Cloud Based Voting System using API Gateway and Lambda

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

Karra Sri Charan Reddy(2210030455)

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 **Cloud Based Voting System Using API Gateway and Lambda** which is a experimental & theoretical work carried out by Karra Sri Charan Reddy(2210030455) 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

1. INTRODUCTION

In the digital era, cloud computing has transformed various industries by providing scalable, secure, and cost-effective solutions. One of the critical areas that can benefit from these advancements is the voting system, where traditional paper-based or electronic voting methods often face challenges related to security, accessibility, and scalability. To address these issues, our project introduces a Cloud-Based Voting System, leveraging AWS services such as API Gateway, AWS Lambda, and DynamoDB to create a secure, scalable, and efficient voting mechanism.

The system is designed to ensure integrity, transparency, and accessibility in electoral processes. The API Gateway acts as the entry point, efficiently handling API requests while enforcing authentication and authorization policies. It ensures that only legitimate users can interact with the system, preventing unauthorized access. AWS Lambda, a serverless compute service, executes business logic dynamically in response to API requests. This eliminates the need for server management and allows the system to scale automatically based on demand. DynamoDB, a highly available NoSQL database, securely stores voting records, ensuring low-latency access and real-time updates without performance degradation.

Security is a primary focus of this project. The system incorporates encryption mechanisms, access control, and audit trails to prevent fraud, unauthorized modifications, or tampering. It also reduces the risk of vote duplication and ensures that all votes remain anonymous and verifiable. Additionally, the cloud-native approach eliminates the need for physical voting infrastructure, making the system cost-effective and highly reliable.

This cloud-based solution is particularly beneficial for remote and large-scale elections, ensuring that voters can securely participate from anywhere while maintaining electoral integrity. By utilizing serverless computing and scalable database management, this system demonstrates how cloud technology can modernize voting systems, making them more secure, transparent, and accessible for the future.

2. AWS Services Used as part of the project

1. Amazon API Gateway

- Acts as the entry point for all voting-related API requests.
- Ensures secure communication by enforcing authentication and authorization policies.
- Handles request validation, rate limiting, and traffic management, ensuring high availability and scalability.

2. AWS Lambda

- A serverless compute service that executes voting-related functions dynamically.
- Handles tasks such as vote submission, validation, counting, and result generation.
- Eliminates the need for dedicated servers, reducing costs while automatically scaling based on demand.

3. Amazon DynamoDB

- A NoSQL database used for securely storing voter information, casted votes, and election results.
- Provides high availability, low-latency access, and automatic scaling to handle large volumes of voting data.
- Ensures data integrity with built-in encryption and access controls.

3.Steps involved in solving project problem statement

1. Designing the System Architecture

- The system is based on AWS API Gateway, AWS Lambda, and DynamoDB to ensure scalability, security, and efficiency.
- API Gateway acts as the communication interface, Lambda functions process business logic, and DynamoDB serves as the database.
- A Flask-based backend interacts with AWS services, and an HTML/Tailwind CSS frontend provides a user interface.

2.Creating the DynamoDB Table (Candidates)

- A DynamoDB table was set up to store candidate details, including:
    - reg_no (Primary Key)
    - name
    - image
    - votes (initialized to 0)
- This table ensures quick retrieval and update of voting data.

3.Developing AWS Lambda Functions

- Submit Vote Function (submit_vote)
    - Reads the candidate's registration number from the request.
    - Increments the vote count in DynamoDB using update_item.
    - Returns the updated vote count.
- Fetch Candidates Function (fetch_candidates)
    - Scans the DynamoDB table and returns a list of candidates.
    - Converts Decimal values to int for proper JSON serialization.

4. Configuring API Gateway

- Created REST API Endpoints:
    - GET /candidates → Fetches candidate details.
    - POST /vote → Records a vote for a candidate.
- Integrated Lambda functions to handle API requests.

