Serverless Blog Engine with AWS DynamoDB and S3

A Course Project Report Submitted in partial fulfillment of the course requirements for the award of grades in the subject of

CLOUD BASED AIML SPECIALITY
(22SDCS07A)

by

M. Sai Sree

(2210030045)

Under the esteemed guidance of

Ms. P. Sree Lakshmi
Assistant Professor,
Department of Computer Science and Engineering

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

K L Deemed to be UNIVERSITY

Aziznagar, Moinabad, Hyderabad,
Telangana, Pincode: 500075

April 2025

K L Deemed to be UNIVERSITY

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

Certificate

This is Certified that the project entitled **"Serverless Blog Engine with AWS DynamoDB and S3"** which is a experimental &/ Simulation work carried out by M. Sai Sree (2210030045), in partial fulfillment of the course requirements for the award of grades in the subject of **CLOUD BASED AIML SPECIALITY**, during the year **2024-2025**. The project has been approved as it satisfies the academic requirements.

Ms.P.Sree Lakshmi

Course Coordinator

Dr. Arpita Gupta

Head of the Department

Ms. P. Sree Lakshmi

Course Instructor

CONTENTS

	Page No.
1. Introduction	1
2. AWS Services Used as part of the project	2
3. Steps involved in solving project problem statement	5
4. Stepwise Screenshots with brief description	7
5. Learning Outcomes	13
6. Conclusion	14
7. References	15

1. INTRODUCTION

The increasing adoption of serverless computing has paved the way for building highly scalable, cost-efficient, and low-maintenance web applications. Among its various use cases, deploying a blog engine in a serverless architecture is both practical and insightful. By leveraging services such as Amazon DynamoDB and Amazon S3, developers can build a dynamic yet serverless blog platform without managing backend infrastructure.

Amazon S3 serves as a reliable and scalable storage service for static assets including HTML, CSS, JavaScript, and media files. It also supports static website hosting, enabling developers to deliver web content directly from an S3 bucket to users' browsers [1]. Amazon DynamoDB, a fully managed NoSQL database, stores structured data like blog posts, metadata, user profiles, and comments. It offers low-latency access with built-in scalability and fault tolerance [2].

To handle dynamic content and HTTP requests, AWS Lambda functions are employed as backend processors. These functions execute code in response to events such as user interactions or API calls, and they scale automatically to meet demand [3]. Integration with Amazon API Gateway allows the application to expose RESTful APIs, enabling users to read, write, and interact with blog content through standard HTTP methods [4].

2. AWS SERVICES USED AS PART OF THE PROJECT

In this project, several AWS services were utilized to build a fully serverless blog engine. Each service played a specific role in enabling seamless content delivery, data management, backend processing, and user interaction. Below are the key AWS services used in the project:

Amazon S3 (Simple Storage Service)

Amazon S3 is used to store application versions, static assets, and logs [1].

- Used to host static website content (HTML, CSS, JavaScript, images).
- Provides scalable and durable object storage.
- Supports static website hosting with custom domains and public access settings

Amazon DynamoDB

Amazon DynamoDB is used as a NoSQL database to store dynamic content such as blog posts, comments, and user metadata [2].

- Offers high availability and automatic scaling.
- Enables fast and consistent read/write performance.
- Supports key-value and document data models.

AWS Lambda

AWS Lambda runs backend application logic in response to API requests without provisioning or managing servers [3].

- Automatically scales with demand.
- Executes code in response to events (e.g., HTTP requests, DynamoDB updates).
- Enables pay-per-use billing with zero idle time cost.

AWS IAM (Identity and Access Management)

AWS IAM provides secure access management to AWS resources .

- Defines roles and policies for secure service interactions.
- Controls access to S3, Lambda, API Gateway, and DynamoDB.
- Ensures the principle of least privilege is enforced.

Amazon API Gateway

Amazon API Gateway is used to expose RESTful APIs that connect frontend clients to backend Lambda functions [4].

- Supports creating, deploying, and monitoring APIs.
- Handles request validation, throttling, and authorization.
- Seamlessly integrates with Lambda and DynamoDB.

