DALHOUSIE UNIVERSITY

Sprint 3 Team Report

Team 36

Christin Saji - B00977669
Jay Jagdishbhai Patel - B00981520
Jeet Jani - B00972192
Kuldeep Rajeshbhai Gajera – B00962793

GitLab Repo Link: https://git.cs.dal.ca/patel45/serverless_group_36/-/tree/main?ref_type=heads

Table of Contents

Details of Research

Module 3: Message Passing		
Research Area	Source	URL
Creating a Pub/Sub Topic	Google Cloud Pub/Sub	https://cloud.google.com/pubsub/docs/create-topic
Publishing in a Pub/Sub Topic from AWS	Google Cloud Pub/Sub	https://cloud.google.com/pubsub/docs/publisher
Setting a Cloud Function as a Subscriber	Google Cloud – Cloud Functions	https://cloud.google.com/functions/docs/calling/pubsub#:~:text=If%20you%20are%20deploying%20using%20the%20Google%20Cloud%20console%2C%20you,trigger%20calls%20to%20your%20function.
Storing Information in Firestore using Cloud Functions	Firebase - Firestore	https://firebase.google.com/docs/firestore/extend-with-functions
Implementing Chat System using Firestore	FreeCodeCamp	https://www.freecodecamp.org/news/building-a-real-time-chat-app-with-reactjs-and-firebase/
Module 4: Notification		
Research Area	Source	URL
What is Amazon SNS?	AWS Documentation	https://docs.aws.amazon.com/sns/latest/dg/welcome.html
What is Amazon SQS?	AWS Documentation	https://docs.aws.amazon.com/AWSSimpleQueueService/latest/SQSDeveloperGuide/welcome.html
Sending Amazon SNS messages to an Amazon SQS queue in different accounts	AWS Documentation	https://docs.aws.amazon.com/sns/latest/dg/sns-sqs-as-subscriber.html

How to set up a serverless notification system on AWS	Medium, Towards Data Science	https://lo0o0p.medium.com/build-a-serverless-push-and-sms-notification-service-on-aws-f92fdbf7ece3

Module 5: Data Analysis and Visualization

Research Area	Source	URL
AWS Cognito SDK to interact with Cognito Services	NPM (Node Package Manager) Repository	https://www.npmjs.com/package/amazon-cognito-identity-js
Exploring different APIs and Features available to call for different task	GCP Cloud Natural Processing Documentation	https://cloud.google.com/natural-language/docs/basics#interpreting_sentiment_analysis_values
Calling analyzeSentiment API using NodeJS.	GCP Cloud GCP Cloud Natural Processing Documentation	https://cloud.google.com/natural-language/docs/analyzing-sentiment
To Know more about NPM library offered by Google for NLP Tasks	NPM Documentation of GCP NLP Library	https://www.npmjs.com/package/@google-cloud/language
Cognito Post Authentication Lambda Trigger	AWS Documentation	https://docs.aws.amazon.com/cognito/latest/developerguide/user-pool-lambda-post-authentication.html
Update Google Sheets using Apps Script	Google Sheets Documentation	https://developers.google.com/sheets/api/guides/batchupdate
Connect Google Sheets with Locker Studio for Data Analytics and Visualizations	Locker Studio Support	https://support.google.com/looker-studio/answer/7450249?hl=en#zippy=%2Cin-this-article
Features	Locker Studio Documentation	https://developers.looker.com/embed/getting-started

Module 6: Web Application Building and Deployment

Research Area	Source	URL
Google Cloud Run	Google Cloud Run Documentation	https://cloud.google.com/build/docs/deploying-builds/deploy-cloud-run
Using Google Cloud Build Service to build Docker Image and push to Google Artifact Repository	Google Cloud Build Documentation	https://cloud.google.com/build/docs/building/build-containers
Create a Gitlab pipeline to push to Google Artifact Registry	Gitlab Documentation	https://docs.gitlab.com/ee/tutorials/create_gitlab_pipeline_push_to_google_artifact_registry/
Use Gitlab CI/CD to build Application	Gitlab Documentation	https://docs.gitlab.com/ee/topics/build_your_application.html

How we have Implemented:

Module 1: User Management & Authentication Module

We have extended this module further by adding Post Authentication Trigger which will log the user login statistics in the DynamoDB Table. This stored login statistics will help us to visualize further in Data analysis and Visualization Module.

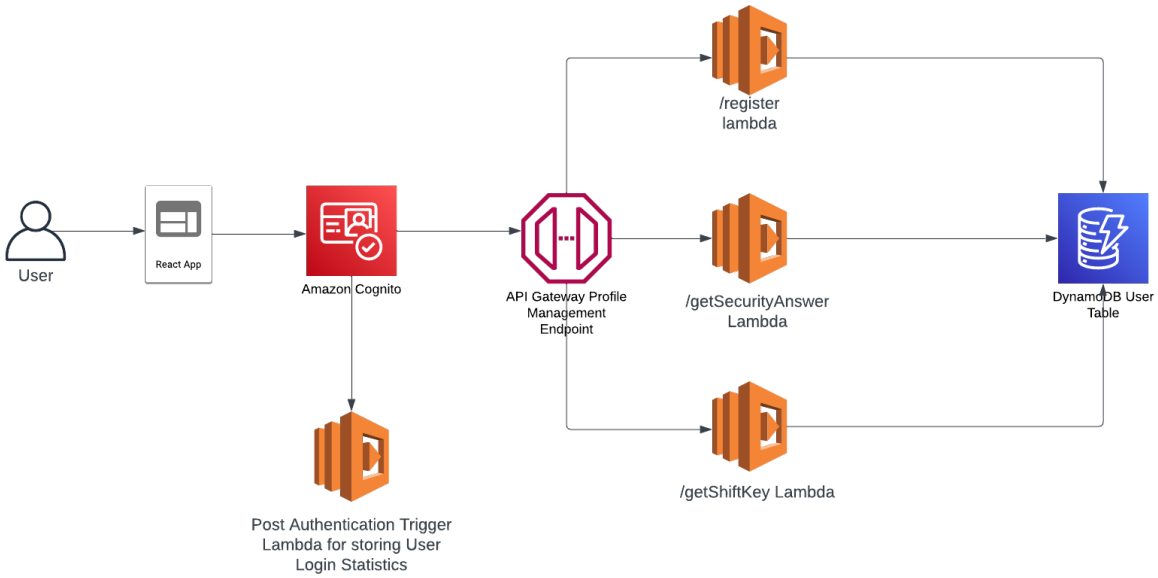

Figure 1: Architecture Diagram of User Management and Authentication Module

Module 2: Virtual Assistance Module

This module was extended in Sprint 3 to fulfil the customer concern functionality. HandoffCustomerSupport was modified to publish the user and concern details to GCP Pub/Sub topic [1][2]. IntentHandler and FetchBookingDetails were modified to support the changes made for role-based response generation.

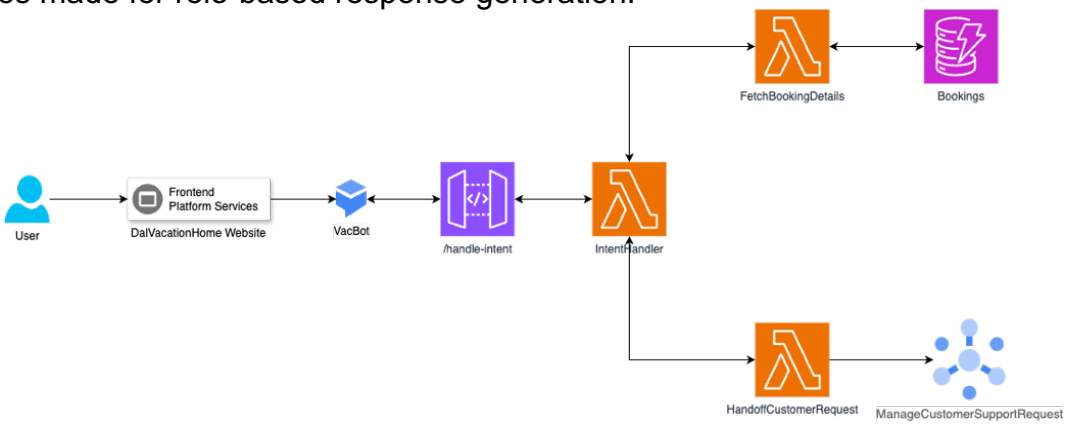

Figure 2: Architecture Diagram of Virtual Assistant Module

Module 3: Message Passing

As implemented in the previous module, when a user posts their concern to the chatbot, the HandoffCustomerSupport Lambda function dealt with sending back an appropriate response to the user. This module comprises two parts - publishing the concern to create a ticket for the user and implementing a live chat.

The first functionality is an extension of the second module and closely associated with it. HandoffCustomerSupport will publish the user details like the email id, concern along with other metadata through a GCP Pub/Sub topic - ManageCustomerSupportRequest [1][2]. A Cloud Function – AgentAssigner, is subscribed to this topic and receives the information passed on by it [3].

Following that, AgentAssigner will fetch the list of admins from the DynamoDB table User, using a Lambda Function – AdminFetcher, and assign a random agent to this customer support request. It will create and store this new concern in a collection called Issues inside the default Firestore database [4]. Each support request is stored as a separate issue, having a unique, auto-assigned document ID. That concludes the first functionality.

The second feature involves creating a new page in the frontend which users can use to keep track of all their posted concerns and chat with the assigned agent. Every message that is sent by the user or the agent, is sent as an event to a Cloud Function called ChatLogger. This adds every message as a JSON object inside the chat array, created by AgentAssigner. Another Cloud Function – ChatRetriever fetches all the previous chats between them.

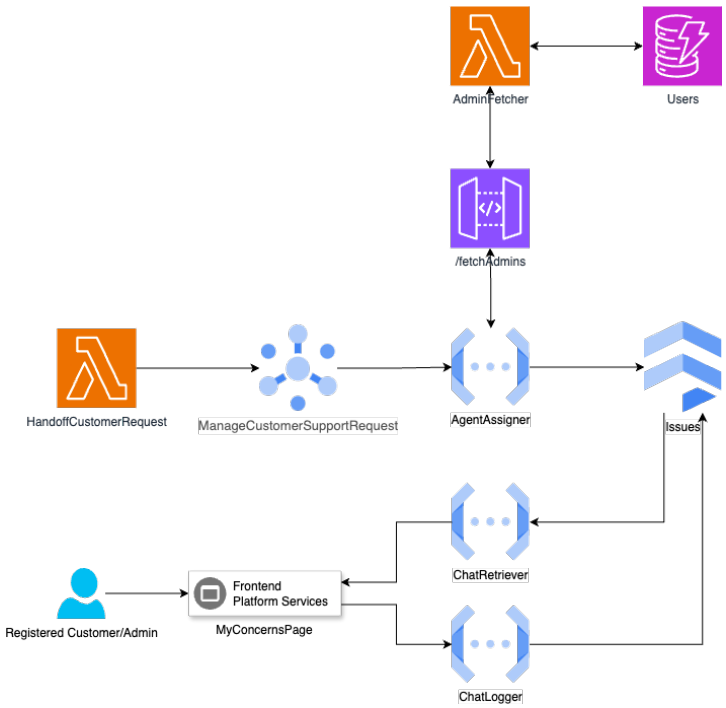

Figure 3: Architecture Diagram of Message Passing Module

Module 4: Notification

In the implementation of our notification module, we leveraged a combination of Amazon Simple Notification Service (SNS), Amazon Simple Queue Service (SQS), and AWS Lambda to create a robust and scalable notification system [6][7]. This design choice was driven by the need for reliable real-time communication to users for various system events such as registration confirmations and booking updates. SNS facilitates the management and dispatch of notification messages to different channels, ensuring broad reach and high deliverability. Each notification type is associated with a specific SNS topic that routes messages to an SQS queue, which serves as a buffer to absorb any fluctuations in message volume and maintain system resilience against spikes in demand [8].

Upon arrival of messages in SQS, AWS Lambda functions are triggered to process these messages asynchronously [9]. This setup allows for the decoupling of notification logic from the application's core functionality, enhancing maintainability and scalability. The Lambda functions handle the task of parsing messages from SQS, formatting them appropriately, and executing the delivery mechanism, which could range from sending emails, pushing mobile notifications, or interfacing with other AWS services for additional actions. The use of SQS ensures that no message is lost, even during periods of high load, providing a fail-safe mechanism that reinforces the reliability of our notification system. This architecture not only meets our current needs but is also designed to scale seamlessly as user demand grows, exemplifying a modern cloud-native approach to application development and system design.

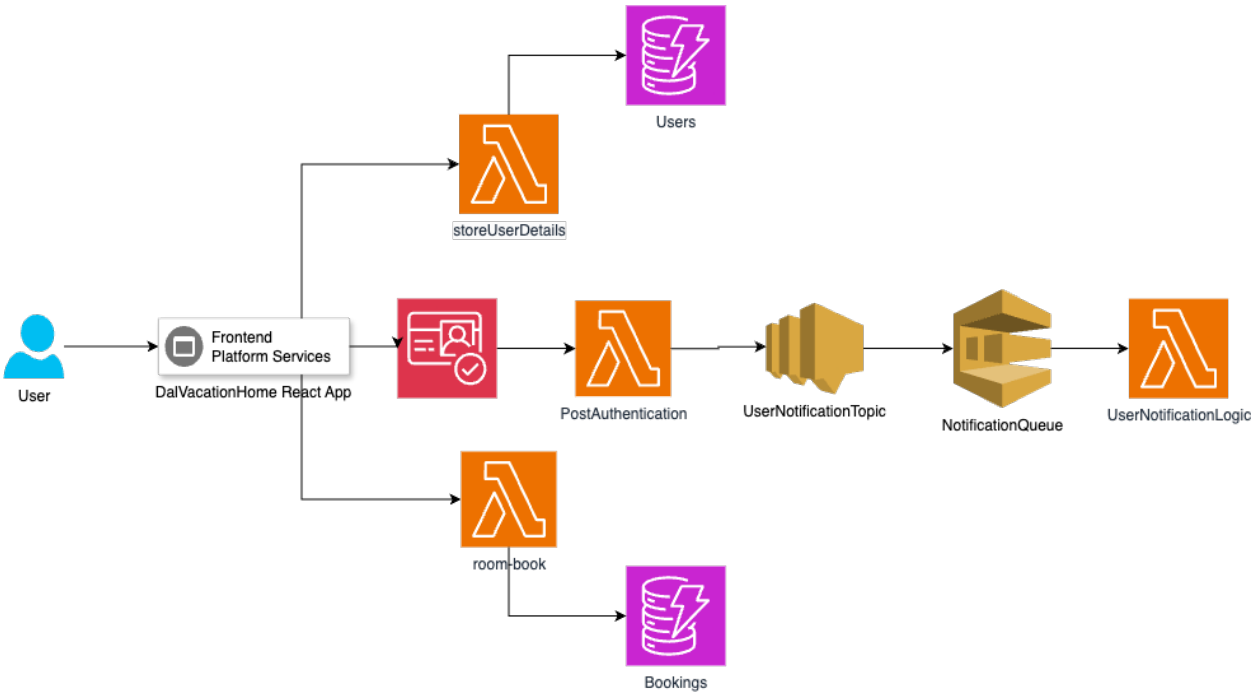

Figure 4: Architecture diagram for notification module

Module 5: Data Analysis and Visualization

After going over the official documentation of various cloud services, we got an overall idea of how we will be implementing the Data Analytics and Visualization module.

We have used Google Looker Studio, which allows us to present our raw data in informative, easy-to-read ways such that we can get insights from the data by visualizing the data using different diagrams and displayed pie charts, and interactive table. This tool is used for Data Analytics and Visualization purposes [16][17].

This tool provides a feature to visualize Google Sheets, so we have written a Google App Script that will periodically fetch data from Lambda function via API Gateway and populate the Google Sheets. We then use Looker Studio to analyze that Google Sheet and then embed that Looker Studio Report in React App using an iFrame [15].

According to the requirement, we have displayed Login Statistics of the users on the Admin Page. We have stored login statistics in the AWS DynamoDB after Login. After the successful login into the application using AWS Cognito, we have set up a post-authentication trigger which will run a Lambda function to store data about Login Statistics of the user in DynamoDB [14].

We also need to display feedback of each room in a tabular structure in the front end according to the requirement. To accomplish this, we have used Google NLP API, which we expose using GCP Cloud Function and called that cloud function in the AWS Lambda for analyzing the user feedback. So as soon as the user posts feedback for a particular room, it will first come in Lambda and in Lambda, we called the cloud function with the feedback text to analyze the sentiment and then we stored the feedback along with the score and magnitude of the feedback in DynamoDB [11][12].

Moreover, we also need to analyze all feedback and show the sentiment of each feedback in the front-end. For this, we have gone through the Google Natural Language Processing API Documentation. The Natural Language API has several methods for performing analysis and annotation on your text. Out of all, the analyzeSentiment method takes given text as input and in response provides score (-1 to 1) and magnitude (0 to 1) values. Different score values give different sentiment levels like Clearly positive (above 0.8), Clearly Negative (below -0.6), Neutral (0.1), Mixed (0.0) [11].

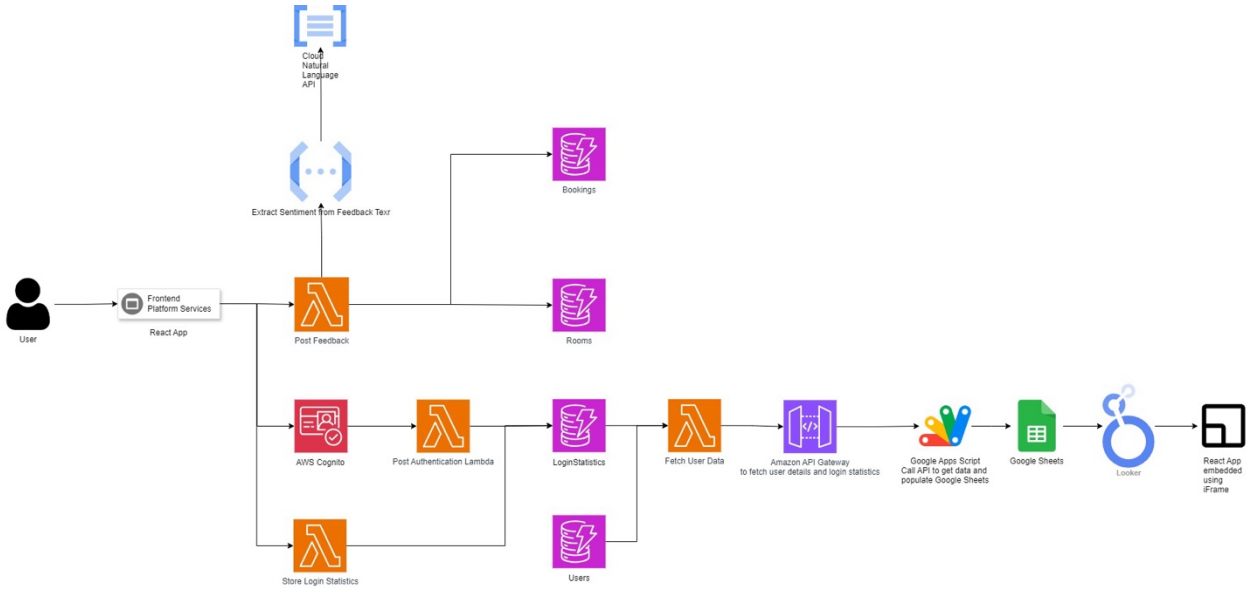

Figure 5: Architecture Diagram of Data Analytics and Visualization Module

Module 6: Web Application Building and Deployment

After going over the official documentation of Google Cloud Run and blogs on how to deploy React App on Cloud Run, we have configured CI/CD pipeline from GitLab and deploy React App on Google Cloud Run.

Google Cloud Build is used for building images of applications, and that image is deployed on Cloud Run. We first need to create a Dockerfile to create the image of the React App and then push that image to Google Cloud Build or Google Cloud Repository, and then from that, we run a container created out of that image on Cloud Run. Every step is automated using GitLab CI/CD pipeline and terraform for automatic deployment of Google Cloud Resources and React App on Google Cloud Run [18][19][20][21]. Additionally, AWS resources are created using AWS CloudFormation, ensuring a robust and repeatable infrastructure setup.

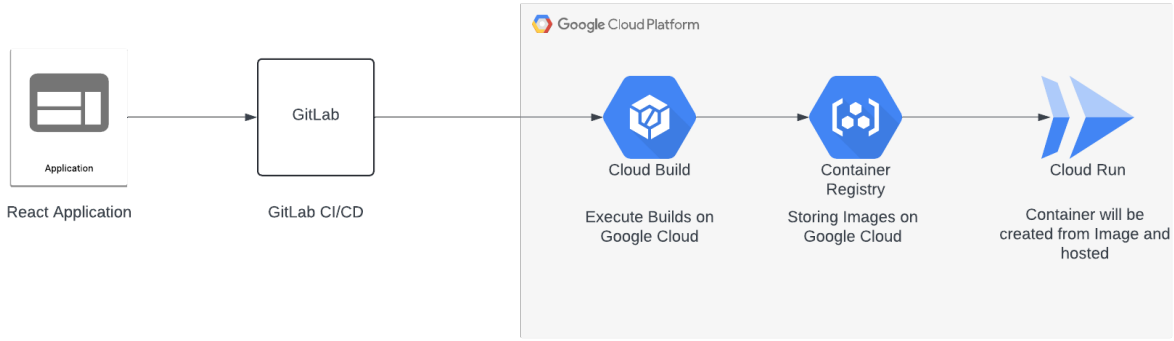

Figure 6: Architecture Diagram of Web Application Building and Deployment Module

Individual Contribution:

Christin Saji:

1. **Data Analysis and Visualization:**
    - Created ‘post-authentication-logging' Lambda function to store user login statistics, triggered post-authentication in Cognito.
    - Developed ‘storeLoginStatistics’ Lambda function to log user actions (security question completion, Caesar cipher completion, logout) via API Gateway.
    - Stored login statistics in DynamoDB table ‘LoginStatistics’.
    - Integrated Lambda function to fetch user and login statistics from DynamoDB and transfer to Google Sheets via Apps Script, utilized in Looker Studio for frontend visualization.

2. **Lambda Functions:**
    - **post-authentication-trigger:** Logs user login attempts in the DynamoDB.
    - **storeLoginStatistics:** Appends login actions to DynamoDB.
    - **fetchUserData:** Fetches user and login statistics from DynamoDB.

3. **Infrastructure as Code (IaC):**
    - Created CloudFormation stacks for all Lambda functions and DynamoDB setup.
    - Utilized Terraform to deploy Dialogflow virtual assistant.
    - Stored Lambda function zip files in S3 and referenced them in CloudFormation templates for automated setup
4. **Frontend Development:**
    - Integrated EventLoggingService to trigger login events and log via API Gateway to Lambda.
    - Enhanced various aspects of the user interface for better usability and visual appeal and resolved chatbot rendering issues.

5. **GitLab Board Management:**
    - Actively managed the progress of the Data Analysis and Visualization and Web Application and Deployment Module on the GitLab board.
    - Regularly commented on the progress of tasks and made frequent commits to ensure transparency and continuous integration.
    - Collaborated with team members to track milestones, review code, and ensure timely completion of project goals.

6. **Documentation:**
    - Updated the Data Analysis and Visualization module architecture diagram and integrated that in final architecture diagram.
    - Maintained comprehensive meeting logs for Sprint 3, capturing agendas, outcomes of discussions, and links to meeting recordings.

