DALHOUSIE UNIVERSITY

CSCI 5409: Adv. Topic in Cloud Computing

Term Project

by

Axata Darji (B00985046)

Submitted to

Prof. Dr. Lu Yang
Faculty of Computer Science
Dalhousie University

Date: 5th August 2024

Table of Contents

Aim:

The aim of this project is to create a diary application for managing pending items or events, with functionalities to add or delete items and a reminder feature. The reminder feature will send an email each day with the day's pending items or events. This application will utilize several AWS services, including EC2, SQS, SNS, API Gateway, Lambda, Dynamo DB, and Event Bridge. The infrastructure for this application will be created using AWS Cloud Formation.
Audience:

- Individuals managing their tasks
- Students and Professionals

Thought Process:

Design Considerations:

1. **Frontend Design**: The frontend is designed using HTML, CSS, and JavaScript, with Bootstrap for styling and layout.
2. **Backend Architecture**: The backend is powered by AWS services such as Lambda functions written in Node.js (using ES modules), utilizing the AWS SDK for JavaScript to interact with DynamoDB. All Lambda functions for adding, fetching, and retrieving event information are hosted using REST API created via API Gateway. The backend also includes EC2 for hosting and deploying the application. For the reminder feature, EventBridge is used to set the everyday trigger. SQS holds the current day's events, and a Lambda function reads information from SQS and utilizes SNS to send an email.
3. **Data Storage**: Events are stored in DynamoDB.
4. **API Integration**: The frontend communicates with the backend APIs via AJAX requests (using jQuery), enabling CRUD operations on events.
5. **Error Handling**: Error handling is implemented to provide appropriate feedback to users in case of failures.

Comparison of alternative services:

Table 1: Comparison of alternative services

Service used	Purpose of Service	Alternative Services	Comparison
EC2	For hosting front end application	S3,Elastic Beanstalk	• S3 is better option for hosting static content. • Also EC2 allows to install and run any software and dependencies that application might need which is not possible with S3 • S3 provides high scalability and availability without managing servers with less cost compared to EC2, but in terms of

			advanced security and capabilities EC2 is more suitable. • AWS handles the provisioning, load balancing, scaling, and monitoring for Elastic beanstalk while for EC2 , complete control over the OS, installed software, and configuration • In Elastic Beanstalk, limited control over the underlying resources compared to EC2
Lambda	Backend services	Step Functions	• Lambda Supports various triggers, such as HTTP requests, S3 bucket events, Dynamo DB streams, and more while step functions uses state machines to define and manage the sequence of steps involved in an application workflow. • Step functions are Ideal for building complex workflows that require multiple steps, retries, and error handling. Since application needs to execute short, stateless and event driven tasks , lambdas are best solution. • As per application, lambda functions will cost lesser compared to state functions
DynamoDB	As a storage	S3, RDS	• DynamoDB is a NoSQL type storage while S3 is an object type storage and RDS is SQL type relational database. • DynamoDB is a schema less data model which supports key value pairs and JSON documents which makes best choice for scalability • S3 is the best choice for storing object type large volumes of unstructured data such as media files, big data and backups. While RDS is the best choice for relational data models and for complex queries, transactions. • Since application needs high-performance scalable NoSQL database which can handle flexible schema, DynamoDB is the best choice.
API Gateway	For creating RestAPI	Event Bridge, Elastic Beanstalk, Lambda	• EventBridge is best service to create event driven architecture, and API Gateway is the best choice for REST API Management • Elastic Beanstalk is also application deployment and management service. Ideal for deploying and managing full-stack

			applications, including web servers that handle REST API requests. • Both Event Bridge and Elastic beanstalk are well suited for deploying and managing full stack applications while API gateway is designed specifically for creating, deploying and managing REST APIs. • AWS Lambda is a compute service that allows to run code in response to events without managing servers. It is often used to handle backend logic, process data, or respond to events • API Gateway and Lambda are often used together to build serverless APIs: API Gateway handles the HTTP requests and routes them to Lambda functions, which then execute the business logic and interact with other AWS services as needed.
SQS,SNS	For reminder functionality and dropping email	Amazon Kinesis, Amazon SES	• SES is not supported for lab role in learning lab so only option left for dropping an email/text is SNS. SNS provides managed email service for sending emails • Amazon SQS is best suited for traditional message queuing scenarios where you need to decouple application components, handle asynchronous processing, and manage simple message workflows. It offers reliable message delivery with options for ordering and deduplication • Amazon Kinesis Data Streams designed for applications that need to process and analyze data streams in real-time, handle large volumes of data, and retain data for extended periods. This makes it not suitable for this application

Deployment Model:

The application currently lacks a login functionality. The data which this application stores is personal events or pending items which is not that crucial nor it affects individual privacy. So I am choosing **public cloud** as deployment cloud.

Features of public cloud:

1. **Cost Efficiency**

- **Lower Costs:** Public cloud services like AWS, Azure, and Google Cloud offer pay-as-you-go pricing models, which are cost-effective for hosting applications with low to moderate resource requirements.
- **No Upfront Investment:** There is no need for significant upfront capital expenditure on hardware or infrastructure, which is ideal for personal projects or small-scale applications.

2. **Ease of Deployment and Management**

- **Simplified Deployment:** Public clouds offer various tools and services that simplify the deployment process, such as managed container services, serverless computing (e.g., AWS Lambda), and managed databases.
- **Automated Management:** Many public cloud services come with automated management features, such as automated backups, patch management, and monitoring, reducing the operational burden.

3. **Accessibility and Convenience**

- **Global Reach:** Public cloud providers have data centers in multiple regions worldwide, ensuring that application can be accessed with low latency from different geographic locations.
- **Anytime, Anywhere Access:** Being hosted on the public cloud, application and its data can be accessed from anywhere with an internet connection, providing convenience and flexibility.

Following is my justification for using Public Cloud as deployment model for the application:

- **Non-Critical Data:** Since application stores personal events or pending items that are not crucial or privacy-sensitive, the security and compliance measures provided by public cloud platforms are sufficient to protect data.
- **Cost-Effective Hosting:** Public cloud deployment is cost-effective, especially for personal projects or applications with low resource demands, ensuring only pay for what you use.
- **Convenience:** The ease of deployment and management, along with the accessibility of the public cloud, makes it a practical choice for the application, allowing to focus on development rather than infrastructure management.

By choosing a public cloud for deploying the task management application, it benefits from cost efficiency, scalability, ease of management, and access to a wide range of services, all while ensuring non-critical data is securely hosted and accessible.

Delivery Model:

The project aims to deliver a fully functional diary application for managing pending items with add, delete, and reminder features, all of which are typical characteristics of a SaaS offering. The use of various AWS services for functionality and the automated infrastructure management through Cloud Formation further solidify the SaaS classification. Therefore, the delivery model for the project is **Software as a Service (SaaS)**.
Following is my justification for choosing SaaS as delivery model:

1. **End-User Focused Application:**
    - **Diary Application for Managing Pending Items:** The application is designed to be used by individual to manage pending items, add or delete items, and receive daily reminders. This aligns with typical SaaS offerings, which are user-centric and deliver software applications over the internet.
2. **Hosted and Managed on the Cloud:**
    - **AWS Services Utilization:** The application leverages various AWS services (EC2, SQS, SNS, API Gateway, Lambda, Dynamo DB, Event Bridge) to provide its functionality. The infrastructure and services are managed in the cloud, ensuring that users do not need to manage the underlying hardware or software.
3. **Daily Reminder Feature:**
    - **Automated Email Notifications:** The daily email reminder feature, implemented using AWS services, adds to the user experience by providing automated notifications. This is a common feature in SaaS applications that aim to enhance user productivity and engagement.
4. **Infrastructure as Code:**
    - **AWS Cloud Formation:** The use of AWS Cloud Formation to create and manage the infrastructure aligns with modern SaaS practices, where the deployment and management of infrastructure are automated and handled by the cloud provider.
5. **Accessibility and Convenience:**
    - **Internet-Based Access:** SaaS applications are typically accessible over the internet, offering convenience and flexibility to users. The diary application, being hosted on AWS and accessible via API Gateway, fits this model as users can access it from anywhere with an internet connection.

Architecture of System:

The architecture ensures a robust and scalable diary application with efficient user interaction, backend processing, data storage, and daily reminder notifications. The use of AWS services like EC2, Lambda, DynamoDB, SQS, SNS, and EventBridge provides a reliable and cost-effective solution.

Here's a detailed description of the architecture:

1. **EC2 Instance Hosting the Frontend**:
    - **Frontend Deployment**: An EC2 instance running an Nginx server is set up to host the frontend application. The HTML and CSS files for the frontend are fetched from a GitHub repository and deployed on this EC2 instance.
    - **Nginx Server**: Nginx is used to serve the static frontend files, ensuring efficient and fast delivery of web pages to users.
2. **User Interaction and REST API**:
    - **User Actions**: Users interact with the frontend to add, fetch, or delete events. These actions trigger REST API calls.

- **API Gateway**: The REST API is implemented using Amazon API Gateway, which acts as a secure and scalable interface for invoking backend services.

3. **Backend Processing with Lambda Functions**:
   - **Add Event**: When a user adds an event, the API Gateway triggers a Lambda function that connects to DynamoDB and stores the event details.
   - **Fetch Events**: To fetch events, another Lambda function is triggered by the API Gateway, which retrieves the relevant data from DynamoDB.
   - **Delete Event**: Deleting an event also triggers a specific Lambda function that removes the event from DynamoDB.
4. **Data Storage in DynamoDB**:
   - **Event Storage**: All events and tasks are stored in DynamoDB, which provides fast and scalable NoSQL data storage.
5. **Daily Reminder Emails**:
   - **EventBridge Trigger**: An Amazon EventBridge rule is set to trigger every day. This rule invokes a Lambda function to process the daily reminders.
   - **Fetch and Queue Events**: The triggered Lambda function connects to DynamoDB to fetch the current day's events and pushes these details to an SQS queue.
   - **Email Notification**: Another Lambda function, triggered by the SQS queue, reads the events and uses Amazon SNS to send out the daily reminder emails to users.

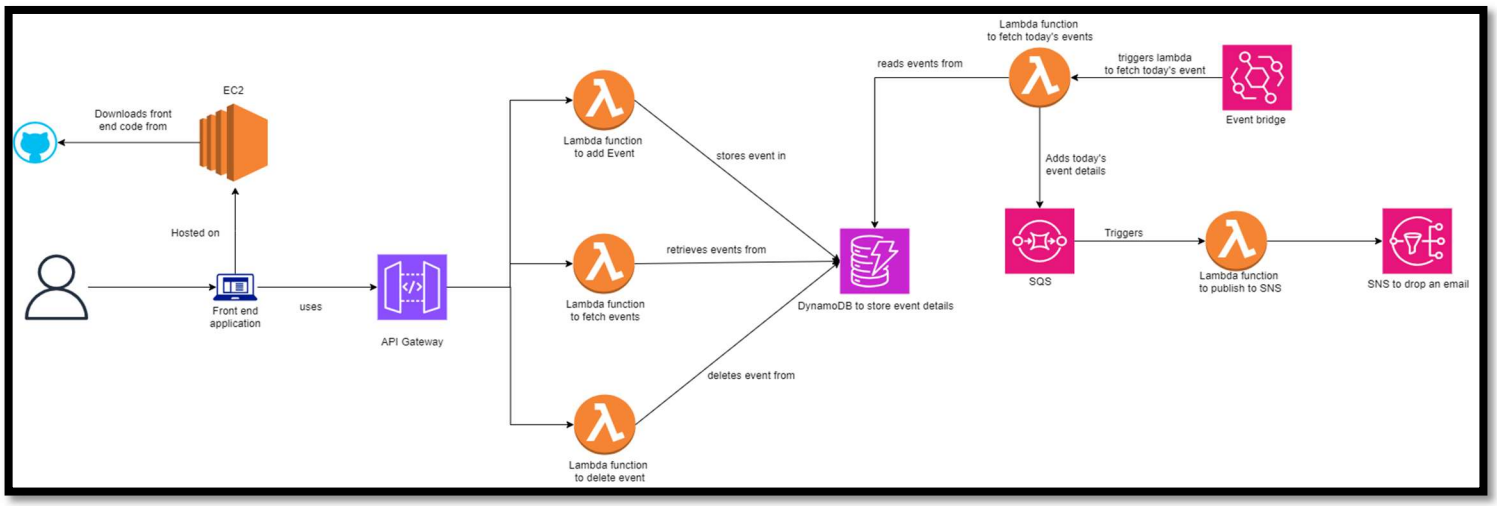

Figure 1: System Architecture

Programming Languages:

1. JavaScript (Node.js)

- **Why Used**: Node.js is chosen for its non-blocking, event-driven architecture, making it well-suited for serverless functions. Its seamless integration with AWS Lambda and extensive library support via npm also make it a preferred choice for this application.
- **Where Used**:
  - **Lambda Functions**: All backend logic for adding, fetching, and deleting events in DynamoDB is implemented using Node.js. This includes interaction with the AWS SDK for DynamoDB and handling API Gateway requests.
  - **SQS and SNS Integration**: Lambda functions that interact with SQS and SNS for managing queues and sending emails are also written in Node.js.

