Serverless Web Application Deployment using AWS

TEAM MEMBERS:

P.Prudhvi Sai-23BIT0429

B.K.Pranavi -23BIT0314

D.Lakshmi Aneela – 23BIT0415

K.Sai Vibhas – 23BIT0438

TITLE: Serverless Web Application Deployment

DESCRIPTION:

Managing student information is a common requirement for educational institutions. Traditional web applications often rely on dedicated servers, which require continuous management, scaling, and maintenance, leading to higher operational costs and complexity. This project addresses the need for a modern, cost-effective, and scalable solution for managing student data. By leveraging a serverless three-tier architecture on AWS, we can build a system that is resilient, scales automatically, and operates on a pay-as-you-go model, minimizing idle costs.

OUTCOMES:

- The web application successfully performs data storage and retrieval operations.
- The frontend is hosted on AWS S3 and accessible via a CloudFront HTTPS URL.
- The application scales automatically without any manual server management.
- Demonstrates cost-efficient and secure deployment using AWS’s serverless ecosystem.

PROBLEM STATEMENT:

Traditional web applications rely on dedicated servers for hosting and backend processing, which require continuous management, scaling, and maintenance. This leads to higher operational costs and increased complexity.

There is a need for a modern, cost-effective, and scalable solution that allows developers to focus on functionality rather than infrastructure.

This project addresses that challenge by implementing a **Serverless Web Application Deployment using AWS**, where backend logic, data storage, and hosting are fully managed by AWS services. The aim is to develop a **student data management system** that scales automatically, minimizes overhead, and operates on a pay-as-you-use model.

KEY TECHNOLOGIES USED:

AWS Service	Purpose in this Project
Amazon S3	Hosts the static frontend website (HTML, CSS, JavaScript) and serves as the origin for CloudFront.
Amazon DynamoDB	Fully managed NoSQL database used to store the studentData items.
AWS Lambda	Provides the serverless compute logic (Python/boto3) for the getStudent and insertStudentData functions.
Amazon API Gateway	Creates and manages the RESTful API endpoints (GET, POST) that trigger the Lambda functions.
Amazon CloudFront	(Enhancement) Acts as a CDN to securely and quickly deliver the S3 website to users worldwide.
IAM	Manages permissions and roles, specifically the LambdaDynamoDb-Role which allows Lambda to access DynamoDB.
Programming Language	Python (used for developing AWS Lambda functions)
Frontend Technologies	HTML, CSS, and JavaScript (used to design and control the web interface)

Data Format	JSON (used for sending and receiving data between frontend, API Gateway, and Lambda functions)

Web Browser	Google Chrome (used to run and test the deployed web application)

OVERVIEW:

- The project focuses on deploying a **serverless web application** using **Amazon Web Services (AWS)**.
- It demonstrates how to build and host a **student data management system** without managing servers.
- The application allows users to **add** and **view** student details such as ID, name, class, and age.
- **AWS Lambda** handles backend logic for inserting and retrieving data.
- **Amazon DynamoDB** stores all student records in a NoSQL format.
- **Amazon API Gateway** connects the frontend to backend Lambda functions.
- **Amazon S3** hosts the static frontend files (HTML, CSS, JavaScript).
- **Amazon CloudFront** provides global HTTPS access and improved security.
- The system ensures **automatic scalability**, **cost-efficiency**, and **simplified deployment**.
- The project demonstrates a complete **end-to-end serverless architecture** for modern web development.

OBJECTIVES:

- To design and deploy a **serverless web application** using Amazon Web Services (AWS).
- To develop a **student data management system** that allows adding and viewing student details.
- To implement **AWS Lambda functions** for backend processing without using traditional servers.
- To integrate **Amazon API Gateway** with Lambda for handling HTTP GET and POST requests.
- To use **Amazon DynamoDB** as a NoSQL database for storing student information.
- To host the frontend (HTML, CSS, JavaScript) on **Amazon S3** as a static website.
- To enhance application **security and performance** using **Amazon CloudFront** for HTTPS access.
- To demonstrate **cost optimization, scalability, and reliability** through a serverless architecture.

Architecture Design :

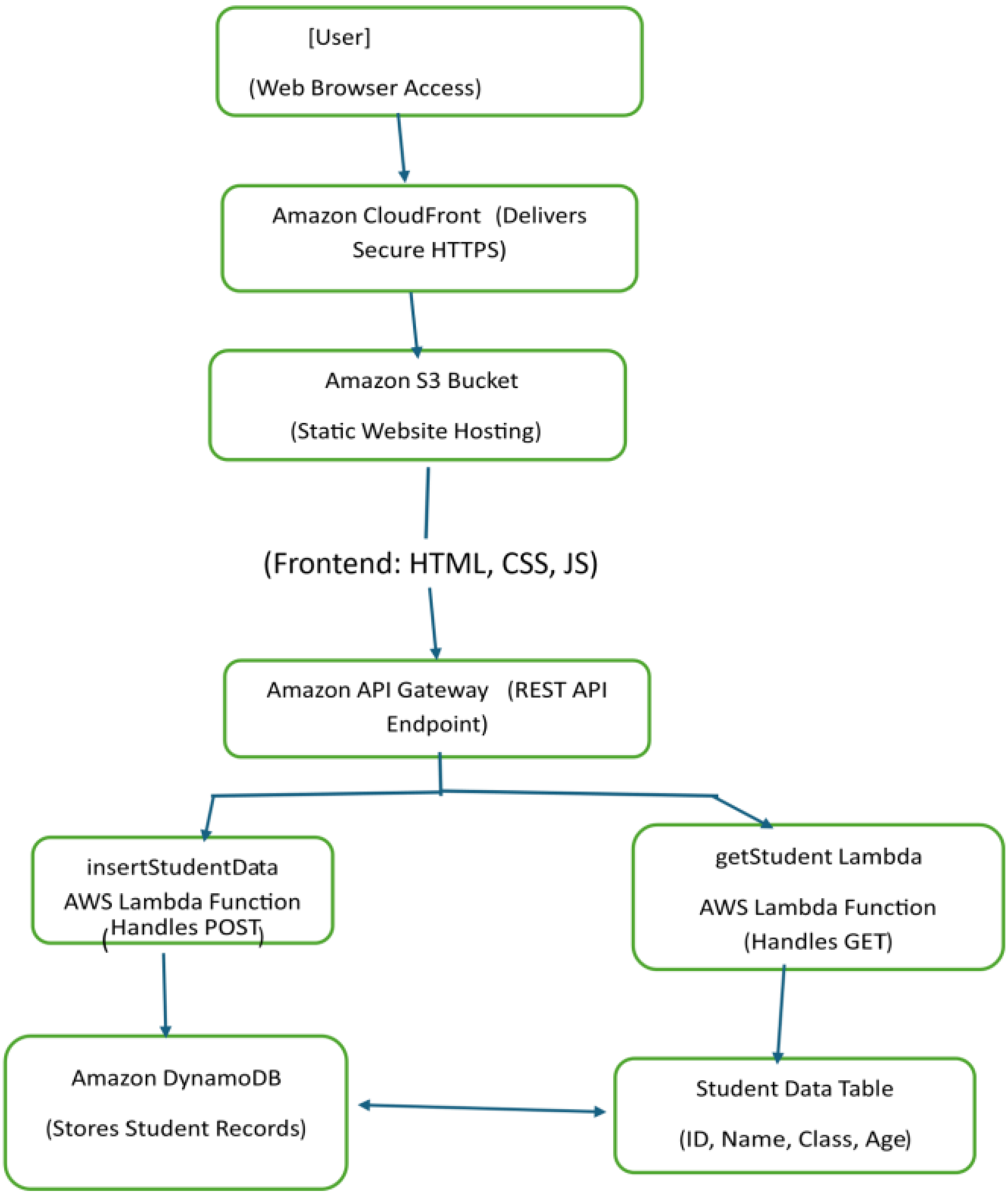
The system follows a classic three-tier architecture, implemented with serverless components:

1. **Presentation Tier (Frontend):**
   - **Amazon S3**: Used for static website hosting. It stores the index.html, scripts.js, and any CSS files.
   - **Amazon CloudFront**: Acts as the Content Delivery Network (CDN). It provides a secure, public-facing URL (https://...cloudfront.net) and caches the website at edge locations, improving performance. It also allows the S3 bucket itself to remain private, enhancing security.
2. **Application Tier (Logic):**
   - **Amazon API Gateway:** Provides the RESTful API "front door." It exposes HTTP endpoints (e.g., POST / and GET /) that the frontend JavaScript can call.
   - **AWS Lambda:** Contains the serverless business logic (written in Python). Two functions are used:

     ->insertStudentData: Triggered by the POST request. It parses the incoming student data and saves it to DynamoDB.

