

# **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:**

<b>AWS Service</b>	<b>Purpose in this Project</b>
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):**

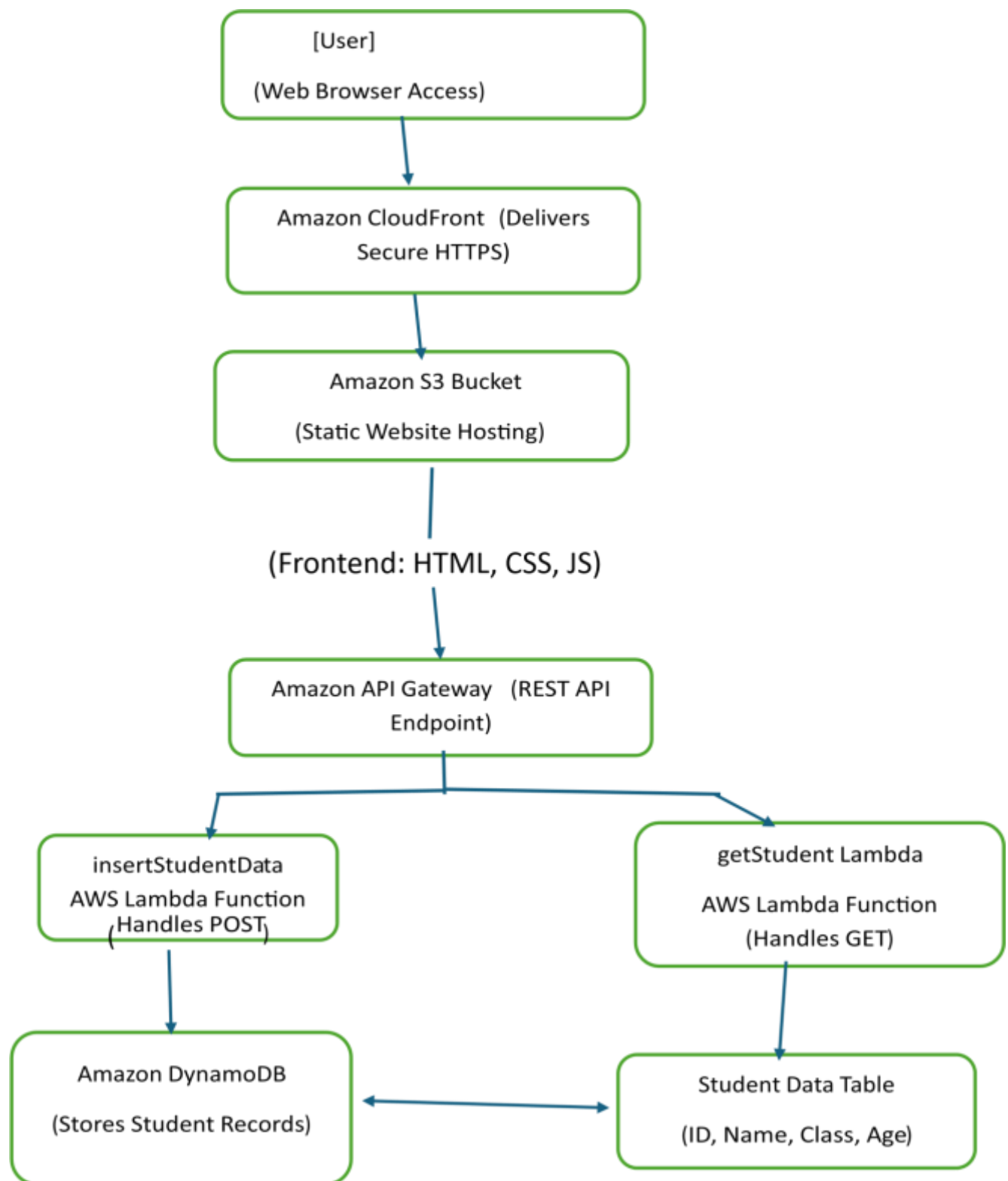
- **Amazon API Gateway:** Provides the RESTful API "front door." It exposes HTTP endpoints (e.g., POST / and GET /) that the frontend JavaScript can call.
- **AWS Lambda:** Contains the serverless business logic (written in Python). Two functions are used:
  - >insertStudentData: Triggered by the POST 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.