Jay Jagdishbhai Patel:

1. **Data Analysis and Visualization:**
    - Implemented Google Cloud function and called the GCP NLP API using cloud language library to analyze the sentiment of the text and return the score and magnitude of it.
    - And created a lambda function which will call Cloud function with feedback text as request and store the return response (score, magnitude) and feedback for a particular room in the DynamoDB Room Table.

2. **Web Application Deployment:**
    - Build CI/CD pipeline from Gitlab to Google cloud. Written script for gitlab-ci.yml, Dockerfile, Cloudbuild.yaml, and nginx.conf. Used Cloud Build for executing build process and create Docker image and push it in Artifact Registry and deploy on Cloud Run. These above steps are automated using Gitlab CI/CD pipeline.

3. **Lambda Functions:**
    a. **post-feedback:** Allows Customers to provide feedback for room after their stay ends. Also written logic to process sentiment of feedback by fetching response from cloud function and store in DynamoDB.

4. **Cloud Functions:**
    - **extract-sentiment:** take feedback text as request and used GCP NLP API to extract sentiment out of it and return the score (-1 to 1) and magnitude (0 to 1).

5. **Infrastructure as Code (IaC):**
    - **User Management and Authentication Module:** Used Cloud formation to automatically create stack for entire Cognito module (even post authentication trigger lambda)
    - **Virtual Assistance:** Used Terraform for creating script for automatic deployment of Dialog Flow virtual assistant service.
    - **Pub/Sub Module:** Used Terraform for automatic deployment of Pub/Sub service.
    - **Cloud Functions:** User Terraform for automatic deployment of all 4 Cloud functions needed in this project. Upload cloud function zip to cloud storage and reference the bucket and object name in Terraform Script for creating Cloud function.

6. **Frontend Development:**
    - Contributed to designing and developing the Data analysis and visualization frontend using React.
    - Integrated the Lambda API endpoints with the frontend for real-time data interaction.

- Integrated Looker Studio in the frontend using iFrame for displaying analytics about login statistics.
- Connected Pub/Sub module in the frontend and enabled asynchronous chat between property agent and customer. Design frontend page for customer and property agent side to view raised tickets.

7. **GitLab Board Management:**
   - Managed the project workflow using the GitLab board.
   - Organized tasks, tracked progress, and ensured timely completion of project milestones.
   - Facilitated team collaboration and done code reviews.

8. **Improve Code Quality and Commenting:**
   Contributed to Improve Code Quality and add Doc String for every Lambda and Cloud Function.

Jeet Jani

1. **Virtual Assistant Module:**
   - Updated the Dialogflow ES chatbot, defined the intents, parameters, entity types, training phrases and bot responses for various user requests like retrieving booking information, or submitting customer concerns.
   - Updated Lambda functions to correctly handle the backend logic for various user requests.
   - Updated an API endpoint for the IntentHandler Lambda function and set it as the webhook URL for connecting the chatbot with the function, and sending parameters passed by the users.

2. **Message Passing Module:**
   - Created Pub/Sub topic ManageCustomerSupportRequest to Handoff customer support request to a Cloud Function set as a subscriber, which can proceed with the flow.
   - Created AgentAssigner to assign fetch admins and assign a random agent to manage the customer concern.
   - Created a Lambda function to assist AgentAssigner to fetch agent details.
   - Wrote cloud functions to store and retrieve chats from a Firestore collection to offer users a complete chat system.

3. **Lambda Functions**
   - Created AdminFetcher function to solve the multi-cloud credentials security problem. Cloud function AgentAssigner will invoke AdminFetcher using the endpoint /fetchAdmins, and AdminFetcher will query the Users DynamoDB table. It will search for users with the role = 1 and return the retrieved results to AgentAssigner.
   - Updated HandoffCustomerSupport function to handle the Customer Support Request Intent. It hands over the concern and user metadata to the GCP

Pub/Sub topic ManageCustomerSupportRequest, and notifies the user that the query has been submitted.
- Updated IntentHandler function to receive the parameter passed by the user from the bot. IntentHandler determines what intent has been called and calls other Lambda functions to execute the necessary logic. It also returns the output given by these Lambda functions back to the chatbot interface.
- Updated FetchBookingDetails function to handle the Booking Info Intent. It queries the DynamoDB table, Users, and returns the room details if found in the table. Else returns the message that no such booking has been made.

4. **Cloud Functions:**
   - Wrote AgentAssigner which is triggered whenever a message is published in ManageCustomerSupportRequest. Then it calls AdminFetcher to fetch the agents' list. It will assign a random agent to the concern and store the metadata along with the assigned agent details in a Firestore collection - Issues. It will also create an empty array of chats which will be updated if any chat is recorded between the user and the assigned agent.
   - Wrote ChatLogger that logs the messages sent by the user and the agent in the Issues collection. The chat array gets appended with this new message and metadata in the form of JSON documents. Each document represents a separate message.
   - Wrote ChatRetriever that retrieves all the chats between the user and the agent for every concern requested.

5. **GitLab Board Management:**
   - Used GitLab board to manage project tasks.
   - Kept track of tasks, progress, and made sure we met deadlines.
   - Helped the team work together and reviewed code.

6. **Documentation:**
   - Designed architecture diagrams for the Virtual Assistant module and the entire project, including plans for future service integrations.
   - Assisted teammates in formatting the final report in a well-structured manner.

Kuldeep Gajera

1. **User Management & Authentication Module with AWS Cognito:**
   - Implemented user and property agent sign-up features on the frontend. This involved integrating the signup forms with AWS Lambda functions and DynamoDB to handle data storage and retrieval securely and efficiently.
   - Provided support to team members for smooth integration of user management features.
2. **Lambda Functions:**
   - **StoreUserDetails**: The function for storing user registration details in DynamoDB, ensuring data consistency and security.

- **GetShiftKey**: The function to retrieve a user's shift key from DynamoDB, facilitating secure user verification processes.
- **GetSecurityAnswer**: The function to fetch the security answer to a user's security question from DynamoDB, crucial for user authentication.
- **DeleteRoomWithImage**: The function to delete a room and its associated image from Amazon S3, maintaining the integrity and freshness of room data.
- **UpdateRoomDetailsWithImage**: The function to update room details and optionally an image in S3, allowing dynamic updates to room listings.
- UserNotificationLogic : The function to send notifications to end users upon successful registration, successful login, successful booking and booking failer.

3. **Notification Module:**
   - Designed and implementation of a scalable notification system using AWS SNS, SQS, and Lambda, ensuring efficient communication across user activities such as registrations, logins, and bookings. I configured SNS topics linked to SQS queues to handle message surges and implemented Lambda functions for processing these messages.
   - Contribution extended to integrating these services to trigger notifications for key user actions—successfully registering, logging in, confirming bookings, and handling booking failures. This integration was critical in enabling real-time, reliable notifications that enhanced the user experience by providing timely feedback on their interactions with our application.

4. **Documentation**
   - Added references and wrote captions for documentation images.
   - Helped format and manage the project report for clarity and professionalism.

GitLab Code Repository:

https://git.cs.dal.ca/patel45/serverless_group_36/-/tree/main?ref_type=heads

System architecture:

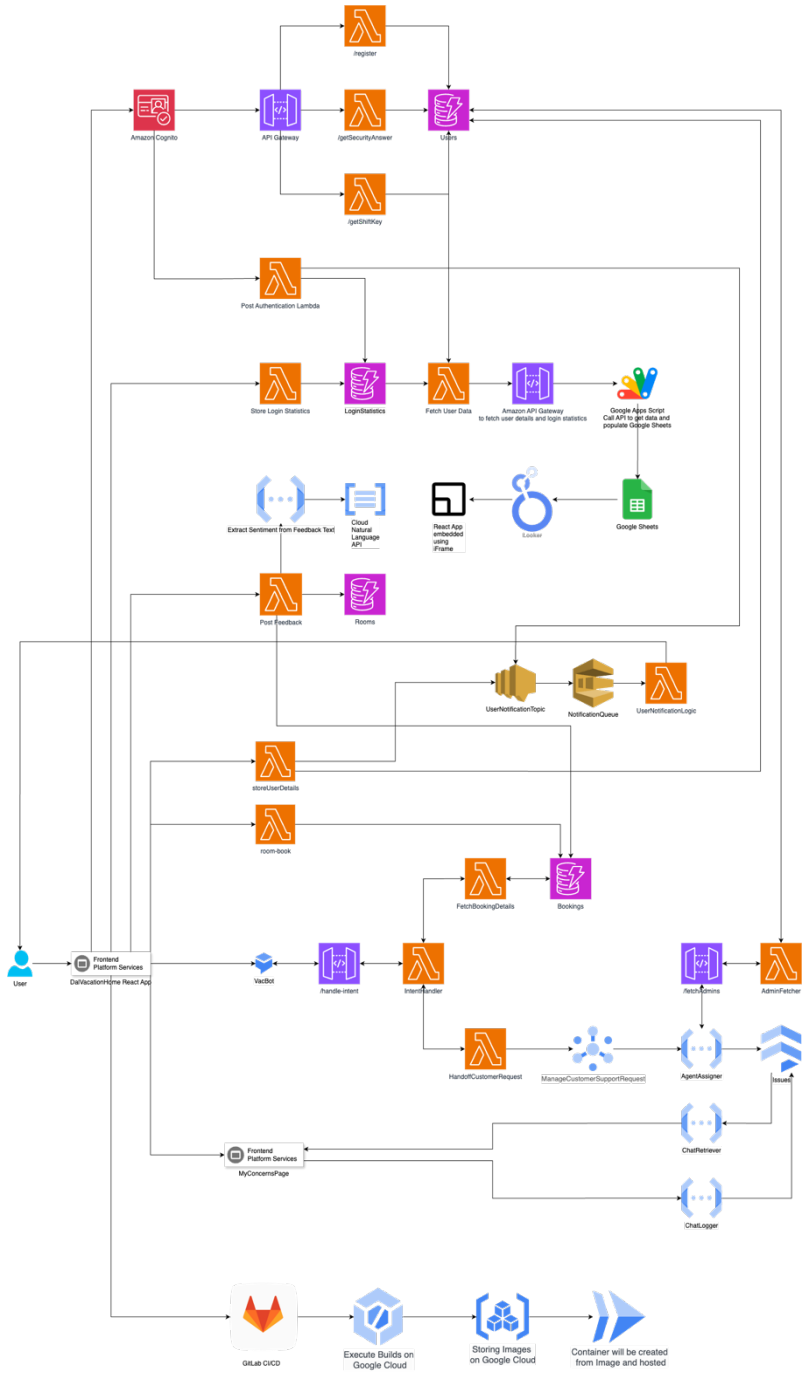

Figure 7: System Architecture Diagram of Application

Pseudocode/Algorithm for Important Modules' logic:

Module 2: Virtual Assistant

HandoffCustomerSupport

Pseudocode:

```
FUNCTION lambda_handler(event):
    TRY:
        IF event is a string:
            Parse event to JSON and assign to body
        ELSE:
            Assign event to body

        Extract issue from body.Issue
        Extract bookingReferenceCode from body.BookingReferenceCode

        Log triggered by IntentHandler
        Log issue received
        Log booking reference code received

Store issue and bookingReferenceCode in DynamoDB table 'CustomerConcerns'

        RETURN success message 'Your query has been submitted. A representative will contact you shortly.'
    CATCH error:
        Log error

        RETURN failure message 'There was an error processing your request. Please try again later.'
END FUNCTION lambda_handler
```

Algorithm

1. Initialize Variables
2. Parse Event
   a. If event is a string:
      i. Parse event to JSON and assign to body.
   b. Else:
      i. Assign event to body.
3. Extract Parameters
   a. Extract issue from body.Issue.
   b. Extract bookingReferenceCode from body.BookingReferenceCode.
4. Log Details
   a. Log 'triggered by IntentHandler'.
   b. Log issue received.
   c. Log booking reference code received.

5. Store Customer Concerns
    a. Store issue and bookingReferenceCode in DynamoDB table 'CustomerConcerns'.
6. Return Response
    a. Return success message 'Your query has been submitted. A representative will contact you shortly.'.
7. Handle Errors
    a. In the catch block:
        i. Log error.
        ii. Return failure message 'There was an error processing your request. Please try again later.'.

Module 3: Message Passing

AdminFetcher

Pseudocode:

```
FUNCTION handler(event)
    INITIALIZE AWS SDK
    INITIALIZE DynamoDB DocumentClient

    SET params
        TableName = 'Users'
        FilterExpression = '#role = :role'
        ExpressionAttributeNames = {'#role': 'role'}
        ExpressionAttributeValues = {':role': '1'}

    TRY
        LOG 'Fetching admins with params:' and params
        FETCH data FROM dynamoDb.scan(params)
        EXTRACT admins FROM data.Items

        LOG 'Fetched Admins:' and admins

        RETURN
            statusCode = 200
            body = JSON.stringify(admins)
    CATCH error
        LOG 'Error fetching admins:' and error
        RETURN
            statusCode = 500
            body = JSON.stringify({ error: 'Could not fetch admins' })
END FUNCTION
```

Algorithm:

1. Initialize AWS SDK
2. Initialize DynamoDB DocumentClient.
3. Set Parameters
4. Define params with the following properties:
    a. TableName set to 'Users'.
    b. FilterExpression set to '#role =
5. '.
    a. ExpressionAttributeNames with '#role' mapped to 'role'.
    b. ExpressionAttributeValues with '
6. ' mapped to '1'.
7. Log Fetching Details
8. Log the message 'Fetching admins with params:' followed by the params.
9. Fetch Data
10. Use dynamoDb.scan(params).promise() to fetch data from DynamoDB.
11. Extract and Log Admins
12. Extract admins from data.Items.
13. Log the message 'Fetched Admins:' followed by the admins.
14. Return Success Response
15. Return an object with:
    a. statusCode set to 200.
    b. body set to JSON stringified admins.
16. Handle Errors
17. In the catch block:
    a. Log the message 'Error fetching admins:' followed by the error.
    b. Return an object with:
        i. statusCode set to 500.
        ii. body set to JSON stringified { error: 'Could not fetch admins' }.

AgentAssigner

Pseudocode:

```
IMPORT necessary libraries and modules

INITIALIZE Firestore client

FUNCTION fetch_admins
   SET url to API endpoint
   TRY
      SEND GET request to url
      CHECK response for HTTP errors
```

```
        PARSE response JSON to get admins
        RETURN admins
    CATCH RequestException as e
        LOG error message
        RETURN empty list

FUNCTION hello_pubsub(cloud_event)
    DECODE pubsub_data from base64
    TRY
        PARSE pubsub_data as JSON to get message_data

        EXTRACT bookingReferenceCode, issue, and email from message_data

        LOG bookingReferenceCode, issue, and email

        CALL fetch_admins to get list of admins
        IF admins list is not empty
            SELECT a random admin from the list
            EXTRACT name and email of the selected admin
            LOG selected admin's name and email
        ELSE
            SET assigned_admin_name to "No Admin Available"
            SET assigned_admin_email to "No Admin Available"
            LOG "No Admin Available"

        INITIALIZE empty chats list
        GET current date and time as dateRaised
        SET isActive to True

        ADD document to Firestore 'Issues' collection with extracted and generated
data
        GET document ID of the added document
        LOG success message with document ID
    CATCH JSONDecodeError as e
        LOG error message
    CATCH Exception as e
        LOG error message
```

Algorithm:

1. Import Libraries
2. Import necessary libraries: base64, json, random, requests, functions_framework, google.cloud.firestore, and datetime.
3. Initialize Firestore Client
4. Initialize a Firestore client.
5. Define fetch_admins Function
6. Set API Endpoint

    a. Define url for the API endpoint.
7. Try to Fetch Admins
    a. Send a GET request to the API endpoint.
    b. Check for HTTP errors.
    c. Parse the response JSON to get admins.
    d. Return the list of admins.
8. Handle Errors
    a. Catch RequestException.
    b. Log the error message.
    c. Return an empty list.
9. Define hello_pubsub Function
10. Decode Pub/Sub Data
    a. Decode pubsub_data from base64.
11. Try to Process Data
    a. Parse pubsub_data as JSON to get message_data.
    b. Extract bookingReferenceCode, issue, and email from message_data.
    c. Log bookingReferenceCode, issue, and email.
    d. Fetch Admins
        i. Call fetch_admins to get the list of admins.
        ii. If admins is not empty:
            1. Select a random admin from the list.
            2. Extract the name and email of the selected admin.
            3. Log the selected admin's name and email.
        iii. Else:
            1. Set assigned_admin_name to "No Admin Available".
            2. Set assigned_admin_email to "No Admin Available".
            3. Log "No Admin Available".
    e. Initialize Variables
        i. Initialize an empty chats list.
        ii. Get the current date and time as dateRaised.
        iii. Set isActive to True.
    f. Store Issue in Firestore
        i. Add a document to the Firestore 'Issues' collection with the extracted and generated data.
        ii. Get the document ID of the added document.
        iii. Log a success message with the document ID.
12. Handle Errors
    a. Catch JSONDecodeError.
    b. Log the error message.
    c. Catch other exceptions.
    d. Log the error message.

ChatLogger

Pseudocode:

```
IMPORT necessary libraries and modules

INITIALIZE Firestore client

DEFINE hello_http(request)
   ALLOW cross-origin requests

   PARSE request JSON body

   IF request JSON is empty
      RETURN error response "Invalid request, JSON body is required" with
status 400

   TRY
      EXTRACT concernId and chats from request JSON

      IF concernId is empty
         RETURN error response "Invalid request, concernId is required" with
status 400

      IF chats is empty or not a list
         RETURN error response "Invalid request, 'chats' should be a list and
cannot be empty" with status 400

      FOR each chat in chats
         IF chat does not have 'from', 'message', and 'timestamp'
            RETURN error response "Invalid request, each chat must have 'from',
'message', and 'timestamp'" with status 400

      FETCH existing issue document from Firestore using concernId
      IF document does not exist
         RETURN error response "Issue not found" with status 404

      CONVERT document to dictionary

      APPEND new chats to existing chats in the issue document

      UPDATE issue document in Firestore with the new chats array

      RETURN success response "Chats updated successfully" with status 200

   CATCH Exception as e
      RETURN error response with exception message and status 500
```

Algorithm:

1. Import Libraries
2. Import necessary libraries: functions_framework, google.cloud.firestore, flask.jsonify, flask.request, and flask_cors.cross_origin.
3. Initialize Firestore Client
4. Initialize a Firestore client.
5. Define hello_http Function
6. Allow Cross-Origin Requests
    a. Decorate the function with @cross_origin() to allow cross-origin requests.
7. Parse Request JSON
    a. Parse the JSON body from the request.
    b. If the JSON body is empty:
        i. Return an error response with message "Invalid request, JSON body is required" and status 400.
8. Try to Process Data
    a. Extract concernId and chats from the request JSON.
    b. If concernId is empty:
        i. Return an error response with message "Invalid request, concernId is required" and status 400.
    c. If chats is empty or not a list:
        i. Return an error response with message "Invalid request, 'chats' should be a list and cannot be empty" and status 400.
    d. Validate Chats
        i. For each chat in chats:
            1. If the chat does not have 'from', 'message', and 'timestamp':
                a. Return an error response with message "Invalid request, each chat must have 'from', 'message', and 'timestamp'" and status 400.
    e. Fetch Existing Issue Document
        i. Fetch the existing issue document from Firestore using concernId.
        ii. If the document does not exist:
            1. Return an error response with message "Issue not found" and status 404.
    f. Append and Update Chats
        i. Convert the document to a dictionary.
        ii. Append new chats to the existing chats in the issue document.
        iii. Update the issue document in Firestore with the new chats array.
    g. Return Success Response
        i. Return a success response with message "Chats updated successfully" and status 200.
9. Handle Errors
    a. Catch exceptions.
    b. Return an error response with the exception message and status 500.

ChatRetriever

Pseudocode:

```
IMPORT necessary libraries and modules

INITIALIZE Firestore client

DEFINE ChatRetriever(request)
   ALLOW cross-origin requests

   PARSE request JSON body and request arguments

   INITIALIZE customer_email and agent_email to None

   IF request JSON body exists
      SET customer_email from request JSON
      SET agent_email from request JSON
   ELSE IF request arguments exist
      SET customer_email from request arguments
      SET agent_email from request arguments

   IF customer_email is None AND agent_email is None
      RETURN error response "customerEmail or agentEmail is required!" with
status 400

   TRY
      INITIALIZE reference to Firestore 'Issues' collection

      IF customer_email exists
         CREATE query to filter issues by customer_email
      ELSE
         CREATE query to filter issues by agent_email

      EXECUTE query and get documents

      INITIALIZE empty issues list

      FOR each document in query result
         CONVERT document to dictionary
         ADD document ID to dictionary as concernId
         APPEND dictionary to issues list

      RETURN success response with issues list and status 200

   CATCH Exception as e
      RETURN error response with exception message and status 500
```

Algorithm:

1. Import Libraries
2. Import necessary libraries: functions_framework, google.cloud.firestore, flask.jsonify, and flask_cors.cross_origin.
3. Initialize Firestore Client
4. Initialize a Firestore client.
5. Define ChatRetriever Function
6. Allow Cross-Origin Requests
    a. Decorate the function with @cross_origin() to allow cross-origin requests.
7. Parse Request Data
    a. Parse the JSON body and arguments from the request.
    b. Initialize customer_email and agent_email to None.
    c. If the request JSON body exists:
        i. Set customer_email from the request JSON.
        ii. Set agent_email from the request JSON.
    d. Else if the request arguments exist:
        i. Set customer_email from the request arguments.
        ii. Set agent_email from the request arguments.
8. Validate Emails
    a. If both customer_email and agent_email are None:
        i. Return an error response with message "customerEmail or agentEmail is required!" and status 400.
9. Try to Retrieve Chats
    a. Initialize a reference to the Firestore 'Issues' collection.
    b. If customer_email exists:
        i. Create a query to filter issues by customer_email.
    c. Else:
        i. Create a query to filter issues by agent_email.
    d. Execute the query and get the documents.
    e. Initialize an empty issues list.
    f. For each document in the query result:
        i. Convert the document to a dictionary.
        ii. Add the document ID to the dictionary as concernId.
        iii. Append the dictionary to the issues list.
    g. Return a success response with the issues list and status 200.
10. Handle Errors
    a. Catch exceptions.
    b. Return an error response with the exception message and status 500.