     ->getStudent: Triggered by the GET request. It scans the DynamoDB table and returns a list of all students.
3. **Data Tier (Database):**
   - **Amazon DynamoDB**: A fully managed NoSQL database. A table named studentData is used to store student records, with studentId as the partition key.

<u>WORKFLOW:</u>



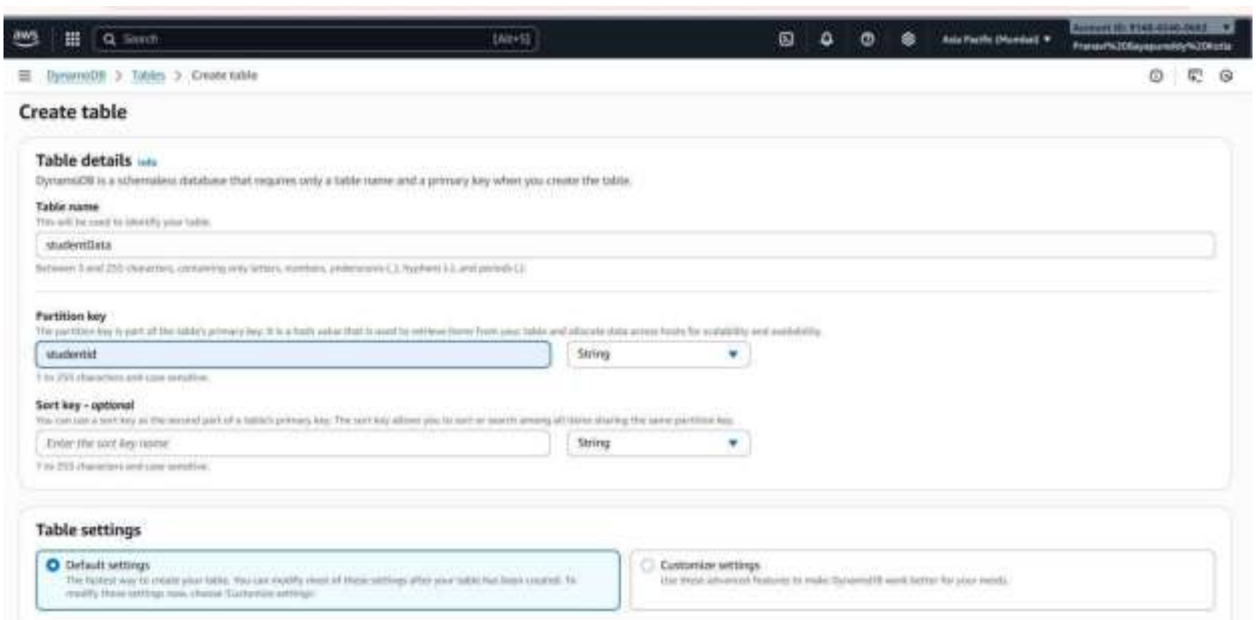
Implementation Steps & Screenshots:

This section details the step-by-step implementation, as captured in the project screenshots.

Part 1: Data & Logic Tiers (DynamoDB & Lambda):-

1.Create DynamoDB Table

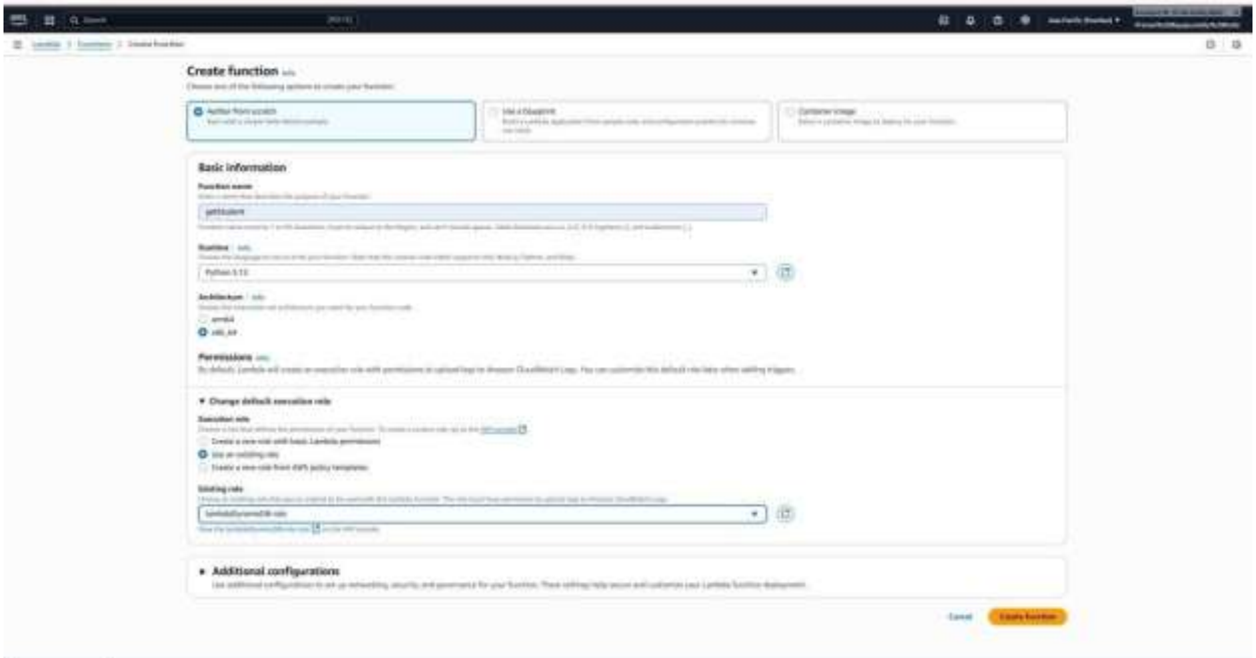
- Open the AWS Management Console and navigate to DynamoDB.
- Click Create Table and name it studentData.
- Set the Partition Key as Studentid (String).
- Leave other settings as default and create the table.



2.Create Lambda Function for Retrieving Data (GET) • Go

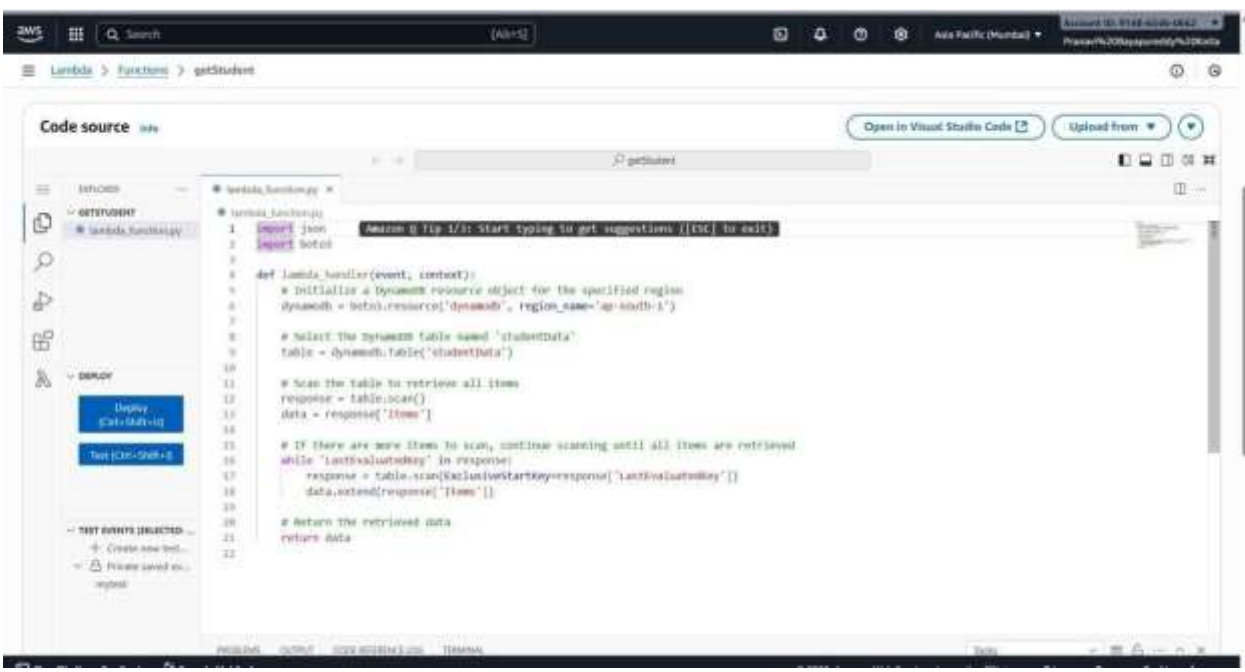
to AWS Lambda → Create Function → From Scratch.

