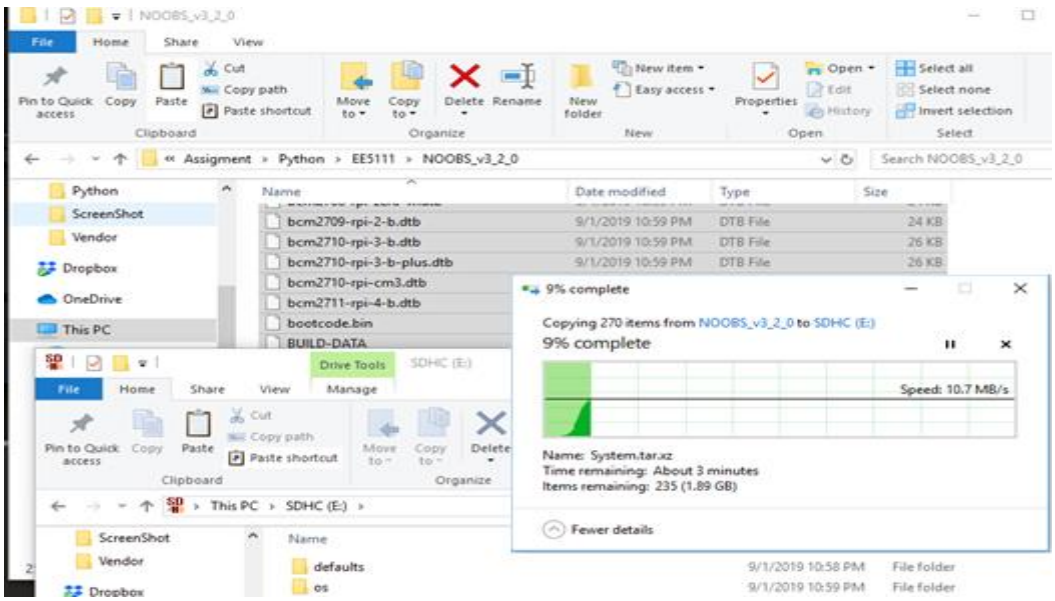

Figure 71: Setup SD card for Raspberry Pi

Raspberry Pi Power Up

SD card is then slot into to Raspberry Pi and then connect a separate monitor, keyboard and mouse directly to Raspberry Pi via HDMI and USB ports. After that, connect the micro USB power source into Raspberry Pi and turn on the power source. Raspberry Pi will have red LED lit up and blinking green LED.

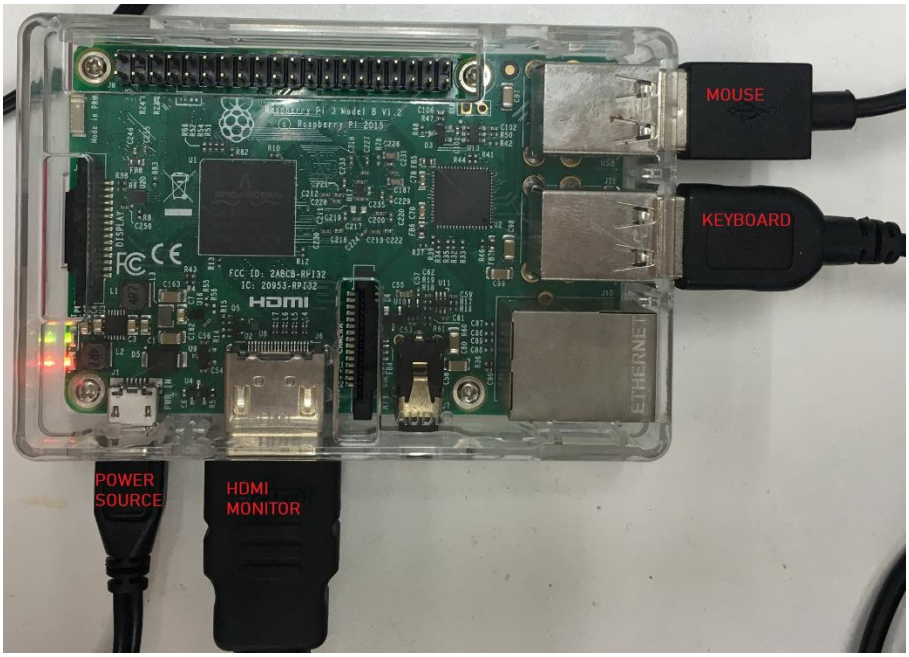

Figure 72: Raspberry Pi Hardware Ports

Raspberry Pi WIFI Setup

Way to setup and establish wireless network connection is open command prompt of Raspberry Pi. First to add the network details to Raspberry Pi with the command of "*sudo nano /etc/wpa_supplicant/wpa_supplicant.conf"* , then add the network info to wpa_supplicant.conf file as shown in Figure below

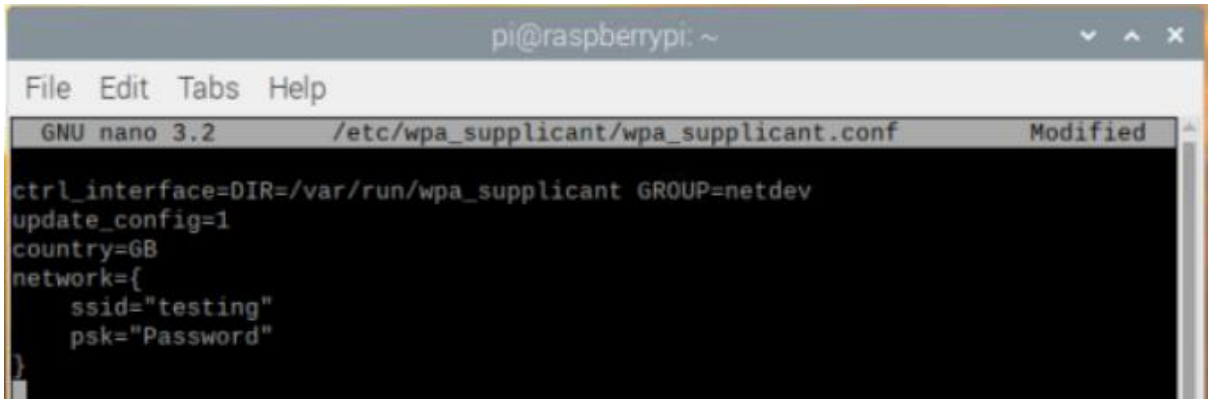

Figure 73: Set up Raspberry Pi Network

Once the WIFI is setup and Raspberry Pi is connected to WIFI, then we could remote access to Raspberry PI via VNC with IP address.

Raspberry Pi Software Packages Setup

Check python version in Raspberry Pi via command prompt with the "*python3 –version*"

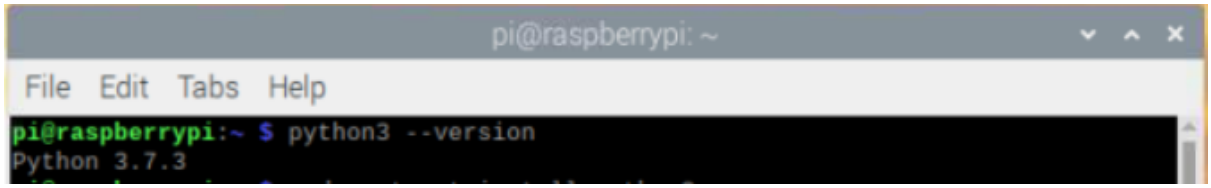

Figure 74: Check Python Version

If the python is not install, then to install python in Raspberry Pi via command prompt with the following command *"sudo apt-get install python3"* as shown in Figure below,

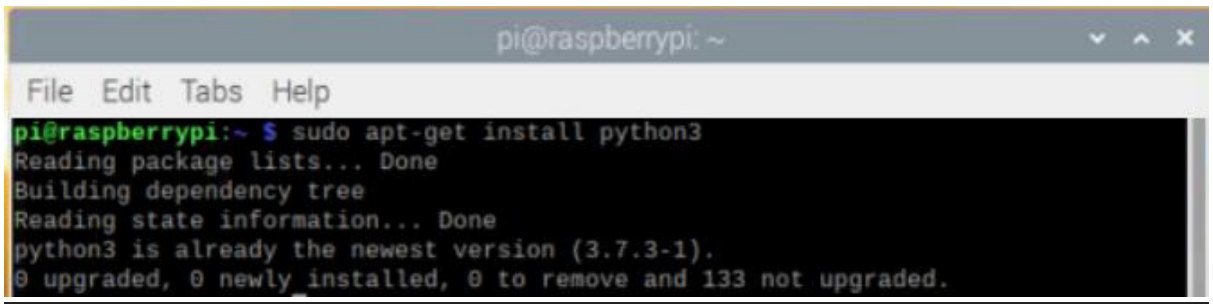

Figure 75: Python Installation

Connection between AWS IoT and Raspberry Pi is establish via MQTT protocol which MQTT library is built in AWS IoT Python SDK. Then, download AWS IoT Python SDK from https://s3.amazonaws.com/aws-iot-device-sdk-python/aws-iot-device-sdk-python-latest.zip

by using the web browser in Raspberry Pi. Unzip the file and install in Raspberry Pi with the following command *"sudo python setup.py install"* in command prompt.

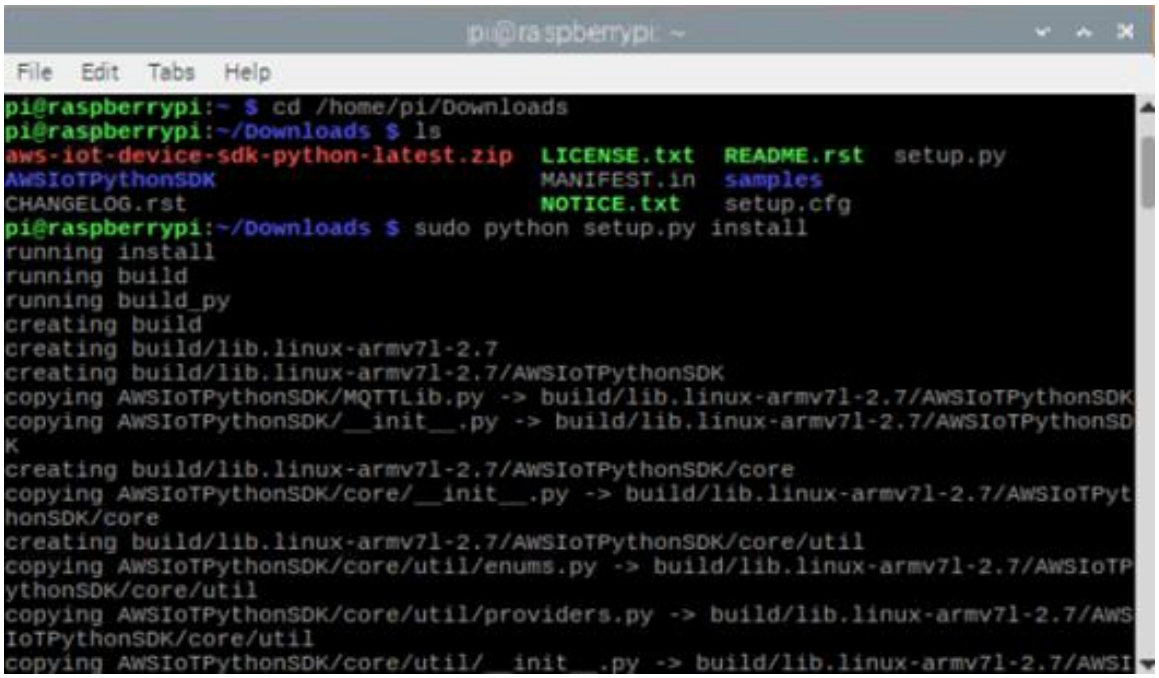

Figure 76: AWS IoT Python SDK Installation

Raspberry Pi AWS Certificate Download

In order to allow Raspberry Pi authenticate and get access to AWS IoT things , the authentication certificates of AWS IoT things are download and put into Raspberry Pi. Besides that, security keys are generate and download from AWS website https://docs.aws.amazon.com/iot/latest/developerguide/managing-device-certs.html.