**WORKFLOW:**





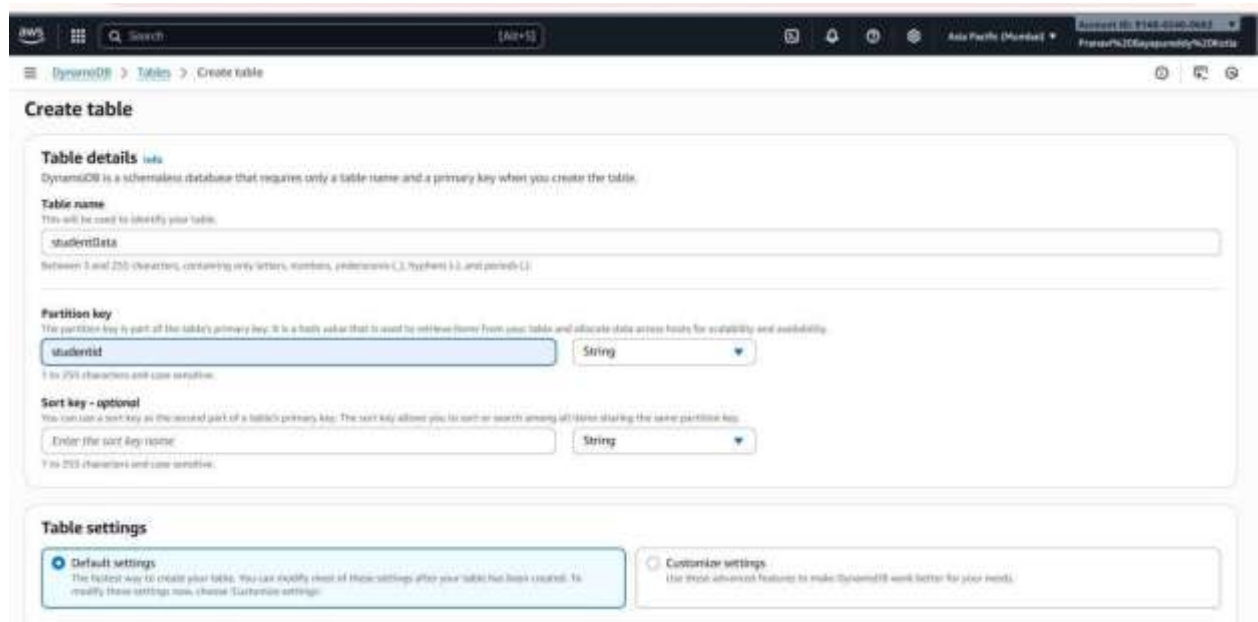
### Implementation Steps & Screenshots:

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.



The screenshot shows the 'Create table' page in the AWS Management Console. The page is titled 'Create table' and has a breadcrumb trail: 'DynamoDB > Tables > Create table'. The 'Table details' section includes a 'Table name' field with the value 'studentData' and a 'Partition key' field with the value 'studentid' and a dropdown set to 'String'. The 'Sort key - optional' section is empty. The 'Table settings' section has two tabs: 'Default settings' (selected) and 'Customize settings'.

### 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.

**Create function**

Choose one of the following options to create your function:

- Author from scratch** (Selected)
- Use a template**
- Container image**

**Basic information**

**Function name**  
Enter a name that describes the purpose of your function.

**Runtime**  
Select the programming language and version for your function. (See the [runtime support page](#) for details.)

**Architecture**  
Select the architecture for your function.  
☒ arm64 ☐ x86\_64

**Permissions**  
By default, Lambda will create an execution role with permissions to upload logs to Amazon CloudWatch Logs. You can customize this default role later when setting VPCs.

**Change default execution role**

**Execution role**  
Select a role that allows the permissions of your function. To create a custom role, click the [get IAM role](#) link.

☐ Create a new role with basic Lambda permissions

☒ Use an existing role

☐ Create a new role from default policy templates

**Starting role**  
Select a starting role for your function. (See [IAM Roles for Lambda Functions](#) for details.)

**Additional configurations**  
Add additional configurations to set up networking, security, and governance for your function. These options help secure and optimize your Lambda function deployment.

[Cancel](#) [Create function](#)

- 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).

**Code source**

[Open in Visual Studio Code](#) [Upload from](#)

```

1 import json
2 import boto3
3
4 def lambda_handler(event, context):
5     # Initialize a DynamoDB resource object for the specified region
6     dynamodb = boto3.resource('dynamodb', region_name='ap-south-1')
7
8     # Select the DynamoDB table named 'studentData'
9     table = dynamodb.Table('studentData')
10
11     # Scan the table to retrieve all items
12     response = table.scan()
13     data = response['Items']
14
15     # If there are more items to scan, continue scanning until all items are retrieved
16     while 'LastEvaluatedKey' in response:
17         response = table.scan(ExclusiveStartKey=response['LastEvaluatedKey'])
18         data.extend(response['Items'])
19
20     # Return the retrieved data
21     return data
22
  
```

[Deploy](#) [Get IAM role](#) [Test \(Ctrl+Shift+S\)](#)

