A Fully Serverless Web Application on AWS Using Amplify, API Gateway, Lambda, and DynamoDB

Submitted by:
Purushothaman S (G24AI1042)
Swatantra Yadav (G24AI1065)
Nalin Bhatt (G24AI1070)

Submitted to:
Dr. Sumit Karla (Assistant Professor)
School of Artificial Intelligence and Data Engineering. *IIT-JODHPUR*

Institute: **Indian Institute of Technology – Jodhpur**
Date: **15th April 2025.**

Index

Abstract

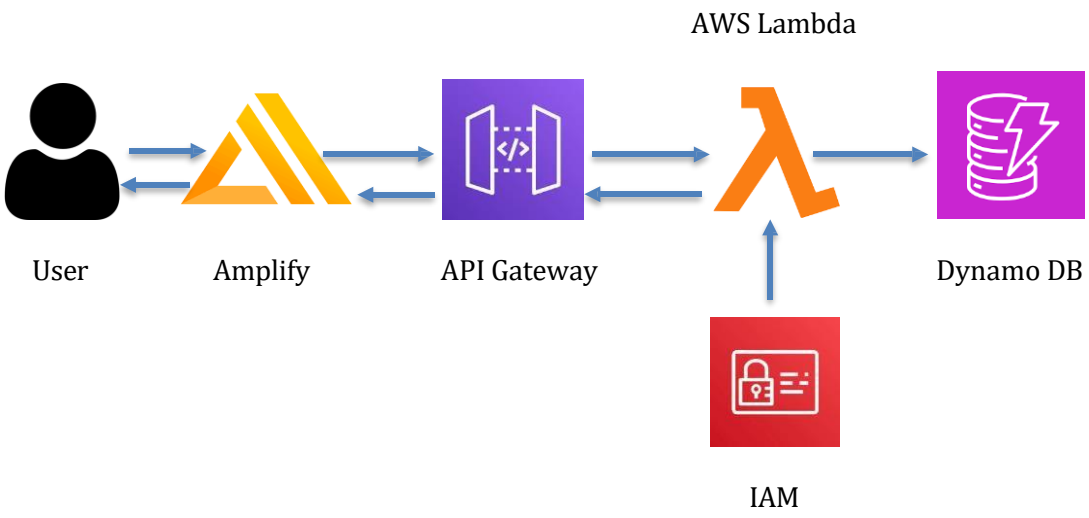
This project demonstrates the design and implementation of a fully serverless web application on Amazon Web Services (AWS), built to perform exponential calculations through a browser-based interface. The application stack includes AWS Amplify for static site hosting, HTML and JavaScript for the client interface, AWS API Gateway for RESTful communication, AWS Lambda for backend logic processing, and AWS DynamoDB for persistent data storage. A streamlined architecture ensures high scalability, zero server management, and minimal operational overhead. With secure IAM policies and automated deployment, this application exemplifies modern cloud-native development practices, providing a practical solution for lightweight, real-time computational tasks.

Introduction

This document presents a comprehensive overview of a fully serverless web application built using Amazon Web Services (AWS). The goal of this project is to demonstrate how different AWS services can be integrated to construct a scalable and efficient application. The solution leverages AWS Amplify for front-end hosting, HTML and JavaScript for the user interface, AWS API Gateway for RESTful communication, AWS Lambda for backend logic, and AWS DynamoDB for persistent data storage.

Architecting and Building an End-to-End AWS Web Application and System Architecture

The web application is architected to be entirely serverless, enabling high availability, minimal maintenance, and cost-efficiency. Users interact with a web interface hosted via AWS Amplify. Their inputs are sent to an API Gateway, which triggers AWS Lambda functions to process the data. The results of these computations are then stored in a DynamoDB table.