2. HTML, CSS, JavaScript

- **Why Used**: HTML, CSS, and JavaScript are the foundational technologies for web development. They provide structure, style, and interactivity for the frontend of the application.
- **Where Used**:
  - **Frontend Application**: The static files for the frontend application, including the user interface for managing events, are built using HTML for structure, CSS for styling, and JavaScript for interactivity.
  - **AJAX Requests**: JavaScript is used on the frontend to make asynchronous API calls to the backend REST API endpoints.

Parts of the Application Requiring Code

1. **Frontend Application**:
   - **HTML**: Structure of the web pages.
   - **CSS**: Styling and layout using Bootstrap for responsive design.
   - **JavaScript**: Interactivity and AJAX requests to the backend.
2. **Backend Logic**:
   - **Node.js Lambda Functions**:
     - **Add Event**: Code to add an event to DynamoDB.
     - **Fetch Events**: Code to retrieve events from DynamoDB.
     - **Delete Event**: Code to delete an event from DynamoDB.
     - **Data Processing**: Code to process event data and prepare daily reminders.
     - **EventBridge Trigger**: Code triggered by EventBridge to fetch and queue daily events.
3. **API Management**:
   - **API Gateway**: Configuration and integration with Lambda functions to expose REST API endpoints.
4. **Queue Management and Notifications**:
   - **SQS Integration**: Code in Lambda functions to push events to SQS.
   - **SNS Integration**: Code in Lambda functions to send emails using SNS