Module 5: Data Analysis and Visualization

Lambda Function: Post Feedback

Pseudocode:

```
FUNCTION lambda_handler(event, context):

    DEFINE apiUrl as 'https://us-central1-dalvacationhome-
        429314.cloudfunctions.net/extract-sentiment'

    INITIALIZE body to undefined
    INITIALIZE statusCode to 200
    INITIALIZE headers with "Content-Type" set to "application/json"

    TRY:
        IF event.body is a string:
            PARSE event.body to JSON and assign to requestJSON
        ELSE:
            ASSIGN event.body directly to requestJSON

        DEFINE payload with text set to requestJSON.feedbackText

        DEFINE response as result of POST request to apiUrl with payload

        IF response is not OK:
            THROW Error with message containing response status

        PARSE response JSON to data

        DEFINE feedback with properties from requestJSON and data (including
current date)

        DEFINE result as result of UpdateCommand to roomsTable to append
feedback

        DEFINE resultUpdateFeedbackStatus as result of UpdateCommand to
bookingsTable to set feedbackRecived to true
        SET body to success message with roomid from requestJSON
    CATCH error:
        SET statusCode to 400
        SET body to error message
    FINALLY:
        CONVERT body to JSON string
    RETURN response with statusCode, body, and headers

END FUNCTION lambda_handler
```

Algorithm:

1. Initialize Variables
    a. Define apiUrl for the sentiment analysis service.
    b. Initialize body to undefined.
    c. Set statusCode to 200.
    d. Define response headers with content type as JSON.
2. Parse Event Body
    a. If event.body is a string, parse it to JSON and assign it to requestJSON.
    b. Otherwise, assign event.body directly to requestJSON.
3. Prepare Payload for Sentiment Analysis
    a. Create a payload object with text set to requestJSON.feedbackText.
4. Call Sentiment Analysis API
    a. Send a POST request to apiUrl with the payload.
    b. If the response is not OK, throw an error with the response status.
5. Parse API Response
    a. Parse the JSON response from the API to data.
6. Create Feedback Object
    a. Create a feedback object with properties from requestJSON and data, including the current date.
7. Update DynamoDB Tables
    a. Update Rooms Table:
        i. Append the feedback object to the feedback list in the roomsTable.
    b. Update Bookings Table:
        i. Mark the feedback as received in the bookingsTable.
8. Set Response Body
    a. Set the response body to a success message using the roomid from requestJSON.
9. Handle Errors
    a. In the catch block:
        i. Set statusCode to 400.
        ii. Set the response body to the error message.
10. Convert Response Body
    a. Convert the response body to a JSON string.
11. Return Response
    a. Return a JSON response with statusCode, body, and headers.

Cloud Function: Extract Sentiment

Pseudocode:

```
FUNCTION helloHttp(req, res):
  EXTRACT text from req.body
  INITIALIZE client as new instance of language.LanguageServiceClient
  DEFINE document with content set to text and type set to 'PLAIN_TEXT'

  TRY:
    CALL analyzeSentiment method on client with document
    ASSIGN result to the first element of the response array
    EXTRACT sentiment from result.documentSentiment
    DEFINE response object with text, sentimentScore, and
sentimentMagnitude from sentiment
    RETURN response with 200 status code and JSON body
  CATCH error:
    LOG error
    RETURN error message with 500 status code
END FUNCTION helloHttp
```

Algorithm:

1. Parse Request Body
    a. Extract text from req.body.
2. Initialize Language Service Client
    a. Initialize a new instance of language.LanguageServiceClient.
3. Prepare Document for Sentiment Analysis
    a. Define a document object with:
        i. content set to the extracted text.
        ii. type set to 'PLAIN_TEXT'.
4. Analyze Sentiment
    a. Use a try-catch block to handle potential errors:
        i. Try Block:
            1. Call the analyzeSentiment method on the client with the document.
            2. Assign result to the first element of the response array.
            3. Extract sentiment from result.documentSentiment.
            4. Define a response object with:
                a. text set to the original text.
                b. sentimentScore set to sentiment.score.
                c. sentimentMagnitude set to sentiment.magnitude.
            5. Return the response with a 200 status code and JSON body.
        ii. Catch Block:
            1. Log the error.
            2. Return an error message with a 500 status code.
5. Return Response

a. Based on the outcome of the sentiment analysis, return the appropriate HTTP status code and message:
    i. 200 with the sentiment analysis result if successful.
    ii. 500 with an error message if there is an issue with the sentiment analysis.

Lambda Function: post-authentication-trigger

Pseudocode:

```
FUNCTION handler(event, context):
  INITIALIZE dynamoDBClient, dynamoDocument, snsClient

  EXTRACT email, name, roleCode from event.request.userAttributes

  DETERMINE role based on roleCode
  GET current date and time

  DEFINE params for updating DynamoDB table LoginStatistics

  TRY:
    SEND UpdateCommand to dynamoDocument with params
    LOG success message

    CREATE notification message
    DEFINE snsParams with TopicArn and message

    SEND PublishCommand to snsClient with snsParams
    LOG success message
  CATCH error:
    LOG error message

  RETURN event
END FUNCTION
```

Algorithm:

1. Initialize Variables:
    a. Initialize dynamoDBClient, dynamoDocument, and snsClient.
2. Extract User Information:
    a. Extract email, name, and roleCode from event.request.userAttributes.
3. Determine User Role:
    a. Determine role based on roleCode.
4. Get Current Date and Time:
    a. Get the current date and time.
5. Define DynamoDB Update Parameters:
    a. Define params for updating the LoginStatistics table in DynamoDB.
6. Try Block:

    a. Update DynamoDB:
        i. Send UpdateCommand to dynamoDocument with params.
        ii. Log success message.
    b. Send Notification:
        i. Create notification message.
        ii. Define snsParams with TopicArn and message.
        iii. Send PublishCommand to snsClient with snsParams.
        iv. Log success message.
7. Catch Block:
    a. Log error message if an exception occurs.
8. Return Event:
    a. Return the event object.

Lambda Function: storeLoginStatistics

Pseudocode:

```
FUNCTION handler(event, context):
  INITIALIZE client, dynamo
  INITIALIZE body

  IF event.body is a string:
    PARSE event.body to JSON and assign to body
  ELSE:
    ASSIGN event.body to body

  EXTRACT email, date, time, action from body

  DEFINE params for updating DynamoDB table LoginStatistics

  TRY:
    SEND UpdateCommand to dynamo with params
    RETURN success response with result attributes
  CATCH error:
    LOG error message
    RETURN error response
END FUNCTION
```

Algorithm:

1. Initialize Variables:
    a. Initialize client and dynamo.
2. Parse Event Body:
    a. If event.body is a string, parse it to JSON and assign to body.
    b. Otherwise, assign event.body to body.
3. Extract Parameters:
    Extract email, date, time, and action from body.

4. Define DynamoDB Update Parameters:
    a. Define params for updating the LoginStatistics table in DynamoDB.
5. Try Block:
    a. Update DynamoDB:
        i. Send UpdateCommand to dynamo with params.
        ii. Return success response with result attributes.
6. Catch Block:
    a. Log error message if an exception occurs.
    b. Return error response.

Lambda Function: fetchUserData

Pseudocode:

```
FUNCTION handler(event, context):
  INITIALIZE client, dynamo

  TRY:
    DEFINE usersParams for scanning Users table
    SCAN Users table and assign result to users

    DEFINE loginStatsParams for scanning LoginStatistics table
    SCAN LoginStatistics table and assign result to loginStats

    RETURN success response with users and loginStats
  CATCH error:
    LOG error message
    RETURN error response
END FUNCTION
```

Algorithm:

1. Initialize Variables:
    a. Initialize client and dynamo.
2. Try Block:
    a. Fetch Users:
        i. Define usersParams for scanning the Users table.
        ii. Scan Users table and assign the result to users.
    b. Fetch Login Statistics:
        i. Define loginStatsParams for scanning the LoginStatistics table.
        ii. Scan LoginStatistics table and assign the result to loginStats.
    c. Return Success Response:
        i. Return success response with users and loginStats.
3. Catch Block:
    a. Log error message if an exception occurs.
    b. Return error response.

Test Cases:

Lambda Test Post Feedback Event:

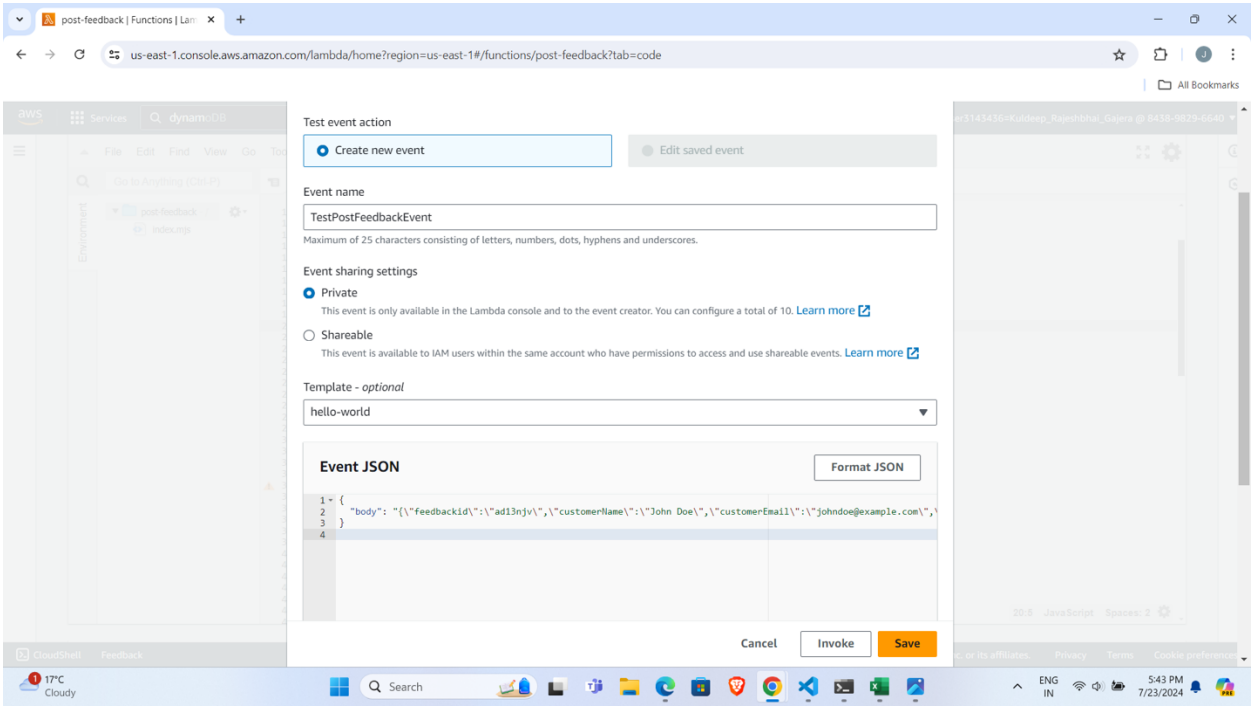

Figure 8: Setup and execution of the Post Feedback test event in AWS Lambda.

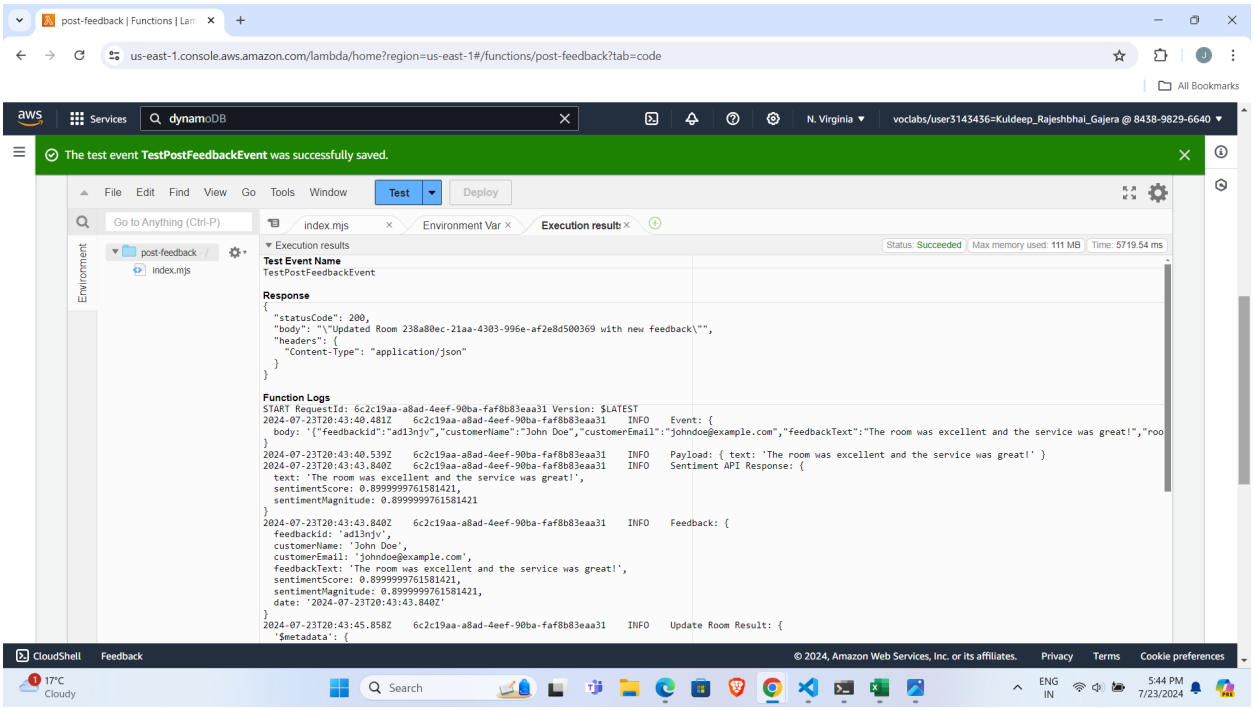

Figure 9: CloudWatch log showing the results of the Post Feedback test event execution.

Cloud Function Extract Sentiment Test Event:

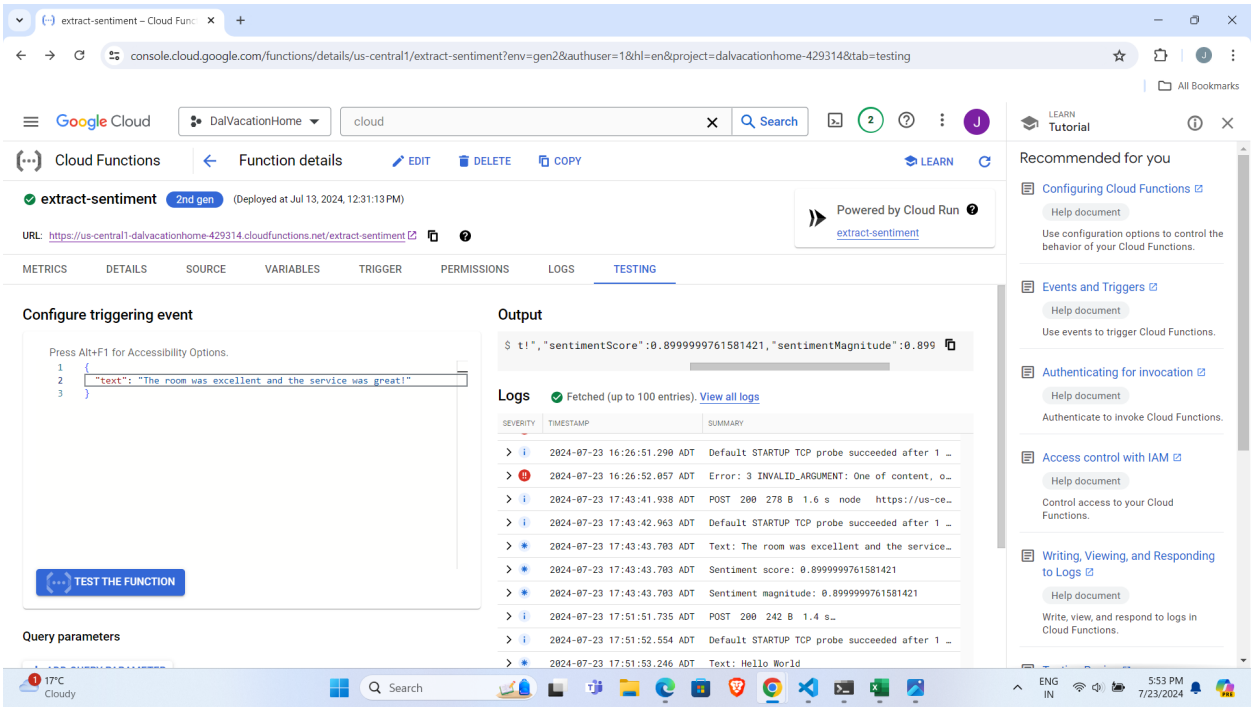

Figure 10: Configuring and executing Extract Sentiment Cloud Function test event in GCP Console.

Lambda Test Post Authentication Logging Event:

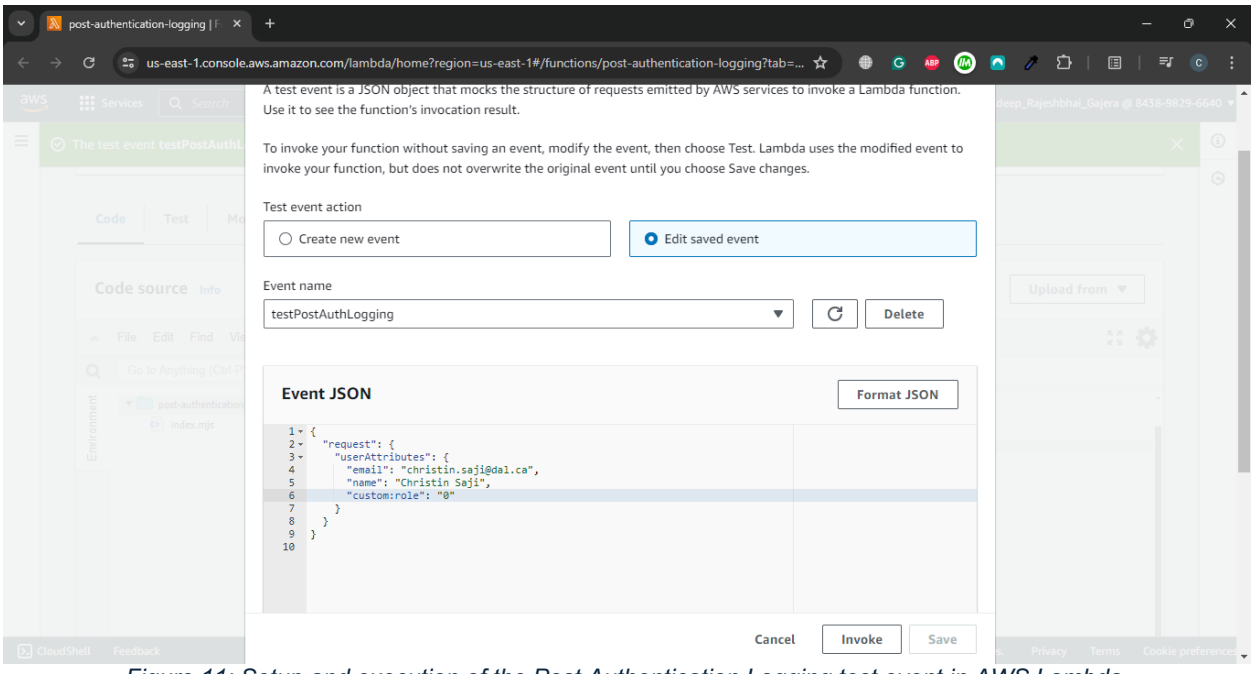

Figure 11: Setup and execution of the Post Authentication Logging test event in AWS Lambda.

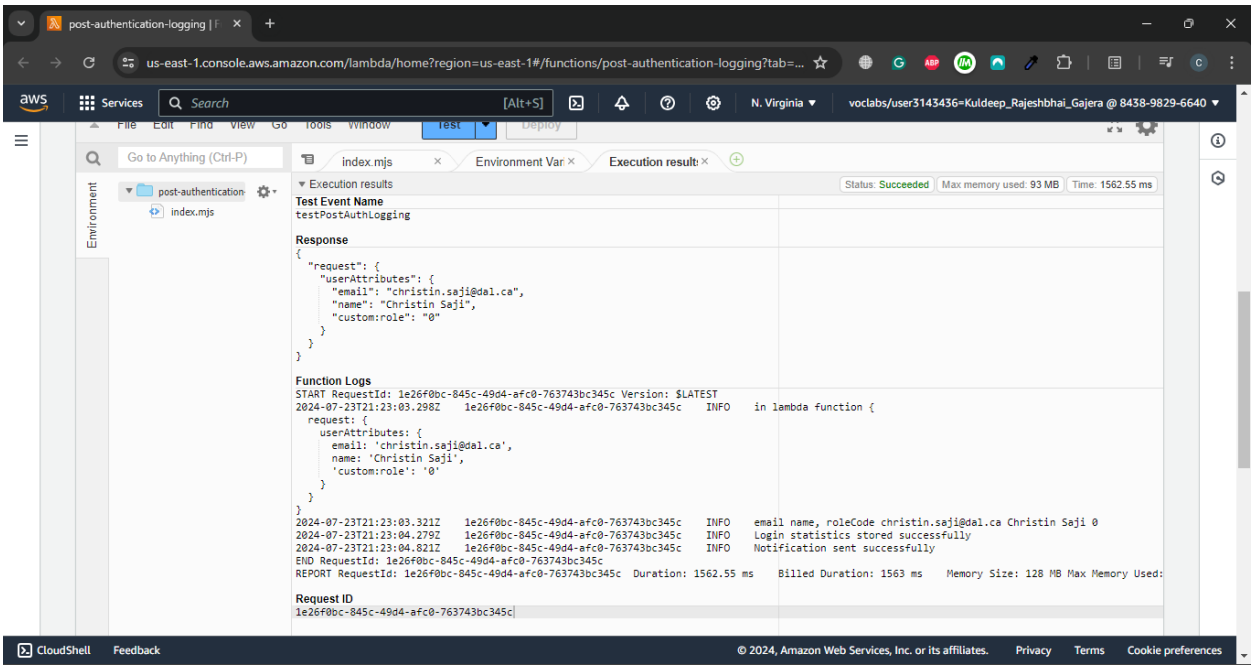

Figure 12: CloudWatch log showing the results of the Post Authentication Logging test event execution.

Lambda Test Store Login Statistics Event:

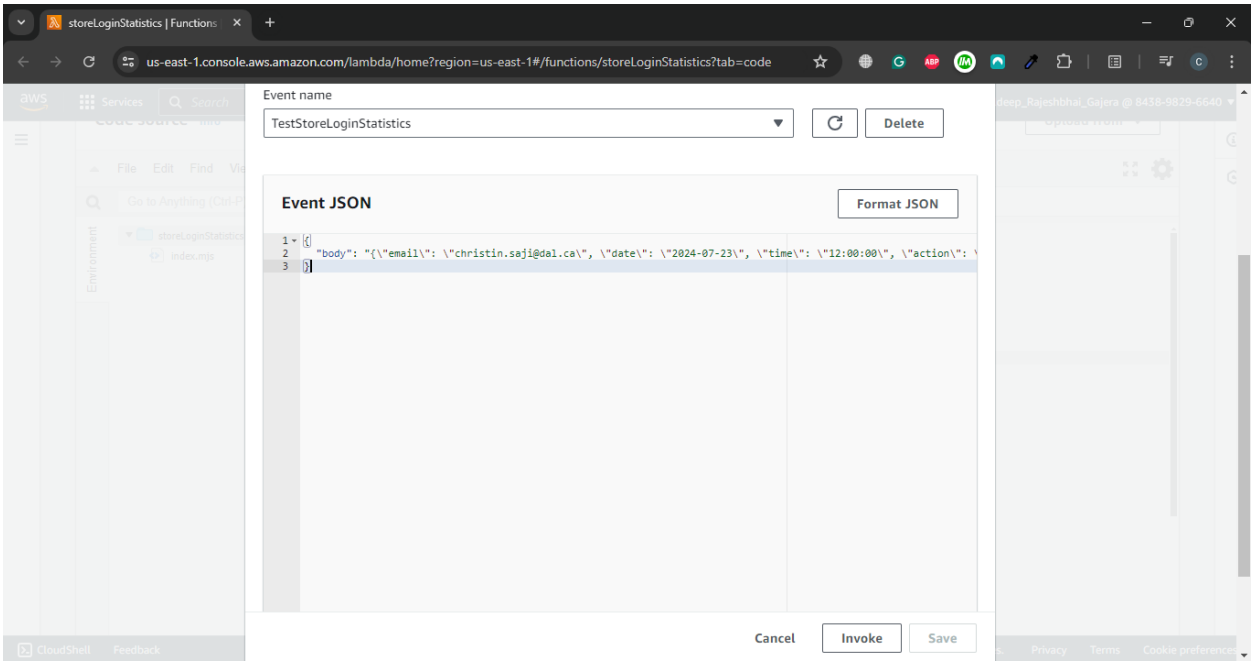

Figure 13: Setup and execution of the Store Login Statistics test event in AWS Lambda.