Front-End Development and Hosting

The front-end was developed using basic HTML, CSS, and JavaScript. It consists of a simple form that accepts a base and an exponent from the user. These values are used to calculate a power expression. Once the user clicks the 'CALCULATE' button, the application triggers a JavaScript function that sends a POST request with the user input to an API Gateway endpoint. The HTML and JavaScript files were deployed to AWS Amplify, which provides a secure and scalable hosting environment. Amplify also automatically provisions a globally accessible HTTPS URL and supports CI/CD pipelines for ongoing updates

Scripting:

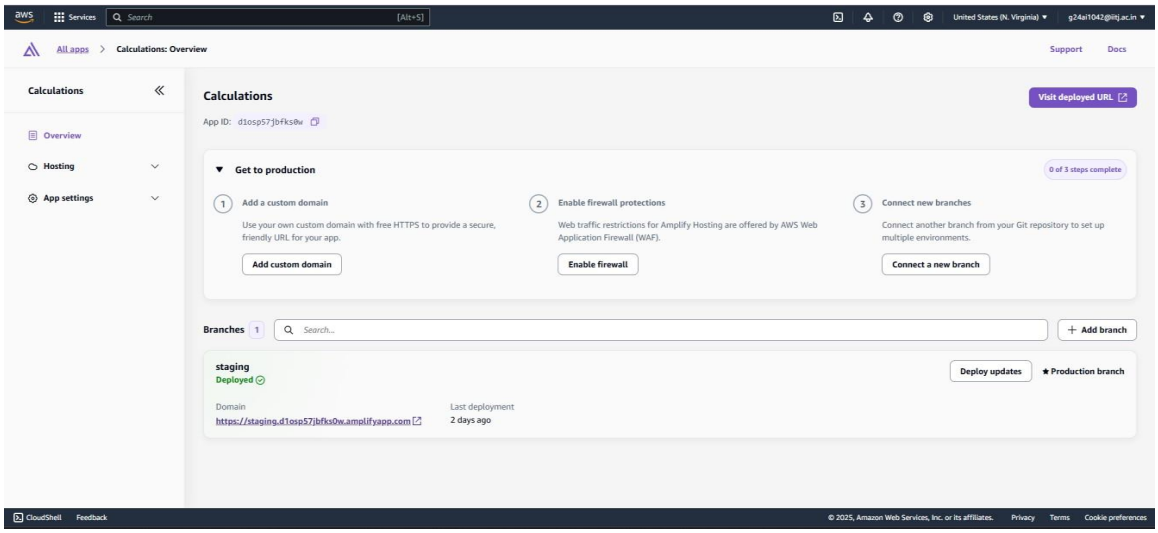
```
<!DOCTYPE html>
<html>
<head>
  <meta charset="UTF-8">
  <title>To the Power of Math!</title>
  <!-- Styling for the client UI -->
  <style>
  h1 {
    color: #FFFFFF;
    font-family: system-ui;
    margin-left: 20px;
    }
  body {
    background-color:  #222629;
    }
  label {
    color: #86C232;
    font-family: system-ui;
    font-size: 20px;
    margin-left: 20px;
```

