**Vivekanand Education Society’s**

**Institute of Technology**

**(Affiliated to University of Mumbai, Approved by AICTE & Recognized by Govt. of Maharashtra)**

**Department of Information Technology**

IOE Lab

CA Assignment - 3

**Aim: Implement Edge to cloud Protocols (Minimum 3) using a dummy data set.**

| Roll No. | 70 |
| --- | --- |
| Name | MAYURI SHRIDATTA YERANDE |
| Class | D20B |
| Subject | Internet of Everything |
| Grade: |  |

**AIM**: Implement Edge to cloud Protocols using a dummy data set.

**TO-DO:**

* Explain in brief each of the protocols
* Code
* Output.

**THEORY**:

**MQTT:**  
Message Queuing Telemetry Transport (MQTT) is a lightweight messaging protocol designed for use in situations where low bandwidth, high latency, or unreliable networks are common. It was developed by IBM in the late 1990s but has since become an open standard with a wide range of implementations and use cases. MQTT is particularly popular in the Internet of Things (IoT) and machine-to-machine (M2M) communication due to its efficiency and flexibility.

**Key features of MQTT:**

1. **Publish/Subscribe Model:** MQTT follows a publish/subscribe messaging pattern. In this model, there are two main entities: publishers and subscribers. Publishers send messages (also known as "publish" messages) to specific topics, and subscribers express interest in specific topics by subscribing to them. When a message is published to a topic, all subscribers interested in that topic receive the message.
2. **Quality of Service (QoS) Levels:** MQTT supports three different levels of message delivery assurance, which allows you to choose the level of reliability you need:

QoS 0 (At most once): Messages are delivered with no confirmation. This is the fastest but least reliable option.

QoS 1 (At least once): Messages are guaranteed to be delivered at least once to the receiver. Some duplicate messages may occur.

QoS 2 (Exactly once): Messages are guaranteed to be delivered exactly once. This is the most reliable but comes with higher overhead.

1. **Broker:** MQTT uses a central message broker as an intermediary between publishers and subscribers. The broker is responsible for routing messages from publishers to the appropriate subscribers based on topic subscriptions. Popular MQTT brokers include Mosquitto, HiveMQ, and AWS IoT Core.
2. **Topics:** Topics are hierarchical strings used to categorize messages. Subscribers can express interest in specific topics using wildcard characters. For example, a subscriber could subscribe to "sensors/temperature" to receive temperature data from various sensors or "sensors/+" to receive data from all sensors.
3. **Last Will and Testament (LWT):** Clients can specify a "last will" message to be published by the broker in case the client unexpectedly disconnects. This is useful for handling client failures gracefully.
4. **Retained Messages:** MQTT allows the broker to retain the last message sent on a specific topic. When a new subscriber subscribes to a topic with a retained message, it immediately receives the last retained message.

**HTTPS:**

Using HTTPS (Hypertext Transfer Protocol Secure) for IoT (Internet of Things) devices and applications is a crucial security practice to protect data and communication in IoT ecosystems. HTTPS is the secure version of HTTP, and it employs encryption, authentication, and data integrity mechanisms to ensure that data transmitted between IoT devices, servers, and services remains confidential and tamper-proof.

**Key features of HTTPS:**

1. **Data Encryption:** HTTPS uses Transport Layer Security (TLS) or its predecessor, Secure Sockets Layer (SSL), to encrypt data during transmission. This encryption ensures that sensitive information exchanged between IoT devices and servers cannot be easily intercepted or deciphered by unauthorized parties.
2. **Authentication:** TLS in HTTPS provides a means for mutual authentication between the IoT device and the server. This helps in verifying the identity of both parties involved in the communication. Device authentication ensures that only authorized devices can access specific services or data, and server authentication ensures that devices are communicating with legitimate servers.
3. **Data Integrity:** HTTPS also ensures data integrity by employing cryptographic hashes to verify that the data received has not been tampered with during transmission. This helps prevent "man-in-the-middle" attacks where an attacker might modify data packets in transit.
4. **Privacy:** Using HTTPS enhances user privacy by preventing eavesdroppers from monitoring the data being transmitted between IoT devices and servers. This is especially important when dealing with personal or sensitive data.
5. **Secure APIs:** Many IoT ecosystems involve web-based APIs for data exchange and control. Securing these APIs with HTTPS is essential to prevent unauthorized access and data exposure.
6. **End-to-End Security:** HTTPS provides end-to-end security, which means that data is protected from the IoT device through the network to the server and back. This holistic approach to security is crucial in ensuring the overall security of the IoT system.

**WEBSOCKET:**

WebSocket API is a communication protocol that enables full-duplex, bidirectional data transfer between a client and server over a single, long-lived connection. It's commonly used for real-time applications such as chat, gaming, and live updates.

**Key Features:**

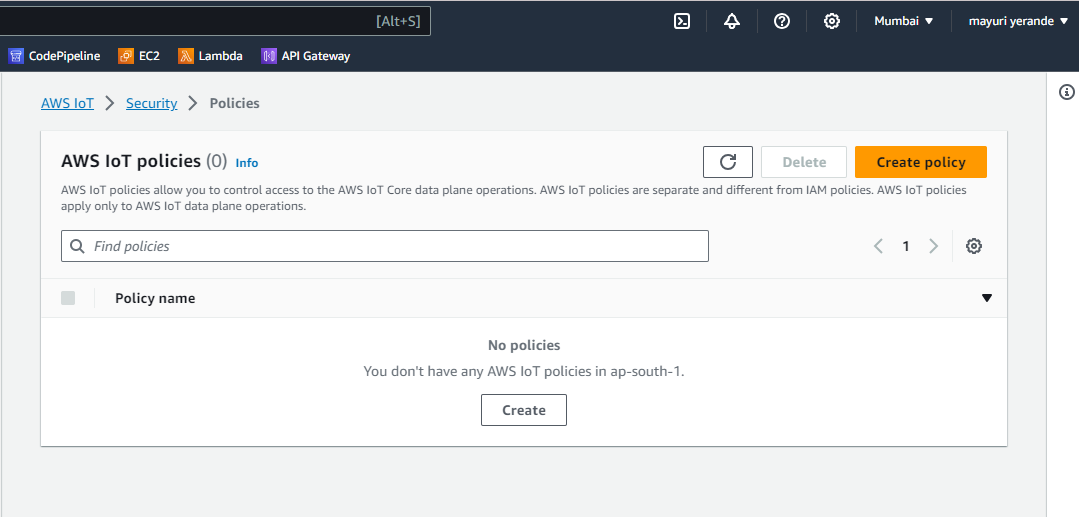
1. **Low Latency:** Provides low-latency, real-time communication by maintaining an open connection, reducing the overhead of repeatedly establishing connections.
2. **Bi-directional Communication:** Allows both the client and server to initiate data exchanges, facilitating real-time interactions.
3. **Scalability:** Supports horizontal scaling and is well-suited for handling a large number of concurrent connections efficiently.
4. **Efficient for Real-time Applications:** It's ideal for real-time applications like online gaming, instant messaging, live streaming, and collaborative tools where low latency is crucial.
5. **Reduced Overhead:** WebSocket eliminates the need for repeatedly opening and closing connections, reducing the overhead associated with traditional HTTP requests.
6. **Two-way Communication:** Enables full two-way communication between the client and server, making it easy to push updates and notifications to clients.
7. **Bi-directional Flow:** Supports simultaneous data transmission in both directions, allowing clients and servers to send messages independently.

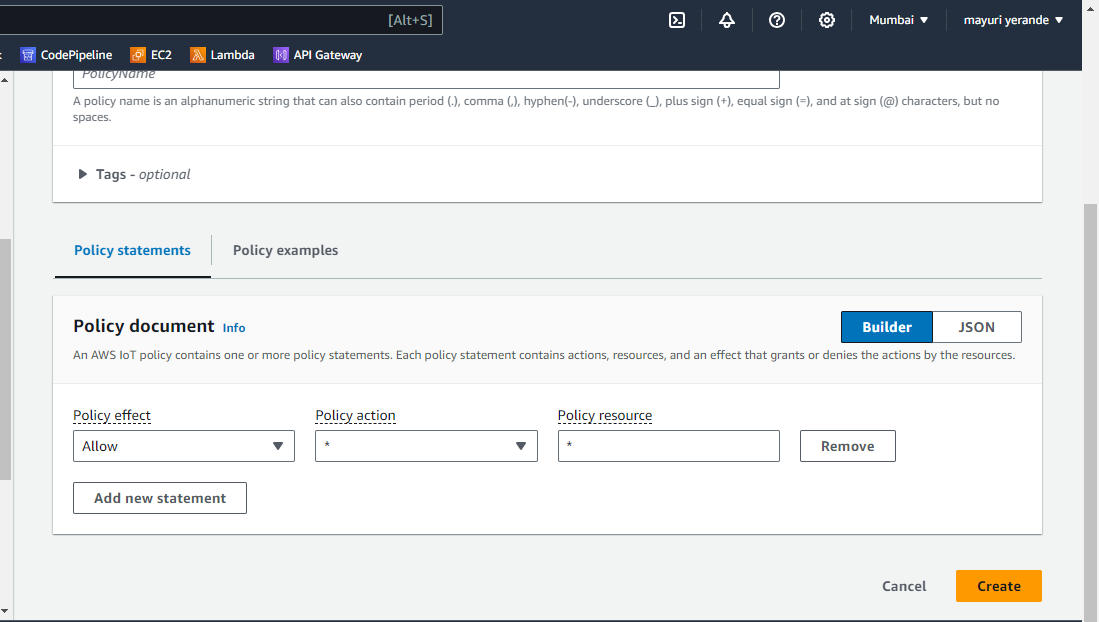
**IMPLEMENTATION:**

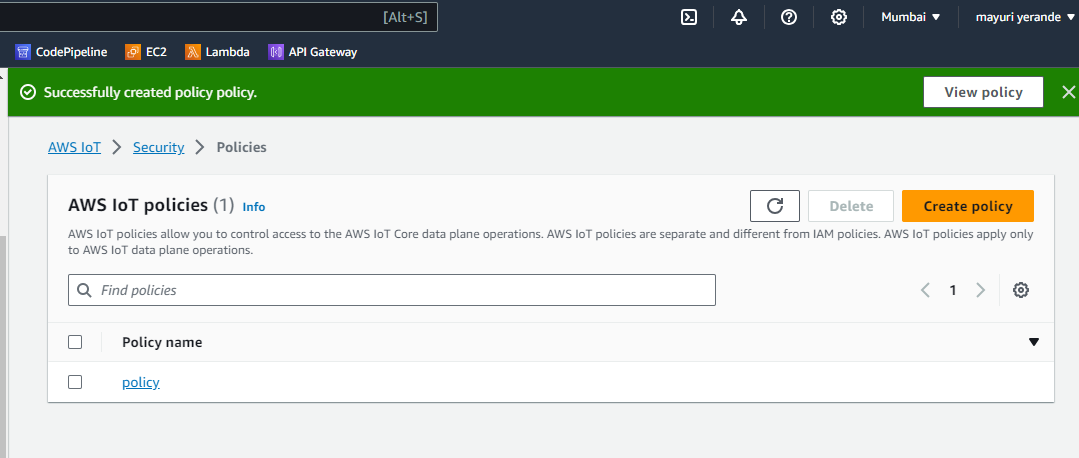
* **MQTT**

**Step 1:** Search for IOT Core

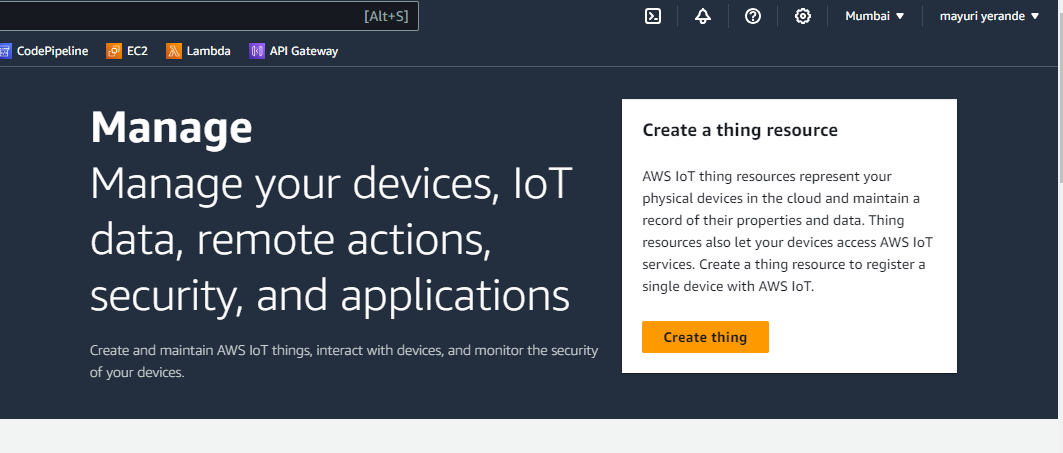
**Step 2:** Create a policy add add policy rule