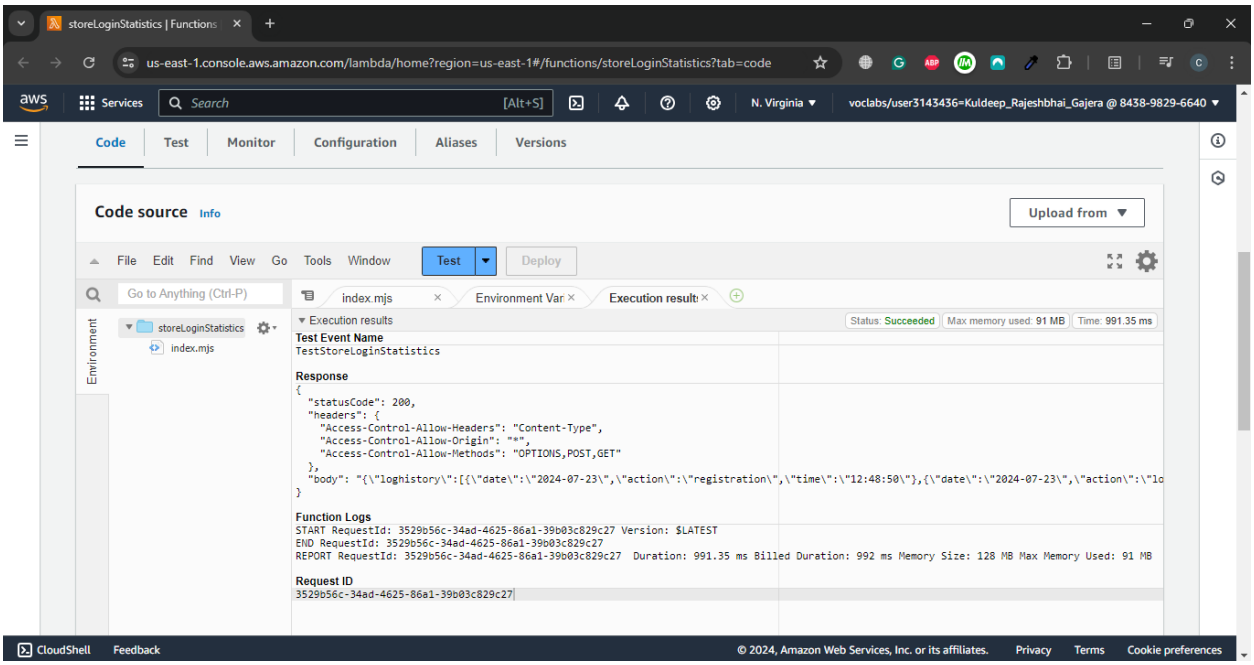

Figure 14: CloudWatch log showing the results of the Store Login Statistics test event execution.

Lambda Test Fetch User Data Event:

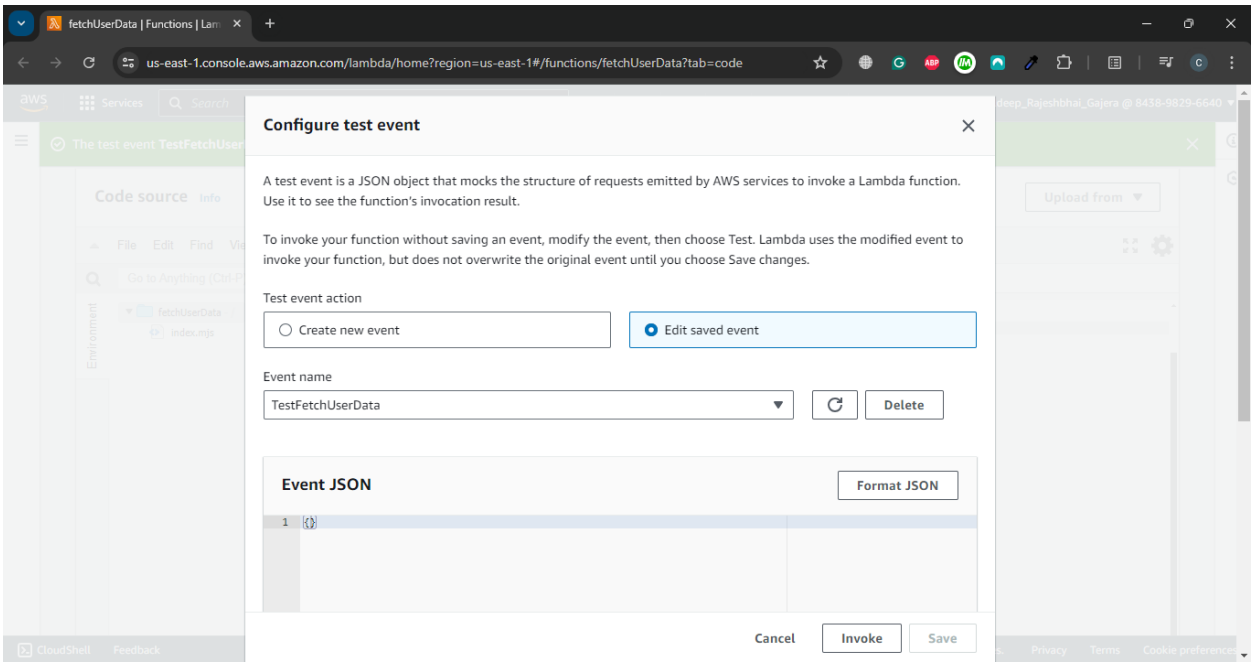

Figure 15: Setup and execution of the Fetch User Data test event in AWS Lambda.

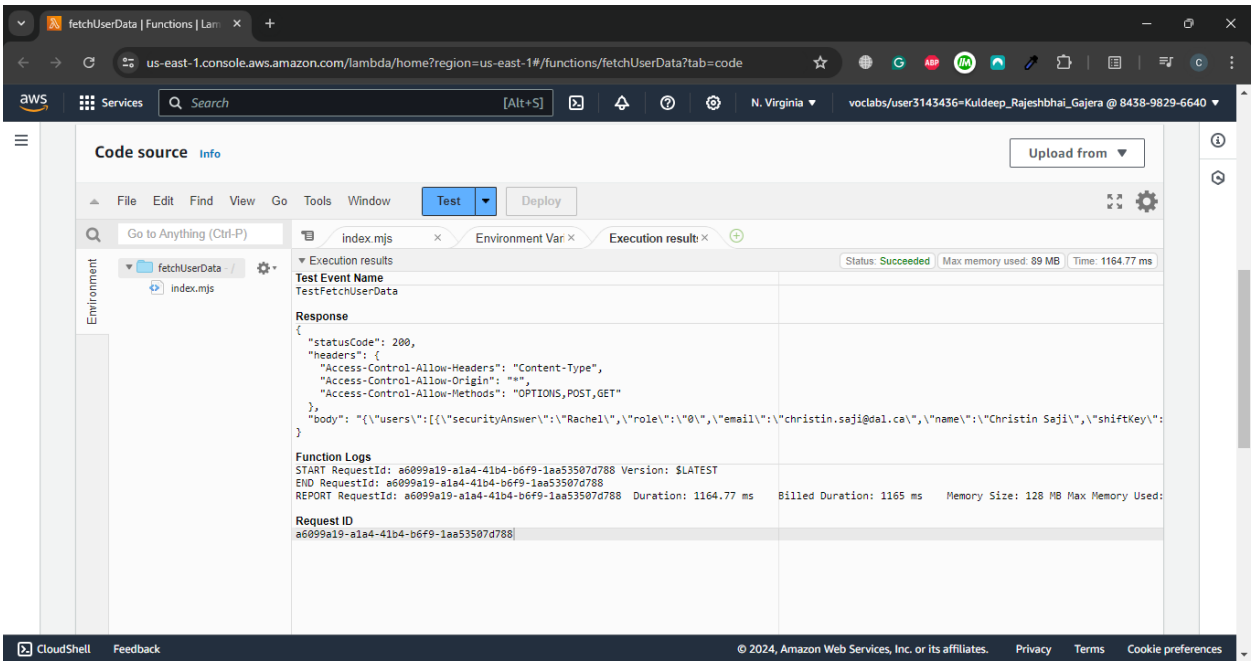

Figure 16: CloudWatch log showing the results of the Fetch User Data test event execution.

HandoffCustomerSupport

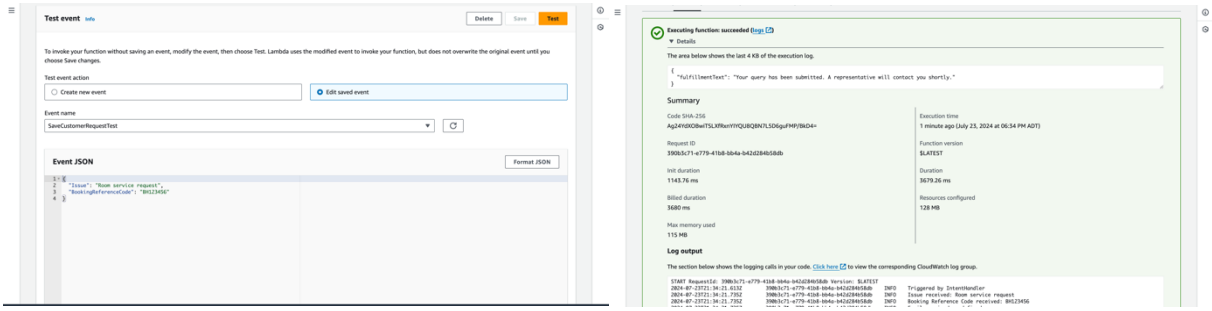

Figure 17: Executing the HandoffCustomerSupport test with corresponding CloudWatch log outcomes.

AgentAssigner:

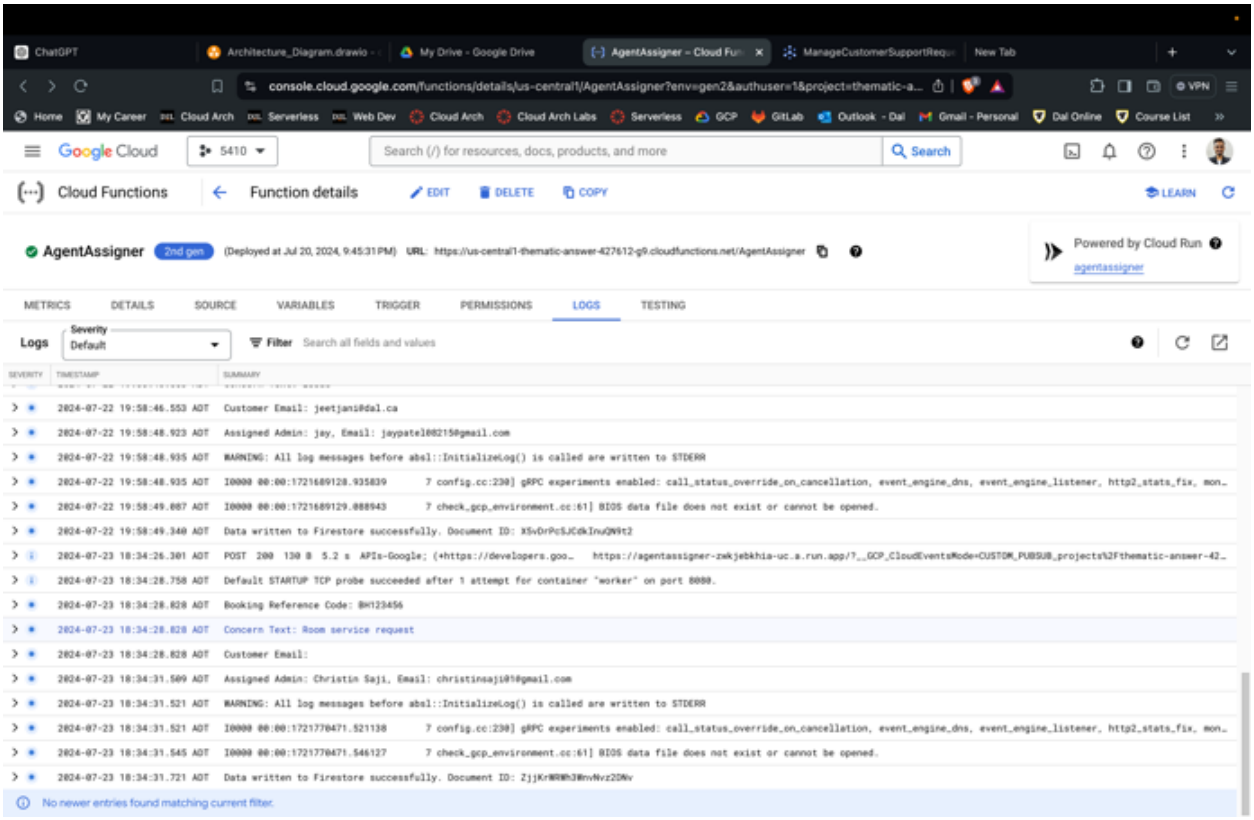

Figure 18: Setting up and executing the IntentHandler test, with results shown in the CloudWatch logs.

AdminFetcher:

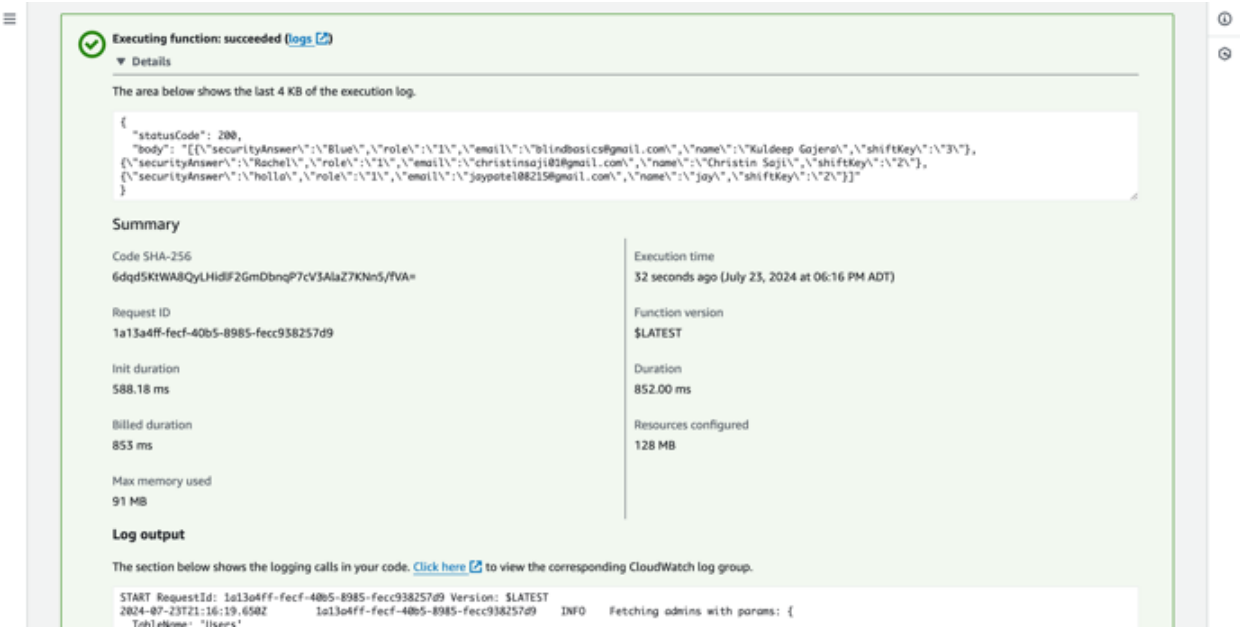

Figure 19: Running the FetchBookingDetails test and displaying execution logs in CloudWatch.

ChatLogger

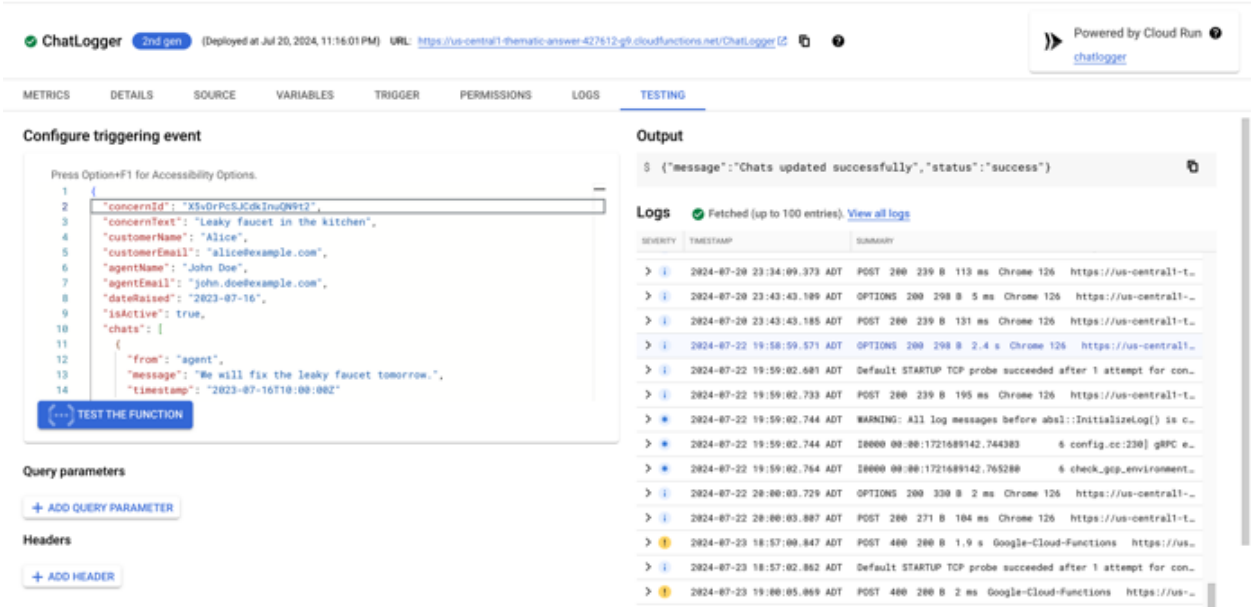

Figure 20: Running the FetchBookingDetails test and displaying execution logs in CloudWatch.

ChatRetriever

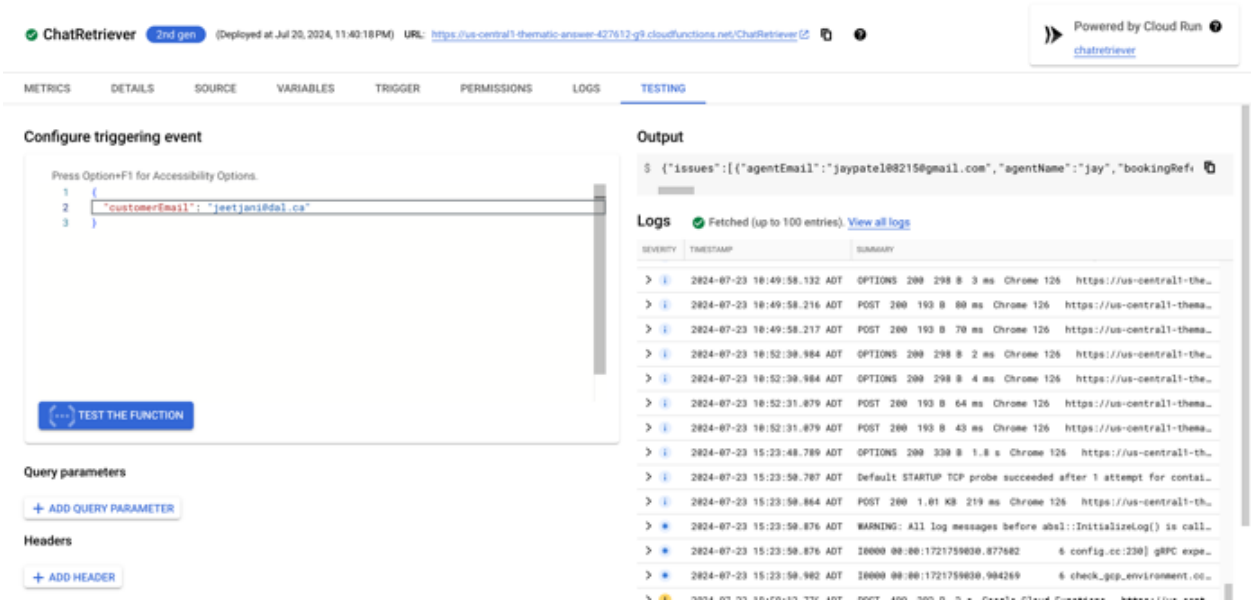

Figure 21: Running the FetchBookingDetails test and displaying execution logs in CloudWatch.

Notification testing using SNS publish topic

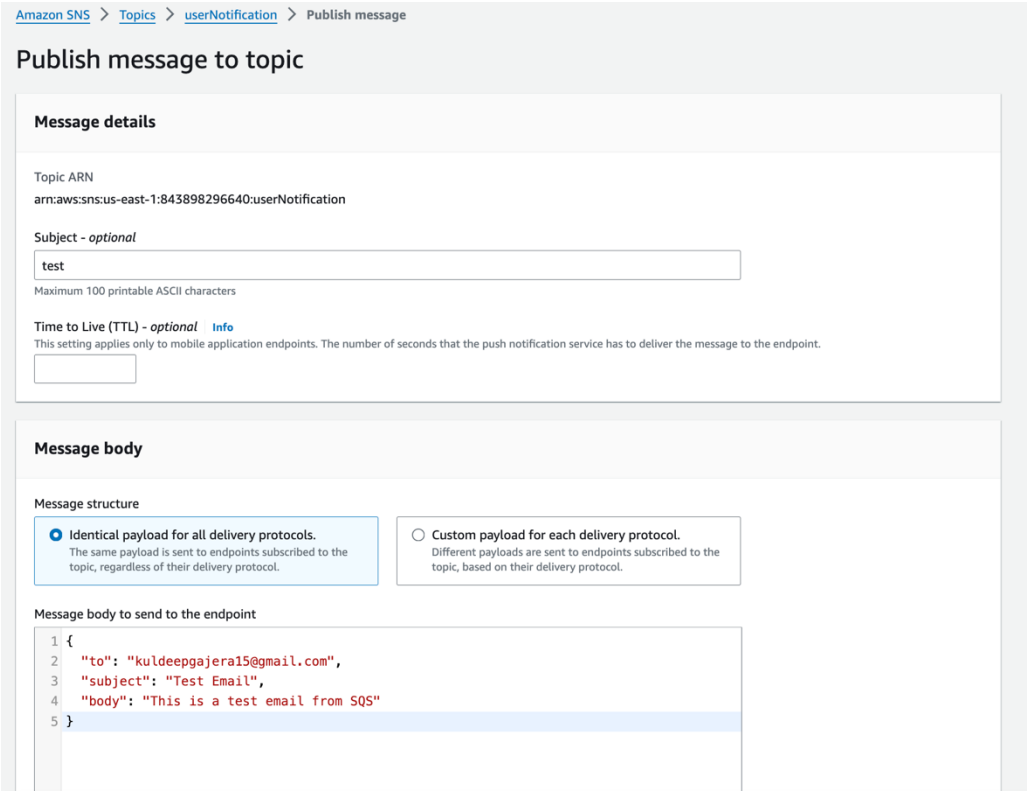

Figure 22: Running the FetchBookingDetails test and displaying execution logs in CloudWatch.