**TEST EVENTS (SELECTED)**

- Create new test...
- Private saved test...
- request

### 3.Create Lambda Function for Inserting Data (POST)

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

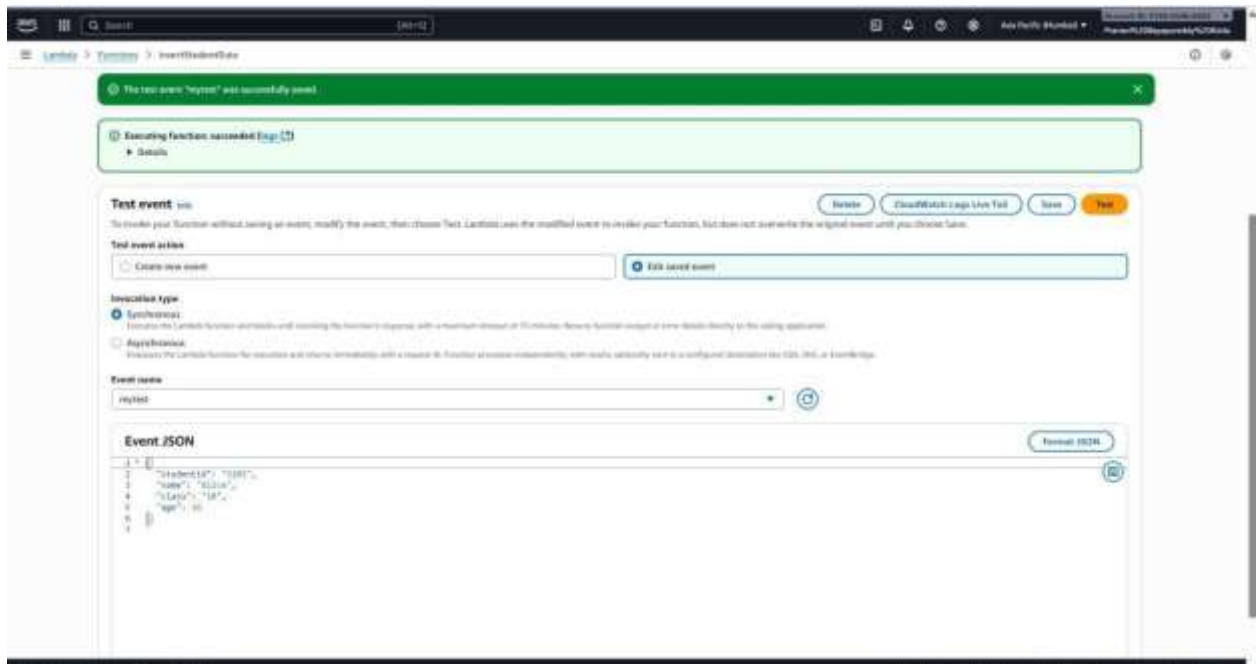
The screenshot shows the AWS Lambda console's 'Create Function' page. The 'Function name' field is populated with 'insertStudentData'. The 'Runtime' is set to 'Python 3.12'. The 'Architecture' is set to 'x86\_64'. The 'Permissions' section is expanded, showing 'Change default execution role' with 'Use an existing role' selected, and the role is 'lambda:role-arn'. The 'Additional configurations' section is empty. The 'Create Function' button is visible at the bottom right.

- 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:

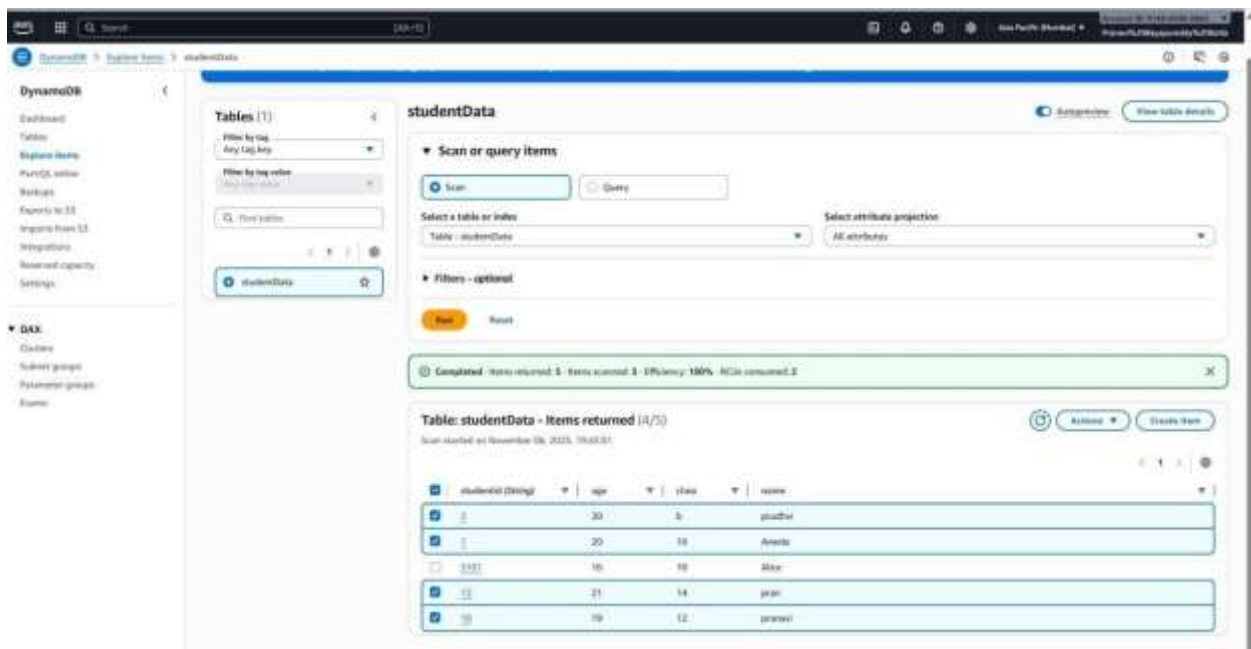
- The insertStudentData function was tested directly in the Lambda console.
- A test event named mytest was configured with sample JSON data for a student.