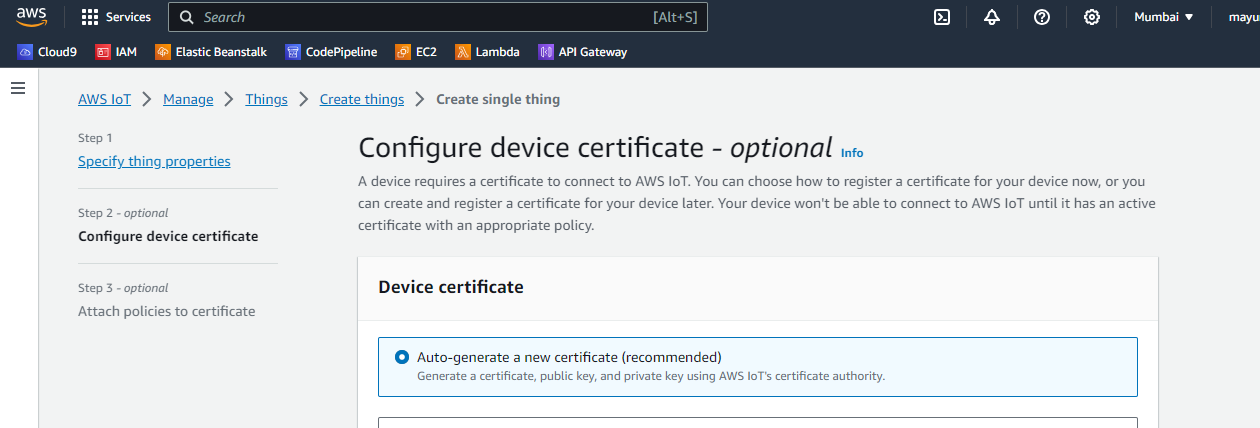
****

****

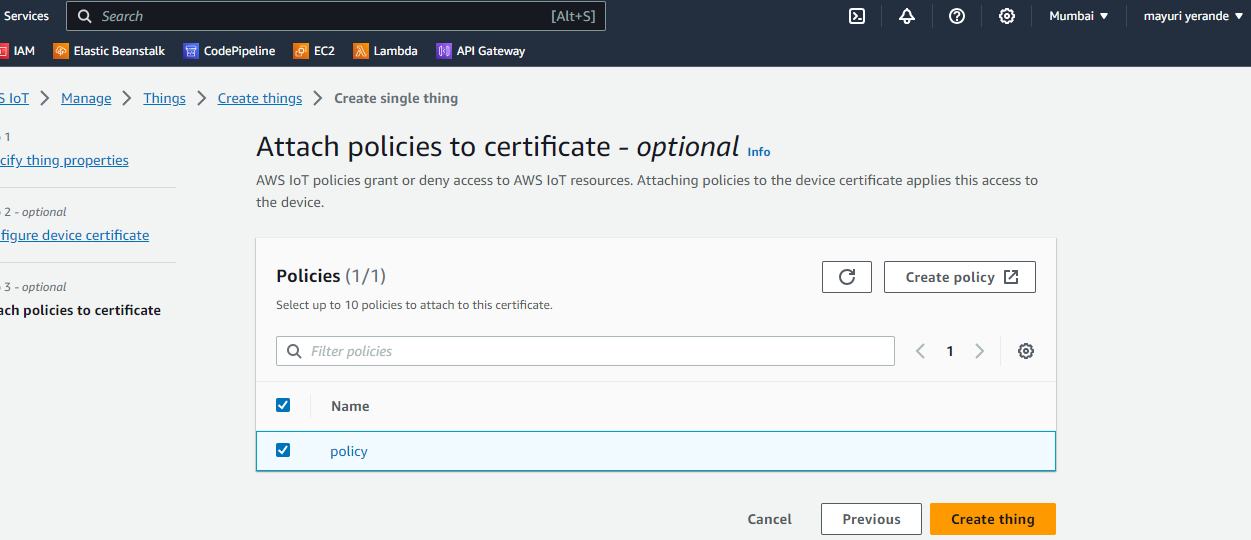
****

**Step 3:** Click on create thing and add the thing name

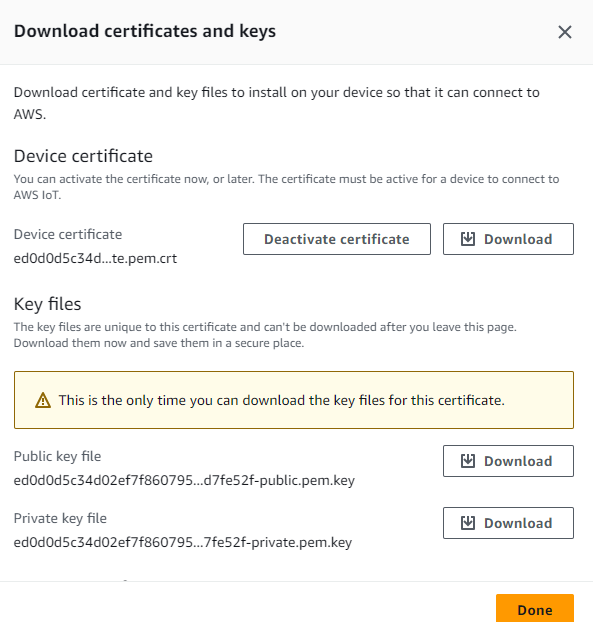
****

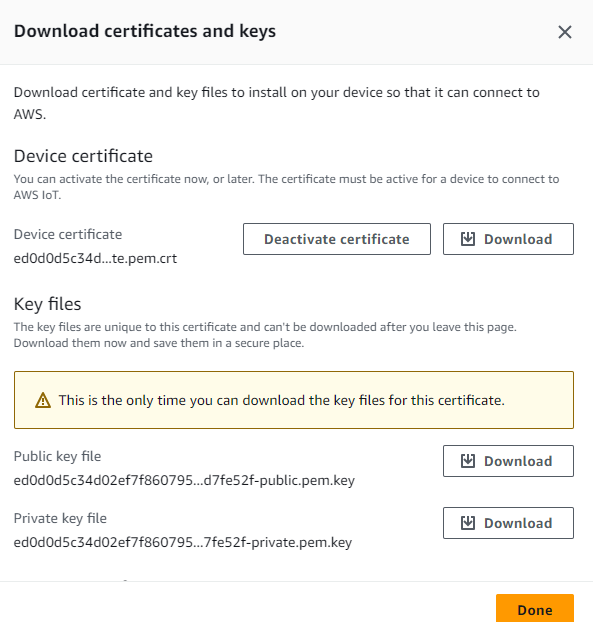
****

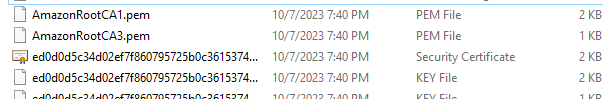
**Step 5:** Attach policy and click on create thing

****

**Step 6:** Download all certificates and keys at the same location

****

****

****

**Step 7:** Clone AWS-MQTT-SDK git repo using the below link in the same folder where you have downloaded the keys and certificate

-git clone https://github.com/aws/aws-iot-device-sdk-python.git

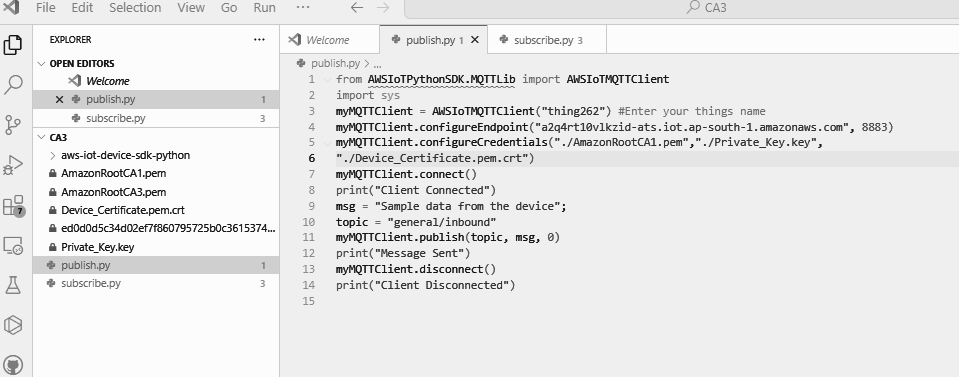
**Step 8:** After cloning Run below commands

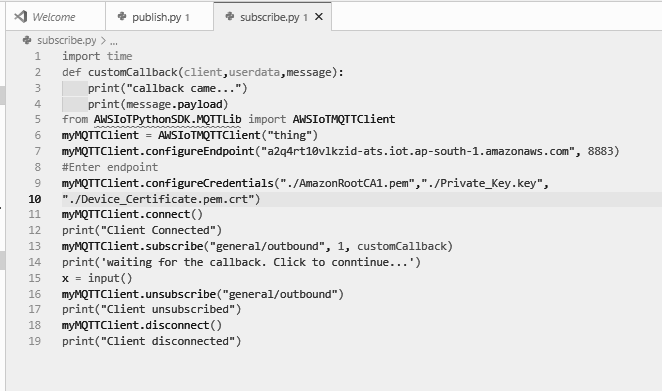
-cd aws-iot-device-sdk-python

-python setup.py install (2)

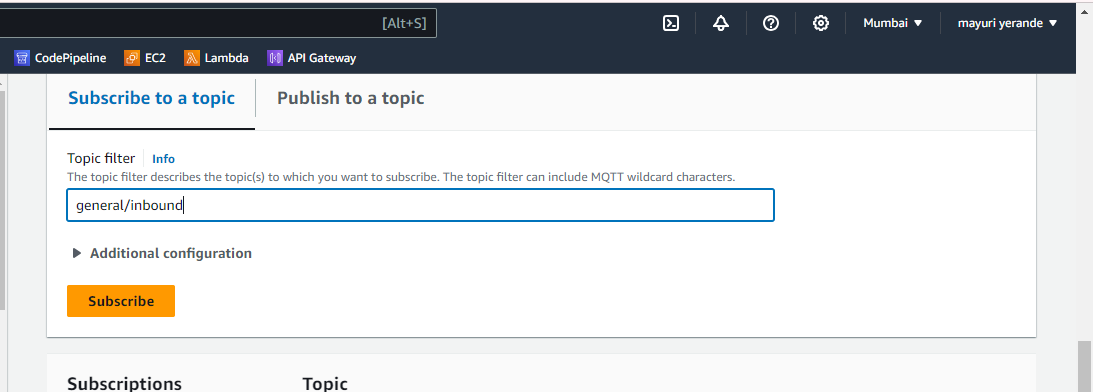
**Step 9:** In main directory save two codes one to publish message to MQTT server and one to subscribe

**Step 10:** In both of above codes replace endpoint with your MQTT test client endpoint

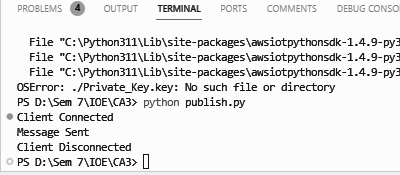
****

****

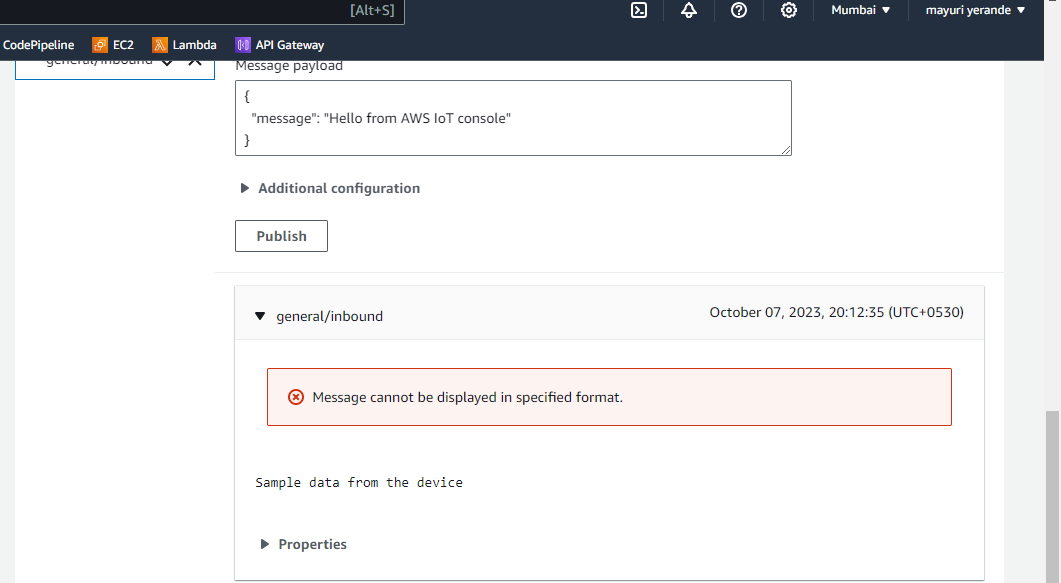
**Step 11:** In MQTT go to Subscribe to a topic and type ‘general/inbound’ and click subscribe ,do not close this page.

****

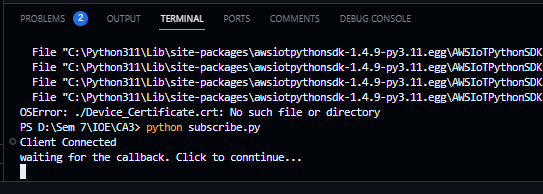
**Step 12:** After that run publish.py and go to subscribe page

****

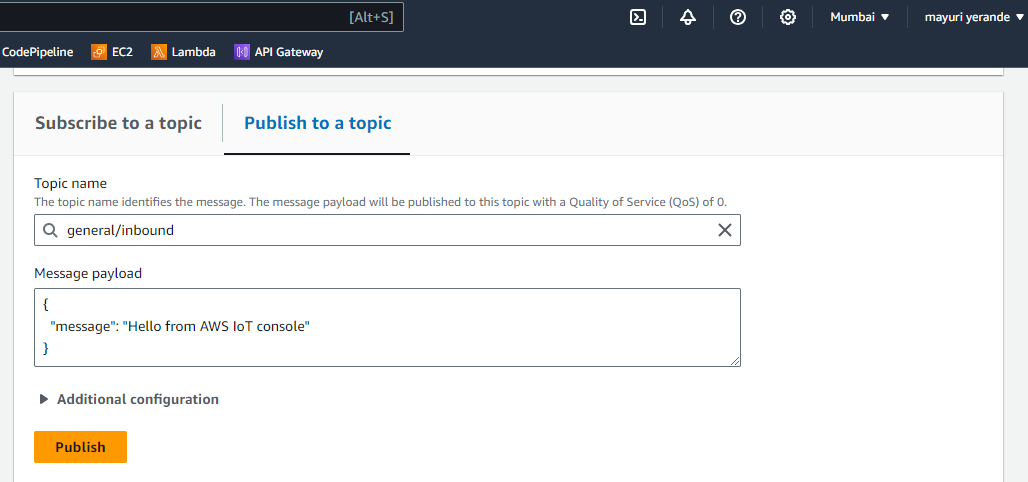
**Step 13:** You can see message on subscribe page