5. Setting Up the Backend with Flask

- A Flask server was created to handle API requests from the frontend.
- API endpoints:

  /get_candidates → Fetches the list of candidates from DynamoDB.

  /vote → Submits a vote via API Gateway and updates the vote count.
- The Flask app integrates with AWS API Gateway to send and receive data securely.

6. Testing and Deployment

- Tested the Flask backend by making API requests to check data flow.
- Tested the frontend UI to ensure votes were being counted and displayed correctly.
- Deployed the system with AWS services, ensuring scalability and security.

4. Stepwise Screenshots with brief description

1.AWS DynamoDB Setup

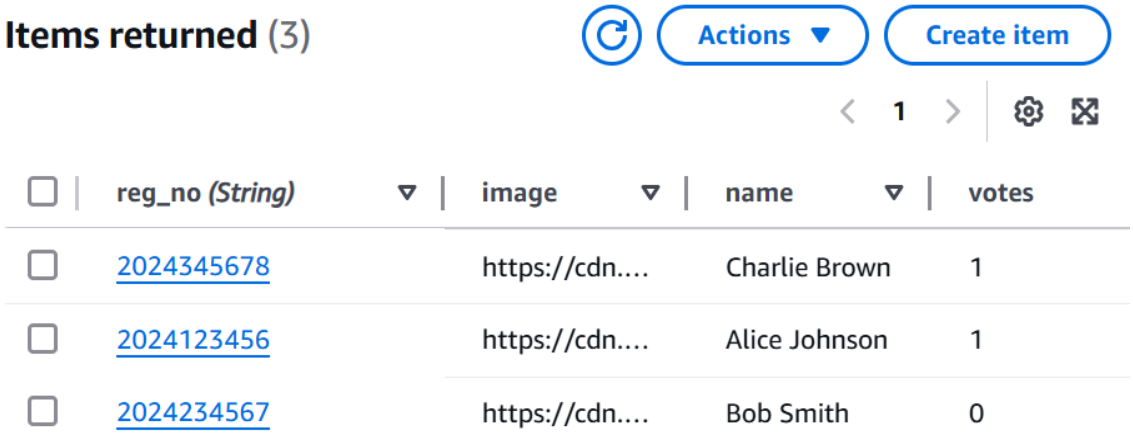

Fig 4.1: Shows the "Candidates" table in DynamoDB containing candidate names, registration numbers, images, and vote counts.