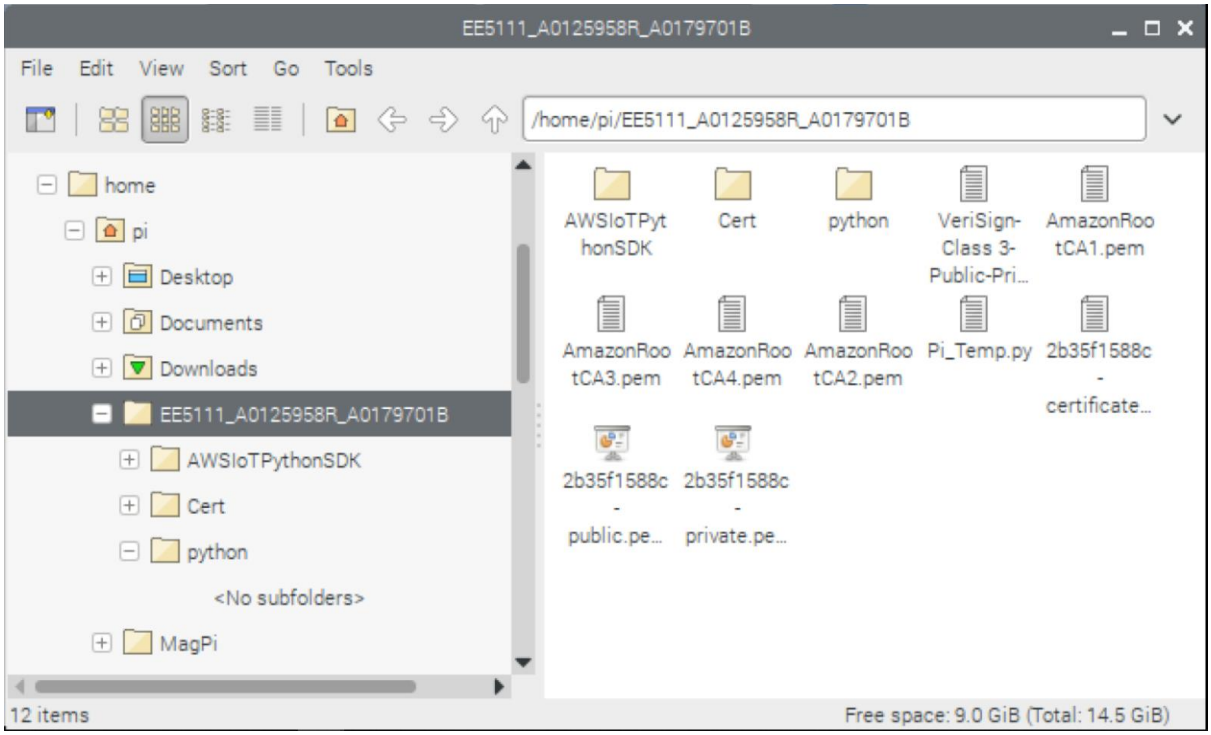

Figure 77: Raspberry Pi Project Folder

Raspberry Pi Python Code

The run the python code below on Raspberry Pi where get its board temperature and publish the temperature value to AWS IoT Thing and then the IoT Rule helps transfer the data to dynamoDB. In this demonstration, we limit the publish number to 2000 times.

```
from  AWSIoTPythonSDK.MQTTLib import AWSIoTMQTTShadowClient
import random, time, datetime, os


# A random programmatic shadow client ID.
SHADOW_CLIENT = "Pi_A0125958R_A0179701B"


# The unique hostname that &IoT; generated for
# this device.
HOST_NAME = "a1xli7oxtwsplh-ats.iot.us-west-2.amazonaws.com"


# The relative path to the correct root CA file for &IoT;,
# which you have already saved onto this device.
ROOT_CA = "AmazonRootCA1.pem"
```

```
# The relative path to your private key file that
# &IoT; generated for this device, which you
# have already saved onto this device.
PRIVATE_KEY = "2b35f1588c-private.pem.key"

# The relative path to your certificate file that
# &IoT; generated for this device, which you
# have already saved onto this device.
CERT_FILE = "2b35f1588c-certificate.pem.crt"

# A programmatic shadow handler name prefix.
SHADOW_HANDLER = "Pi_A0125958R_A0179701B"

# Automatically called whenever the shadow is updated.
def myShadowUpdateCallback(payload, responseStatus, token):
  print()
  print('UPDATE: $aws/things/' + SHADOW_HANDLER +
    '/shadow/update/#')
  print("payload = " + payload)
  print("responseStatus = " + responseStatus)
  print("token = " + token)

# Create, configure, and connect a shadow client.
myShadowClient = AWSIoTMQTTShadowClient(SHADOW_CLIENT)
myShadowClient.configureEndpoint(HOST_NAME, 8883)
myShadowClient.configureCredentials(ROOT_CA, PRIVATE_KEY,
  CERT_FILE)
myShadowClient.configureConnectDisconnectTimeout(10)
myShadowClient.configureMQTTOperationTimeout(5)
myShadowClient.connect()
myShadowClient.disconnect()
myShadowClient.connect()
# Create a programmatic representation of the shadow.
```

```
myDeviceShadow = myShadowClient.createShadowHandlerWithName(
  SHADOW_HANDLER, True)

def get_temp():
    temp = os.popen("vcgencmd measure_temp").readline()
    return (temp.replace("temp=",""))

# ***************************************************
# Main script runs from here onwards.
# To stop running this script, press Ctrl+C.
# ***************************************************

# Get the labels out
sensor_name = ['Temp'+ str(i) for i in range(1,2)]
dataLabels = ['id', 'timestamp', 'matricNumber'] + sensor_name

matricNumber = 'A0125958R_A0179701B'

for i in range(0,len(dataLabels)):
    dataLabels[i] = '\"' + dataLabels[i] + '\"'


dataString = []
modifiedData = []

head = '{"state":{"reported":{'
tail = '}}}'
count = 0

while True:

    temp = get_temp()
    temp = temp[0:4]
    print(temp)
```

```
count = count+1
print(count);


modifiedData = []
modifiedData.append(str('Raspberry' + 'Pi'))
modifiedData.append(str(datetime.datetime.utcnow()))
modifiedData.append(matricNumber)
for j in range(2,len(sensor_name)):
    modifiedData.append(temp)


ColumnLabels = []
ColumnLabels.append(str(dataLabels[0] + ':'))
ColumnLabels.append(str('"' + modifiedData[0] + '",'))
ColumnLabels.append(str(dataLabels[1] + ':'))
ColumnLabels.append(str('"' + str(datetime.datetime.now()) + '",'))
ColumnLabels.append(str(dataLabels[2] + ':'))
ColumnLabels.append(str('"' + matricNumber + '",'))



for i in range(3,len(dataLabels)):
    ColumnLabels.append(str(dataLabels[i] + ':'))
    ColumnLabels.append(str('"' + temp + '",'))


string = ''.join(ColumnLabels)
string = string[:-1]


data = []
data.append(head)
data.append(string)
data.append(tail)
data.append('\n')
dataString = ''.join(data)
print(dataString)
myDeviceShadow.shadowUpdate(dataString,myShadowUpdateCallback, 5)
```

```
    time.sleep(5)
    if count>2000:break
```

The figure below showed that the temperature of RaspberryPi was received and stored in table name Pi_A0125958R_A0179701B on AWS DynamoDB

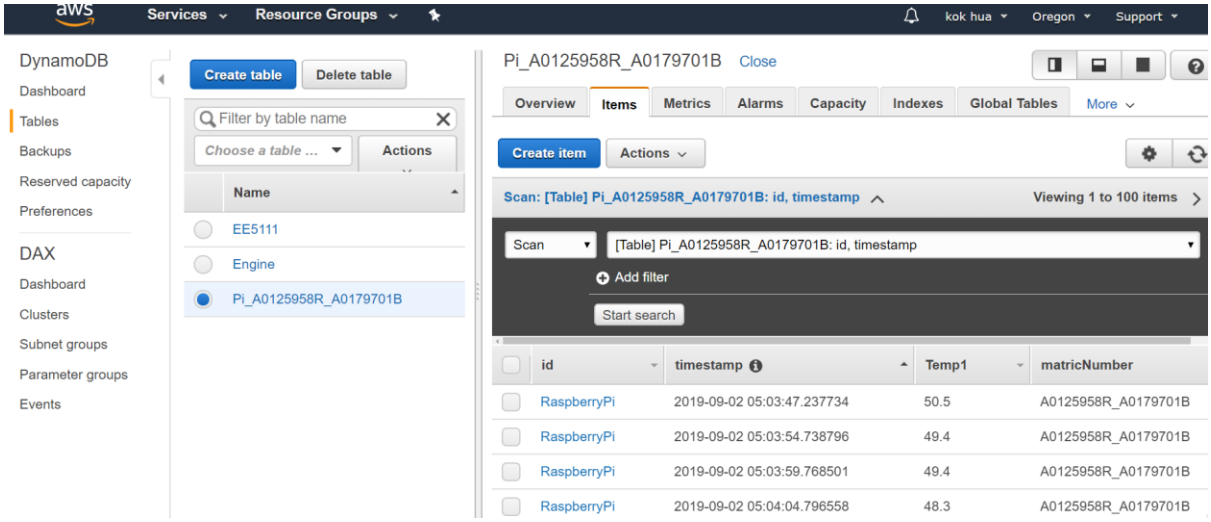

Figure 78: Temperature Data Stored in DynamoDB Table (Pi_A0125958R_A0179701B)

Data Visualization on AWS QuickSight

Amazon QuickSight is a business analytics service that has capability to build visualizations, perform analysis, and extract business insights data. It can automatically discover AWS data sources. To visualize data (temperature) from the Raspberry Pi that store in DynamoDB table, the data has to be transfer and back up to Amazon Simple Storage Service (S3). AWS S3 allows user store and retrieve of data at any time and from anywhere on the web. AWS S3 provides high scalable, inexpensive data storage infrastructure that Amazon uses to run its own global network of web sites.

AWS S3 Bucket and Folder Creation

Go to AWS S3, click on “Create Bucket” then follow the screenshots below to create a bucket.

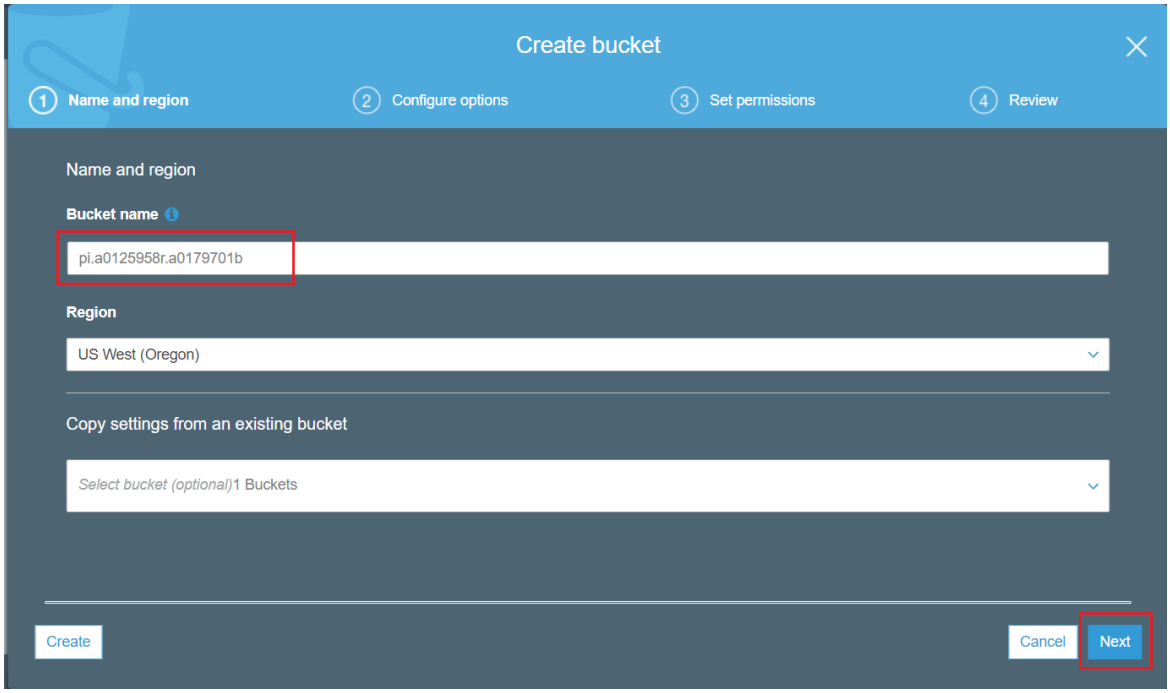

Figure 79: Create Bucket Step 1 (Name and Region)

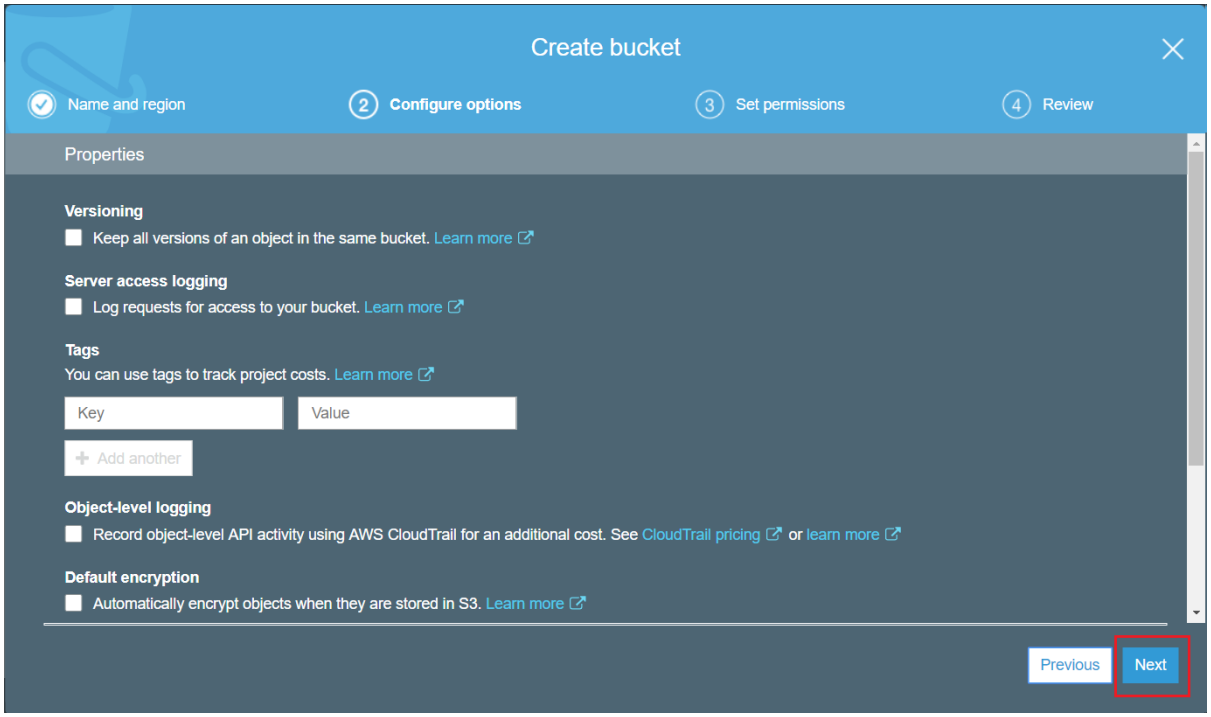

Figure 80: Create Bucket Step 2 (Configure options)