****

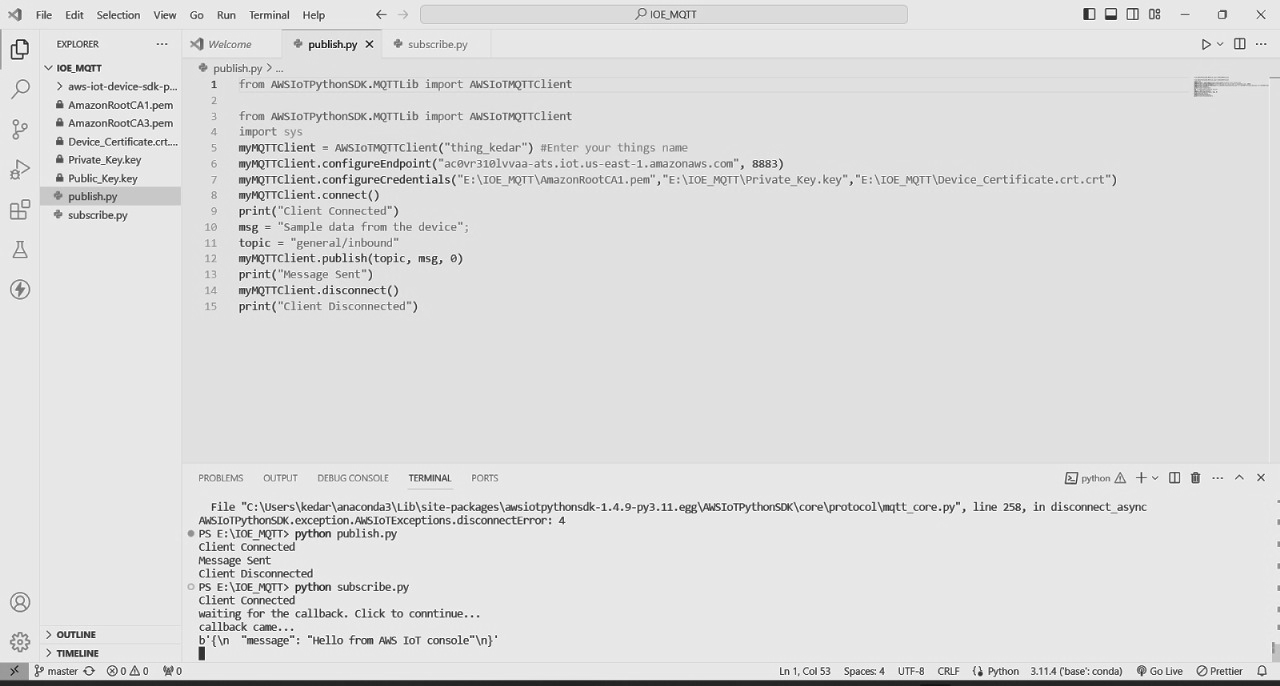
**Step 14:** Then run subscribe.py

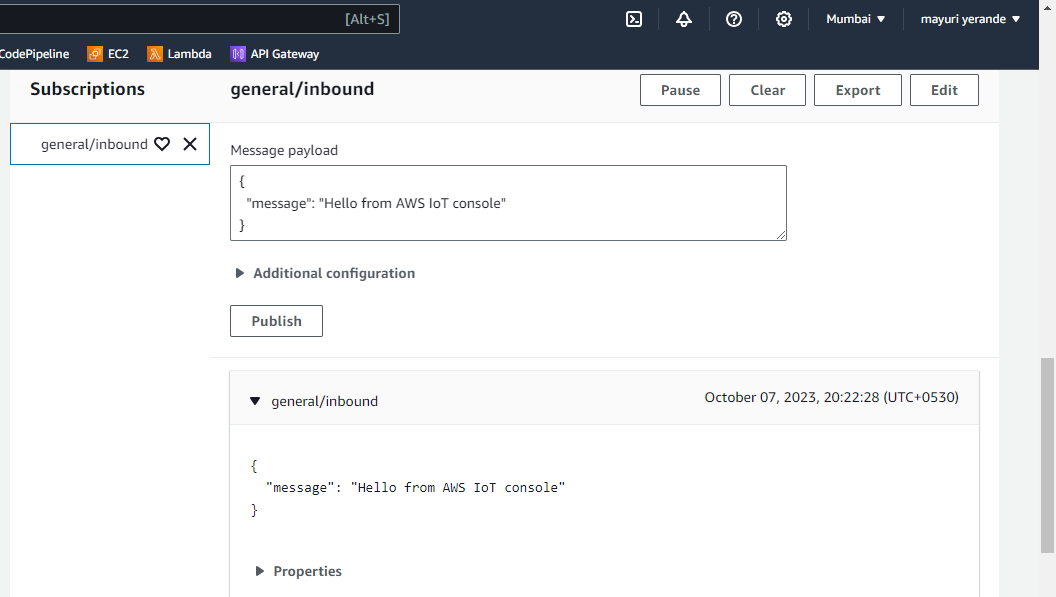
****

**Step 15:** Go to publish to topic and type general/outbound and click on publish

****

**Step 16:** After clicking on publish go to terminal and you can see the message

****

****

* **HTTPS**

**Code**

import requests

import argparse

# define command-line parameters

parser = argparse.ArgumentParser(description="Send messages through an HTTPS connection.")

parser.add\_argument('--endpoint', required=True, help="Your AWS IoT data custom endpoint,not including a port. " +"Ex:\"abcdEXAMPLExyz-ats.iot.us-east-1.amazonaws.com\"")

parser.add\_argument('--cert', required=True, help="File path to your client certificate, in PEM format.")

parser.add\_argument('--key', required=True, help="File path to your private key, in PEM format.")

parser.add\_argument('--topic', required=True, default="test/topic", help="Topic to publish messages to.")

parser.add\_argument('--message', default="Hello World!", help="Message to publish. " + "Specify empty string to publish nothing.")

# parse and load command-line parameter values

args = parser.parse\_args()

publish\_url = 'https://' + args.endpoint + ':8443/topics/' + args.topic + '?qos=1'

publish\_msg = args.message.encode('utf-8')

# make request

publish = requests.request('POST',publish\_url, data=publish\_msg, cert=[args.cert, args.key])

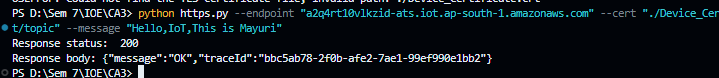
# print results

print("Response status: ", str(publish.status\_code))

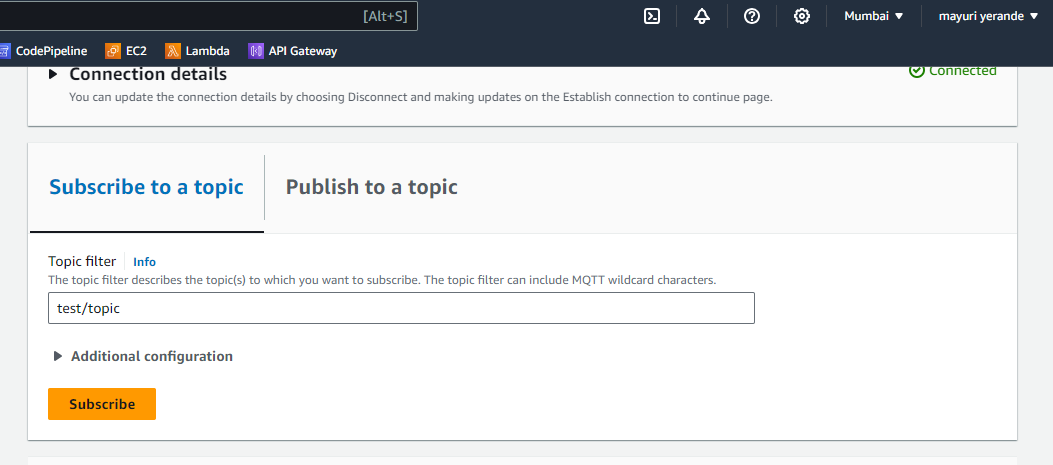
if publish.status\_code == 200:

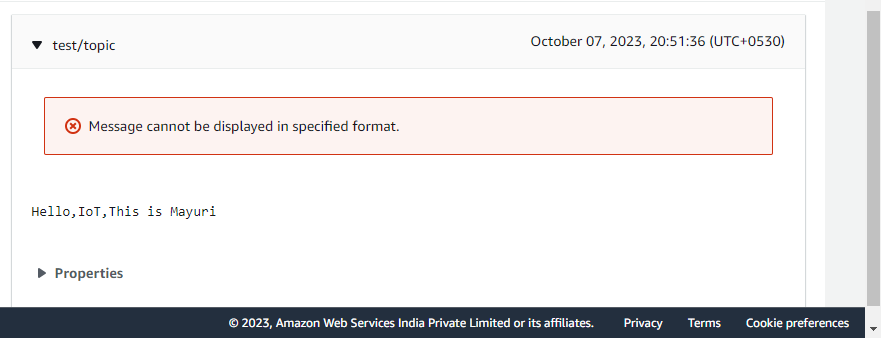
print("Response body:", publish.text)

**Step 1:** Change your endpoint and device certificate and private key file name in the below code python http\_ioe.py --endpoint "a1x4rget4tmruj-ats.iot.eu-north-1.amazonaws.com" --cert "./Device\_Certificate.crt" --key "./Private\_Key.key" --topic "test/topic" --message "Hello, IoT, This is Mayuri”

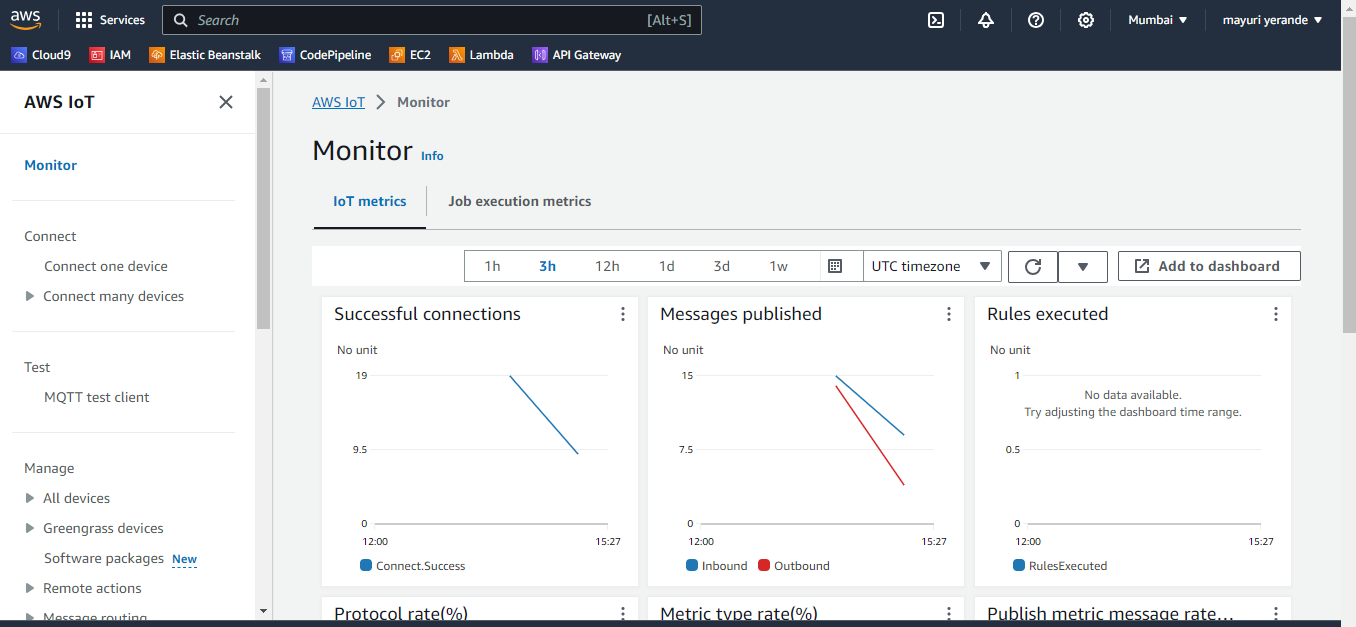
****

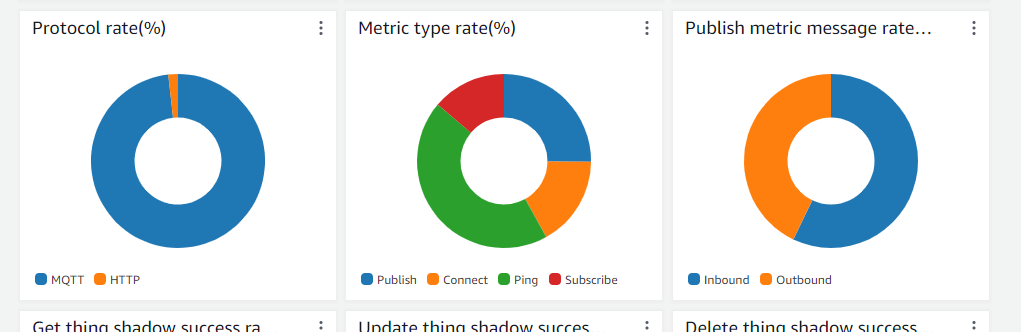
**Step 2:** Check message received by subscribing to the topic

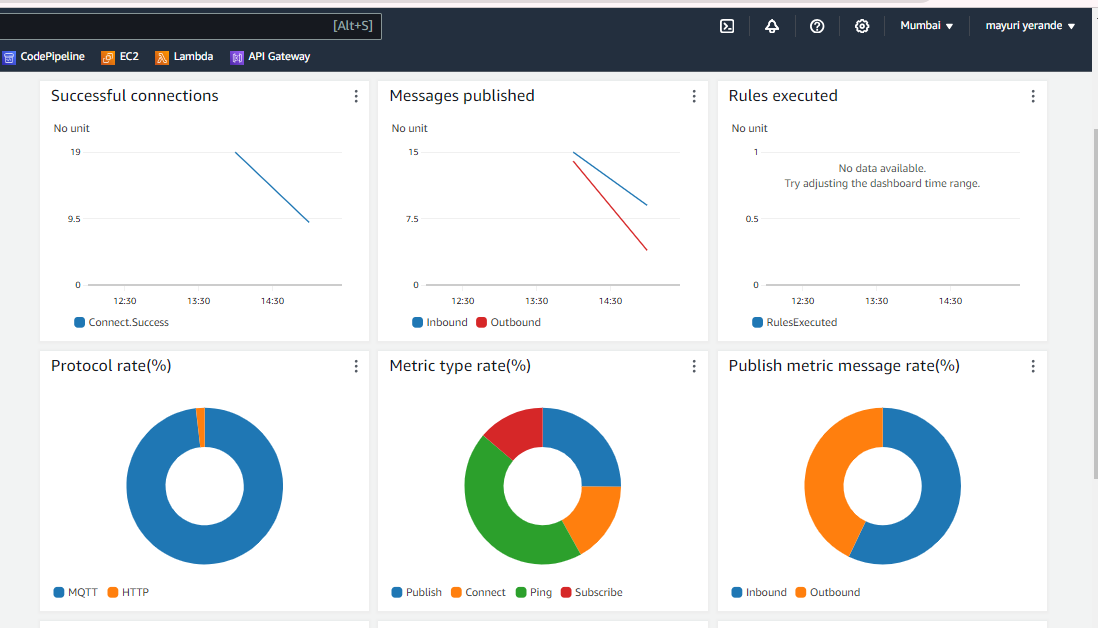
****

****

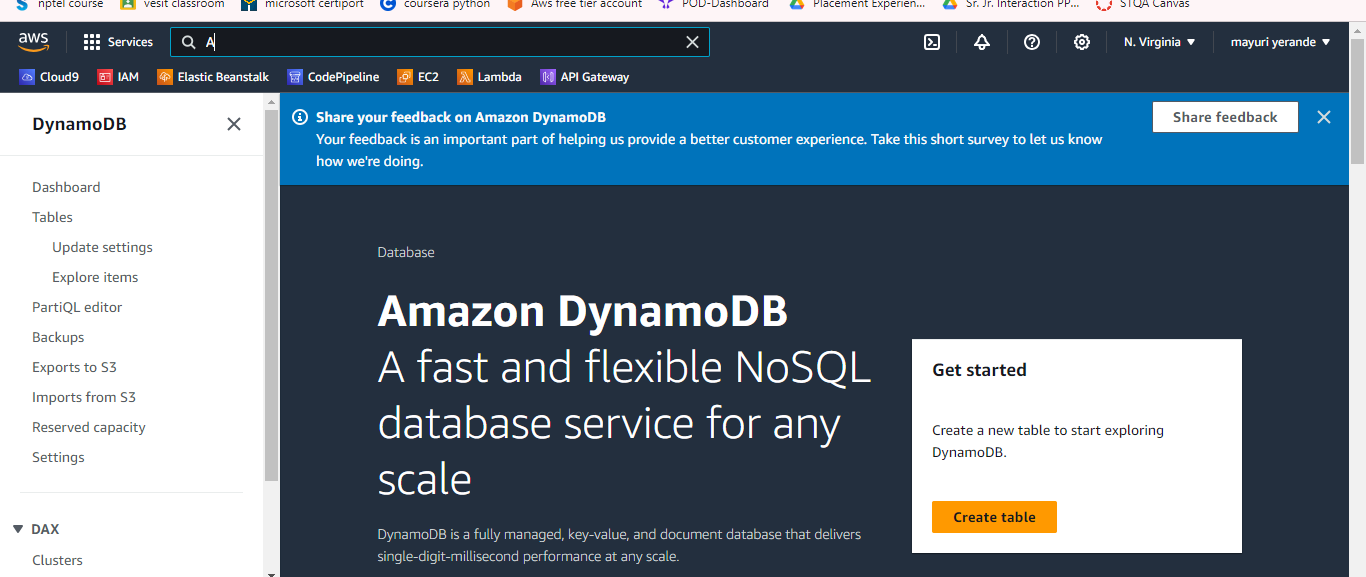
**Step 3:** On dashboard you can see both protocols has been implemented successfully