- Function name: getStudent.
- Runtime: Python 3.12.
- Assign an existing IAM role with DynamoDB read permissions.



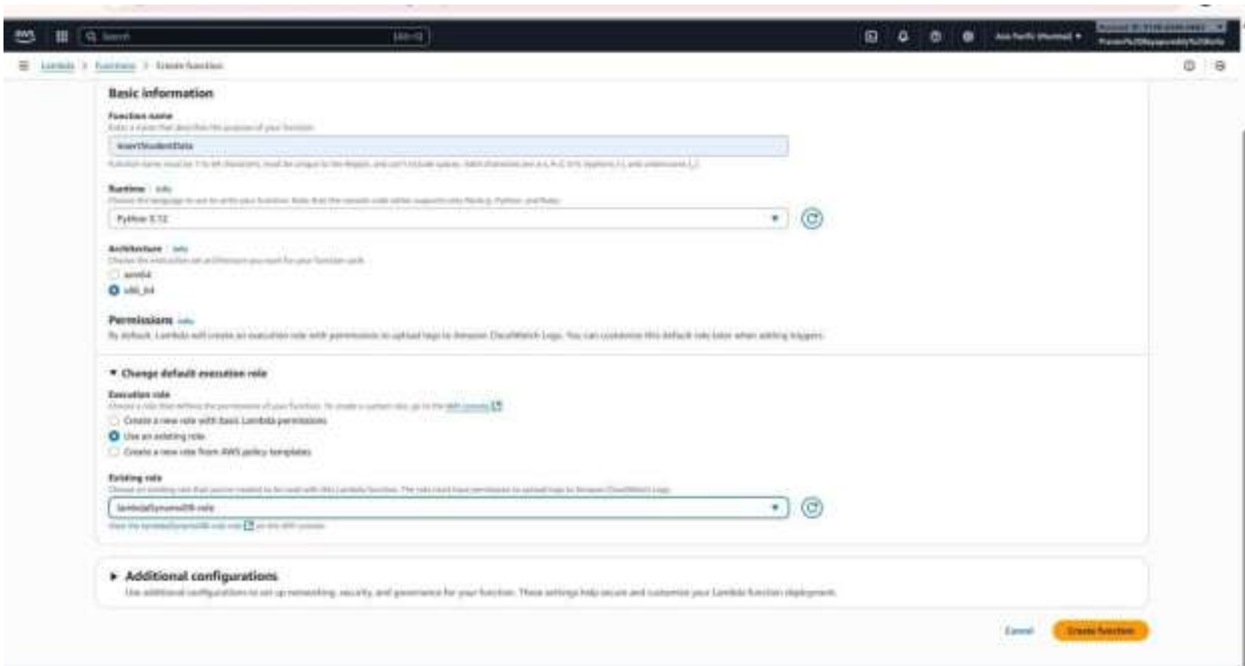
- Add Python code using boto3 to scan and return all items from the DynamoDB table.

- Click Deploy and test using a sample event (empty payload).

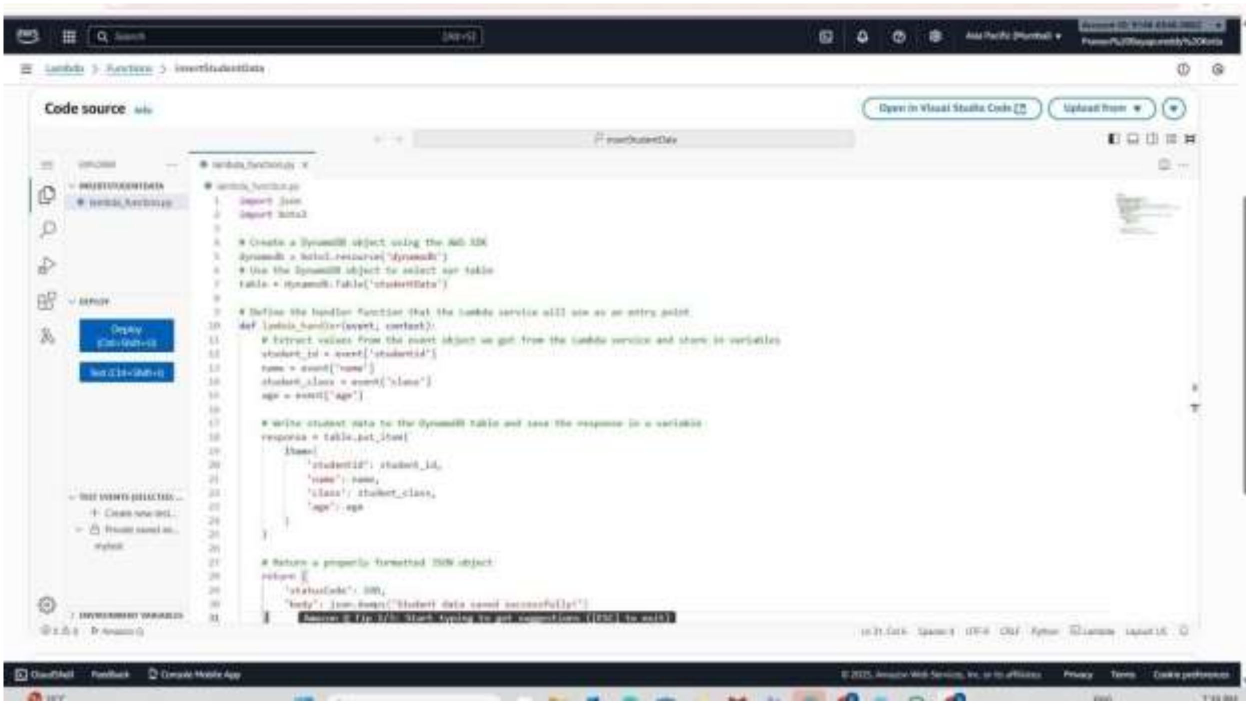


3.Create Lambda Function for Inserting Data (POST)

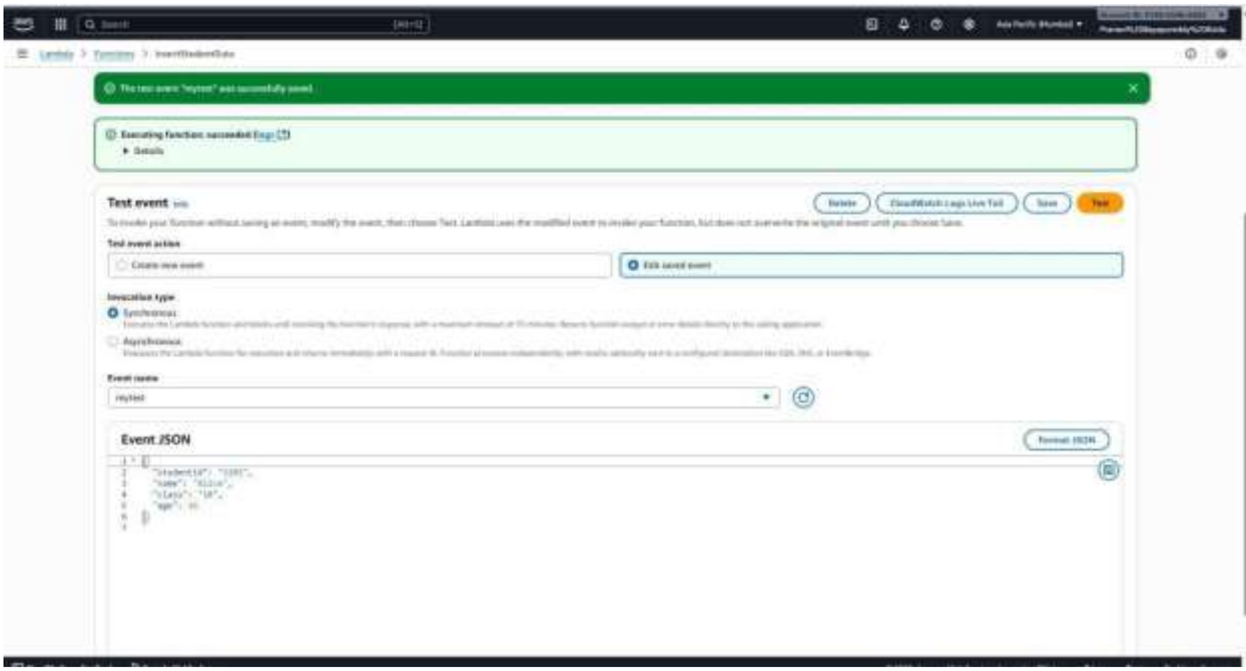
- Create another function named insertStudentData.
- Use the same runtime (**Python 3.12**) and role as before.