How system is deployed to the cloud

The system is deployed on cloud using following cloud formation script:

```
AWSTemplateFormatVersion: '2010-09-09'
Description: AWS CloudFormation for creating Lambda functions, API Gateway, SQS, and SNS

Parameters:
  DeploymentTimestamp:
    Type: String
    Description: "Timestamp to ensure unique deployment"
  KeyName:
    Type: AWS::EC2::KeyPair::KeyName
    Description: Name of an existing EC2 KeyPair to enable SSH access to the instance
    Default: WebServer01Keypair

Resources:
  DynamoDBTable:
    Type: "AWS::DynamoDB::Table"
    Properties:
```

```
      TableName: "TestEvent"
      AttributeDefinitions:
        - AttributeName: "eventId"
          AttributeType: "S"
      KeySchema:
        - AttributeName: "eventId"
          KeyType: "HASH"
      ProvisionedThroughput:
        ReadCapacityUnits: 5
        WriteCapacityUnits: 5

  AddEventsLambdaFunction:
    Type: AWS::Lambda::Function
    Properties:
      FunctionName: FnAddEvent
      Handler: index.handler
      Role: arn:aws:iam::698915160078:role/LabRole
      Runtime: nodejs18.x
      Code:
        ZipFile: |
          const { DynamoDBClient, PutItemCommand } = require('@aws-sdk/client-dynamodb');

          const dynamoDBClient = new DynamoDBClient({
              region: "us-east-1", // Change the region to match your DynamoDB region
          });

          const mainTableName = 'TestEvent';

          module.exports.handler = async (event) => {
              try {
                  console.log("Input is: ", event);

                  // Check if event.body exists and parse it if it's a string
                  const body = event.body ? (typeof event.body === 'string' ?
JSON.parse(event.body) : event.body) : event;

                  // Generate eventId using current timestamp
                  const eventId = Date.now().toString();

                  // Add item to main table
                  const item = {
                      eventId: { S: eventId },
                      date: { S: body.date },
                      time: { S: body.time },
                      description: { S: body.description }
```

```
                };
                await addItemToMainTable(item);

                console.log("Item added successfully");

                // Return success response with CORS headers
                return {
                    statusCode: 200,
                    headers: {
                        "Access-Control-Allow-Origin": "*", // Allow all origins
                        "Access-Control-Allow-Methods": "POST, OPTIONS",
                        "Access-Control-Allow-Headers": "Content-Type",
                    },
                    body: JSON.stringify({ message: 'Item added successfully.', eventId:
eventId }),
                };
            } catch (error) {
                console.error("Error occurred which is: ", error);
                // Return error with CORS headers
                return {
                    statusCode: 500,
                    headers: {
                        "Access-Control-Allow-Origin": "*", // Allow all origins
                        "Access-Control-Allow-Methods": "POST, OPTIONS",
                        "Access-Control-Allow-Headers": "Content-Type",
                    },
                    body: JSON.stringify({ message: 'Error adding item to DB', error:
error.message }),
                };
            }
        };

        async function addItemToMainTable(item) {
            const params = {
                TableName: mainTableName,
                Item: item,
                ConditionExpression: 'attribute_not_exists(eventId)' // Ensure eventId is
unique
            };
            await dynamoDBClient.send(new PutItemCommand(params));
        }
    Environment:
      Variables:
        MAIN_TABLE_NAME: 'TestEvent'
    Timeout: 30
```

```
CheckEventsLambdaFunction:
  Type: AWS::Lambda::Function
  Properties:
    FunctionName: FnProcessEvents
    Handler: index.handler
    Role: arn:aws:iam::698915160078:role/LabRole
    Runtime: nodejs18.x
    Code:
      ZipFile: |
        const { DynamoDBClient, ScanCommand } = require('@aws-sdk/client-dynamodb');
        const { SNSClient, PublishCommand } = require('@aws-sdk/client-sns');
        const { SQSClient, SendMessageCommand } = require('@aws-sdk/client-sqs');

        const dynamodb = new DynamoDBClient({ region: 'us-east-1' });
        const sns = new SNSClient({ region: 'us-east-1' });
        const sqs = new SQSClient({ region: 'us-east-1' });

        const TABLE_NAME = 'TestEvent';
        const DATE_COLUMN = 'date';
        const DESCRIPTION_COLUMN = 'description';
        const TIME_COLUMN = 'time';
        const SNS_TOPIC_ARN = 'arn:aws:sns:us-east-1:698915160078:MyTopic';
        const SQS_QUEUE_URL = 'https://sqs.us-east-1.amazonaws.com/698915160078/MyQueue';

        exports.handler = async (event) => {
            const currentDate = new Date().toISOString().split('T')[0];

            const params = {
                TableName: TABLE_NAME,
                FilterExpression: '#date = :current_date',
                ExpressionAttributeNames: {
                    '#date': 'date',
                },
                ExpressionAttributeValues: {
                    ':current_date': { S: currentDate },
                },
            };

            try {
                const data = await dynamodb.send(new ScanCommand(params));
                const items = data.Items;

                for (const item of items) {
                    const description = item[DESCRIPTION_COLUMN]?.S;
```

```
                    const time = item[TIME_COLUMN]?.S;

                    let message = `Event Description: ${description}\n`;

                    message += `\nHey buddy,\n\n`
                            + `How are you doing today? You are strong buddy and you can do
anything!!\n`
                            + `Just a reminder that today you have Event: ${description}`;

                    if (time && time.trim() !== '') {
                        message += ` at: ${time}\n`;
                    }

                    message += `\n\nDo not forget about this and keep rocking!!\nHave a
fantastic day!!`;

                    await sns.send(new PublishCommand({
                        TopicArn: SNS_TOPIC_ARN,
                        Subject: `Event Notification for ${currentDate}`,
                        Message: message,
                    }));

                    await sqs.send(new SendMessageCommand({
                        QueueUrl: SQS_QUEUE_URL,
                        MessageBody: message,
                    }));
                }

                return {
                    statusCode: 200,
                    body: `Sent ${items.length} notifications`,
                };
            } catch (error) {
                console.error('Error sending notifications:', error);
                return {
                    statusCode: 500,
                    body: `Failed to send notifications: ${error.message}`,
                };
            }
        };
    Environment:
      Variables:
        TABLE_NAME: 'TestEvent'
        SNS_TOPIC_ARN: 'arn:aws:sns:us-east-1:698915160078:MyTopic'
        SQS_QUEUE_URL: 'https://sqs.us-east-1.amazonaws.com/698915160078/MyQueue'
```

```
    Timeout: 60

DeleteEventLambdaFunction:
  Type: AWS::Lambda::Function
  Properties:
    FunctionName: FnDeleteEvent
    Handler: index.handler
    Role: arn:aws:iam::698915160078:role/LabRole
    Runtime: nodejs18.x
    Code:
      ZipFile: |
        const { DynamoDBClient, DeleteItemCommand } = require("@aws-sdk/client-dynamodb");

        const dynamoDBClient = new DynamoDBClient({
            region: "us-east-1",
        });

        const tableName = 'TestEvent';

        const handler = async (event) => {
            try {
                console.log("Input event:", JSON.stringify(event, null, 2));

                // Handle CORS preflight request
                if (event.httpMethod === 'OPTIONS') {
                    console.log("CORS preflight request");
                    return {
                        statusCode: 200,
                        headers: {
                            "Access-Control-Allow-Origin": "*",
                            "Access-Control-Allow-Methods": "POST, OPTIONS",
                            "Access-Control-Allow-Headers": "Content-Type",
                        },
                        body: JSON.stringify({ message: 'CORS preflight response' }),
                    };
                }

                // Extract and parse the body
                let body = event.body;
                console.log("Raw body:", body);

                // If the body contains extra escaping, handle it here
                let parsedBody;
                try {
                    parsedBody = JSON.parse(body);
```

```
            console.log("Parsed body:", parsedBody);
        } catch (err) {
            console.error("Error parsing body:", err);
            throw new Error('Error parsing request body');
        }

        // Extract eventId from the nested body
        let nestedBody;
        try {
            nestedBody = JSON.parse(parsedBody.body);
            console.log("Nested body:", nestedBody);
        } catch (err) {
            console.error("Error parsing nested body:", err);
            throw new Error('Error parsing nested request body');
        }

        const { eventId } = nestedBody;
        console.log("Extracted eventId:", eventId);

        if (!eventId) {
            throw new Error('Event ID is required');
        }

        // Delete the event from DynamoDB
        await deleteEvent(eventId);
        console.log("Event deleted successfully:", eventId);

        // Return success response with CORS headers
        return {
            statusCode: 200,
            headers: {
                "Access-Control-Allow-Origin": "*",
                "Access-Control-Allow-Methods": "POST, OPTIONS",
                "Access-Control-Allow-Headers": "Content-Type",
            },
            body: JSON.stringify({ message: 'Event deleted successfully.' }),
        };
    } catch (error) {
        console.error("Error occurred:", error.message);
        console.error("Stack trace:", error.stack);

        // Return error with CORS headers
        return {
            statusCode: 500,
            headers: {
```

```
                        "Access-Control-Allow-Origin": "*",
                        "Access-Control-Allow-Methods": "POST, OPTIONS",
                        "Access-Control-Allow-Headers": "Content-Type",
                    },
                    body: JSON.stringify({ message: 'Error deleting event from DB', error:
error.message }),
                };
            }
        };

        async function deleteEvent(eventId) {
            try {
                console.log("Deleting event from DynamoDB with eventId:", eventId);
                const params = {
                    TableName: tableName,
                    Key: {
                        eventId: { S: eventId }
                    }
                };
                await dynamoDBClient.send(new DeleteItemCommand(params));
                console.log("Successfully deleted event from DynamoDB");
            } catch (error) {
                console.error("Error deleting event from DynamoDB:", error.message);
                throw error;
            }
        }

        module.exports = { handler };
    Environment:
      Variables:
        TABLE_NAME: 'TestEvent'
    Timeout: 30

  RetrieveEventsLambdaFunction:
    Type: AWS::Lambda::Function
    Properties:
      FunctionName: FnRetrieveEvents
      Handler: index.handler
      Role: arn:aws:iam::698915160078:role/LabRole
      Runtime: nodejs18.x
      Code:
        ZipFile: |
          const { DynamoDBClient, ScanCommand } = require('@aws-sdk/client-dynamodb');

          const dynamoDBClient = new DynamoDBClient({
```

```
    region: "us-east-1",
});

const TABLE_NAME = 'TestEvent';

exports.handler = async (event) => {
    try {
        console.log("Input is: ", event);

        // Retrieve all items from DynamoDB
        const params = {
            TableName: TABLE_NAME,
        };

        const data = await dynamoDBClient.send(new ScanCommand(params));
        console.log('DynamoDB Response:', data);

        // Simplify the response
        const items = data.Items.map(item => {
            return {
                eventId: item.eventId.S,
                date: item.date.S,
                time: item.time.S,
                description: item.description.S
            };
        });

        // Return success response
        return {
            statusCode: 200,
             headers: {
                "Access-Control-Allow-Origin": "*",
                "Access-Control-Allow-Methods": "GET, OPTIONS",
                "Access-Control-Allow-Headers": "Content-Type",
            },
            body: JSON.stringify(items),
        };
    } catch (error) {
        console.error("Error occurred which is: ", error);
        // Return error
        return {
            statusCode: 500,
             headers: {
                "Access-Control-Allow-Origin": "*",
                "Access-Control-Allow-Methods": "GET, OPTIONS",
```

```
                        "Access-Control-Allow-Headers": "Content-Type",
                    },
                    body: JSON.stringify({ message: 'Could not retrieve items from DB',
error: error.message }),
                };
            }
        };
    Environment:
      Variables:
        TABLE_NAME: 'TestEvent'
    Timeout: 30

  ApiGatewayRestApi:
    Type: AWS::ApiGateway::RestApi
    Properties:
      Name: EventsApi
      EndpointConfiguration:
        Types:
          - REGIONAL

  ApiGatewayEventsResource:
    Type: AWS::ApiGateway::Resource
    Properties:
      ParentId: !GetAtt ApiGatewayRestApi.RootResourceId
      PathPart: events
      RestApiId: !Ref ApiGatewayRestApi

  ApiGatewayAddResource:
    Type: AWS::ApiGateway::Resource
    Properties:
      ParentId: !Ref ApiGatewayEventsResource
      PathPart: add
      RestApiId: !Ref ApiGatewayRestApi

  ApiGatewayDeleteResource:
    Type: AWS::ApiGateway::Resource
    Properties:
      ParentId: !Ref ApiGatewayEventsResource
      PathPart: delete
      RestApiId: !Ref ApiGatewayRestApi

  ApiGatewayShowResource:
    Type: AWS::ApiGateway::Resource
    Properties:
      ParentId: !Ref ApiGatewayEventsResource
```

```
      PathPart: show
      RestApiId: !Ref ApiGatewayRestApi

  AddEventsApiGatewayMethod:
    Type: AWS::ApiGateway::Method
    Properties:
      RestApiId: !Ref ApiGatewayRestApi
      ResourceId: !Ref ApiGatewayAddResource
      HttpMethod: POST
      AuthorizationType: NONE
      Integration:
        Type: AWS
        IntegrationHttpMethod: POST
        Uri: !Sub arn:aws:apigateway:${AWS::Region}:lambda:path/2015-03-
31/functions/${AddEventsLambdaFunction.Arn}/invocations
        RequestTemplates:
          application/json: "{ \"body\": $input.json('$') }"
        IntegrationResponses:
          - StatusCode: 200
            ResponseTemplates:
              application/json: ""
            ResponseParameters:
              method.response.header.Access-Control-Allow-Origin: "'*'"
              method.response.header.Access-Control-Allow-Methods: "'POST,OPTIONS'"
              method.response.header.Access-Control-Allow-Headers: "'Content-Type'"
      MethodResponses:
        - StatusCode: 200
          ResponseModels:
            application/json: Empty
          ResponseParameters:
            method.response.header.Access-Control-Allow-Origin: true
            method.response.header.Access-Control-Allow-Methods: true
            method.response.header.Access-Control-Allow-Headers: true

  AddEventsApiOptionsGatewayMethod:
    Type: AWS::ApiGateway::Method
    Properties:
      AuthorizationType: NONE
      HttpMethod: OPTIONS
      ResourceId: !Ref ApiGatewayAddResource
      RestApiId: !Ref ApiGatewayRestApi
      Integration:
        IntegrationHttpMethod: POST
        Type: MOCK
        RequestTemplates:
```

```
          application/json: '{"statusCode": 200}'
        IntegrationResponses:
          - StatusCode: 200
            ResponseParameters:
              method.response.header.Access-Control-Allow-Origin: "'*'"
              method.response.header.Access-Control-Allow-Methods: "'OPTIONS,POST'"
              method.response.header.Access-Control-Allow-Headers: "'Content-Type,X-Amz-
Date,Authorization,X-Api-Key,X-Amz-Security-Token'"
            ResponseTemplates:
              application/json: ''
      MethodResponses:
        - StatusCode: 200
          ResponseModels:
            application/json: 'Empty'
          ResponseParameters:
            method.response.header.Access-Control-Allow-Origin: true
            method.response.header.Access-Control-Allow-Methods: true
            method.response.header.Access-Control-Allow-Headers: true

  DeleteEventsApiGatewayMethod:
    Type: AWS::ApiGateway::Method
    Properties:
      RestApiId: !Ref ApiGatewayRestApi
      ResourceId: !Ref ApiGatewayDeleteResource
      AuthorizationType: NONE
      HttpMethod: POST
      Integration:
        IntegrationHttpMethod: POST
        Type: AWS_PROXY
        Uri: !Sub arn:aws:apigateway:${AWS::Region}:lambda:path/2015-03-
31/functions/${DeleteEventLambdaFunction.Arn}/invocations
        RequestTemplates:
          application/json: "{ \"body\": $input.json('$') }"
        IntegrationResponses:
          - StatusCode: 200
            ResponseTemplates:
              application/json: ""
            ResponseParameters:
              method.response.header.Access-Control-Allow-Origin: "'*'"
              method.response.header.Access-Control-Allow-Methods: "'POST,OPTIONS'"
              method.response.header.Access-Control-Allow-Headers: "'Content-Type'"
      MethodResponses:
        - StatusCode: 200
          ResponseModels:
            application/json: Empty
```

```
        ResponseParameters:
          method.response.header.Access-Control-Allow-Origin: true
          method.response.header.Access-Control-Allow-Methods: true
          method.response.header.Access-Control-Allow-Headers: true

  DeleteEventsApiOptionsGatewayMethod:
    Type: AWS::ApiGateway::Method
    Properties:
      AuthorizationType: NONE
      HttpMethod: OPTIONS
      ResourceId: !Ref ApiGatewayDeleteResource
      RestApiId: !Ref ApiGatewayRestApi
      Integration:
        IntegrationHttpMethod: POST
        Type: MOCK
        RequestTemplates:
          application/json: '{"statusCode": 200}'
        IntegrationResponses:
          - StatusCode: 200
            ResponseParameters:
              method.response.header.Access-Control-Allow-Origin: "'*'"
              method.response.header.Access-Control-Allow-Methods: "'OPTIONS,POST'"
              method.response.header.Access-Control-Allow-Headers: "'Content-Type,X-Amz-
Date,Authorization,X-Api-Key,X-Amz-Security-Token'"
            ResponseTemplates:
              application/json: ''
      MethodResponses:
        - StatusCode: 200
          ResponseModels:
            application/json: 'Empty'
          ResponseParameters:
            method.response.header.Access-Control-Allow-Origin: true
            method.response.header.Access-Control-Allow-Methods: true
            method.response.header.Access-Control-Allow-Headers: true

  ShowEventsApiGatewayMethod:
    Type: AWS::ApiGateway::Method
    Properties:
      RestApiId: !Ref ApiGatewayRestApi
      ResourceId: !Ref ApiGatewayShowResource
      AuthorizationType: NONE
      HttpMethod: GET
      Integration:
        IntegrationHttpMethod: POST
        Type: AWS
```

```
        Uri: !Sub arn:aws:apigateway:${AWS::Region}:lambda:path/2015-03-
31/functions/${RetrieveEventsLambdaFunction.Arn}/invocations
        RequestTemplates:
          application/json: "{ \"statusCode\": 200 }"
        IntegrationResponses:
          - StatusCode: 200
            ResponseTemplates:
              application/json: ""
            ResponseParameters:
              method.response.header.Access-Control-Allow-Origin: "'*'"
              method.response.header.Access-Control-Allow-Methods: "'GET,OPTIONS'"
              method.response.header.Access-Control-Allow-Headers: "'Content-Type'"
      MethodResponses:
        - StatusCode: 200
          ResponseModels:
            application/json: Empty
          ResponseParameters:
            method.response.header.Access-Control-Allow-Origin: true
            method.response.header.Access-Control-Allow-Methods: true
            method.response.header.Access-Control-Allow-Headers: true

  ShowEventsApiOptionsGatewayMethod:
    Type: AWS::ApiGateway::Method
    Properties:
      AuthorizationType: NONE
      HttpMethod: OPTIONS
      ResourceId: !Ref ApiGatewayShowResource
      RestApiId: !Ref ApiGatewayRestApi
      Integration:
        IntegrationHttpMethod: POST
        Type: MOCK
        RequestTemplates:
          application/json: "{}"
        IntegrationResponses:
          - StatusCode: 200
            ResponseParameters:
              method.response.header.Access-Control-Allow-Origin: "'*'"
              method.response.header.Access-Control-Allow-Methods: "'OPTIONS,GET'"
              method.response.header.Access-Control-Allow-Headers: "'Content-Type'"
      MethodResponses:
        - StatusCode: 200
          ResponseParameters:
            method.response.header.Access-Control-Allow-Origin: true
            method.response.header.Access-Control-Allow-Methods: true
            method.response.header.Access-Control-Allow-Headers: true
```

```
  ApiGatewayDeployment:
    Type: AWS::ApiGateway::Deployment
    DependsOn:
      - AddEventsApiGatewayMethod
      - AddEventsApiOptionsGatewayMethod
      - DeleteEventsApiGatewayMethod
      - DeleteEventsApiOptionsGatewayMethod
      - ShowEventsApiGatewayMethod
      - ShowEventsApiOptionsGatewayMethod
    Properties:
      RestApiId: !Ref ApiGatewayRestApi
      StageName: dev
      Description: !Sub "Deployment created at ${DeploymentTimestamp}"

  AddEventsLambdaPermission:
    Type: AWS::Lambda::Permission
    Properties:
      Action: lambda:InvokeFunction
      FunctionName: !GetAtt AddEventsLambdaFunction.Arn
      Principal: apigateway.amazonaws.com
      SourceArn: !Sub arn:aws:execute-
api:${AWS::Region}:${AWS::AccountId}:${ApiGatewayRestApi}/*/POST/events/add

  DeleteEventsLambdaPermission:
    Type: AWS::Lambda::Permission
    Properties:
      Action: lambda:InvokeFunction
      FunctionName: !GetAtt DeleteEventLambdaFunction.Arn
      Principal: apigateway.amazonaws.com
      SourceArn: !Sub arn:aws:execute-
api:${AWS::Region}:${AWS::AccountId}:${ApiGatewayRestApi}/*/POST/events/delete

  ShowEventsLambdaPermission:
    Type: AWS::Lambda::Permission
    Properties:
      Action: lambda:InvokeFunction
      FunctionName: !GetAtt RetrieveEventsLambdaFunction.Arn
      Principal: apigateway.amazonaws.com
      SourceArn: !Sub arn:aws:execute-
api:${AWS::Region}:${AWS::AccountId}:${ApiGatewayRestApi}/*/GET/events/show

  MyQueue:
    Type: AWS::SQS::Queue
    Properties:
```

```
    QueueName: MyQueue

MyTopic:
  Type: AWS::SNS::Topic
  Properties:
    TopicName: MyTopic

CheckAndPushLambda:
  Type: AWS::Lambda::Function
  Properties:
    FunctionName: CheckAndPushLambda
    Handler: index.handler
    Role: arn:aws:iam::698915160078:role/LabRole
    Runtime: nodejs20.x
    Code:
      ZipFile: |
        const { DynamoDBClient, ScanCommand } = require('@aws-sdk/client-dynamodb');
        const { SQSClient, SendMessageCommand } = require('@aws-sdk/client-sqs');

        const dynamo = new DynamoDBClient({ region: 'us-east-1' });
        const sqs = new SQSClient({ region: 'us-east-1' });

        exports.handler = async (event) => {
          const currentDate = new Date().toISOString().split('T')[0];
          const params = {
            TableName: 'TestEvent',
            FilterExpression: '#date = :current_date',
            ExpressionAttributeNames: {
              '#date': 'date',
            },
            ExpressionAttributeValues: {
              ':current_date': { S: currentDate },
            },
          };

          try {
            const result = await dynamo.send(new ScanCommand(params));
            console.log('Result from DynamoDB is', result);

            if (result.Items.length > 0) {
              const messageParams = {
                QueueUrl: process.env.SQS_QUEUE_URL,
                MessageBody: JSON.stringify(result.Items),
              };
```

```
              console.log('Before sending to SQS, params are', messageParams);
              await sqs.send(new SendMessageCommand(messageParams));
              console.log('After sending to SQS');
            }
          } catch (error) {
            console.error('Error processing the DynamoDB scan or sending SQS message:',
error);
            throw error;
          }
        };
    Environment:
      Variables:
        SQS_QUEUE_URL: !Ref MyQueue

  ScheduleRule:
    Type: AWS::Events::Rule
    Properties:
      ScheduleExpression: cron(00 07 * * ? *)
      Targets:
        - Arn: !GetAtt CheckAndPushLambda.Arn
          Id: "CheckAndPushLambda"
      State: ENABLED

  PermissionForEventsToInvokeLambda:
    Type: AWS::Lambda::Permission
    Properties:
      FunctionName: !Ref CheckAndPushLambda
      Action: lambda:InvokeFunction
      Principal: events.amazonaws.com
      SourceArn: !GetAtt ScheduleRule.Arn

  SNSToSQSSubscription:
    Type: AWS::SNS::Subscription
    Properties:
      TopicArn: !Ref MyTopic
      Protocol: sqs
      Endpoint: !GetAtt MyQueue.Arn

  PublishToSNSLambda:
    Type: AWS::Lambda::Function
    Properties:
      FunctionName: PublishToSNSLambda
      Handler: index.handler
      Role: arn:aws:iam::698915160078:role/LabRole
      Runtime: nodejs20.x
```

```
      Code:
        ZipFile: |
          const { SNSClient, PublishCommand } = require('@aws-sdk/client-sns');

          const sns = new SNSClient({ region: 'us-east-1' });

          exports.handler = async (event) => {
            const records = event.Records || [];
            for (const record of records) {
              const messageBody = JSON.parse(record.body);
              const formattedMessage = messageBody.map(item => {
                const description = item.description.S;
                const date = item.date.S;
                return `Event Description: ${description}\nEvent Date: ${date}`;
              }).join('\n\n');

              const params = {
                TopicArn: process.env.SNS_TOPIC_ARN,
                Message: JSON.stringify({
                  default: `Hello,\n\nHere is the event detail:\n\n${formattedMessage}\n\nBest
regards,\nYour Event Notification Service`,
                  email: `Hello,\n\nHere is the event detail:\n\n${formattedMessage}\n\nBest
regards,\nYour Event Notification Service`,
                  subject: 'New Event Notification'
                }),
                MessageStructure: 'json'
              };

              try {
                await sns.send(new PublishCommand(params));
                console.log('Message published to SNS topic');
              } catch (error) {
                console.error('Error publishing message to SNS:', error);
                throw error;
              }
            }
          };
      Environment:
        Variables:
          SNS_TOPIC_ARN: !Ref MyTopic

  SQSTriggerMapping:
    Type: AWS::Lambda::EventSourceMapping
    Properties:
      EventSourceArn: !GetAtt MyQueue.Arn
```

```
    FunctionName: !Ref PublishToSNSLambda
    Enabled: true
    BatchSize: 10

EmailSubscription:
  Type: AWS::SNS::Subscription
  Properties:
    TopicArn: !Ref MyTopic
    Protocol: email
    Endpoint: ax583820@dal.ca

MySecurityGroup:
  Type: AWS::EC2::SecurityGroup
  Properties:
    GroupDescription: 'Allow HTTP, HTTPS, and SSH access'
    SecurityGroupIngress:
      - IpProtocol: 'tcp'
        FromPort: 22
        ToPort: 22
        CidrIp: '0.0.0.0/0'
      - IpProtocol: 'tcp'
        FromPort: 80
        ToPort: 80
        CidrIp: '0.0.0.0/0'
      - IpProtocol: 'tcp'
        FromPort: 443
        ToPort: 443
        CidrIp: '0.0.0.0/0'

MyEC2Instance:
  Type: AWS::EC2::Instance
  Properties:
    InstanceType: t2.micro
    KeyName: !Ref KeyName
    ImageId: ami-06c68f701d8090592
    SecurityGroups:
      - Ref: MySecurityGroup
    UserData:
      Fn::Base64: !Sub |
        #!/bin/bash
        # Update packages
        yum update -y

        # Install Git and Nginx
        yum install -y git nginx
```

```
# Stop and disable Apache if it is running
if systemctl is-active --quiet httpd; then
  systemctl stop httpd
  systemctl disable httpd
fi

# Start and enable Nginx
systemctl start nginx
systemctl enable nginx

# Clone the GitHub repository
git clone https://github.com/AxataDarji/CLoudProject.git /tmp/CLoudProject

# Set up Nginx to serve content from /var/www/html
cat <<EOF > /etc/nginx/nginx.conf
worker_processes 1;

events {
    worker_connections 1024;
}

http {
    include       mime.types;
    default_type  application/octet-stream;

    log_format  main  '\$remote_addr - \$remote_user [\$time_local] "\$request" '
                      '\$status \$body_bytes_sent "\$http_referer" '
                      '"\$http_user_agent" "\$http_x_forwarded_for"';

    access_log  /var/log/nginx/access.log  main;
    error_log   /var/log/nginx/error.log;

    sendfile        on;
    tcp_nopush      on;
    tcp_nodelay     on;
    keepalive_timeout  65;
    gzip  on;

    include /etc/nginx/conf.d/*.conf;

    server {
        listen       80;
        server_name  localhost;
```

```
                location / {
                    root   /var/www/html;
                    index  todo-list.html;
                }

                location ~* \.(jpg|jpeg|gif|png|ico|css|js)$ {
                    expires 30d;
                }
            }
        }
        EOF

        # Create and set permissions for /var/www/html
        mkdir -p /var/www/html
        cp -r /tmp/CLoudProject/* /var/www/html/

        # Set correct permissions
        chown -R nginx:nginx /var/www/html
        chmod -R 755 /var/www/html

        # Restart Nginx to apply configuration
        systemctl restart nginx

        # Clean up
        rm -rf /tmp/CLoudProject

Outputs:
  SQSQueueURL:
    Value: !Ref MyQueue
    Description: URL of the SQS queue

  SNSTopicARN:
    Value: !Ref MyTopic
    Description: ARN of the SNS topic

  ApiEndpoint:
    Value: !Sub "https://${ApiGatewayRestApi}.execute-api.${AWS::Region}.amazonaws.com/dev/"
    Description: "API Gateway endpoint URL"

  InstanceId:
    Description: The Instance ID of the newly created EC2 instance
    Value: !Ref MyEC2Instance
  PublicIP:
    Description: The public IP address of the EC2 instance
    Value: !GetAtt MyEC2Instance.PublicIp
```