****

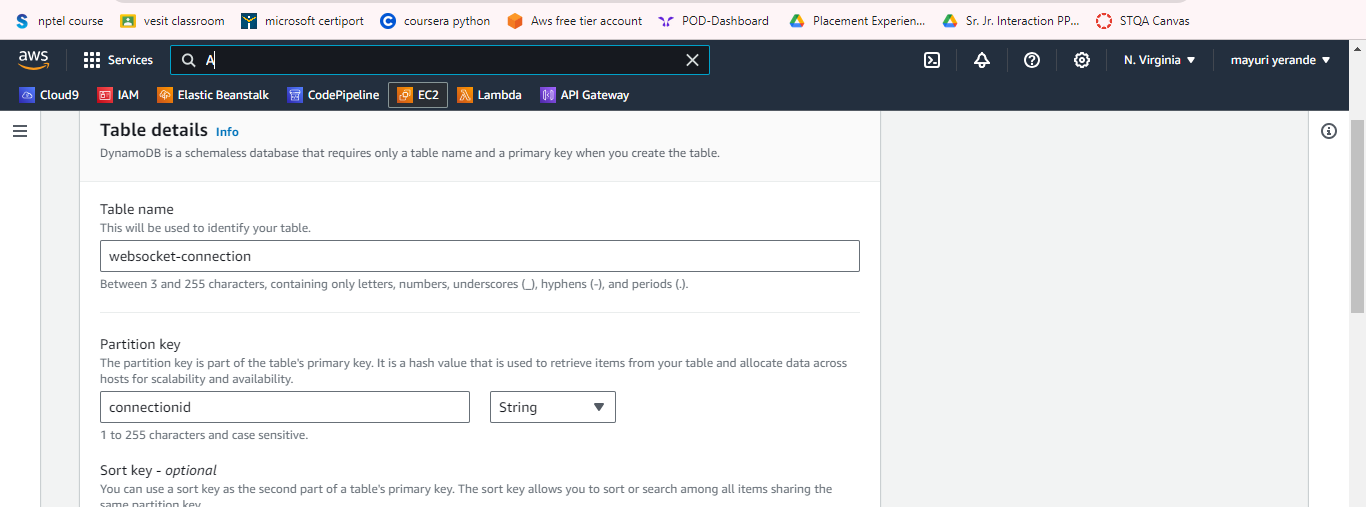
****

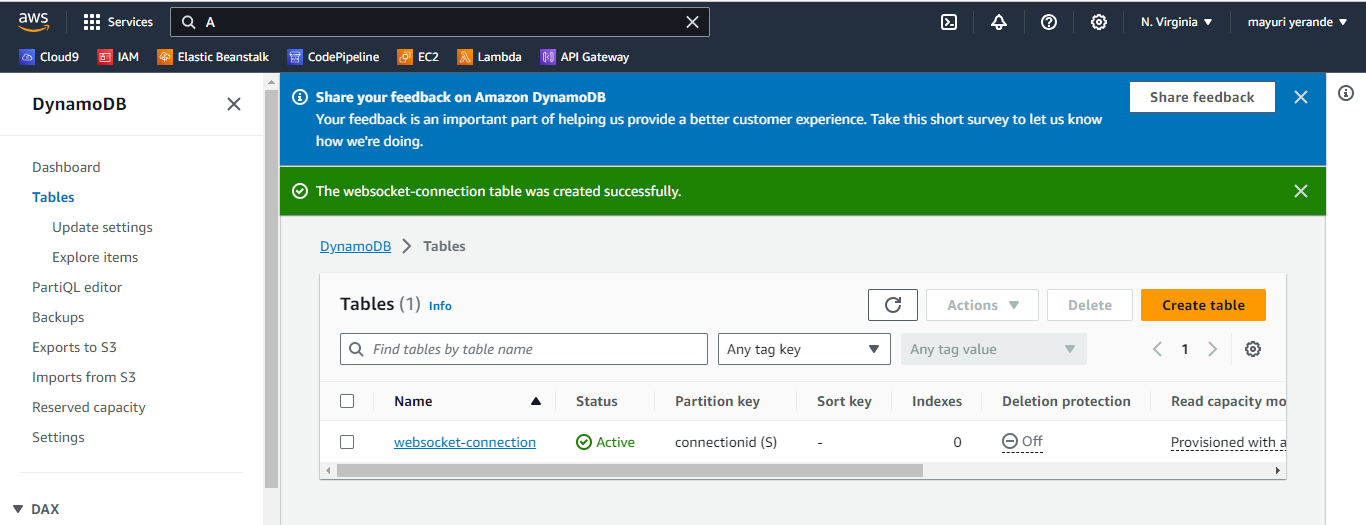
****

* **WEBSOCKET**
* Open DynamoDB

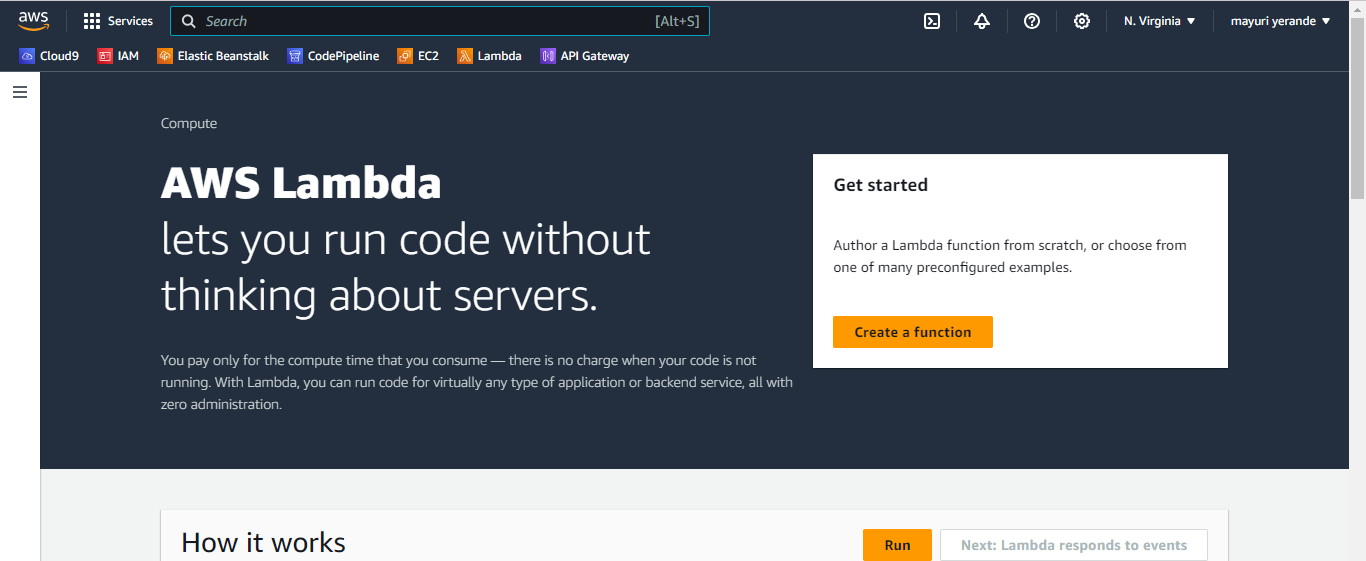
****

* Create DynamoDB Table

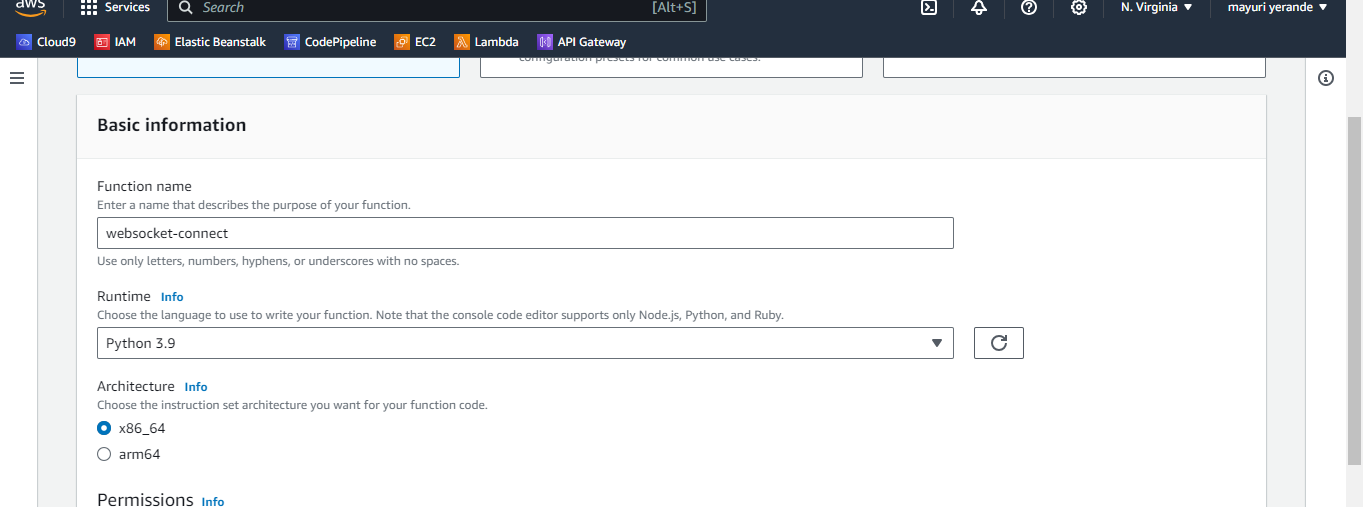
****

****

* Now we will be creating our lambda functions for connecting ,disconnecting and sending data for websocket

****

* Creation of Lambda function for connecting websocket

****

**Add this code in our function**

import json

import boto3

import os

dynamodb = boto3.client('dynamodb')

def lambda\_handler(event, context):

connectionId = event['requestContext']['connectionId']