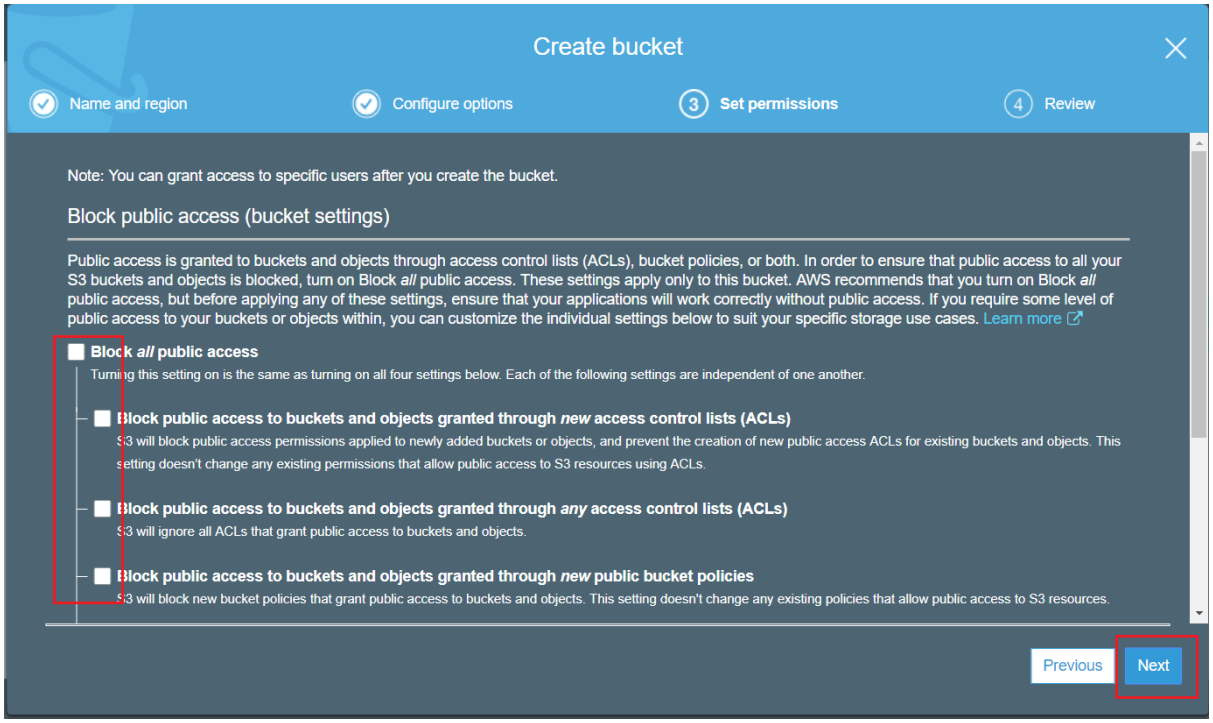

Figure 81: Create Bucket Step 3 (Set Permission to Public)

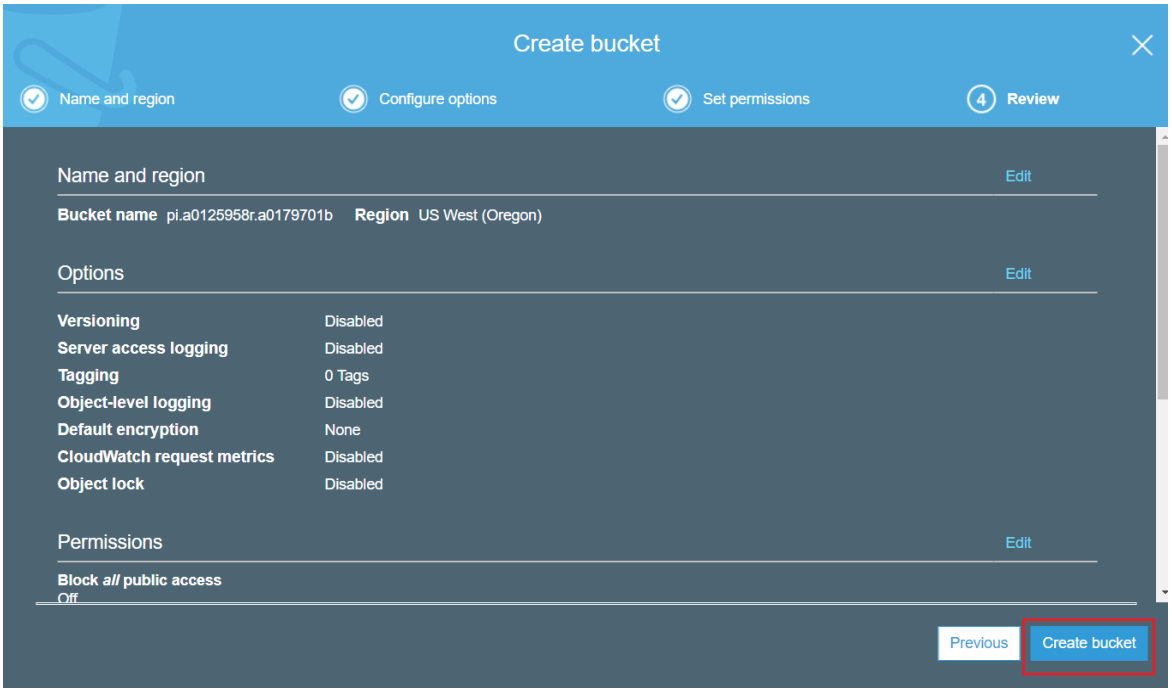

Figure 82: Create Bucket Step 4 (Preview and Create)

After create bucket, then create a folder name “Raspberry Pi” and make the folder to public.

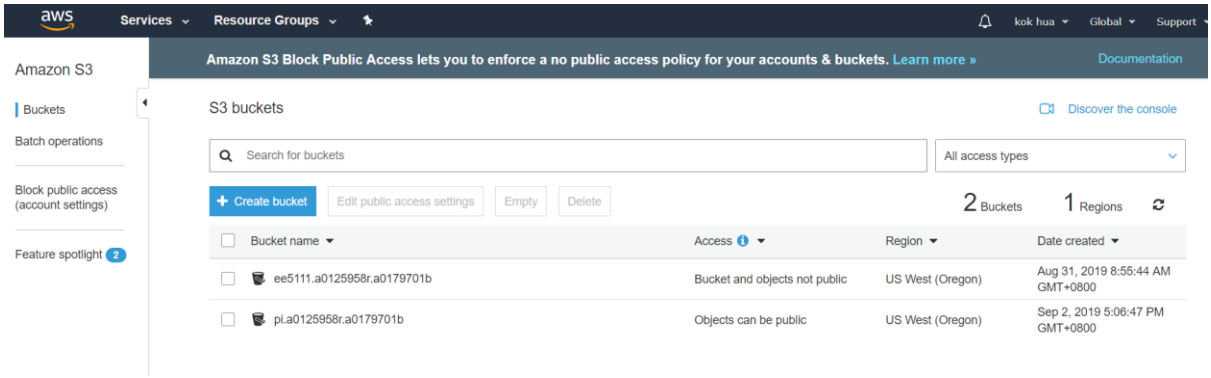

Figure 83: Bucket Creation Successfully

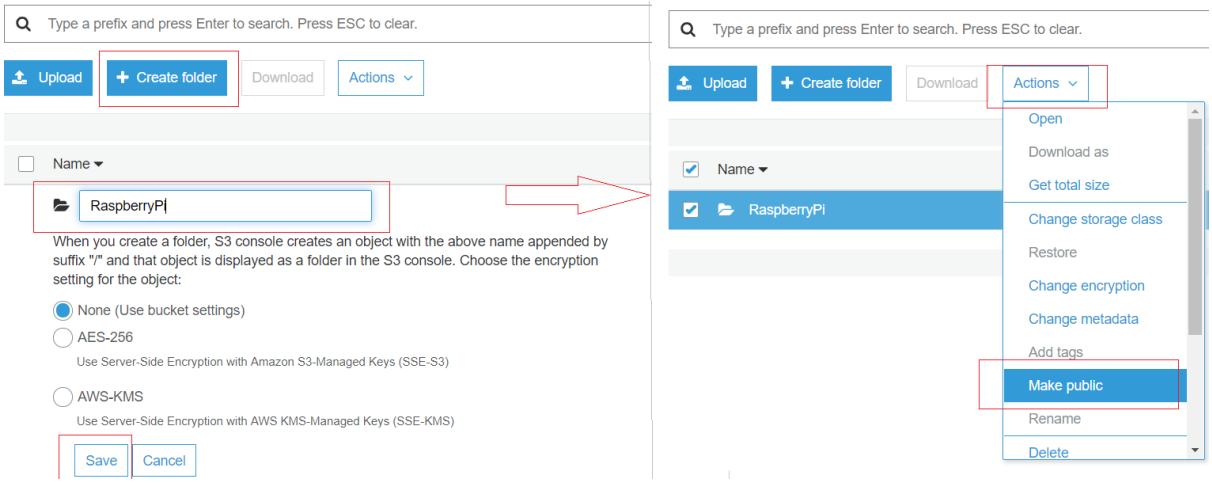

Figure 84: Create a Public Access Folder name RaspberryPi

AWS DynamoDB to AWS S3 (Data Pipeline)

AWS Data Pipeline is a web service that use to automate the movement and transformation of data. In this project, we are using the Data Pipeline to perform data backup. Under Data Pipeline section, click on the “Create new pipeline” to create pipeline with the follow field fill up as below,

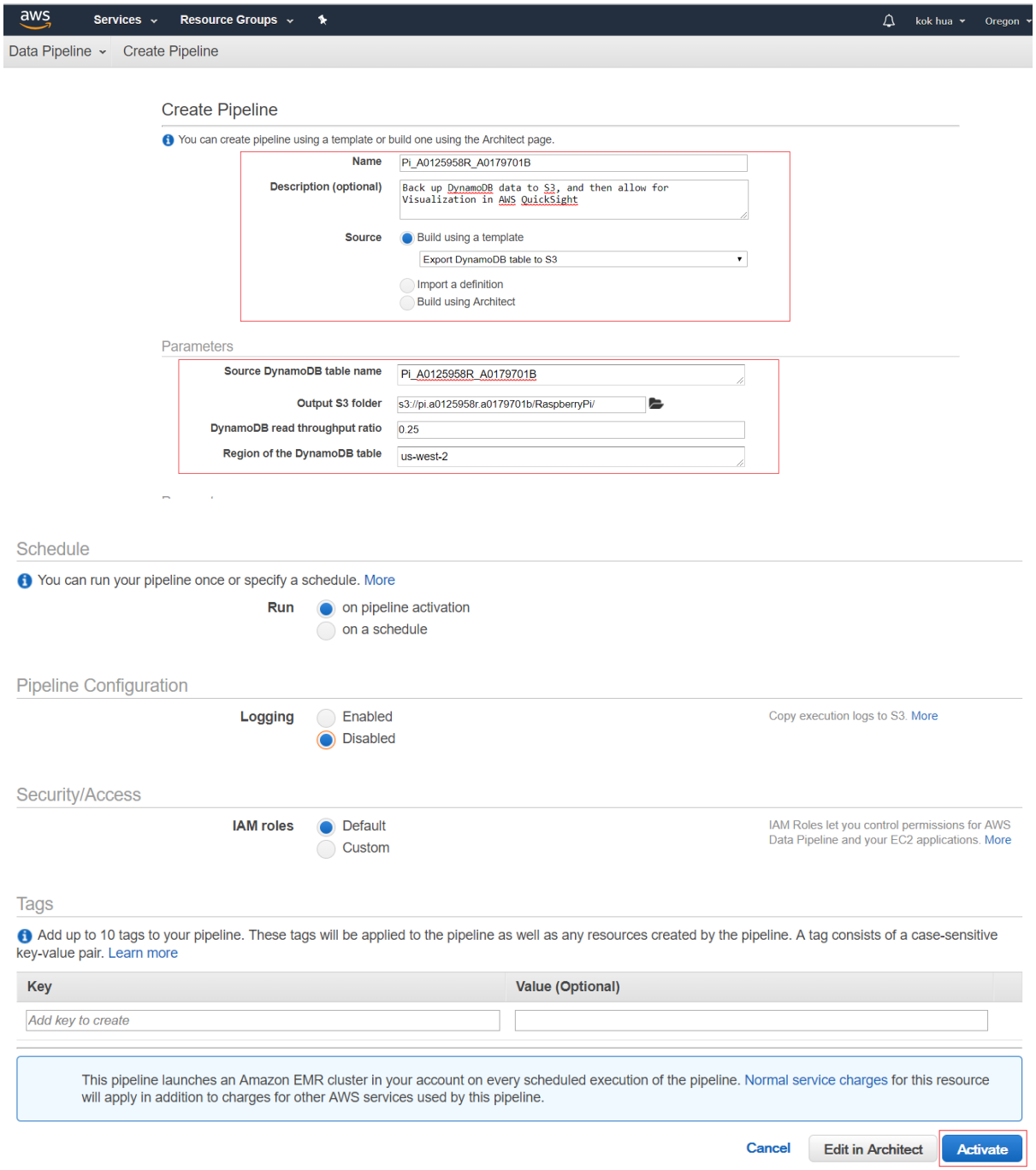

Figure 85: Data Pipeline Creation and Activation

Once the Data Pipeline activated, it take some time to run and complete the backup and copy of data from dynamoDB to S3.