2. AWS Lambda - Fetch Candidates Function

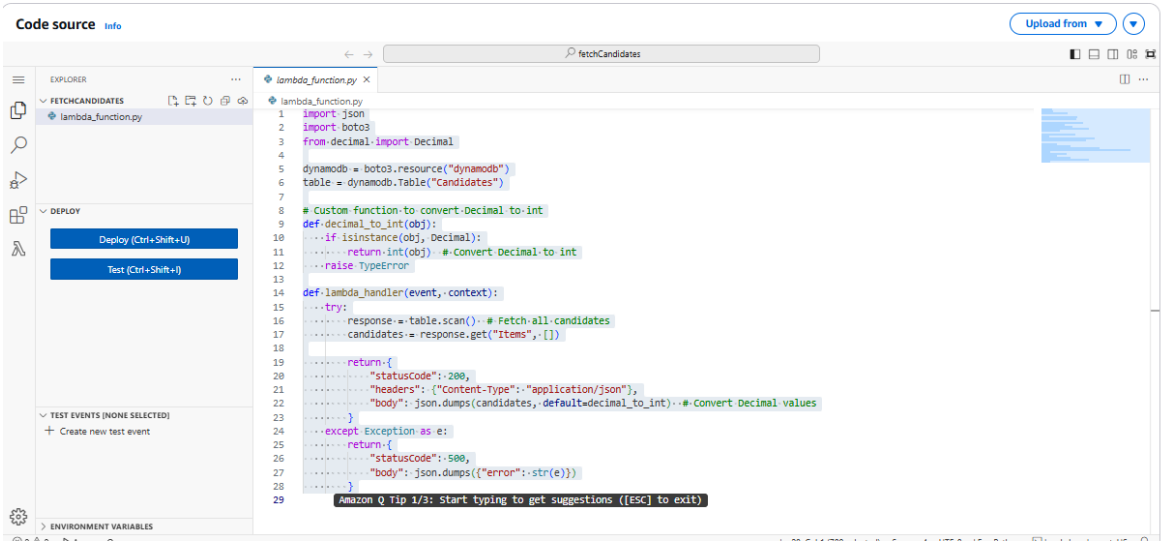

Fig 4.2: Displays the fetch_candidates Lambda function, which retrieves all candidates from the DynamoDB table and returns them as a JSON response.

3. AWS Lambda - Submit Vote Function

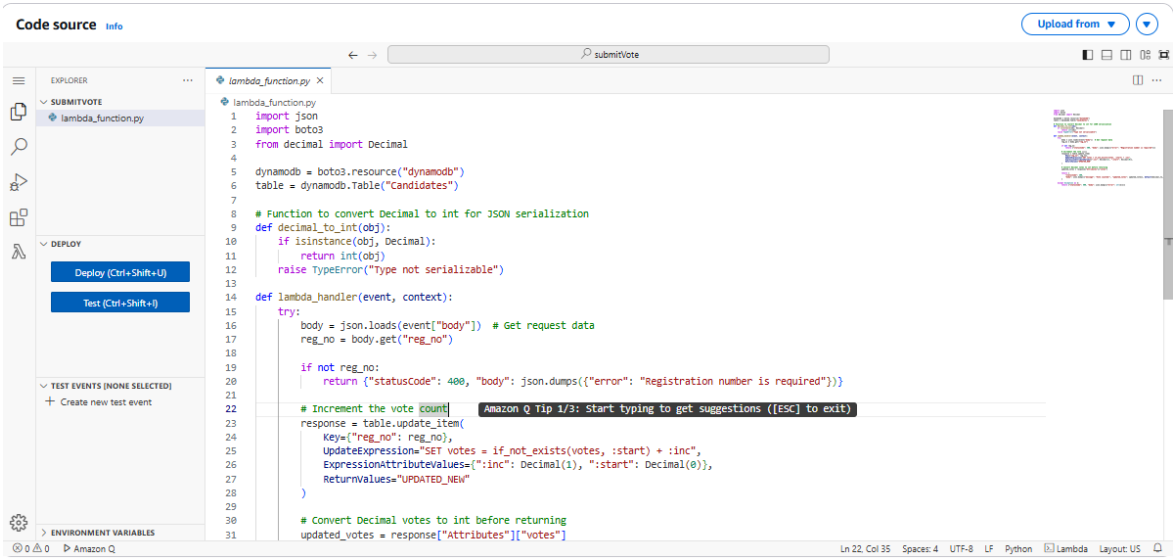

Fig 4.3: Shows the submit_vote Lambda function, which increments the vote count for a selected candidate in DynamoDB.