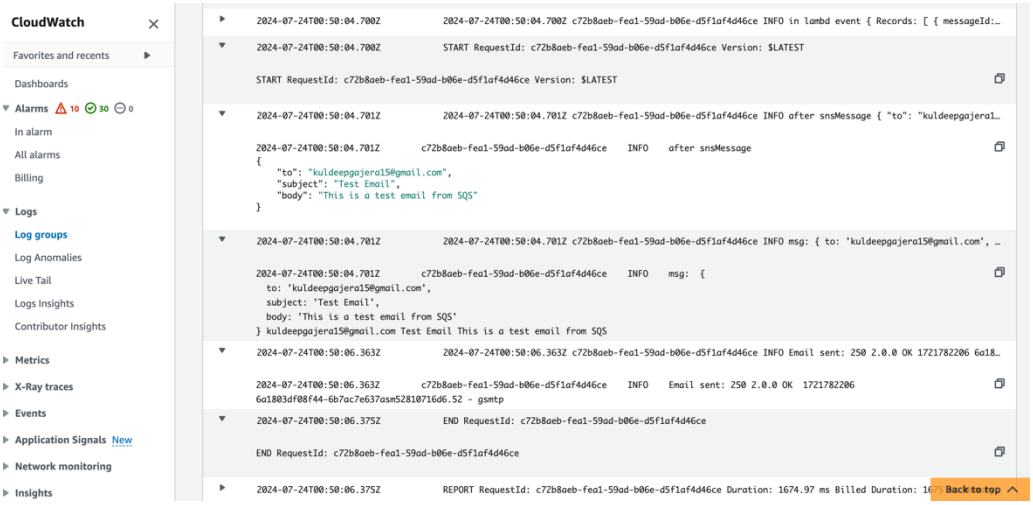

Figure 23: Running the FetchBookingDetails test and displaying execution logs in CloudWatch.

API Testing:

Post Feedback:

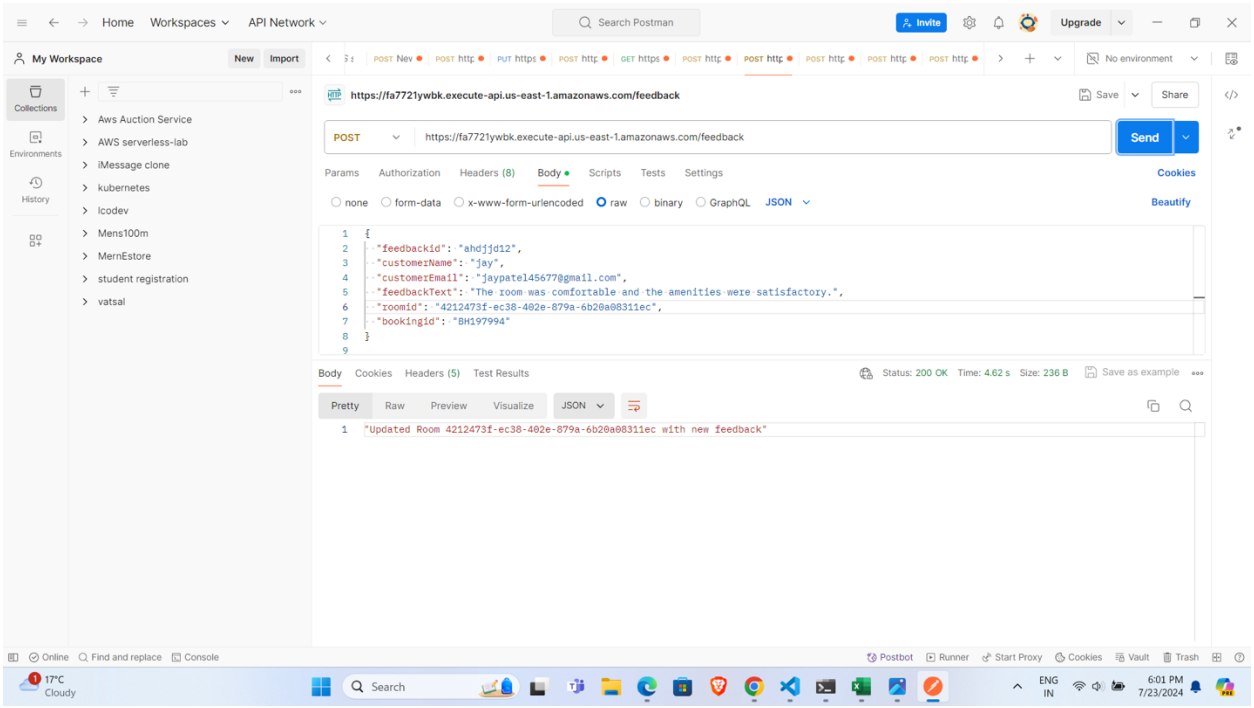

Figure 24: API call execution to Post Feedback for a specific booking of Room, highlighting the request and response.

Extract Sentiment:

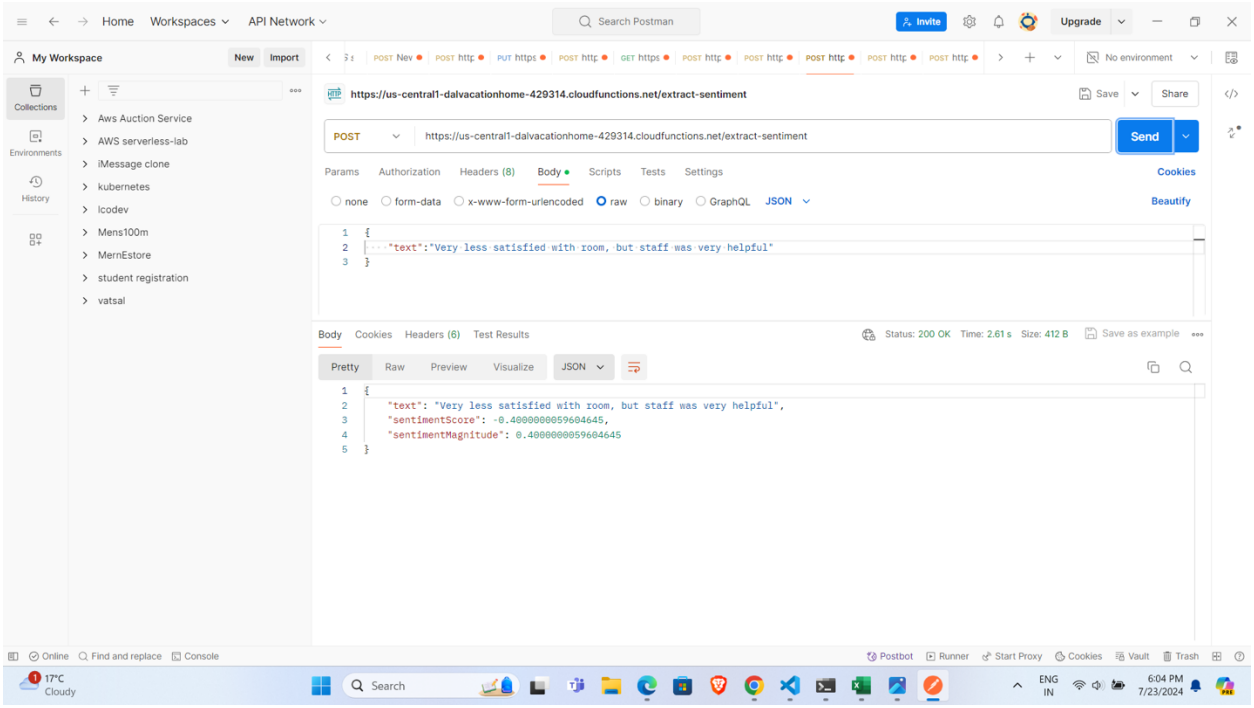

Figure 25: API interaction for extracting sentiment (score and magnitude) of feedback text, displaying request and response details.

Store Login Statistics:

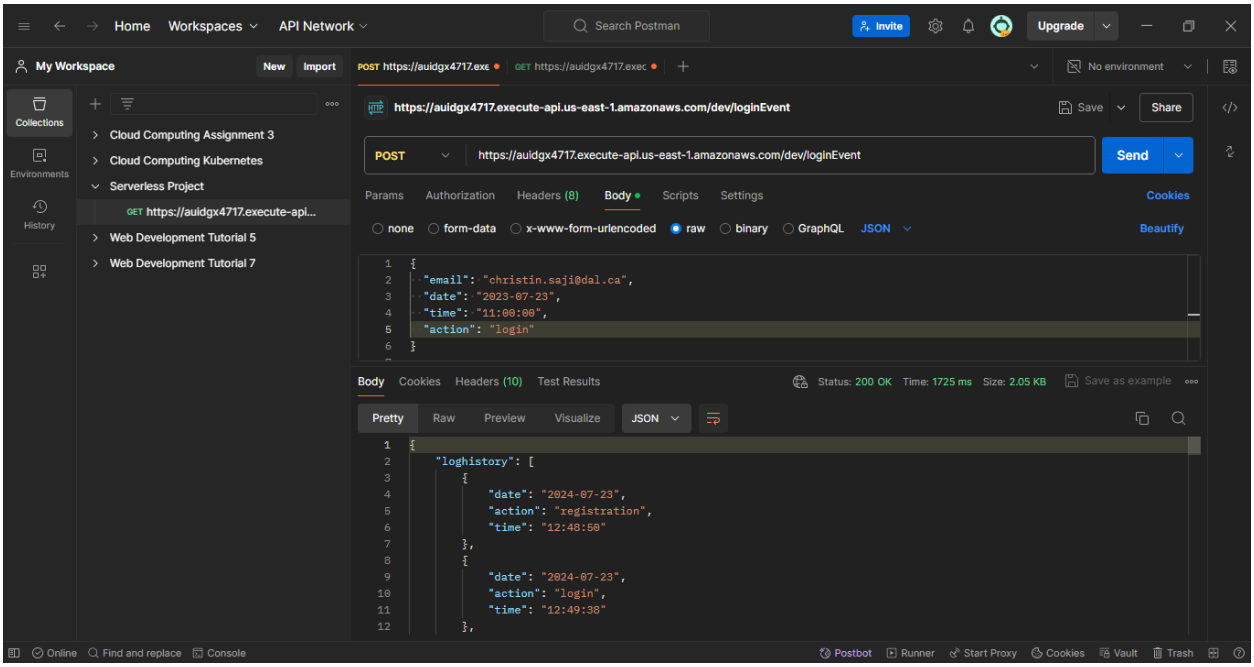

Figure 26: API call to storeLoginStatistics lambda to store logging events of the user

Fetch User Data:

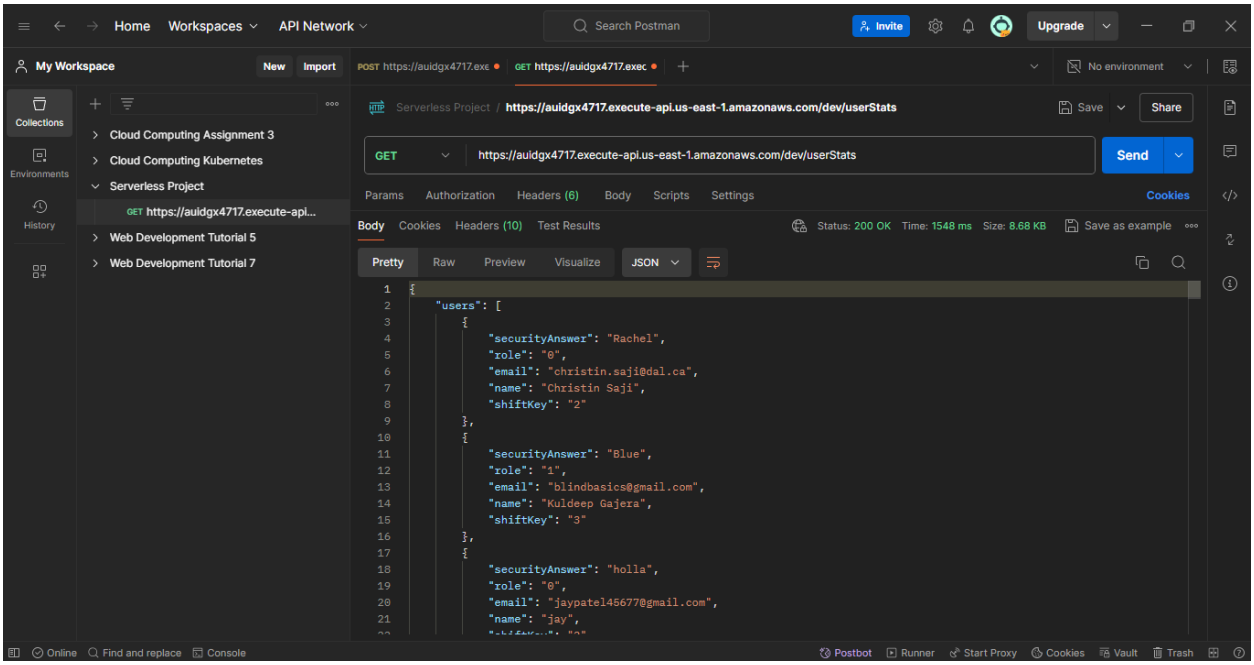

Figure 27: API call to fetchUserData lambda to fetch the user details and the logging events of the user

Fetch Admins:

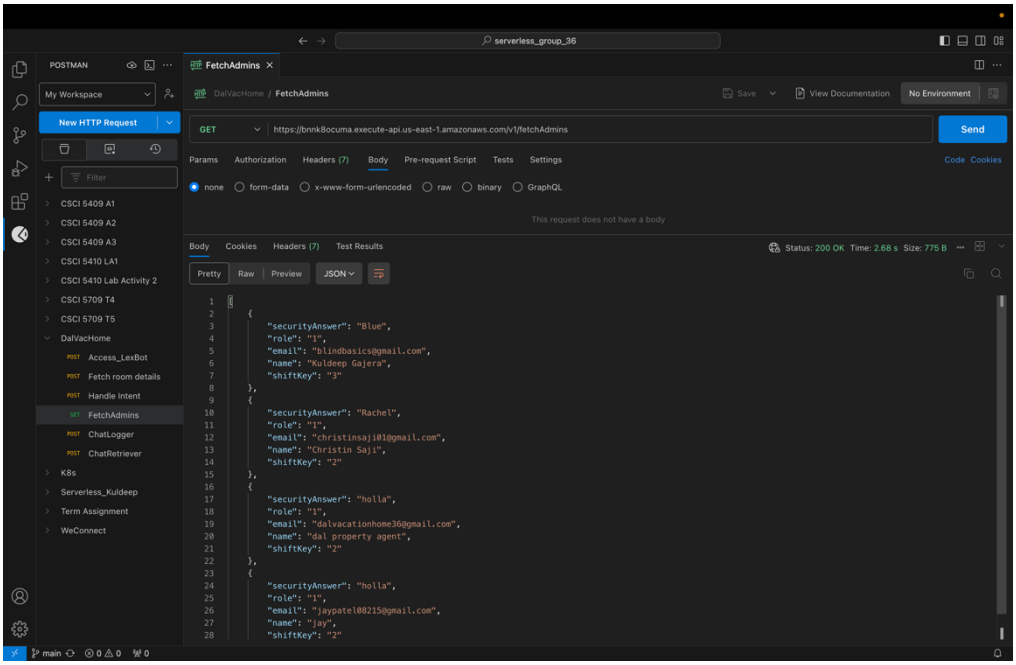

Figure 28: API call to fetchUserData lambda to fetch the user details and the logging events of the user

Log Chats:

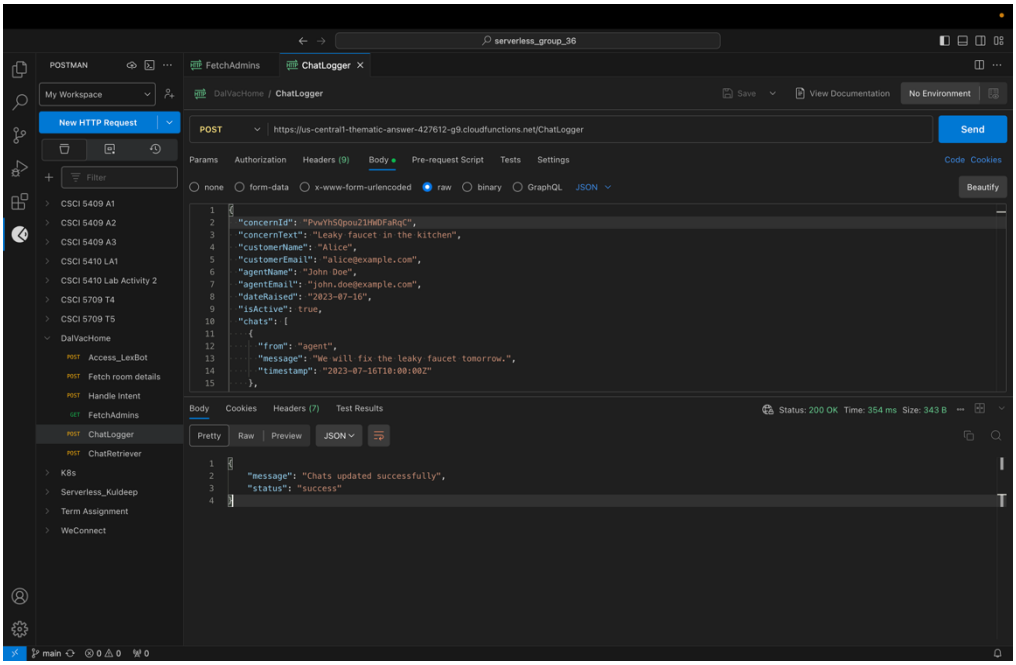

Figure 29: API call to fetchUserData lambda to fetch the user details and the logging events of the user

Retrieve Chats:

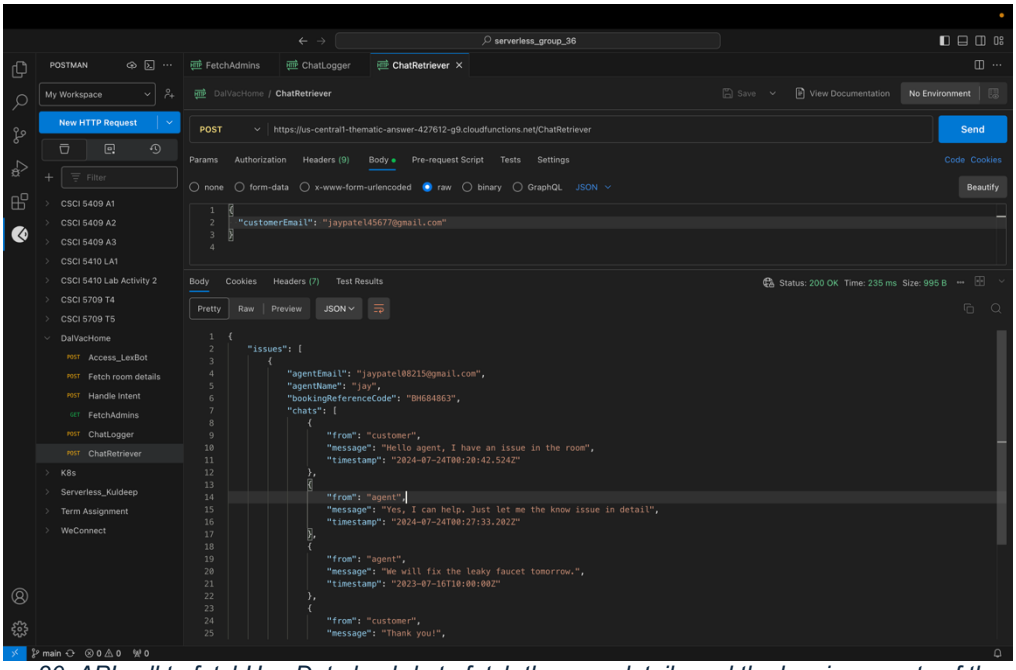

Figure 30: API call to fetchUserData lambda to fetch the user details and the logging events of the user

Evidence of Testing for completed modules:

Notification module Testing:

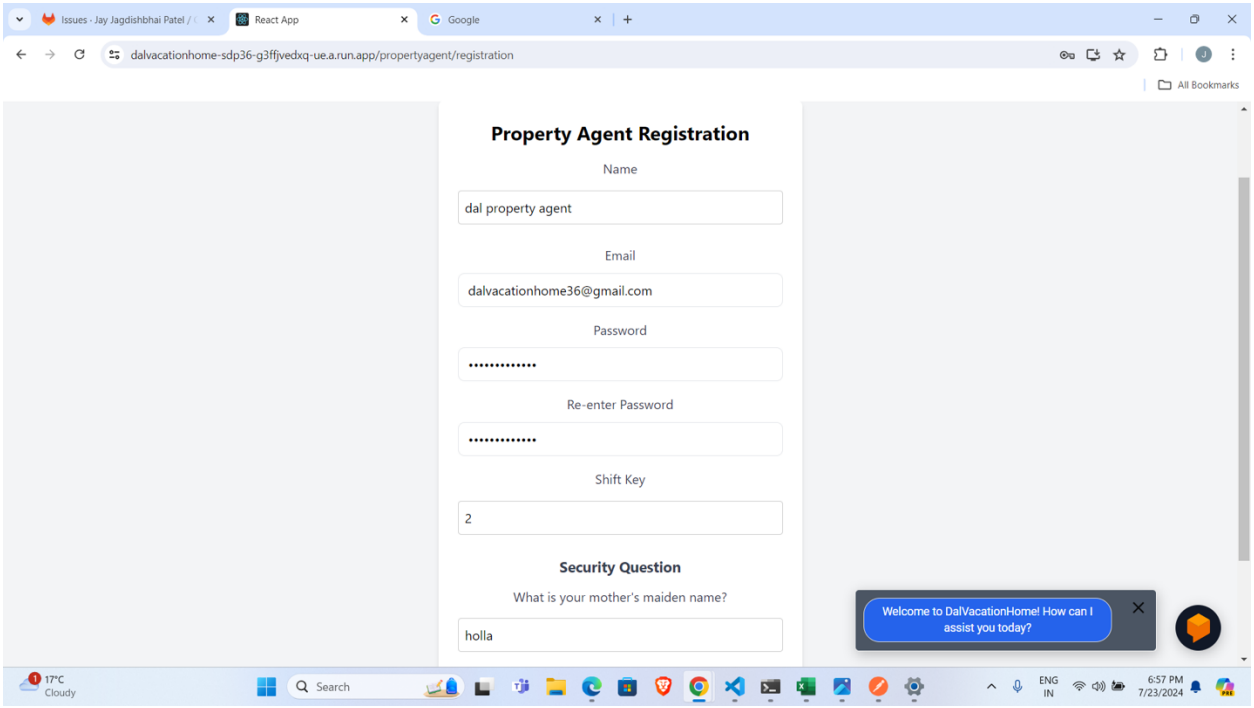

Figure 31: Screenshot of the Property Agent registration form, including fields for shift key and security question.