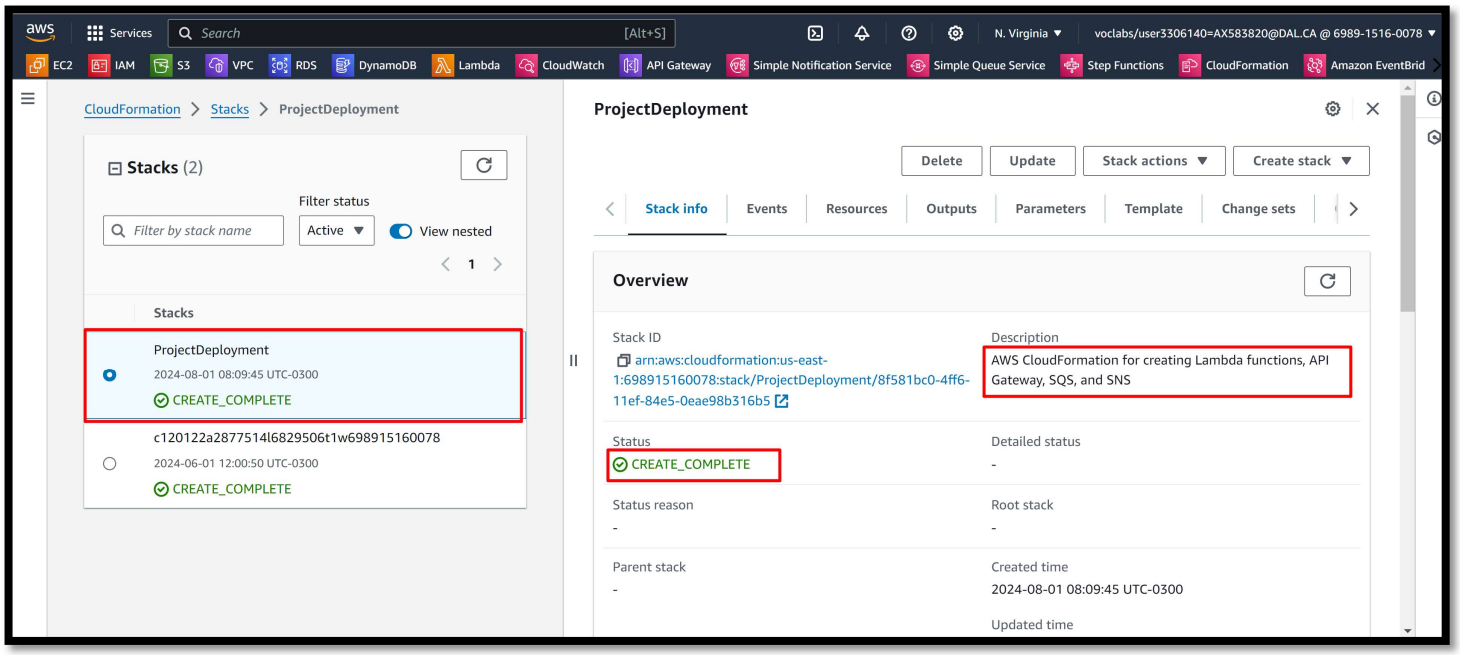

Figure 2: Successful cloud formation stack

Analysis on project's approach on security:

In the current implementation of the diary application, no authorization mechanisms are in place as there is no login functionality. However, several security measures can be added as part of future project improvements to ensure the protection and integrity of the application and its data, especially considering that it interacts with a Dynamo DB database through REST APIs and security groups are already implemented for EC2.

For the application, ensuring data security within the REST API is crucial to protect sensitive user information and interactions with the Dynamo DB database. Implementing HTTPS (SSL/TLS) for all API communications ensures that data transmitted between the client and server is encrypted in transit, safeguarding it from interception and tampering.

Here's a detailed description of the potential security enhancements:

1. **Network Security**
    - **Enhanced VPC Configuration:** Ensure that the application is deployed within a Virtual Private Cloud (VPC) to provide network isolation and make sure that only authorized traffic can access the application and the DynamoDB database.
2. **API Security**
    - **IAM Roles and Policies:** Define and attach IAM roles and policies to the Lambda functions and other AWS resources to restrict access to only those actions and resources necessary for the application. Use the principle of least privilege to minimize the risk of unauthorized access.

3. **Monitoring and Logging**
    - **CloudWatch Logs:** Enable AWS CloudWatch Logs to monitor and log all API requests and responses and other key actions. This helps in detecting and responding to any suspicious activities.
    - **CloudTrail:** Use AWS CloudTrail to log all API calls made within your AWS account. This provides a detailed audit trail of actions taken on the resources, helping in forensic analysis and compliance auditing.
4. **Access Control**
    - **Resource Policies:** Apply resource-based policies to the DynamoDB table to restrict access to specific users or services. This ensures that only authorized entities can perform operations on the database.
    - **API Keys:** While the application does not implement login functionality, consider using API keys to restrict access to the APIs. API keys can be managed and rotated periodically to enhance security.
5. **Login Functionality**
    - **Authentication and Authorization**: Implement a login functionality using AWS Cognito to provide secure user authentication and authorization. This will ensure that only authenticated users can access the application, enhancing the overall security and user experience.

Performance Targets for application:

Here's a detailed analysis of performance targets for the application, covering security, scalability, instance type changes, and database type changes.

1. Security

- **EC2 Instance Security**:
    - **Security Groups**: Ensure that the EC2 instance has a restrictive security group, allowing only necessary inbound and outbound traffic. Typically, allow inbound traffic on port 80 (HTTP) and/or 443 (HTTPS) for web access and restrict other ports.
    - **IAM Roles**: Use IAM roles to provide EC2 instance with permissions to fetch files from the GitHub repository and interact with other AWS services securely.
    - **Nginx Configuration**: Secure Nginx with proper configuration, including SSL/TLS certificates for HTTPS, setting up appropriate headers to prevent attacks like XSS, and limiting the size of request bodies to mitigate DoS attacks.
- **API Gateway Security**:
    - **Authorization**: Implement authorization mechanisms such as IAM roles, Cognito user pools, or API keys to control access to APIs.
    - **Input Validation**: Validate all incoming requests to ensure they conform to expected formats and reject any malicious inputs.
- **Lambda Functions Security**:
    - **Least Privilege**: Apply the principle of least privilege to Lambda functions, ensuring they have only the necessary permissions to perform their tasks.
- **DynamoDB Security**:
    - **Encryption**: Enable encryption at rest for DynamoDB tables to protect data.
    - **Access Control**: Use IAM policies to control access to DynamoDB tables, ensuring only authorized Lambda functions and users can perform operations.
- **SNS and SQS Security**:
    - **Encryption**: Enable encryption for messages in SQS queues and SNS topics.
    - **Access Policies**: Define strict access policies to control who can send and receive messages.