3. STEPS INVOLVED IN SOLVING PROJECT PROBLEM STATEMENT

1. **Requirement Analysis and System Design**
   The initial phase involved understanding the core functionality of the blog engine, including blog creation, updates, and retrieval. Based on these requirements, a serverless architecture was selected for its scalability and cost-efficiency, using services like Amazon S3, AWS Lambda, Amazon API Gateway, and Amazon DynamoDB.
2. **Static Website Hosting Using Amazon S3**
   An S3 bucket was configured to host the frontend assets of the application (HTML, CSS, JavaScript). Static website hosting was enabled, and access permissions were set to allow public viewing. This setup allows the blog interface to be accessed via a browser using an S3 endpoint or custom domain [2].
3. **Database Setup with Amazon DynamoDB**
   A DynamoDB table was created to store blog-related data such as titles, content, timestamps, and author info. On-demand capacity mode was chosen for auto-scaling based on traffic. Proper schema design and indexing were applied following AWS best practices for performance and scalability [3][4].
4. **Backend Logic with AWS Lambda**
   AWS Lambda functions were implemented to handle backend operations like adding, updating, retrieving, and deleting blog posts. Each function was triggered by specific HTTP requests and configured with environment variables and IAM roles for secure access to DynamoDB [1].
5. **API Management via Amazon API Gateway**
   RESTful endpoints were created using Amazon API Gateway. These endpoints routed HTTP requests (GET, POST,) to corresponding Lambda functions. CORS was enabled to support browser-based requests, and multiple deployment stages (e.g., dev, prod) were configured [5][6].
6. **Access Control with AWS IAM**
   IAM roles and policies were configured to ensure secure access across all services. Lambda functions were granted only necessary permissions to interact with

DynamoDB and API Gateway. Security was implemented using the principle of least privilege to minimize risk [7].

7. **Testing, Monitoring, and Deployment**

   The full stack was tested using Postman and browser tools to ensure API endpoints and frontend functionality worked as expected. Amazon CloudWatch was used to monitor logs and function performance. Finally, the static assets were uploaded to the S3 bucket for public availability and production deployment.

4. STEPWISE SCREENSHOTS WITH BRIEF DESCRIPTION

Step 1: NAVIGATE TO IAM

To create a new user in AWS IAM (Identity and Access Management), start by navigating to the IAM section from the AWS Console Home. Begin the process by creating a new user and entering the desired user name. Optionally, you can enable console access if you want the user to log in to the AWS Management Console. Be sure to choose "IAM User" rather than "Identity Center." Next, select the option to attach policies directly, or alternatively, you can add the user to a group. Attach the required policies, specifically DynamoDBFullAccess and AmazonS3FullAccess. After reviewing the configuration and verifying that the correct policies are attached, proceed to create the user. Once completed, the user will be successfully created.

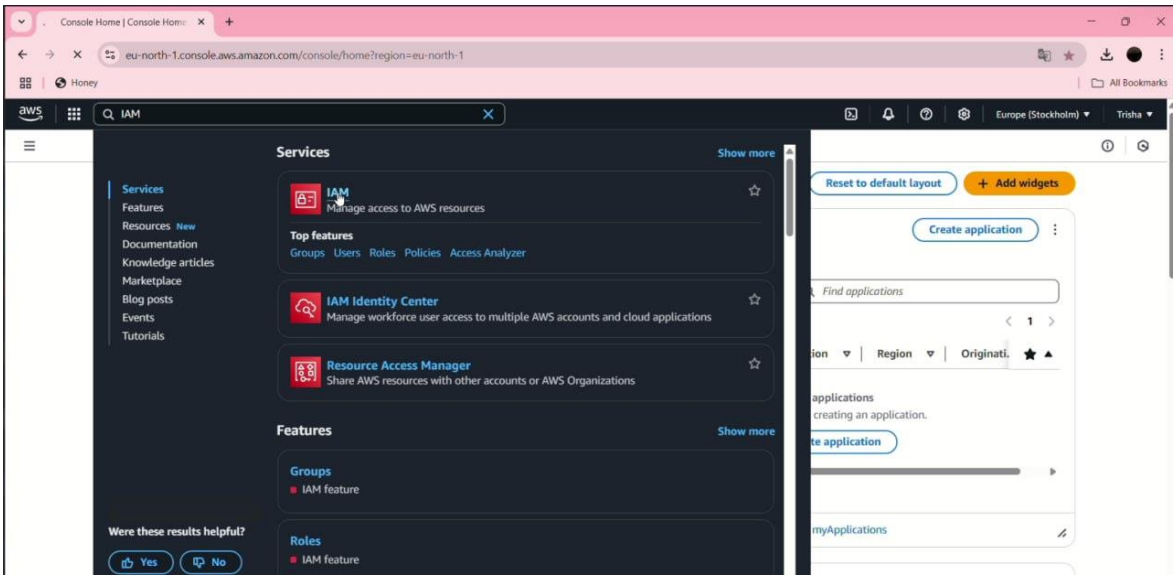


Fig.4.1 Navigate to IAM

These are the roles to select for this project and name it serverless are select create role