- 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

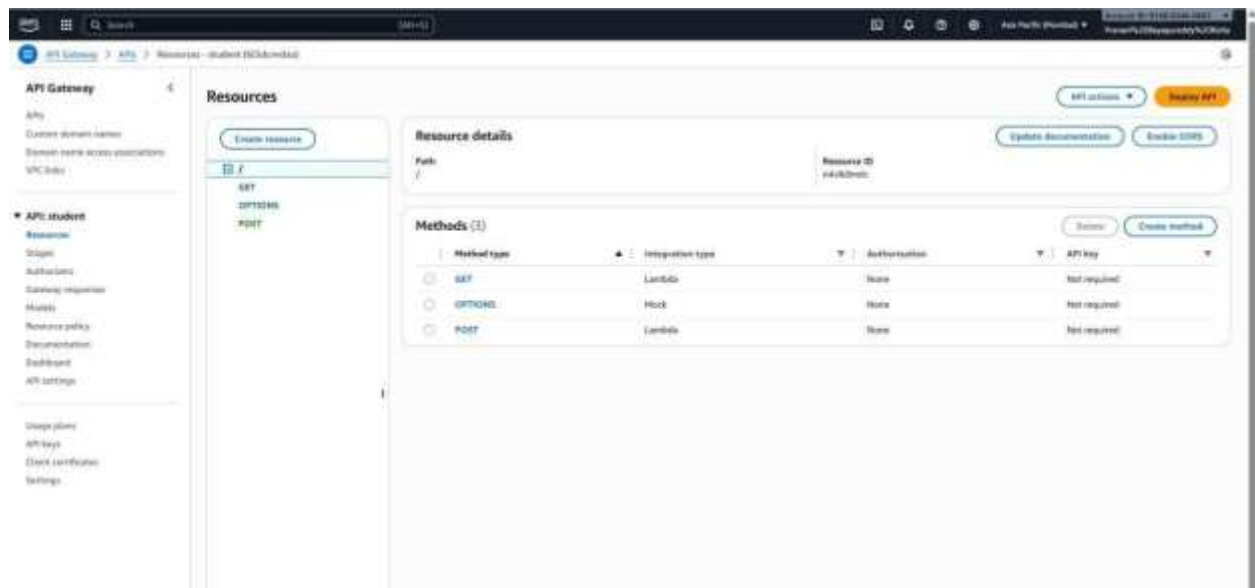
- Navigate to API Gateway in the AWS Console.
- Click Create API → REST API → New API.
- Name it student and choose Edge-Optimized endpoint type for global access.

The screenshot shows the AWS API Gateway console's 'Create REST API' wizard. The 'API details' section is the first step, where users can choose to 'New API', 'Import API', 'Clone existing API', or 'Example API'. The 'API name' field is filled with 'student'. The 'API endpoint type' is set to 'Regional'. The 'IP address type' is set to 'IPv4'. The 'Description - optional' field is empty. The 'Cancel' and 'Create API' buttons are at the bottom right.

## 2.Create API Methods

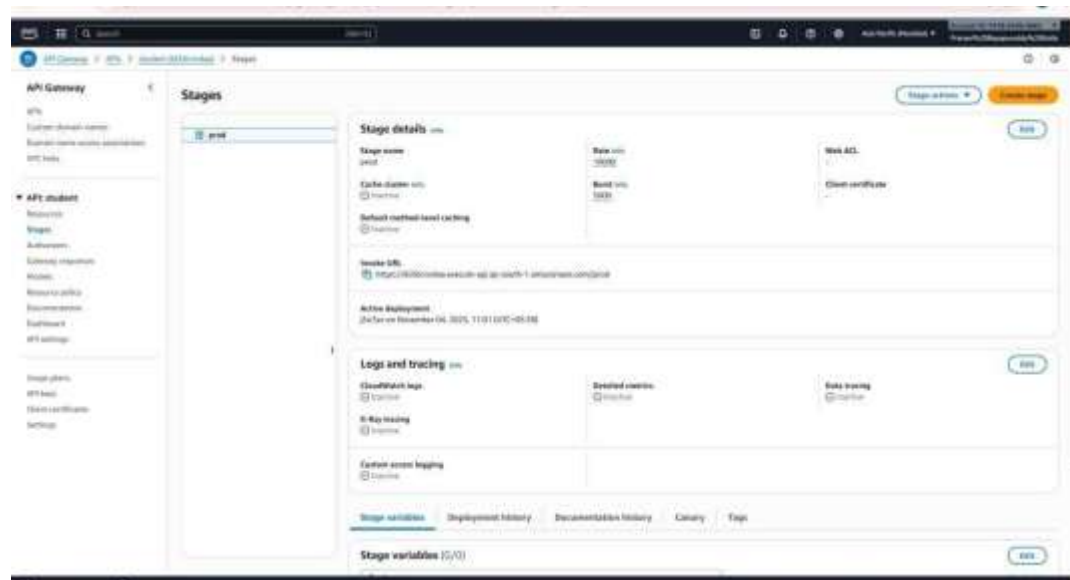
- Add a **GET** method and integrate it with the getStudent Lambda function.
- Add a **POST** method and integrate it with the insertStudentData Lambda function.
- Test both methods using the **Test** option in API Gateway to confirm successful responses.





### 3. Deploy API & Configure CORS

- The API was deployed to a New Stage named prod.
- This generated the public Invoke URL(<https://6l3dcnvdaa.executeapi.ap-south-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-Allow-Origin: \* headers.

## 4.Configure S3 Bucket for Website Hosting

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

Account ID: [redacted] | Region: us-east-1

Buckets > Create bucket

## Create bucket <sup>info</sup>

Buckets are containers for data stored in S3.

### General configuration

**APIs** <sup>info</sup>

Asia Pacific (Mumbai) <sup>api-south-1</sup>

**Bucket type** <sup>info</sup>

☒ **General purpose**  
Recommended for most use cases with access patterns. General-purpose buckets are the original S3 bucket type. They allow a mix of storage classes that automatically store objects as cost-effective durability zones.