- Write Python code to parse the input JSON (Studentid, Name, Class, Age) and insert it into DynamoDB using put_item().
- Deploy and test using a JSON test event to verify that data is stored correctly
- Validate the record in DynamoDB using **Explore Table Items**



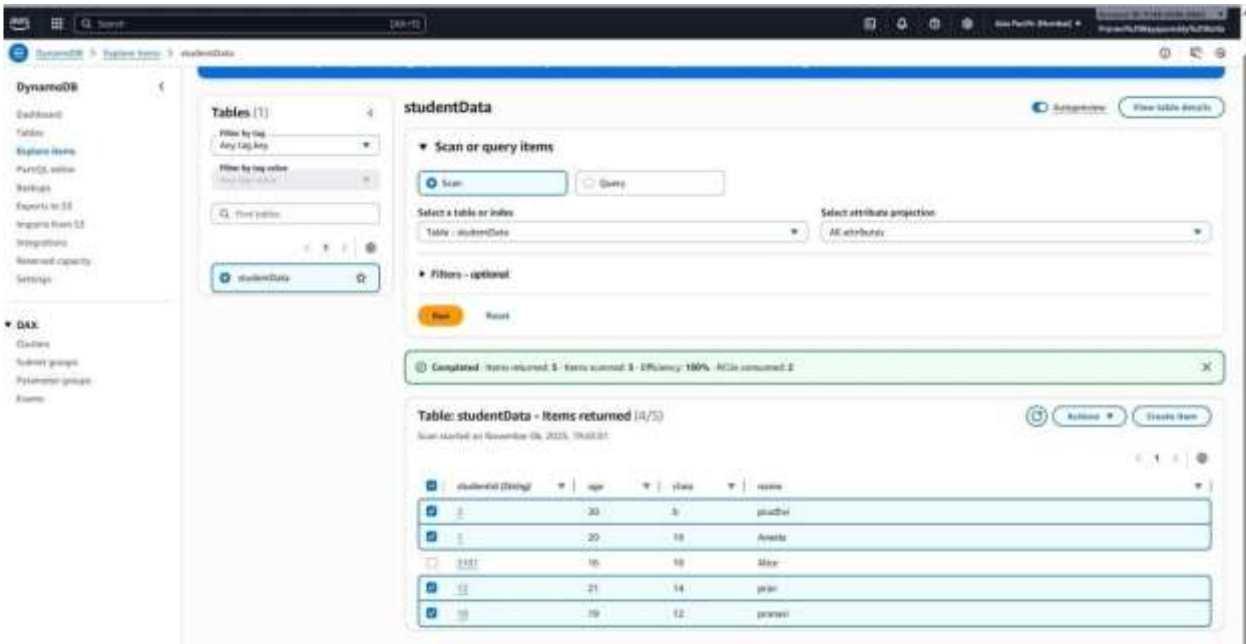
4. **Test Lambda Function:**
    - The insertStudentData function was tested directly in the Lambda console.
    - A test event named mytest was configured with sample JSON data for a student.



- The test executed successfully, returning a statusCode: 200 and a success message.

- The successful insertion was verified by going to DynamoDB > Explore items

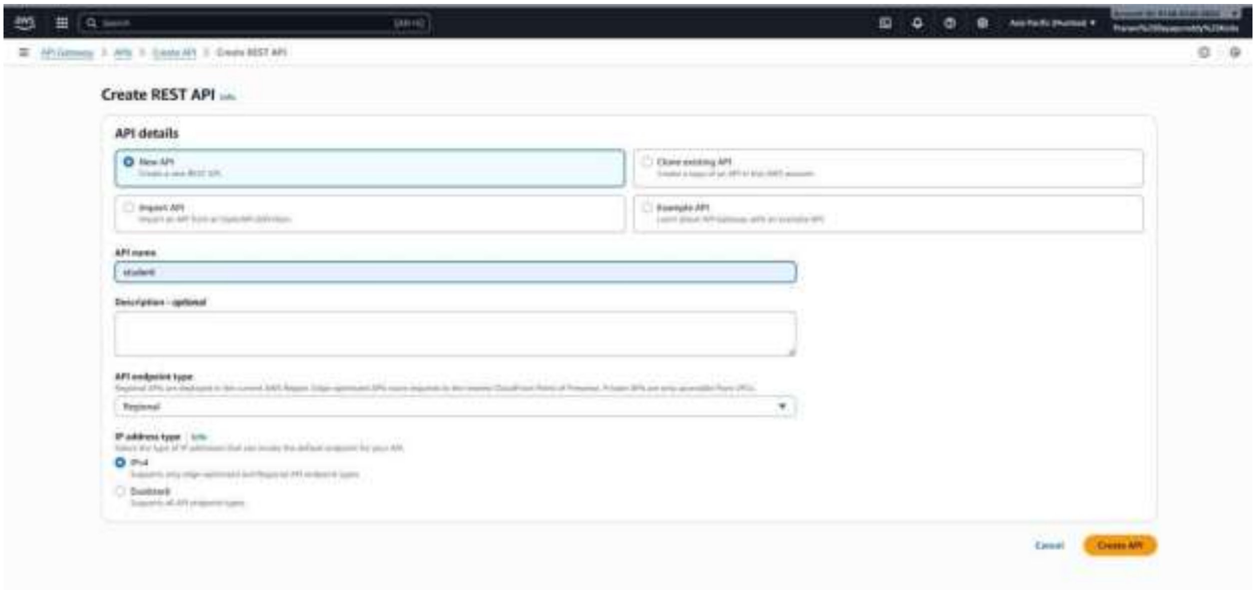
> studentData. The new record was visible in the table



Part 2: Application and Presentation Tiers (API Gateway & S3):-

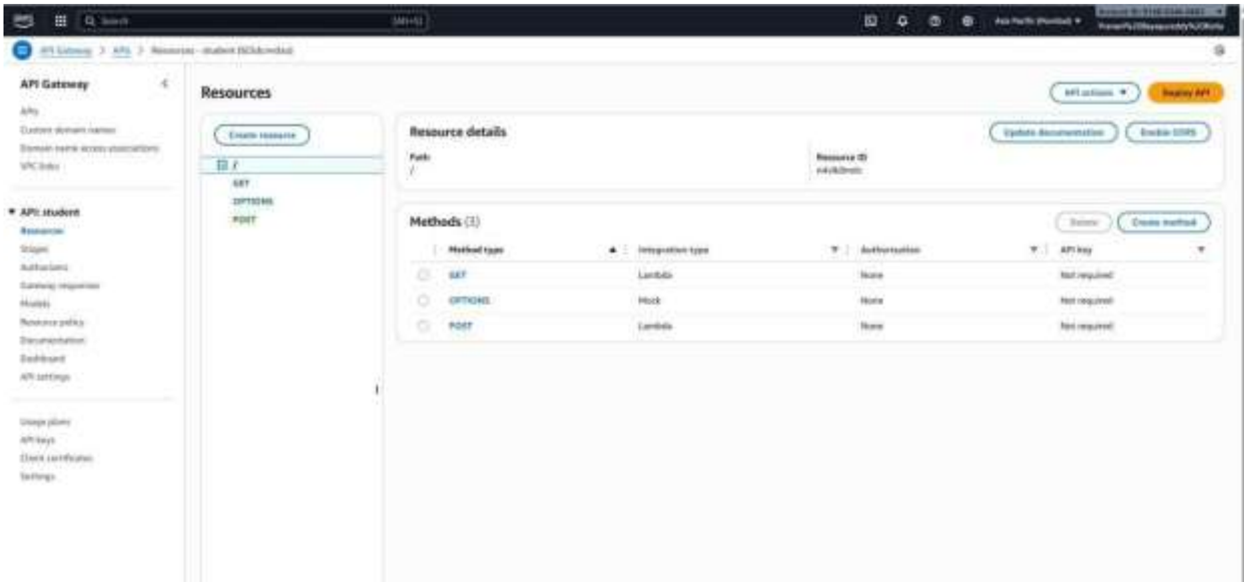
1.Create API Gateway

- Navigate to API Gateway in the AWS Console.
- Click Create API → REST API → New API.
- Name it student and choose Edge-Optimized endpoint type for global access.