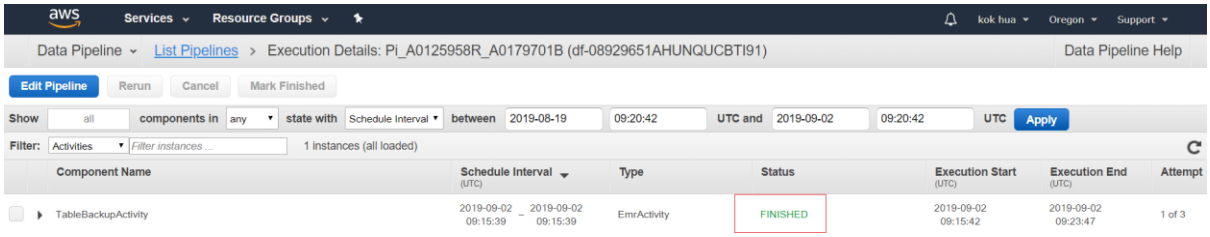

Figure 86: Data Pipeline Finished Backup data from DynamoDB to S3

In S3, a folder is created in RaspberryPi folder. In the folder which contains of 3 files as shown in Figure below. Make all the files to public so the AWS QuickSight could access it later. The data is reside in the file *"d5426601-8d34-4902-91d1-a36e495d13fd"* in JSON format. We will descript that AWS QuickSight to extract the content of this file later.

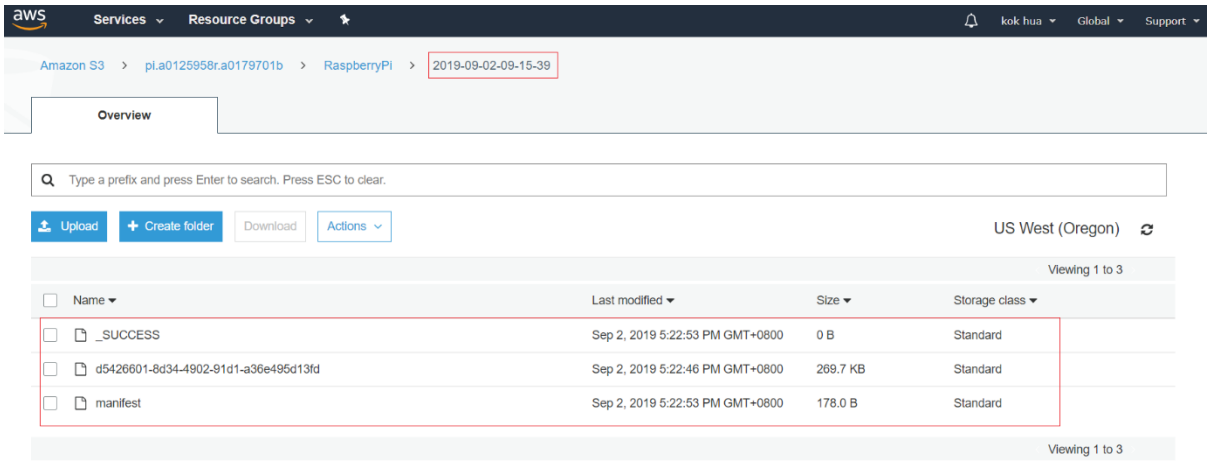

Figure 87: Data backup in S3 successfully

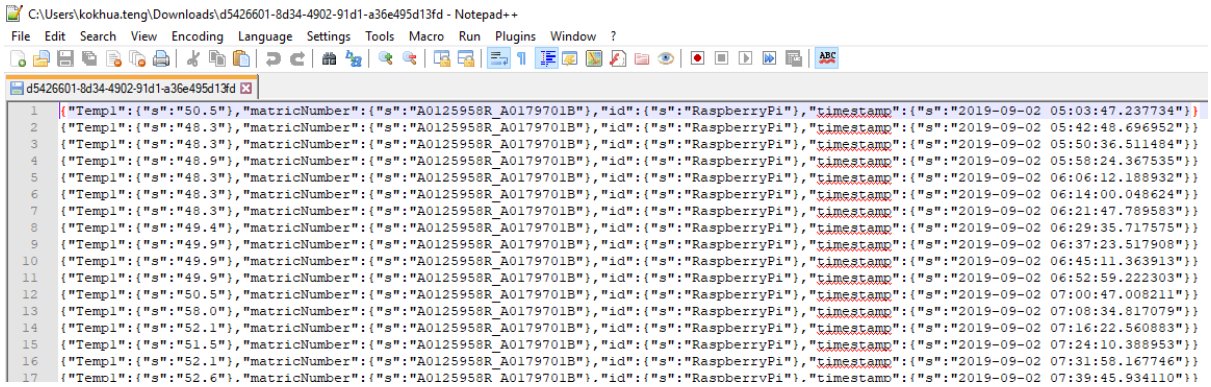

Figure 88: Back up Data by pipeline

AWS QuickSight for Visualization

To start a new visualization in AWS QuickSight by clicking on “New analysis” then “New data set”.

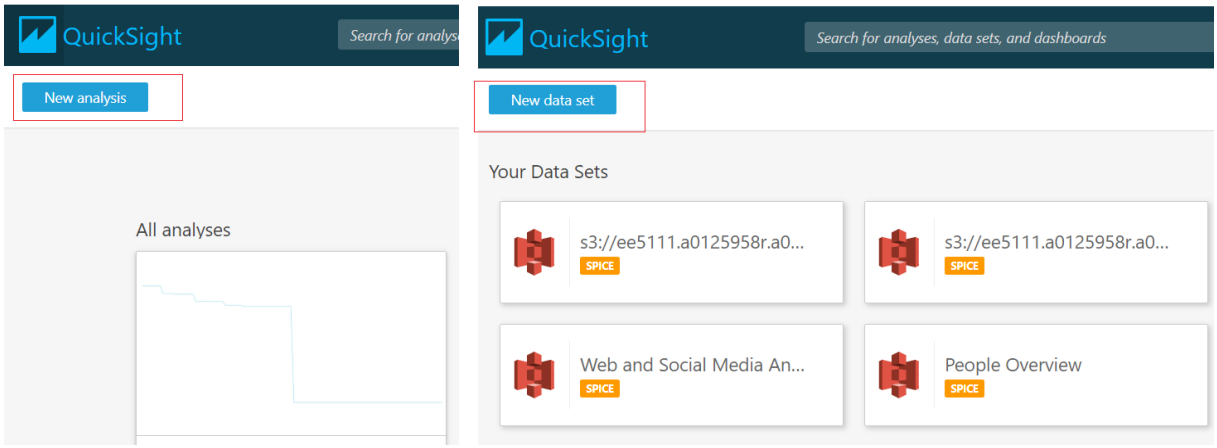

Figure 89: Create New Analysis with New Data Set

Then a window will show to prompt for the data source path and the manifest file. Then go to S3, get the data source path where it store the DynamoDB data in JSON format per screenshot below,

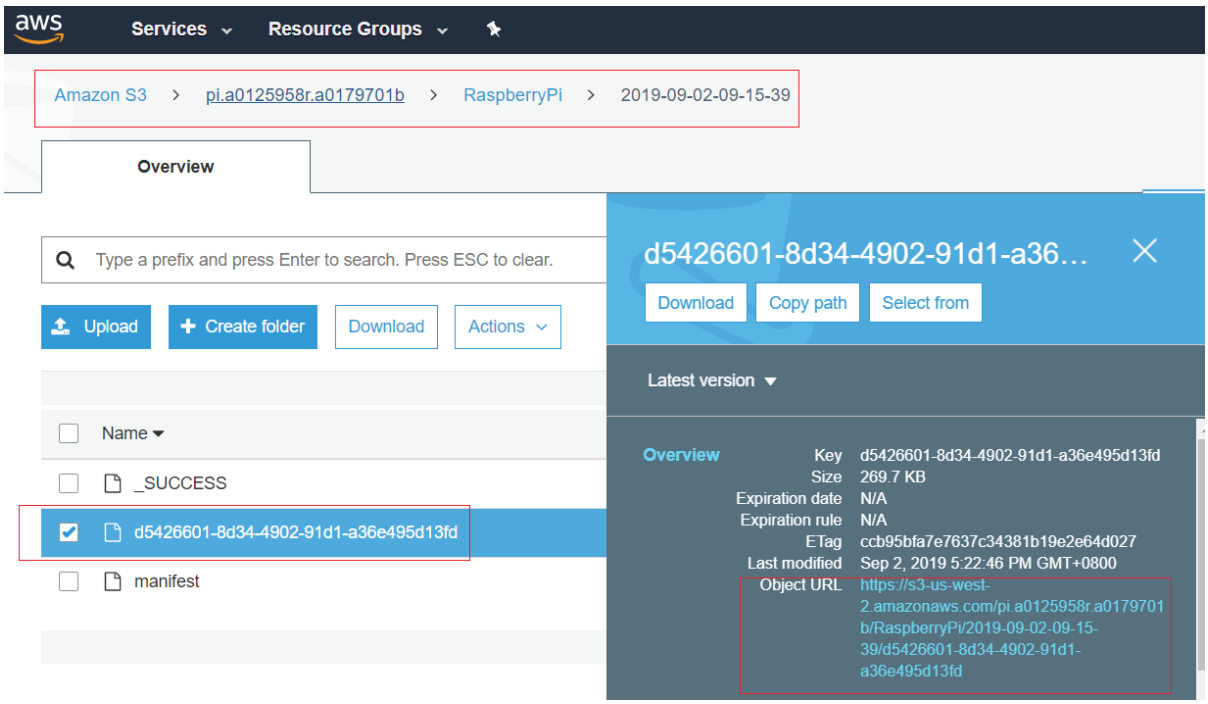

Figure 90: S3 Data Source Path

Enter the S3 data source path and attach the manifest file to AWS QuickSight. Then, click on “Connect”. AWS QuickSight will connect and access to S3 for get the data. The manifest that attached is governs the S3 file location and the data format as shown in Figures below.

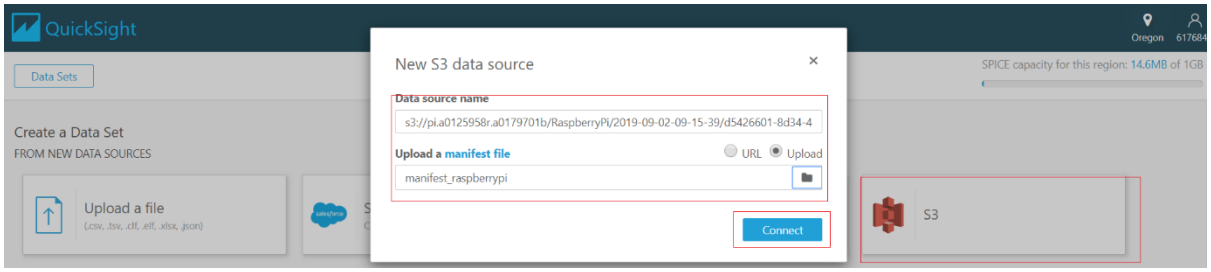

Figure 91: Select S3 and Enter Data Source Path

```
{
    "fileLocations": [
        {
            "URIs": [
                "
                https://s3-us-west-2.amazonaws.com/pi.a0125958r.a0179701b/Raspber
                ryPi/2019-09-02-09-15-39/d5426601-8d34-4902-91d1-a36e495d13fd"
            ]
        },
        {
            "URIPrefixes": [
                "
                https://s3-us-west-2.amazonaws.com/pi.a0125958r.a0179701b/Raspber
                ryPi/2019-09-02-09-15-39/d5426601-8d34-4902-91d1-a36e495d13fd"
            ]
        }
    ],"globalUploadSettings": {"format":"JSON",
    "delimiter":"\n","textqualifier":"'"
    }
}
```

Figure 92: Content of Manifest File

After successfully connected to S3 data source, data is imported to AWS Quicksight. Under the Visualize section, plenty of visual type are available for adding to dashboard. A line chart is selected where x-axis set to timestamp while y-axis set to Temp1 value (Raspberry Pi board temperature). Furthermore, the dashboard is shown in Figure below and it could be share to relevant users via email.