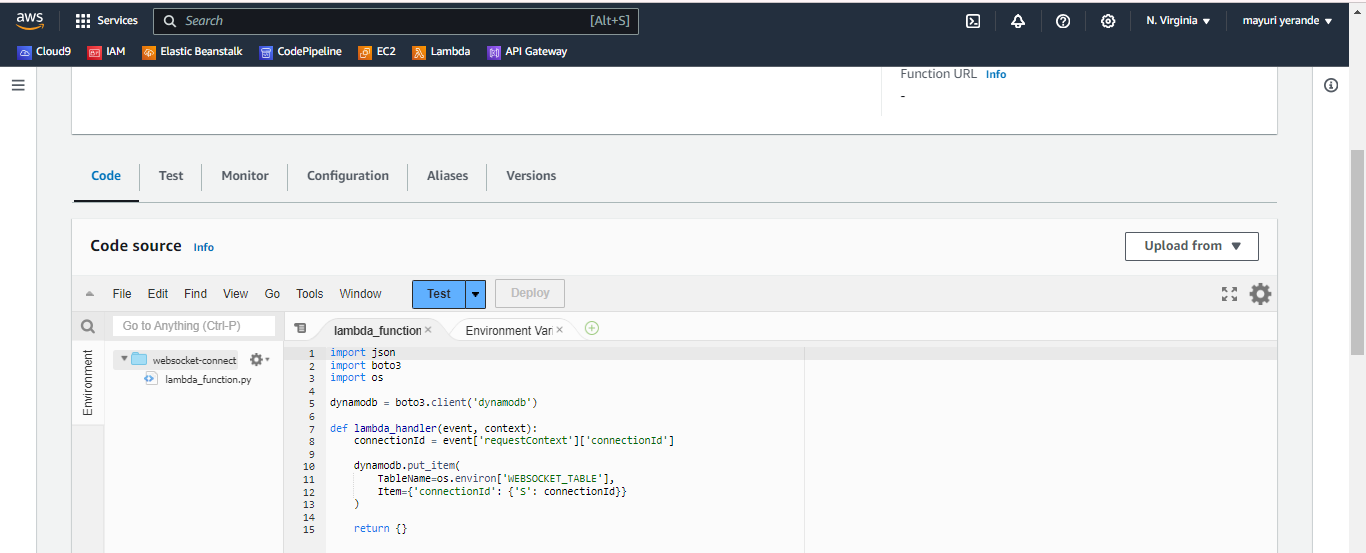
dynamodb.put\_item(

TableName=os.environ['WEBSOCKET\_TABLE'],

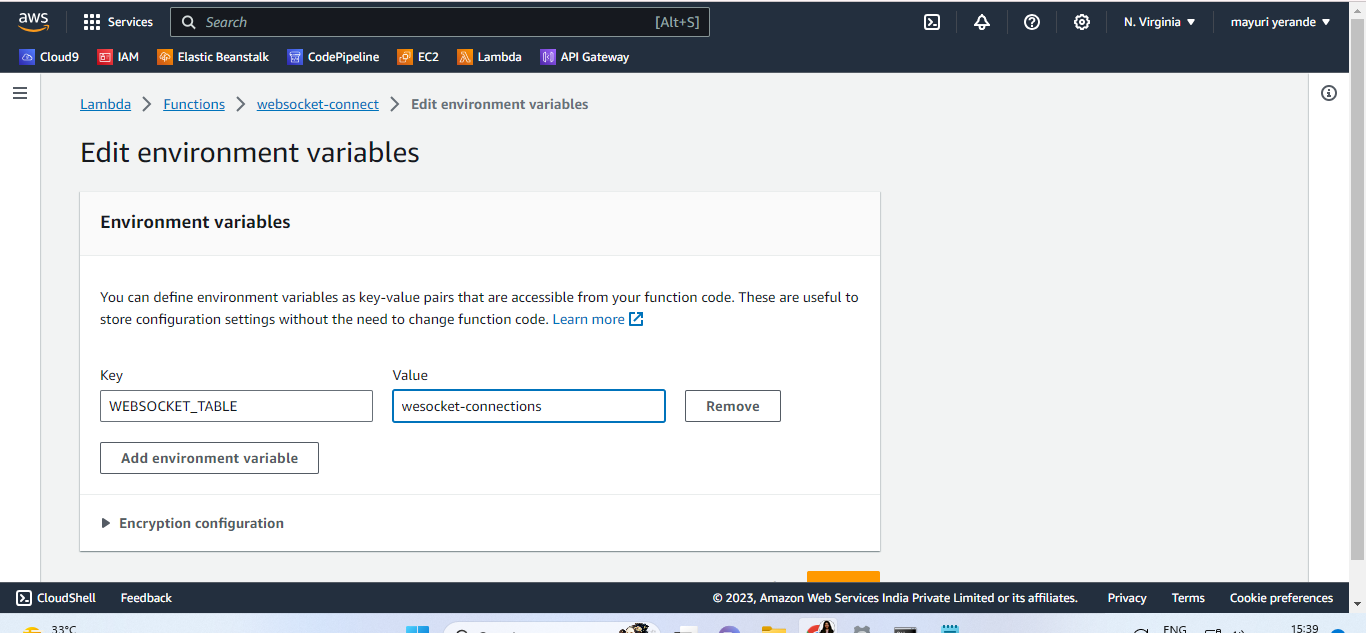
Item={'connectionId': {'S': connectionId}}

)

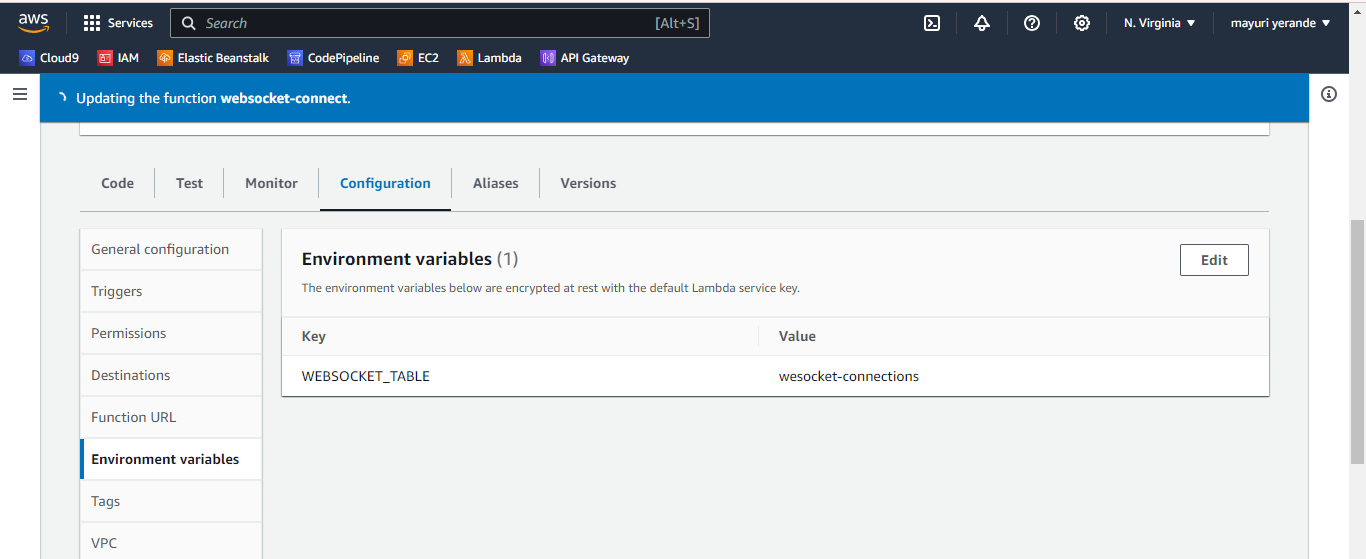
return {}

****

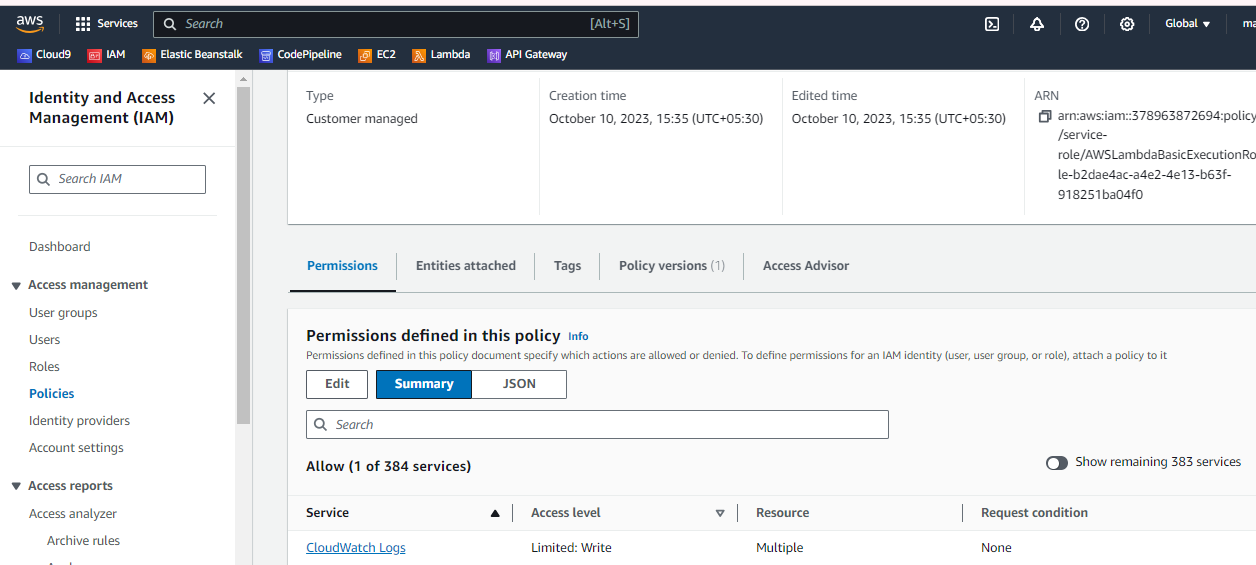
* Go to lambda function configuration and choose environment variables and add this variable to our table.

****

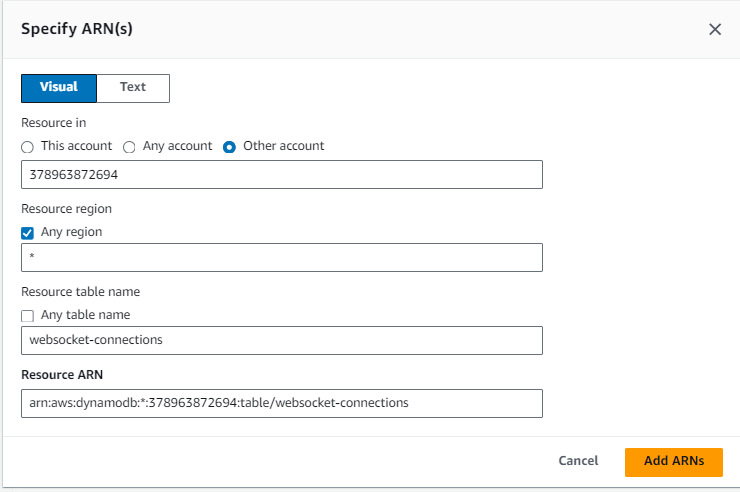
* Our environment variable is created

****

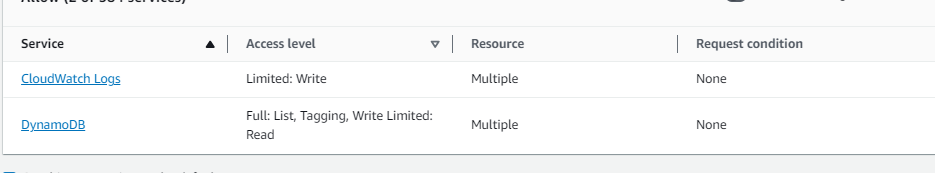
* Now we have to add certain permissions for connecting the websocket.
* For “websocket-connect” function, we will add the permission of “putitem” action

****

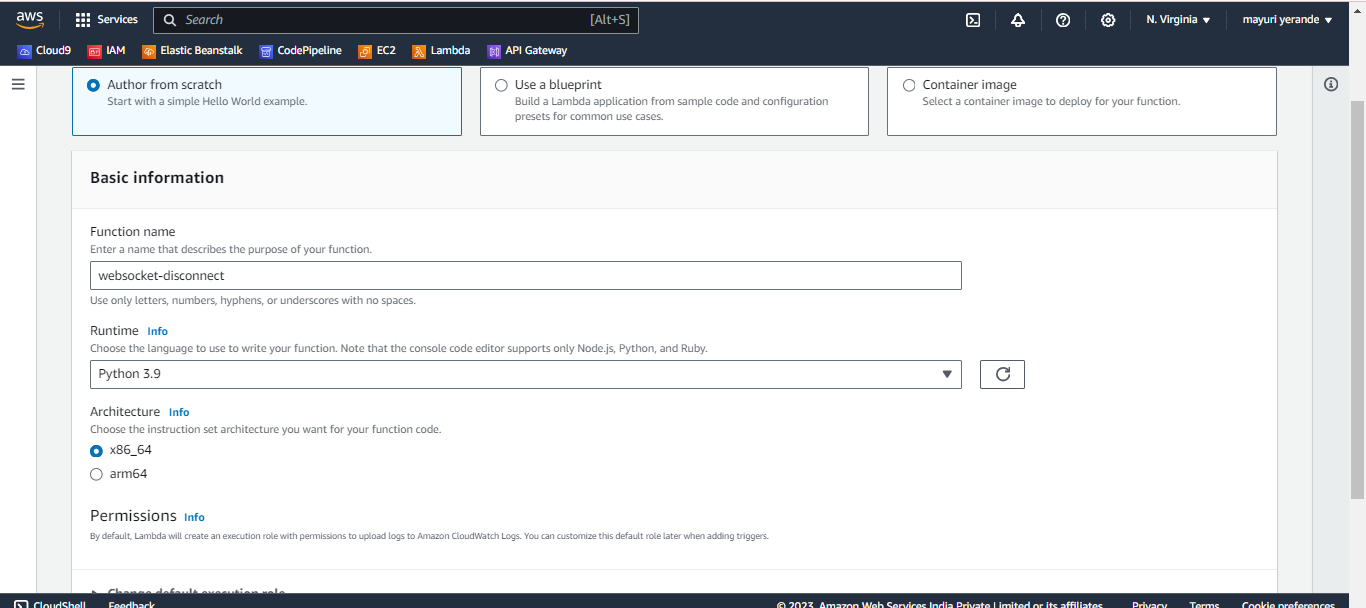
* Add the “arn” for our specific table. Add the table name in it

****

* Our permissions has been added

****

* Create new lambda function for disconnecting from websocket

****

* Add the code for disconnect in this function

import json

import boto3

import os

dynamodb = boto3.client('dynamodb')

def lambda\_handler(event, context):

connectionId = event['requestContext']['connectionId']