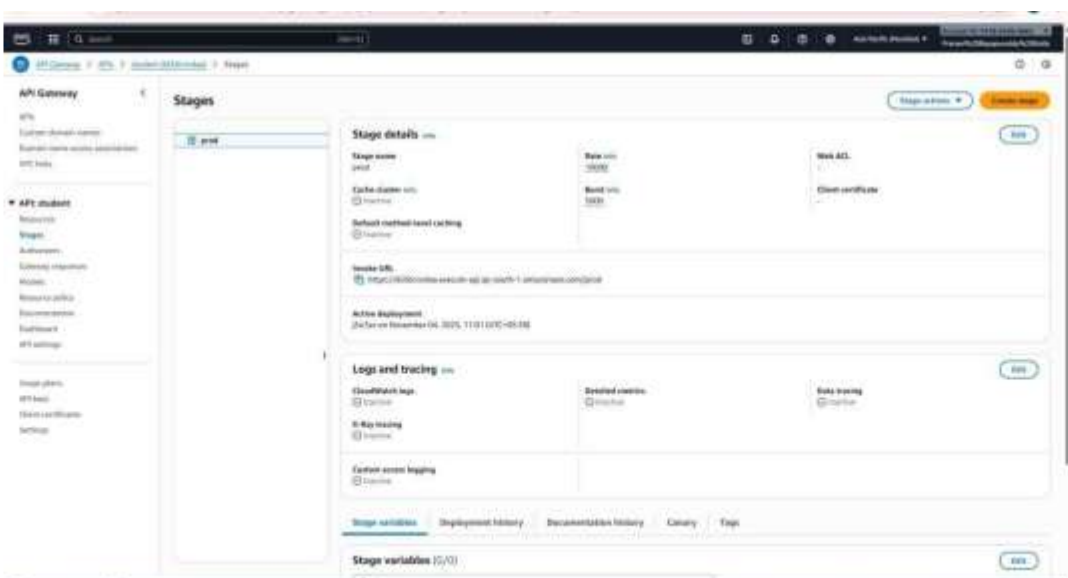
2.Create API Methods

- Add a **GET** method and integrate it with the getStudent Lambda function.
- Add a **POST** method and integrate it with the insertStudentData Lambda function.
- Test both methods using the **Test** option in API Gateway to confirm successful responses.

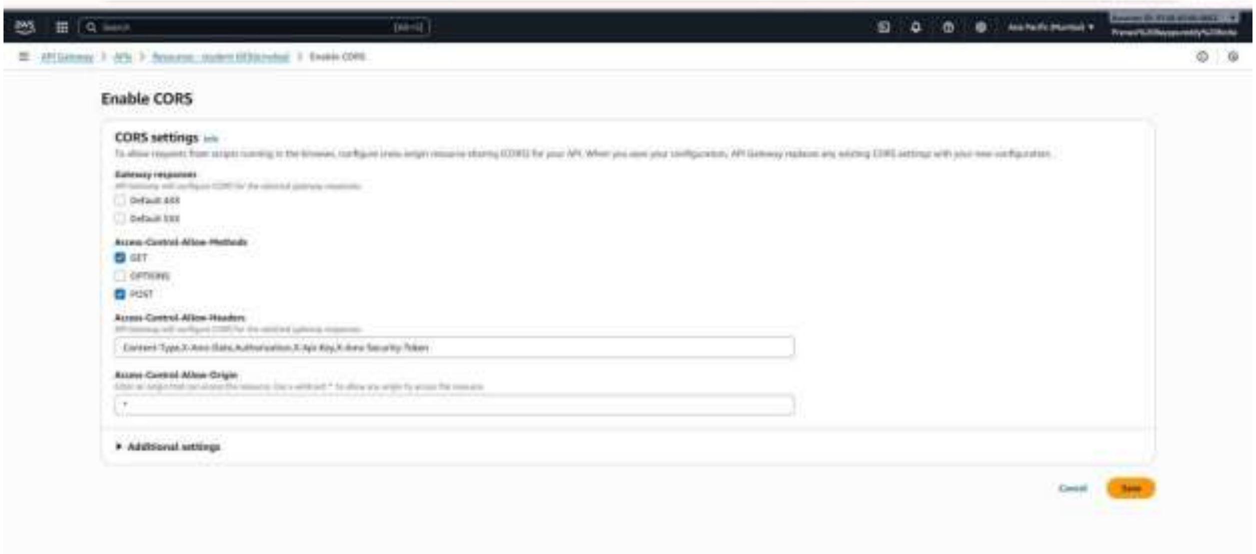


3. Deploy API & Configure CORS

- The API was deployed to a New Stage named prod.
- This generated the public Invoke URL(https://6l3dcnvdaa.executeapi.apsouth-1.amazonaws.com/prod)

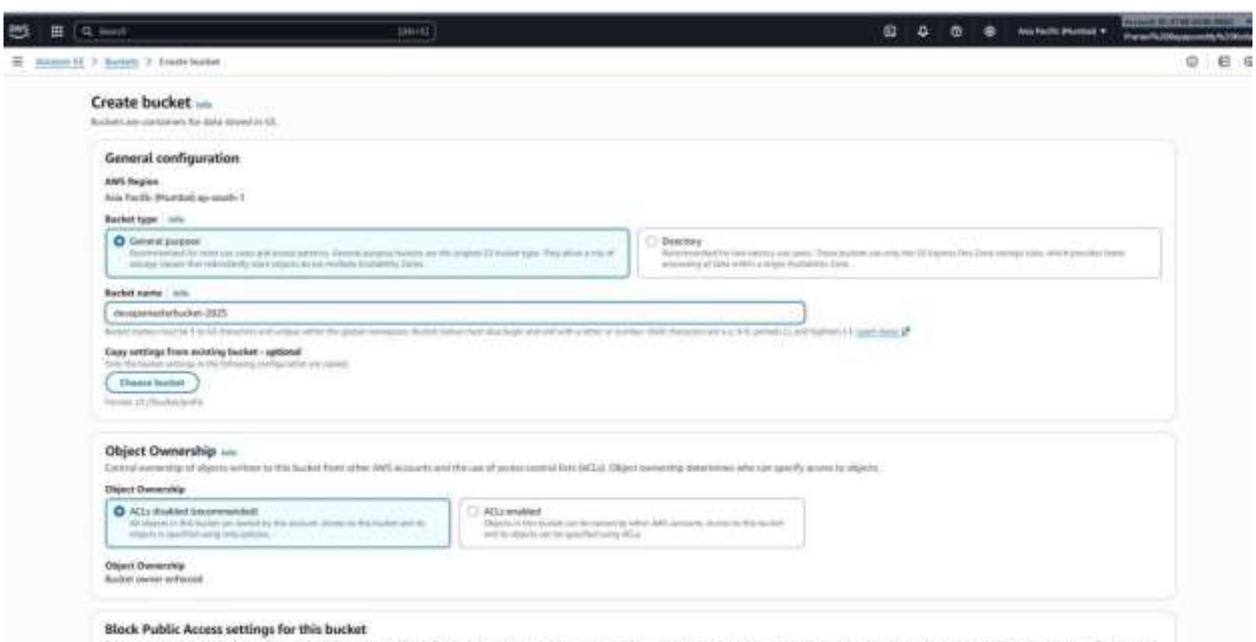


CORS (Cross-Origin Resource Sharing) was enabled for the API to allow the S3 website (on a different domain) to make requests. This automatically set up an OPTIONS method and added the necessary Access-Control-AllowOrigin: * headers.