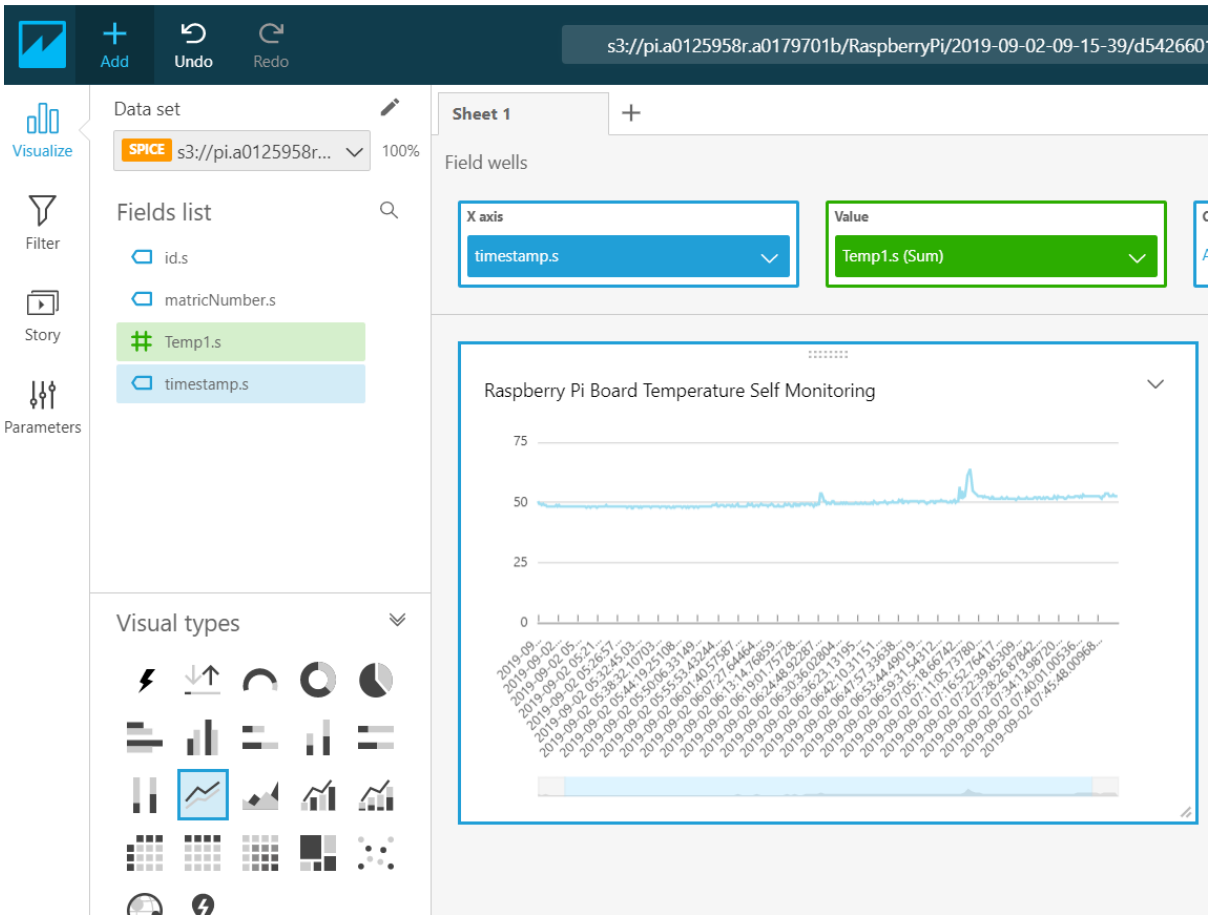

Figure 93: Editor mode in Visualize Section

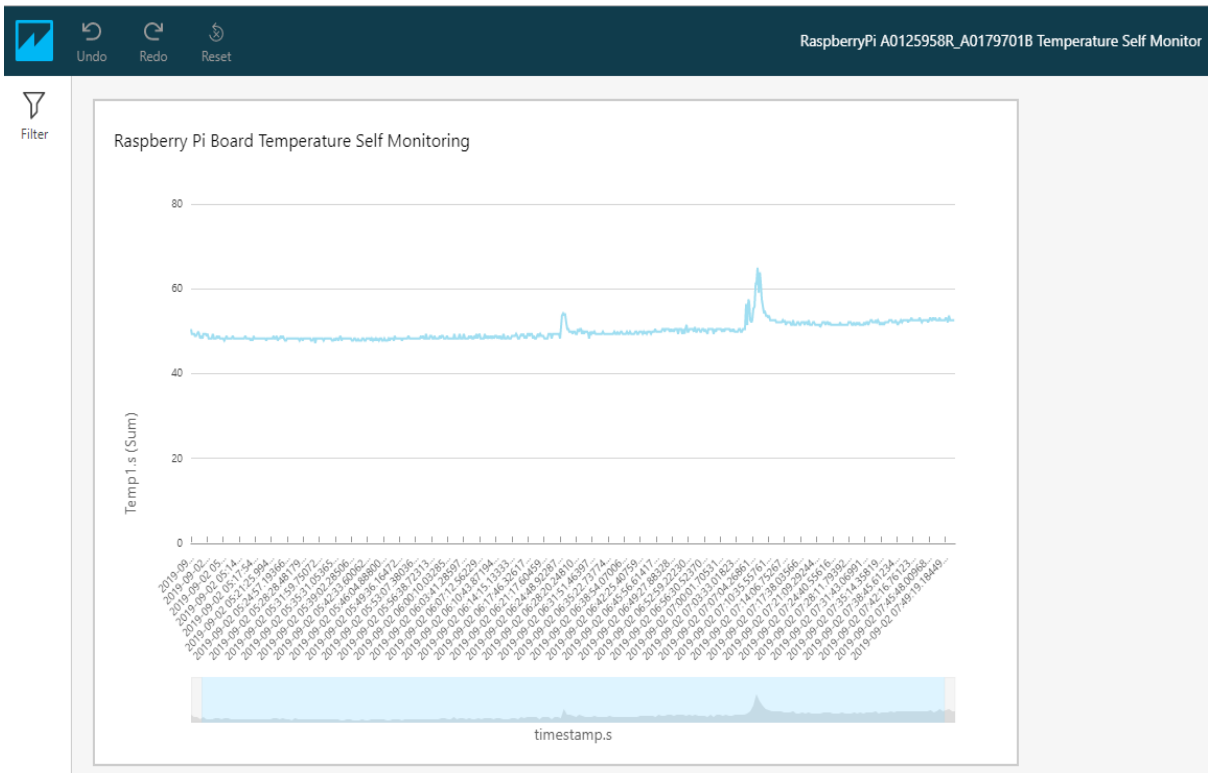

Figure 94: AWS QuickSight Dashboard

Data Prediction with AWS SageMaker for Sea Level (Additional)

In this section, we are predicting time series data using AWS SageMaker which is a fully managed machine learning service. It reduces the complexity of building and training machine learning models by providing several of built in algorithms. The most common algorithm for forecasting scalar (one-dimensional) time series data is DeepAR forecasting algorithm which is supervised learning algorithm [100]. The time based training data use is Global Mean Sea Level (GMSL) Data which can be downloaded from https://climate.nasa.gov/vital-signs/sea-level/ . The following sections are demonstrated the AWS SageMaker notebook creation, pre-process data, train model and test model.

AWS SageMaker Notebook Creation

First go to Amazon SageMaker and then under the Notebook section, select Notebook instances, then choose Create notebook instance (as red boxes). Enter "EE5111" as Notebook instance name. In Permissions and Encryption section, create a new IAM role and give access to S3 bucket "pi.a0125958r.a0179701b". The EE5111 Notebook is created as shown in below (green box).

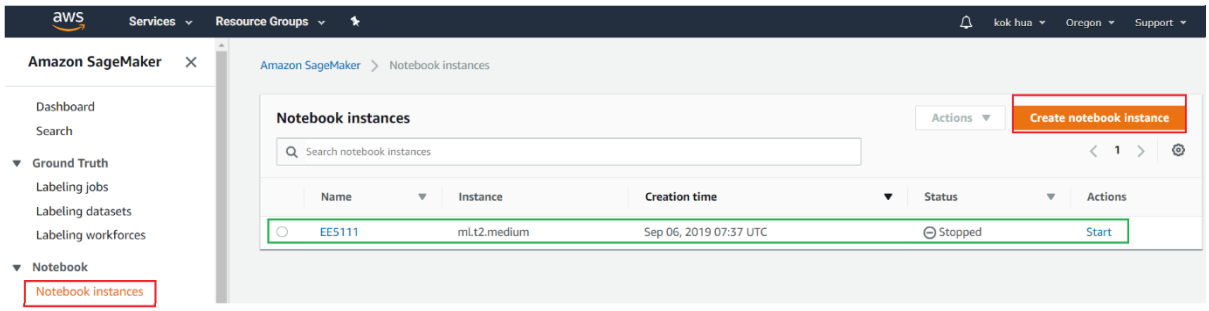

Figure 95: AWS SageMaker Notebook Instances Creation

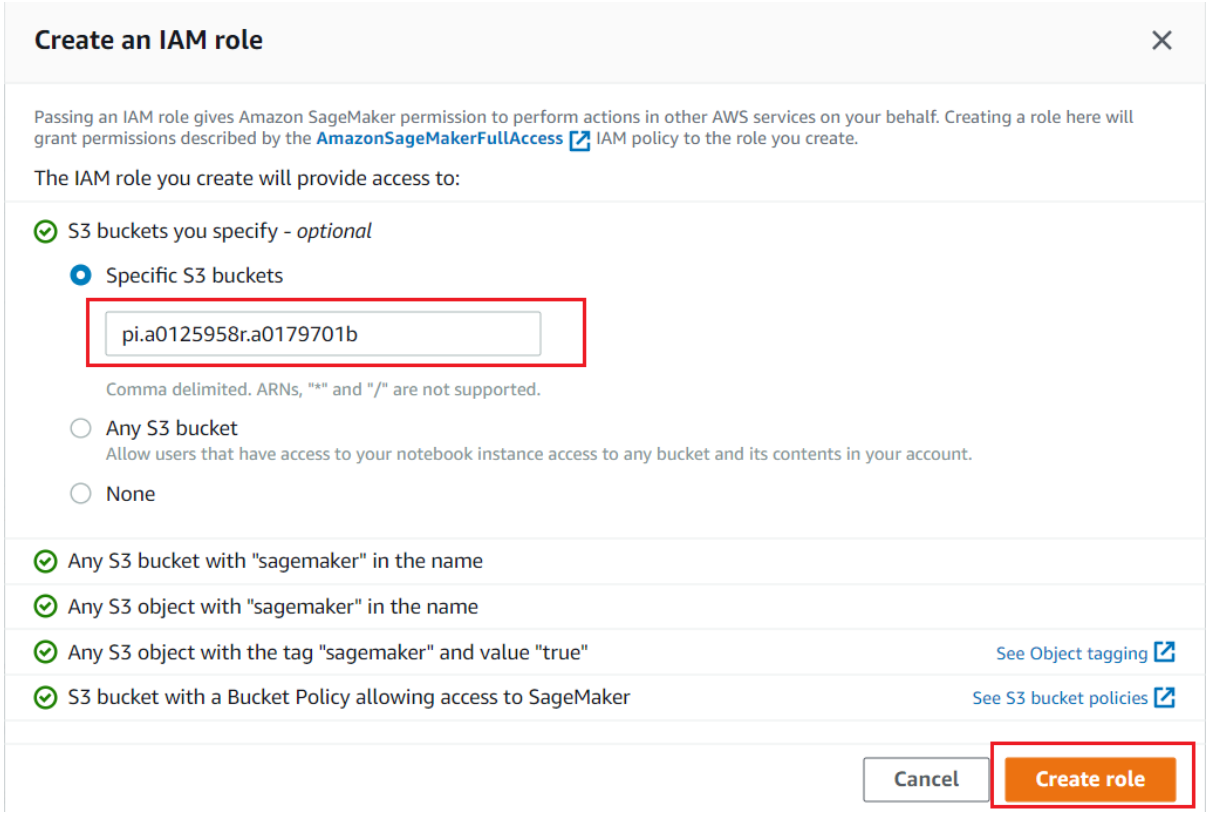

Figure 96: AWS SageMaker Notebook Role Creation

Pre-process Data

Select the “EE5111” Notebook, and click “start”. Once the notebook status shown “InService”, then click on “Open Jupyter” next to its name to open the classic Jupyter dashboard. From dashboard, click on “New” and select “conda_python3” to create a python script (“A0125958R_A0179701B_SeaLevel”) for the data prediction. Click on “Upload” to upload the sea level data that downloaded from NASA.