2. Scalability

- **EC2 Instance Scalability**:
  - **Auto Scaling**: Configure Auto Scaling groups to add or remove EC2 instances based on load. Set up scaling policies based on CPU utilization or network traffic.
  - **Elastic Load Balancing**: Use an Elastic Load Balancer (ELB) to distribute incoming traffic across multiple EC2 instances, ensuring high availability and fault tolerance.
- **API Gateway Scalability**:
  - **Automatic Scaling**: API Gateway automatically scales to handle the number of requests it receives. No additional configuration is needed to scale API Gateway.
- **Lambda Functions Scalability**:
  - **Concurrent Executions**: Monitor and adjust the concurrent execution limits for Lambda functions to handle sudden spikes in traffic.
  - **Provisioned Concurrency**: For critical functions that need to respond immediately, consider enabling provisioned concurrency to reduce cold start times.
- **DynamoDB Scalability**:
  - **Auto Scaling**: Enable auto-scaling for read and write capacity to adjust capacity automatically based on traffic patterns.
  - **Global Tables**: Use global tables for multi-region replication to enhance read and write availability.
- **SQS and SNS Scalability**:
  - **Elastic Scalability**: SQS and SNS are designed to handle high-throughput and are inherently scalable. Ensure that downstream processing can keep up with the message flow.

3. Instance Type Changes

- **EC2 Instance Types**:
  - **Scale-Up/Scale-Down**: Depending on the load, can switch to different EC2 instance types. For example, switch from a t2.micro to a t2.medium for higher memory and CPU capacity.
  - **Spot Instances**: For non-critical workloads, consider using spot instances to reduce costs, with the caveat of potential interruptions.

4. Database Type Changes

- **DynamoDB to RDS**:
  - If you find that workload requires complex querying capabilities that DynamoDB cannot efficiently support, consider switching to Amazon RDS (Relational Database Service). RDS supports complex queries, transactions, and joins.
  - **Migration**: Use AWS Database Migration Service (DMS) to facilitate the migration from DynamoDB to RDS with minimal downtime.
- **DynamoDB to Aurora**:
  - Amazon Aurora is a highly available and scalable database that is compatible with MySQL and PostgreSQL. If application requires high read throughput and resilience, Aurora is a suitable choice.
- **Hybrid Approach**:
  - Use a combination of DynamoDB for high-velocity, low-latency workloads and RDS/Aurora for complex transactional operations. This approach allows to leverage the strengths of both NoSQL and relational databases.

By implementing these performance targets, it will ensure that application is secure, scalable, and adaptable to varying workloads and requirements. Regular monitoring, logging, and reviewing of application's performance metrics will help make informed decisions about scaling and optimizing infrastructure.

Analysis on Cost Metrics:

For the current architecture implementation, following is the cost as per AWS price calculator:

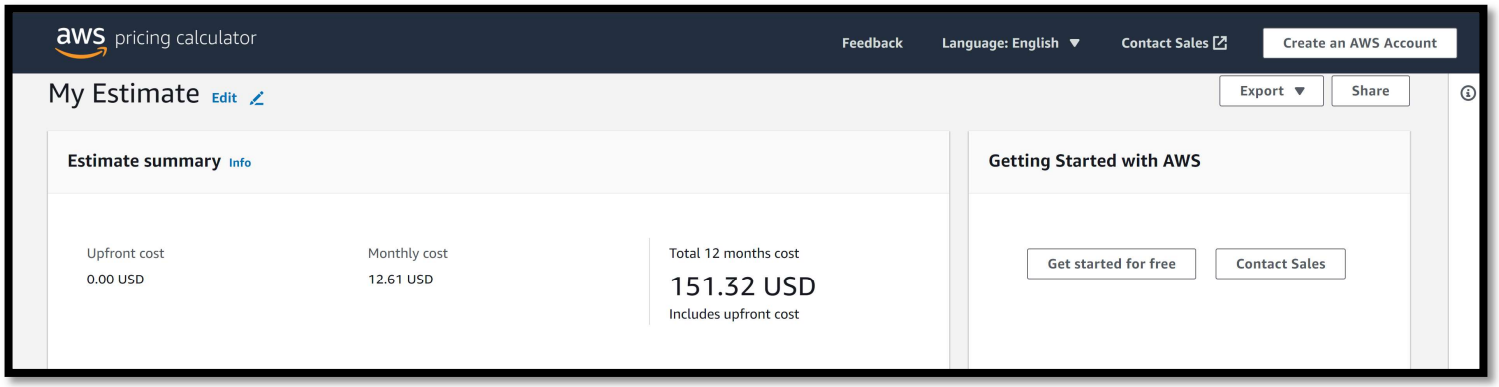

Figure 3: Cost estimation for solution from AWS Pricing calculator

Detailed summary of the cost estimation from AWS price calculator:

Table 2:Total cost for current system architecture

Upfront cost	Monthly cost	Total 12 months cost	Currency
0	12.61	151.32	USD

Following is the breakdown of the total cost:

Table 3: Break down for total cost

Region	Service	Upfront	Monthly	First 12 months total	Currency
US East (N. Virginia)	Amazon EC2	0	6.059	72.71	USD
US East (N. Virginia)	AWS Lambda	0	0	0	USD
US East (N. Virginia)	AWS Lambda	0	0	0	USD
US East (N. Virginia)	AWS Lambda	0	0	0	USD
US East (N. Virginia)	DynamoDB on-demand capacity	0	2.5	30	USD
US East (N. Virginia)	Amazon API Gateway	0	3.15	37.8	USD
US East (N. Virginia)	Amazon Simple Queue Service (SQS)	0	0.9	10.8	USD
US East (N. Virginia)	Standard topics	0	0	0	USD
US East (N. Virginia)	AWS Lambda	0	0	0	USD
US East (N. Virginia)	AWS Lambda	0	0	0	USD
US East (N. Virginia)	Amazon EventBridge	0	0	0	USD

There is no additional cost required.

From the above analysis it is clear that the most expensive service is EC2. By leveraging S3 for hosting static content, we can benefit from a lower cost, simplified management, and efficient scaling, making it a more cost-effective solution compared to using EC2 for the same purpose.

Rather than hosting application on EC2, S3 can be utilized for hosting application. If S3 replaces EC2, then following is the updated cost:

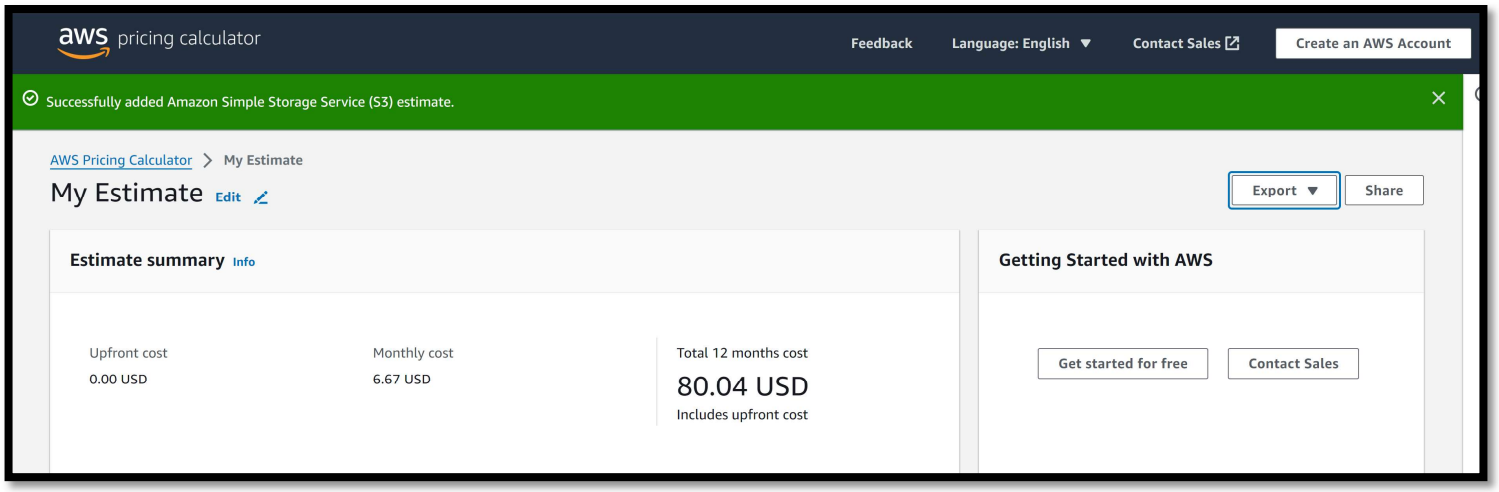

Figure 4: Cost estimation post replacing EC2 with S3

Detailed summary of the cost estimation from AWS price calculator:

Table 4: Total cost for proposed system architecture

Upfront cost	Monthly cost	Total 12 months cost	Currency
0	6.67	80.04	USD

Following is the breakdown of the total cost for proposed architecture:

Table 5: Break down for proposed architecture total cost

Region	Service	Upfront	Monthly	First 12 months total	Currency
US East (N. Virginia)	AWS Lambda	0	0	0	USD
US East (N. Virginia)	AWS Lambda	0	0	0	USD
US East (N. Virginia)	AWS Lambda	0	0	0	USD
US East (N. Virginia)	DynamoDB on-demand capacity	0	2.5	30	USD
US East (N. Virginia)	Amazon API Gateway	0	3.15	37.8	USD
US East (N. Virginia)	Amazon Simple Queue Service (SQS)	0	0.9	10.8	USD
US East (N. Virginia)	Standard topics	0	0	0	USD
US East (N. Virginia)	AWS Lambda	0	0	0	USD
US East (N. Virginia)	AWS Lambda	0	0	0	USD
US East (N. Virginia)	Amazon EventBridge	0	0	0	USD
US East (N. Virginia)	S3 Standard	0	0.12	1.44	USD

Cost and purchase for reproduce in private cloud:

To reproduce the architecture of the application in a private cloud while providing the same level of availability as current cloud implementation, organization would need to invest in both hardware and software infrastructure. Below is a breakdown of the key components would need and a rough estimate of the associated costs.

Hardware Components

1. **Servers**
    - **Compute Servers**: To handle web server, application server, and database functions.
        - Example: Dell PowerEdge R740
        - **Cost**: Approximately $5,000 per server
        - **Quantity**: 3 (for redundancy and load balancing)
        - **Total**: $15,000
2. **Storage**
    - **Storage Area Network (SAN)**: For storing database and application data.
        - Example: Dell EMC Unity XT 480F
        - **Cost**: Approximately $20,000
        - **Quantity**: 1
        - **Total**: $20,000
3. **Networking**
    - **Network Switches**: To connect servers and storage.
        - Example: Cisco Catalyst 9300 Series
        - **Cost**: Approximately $10,000 per switch
        - **Quantity**: 2 (for redundancy)
        - **Total**: $20,000
4. **Load Balancer**
    - **Hardware Load Balancer**: To distribute traffic across multiple servers.
        - Example: F5 BIG-IP LTM
        - **Cost**: Approximately $15,000
        - **Quantity**: 1
        - **Total**: $15,000
5. **Backup and Disaster Recovery**
    - **Backup Server**: For regular backups of critical data.
        - Example: HPE StoreOnce 3640
        - **Cost**: Approximately $15,000
        - **Quantity**: 1
        - **Total**: $15,000

Software Components

1. **Operating Systems**
    - **Linux Distribution**: Such as Ubuntu or CentOS
        - **Cost**: Free or minimal support cost
2. **Database**
    - **NoSQL Database Software**: Such as MongoDB or Cassandra
        - **Cost**: Free (Community Edition) or Enterprise Edition (Approximately $5,000/year)
3. **Web Server**
    - **Nginx**: For hosting the frontend application.
        - **Cost**: Free or minimal support cost
4. **Application Server**
    - **Node.js**: For running backend services.

        - **Cost**: Free
5. **Queue Management**
    - **RabbitMQ or Apache Kafka**: For message queue management.
        - **Cost**: Free (Community Edition) or Enterprise Edition (Approximately $1,500/year)
6. **Email Notifications**
    - **SMTP Server**: Such as Postfix or Sendmail for sending emails.
        - **Cost**: Free
7. **API Management**
    - **API Gateway Software**: Such as Kong or Tyk
        - **Cost**: Free (Community Edition) or Enterprise Edition (Approximately $3,000/year)
8. **Monitoring and Logging**
    - **Monitoring Tools**: Such as Prometheus and Grafana.
        - **Cost**: Free
    - **Logging Tools**: Such as ELK Stack (Elasticsearch, Logstash, Kibana).
        - **Cost**: Free

Additional Costs

1. **Maintenance and Support**
    - **Annual Maintenance Contracts**: For hardware and software.
        - **Cost**: Approximately 20% of initial hardware and software costs
        - **Total**: $17,000/year
2. **Power and Cooling**
    - **Cost**: Dependent on data center location and size.
        - **Estimate**: $5,000/year
3. **IT Staff**
    - **System Administrators and Network Engineers**: To manage and maintain the infrastructure.
        - **Cost**: Approximately $100,000/year per engineer
        - **Quantity**: 2
        - **Total**: $200,000/year

Summary of Costs

- **Initial Hardware Costs**: $85,000
- **Annual Software and Maintenance Costs**: $36,000/year
- **Annual IT Staff Costs**: $200,000/year
- **Total First Year Cost**: $321,000
- **Subsequent Annual Costs**: $236,000/year

This rough estimate provides a high-level view of the costs involved in replicating the application's architecture in a private cloud environment. The actual costs can vary significantly based on the specific requirements, scale, and vendor choices.