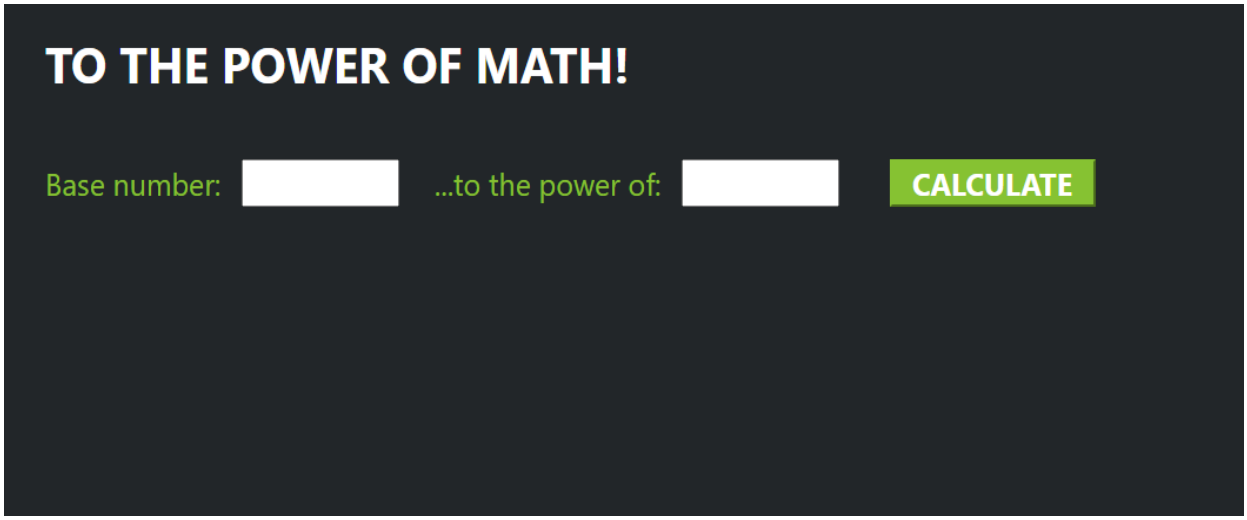
```
  margin-top: 20px;
  }
 button {
  background-color: #86C232;
  border-color: #86C232;
  color: #FFFFFF;
  font-family: system-ui;
  font-size: 20px;
  font-weight:  bold;
  margin-left: 30px;
  margin-top: 20px;
  width: 140px;
  }
 input {
  color: #222629;
  font-family: system-ui;
  font-size: 20px;
  margin-left: 10px;
  margin-top: 20px;
  width: 100px;
  }
</style>
<script>
    var callAPI = (base, exponent) => {
    var myHeaders = new Headers();
    myHeaders.append("Content-Type",  "application/json");
```

```
        var raw = JSON.stringify({ "base": base, "exponent": exponent });


        var requestOptions = {
          method: 'POST',
          headers: myHeaders,
          body: raw,
          redirect: 'follow'
        };


        fetch("https://gdz81dlc8j.execute-api.us-east-1.amazonaws.com/dev",
requestOptions)
          .then(response => {
            console.log("Raw response:", response);
            return response.text();
          })
          .then(result => {
            console.log("Result body:", result);
            alert(JSON.parse(result).body);
          })
          .catch(error => {
            console.error("Error:", error);
            alert(error);
          });
        }
```

```
  </script>
</head>
<body>
  <h1>The Power of Given Number </h1>
  <form>
    <label>Base number:</label>
    <input type="text" id="base">
    <label>...to the power of:</label>
    <input type="text" id="exponent">
    <!-- set button onClick method to call function we defined passing input values as parameters -->
    <button type="button" onclick="callAPI(document.getElementById('base').value,document.getElementById('exponent').value)">CALCULATE</button>
  </form>
</body>
</html>
```

Deployment and Sample Output for Front End:

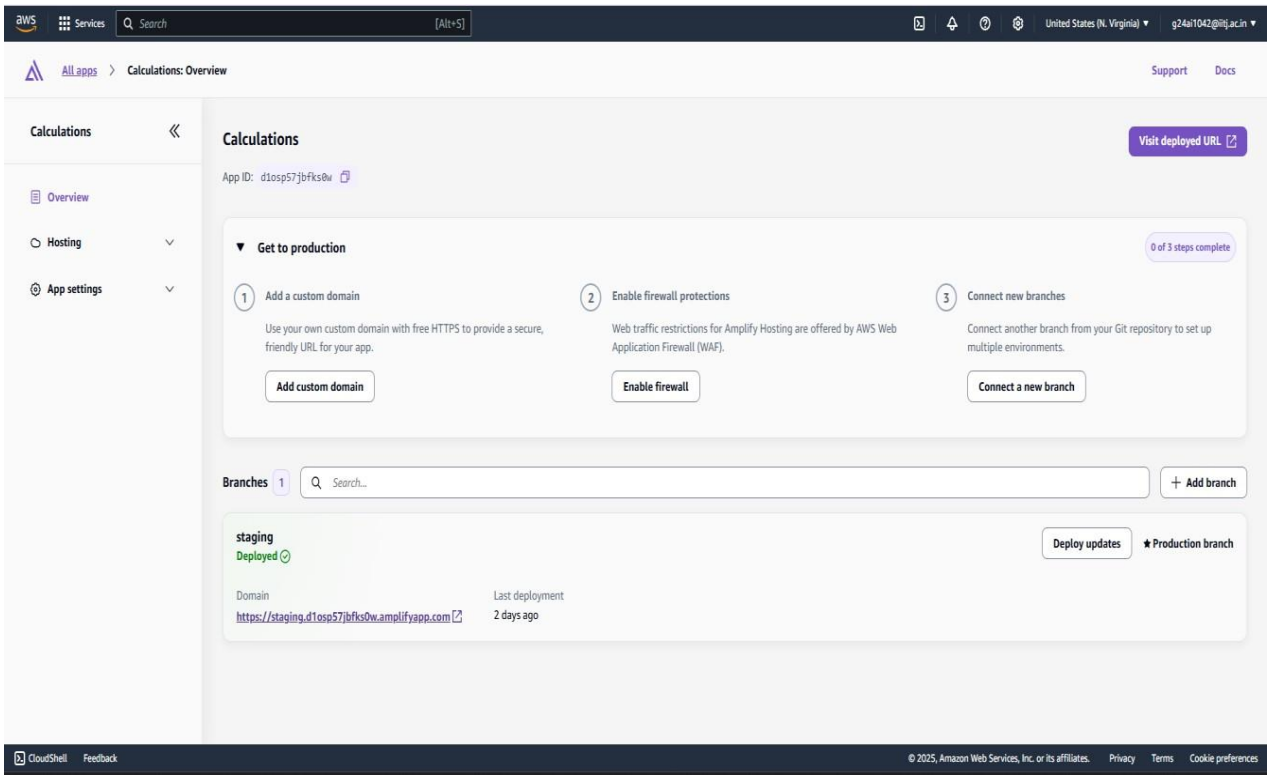




API Gateway Configuration

To bridge communication between the front-end and the back-end, a REST API was created using AWS API Gateway. The API defines a single route (POST /) which is used to handle user requests. Cross-Origin Resource Sharing (CORS) was enabled to permit requests from the Amplify-hosted domain. API Gateway acts as the traffic controller, forwarding valid requests to the associated Lambda function for processing.