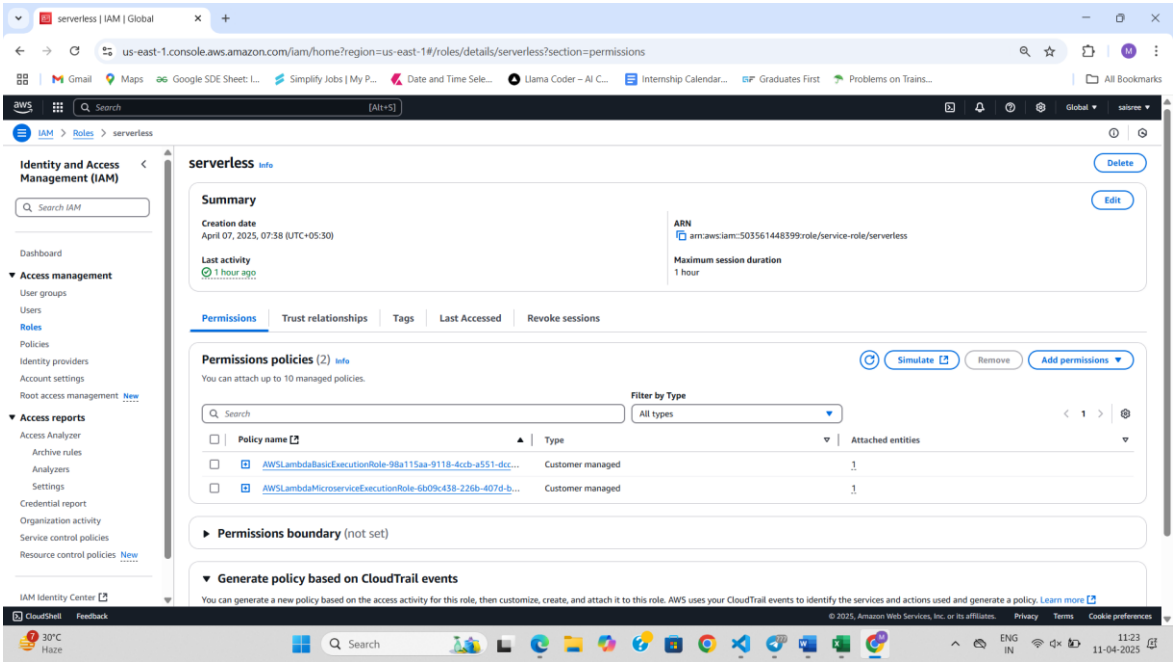


Fig.4.2 Roles Created

Step 2:create s3 bucket

A new Amazon S3 bucket was created with a globally unique name. The bucket serves as a container to store all the frontend resources such as HTML, CSS, JavaScript, images, and other static files**.**

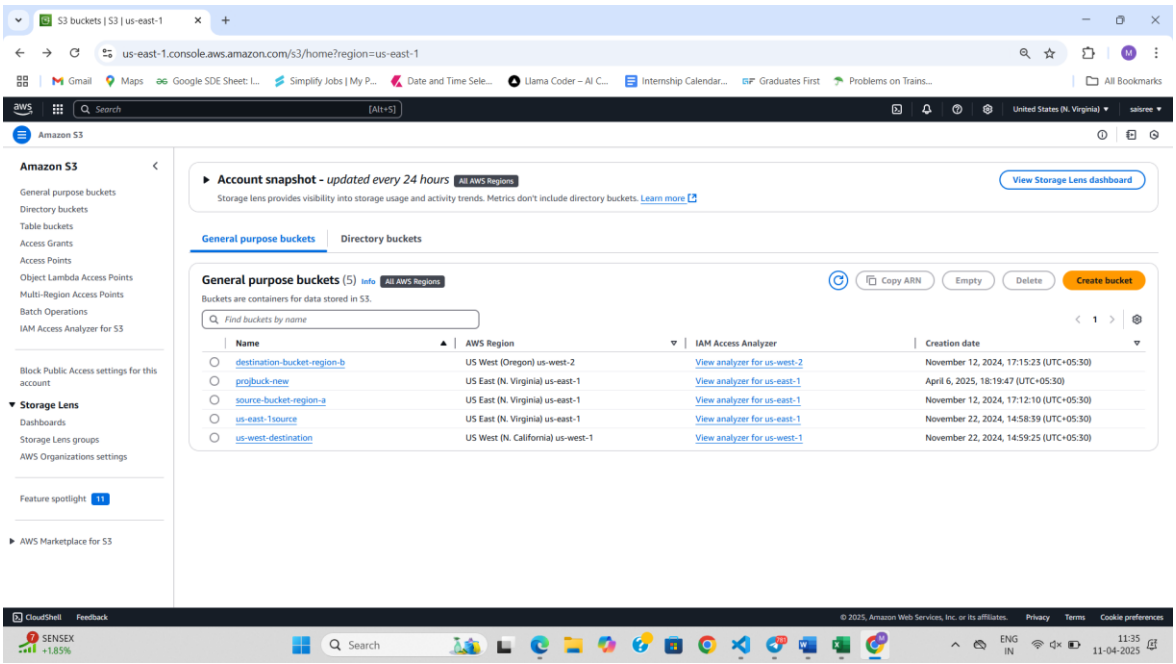


Fig.4.3 create s3 bucket

Step 3: upload file into s3

All static assets for the blog (e.g., index.html, style.css, script.js, etc.) were uploaded into the S3 bucket using the AWS Management Console or AWS CLI. These files define the structure, style, and behavior of the blog's frontend