☐ **Durability**  
Recommended for low-object-key counts. These buckets can only use S3 Express One Zone storage class, which provides faster processing of data within a single Availability Zone.

**Bucket name** <sup>info</sup>

demo-masterbucket-2025

A bucket must meet 3 to 63 character limit under the global namespace. Bucket names must also begin and end with a letter or number. Buckets can't have a + sign, periods (.), or hyphens (-). [Learn more](#).

**Copy settings from existing bucket - optional**

Select the bucket settings to copy in the following configuration you created:

Version: 2.0 / Buckets / info

### Object Ownership <sup>info</sup>

Control ownership of objects written to this bucket from other AWS accounts and the use of access control lists (ACLs). Object ownership determines who can specify access to objects.

**Object Ownership**

☒ **ACLs disabled (recommended)**  
All objects in this bucket are owned by this account. Access to this bucket and its objects is specified using only policies.

☐ **ACLs enabled**  
Objects in this bucket can be owned by either IAM accounts, users in this bucket, and its objects can be specified using ACLs.

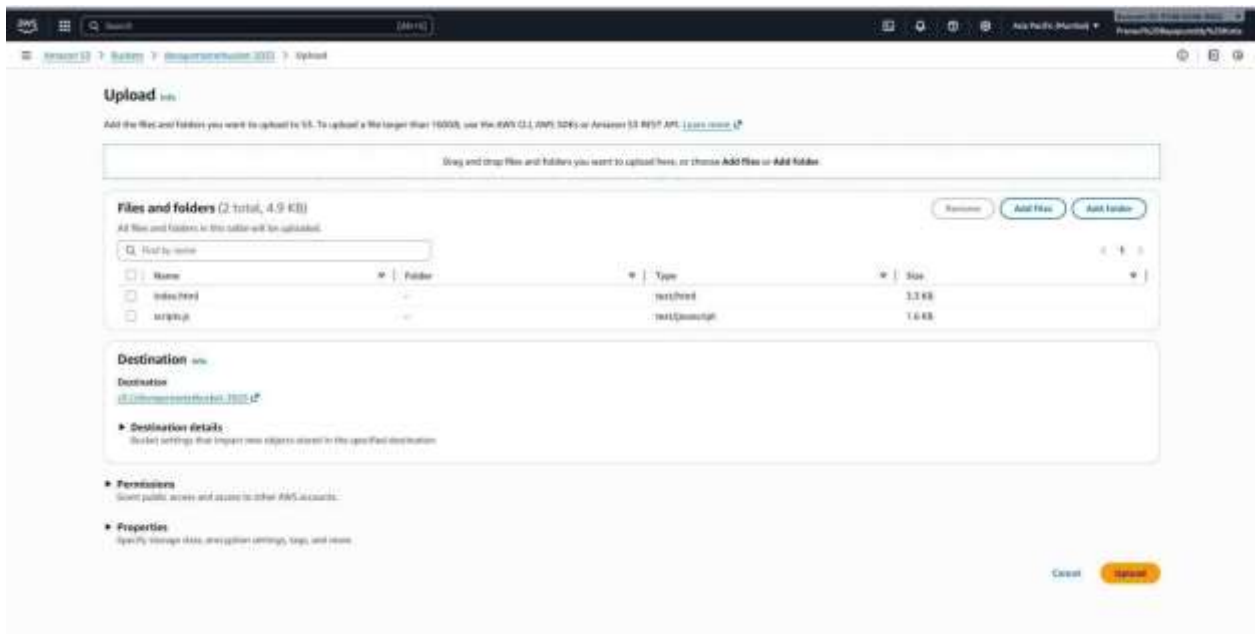
**Object Ownership**

Bucket owner enforced

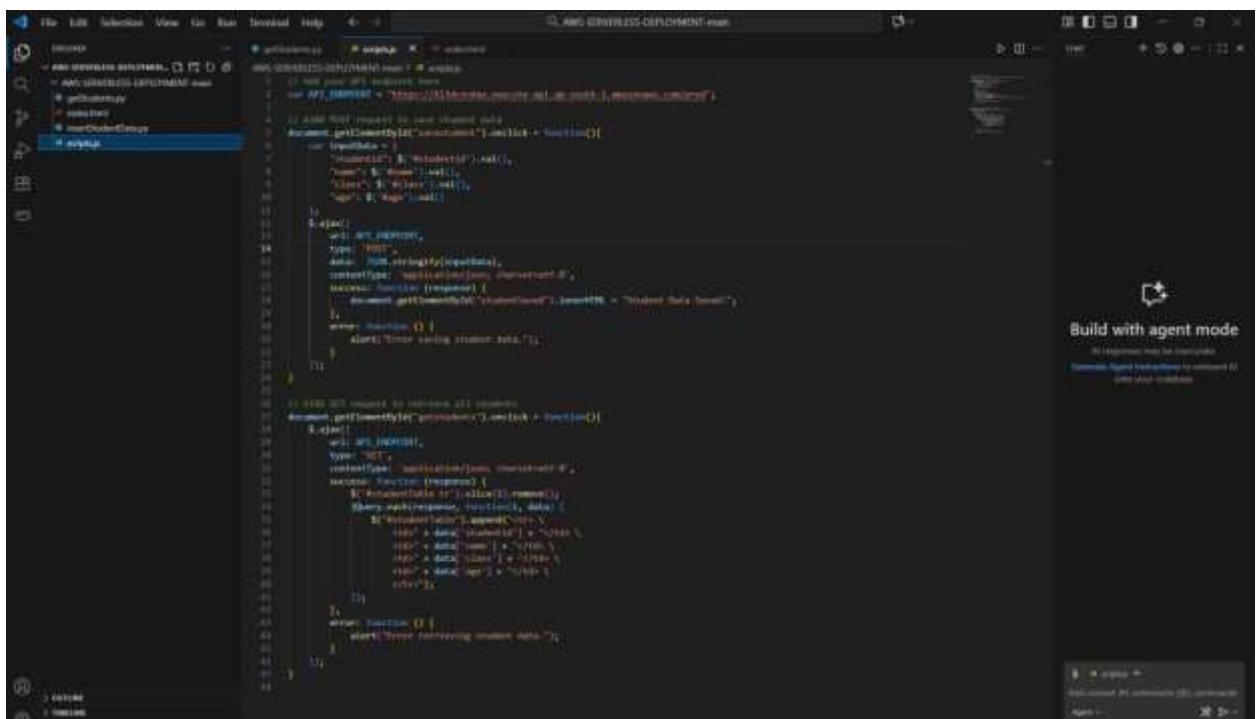
### Block Public Access settings for this bucket