4.API Gateway - Configured Endpoints

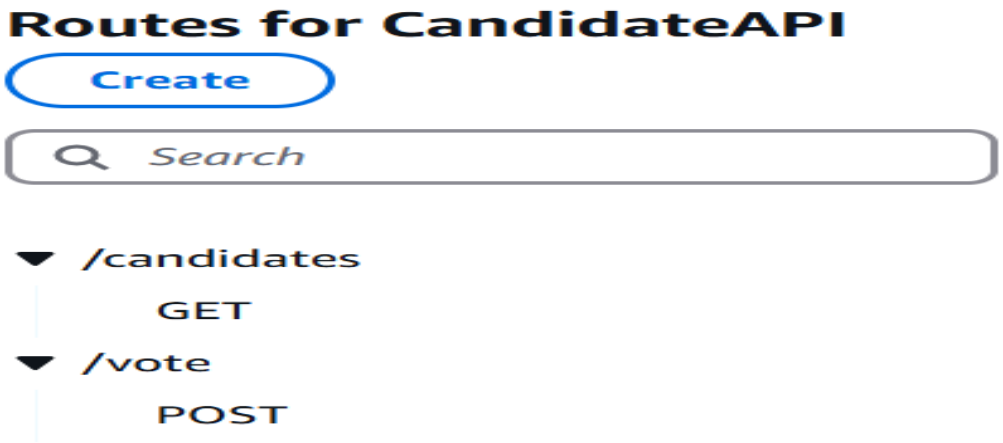

Fig 4.4: Displays the configured API Gateway endpoints for retrieving candidates and submitting votes.

5.Flask Backend Running in VS Code

```python
from flask import Flask, render_template, jsonify, request
import requests

app = Flask(__name__)

# Replace with your actual API Gateway URLs
API_CANDIDATES = "https://v824iep62c.execute-api.ap-south-1.amazonaws.com/candidates"
API_VOTE = "https://v824iep62c.execute-api.ap-south-1.amazonaws.com/vote"

@app.route("/")
def index():
    """Render the frontend page."""
    return render_template("index.html")

@app.route("/get_candidates")
def get_candidates():
    """Fetch candidates from API Gateway, sort by name, and return JSON data."""
    try:
        response = requests.get(API_CANDIDATES)
        response.raise_for_status()
        candidates = sorted(response.json(), key=lambda c: c["name"])
        return jsonify(candidates)
    except requests.exceptions.RequestException as e:
        return jsonify({"error": str(e)}), 500

@app.route("/vote", methods=["POST"])
def vote():
    """Send a vote request to API Gateway."""
    try:
        data = request.json
        response = requests.post(API_VOTE, json=data)
        response.raise_for_status()
        return response.json(), response.status_code
    except requests.exceptions.RequestException as e:
        return jsonify({"error": str(e)}), 500

if __name__ == "__main__":
    app.run(debug=True)
```

```
* Running on http://127.0.0.1:5000
Press CTRL+C to quit
* Restarting with stat
* Debugger is active!
* Debugger PIN: 175-196-323
127.0.0.1 - - [31/Mar/2025 14:34:05] "GET / HTTP/1.1" 200 -
127.0.0.1 - - [31/Mar/2025 14:34:06] "GET /favicon.ico HTTP/1.1" 404 -
127.0.0.1 - - [31/Mar/2025 14:34:07] "GET /get_candidates HTTP/1.1" 200 -
```

Fig 4.5 : Shows the Flask backend running locally, handling requests from the frontend and forwarding them to AWS.