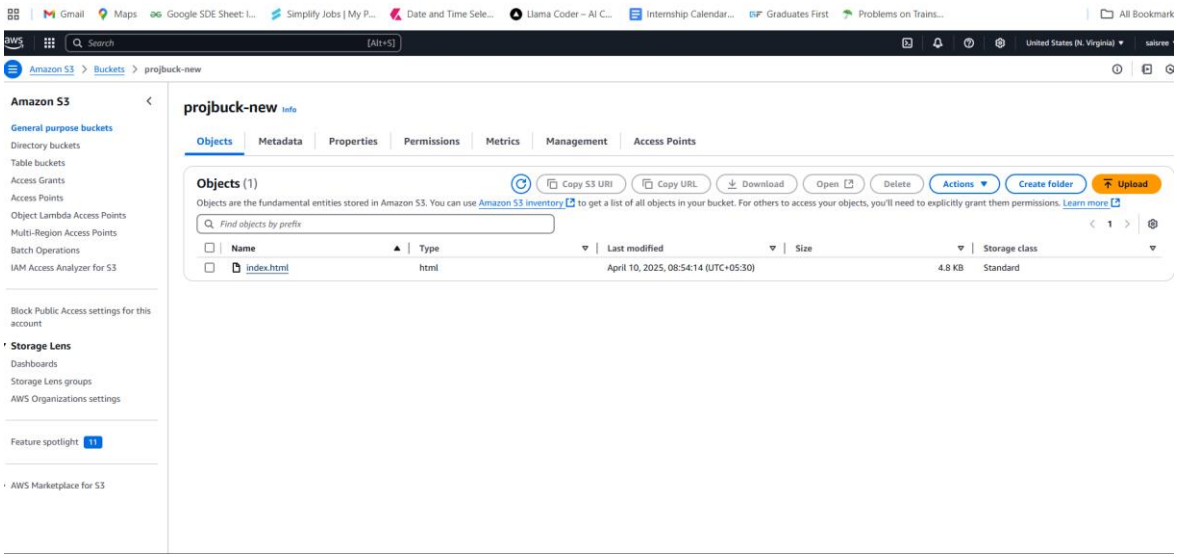


Fig 4.4 upload file

STEP 5: ENABLE STATIC WEBSITE HOSTING

The "Static website hosting" feature in S3 was enabled to allow the bucket to serve web pages directly to users via HTTP. An index document (typically index.html) was specified to act as the entry point, and an error document (like error.html) was defined to handle invalid or broken routes.