Public access is granted to buckets and objects through access control lists (ACLs), bucket policies, access point policies, or all. In order to ensure that public access to this bucket and its objects is blocked, turn on Block all public access. These settings

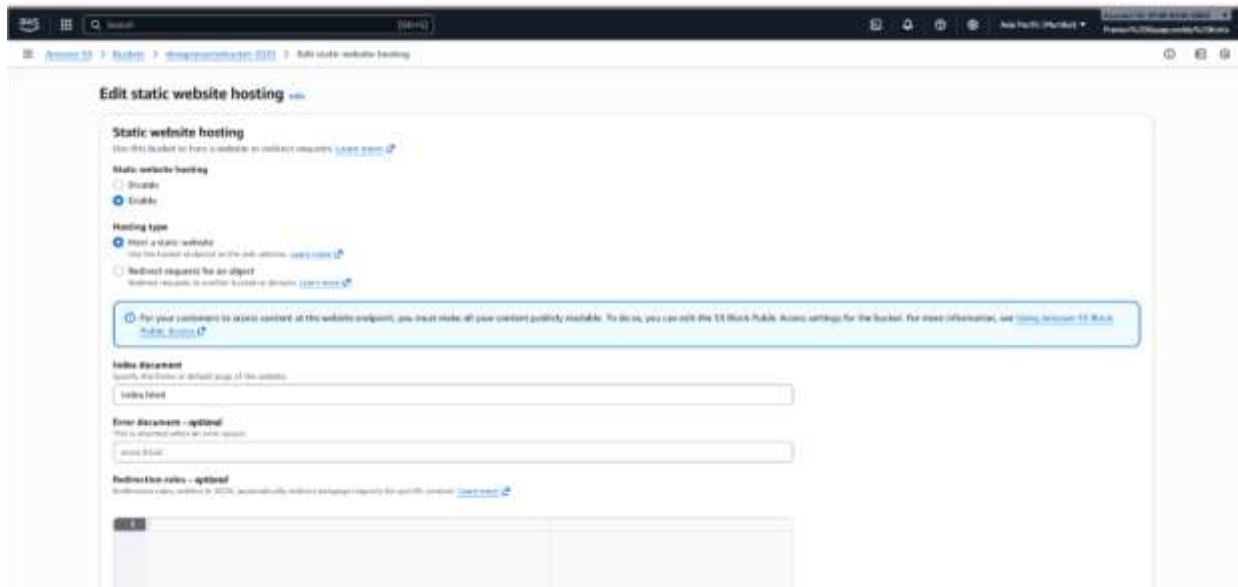
- The frontend files (scripts.js, index.html) were uploaded to the bucket.



- 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": "\*").

Type of Policy

S3 Bucket Policy

Step 2: Add statement(s)

A statement is the formal description of a single permission. See a description of elements [that you can use in statements](#).

Effect

☒ Allow

☐ Deny

Principal

\*

Use a comma to separate multiple values.

Actions

☐ All Actions (\*)

--Select Actions--

GetObject

Amazon Resource Name (ARN)