Monitoring Mechanism:

In the current application architecture, the most important cloud mechanism to monitor closely to avoid unexpected cost escalation is **Amazon EC2**. Here's why:

Reasons for Focusing on Amazon EC2

1. **Variable Cost**: Unlike some other AWS services, EC2 instances can have significant cost variability based on factors like instance type, usage hours, and data transfer. If application experiences unexpected traffic spikes or if instances are not properly scaled or managed, costs can quickly escalate.
2. **Scaling Costs**: EC2 instances are often used for handling web servers and backend services. If application grows or has sudden spikes in demand, might need to scale up or out, which can lead to higher costs. Monitoring and setting up auto-scaling policies can help control these costs.
3. **Data Transfer Costs**: EC2 instances might incur substantial costs related to data transfer, both inbound and outbound, especially if application has high traffic or transfers large amounts of data. This can add up quickly if not monitored.
4. **Idle or Underutilized Instances**: If EC2 instances are left running when not needed or if they are over-provisioned, you may end up paying for unused capacity. Regular monitoring can help identify and terminate idle or underutilized instances.

Monitoring and Cost Management Strategies

1. **CloudWatch Alarms**: Set up Amazon CloudWatch alarms to monitor metrics like CPU usage, network traffic, and instance health. Configure notifications to alert if metrics exceed thresholds that could indicate potential cost overruns.
2. **Cost Explorer and Budgets**: Use AWS Cost Explorer to analyze historical spending and trends. AWS Budgets allows to set custom cost and usage budgets with alerts to notify when you approach or exceed your budget thresholds.
3. **Auto-Scaling**: Implement auto-scaling policies to dynamically adjust the number of EC2 instances based on demand. This helps prevent over-provisioning and ensures you're only paying for the resources you need.
4. **Cost Allocation Tags**: Apply cost allocation tags to your EC2 instances to track costs associated with different projects or environments. This helps identify which parts of your application are generating the most costs.
5. **Review Reserved Instances**: Consider purchasing Reserved Instances or Savings Plans for predictable workloads to reduce costs compared to on-demand pricing.

Future Enhancements and Development Roadmap:

Continuing development on the application could involve several enhancements and new features. Here's a look at potential features and the cloud mechanisms might use to implement them:

1. User Authentication and Authorization
Feature: Add user authentication and authorization to allow multiple users to have personalized diaries and manage their own events securely.
Cloud Mechanisms:

- **AWS Cognito**: Provides user sign-up, sign-in, and access control features.
- **IAM Roles**: Manage permissions for accessing DynamoDB or other services based on user roles.

2. Enhanced User Interface and Experience
Feature: Improve the frontend with a richer user interface, responsive design, and additional user-friendly features.
Cloud Mechanisms:

- **Amazon S3 + CloudFront**: Host static assets (HTML, CSS, JavaScript) on S3 and use CloudFront for fast, global content delivery.
- **AWS Amplify**: Use it for developing and deploying modern web apps with additional frontend capabilities.

3. Mobile Application
Feature: Develop a mobile app to access the diary on the go, with push notifications and offline capabilities.
Cloud Mechanisms:

- **AWS Amplify**: Provides support for building and deploying mobile applications.
- **AWS Pinpoint**: For mobile app analytics and push notifications.

4. **Advanced Reminder Features**
Feature: Introduce advanced reminder features, such as customizable reminder intervals, integration with calendars, and smart reminders based on user behavior.
Cloud Mechanisms:

- **Amazon EventBridge**: Create rules for more complex scheduling and reminders.
- **AWS Lambda**: Enhance Lambda functions to handle advanced reminder logic.

5. **Data Analytics and Reporting**
Feature: Provide analytics and reporting on user activities, event statistics, and usage patterns.
Cloud Mechanisms:

- **Amazon QuickSight**: For creating interactive dashboards and visualizations.
- **AWS Glue**: For ETL processes to prepare data for analytics.

6. Search Functionality
Feature: Implement search capabilities for users to find specific events quickly.
Cloud Mechanisms:

- **Amazon Elasticsearch Service (Amazon OpenSearch Service)**: For full-text search and analytics.
- **AWS Lambda**: To integrate search queries with existing infrastructure.

7. **Backup and Recovery**
Feature: Implement a robust backup and recovery solution to ensure data durability and availability.
Cloud Mechanisms:

- **AWS Backup**: Manage backups for DynamoDB and other AWS services.
- **Amazon S3**: Store backups and use lifecycle policies to manage data retention.

Application Screenshots:

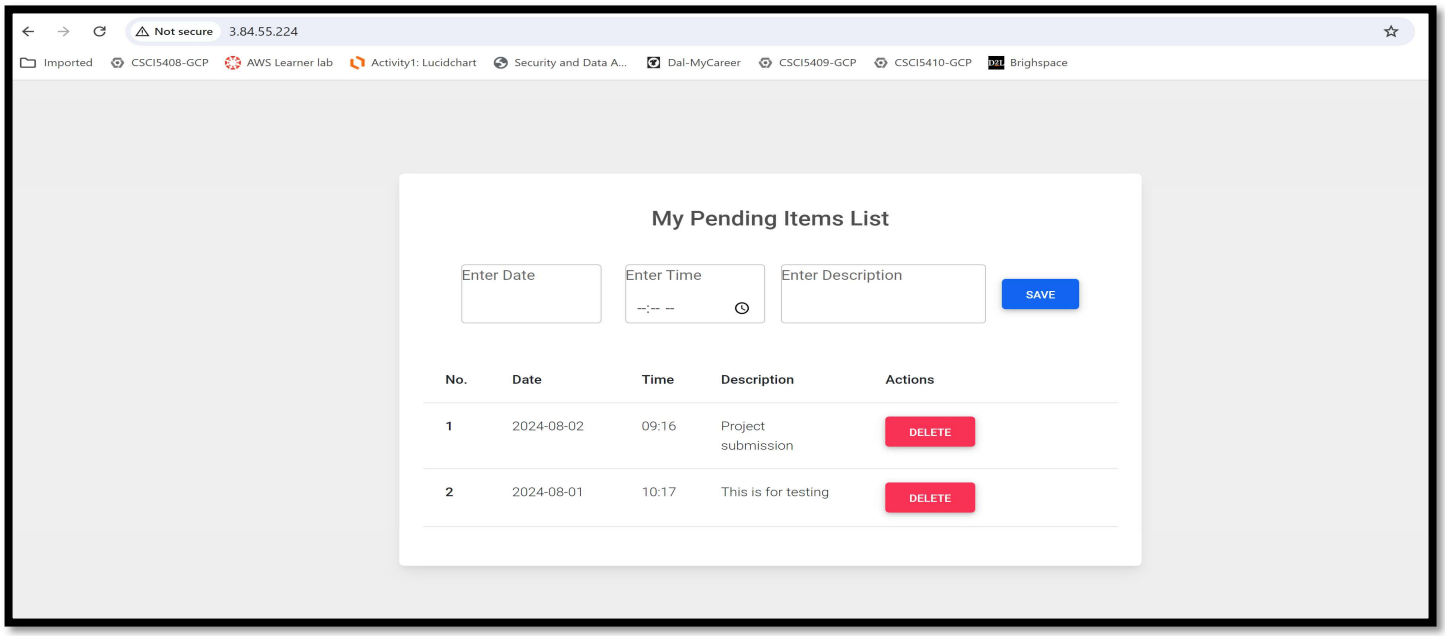

Figure 5: Landing page of application

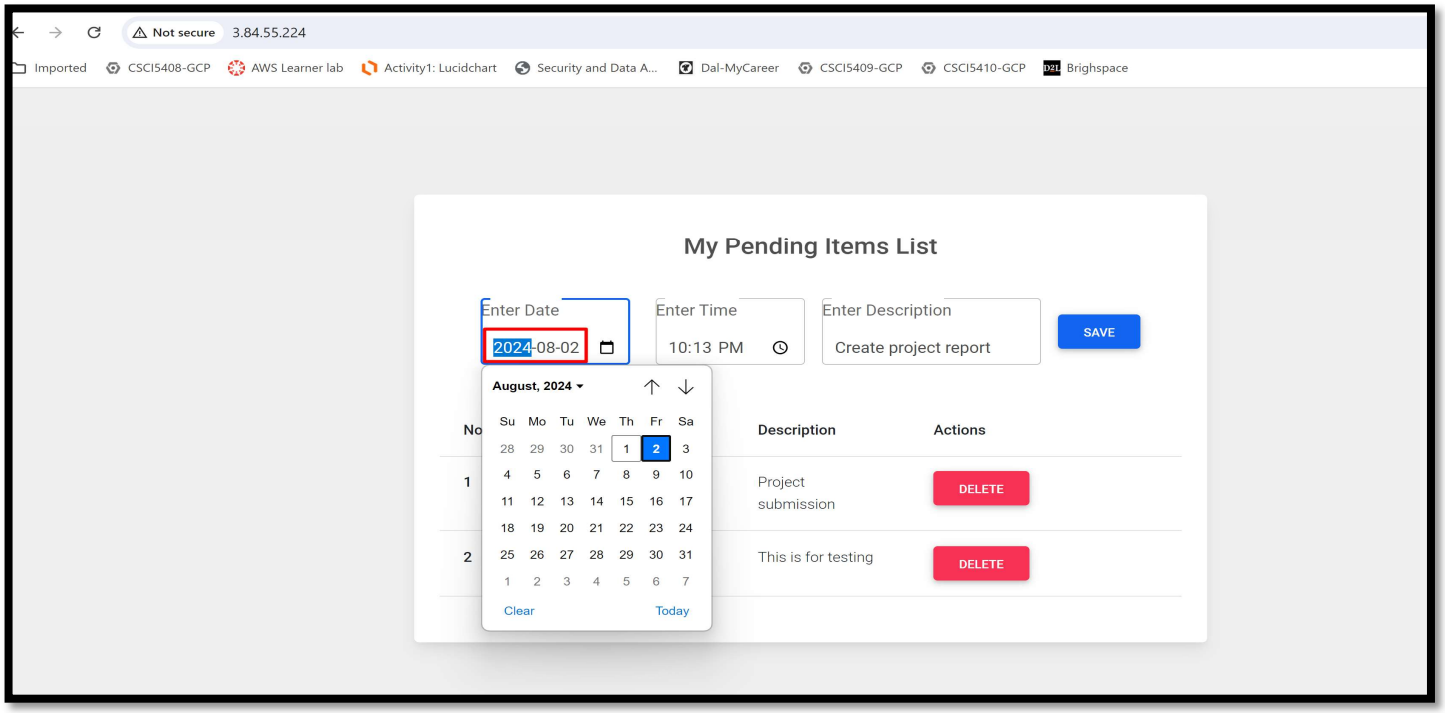

Figure 6: Selection of date for adding new event/pending item

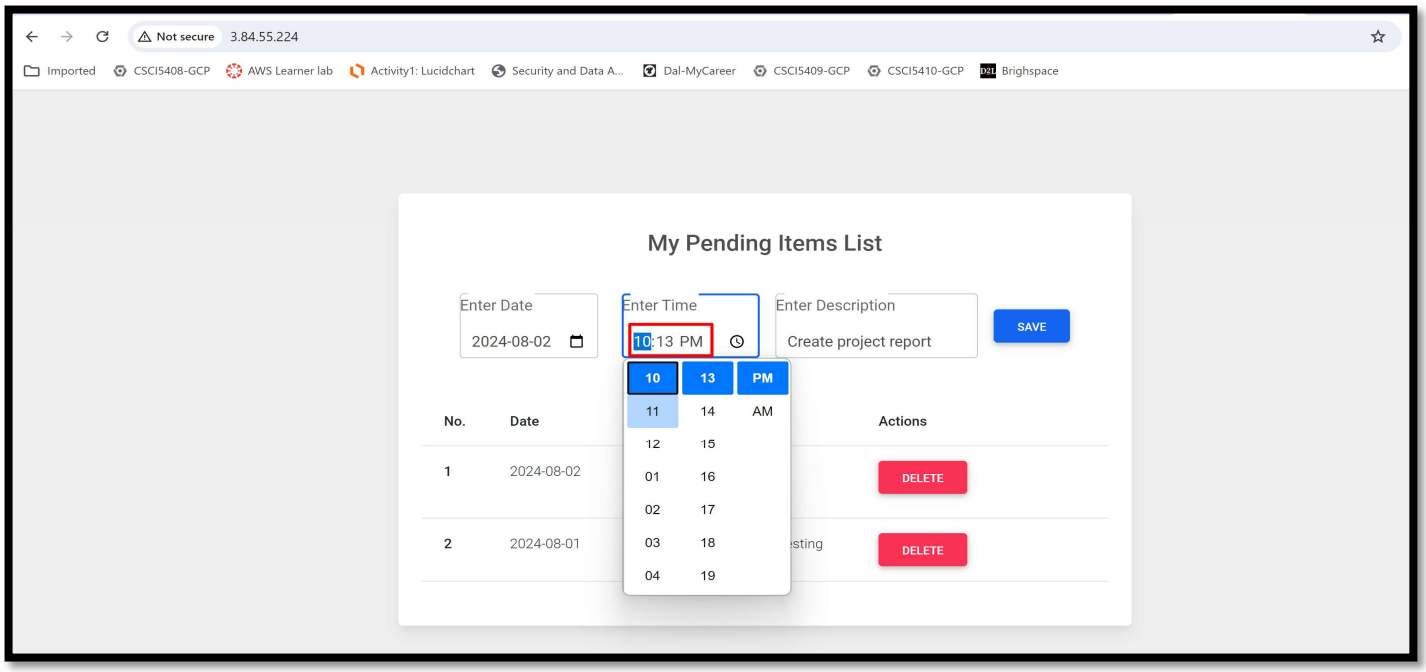

Figure 7:Selection of time for adding new event/pending item

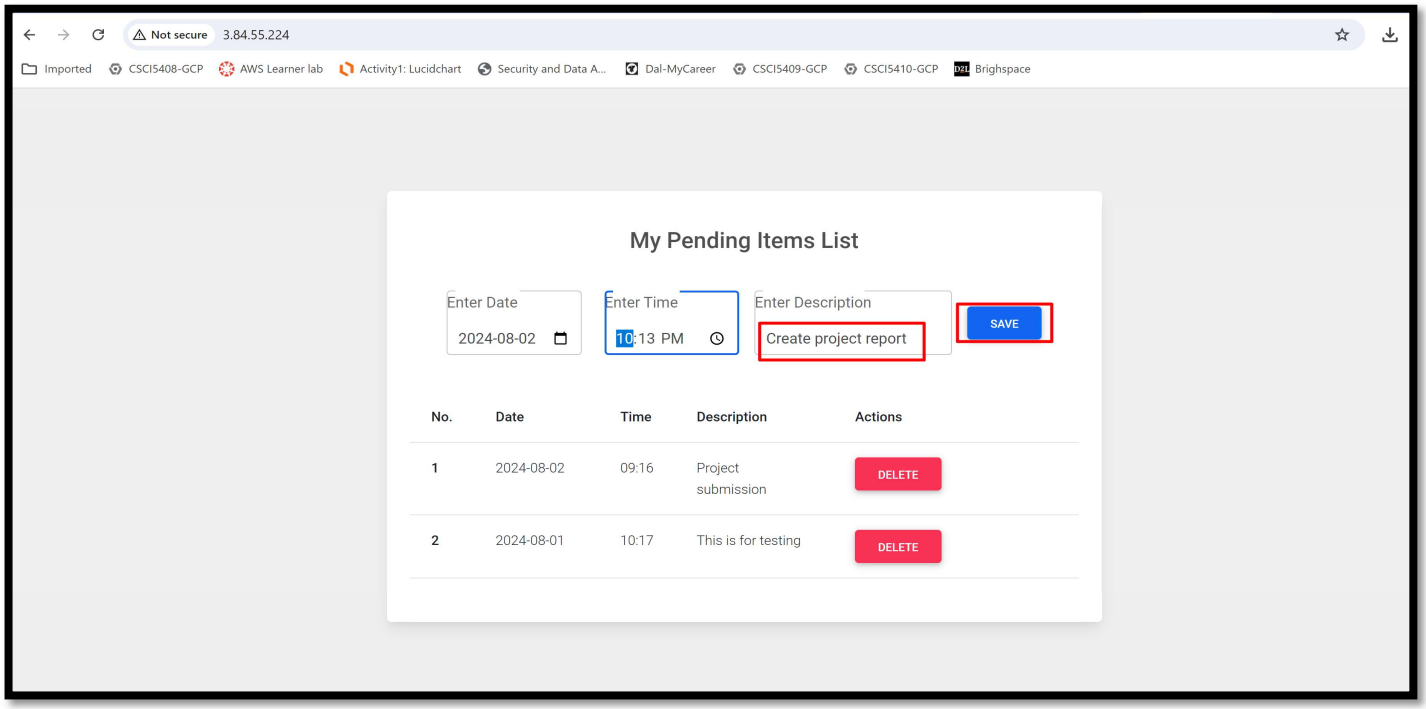

Figure 8:Description for adding new pending item

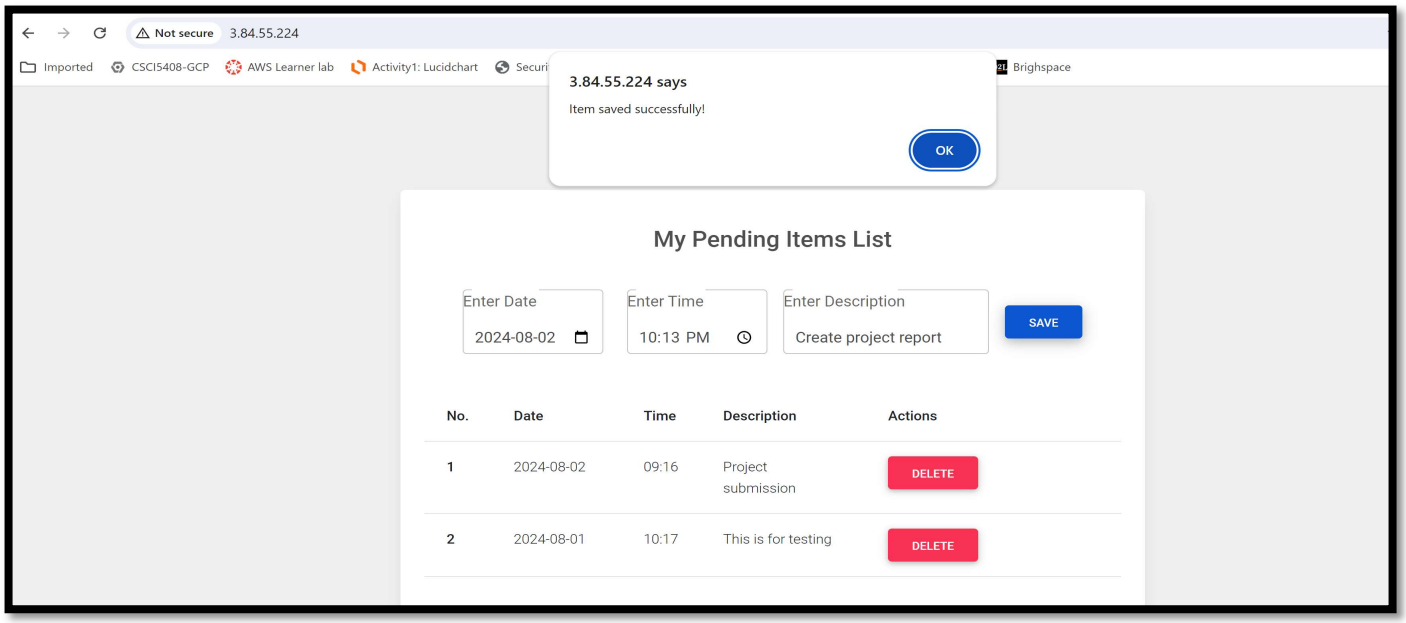

Figure 9: Successful addition of new pending item

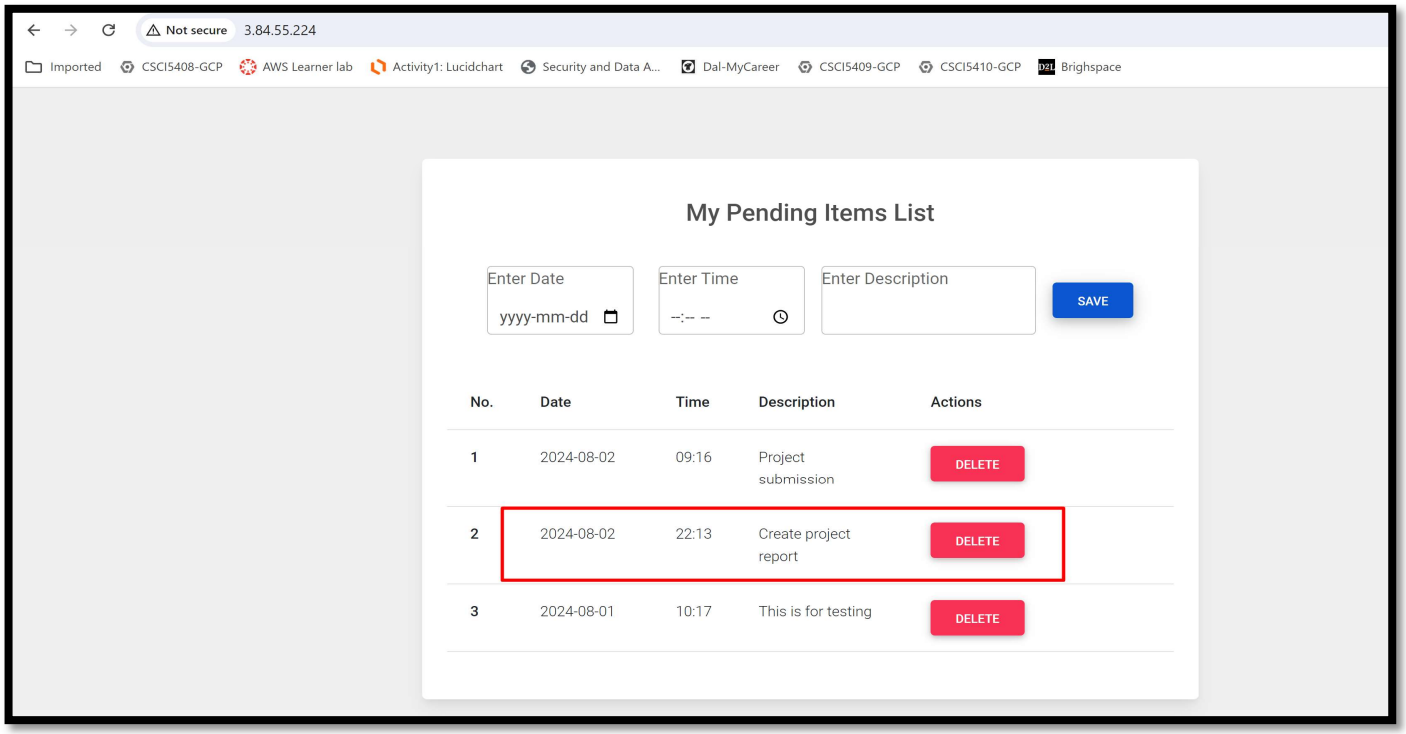

Figure 10: Successful addition of task in list

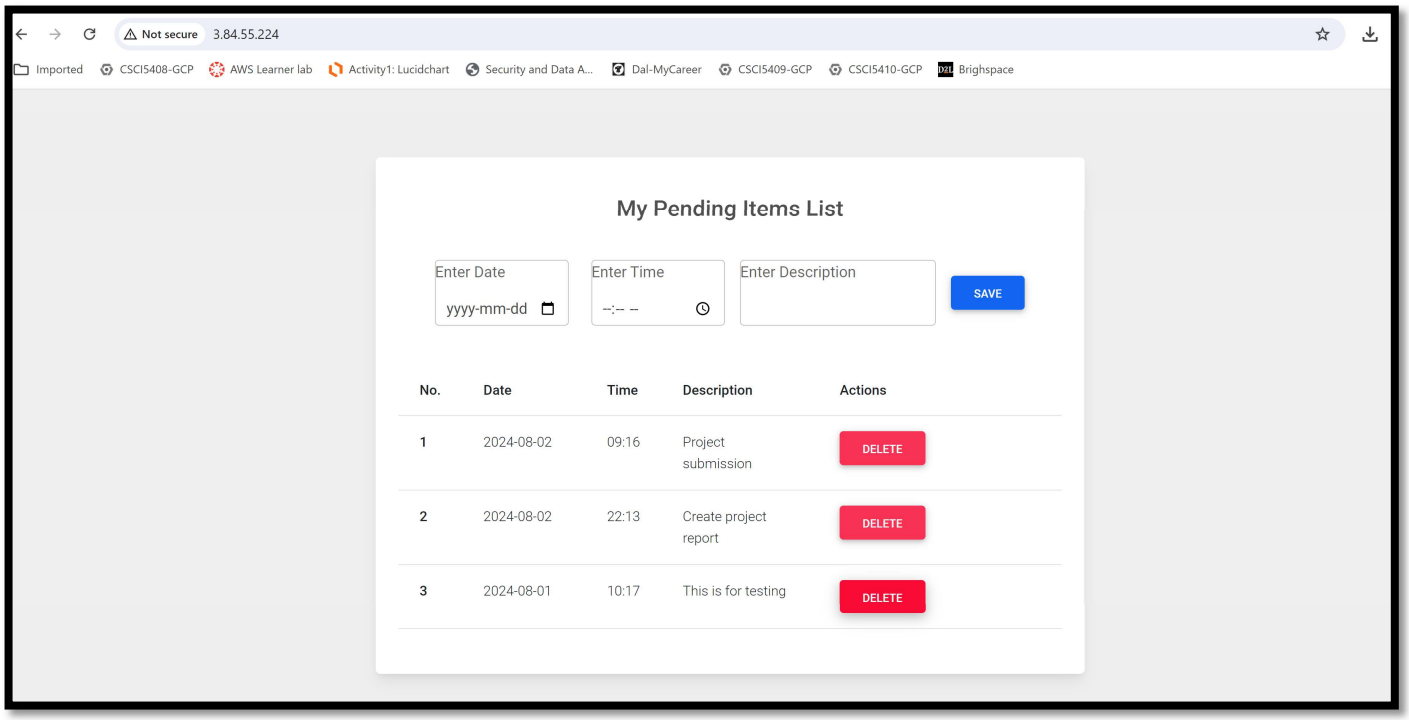

Figure 11: Click on Delete button to delete the item

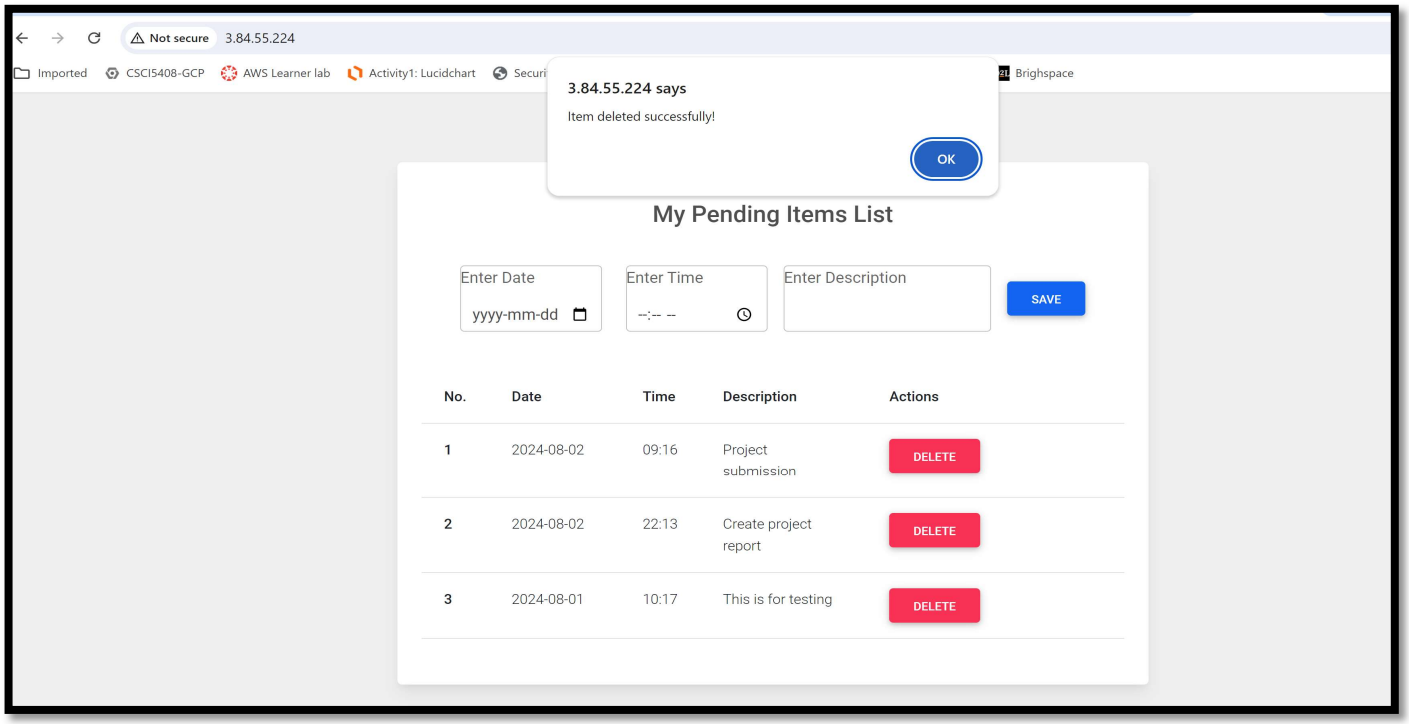

Figure 12: Successful deletion of item

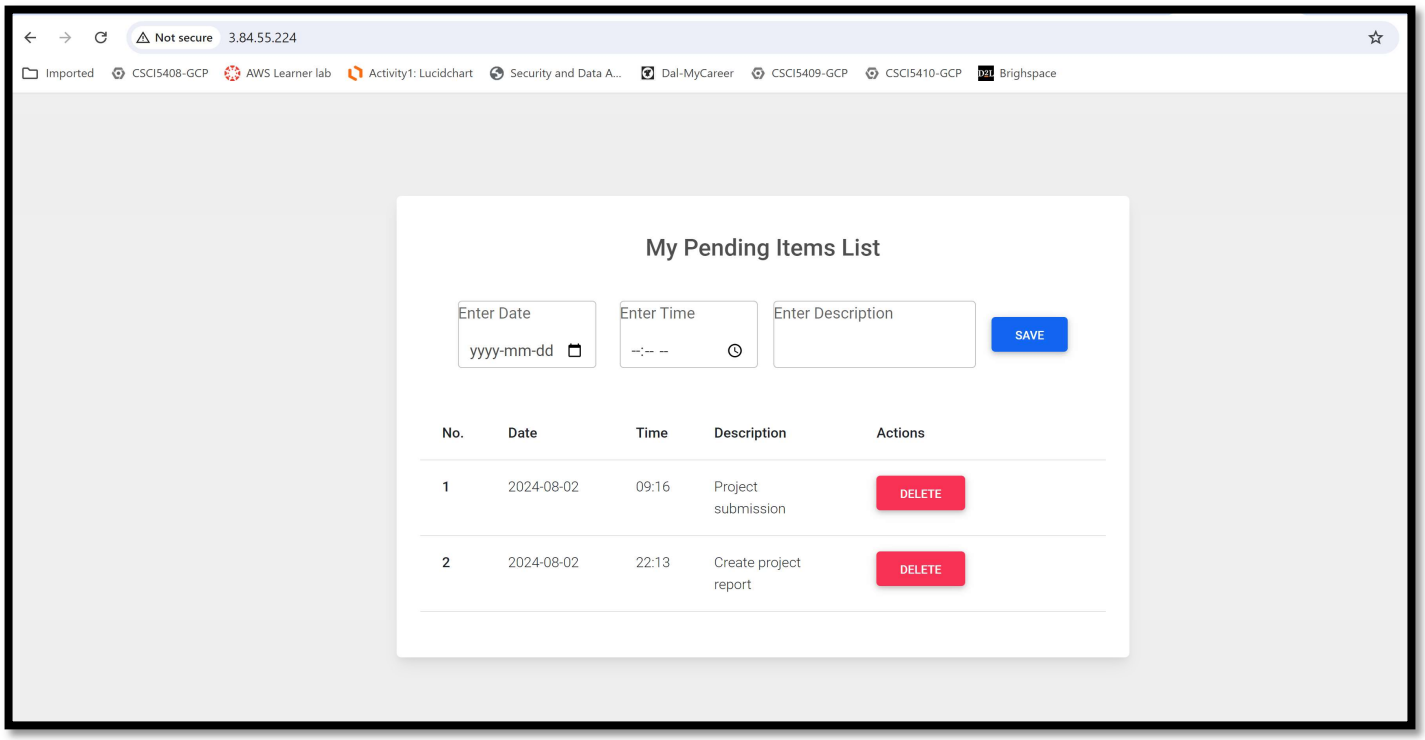

Figure 13: Updated list after deletion of item

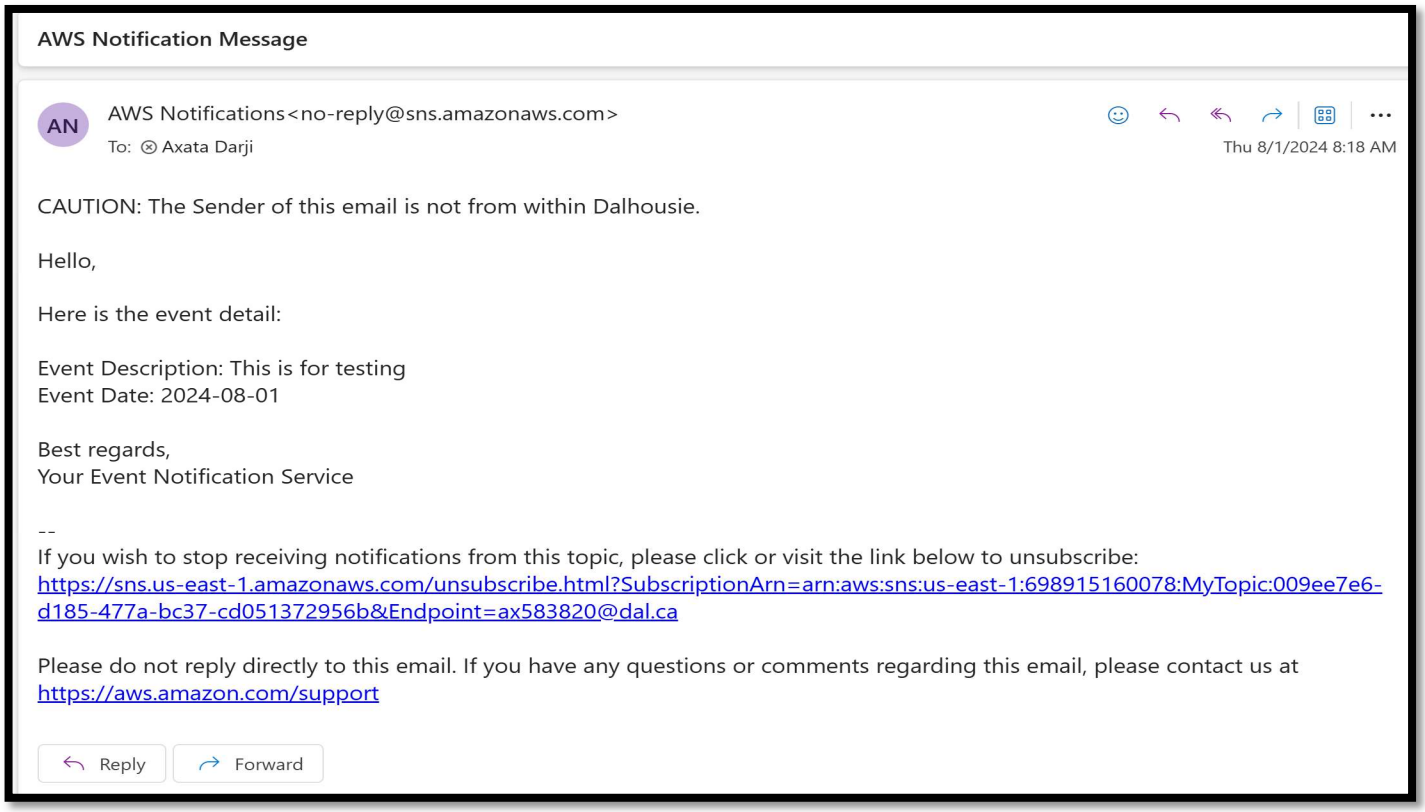

Figure 14: Reminder email with today's event or pending item

References:

[1] "aws-cloudformation-templates," *aws-cloudformation.* [Online]. Available: https://github.com/aws-cloudformation/aws-cloudformation-templates/ [Accessed : July 28, 2024]

[2] OpenAI, "ChatGPT," OpenAI, 2024. [Online]. Available: https://www.openai.com/chatgpt. [Accessed: August 2, 2024].