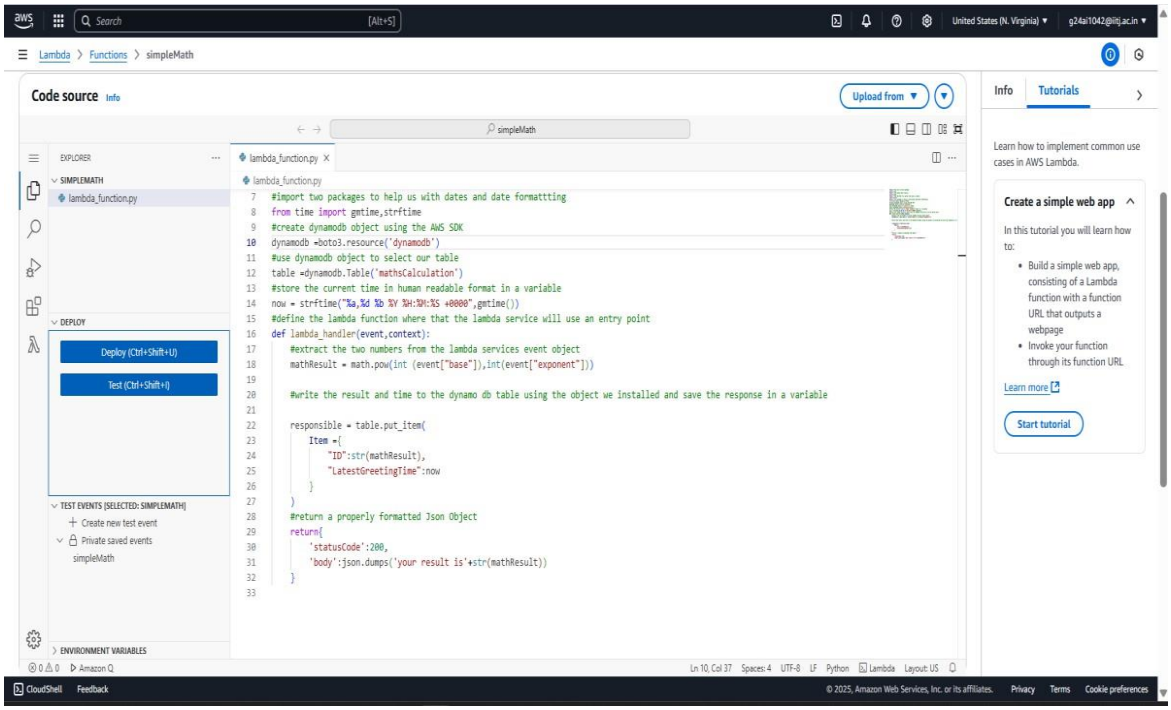
Deployment



Serverless Logic with AWS Lambda

The core computational logic is written in Python and deployed as an AWS Lambda function. Upon receiving a request from API Gateway, the Lambda function extracts the 'base' and 'exponent' values from the JSON payload and uses Python's math.pow() function to compute the result. This result, along with a timestamp generated using the time module, is then inserted into a DynamoDB table named 'mathsCalculation'. The function concludes by returning a JSON response containing the result of the calculation.

Deployment



Lambda Handler Function

#import the json utility package

import json

#import the python math library

import math

#import the AWS SDK (for python the name is boto3)

import boto3

#import two packages to help us with dates and date formattting

from time import gmtime,strftime

```
#create dynamodb object using the AWS SDK
dynamodb =boto3.resource('dynamodb')
#use dynamodb object to select our table
table =dynamodb.Table('mathsCalculation')
#store the current time in human readable format in a variable
now = strftime("%a,%d %b %Y %H:%M:%S +0000",gmtime())
#define the lambda function where that the lambda service will use an entry point
def lambda_handler(event,context):
  #extract the two numbers from the lambda services event object
  mathResult = math.pow(int (event["base"]),int(event["exponent"]))

  #write the result and time to the dynamo db table using the object we installed
#and save the response in a variable
  responsible = table.put_item(
    Item ={
      "ID":str(mathResult),
      "LatestGreetingTime":now
    }
  )
  #return a properly formatted Json Object
  return{
    'statusCode':200,
    'body':json.dumps('your result is'+str(mathResult))
  }
```

Data Persistence with DynamoDB

DynamoDB is used to store the results of each computation. The table 'mathsCalculation' has two key fields: 'ID', which stores the calculated result as a string, and 'LatestGreetingTime', which stores the timestamp of the computation. AWS's boto3 SDK is used within the Lambda function to interact with the database. DynamoDB's auto-scaling capabilities ensure the table can handle varying loads without performance degradation.