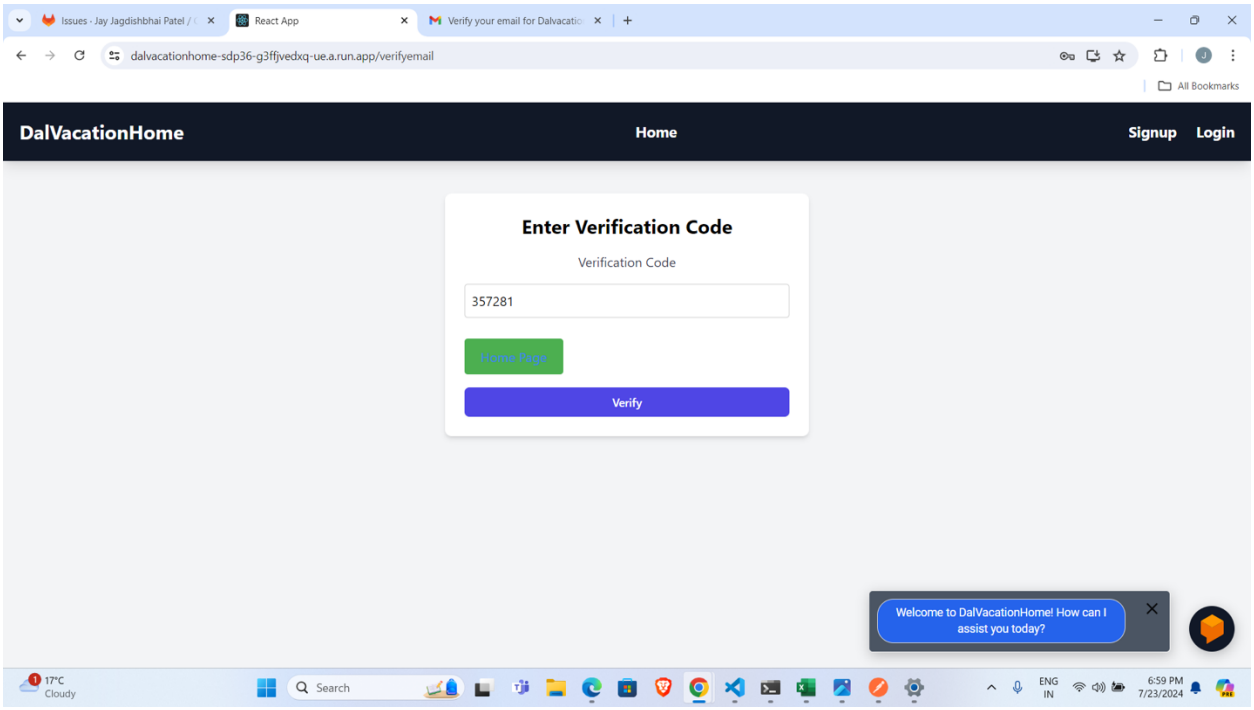

Figure 32: Display of the verification code received via email during registration.

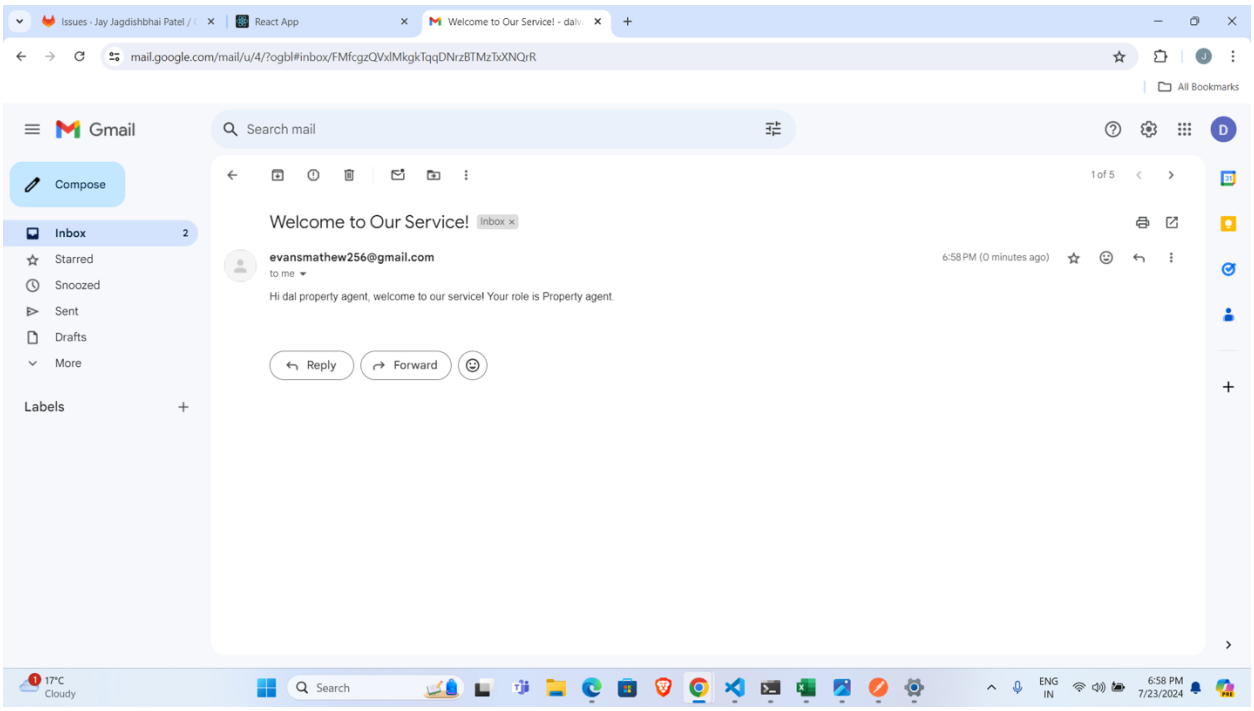

Figure 33: Display of the verification code received via email during registration.

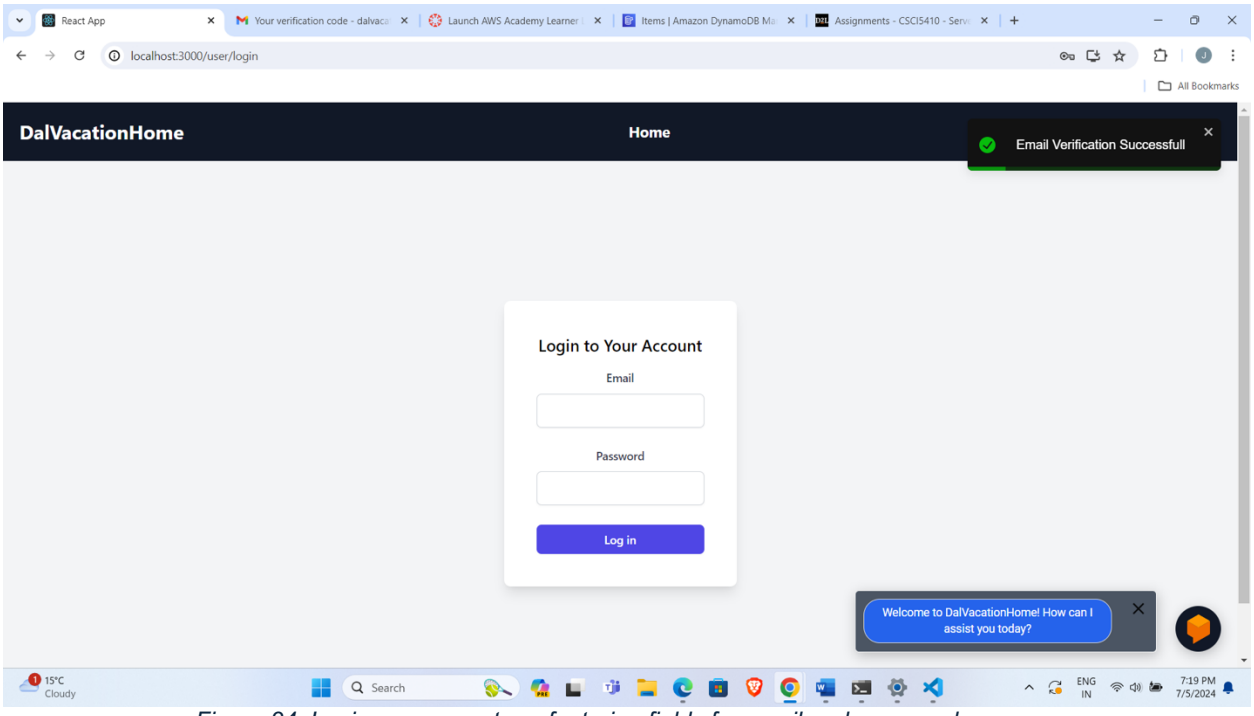

Figure 34: Login screen capture, featuring fields for email and password.

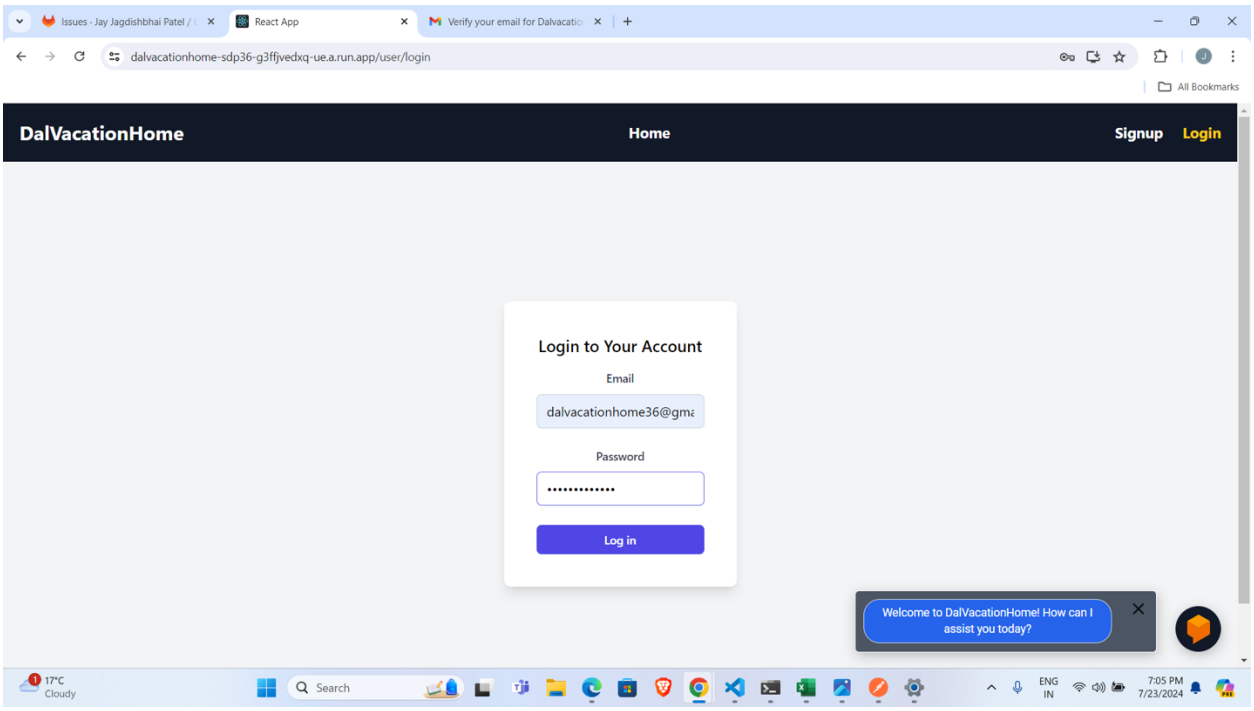

Figure 35: Attempt to log in showing the screen during the process.

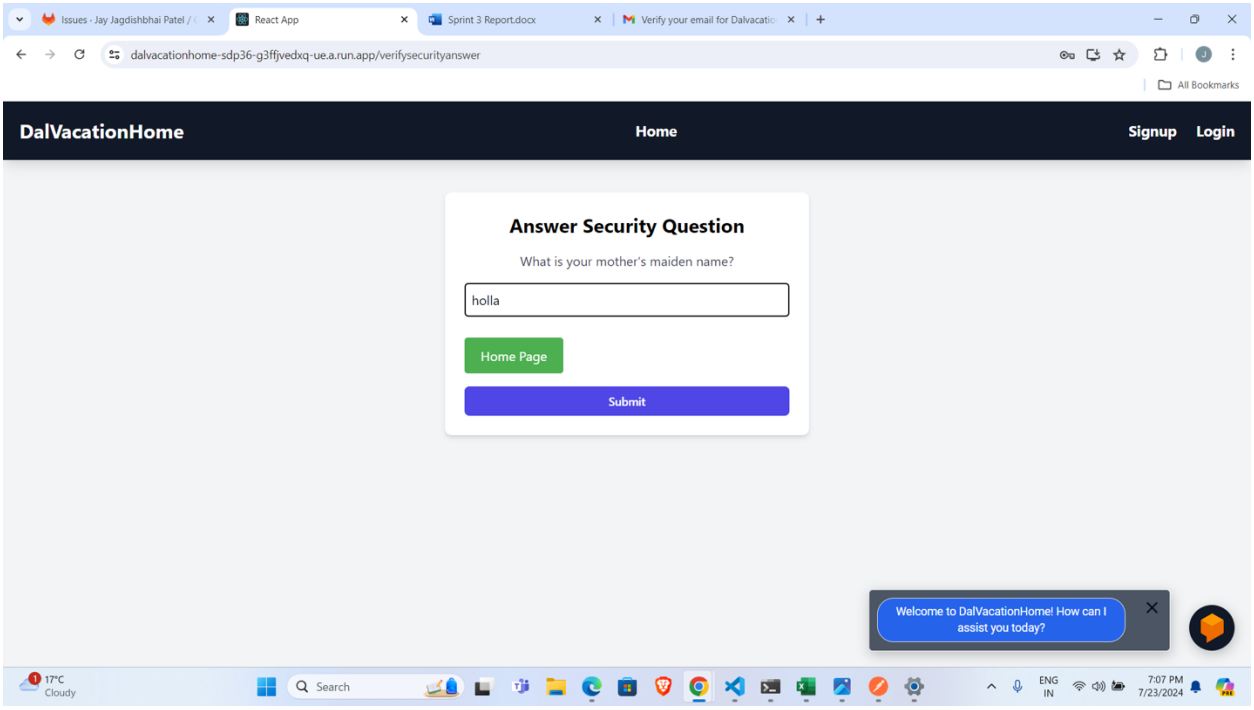

Figure 36: Security question prompt during login for additional verification.

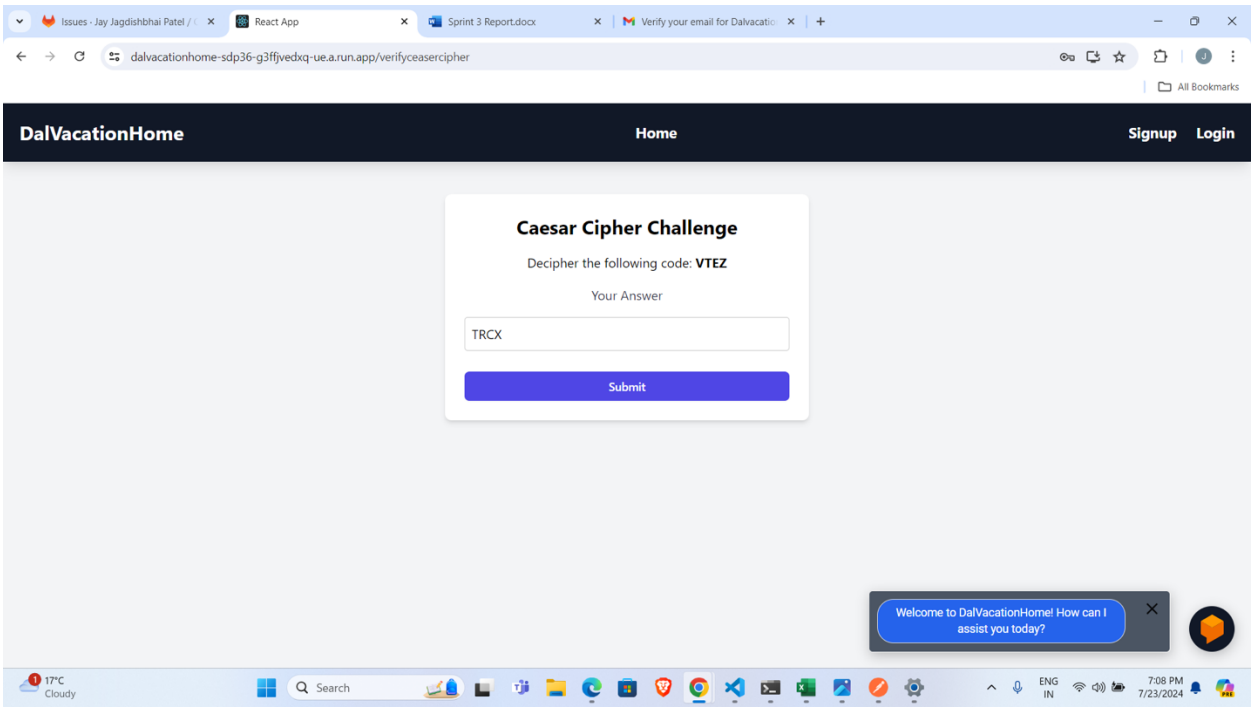

Figure 37: Caesar cipher challenge presented to the user as a login verification step.

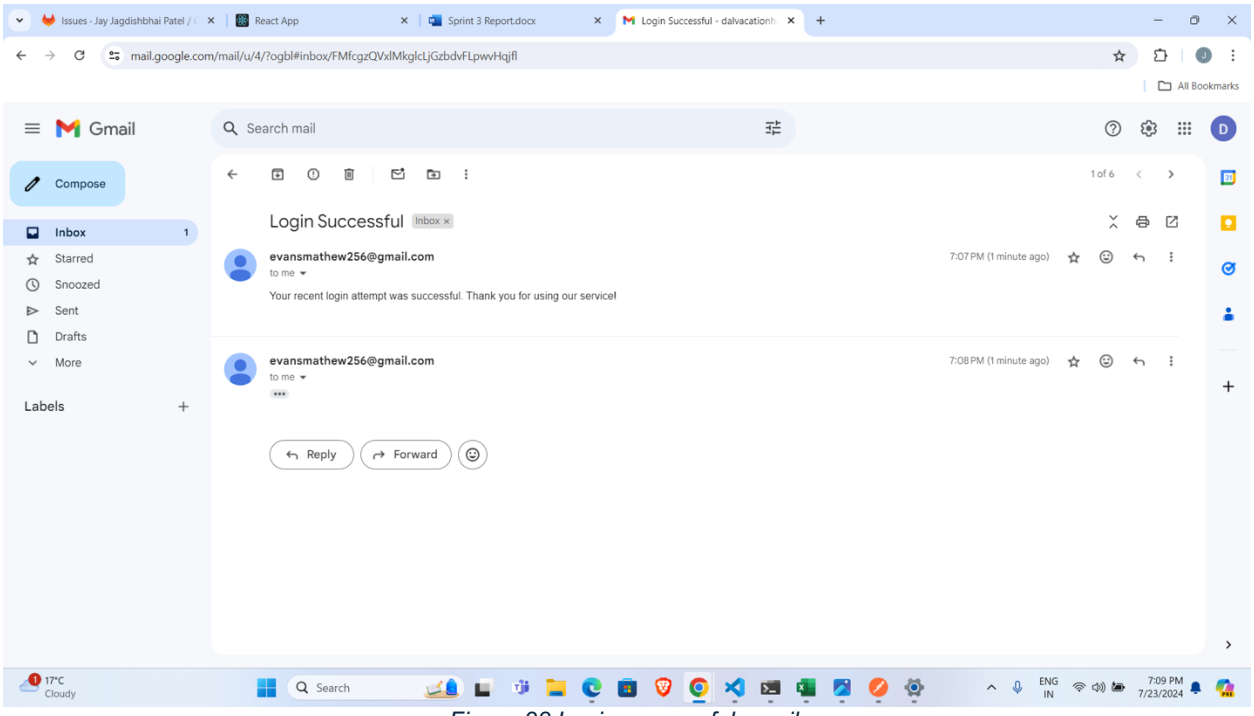

Figure 38:Login successful email.

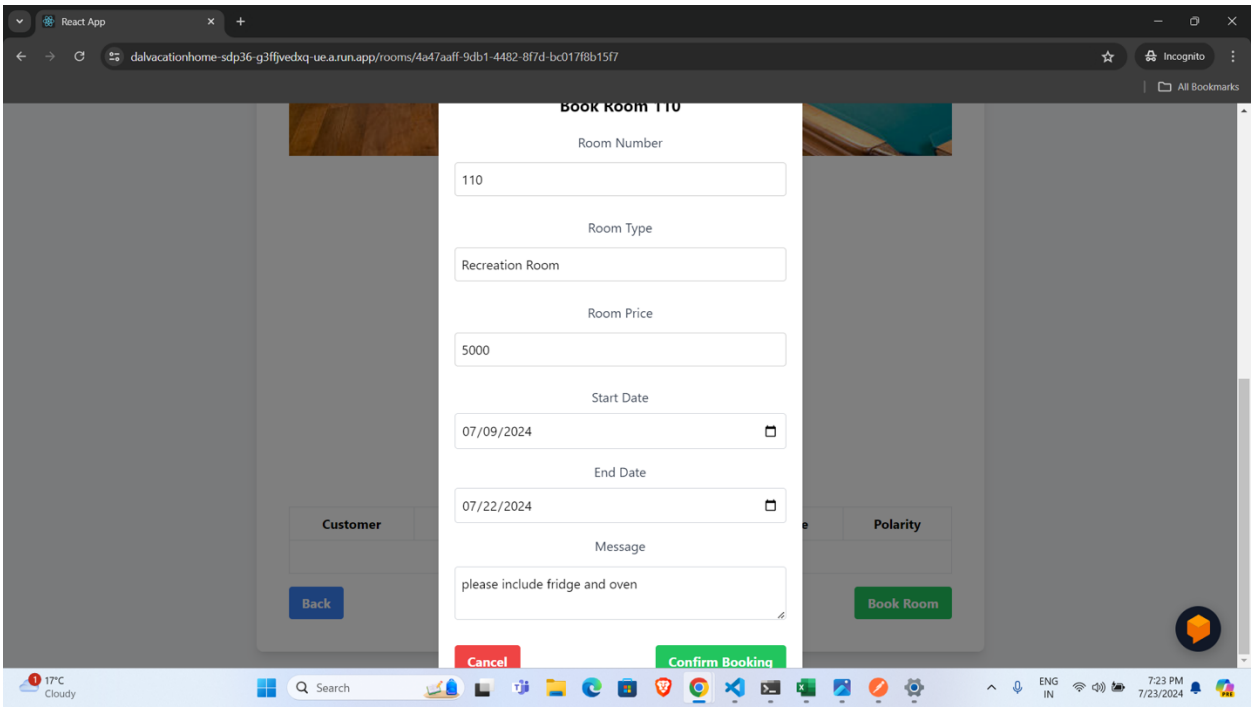

Figure 39: Page for booking confirmation by user.