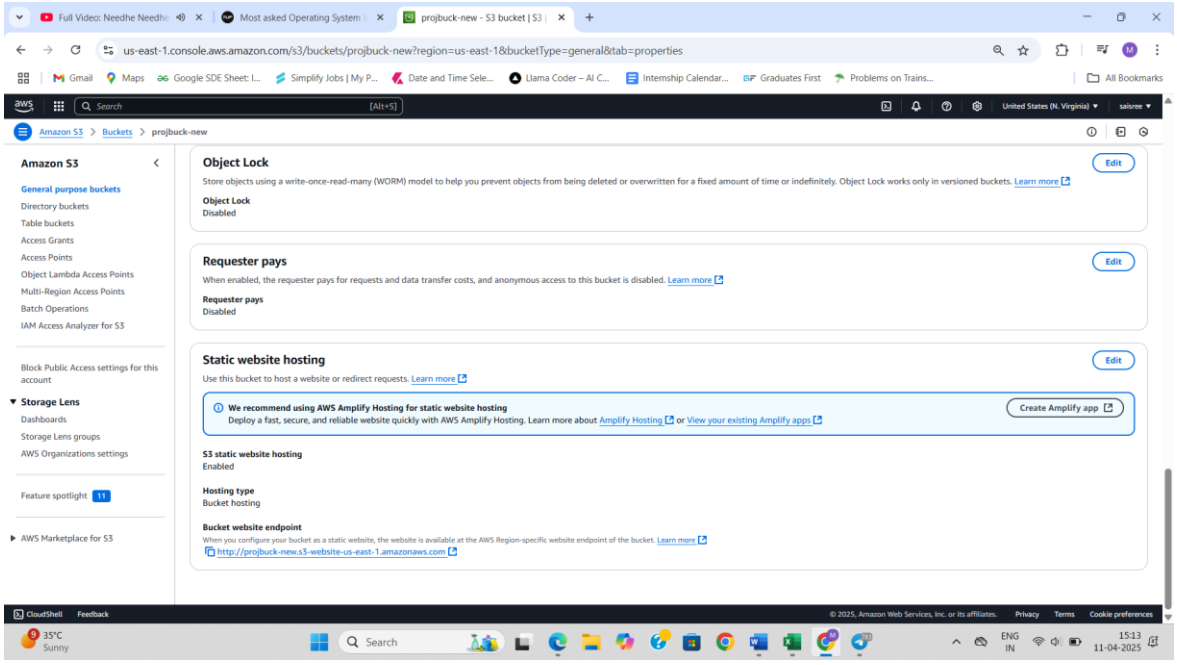


Fig 4.5 unable static website hosting

Step 6: write permission configuration

Bucket policy and object-level permissions were set to make the files publicly accessible, ensuring that end-users can access the blog without authentication. This involved modifying the bucket's Block Public Access settings and adding a custom **bucket policy** allowing s3:GetObject access for all users.

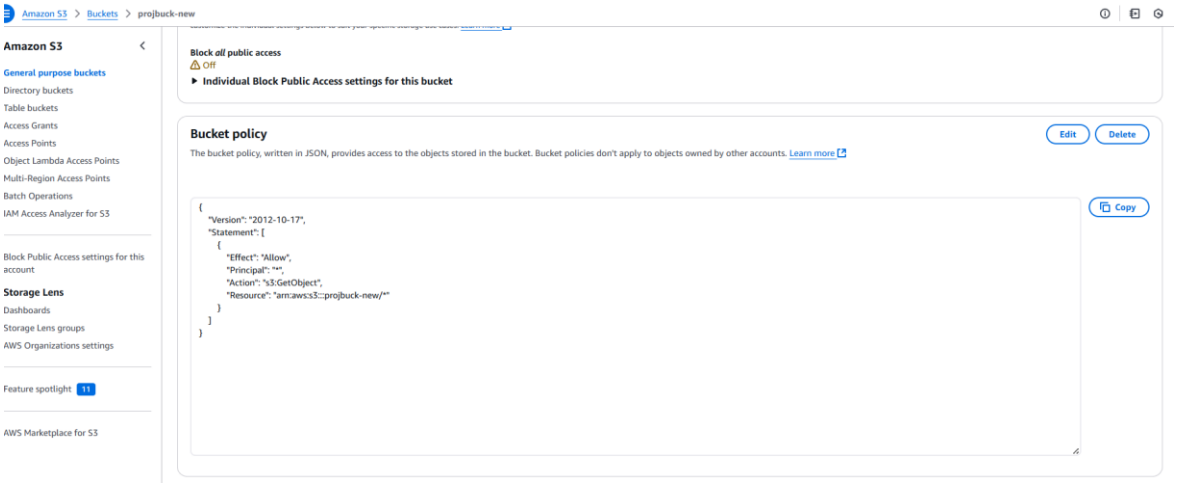


Fig 4.6 permission configuration

Step 7:Create database table

Created a DynamoDB table to store blog posts, with a primary key on msg

Chose on-demand capacity mode for automatic scaling.