Deployment

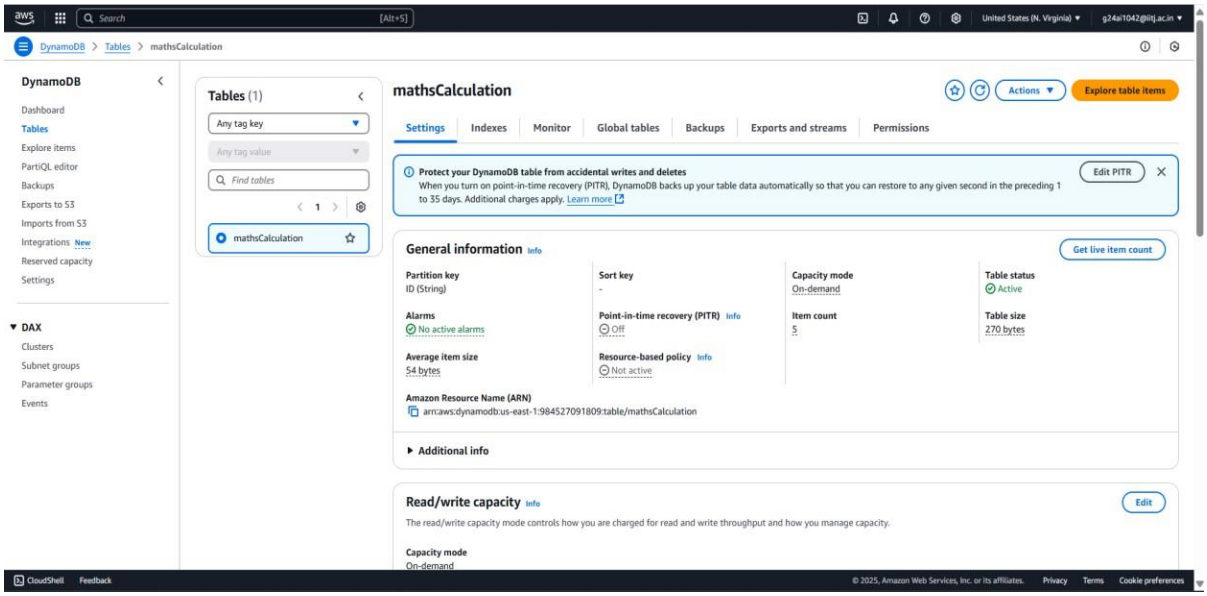
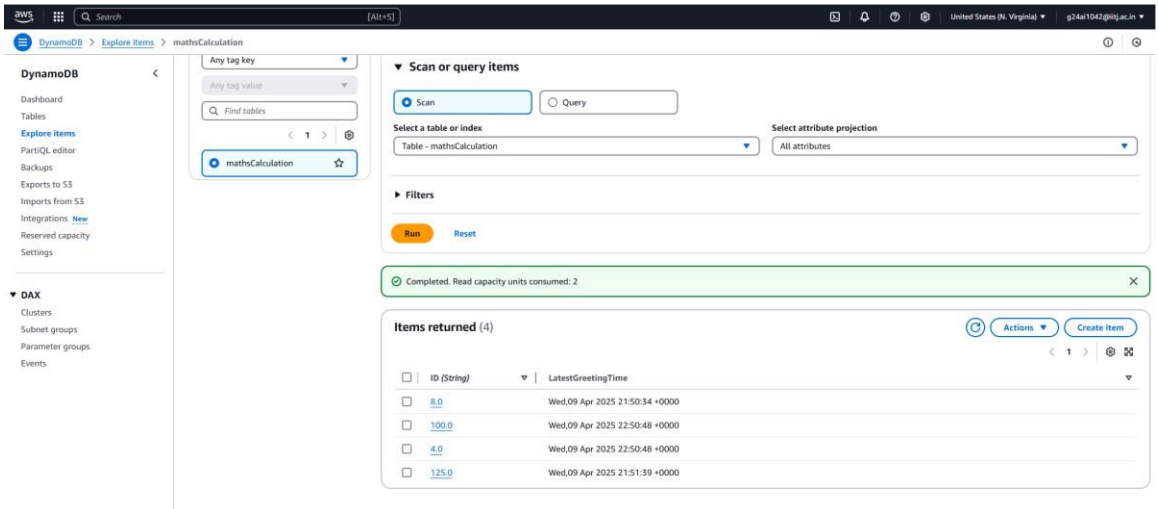