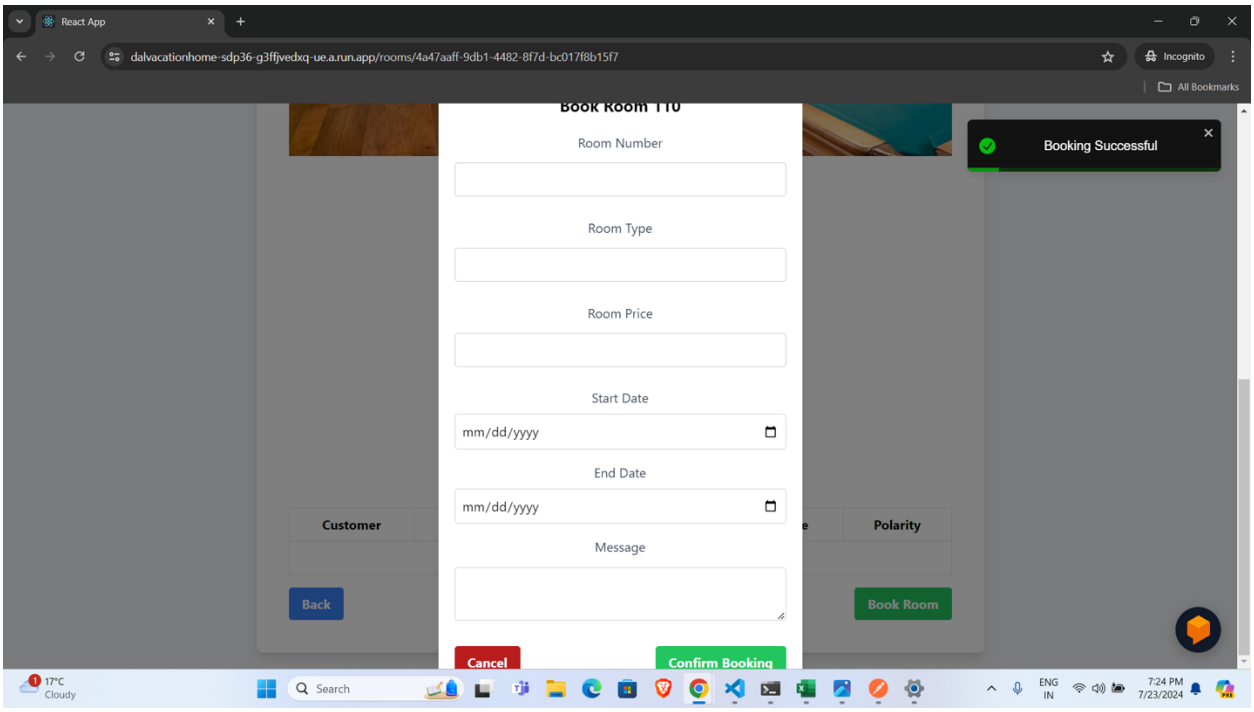

Figure 40: Page 2 for booking confirmation by user

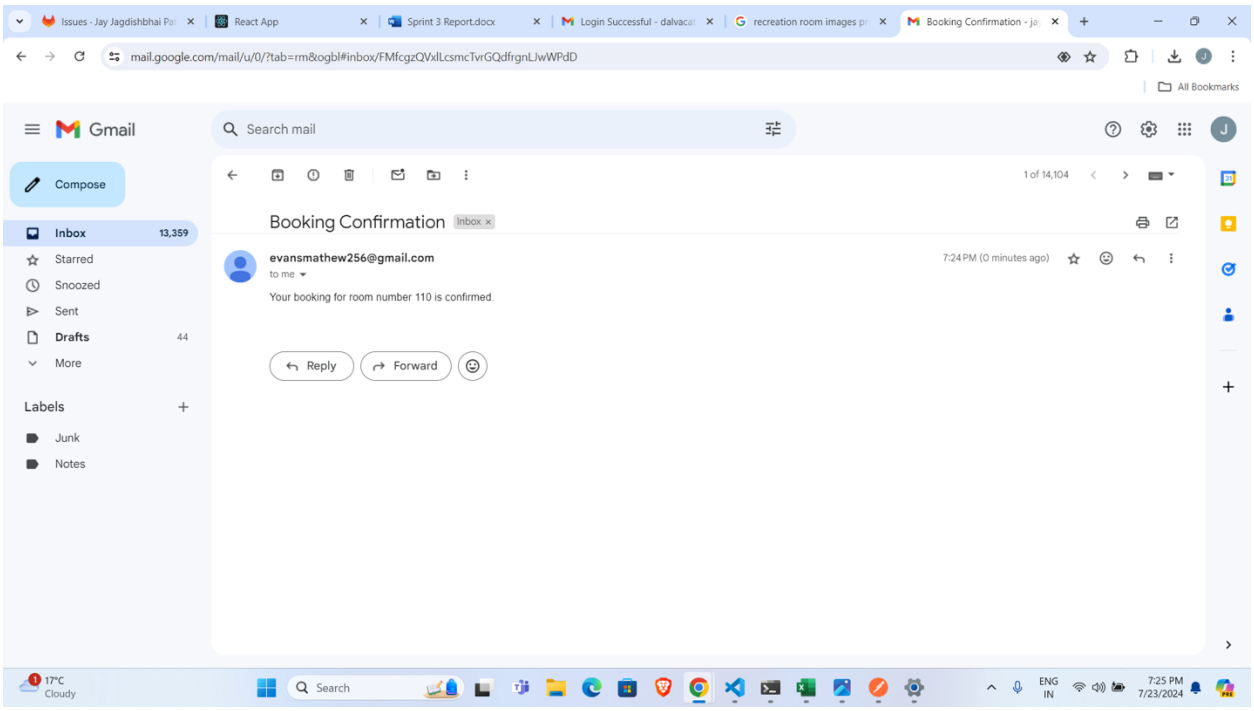

Figure 41: Email notification for confirming booking by user.

Data Analysis and Visualization Module:

All users can access Feedback of Room:

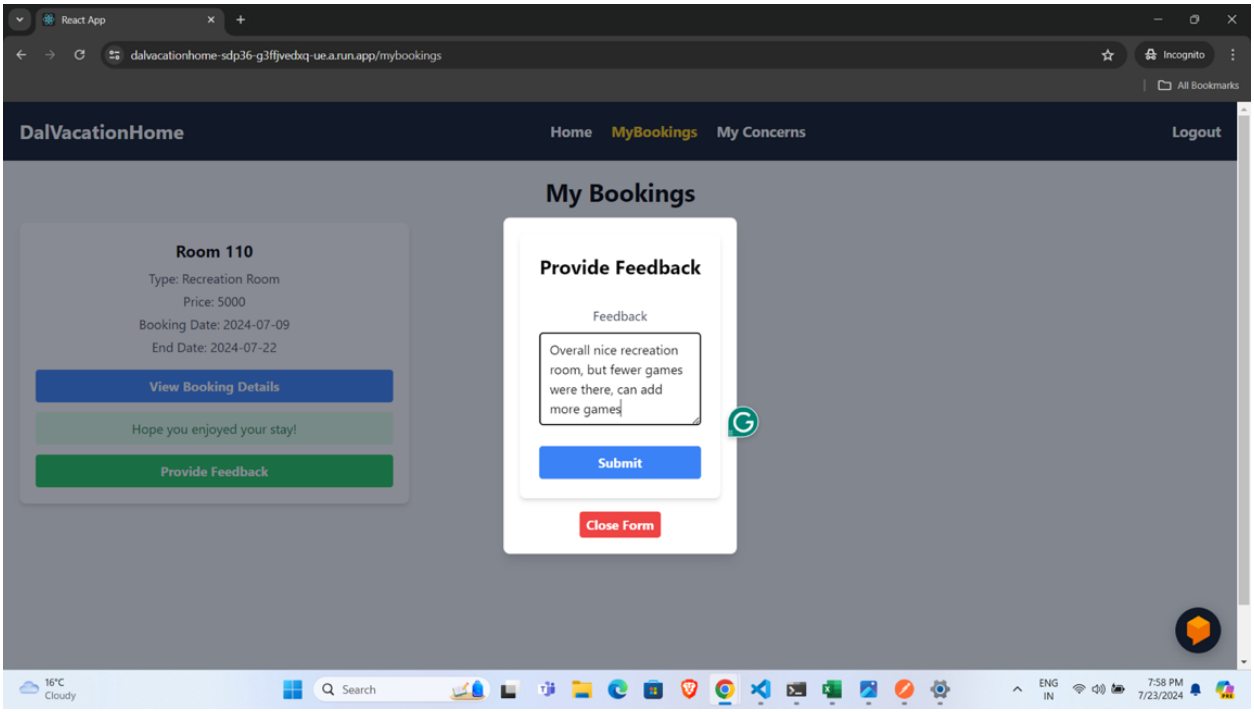

Figure 42: Page for providing feedback

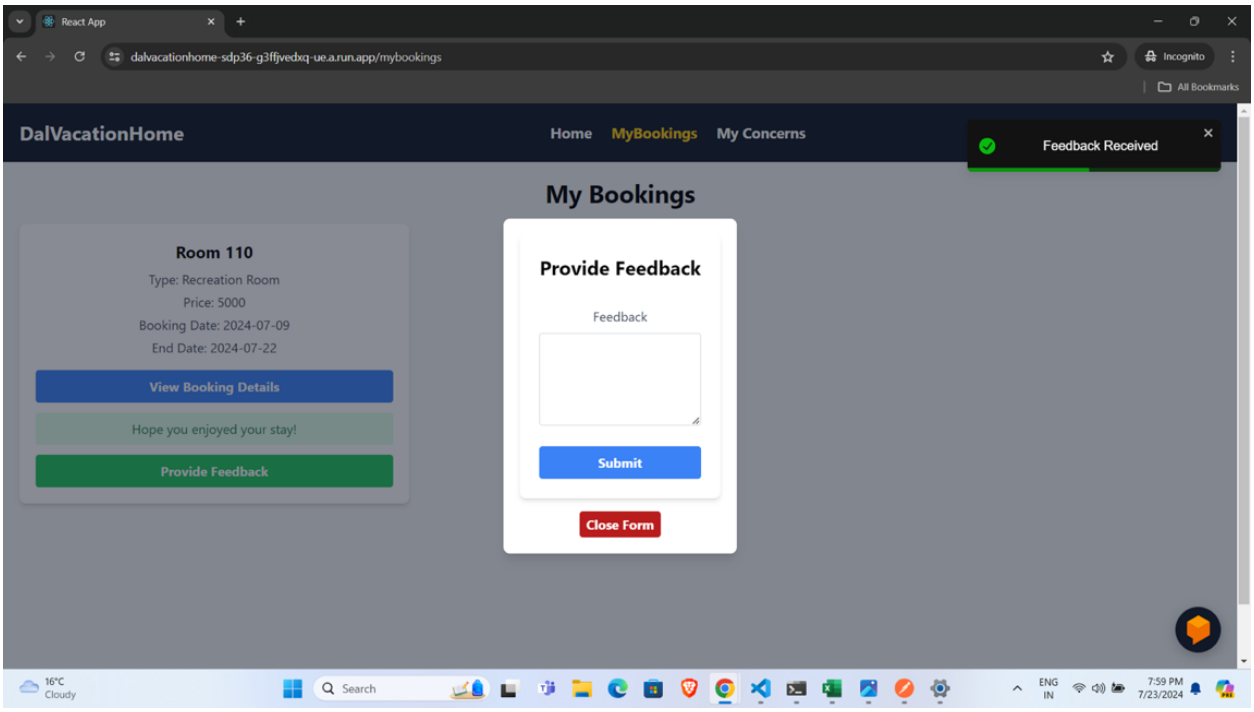

Figure 43: Feedback provided by user form.

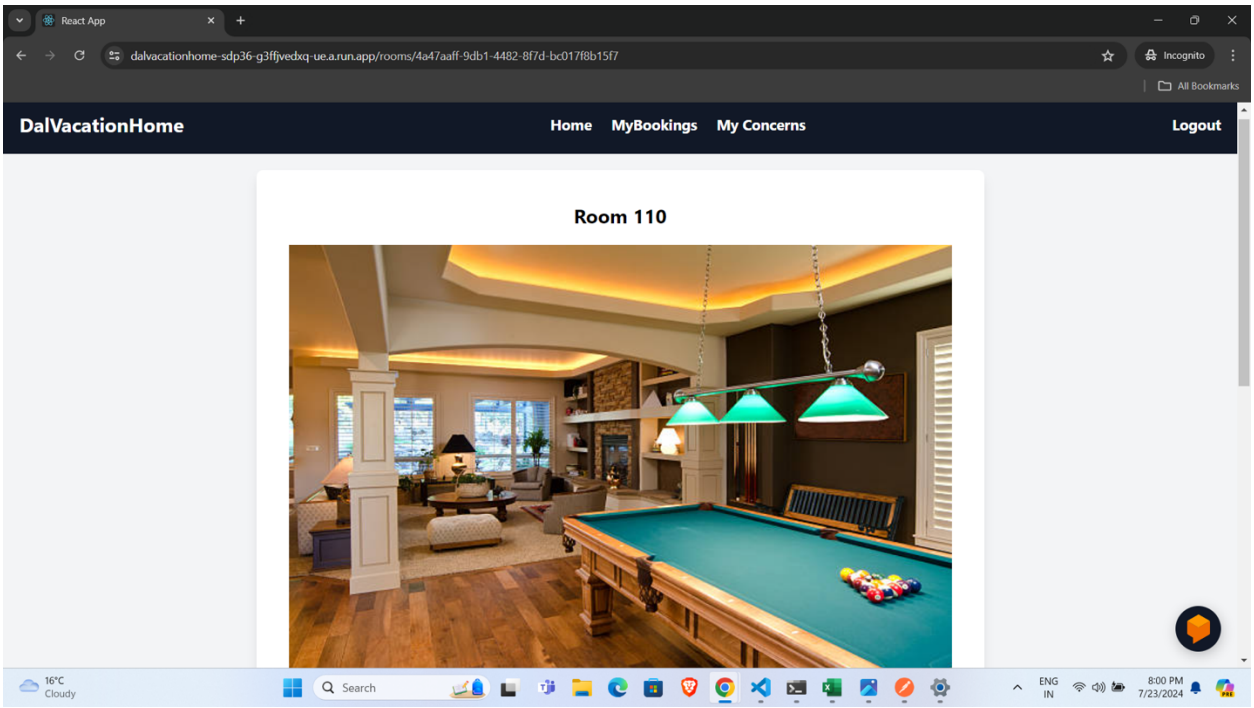

Figure 44: Room Image along with room number on the homepage.

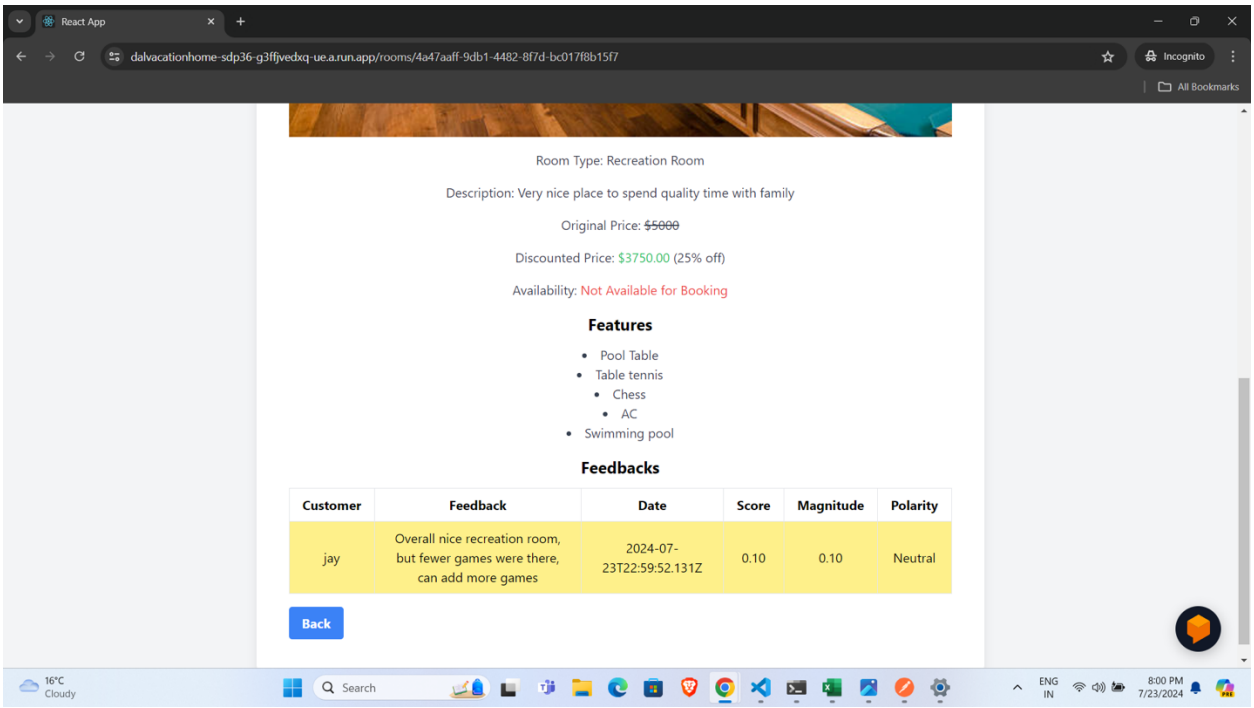

Figure 45: Book details such as number, type, price, feedback, etc.

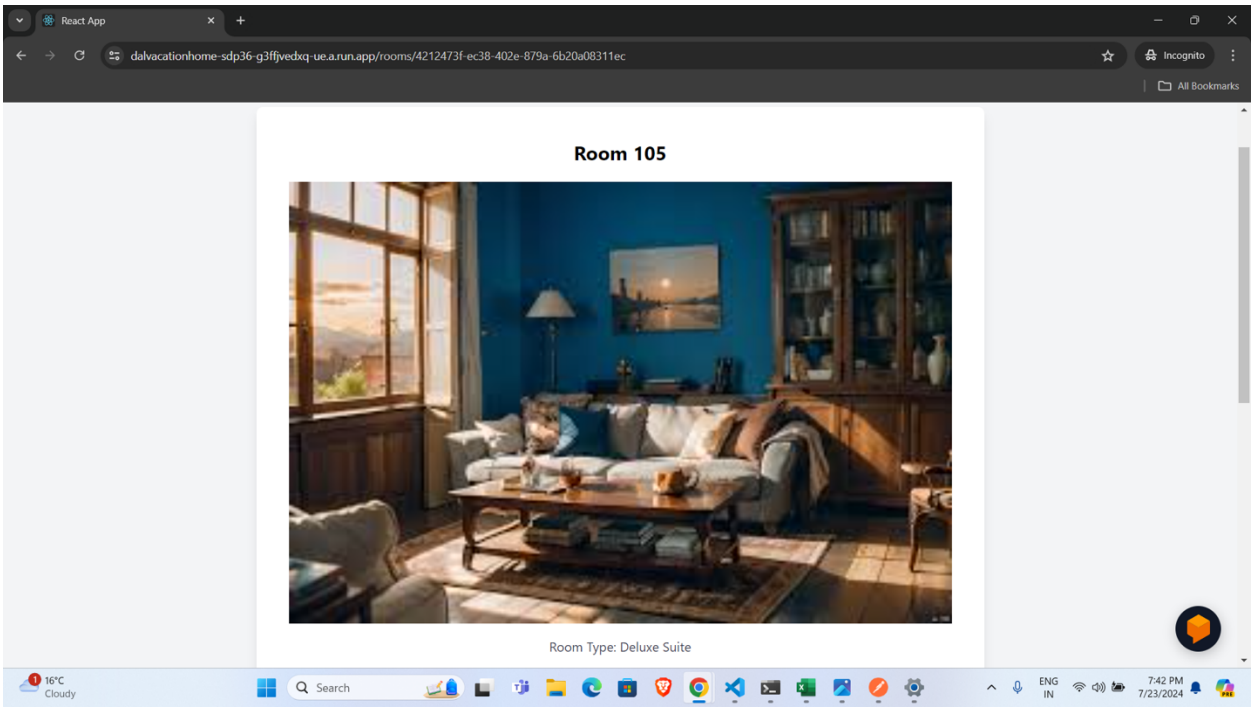

Figure 46: Room 105 with its image.

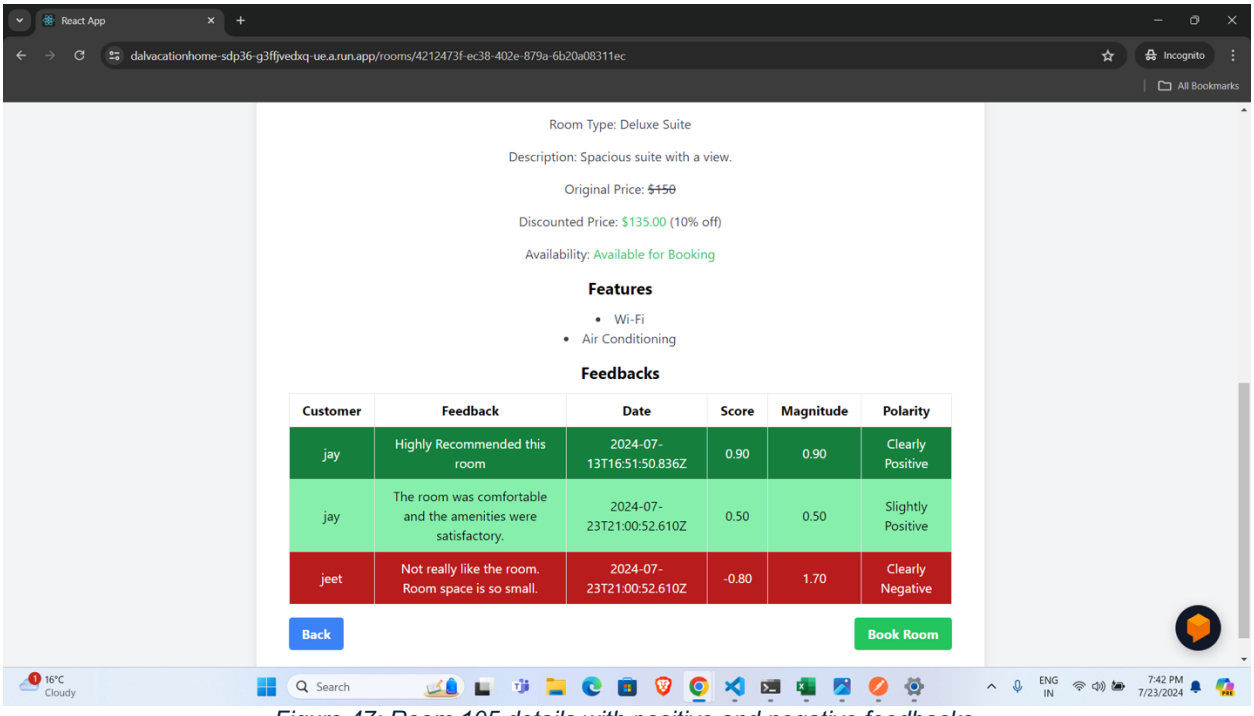

Figure 47: Room 105 details with positive and negative feedbacks.