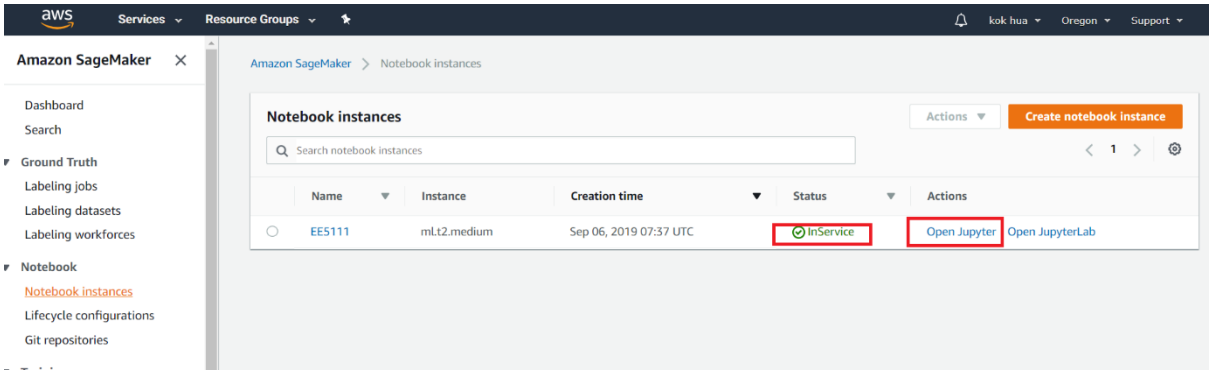

Figure 97: EE5111 Notebook Status

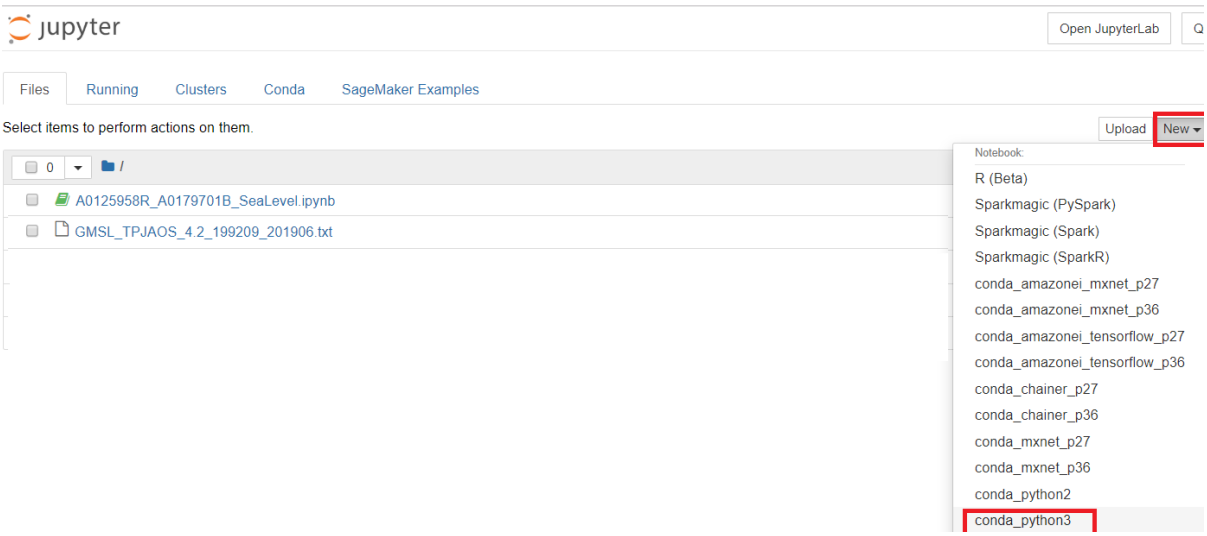

Figure 98: Classic Jupyter Dashboard

The data contains a large number of data points recorded from 1993 to 2019. We are using the smoothed GMSL data in 12th column and with respect to the first data point as our training data set.

GMSL_TPJAOS_4.2_199209_201906.txt

1	0	11	1993.011526	466462	337277.00	-37.24	92.66	-37.02	-37.24	92.66	-37.02	-37.55
2	0	12	1993.038692	460889	334037.31	-40.35	95.39	-38.2	-40.34	95.39	-38.19	-38.06
3	0	13	1993.065858	472123	342416.09	-40.17	92.29	-38.28	-40.16	92.29	-38.27	-37.6
4	0	14	1993.093025	421377	306050.59	-41.92	96.2	-38.56	-41.89	96.19	-38.54	-37.45

Figure 99: GMSL Raw Data

In A0125958R_A0179701B_SurfaceTemp.ipynb, first load the data and perform the data pre-process by grouping the data respect to its year. After that, remove the 2019 data points from the training set while test set hold all the dataset and then write into DeepAR input format as {"start":"1993", "target":[0.0, -0.51, -0.05, 0.1, 1.11,……]}. This is important and we have follow the format strictly in order the DeepAR algorithm to work.

```
# Data download from https://climate.nasa.gov/vital-signs/sea-level/
# Lets perform the data preparation
f = open('GMSL_TPJAOS_4.2_199209_201906.txt', 'r')
data = csv.reader(f,delimiter='\t')

train_key      = 'ee5111_training.json'
test_key       = 'ee5111_test.json'
dataset={}
x=[]
y=[]
count=1
prevYear=0
init_level=-37.55
minYear  = 1993
maxYear  = 2018
prediction_length = 1
for row in data:
        # Remove empty strings caused by multiple spaces between columns
        row = list(filter(None, row))

        year=row[2]
        slevel=float(row[11])-init_level
        year = year[0:4]
        slevel = round(slevel, 4)
        # Data for plotting
        # x list=counter, y list=sealevel
        x.append(count)
        y.append(float(slevel))
        count=count+1

        # Data for training
        # dictionary: key=year, value=list of ordered daily sealevel
        if (year != prevYear):
            dataset[year]=[]
            prevYear=year
        dataset[year].append(float(slevel))
print(y)
nb_samples_per_year = list(map(lambda x: len(x), (dataset[str(year)] for year in range(minYear, maxYear+1))))
nb_samples_per_year = np.unique(nb_samples_per_year).tolist()

fig=plt.figure(figsize=(64, 16))
plt.plot(x,y)
plt.show()

trainingSet = dataset.copy()
del trainingSet["2019"]
testSet = dataset.copy()

def writeDataset(filename, data):
    file=open(filename,'w')
    for year in data.keys():
        # One JSON sample per line
        line = "\"start\":\"{}\", \"target\":{}".format(year,data[year])  # this the require deepar input format
        file.write('{'+line+'}\n')

writeDataset(train_key, trainingSet)
writeDataset(test_key, testSet)
```

Figure 100: AWS SageMaker Code Part 1

Before uploading the data to S3, we need to name the S3 bucket and prefix for training and model data. This should be within the same region as the Notebook Instance, training, and hosting. Then we upload training and test data files to S3.

```
role = get_execution_role()
region = boto3.Session().region_name
bucket='pi.a0125958r.a0179701b' # Replace with your s3 bucket name
prefix = 'SageMaker' # Used as part of the path in the bucket where you store data
bucket_path = 'https://s3-{}.amazonaws.com/{}'.format(region,bucket) # The URL to access the bucket

train_prefix   = '{}/{}'.format(prefix, 'train')
test_prefix    = '{}/{}'.format(prefix, 'test')
output_prefix  = '{}/{}'.format(prefix, 'output')
sagemaker_session = sagemaker.Session()

train_path  = sagemaker_session.upload_data(train_key, bucket=bucket, key_prefix=train_prefix)
test_path   = sagemaker_session.upload_data(test_key,  bucket=bucket, key_prefix=test_prefix)

s3_output_location = 's3://pi.a0125958r.a0179701b/SageMaker'.format(bucket, prefix, 'deepar')
print(train_path)
```

Figure 101: AWS SageMaker Code Part 2

Next we are configuring the training by selecting the corresponding region to create containers to run scripts, train algorithms, or deploy models that are compatible with Amazon SageMaker. Next we need to define hyperparameters: for example, frequency of the time series used, number of data points the model, number of predicted data points. The other hyperparameters concern the model to train (number of layers, number of cells per layer, likelihood function) and the training options such as number of epochs, batch size, and learning rate. In here, we are predicting the 7 data point.

```
containers = {
    'us-east-1': '522234722520.dkr.ecr.us-east-1.amazonaws.com/forecasting-deepar:latest',
    'us-east-2': '566113047672.dkr.ecr.us-east-2.amazonaws.com/forecasting-deepar:latest',
    'us-west-2': '156387875391.dkr.ecr.us-west-2.amazonaws.com/forecasting-deepar:latest',
    'eu-west-1': '224300973850.dkr.ecr.eu-west-1.amazonaws.com/forecasting-deepar:latest'
}

image_name = containers[region]

estimator = sagemaker.estimator.Estimator(
    sagemaker_session=sagemaker_session,
    image_name=image_name,
    role=role,
    train_instance_count=1,
    train_instance_type='ml.c4.8xlarge',
    base_job_name='EE5111',
    output_path=s3_output_location
)
```

```
prediction_length = 7
hyperparameters = {
    "time_freq": 'W', # weekly series
    "context_length": prediction_length,
    "prediction_length": prediction_length, # number of data points to predict
    "num_cells": "40",
    "num_layers": "3",
    #"likelihood": "gaussian",
    "epochs": "200",
    "mini_batch_size": "32",
    "learning_rate": "0.001",
    "dropout_rate": "0.05",
    "early_stopping_patience": "10" # stop if loss hasn't improved in 10 epochs
}

estimator.set_hyperparameters(**hyperparameters)
```

Figure 102: AWS SageMaker Code Part 3

Now, we can start our training. Once the training completed, a model file will be generated and save in S3.

```
data_channels = {"train": train_path, "test": test_path}
```

```
estimator.fit(inputs=data_channels,  logs=True)
```

```
2019-09-10 08:48:05 Uploading - Uploading generated training model
2019-09-10 08:48:05 Completed - Training job completed
Training seconds: 63
Billable seconds: 63
```

Figure 103: AWS SageMaker Code Part 4

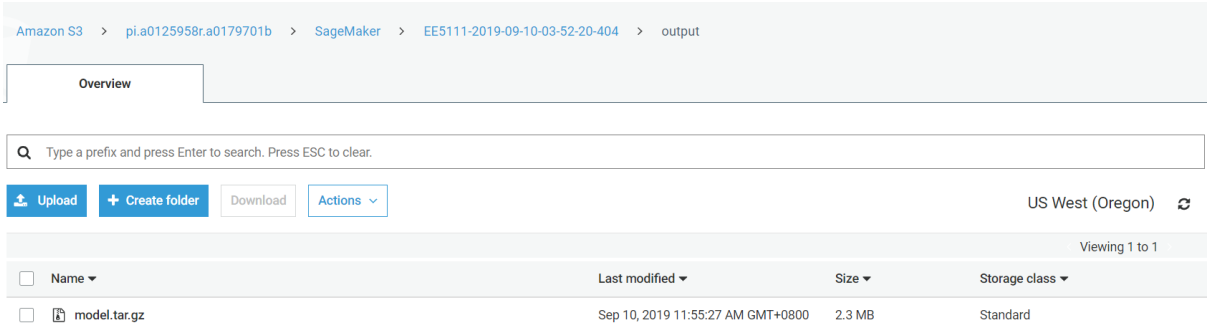

Figure 104: Model Created and Save at S3

Once we have the model trained, then we need to create an endpoint to host our model which is also known as deploying the model.

```
Edit Metadata

job_name = estimator.latest_training_job.name

endpoint_name = sagemaker_session.endpoint_from_job(
    job_name=job_name,
    initial_instance_count=1,
    instance_type='ml.m4.xlarge',
    deployment_image=image_name,
    role=role
)

predictor = sagemaker.predictor.RealTimePredictor(
    endpoint_name,
    sagemaker_session=sagemaker_session,
    content_type="application/json")
```

Figure 105: AWS SageMaker Code Part 5

After model been deployed, we use the model to predict future sea level data. For simplicity, we are predicting the first few series of 2018 data and compare to ground truth.

```
Edit Metadata

year = 2018 #
prediction_data = buildPredictionData(year, trainingSet)
result = predictor.predict(prediction_data)
print(prediction_data)

plotSeries(result,
           truth=True,
           truth_data=trainingSet[str(year)][-prediction_length:],
           truth_label='truth')
```

Figure 106: AWS SageMaker Code Part 6

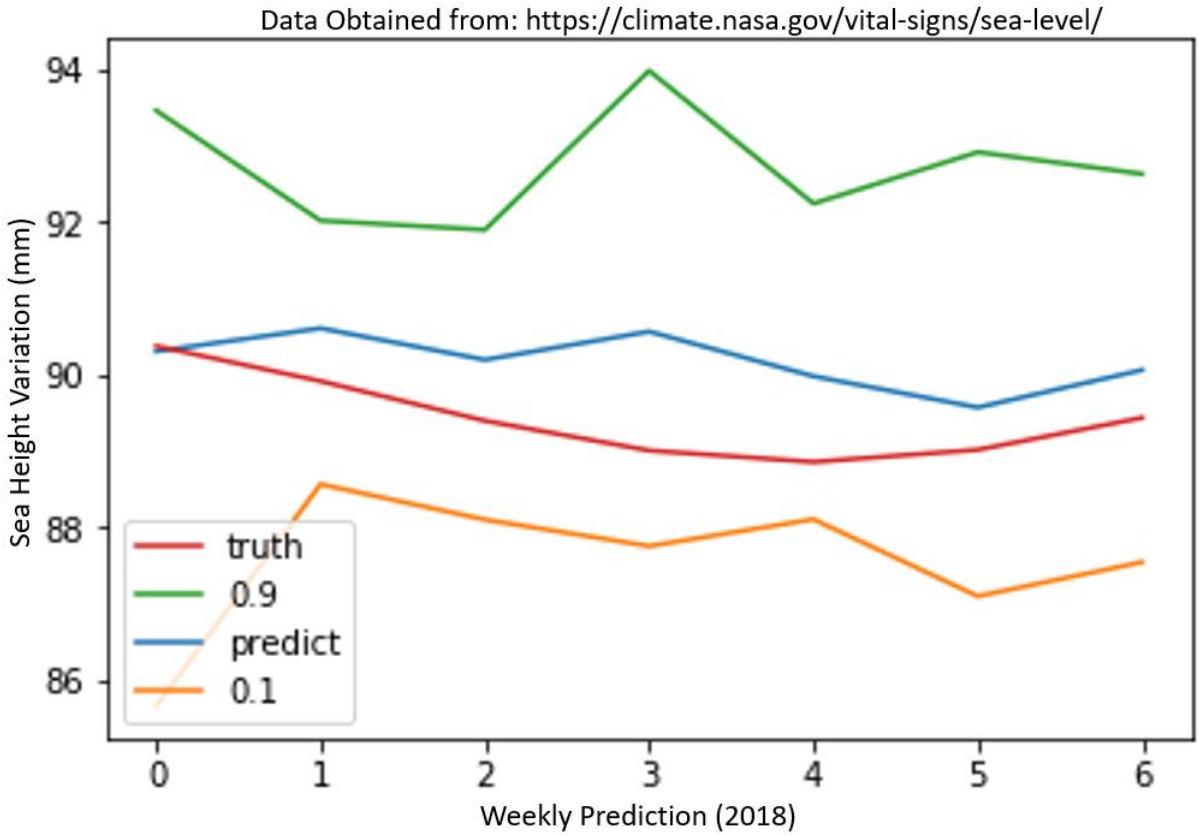

Figure 107: Predict vs Truth (2018)

From the AWS SageMaker dashboard, it summaries our activities. Once training finished and activities completed, stop the notebook instance and deleted the endpoint to prevent additional charges.