Table info after the execution



IAM Policy for Secure Access

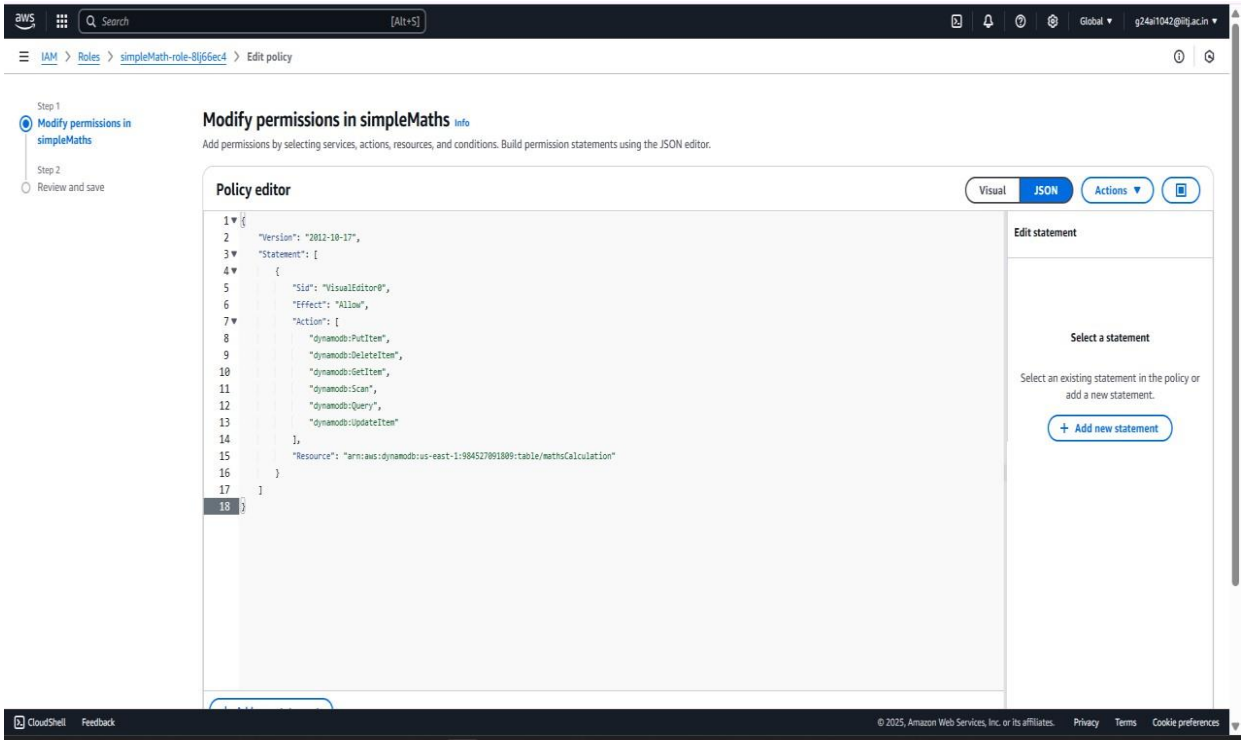
To facilitate secure interaction between the Lambda function and DynamoDB, an IAM policy was defined and attached to the Lambda execution role. This policy grants permissions for common database operations such as PutItem, GetItem, UpdateItem, DeleteItem, Scan, and Query. The scope of access is limited to the specific DynamoDB table using the following **resource ARN:** arn:aws:dynamodb:us-east-1:984527091809:table/mathsCalculation

The policy ensures the application adheres to the principle of least privilege while enabling full functionality.