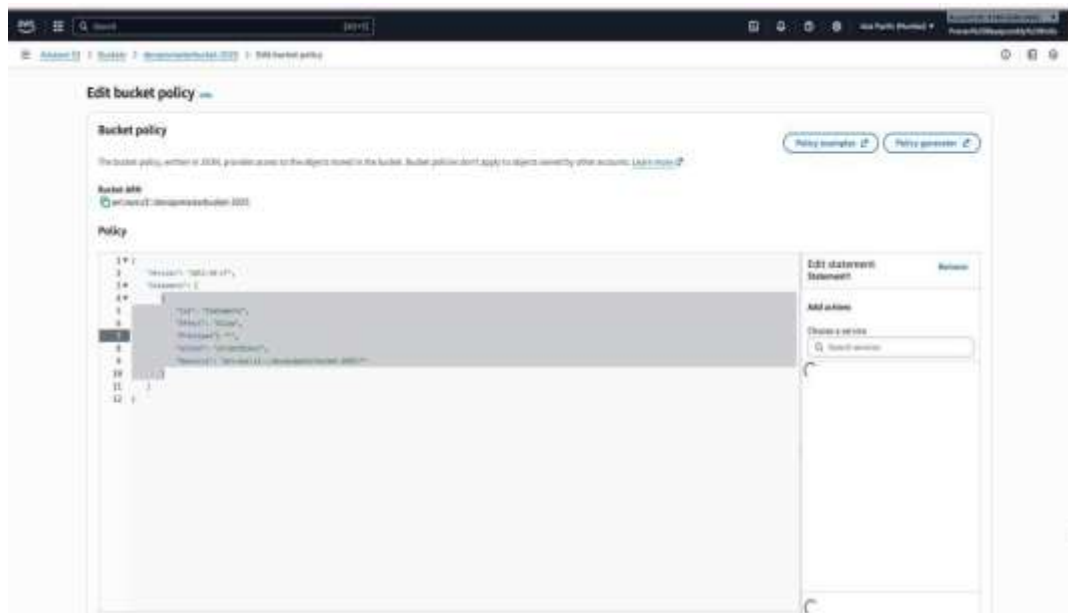
☐ All Resources (\*)

ARNs must follow the following format: arn:aws:s3:::bucketname/objectname. Use a comma to separate multiple values.

\* Add conditions (optional)

Add Statement

Step 3: Generate policy



## 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 ("Jahnvi") was added via the form, which successfully saved and updated the table on the page.

**Save and View Student Data**

Student ID:

Name:

Class:

Age:

Save Student Data

Student Data Saved!

View all Students

Student ID	Name	Class	Age
2	pruthi	9	20
1	Aravind	10	20
8	John	8	11
5(2)	John	10	16
12	pruthi	14	21
10	pruthi	12	19

## DB Table Result:

**studentData**

Scan or query items

Scan ☒ Query ☐

Select a table or index: Table - studentData

Select attribute projection: All attributes

Filters - optional

Run Reset

Completed: Items returned: 6, Items scanned: 6, Efficiency: 100%, RCUs consumed: 2

Table: studentData - Items returned (6)

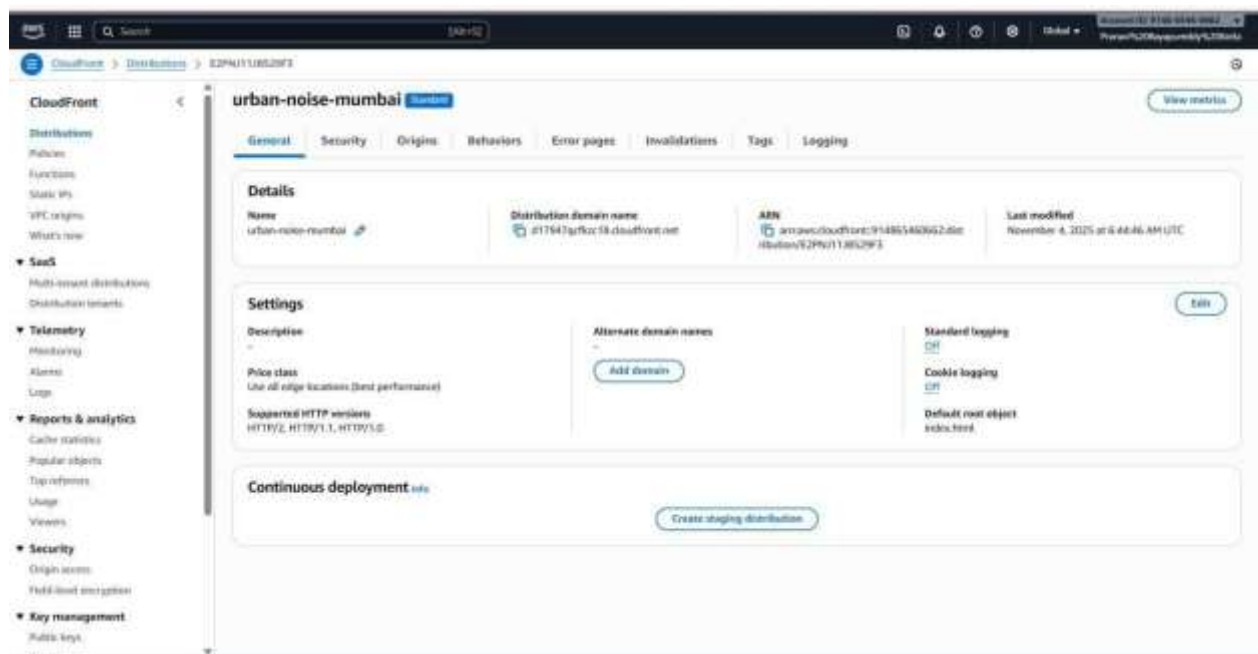
Scan started on November 06, 2021, 23:49:23

studentID (String)	age	class	name
2	20	9	pruthi
1	20	10	Aravind
8	11	8	John
5(2)	16	10	John
12	21	14	pruthi
10	19	12	pruthi