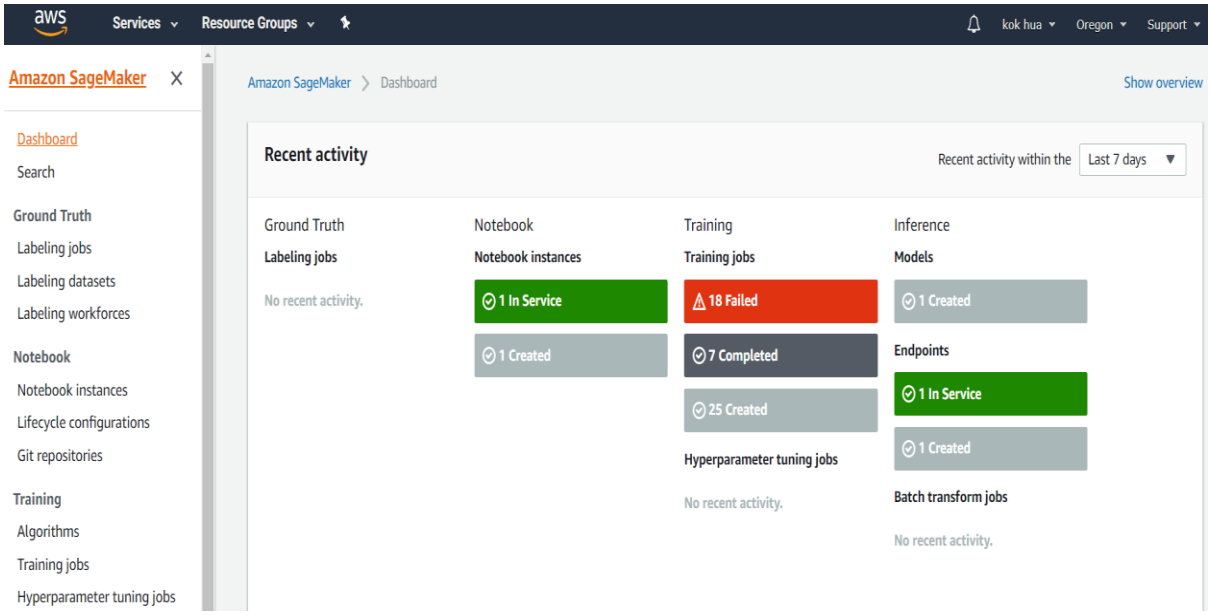

Figure 108: AWS SageMaker Dashboard

Conclusion

In first part of the project, a simulation of two virtual IoT device that setup in two python scripts and published pre-defined jet engine data to AWS IoT with MQTT protocol was achieved. The jet engine data was store to AWS DynamoDB by the rule that defined in AWS IoT platform. Visualization of engine data with Redash which is an open source software was demonstrated. A step by step guidelines could be found in this report.

In the second part, a real-time data ingestion and data publishing to AWS IoT was accomplished with a Raspberry PI that has wireless capability which it act as an IoT device. Raspberry Pi perform self-temperature monitoring and published measured temperature to AWS IoT. The measured real time temperature was then stored in AWS DynamoDB as the rule defined in AWS IoT platform. Besides that, AWS Data Pipeline is established to back up the data to AWS S3 and visualization was achieved with AWS QuickSight.

Last the part, we demonstrated data prediction with AWS SageMaker. Data used was a time series sea height level from NASA and the algorithm used was the built in DeepAR forecasting algorithm in AWS SageMaker. Data preprocessing, upload train and test data set to AWS S3, train the model and deploy the model were performed with cloud computing resource in AWS Sagemaker.

In summary, we have been accomplished IoT, data storage, database, computing, data visualization and machine learning with AWS Cloud Platform. In this report, we provided full detail step by step instructions on the implementations.

Future Work

There is few suggestions for further improvement and upgrades in this project.

1. Real Time Data Visualization – The data in the data visualization does not update real time. Therefore, setup a schedule to update or refresh data in order to enable user to visual the immediate data change.
2. Notification or Alert – In the real world, IoT uses to monitor our environment and connect to user via internet. Thus, IoT could notifies user on any abnormal phenomena observe. This could be achieve by setting up AWS SES to send email notification to users.

References

[1] Github, (2019). “iceberg12”. Retrieve from

https://github.com/iceberg12/NUS_guest_lecture/tree/master/input

[2] AWS, (2019). “what is aws iot”. Retrieve from

https://docs.aws.amazon.com/iot/latest/developerguide/what-is-aws-iot.html

[3] AWS, (2019). “iot ddb rule”. Retrieve from

https://docs.aws.amazon.com/iot/latest/developerguide/iot-ddb-rule.html

[4] AWS, (2019). “sagemaker algo docker registry paths”. Retrieve from

https://docs.aws.amazon.com/sagemaker/latest/dg/sagemaker-algo-docker-registry-paths.html

[5] AWS, (2019). “deepar”. Retrieve from

https://docs.aws.amazon.com/sagemaker/latest/dg/deepar.html

[6] Redash, (2019). “ihelp”. Retrieve from https://redash.io/help/

Appendix

Thing1 Codes:

```
from AWSIoTPythonSDK.MQTTLib import
AWSIoTMQTTShadowClient

import random, time, datetime



# A random programmatic shadow client ID.

SHADOW_CLIENT =
"EE5111_Thing1_A0125958R_A0179701B"



# The unique hostname that &IoT; generated for

# this device.

HOST_NAME = "aekkzs6upiibj-ats.iot.ap-
southeast-1.amazonaws.com"



# The relative path to the correct root CA file for
&IoT;,

# which you have already saved onto this device.

ROOT_CA = "AmazonRootCA1.pem"



# The relative path to your private key file that

# &IoT; generated for this device, which you

# have already saved onto this device.

PRIVATE_KEY = "46de85e2bd-private.pem.key"



# The relative path to your certificate file that

# &IoT; generated for this device, which you

# have already saved onto this device.

CERT_FILE = "46de85e2bd-certificate.pem.crt"



# A programmatic shadow handler name prefix.

SHADOW_HANDLER =
"EE5111_Thing1_A0125958R_A0179701B"



# Automatically called whenever the shadow is
updated.

def myShadowUpdateCallback(payload,
responseStatus, token):

  print()

  print('UPDATE: $aws/things/' +
SHADOW_HANDLER +

    '/shadow/update/#')

  print("payload = " + payload)

  print("responseStatus = " + responseStatus)

  print("token = " + token)



# Create, configure, and connect a shadow client.

myShadowClient =
AWSIoTMQTTShadowClient(SHADOW_CLIEN
T)

myShadowClient.configureEndpoint(HOST_NAM
E, 8883)

myShadowClient.configureCredentials(ROOT_CA
, PRIVATE_KEY,

  CERT_FILE)

myShadowClient.configureConnectDisconnectTim
eout(10)

myShadowClient.configureMQTTOperationTimeo
ut(5)

myShadowClient.connect()



# Create a programmatic representation of the
shadow.

myDeviceShadow =
myShadowClient.createShadowHandlerWithName(

  SHADOW_HANDLER, True)
```

```
#
****************************************
************

# Main script runs from here onwards.

# To stop running this script, press Ctrl+C.

#
****************************************
************



infile = open('train_FD001.txt','r')

outfile = open('train_FD001','a')



# Declare all Data Labels

sensor_name = ['s'+ str(i) for i in range(1,22)]

dataLabels = ['id', 'timestamp', 'Matric_Number',
'te', 'os1', 'os2', 'os3'] + sensor_name



matricNumber = 'A0125958R_A0179701B'



for i in range(0,len(dataLabels)):

    dataLabels[i] = '\"' + dataLabels[i] + '\"'



# Split Columns from FD001.txt files to columns

for line in infile.readlines():

    outfile.write(line)



process = open("train_FD001",'r')



dataString = []

modifiedData = []



head = '{"state":{"reported":{'

tail = '}}}'

# Read data, Add additional texts and Send out
messages

for x in process.readlines():

    newData = x.split(" ")

    modifiedData = []

    modifiedData.append(str('FD001_' +
newData[0]))


modifiedData.append(str(datetime.datetime.utcnow
()))

    modifiedData.append(matricNumber)

    for j in range(2,len(sensor_name)):

        modifiedData.append(newData[j])



    ColumnLabels = []

    ColumnLabels.append(str(dataLabels[0] + ':'))

    ColumnLabels.append(str('"' + modifiedData[0]
+ '",'))

    ColumnLabels.append(str(dataLabels[1] + ':'))

    ColumnLabels.append(str('"' +
str(datetime.datetime.now()) + '",'))

    ColumnLabels.append(str(dataLabels[2] + ':'))

    ColumnLabels.append(str('"' + matricNumber +
'",'))



    for i in range(3,len(dataLabels)):

        ColumnLabels.append(str(dataLabels[i] + ':'))

        ColumnLabels.append(str('"' + newData[i-2] +
'",'))



    string = ''.join(ColumnLabels)

    string = string[:-1]
```

```
    data = []
    data.append(head)
    data.append(string)
    data.append(tail)
    data.append('\n')
    dataString = ''.join(data)
    print(dataString)


myDeviceShadow.shadowUpdate(dataString,mySh
adowUpdateCallback, 5)


    time.sleep(10)
```

Thing 2 Codes:

```
from AWSIoTPythonSDK.MQTTLib import
AWSIoTMQTTShadowClient

import random, time, datetime



# A random programmatic shadow client ID.

SHADOW_CLIENT =
"EE5111_Thing2_A0125958R_A0179701B"



# The unique hostname that &IoT; generated for

# this device.

HOST_NAME = "aekkzs6upiibj-ats.iot.ap-
southeast-1.amazonaws.com"



# The relative path to the correct root CA file for
&IoT;,

# which you have already saved onto this device.

ROOT_CA = "AmazonRootCA1.pem"



# The relative path to your private key file that

# &IoT; generated for this device, which you

# have already saved onto this device.

PRIVATE_KEY = "4275af5f1c-private.pem.key"



# The relative path to your certificate file that

# &IoT; generated for this device, which you

# have already saved onto this device.

CERT_FILE = "4275af5f1c-certificate.pem.crt"



# A programmatic shadow handler name prefix.

SHADOW_HANDLER =
"EE5111_Thing2_A0125958R_A0179701B"



# Automatically called whenever the shadow is
updated.

def myShadowUpdateCallback(payload,
responseStatus, token):

  print()

  print('UPDATE: $aws/things/' +
SHADOW_HANDLER +

    '/shadow/update/#')

  print("payload = " + payload)

  print("responseStatus = " + responseStatus)

  print("token = " + token)



# Create, configure, and connect a shadow client.

myShadowClient =
AWSIoTMQTTShadowClient(SHADOW_CLIEN
T)

myShadowClient.configureEndpoint(HOST_NAM
E, 8883)

myShadowClient.configureCredentials(ROOT_CA
, PRIVATE_KEY,
```

```
  CERT_FILE)

myShadowClient.configureConnectDisconnectTim
eout(10)

myShadowClient.configureMQTTOperationTimeo
ut(5)

myShadowClient.connect()



# Create a programmatic representation of the
shadow.

myDeviceShadow =
myShadowClient.createShadowHandlerWithName(

  SHADOW_HANDLER, True)



#
**********************************************
************

# Main script runs from here onwards.

# To stop running this script, press Ctrl+C.

#
**********************************************
************



infile = open('train_FD002.txt','r')

outfile = open('train_FD002','a')



# Declare all Data Labels

sensor_name = ['s'+ str(i) for i in range(1,22)]

dataLabels = ['id', 'timestamp', 'Matric_Number',
'te', 'os1', 'os2', 'os3'] + sensor_name



matricNumber =
'Thing2_A0125958R_A0179701B'



for i in range(0,len(dataLabels)):

    dataLabels[i] = '\"' + dataLabels[i] + '\"'



# Split Columns from FD002.txt files to columns

for line in infile.readlines():

    outfile.write(line)



process = open("train_FD002",'r')



dataString = []

modifiedData = []



head = '{"state":{"reported":{'

tail = '}}}'

# Read data, Add additional texts and Send out
messages

for x in process.readlines():

    newData = x.split(" ")

    modifiedData = []

    modifiedData.append(str('FD002_' +
newData[0]))


modifiedData.append(str(datetime.datetime.utcnow
()))

    modifiedData.append(matricNumber)

    for j in range(2,len(sensor_name)):

        modifiedData.append(newData[j])



    ColumnLabels = []

    ColumnLabels.append(str(dataLabels[0] + ':'))

    ColumnLabels.append(str('"' + modifiedData[0]
+ '",'))

    ColumnLabels.append(str(dataLabels[1] + ':'))

    ColumnLabels.append(str('"' +
str(datetime.datetime.now()) + '",'))
```

```
    ColumnLabels.append(str(dataLabels[2] + ':'))
    ColumnLabels.append(str("" + matricNumber +
"','))



    for i in range(3,len(dataLabels)):
        ColumnLabels.append(str(dataLabels[i] + ':'))
        ColumnLabels.append(str("" + newData[i-2] +
"','))


    string = ''.join(ColumnLabels)
    string = string[:-1]


    data = []
    data.append(head)
    data.append(string)
    data.append(tail)
    data.append('\n')
    dataString = ''.join(data)
    print(dataString)



myDeviceShadow.shadowUpdate(dataString,mySh
adowUpdateCallback, 5)


    time.sleep(10)
```