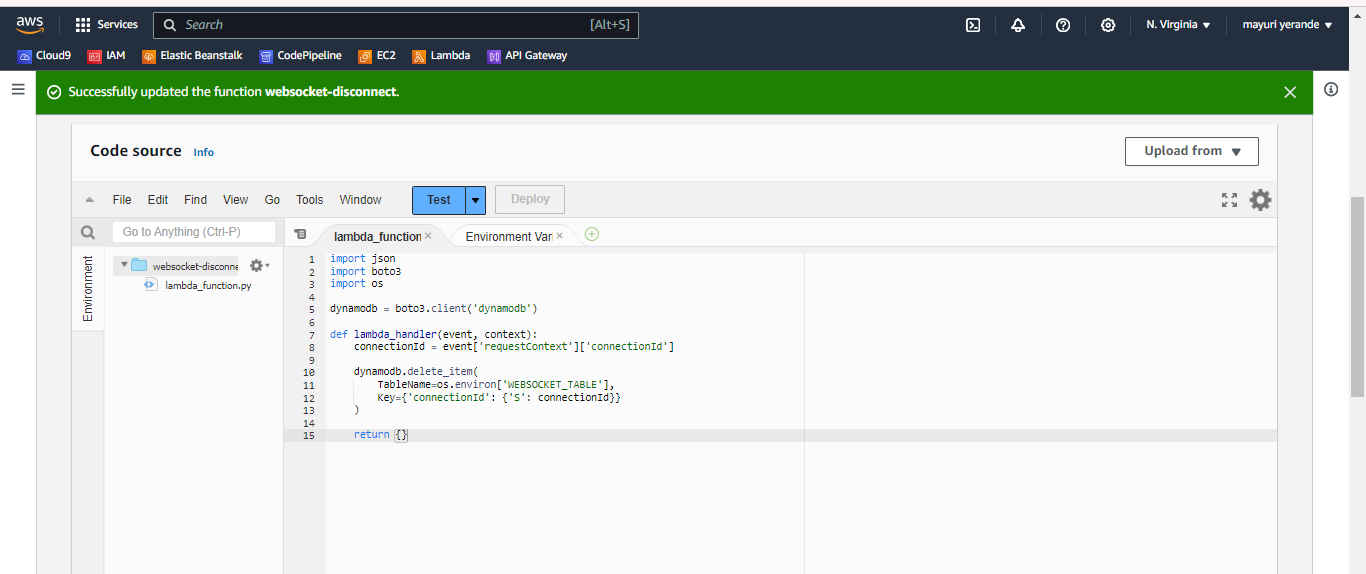
dynamodb.delete\_item(

TableName=os.environ['WEBSOCKET\_TABLE'],

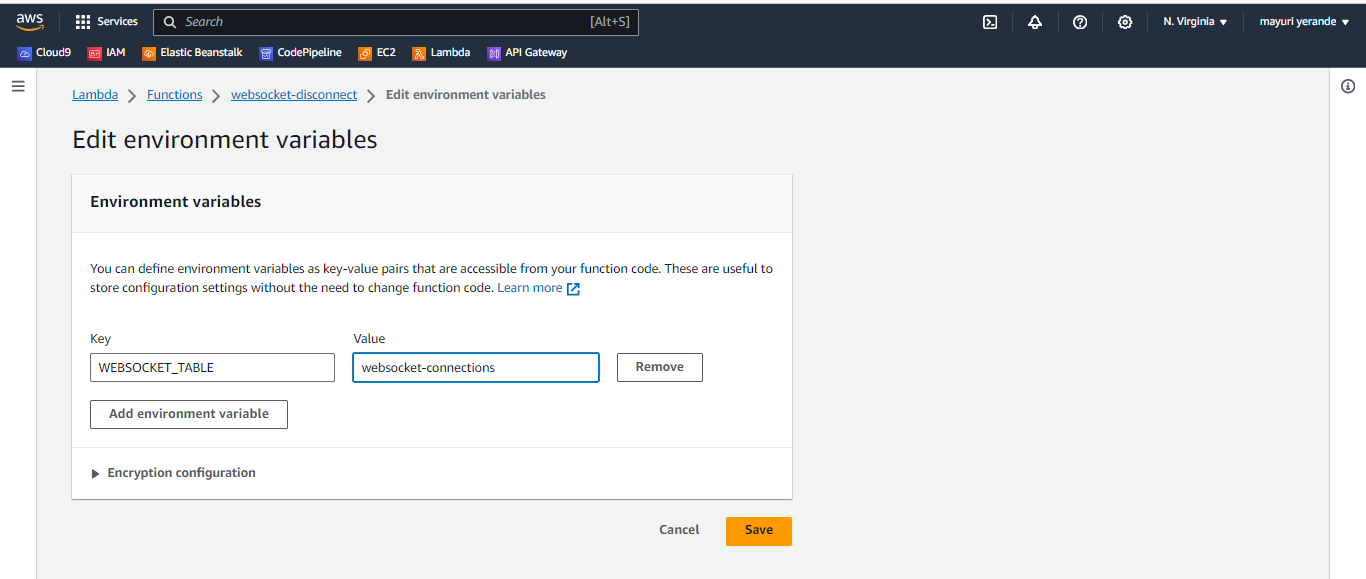
Key={'connectionId': {'S': connectionId}}

)

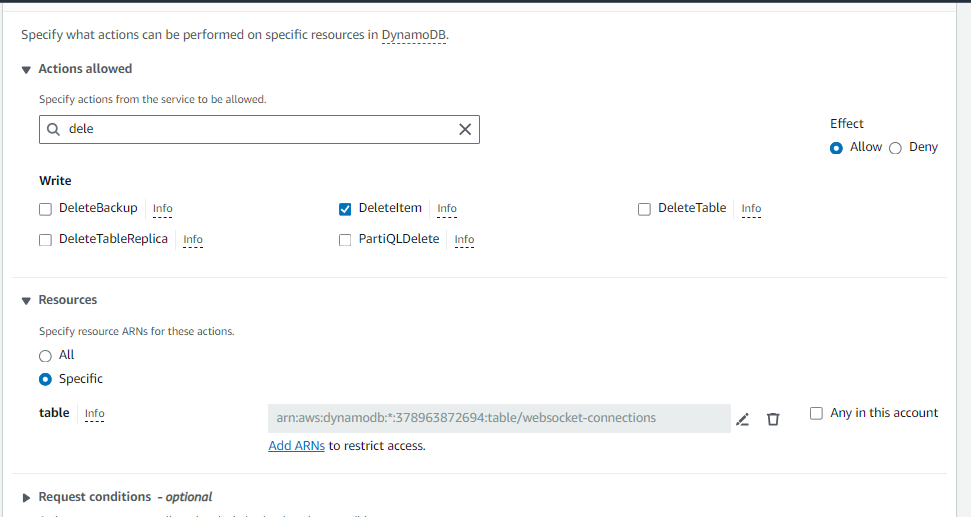
return {}

****

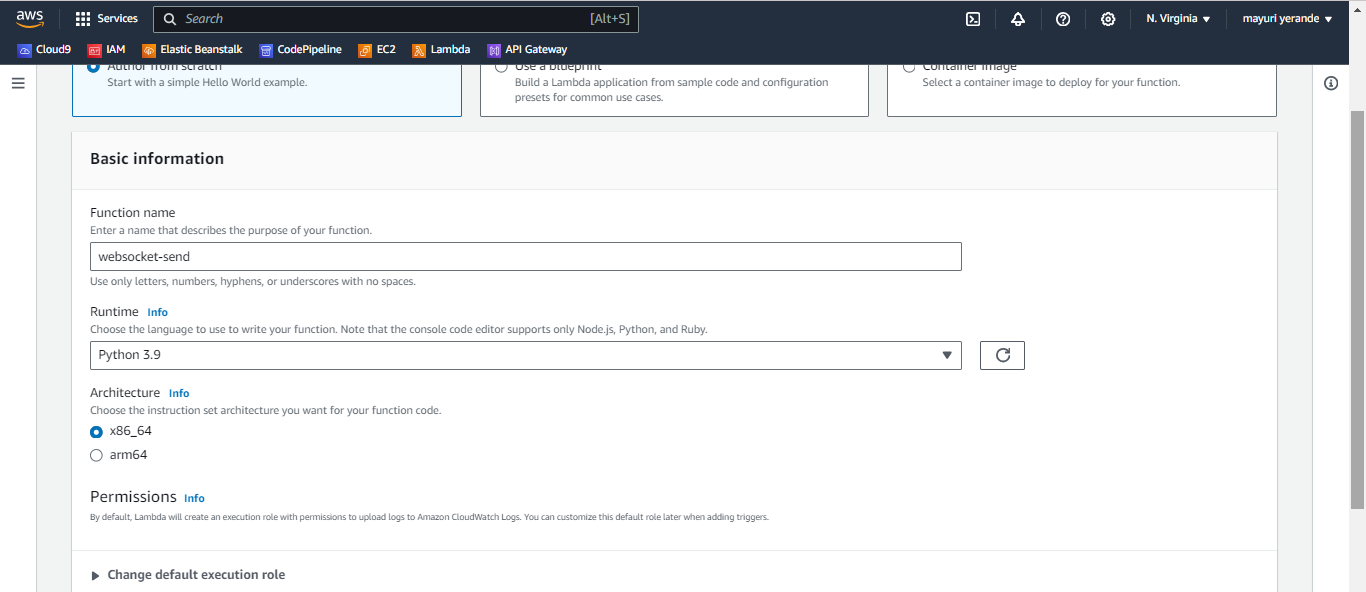
* Repeat same steps of variable creation and adding permissions
* Create new environment variable

****

* Add permission of “delete item” for dynamodb

****

* Now creating our final lambda function named “websocket-send”

****

* Add the code in it and deploy

import json

import boto3

import os

dynamodb = boto3.client('dynamodb')

def lambda\_handler(event, context):

message = json.loads(event['body'])['message']

paginator = dynamodb.get\_paginator('scan')

connectionIds = []

apigatewaymanagementapi = boto3.client(

'apigatewaymanagementapi',

endpoint\_url = "https://" + event["requestContext"]["domainName"] + "/" + event["requestContext"]["stage"]

)

for page in paginator.paginate(TableName=os.environ['WEBSOCKET\_TABLE']):

connectionIds.extend(page['Items'])

for connectionId in connectionIds:

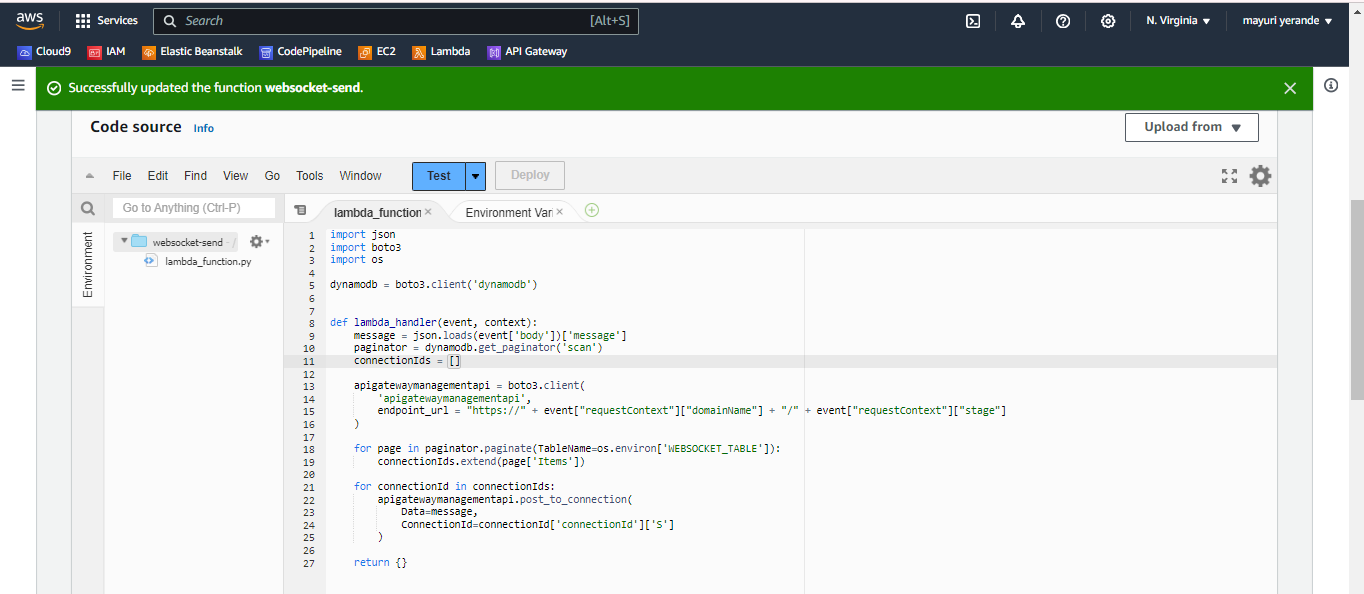
apigatewaymanagementapi.post\_to\_connection(

Data=message,

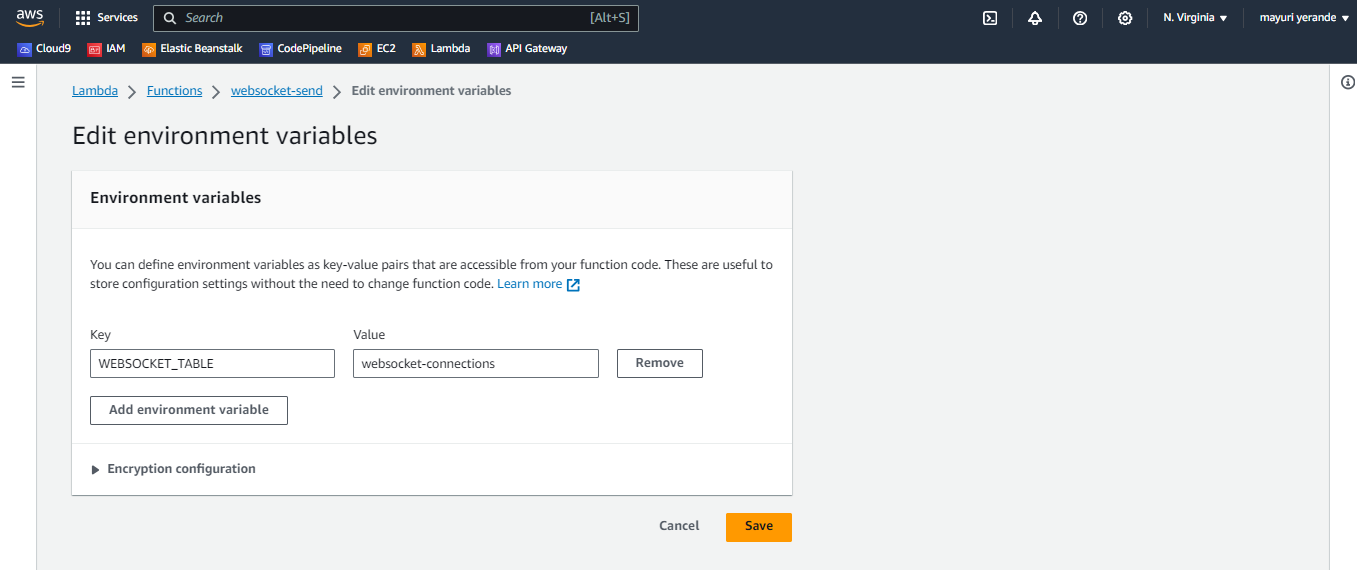
ConnectionId=connectionId['connectionId']['S']

)