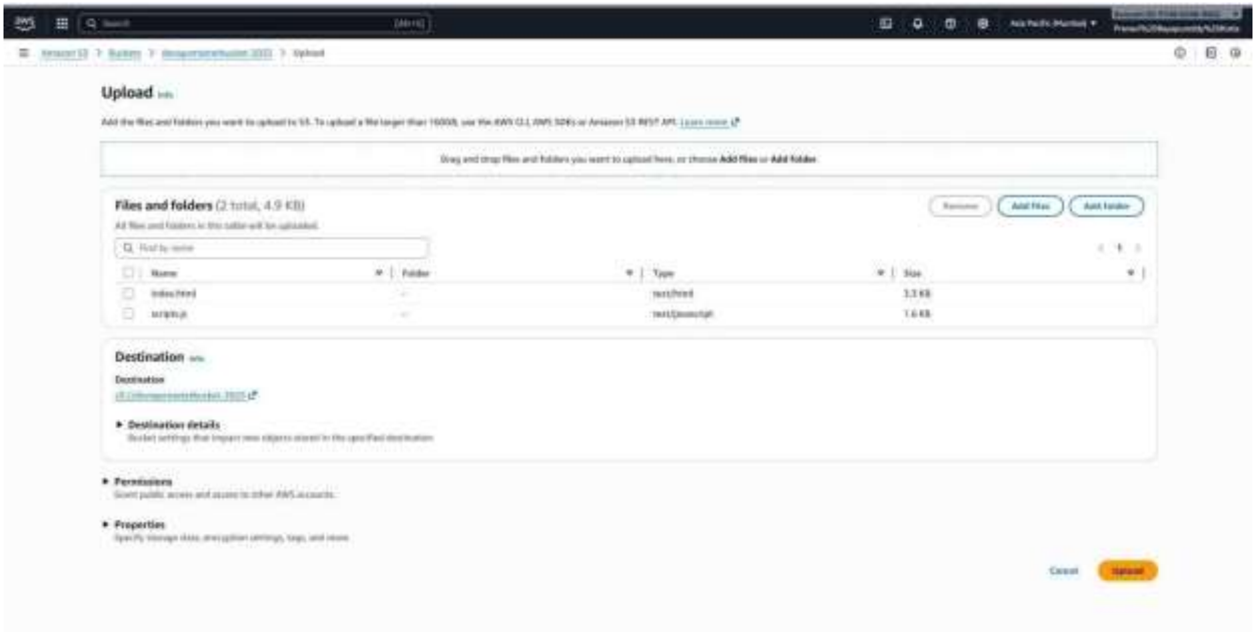


4.Configure S3 Bucket for Website Hosting

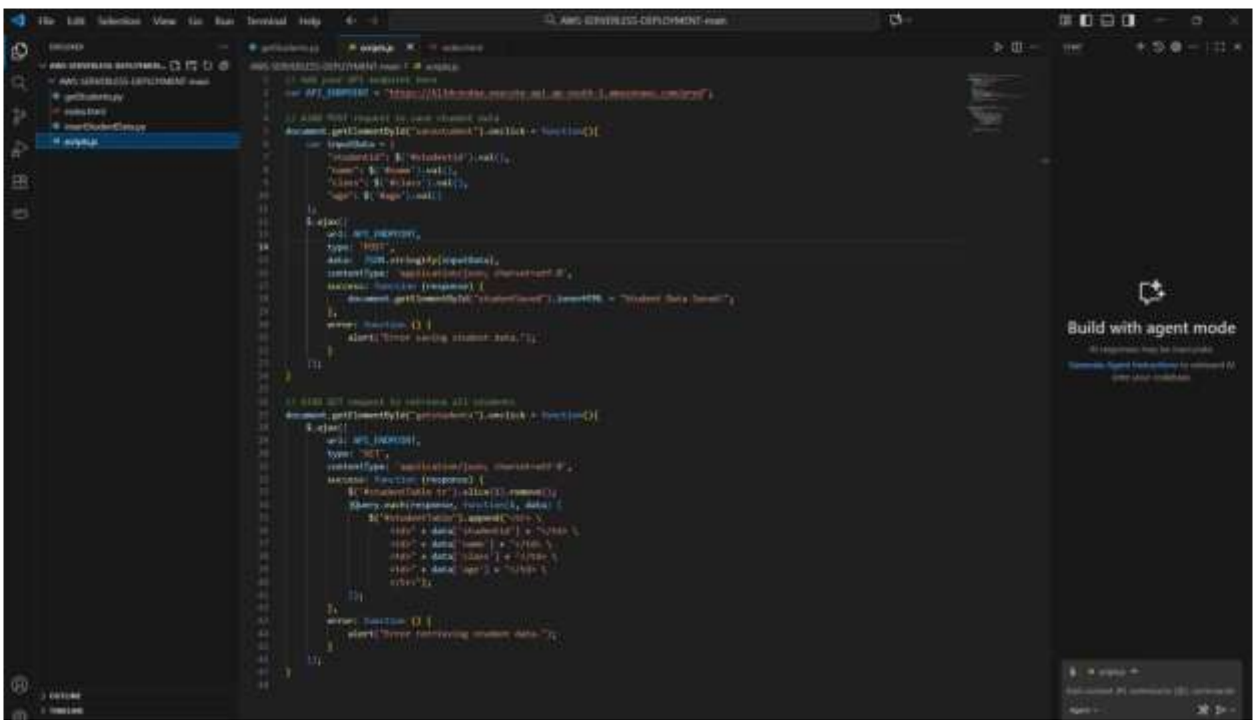
- An S3 bucket named [devopsmasterbucket-2025]() was created in the apsouth-1 region.



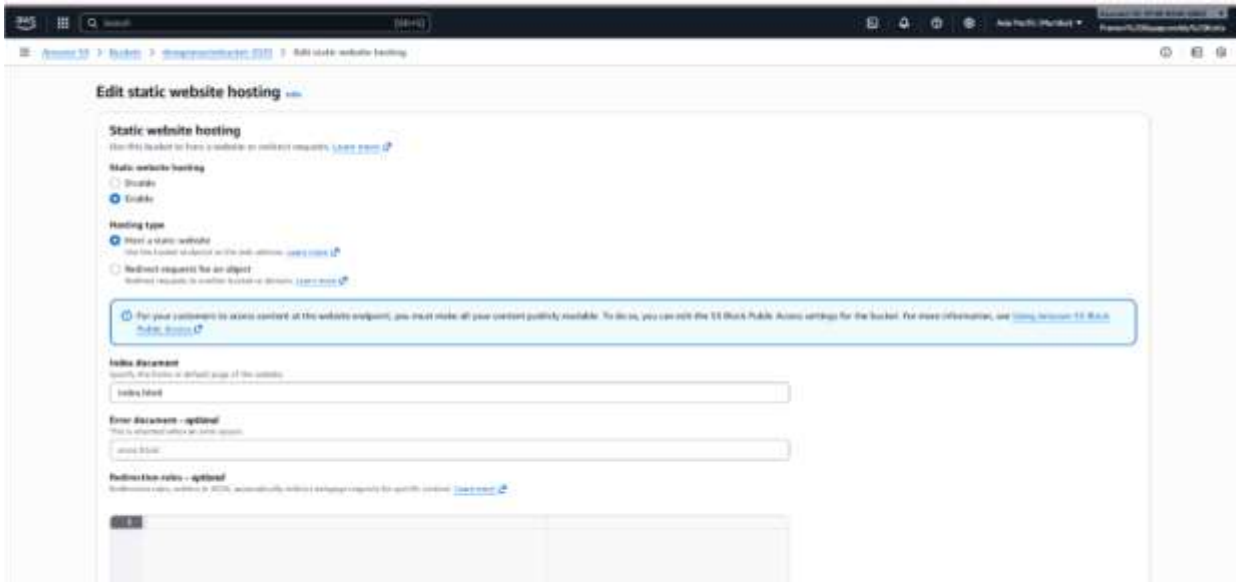
- The frontend files (scripts.js, index.html) were uploaded to the bucket.