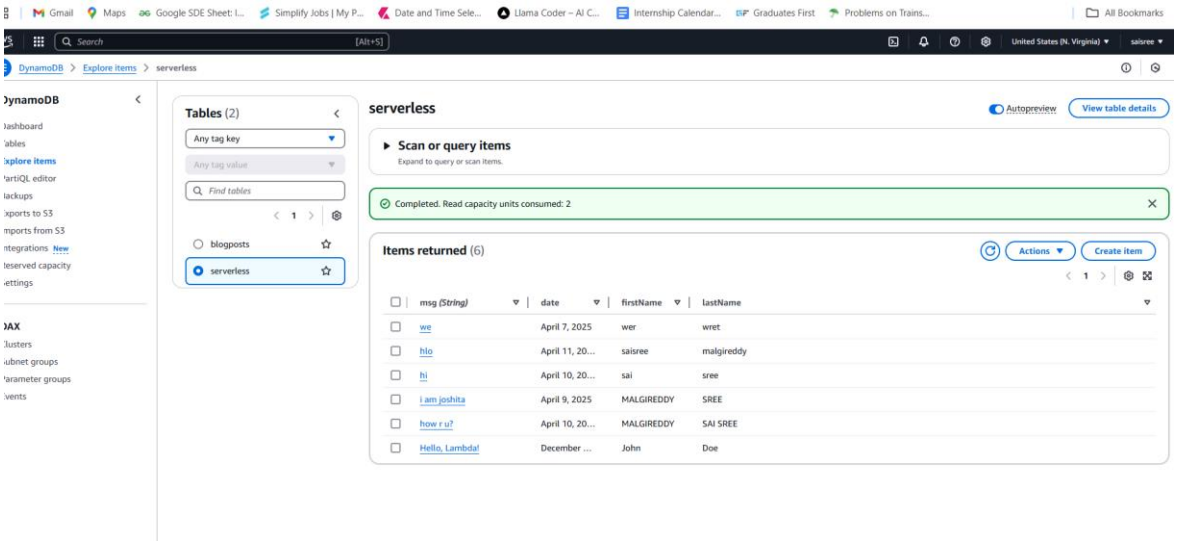


Fig 4.7 create database table

STEP 8: write lambda get and post function

Create Post: Accepts a blog post (title, content, author) from the frontend and adds it as a new entry in the DynamoDB table.

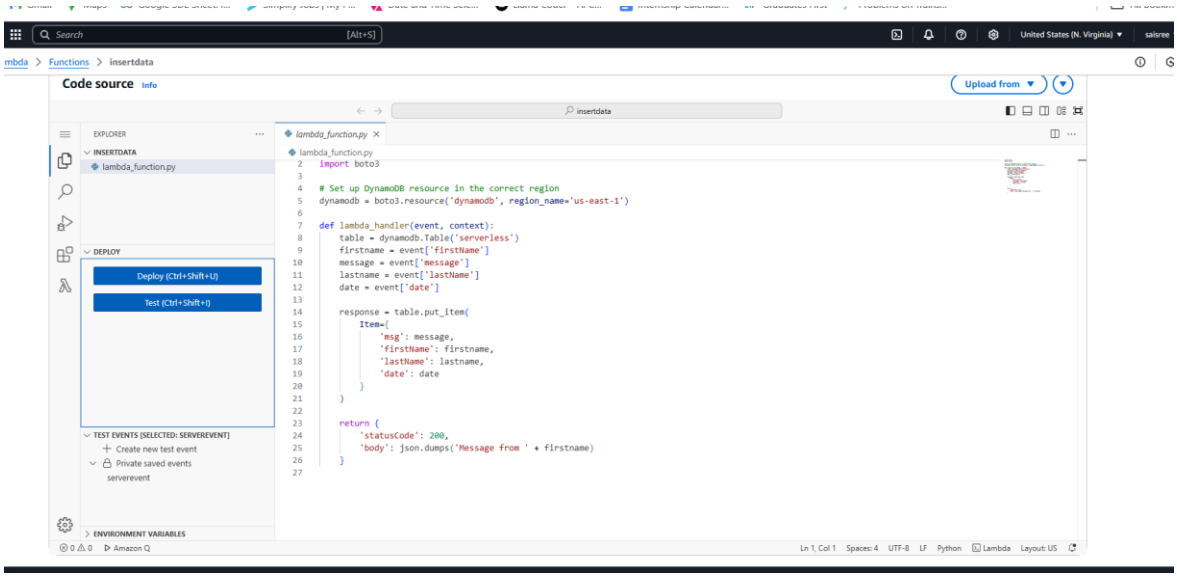


Fig 4.8 lambda post_function

Get Posts: Retrieves all blog posts from DynamoDB and returns them as a JSON response to the frontend.