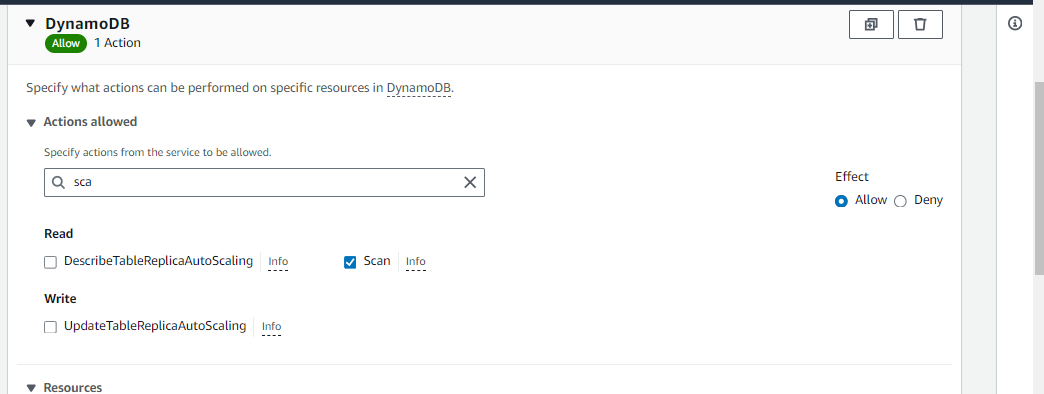
return {}

****

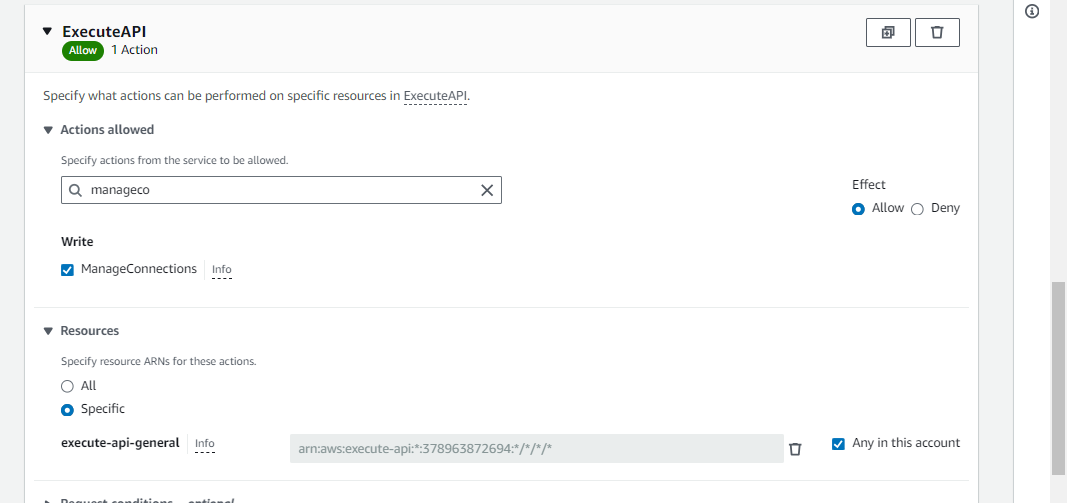
* Repeat the same thing again
* Set up the environment variable

****

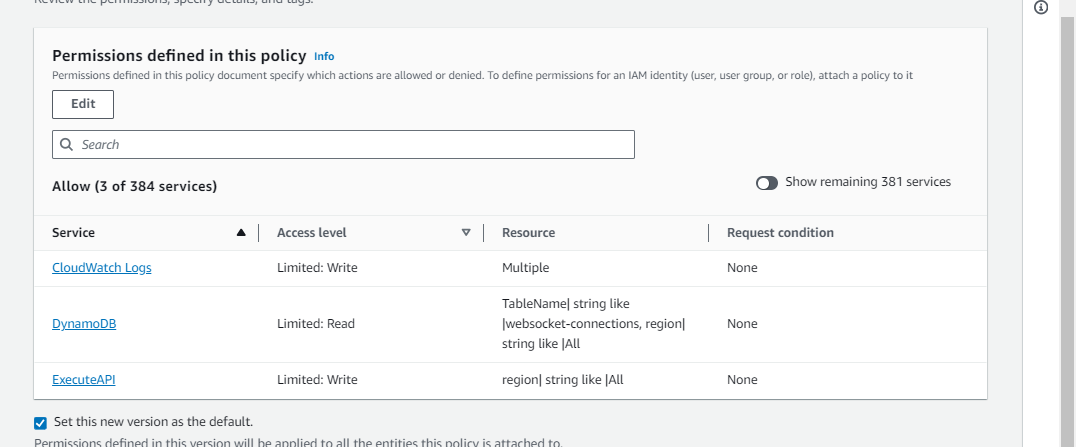
* Add permission here with action “scan”

****

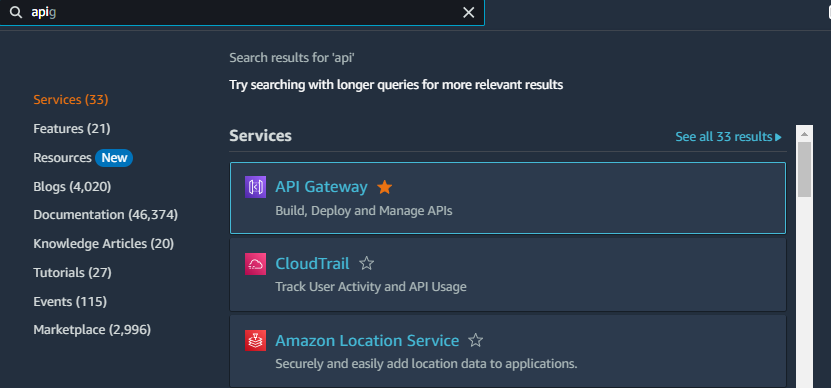
* And add another permission with “executeApi” action