Raspberry Pi Codes:

```
from  AWSIoTPythonSDK.MQTTLib import
AWSIoTMQTTShadowClient

import random, time, datetime, os
```

```
# A random programmatic shadow client ID.

SHADOW_CLIENT =
"Pi_A0125958R_A0179701B"



# The unique hostname that &IoT; generated for

# this device.

HOST_NAME = "a1xli7oxtwsplh-ats.iot.us-west-
2.amazonaws.com"



# The relative path to the correct root CA file for
&IoT;,

# which you have already saved onto this device.

ROOT_CA = "AmazonRootCA1.pem"



# The relative path to your private key file that

# &IoT; generated for this device, which you

# have already saved onto this device.

PRIVATE_KEY = "2b35f1588c-private.pem.key"



# The relative path to your certificate file that
```

```
# &IoT; generated for this device, which you

# have already saved onto this device.

CERT_FILE = "2b35f1588c-certificate.pem.crt"



# A programmatic shadow handler name prefix.

SHADOW_HANDLER =

"Pi_A0125958R_A0179701B"



# Automatically called whenever the shadow is

updated.

def myShadowUpdateCallback(payload,

responseStatus, token):

  print()

  print('UPDATE: $aws/things/' +

SHADOW_HANDLER +

    '/shadow/update/#')

  print("payload = " + payload)

  print("responseStatus = " + responseStatus)

  print("token = " + token)



# Create, configure, and connect a shadow client.

myShadowClient =

AWSIoTMQTTShadowClient(SHADOW_CLIEN

T)

myShadowClient.configureEndpoint(HOST_NAM

E, 8883)

myShadowClient.configureCredentials(ROOT_CA

, PRIVATE_KEY,

  CERT_FILE)

myShadowClient.configureConnectDisconnectTim

eout(10)

myShadowClient.configureMQTTOperationTimeo

ut(5)

myShadowClient.connect()

myShadowClient.disconnect()

myShadowClient.connect()

# Create a programmatic representation of the

shadow.

myDeviceShadow =

myShadowClient.createShadowHandlerWithName(

  SHADOW_HANDLER, True)



def get_temp():
```

```
    temp = os.popen("vcgencmd
measure_temp").readline()
    return (temp.replace("temp=",""))

#
****************************************
************
# Main script runs from here onwards.
# To stop running this script, press Ctrl+C.
#
****************************************
************

# Get the labels out
sensor_name = ['Temp'+ str(i) for i in range(1,2)]
dataLabels = ['id', 'timestamp', 'matricNumber'] +
sensor_name

matricNumber = 'A0125958R_A0179701B'

for i in range(0,len(dataLabels)):
    dataLabels[i] = '\"' + dataLabels[i] + '\"'

dataString = []
modifiedData = []

head = '{"state":{"reported":{'
tail = '}}}'
count = 0

while True:

    temp = get_temp()
    temp = temp[0:4]
    print(temp)
    count = count+1
    print(count);

    modifiedData = []
    modifiedData.append(str('Raspberry' + 'Pi'))
```

```
modifiedData.append(str(datetime.datetime.utcnow
()))
    modifiedData.append(matricNumber)
    for j in range(2,len(sensor_name)):
        modifiedData.append(temp)


    ColumnLabels = []
    ColumnLabels.append(str(dataLabels[0] + ':'))
    ColumnLabels.append(str('"' + modifiedData[0]
+ '",'))
    ColumnLabels.append(str(dataLabels[1] + ':'))
    ColumnLabels.append(str('"' +
str(datetime.datetime.now()) + '",'))
    ColumnLabels.append(str(dataLabels[2] + ':'))
    ColumnLabels.append(str('"' + matricNumber +
'",'))



    for i in range(3,len(dataLabels)):
        ColumnLabels.append(str(dataLabels[i] + ':'))
        ColumnLabels.append(str('"' + temp + '",'))

    string = ''.join(ColumnLabels)
    string = string[:-1]


    data = []
    data.append(head)
    data.append(string)
    data.append(tail)
    data.append('\n')
    dataString = ''.join(data)
    print(dataString)


myDeviceShadow.shadowUpdate(dataString,mySh
adowUpdateCallback, 5)


    time.sleep(5)
    if count>2000:break
```

AWS SageMaker Machine Learning Codes:

```
#!/usr/bin/env python
# coding: utf-8
# In[]: Import Library
```

```
import os

import boto3

import re

import copy

import time

import json

import csv

import numpy as np

from time import gmtime, strftime

from sagemaker import get_execution_role

from matplotlib import pyplot as plt

import sagemaker

from sagemaker.amazon.amazon_estimator import
get_image_uri




# In[]:



# Data download from
https://climate.nasa.gov/vital-signs/sea-level/

# Lets perform the data preparation

f =
open('GMSL_TPJAOS_4.2_199209_201906.txt',
'r')

data = csv.reader(f,delimiter='\t')



train_key      = 'ee5111_training.json'

test_key       = 'ee5111_test.json'

dataset={}

x=[]

y=[]

count=1

prevYear=0

init_level=-37.55

minYear  = 1993

maxYear  = 2018

prediction_length = 1

for row in data:

      # Remove empty strings caused by multiple
spaces between columns

      row = list(filter(None, row))
```

```
        year=row[2]
        slevel=float(row[11])-init_level
        year = year[0:4]
        slevel = round(slevel, 4)
        # Data for plotting
        # x list=counter, y list=sealevel
        x.append(count)
        y.append(float(slevel))
        count=count+1


        # Data for training
        # dictionary: key=year, value=list of ordered
daily sealevel
        if (year != prevYear):
            dataset[year]=[]
            prevYear=year
        dataset[year].append(float(slevel))
print(y)
nb_samples_per_year = list(map(lambda x: len(x),
(dataset[str(year)] for year in range(minYear,
maxYear+1))))
nb_samples_per_year =
np.unique(nb_samples_per_year).tolist()


fig=plt.figure(figsize=(64, 16))
plt.plot(x,y)
plt.show()


trainingSet = dataset.copy()
del trainingSet["2019"]
testSet = dataset.copy()


def writeDataset(filename, data):
    file=open(filename,'w')
    for year in data.keys():
        # One JSON sample per line
        line = "\"start\":\"{}\",
\"target\":{}".format(year,data[year])  # this the
require deepar input format
        file.write('{'+line+'}\n')


writeDataset(train_key, trainingSet)
```

```
writeDataset(test_key, testSet)
```

```
# In[]::
```

```
role = get_execution_role()

region = boto3.Session().region_name

bucket='pi.a0125958r.a0179701b' # Replace with your s3 bucket name

prefix = 'SageMaker' # Used as part of the path in the bucket where you store data

bucket_path = 'https://s3-{}.amazonaws.com/{}'.format(region,bucket) # The URL to access the bucket
```

```
train_prefix   = '{}/{}'.format(prefix, 'train')

test_prefix    = '{}/{}'.format(prefix, 'test')

output_prefix  = '{}/{}'.format(prefix, 'output')

sagemaker_session = sagemaker.Session()
```

```
train_path  = sagemaker_session.upload_data(train_key, bucket=bucket, key_prefix=train_prefix)

test_path   = sagemaker_session.upload_data(test_key, bucket=bucket, key_prefix=test_prefix)
```

```
s3_output_location = 's3://pi.a0125958r.a0179701b/SageMaker'.format(bucket, prefix, 'deepar')

print(train_path)
```

```
# In[]:
```

```
containers = {

   'us-east-1': '522234722520.dkr.ecr.us-east-1.amazonaws.com/forecasting-deepar:latest',

   'us-east-2': '566113047672.dkr.ecr.us-east-2.amazonaws.com/forecasting-deepar:latest',
```

```
    'us-west-2': '156387875391.dkr.ecr.us-west-
2.amazonaws.com/forecasting-deepar:latest',
    'eu-west-1': '224300973850.dkr.ecr.eu-west-
1.amazonaws.com/forecasting-deepar:latest'
}

image_name = containers[region]

estimator = sagemaker.estimator.Estimator(
    sagemaker_session=sagemaker_session,
    image_name=image_name,
    role=role,
    train_instance_count=1,
    train_instance_type='ml.c4.8xlarge',
    base_job_name='EE5111',
    output_path=s3_output_location
)

# In[]:

prediction_length = 7
hyperparameters = {
    "time_freq": 'W', # weekly series
    "context_length": prediction_length,
    "prediction_length": prediction_length, # number
of data points to predict
    "num_cells": "40",
    "num_layers": "3",
    #"likelihood": "gaussian",
    "epochs": "200",
    "mini_batch_size": "32",
    "learning_rate": "0.001",
    "dropout_rate": "0.05",
    "early_stopping_patience": "10" # stop if loss
hasn't improved in 10 epochs
}

estimator.set_hyperparameters(**hyperparameters)
```

```
# In[]::


data_channels = {"train": train_path, "test":
test_path}


# In[]:


estimator.fit(inputs=data_channels,  logs=True)


# In[]:


job_name = estimator.latest_training_job.name


endpoint_name =
sagemaker_session.endpoint_from_job(
    job_name=job_name,
    initial_instance_count=1,
    instance_type='ml.m4.xlarge',
    deployment_image=image_name,
    role=role
)


predictor =
sagemaker.predictor.RealTimePredictor(
    endpoint_name,
    sagemaker_session=sagemaker_session,
    content_type="application/json")


# In[]:


q1 = '0.1'        # compute p10 quantile
q2 = '0.9'        # compute p90 quantile
num_samples = 50  # predict 50 sample series
```

```
def buildPredictionData(year, data):
    year_temps = data[str(year)][-prediction_length:]
    s = {"start": "{}-01-01 00:00:00".format(year),
"target": year_temps}
    series = []
    series.append(s)
    configuration = {
        "output_types": ["mean", "quantiles",
"samples"],
        "num_samples": num_samples,
        "quantiles": [q1, q2]

    }
    http_data = {
        "instances": series,
        "configuration": configuration
    }
    return json.dumps(http_data)
```

```
# In[]:


def getPredictedSeries(result):
    import random
    json_result = json.loads(result)
    y_data      = json_result['predictions'][0]
    y_mean      = y_data['mean']
    y_q1        = y_data['quantiles'][q1]
    y_q2        = y_data['quantiles'][q2]
    y_sample    =
y_data['samples'][random.randint(0,
num_samples)]
    return y_mean, y_q1, y_q2, y_sample


# In[]:


def plotSeries(result, truth=False, truth_data=None,
truth_label=None):
```

```
    x = range(0,prediction_length)

    y_mean, y_q1, y_q2, y_sample =
getPredictedSeries(result)

    plt.gcf().clear()

    predict_label,   = plt.plot(x, y_mean,
label='predict')

    q1_label,     = plt.plot(x, y_q1, label=q1)

    q2_label,     = plt.plot(x, y_q2, label=q2)

    #sample_label, = plt.plot(x, y_sample,
label='sample')



    if truth:

        ground_truth, = plt.plot(x, truth_data,
label=truth_label)

        plt.legend(handles=[ground_truth, q2_label,
predict_label, q1_label])

    else:

        plt.legend(handles=[q2_label, predict_label,
q1_label])

    plt.yticks()

    plt.show()
```

```
# In[]:



year = 2018 # “He who controls the past controls
the future."

prediction_data = buildPredictionData(year,
trainingSet)

result = predictor.predict(prediction_data)

print(prediction_data)



plotSeries(result,

        truth=True,

        truth_data=trainingSet[str(year)][-
prediction_length:],

        truth_label='truth')

# In[]:

year = 2019

year_sealevel={}

year_sealevel[str(year)] = np.random.normal(90,
1.0, 7).tolist()
```

```
prediction_data = buildPredictionData(year,
year_sealevel)

result = predictor.predict(prediction_data)

print(prediction_data)

plotSeries(result,
        truth=True,
        truth_data=year_sealevel[str(year)][-
prediction_length:],
        truth_label='truth')

# In[ ]:

# In[ ]:

# In[ ]:
```