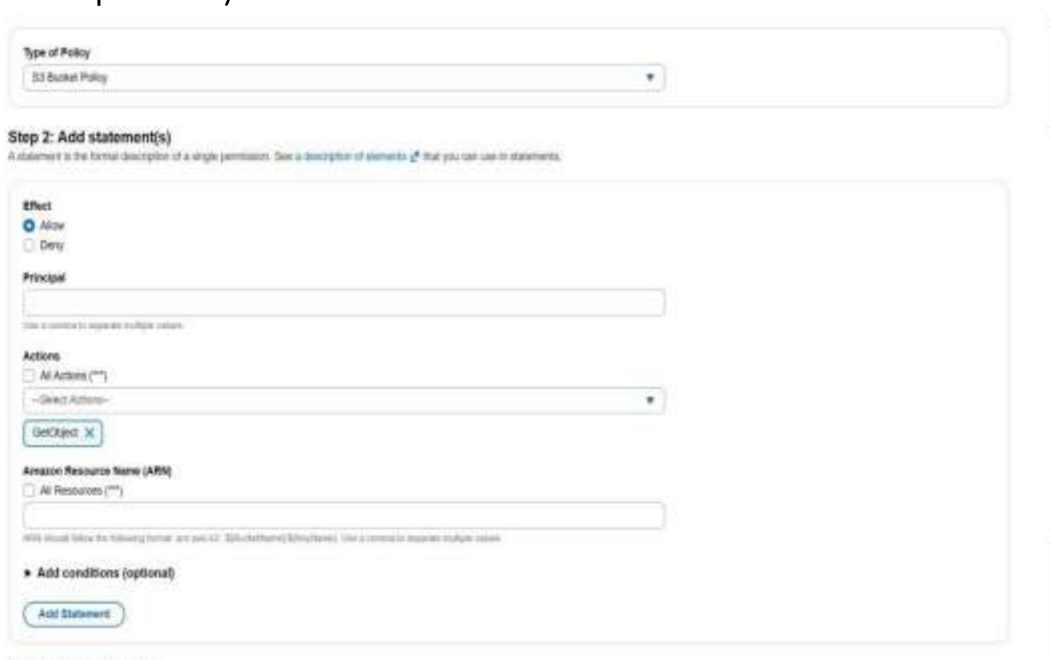
- The scripts.js file was edited to include the API Gateway Invoke URL in the API_ENDPOINT variable.

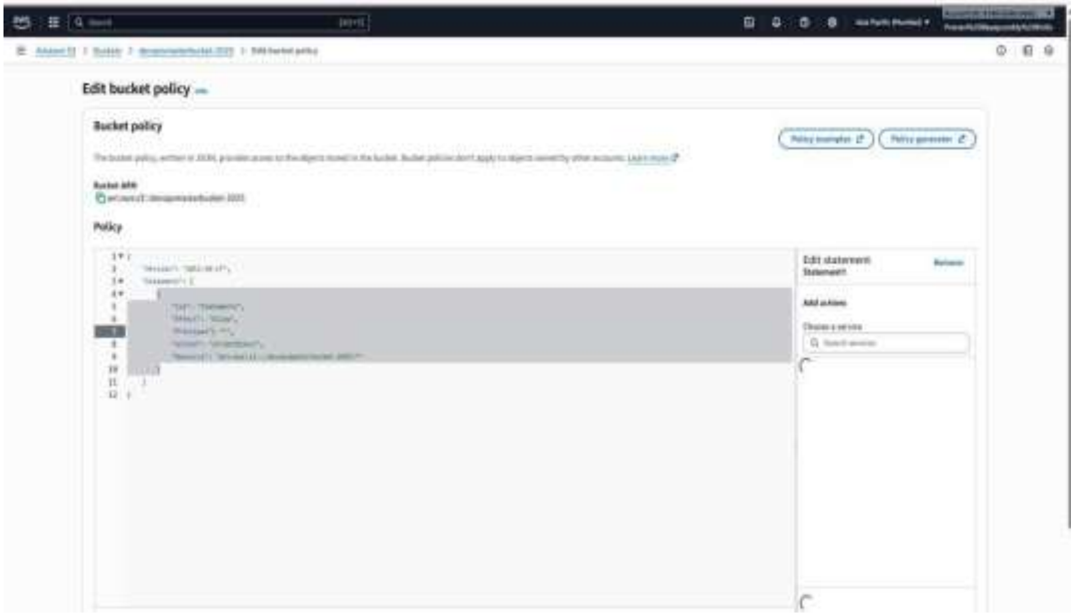


- Static website hosting was enabled on the bucket, with index.html as the index document.



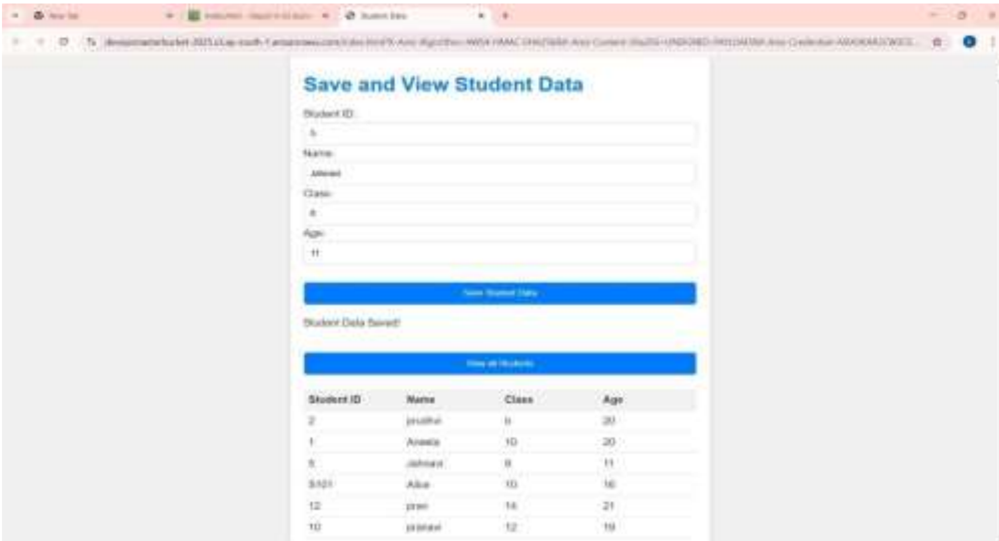
- A public bucket policy was generated (using the AWS Policy Generator) and applied to the bucket to allow s3:GetObject actions from any principal ("Principal": "*").



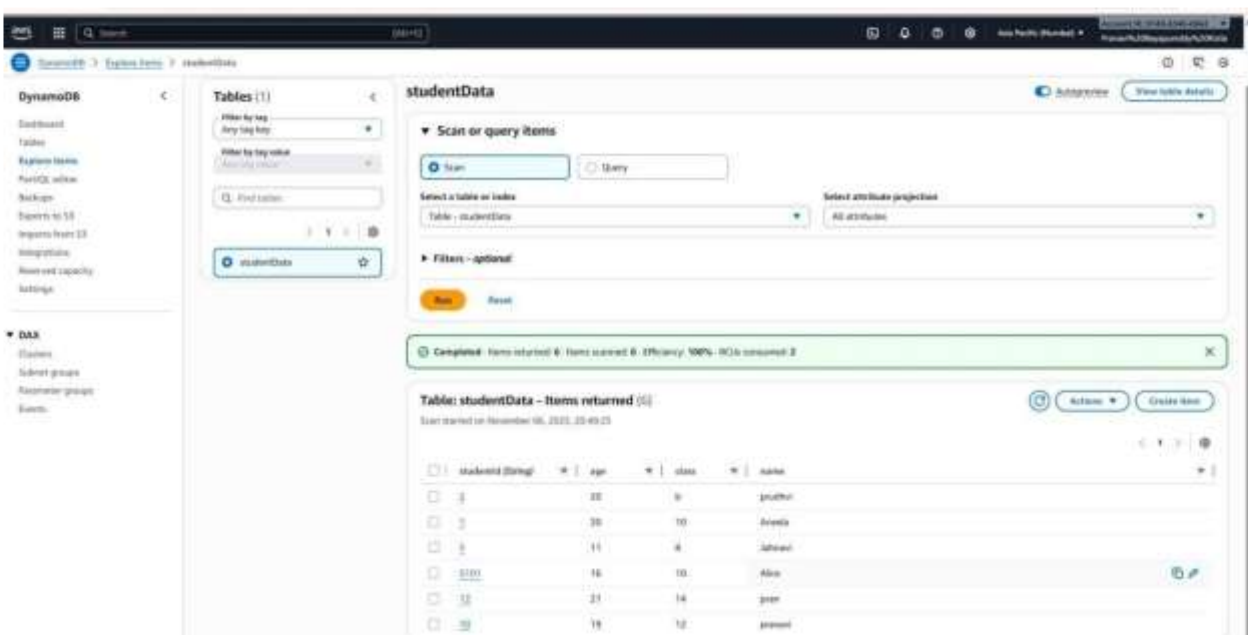


5. Test Final Website (S3 URL):

- The S3 website endpoint URL was opened in a browser.
- The "View all Students" button successfully fetched and displayed the initial test data.
- A new student ("Jahnavi") was added via the form, which successfully saved and updated the table on the page.