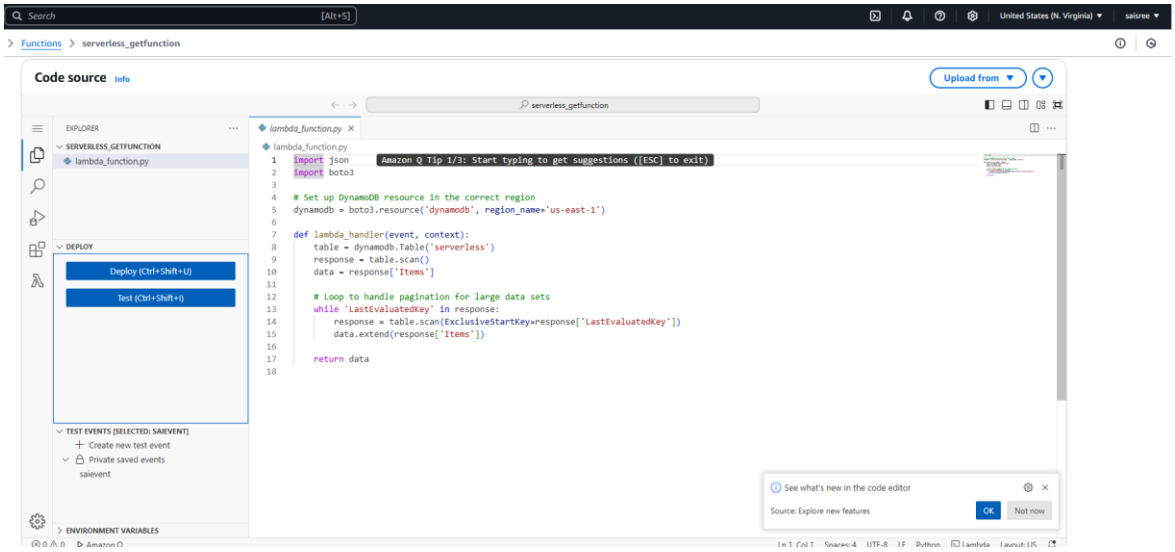


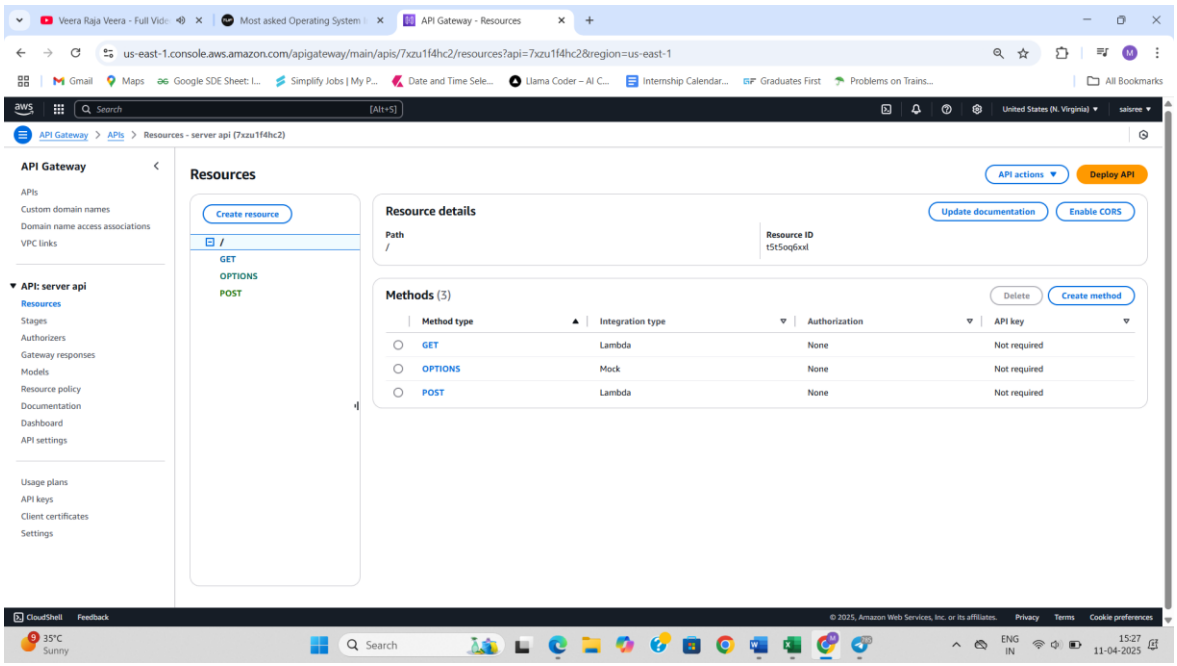
Fig 4.9 lambda get_function

Step 9: Create REST API

Set up a REST API using API Gateway to handle HTTP requests.

Created resources like /posts and mapped them to Lambda functions.

Enabled CORS to allow requests from the S3-hosted frontend.