## Part 3: (Enhancement) Securing with CloudFront

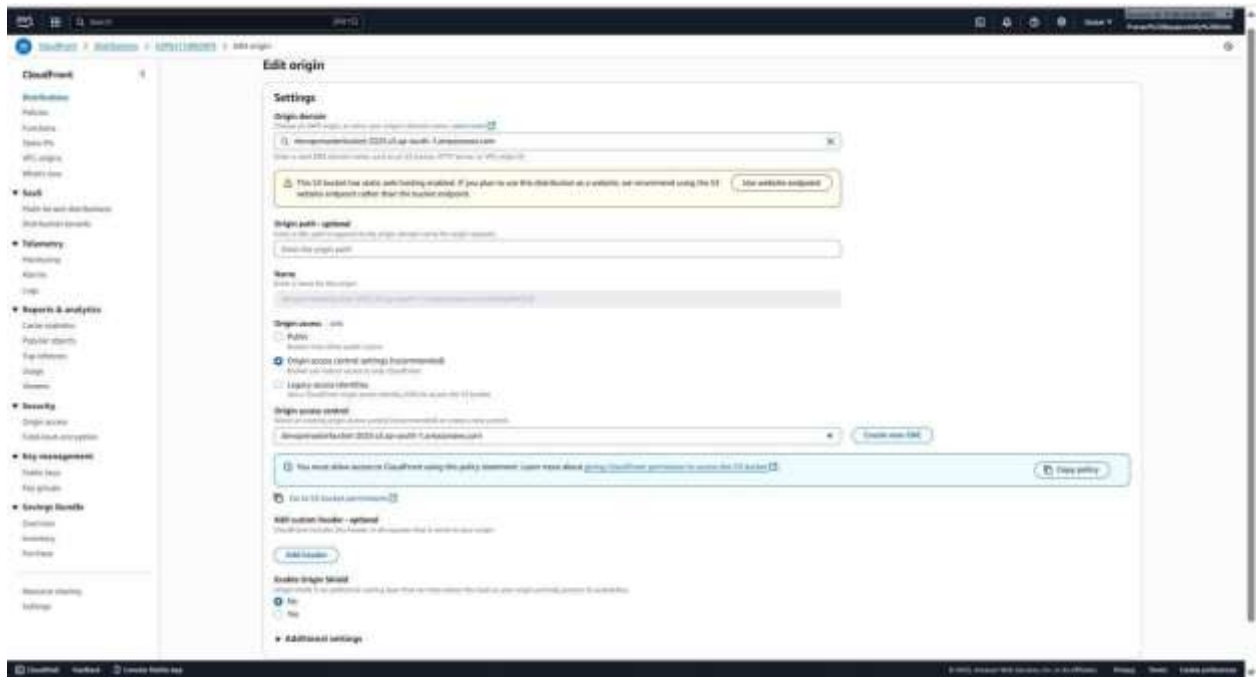
## 1. Create CloudFront Distribution:

- 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).

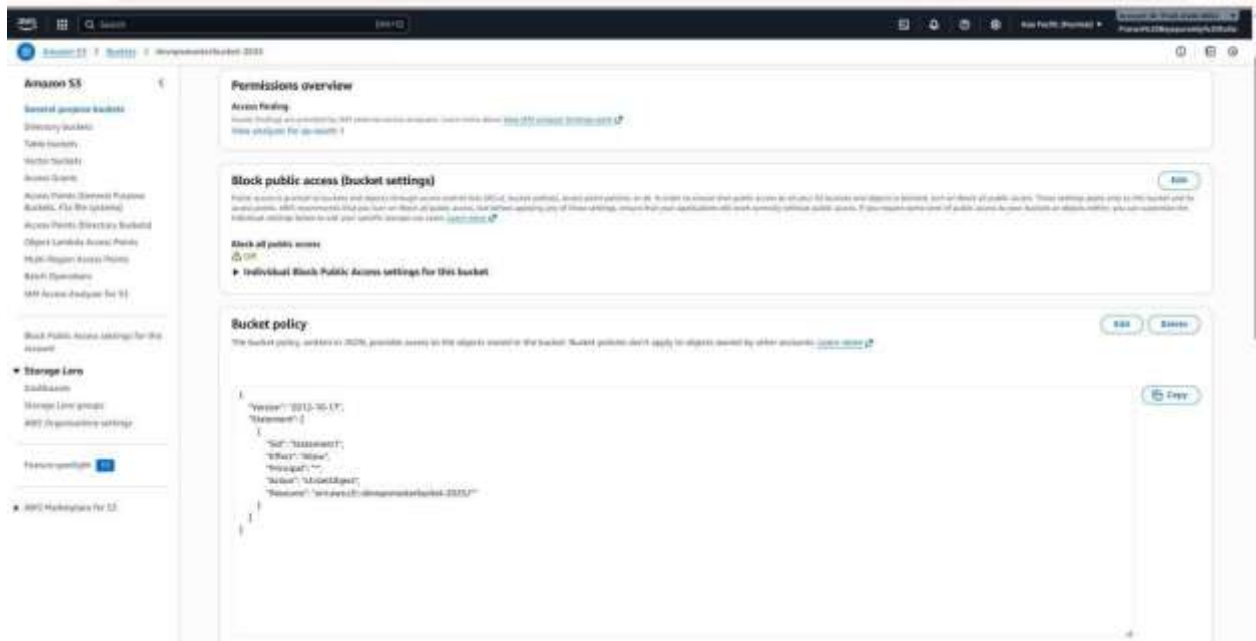


## 2. Secure S3 Bucket:

- 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:







### 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.