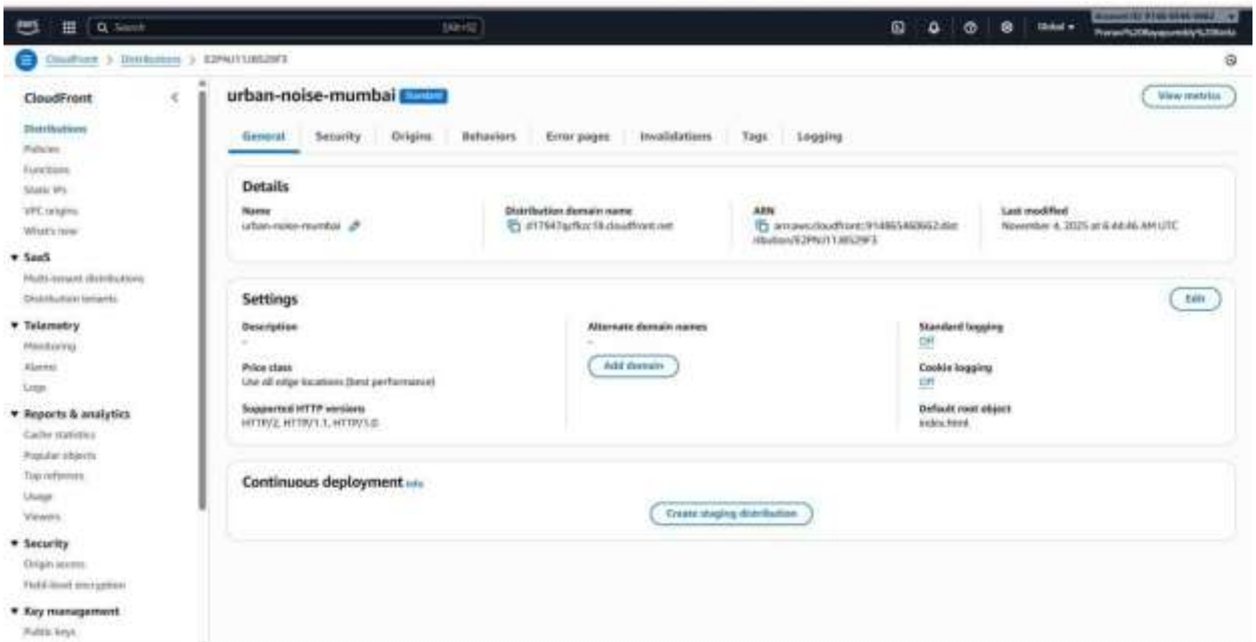
DB Table Result:



Part 3: (Enhancement) Securing with CloudFront

1.Create CloudFront Distribution:

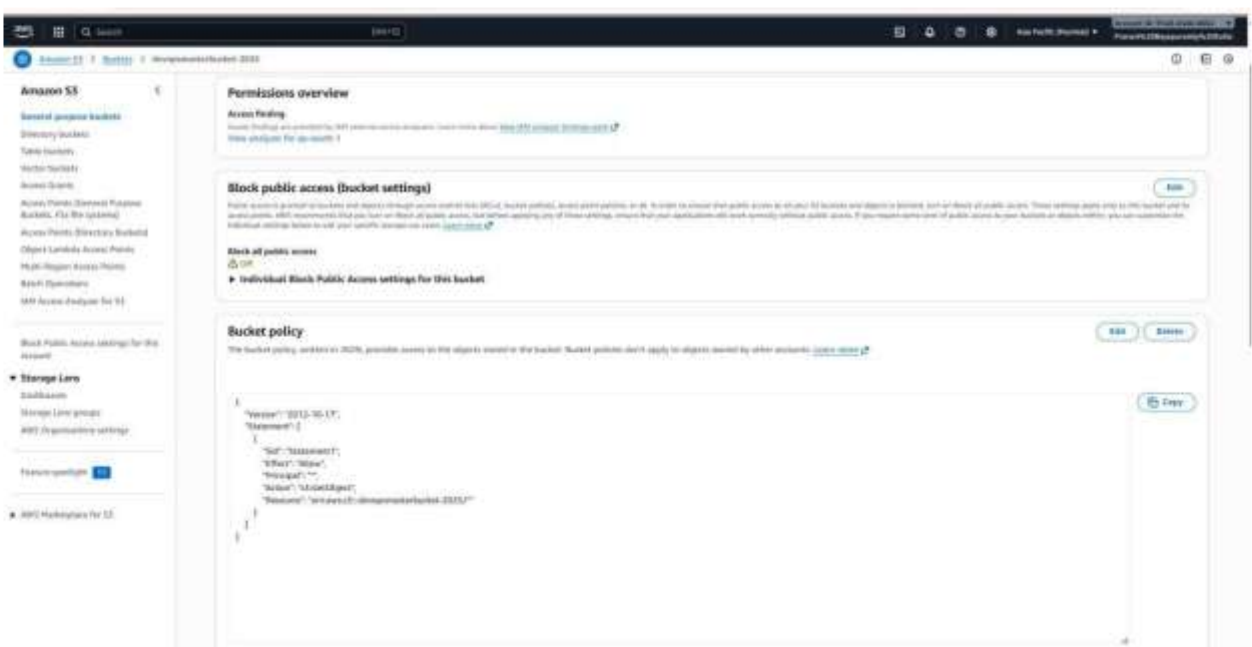
- A CloudFront distribution (aws-three-tier) was created.
- The Origin domain was set to the S3 bucket's static website endpoint (http://devopsmasterbucket-2025.s3-website.ap-south-1.amazonaws.com).



2. Secure S3 Bucket:

- To ensure users only access the site via CloudFront, Origin access control settings (OAC) was configured.
- This generated a new S3 bucket policy that only allows s3:GetObject from the CloudFront service principal.
- The previous public bucket policy was replaced with this new, more secure policy. o Block public access (bucket settings) was then turned On for the S3 bucket.

Block public access and Update new bucket policy:



is

3. Final Secure URL:

The final, secure, and globally distributed URL for the application https://devopsmasterbucket-2025.s3.ap-south-1.amazonaws.com/index.html

ADVANTAGES:

- No need to manage servers — AWS handles everything automatically.
- Very cost-efficient because you only pay for what you use.
- Automatically scales when more users access the application.
- Easy to deploy and update since all components are serverless.

LIMITATIONS:

- Lambda functions may have a short delay during the first run (cold start).
- Limited execution time for AWS Lambda functions.
- Requires an active internet connection to access AWS services.
- Debugging and error tracking can be more difficult than traditional servers.

FUTURE ENHANCEMENTS:

- Add user login and authentication using **Amazon Cognito**.
- Include update and delete options for complete data management.

- Integrate **AWS CloudWatch** for monitoring and performance tracking. Use **AWS Amplify** for automated deployment and hosting.

CONCLUSION:

This project successfully demonstrates the development and deployment of a **serverless web application** using various **AWS cloud services** such as **Lambda, API Gateway, DynamoDB, S3, and CloudFront**. The application manages student data efficiently and allows users to add and view information through a simple web interface.By using a **serverless architecture**, the project removes the need for setting up or maintaining any physical servers. All infrastructure tasks such as scaling, security, and resource management are handled automatically by AWS. This results in **lower costs, faster performance, and easier maintenance**.

The use of **AWS Lambda** for backend logic and **DynamoDB** for data storage ensures that the system can handle multiple requests simultaneously without any downtime. **API Gateway** helps connect the frontend and backend securely, while **S3** and **CloudFront** provide a fast and safe way to host and access the website globally.Overall, this project proves that **serverless computing** is an ideal solution for modern applications. It provides **flexibility, scalability, and cost efficiency**, making it suitable for real-world use in both small and large-scale systems. This project also gives practical experience in integrating multiple AWS services to build a complete, fully functional web-based system.