4.10 creating REST API

Step 10:output

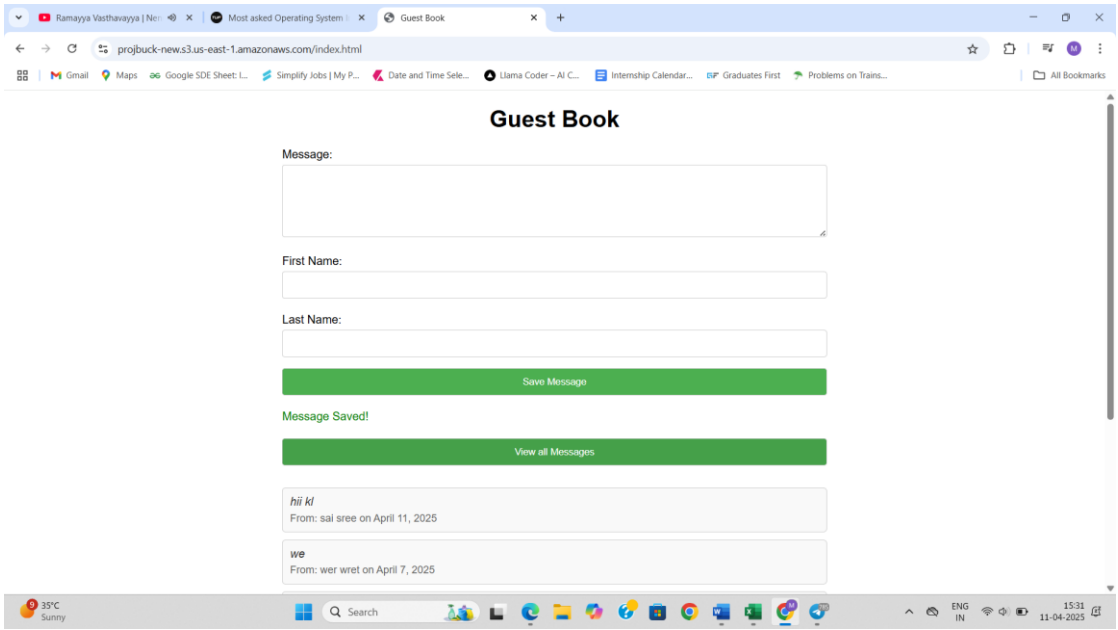


Fig 4.11 output

5.LEARNING OUTCOMES

1. **Understanding Serverless Architecture**

Learners gained a deep understanding of the serverless paradigm, where application components are decoupled and dynamically managed without the need for dedicated server infrastructure. This enhances scalability, reduces cost, and promotes agility in deployment [1].

2. **Static Web Hosting with Amazon S3**

The project demonstrated how Amazon S3 can be used to host static websites, including HTML, CSS, and client-side JavaScript. S3's built-in website hosting capabilities simplify the process of deploying web content globally [2].

3. **Designing Scalable NoSQL Databases with DynamoDB**

Hands-on experience with Amazon DynamoDB provided insights into schema-less database design, efficient partitioning, indexing strategies, and best practices for high-performance applications [3][4].

4. **Implementing REST APIs with API Gateway**

The project introduced learners to building and deploying RESTful APIs using Amazon API Gateway, including configuring resources, methods, stages, and integrating with Lambda functions for backend processing [5].

5. **Event-Driven Processing with AWS Lambda**

AWS Lambda was used to handle backend logic in response to user actions, HTTP requests, or changes in data. This included learning how to write, deploy, and test functions and manage runtime environments without provisioning servers [6].

6. **Security and Access Control in AWS Services**

Students explored how to secure AWS resources using IAM roles, policies, and API keys. This was critical in controlling access to S3 buckets, Lambda functions, and database operations securely [7].

6.CONCLUSION

The development of a serverless blog engine using AWS services such as Amazon S3, DynamoDB, Lambda, and API Gateway showcases the strength and flexibility of modern cloud-native architectures. By decoupling components and adopting an event-driven model, this approach eliminates the need for traditional server management, significantly reduces operational overhead, and enhances scalability and fault tolerance.

Amazon S3 ensures fast and reliable delivery of static content, while DynamoDB provides a high-performance, low-latency datastore for dynamic content. Lambda functions, triggered by API Gateway endpoints, enable efficient handling of user interactions without persistent backend servers. This architecture not only streamlines deployment and maintenance but also aligns well with DevOps practices and pay-as-you-go cost models.

Overall, the serverless paradigm enables rapid application development, simplified scaling, and improved resource utilization, making it an ideal choice for building lightweight, highly available web applications like a blog engine. This project highlights how developers can harness the power of AWS to build robust, secure, and efficient applications with minimal infrastructure complexity.

7.REFERENCES

[1] Amazon S3 User Guide:
https://docs.aws.amazon.com/AmazonS3/latest/userguide/Welcome.html

[2] Amazon DynamoDB Developer Guide:
https://docs.aws.amazon.com/amazondynamodb/latest/developerguide/Welcome.html

[3] AWS Lambda Developer Guide:
https://docs.aws.amazon.com/lambda/latest/dg/welcome.html

[4] Amazon API Gateway Developer Guide:
https://docs.aws.amazon.com/apigateway/latest/developerguide/welcome.html

[5] Hosting a Static Website on Amazon S3:
https://docs.aws.amazon.com/AmazonS3/latest/userguide/WebsiteHosting.html

[6] Set Up API Gateway with Lambda Integration:
https://docs.aws.amazon.com/apigateway/latest/developerguide/apigateway-getting-started-with-rest-apis.html

[7] Best Practices for DynamoDB:
https://docs.aws.amazon.com/amazondynamodb/latest/developerguide/best-practices.html