JSON :

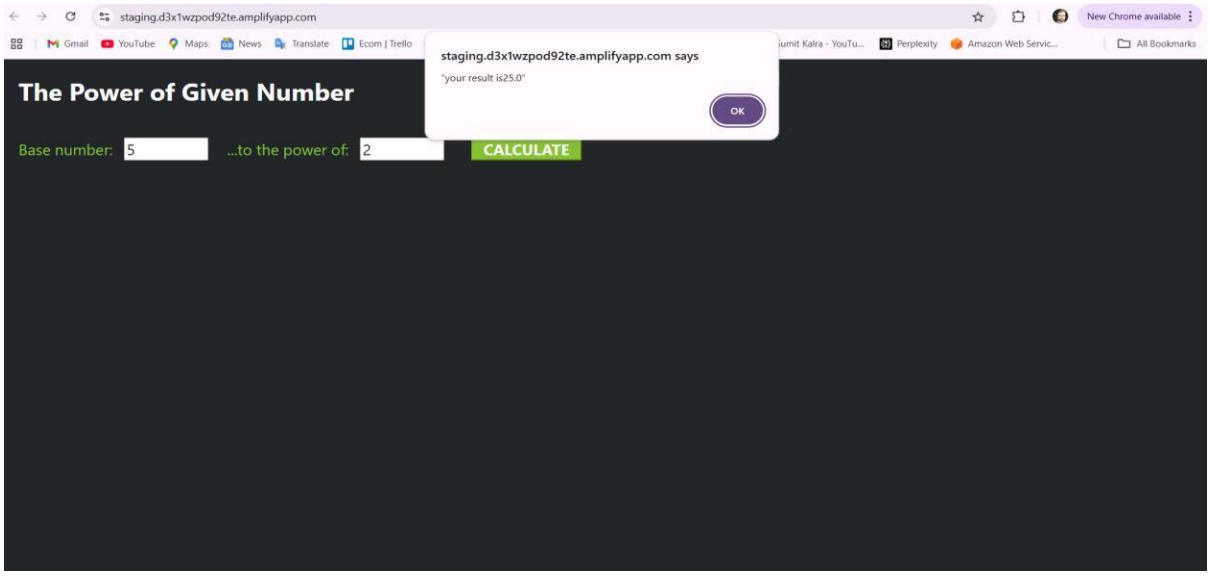

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Sid": "VisualEditor0",
            "Effect": "Allow",
            "Action": [
                "dynamodb:PutItem",
                "dynamodb:DeleteItem",
                "dynamodb:GetItem",
                "dynamodb:Scan",
                "dynamodb:Query",
                "dynamodb:UpdateItem"
            ],
            "Resource": "arn:aws:dynamodb:us-east-1:984527091809:table/mathsCalculation"
        }
    ]
}
```

Deployment



Deployment Process

The deployment process was methodical and structured. First, the front-end was developed and deployed using AWS Amplify. Next, API Gateway was configured to create a REST API that routes requests to the Lambda function. The Lambda function was then authored in Python, tested, and deployed. It was subsequently integrated with the API Gateway. Finally, the DynamoDB table was created, and the necessary IAM policy was assigned. Each step was validated to ensure end-to-end functionality.



Advantages of This Architecture

The serverless architecture offers numerous benefits.

1) It automatically scales with user demand and eliminates the need to provision or manage servers.
2) It is also highly cost-effective, as users are only charged for the compute and storage resources they consume. Moreover, AWS's extensive suite of tools allows for fast development, secure deployment, and seamless integration across services.

Conclusion

This project showcases the power and simplicity of building serverless web applications on AWS. By using AWS Amplify, API Gateway, Lambda, and DynamoDB in conjunction, developers can rapidly prototype and deploy fully functional applications. This architecture not only simplifies development but also ensures reliability, scalability, and minimal operational overhead.

Thank you !!!