****

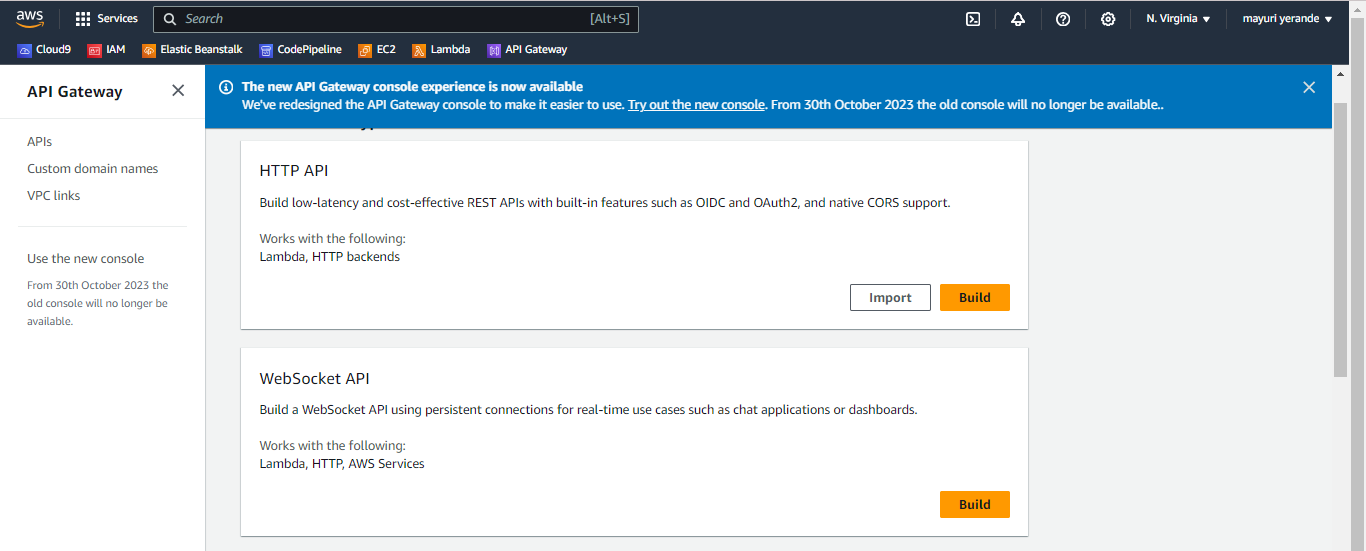
* Save the permissions

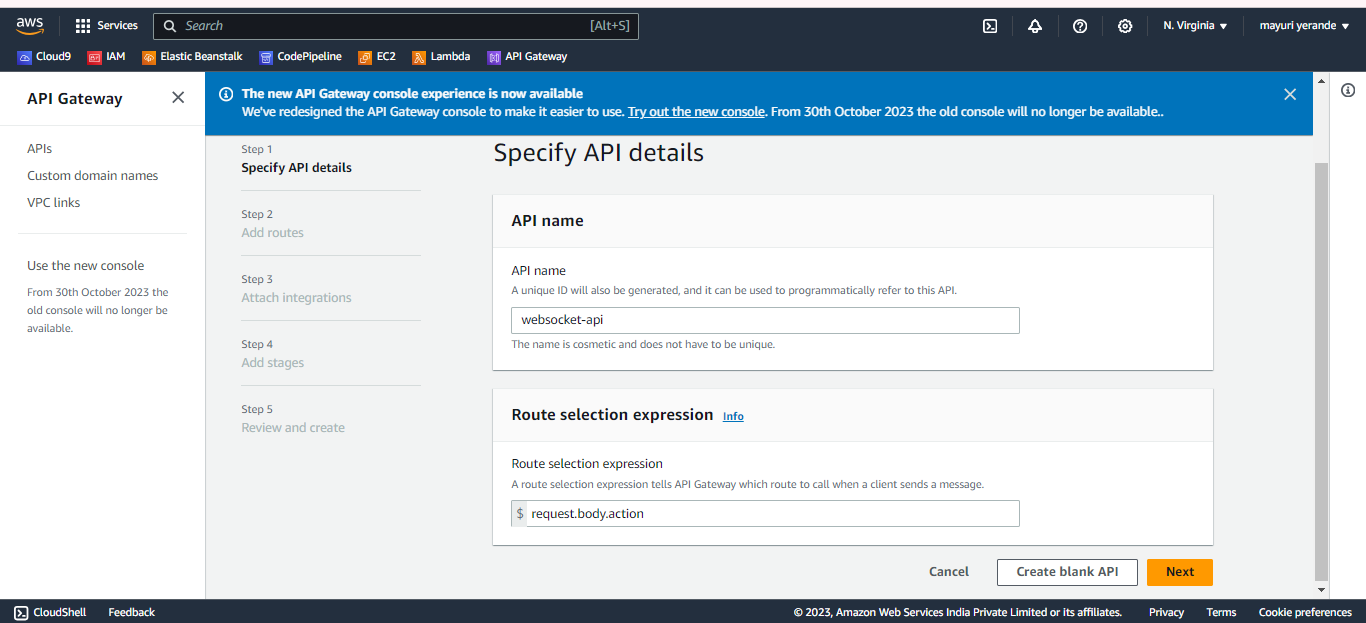
****

* Now we will create the API gateway

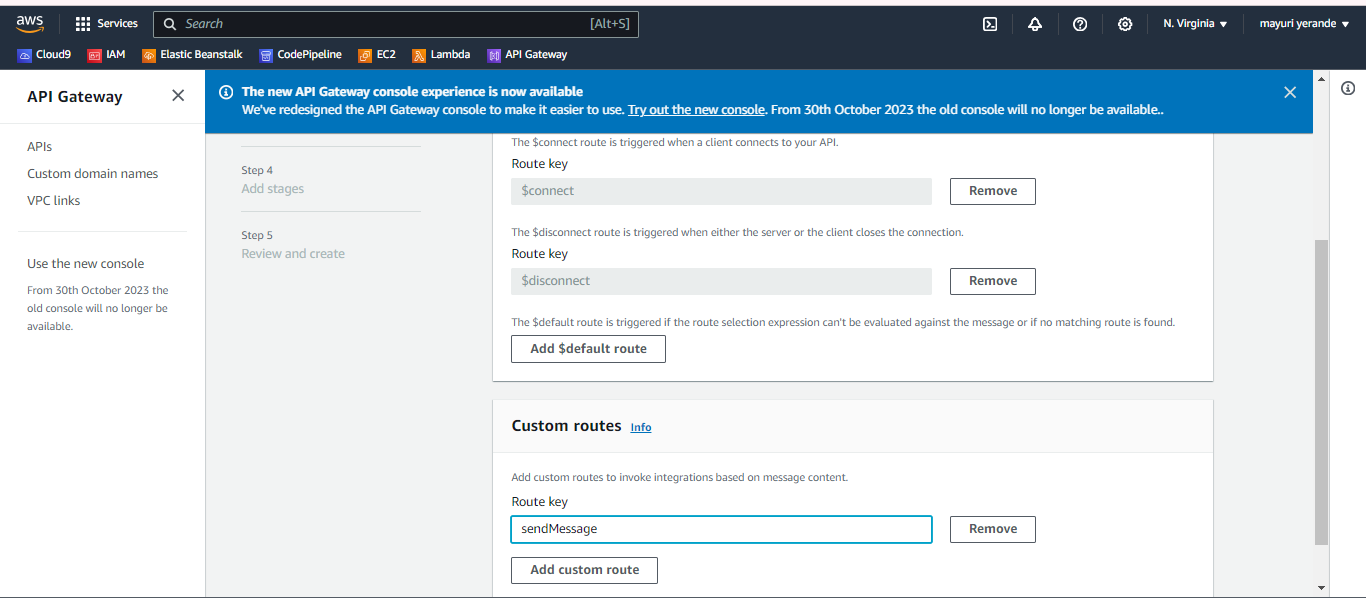
****

* Choose the “websocket api”
* Click on build

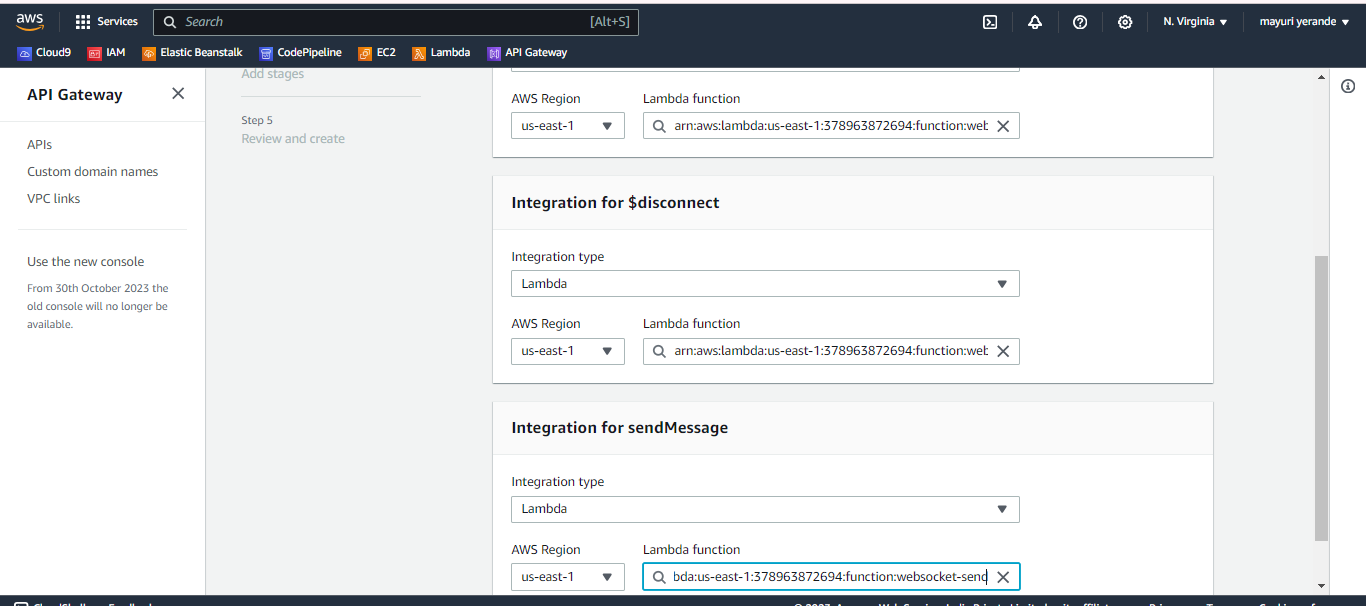
****

****

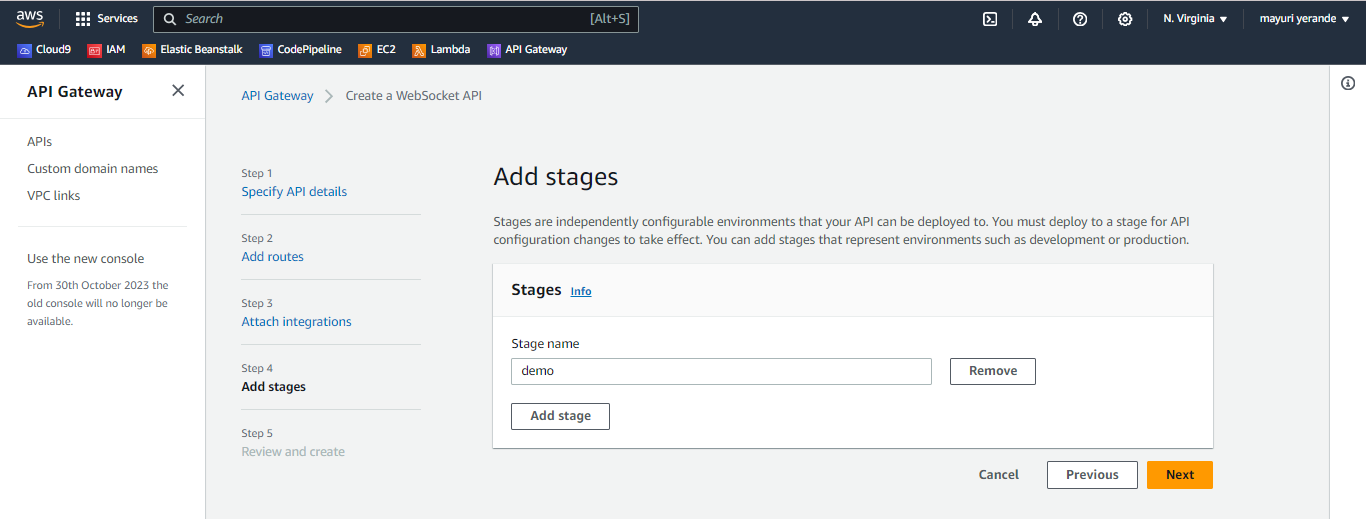
* Now we will add “sendMessage” route which will be custom route
* Click on next

****

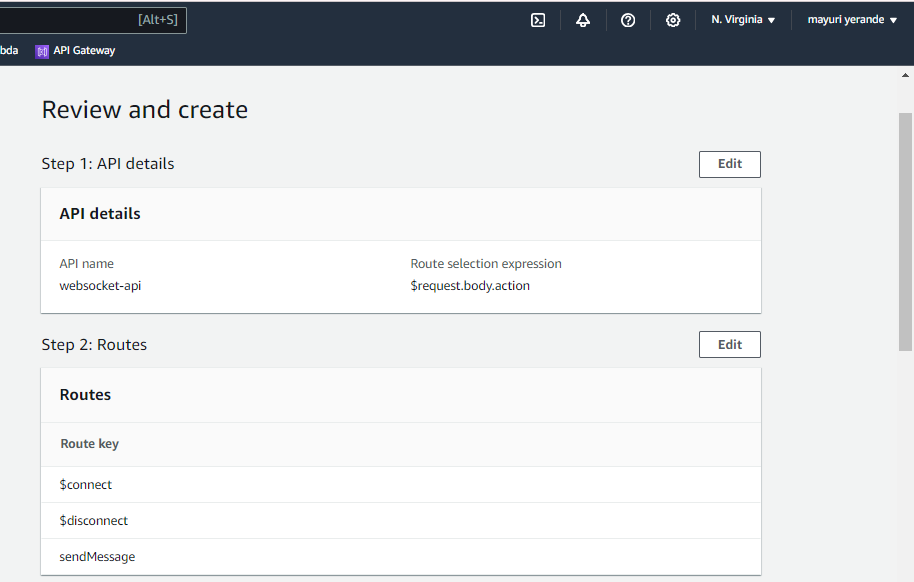
* Now add all the lambda functions here to integrate

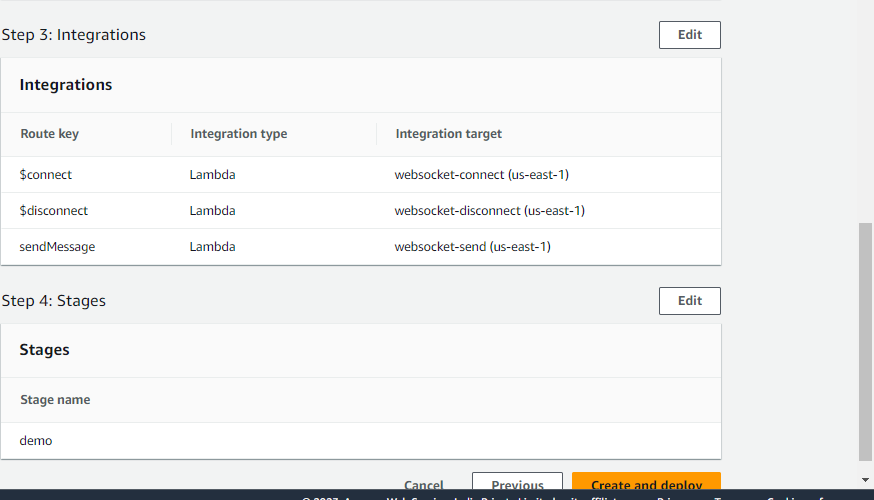
****

* Add the stage name(We named it “demo”)

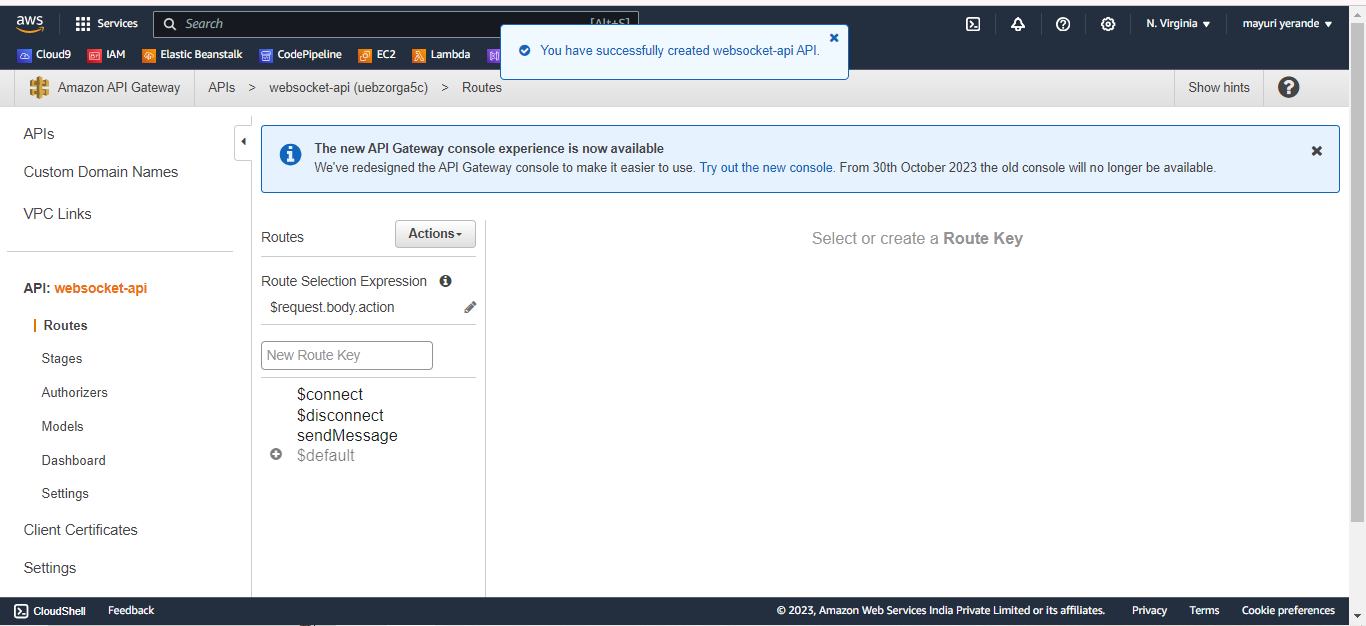
****

* Finally review all the details

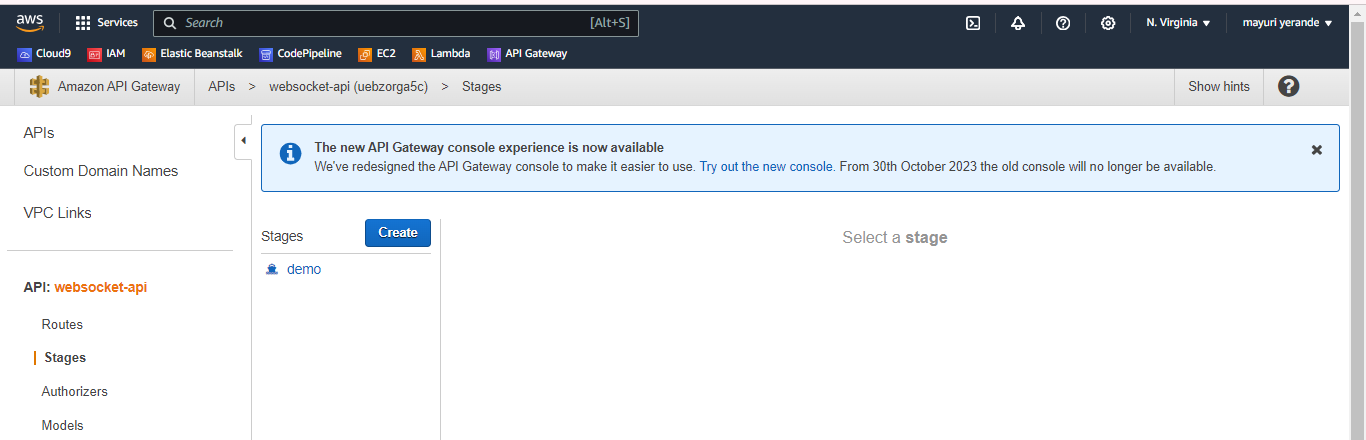
****

****

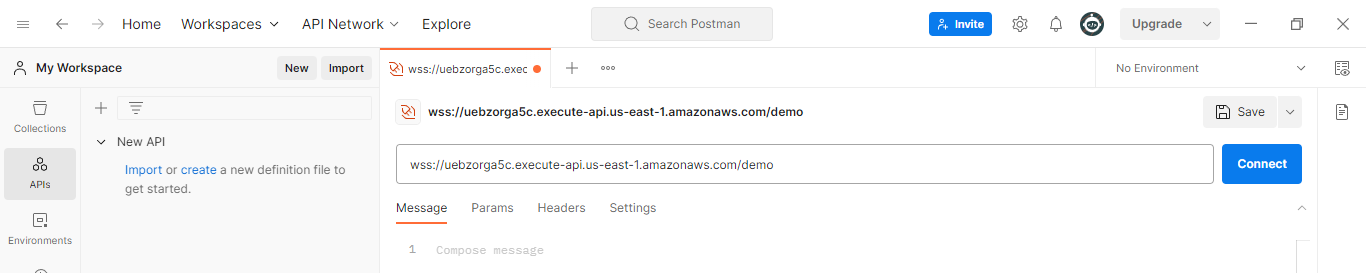
* Our API Gateway is ready

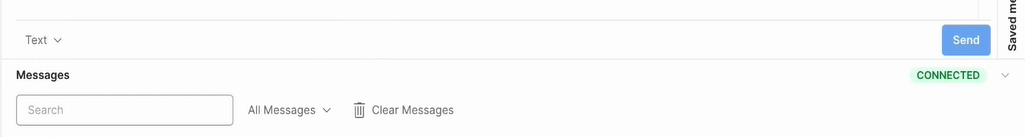
****

* Now go to stages section, Your stage should be visible here

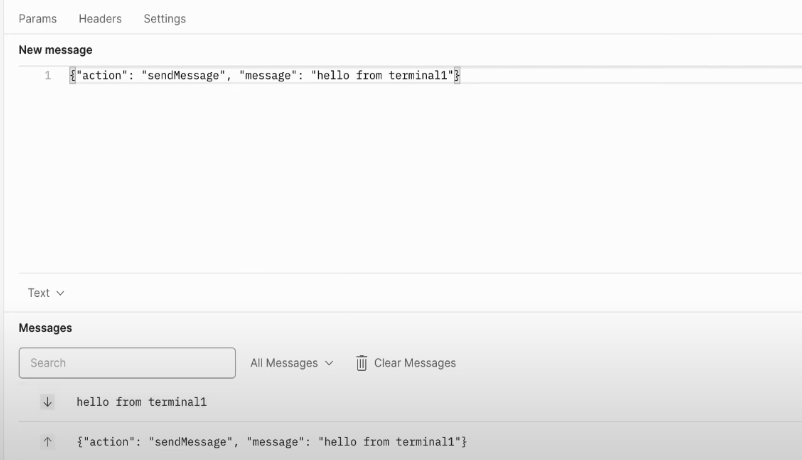
****

* Now we will be using postman api here
* Download postman: <https://www.postman.com/downloads/>
* Sign in with your account
* Go to “API” section
* Then choose “websocket” in postman
* Enter the websocket url in postman
* Click on “connect” and you will see that it gets connected.

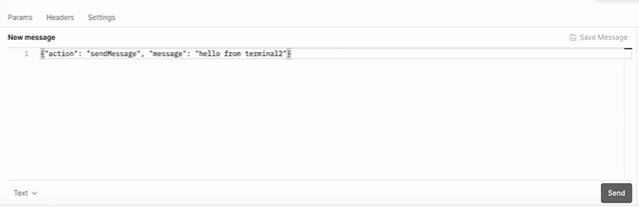
****

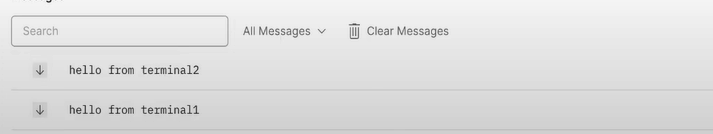
****

* Now we will send a message. Write the following command and you will see your message getting displayed on the terminal

****

* We can add a new message

****

****

**CONCLUSION:**

Thus we successfully implemented MQTT, HTTPS and WEBSOCKET protocols in AWS IOT. Also we implemented a publisher subscriber model in AWS IOT.