6. Frontend UI - Candidate List

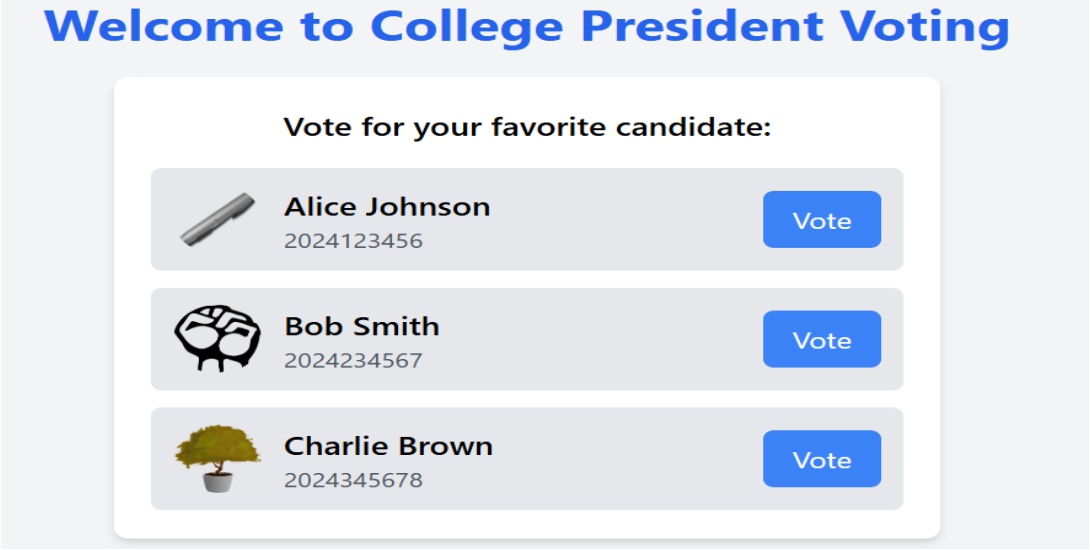

Fig 4.6: Shows the list of candidates displayed in the UI with names, images, and vote buttons.

5.Learning Outcomes

1. AWS & Serverless Architecture

- Learned how to build a serverless application using AWS services such as API Gateway, Lambda, and DynamoDB.
- Used AWS Lambda to create backend logic without needing to manage servers, ensuring automatic scaling and cost efficiency.
- Configured API Gateway to expose endpoints securely and manage HTTP requests.
- Utilized DynamoDB for real-time data storage, ensuring fast and reliable access to candidate and voting data.
- Understood the benefits of a serverless architecture, such as high availability, scalability, and minimal operational overhead.

2. Backend Development with Flask & AWS Integration

- Developed a Flask-based backend API to handle requests from the frontend and interact with AWS services.
- Integrated Flask with API Gateway endpoints, ensuring seamless data communication.
- Implemented secure data transactions, including error handling, input validation, and exception handling, to prevent data corruption.
- Used boto3 (AWS SDK for Python) to interact with DynamoDB, performing operations like scanning for candidates and updating vote counts.
- Ensured stateless execution using AWS Lambda, making the backend more efficient and scalable.

3. Frontend Development for Real-Time Voting

- Designed an interactive UI using HTML, JavaScript, and Tailwind CSS, ensuring a modern and responsive experience.
- Implemented AJAX/Fetch API calls to dynamically fetch candidates and update votes without page refresh.
- Used JavaScript event handling to trigger vote submissions and update the UI in real-time.
- Ensured mobile responsiveness using Tailwind CSS, making the voting system accessible across different devices.
- Provided visual feedback (e.g., disabling buttons, changing colors) to enhance the user experience after voting.

6.Conclusion

This project successfully demonstrates the implementation of a cloud-based voting system using AWS services such as API Gateway, Lambda, and DynamoDB. By leveraging a serverless architecture, the system ensures scalability, cost efficiency, and high availability without the need for dedicated server management.

The Flask-based backend seamlessly interacts with AWS services, handling secure API calls for retrieving candidates and submitting votes. The frontend, built with HTML, JavaScript, and Tailwind CSS, provides a dynamic and user-friendly interface, ensuring a smooth voting experience with real-time updates.

Additionally, data integrity and security were maintained through atomic updates in DynamoDB, preventing vote tampering. Monitoring and debugging were achieved using CloudWatch logs, ensuring efficient system performance.

Overall, this project highlights the effectiveness of serverless computing in developing secure, scalable, and efficient applications, providing valuable hands-on experience in cloud computing and full-stack development

7. References

[1] K. McCorry, P. Haines, F. Hao, *Blockchain-Based E-Voting System*, 2017.

[2] S. D. Anton, J. Kim, R. Brown, *Secure E-Voting Using Cloud Computing*, 2019.

[3] J. Spillner, *Serverless Architectures on AWS: A Practical Analysis*, 2020.

[4] M. Patel, A. Sharma, *Enhancing Electronic Voting Security with AI and Cloud Technologies*, 2021.

[5] L. Zhang, T. Nguyen, *Scalable and Fault-Tolerant Voting Systems Using AWS Lambda*, 2022.