[3] "MDBootstrap 5 download & installation guide," *MDB - Material Design for Bootstrap.* [Online]. Available: https://mdbootstrap.com/docs/standard/getting-started/installation/ [Accessed : June 05, 2024]

[4] "AWS Lambda – Serverless Compute - Amazon Web Services," *Amazon Web Services, Inc.* [Online]. Available: https://aws.amazon.com/lambda/ [Accessed : July 05, 2024]

[5] "AWS SDK for JavaScript," *Amazon Web Services, Inc.* [Online]. Available: https://aws.amazon.com/sdk-for-javascript/ [Accessed : July 05, 2024]

[6] "JS Foundation, jQuery," *Jquery.com*, [Online]. Available: https://jquery.com/ [Accessed : June 15, 2024]

[7] "Enabling CORS for a REST API resource - Amazon API Gateway," *docs.aws.amazon.com*. [Online]. Available: https://docs.aws.amazon.com/apigateway/latest/developerguide/how-to-cors.html [Accessed : July 13, 2024]

[8] "No 'Access-Control-Allow-Origin' header is present on the requested resource from an AWS API Gateway," *Stack Overflow*. [Online]. Available: https://stackoverflow.com/questions/64087985/no-access-control-allow-origin-header-is-present-on-the-requested-resource-fro [Accessed : July 16, 2024]