Property Agent Side:

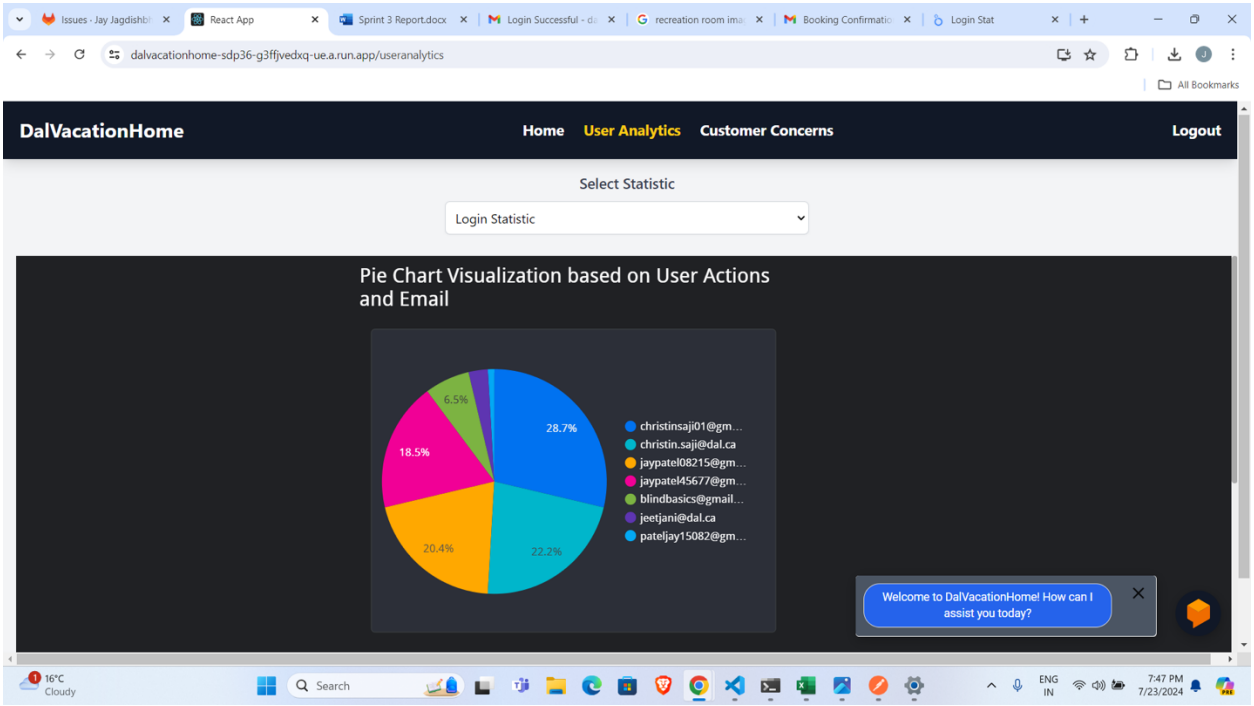

Figure 48: Visualization of user Actions and Email.

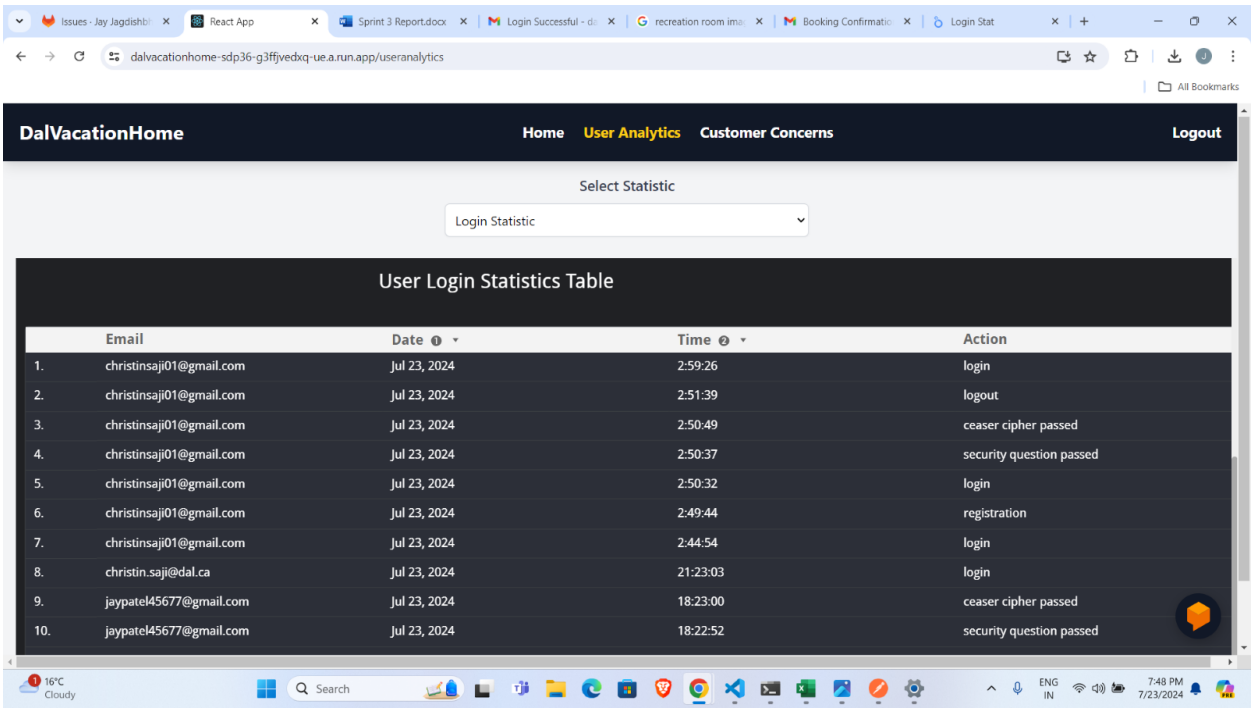

Figure 49: User login Statistics Table.

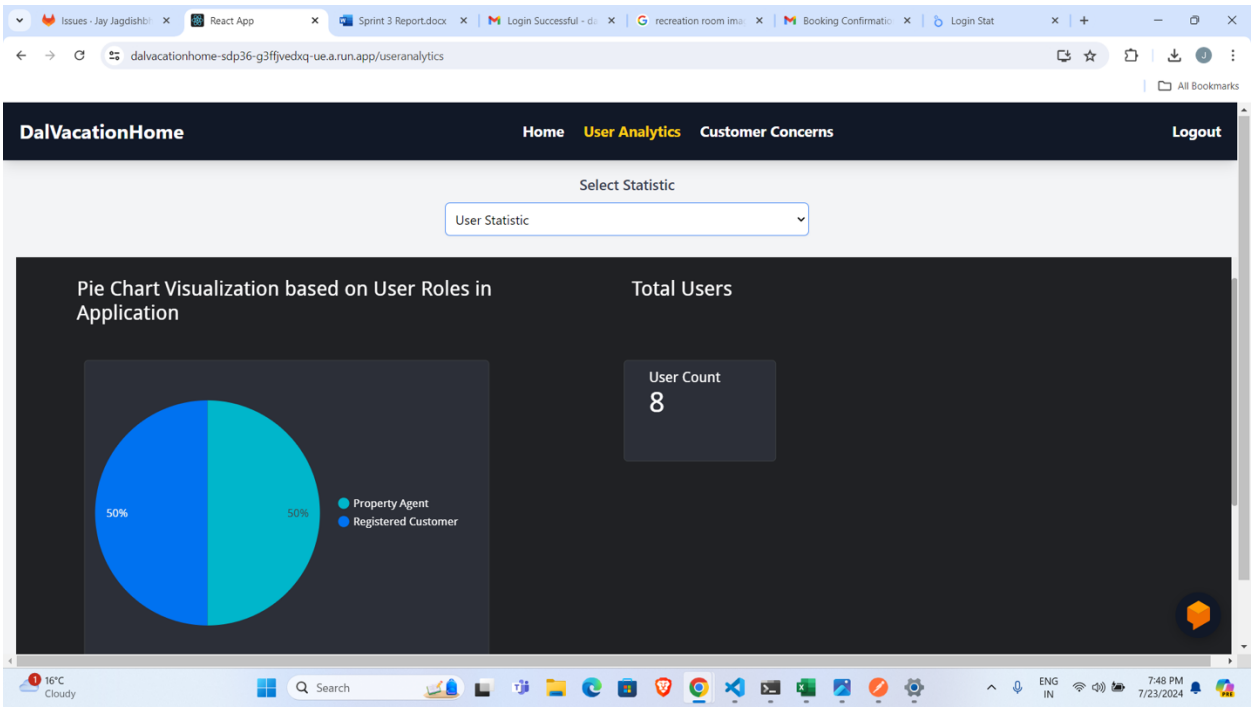

Figure 50: Pie chart visualization based on roles in the application and user count.

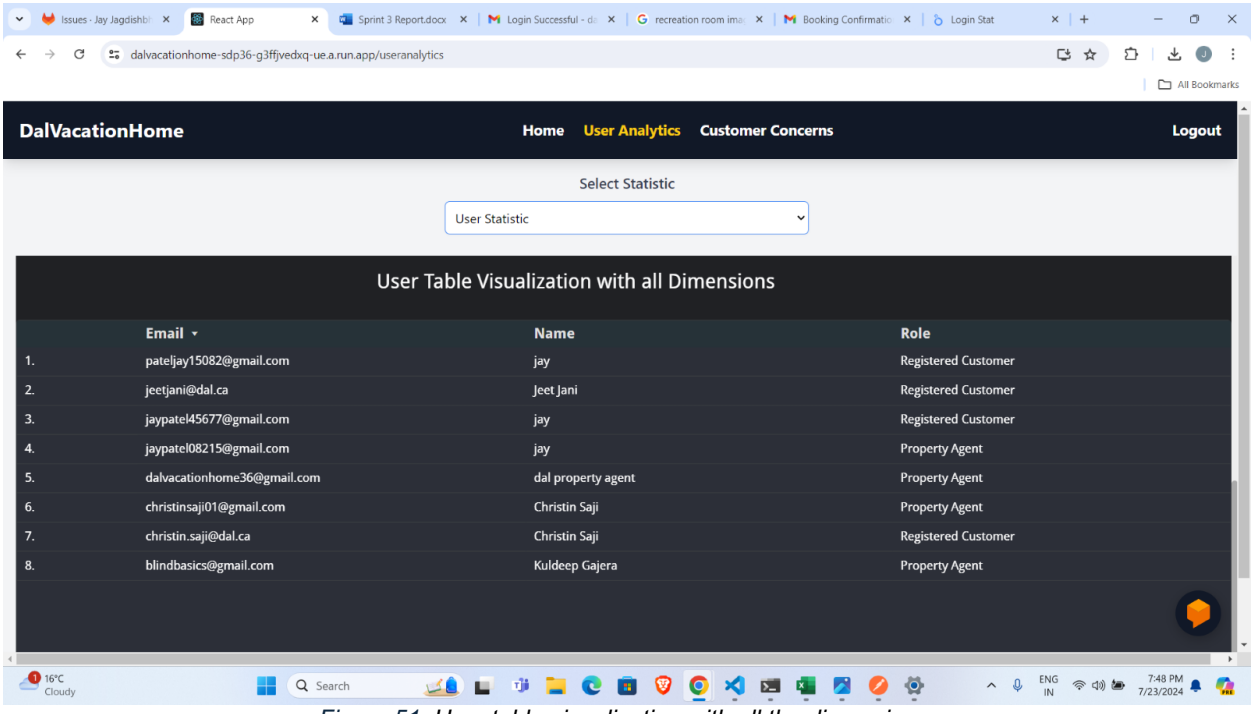

Figure 51: User table visualization with all the dimensions.

Virtual Assistance and Message Passing Module:

Customer Side:

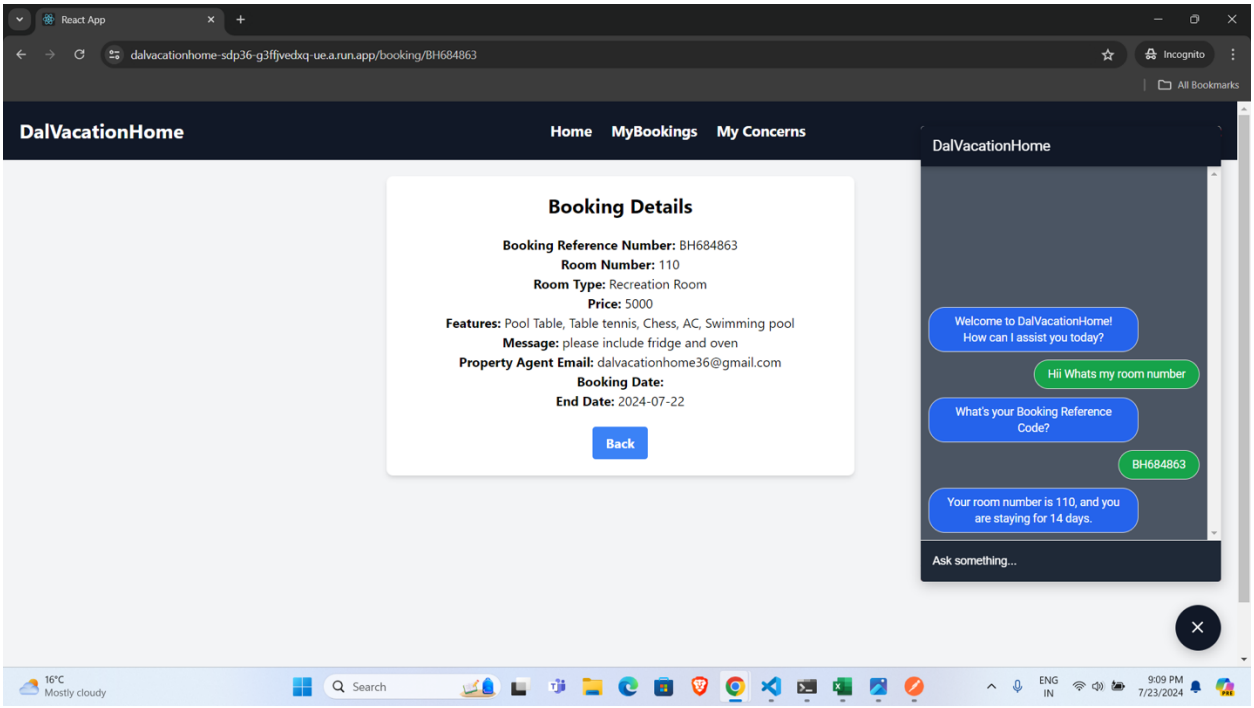

Figure 52: Booking details and chatting with chatbot from customer side.

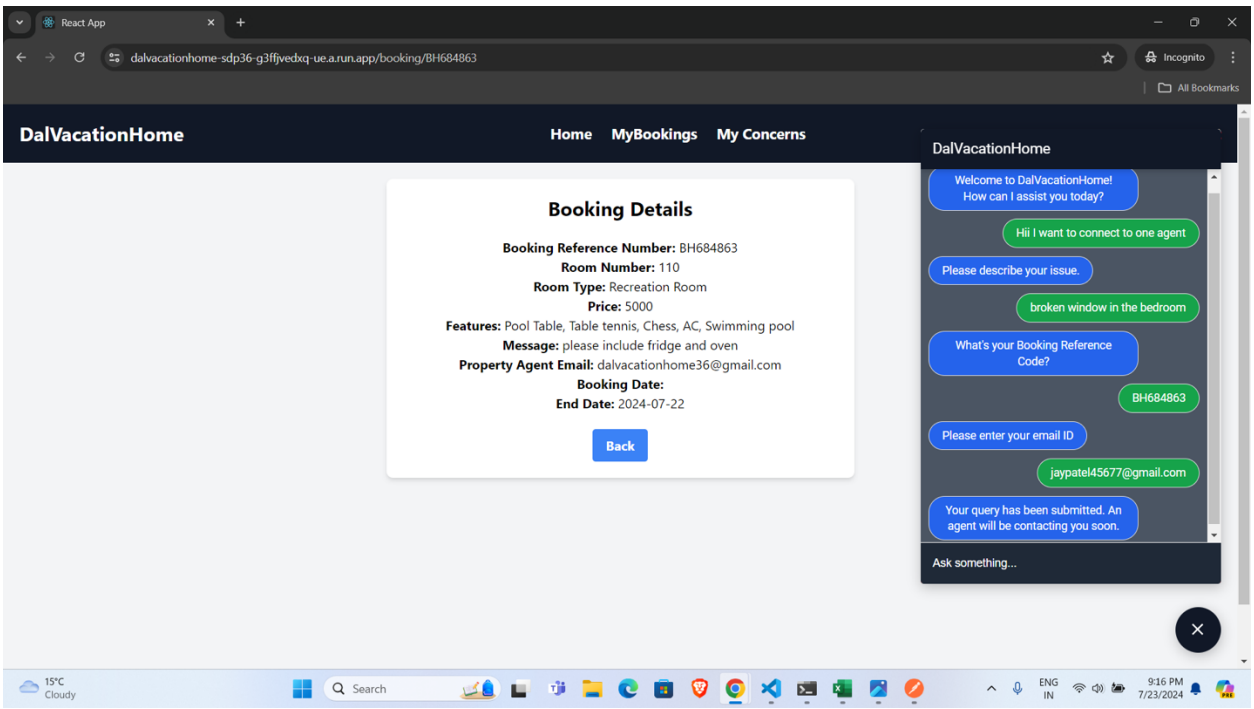

Figure 53: Conversing with chatbot about connecting to property agent.

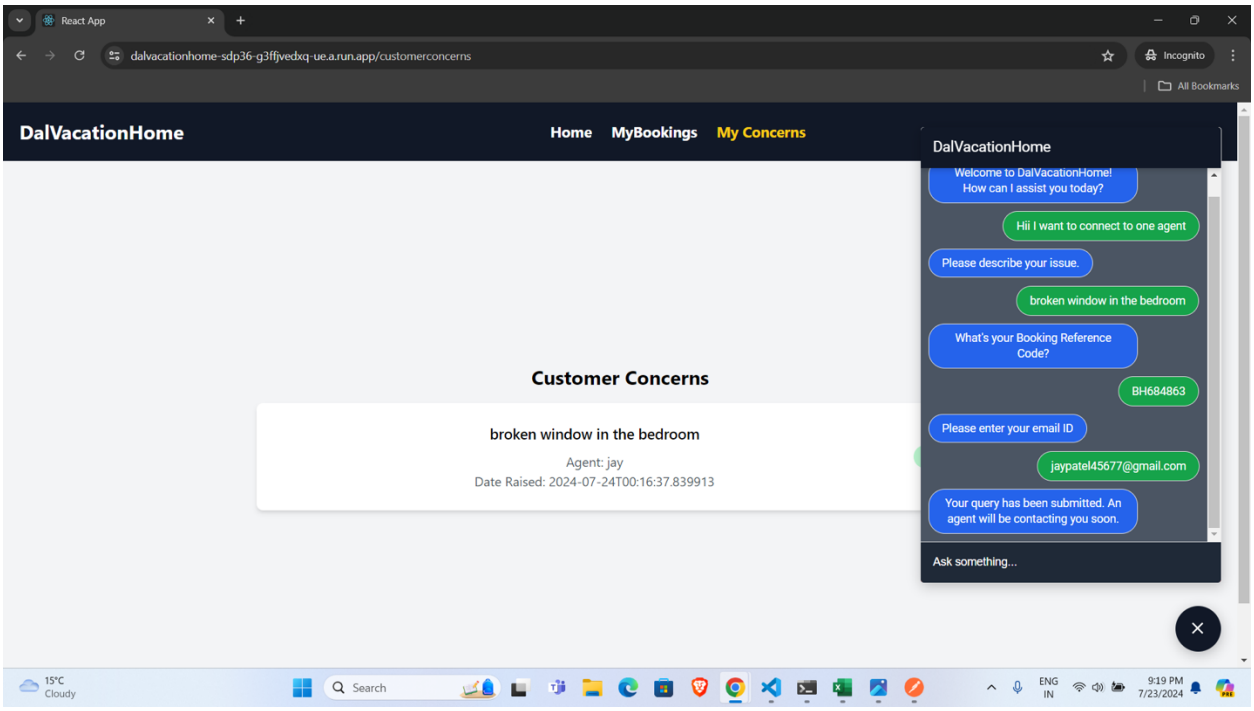

Figure 54: Issue provided with booking id and email id

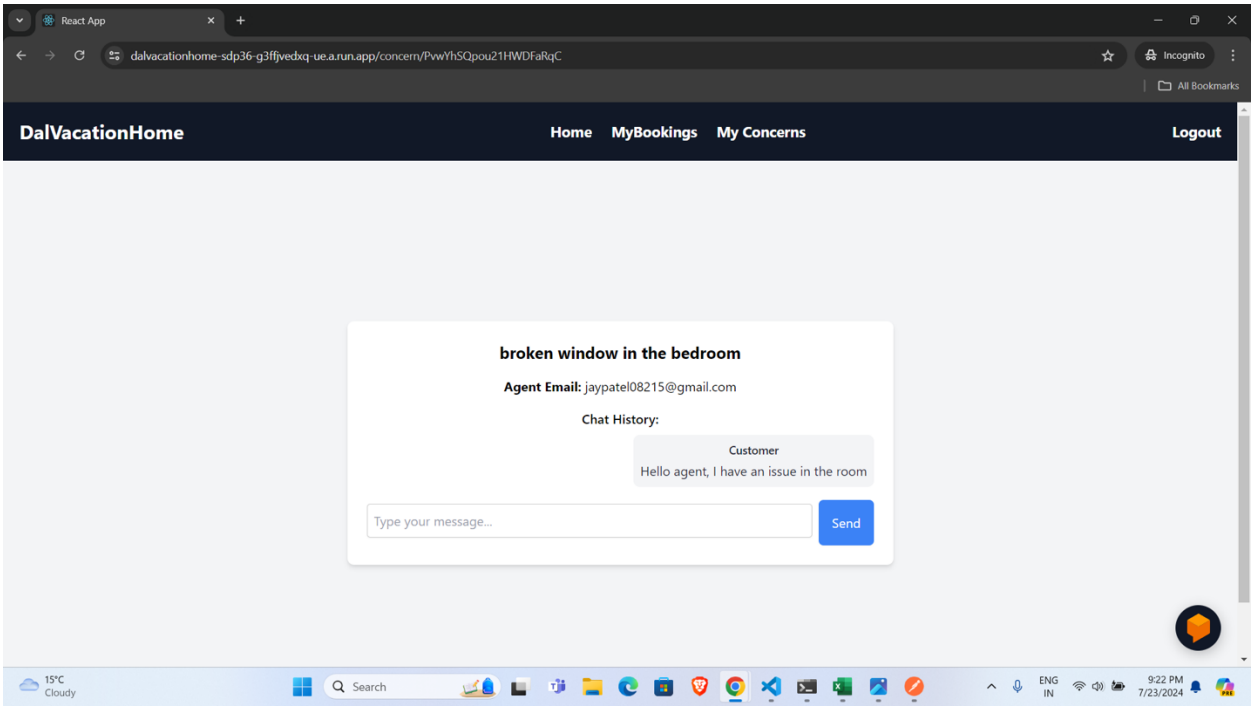

Figure 55: Chatting with property agent about the iseue.

Property Agent Side:

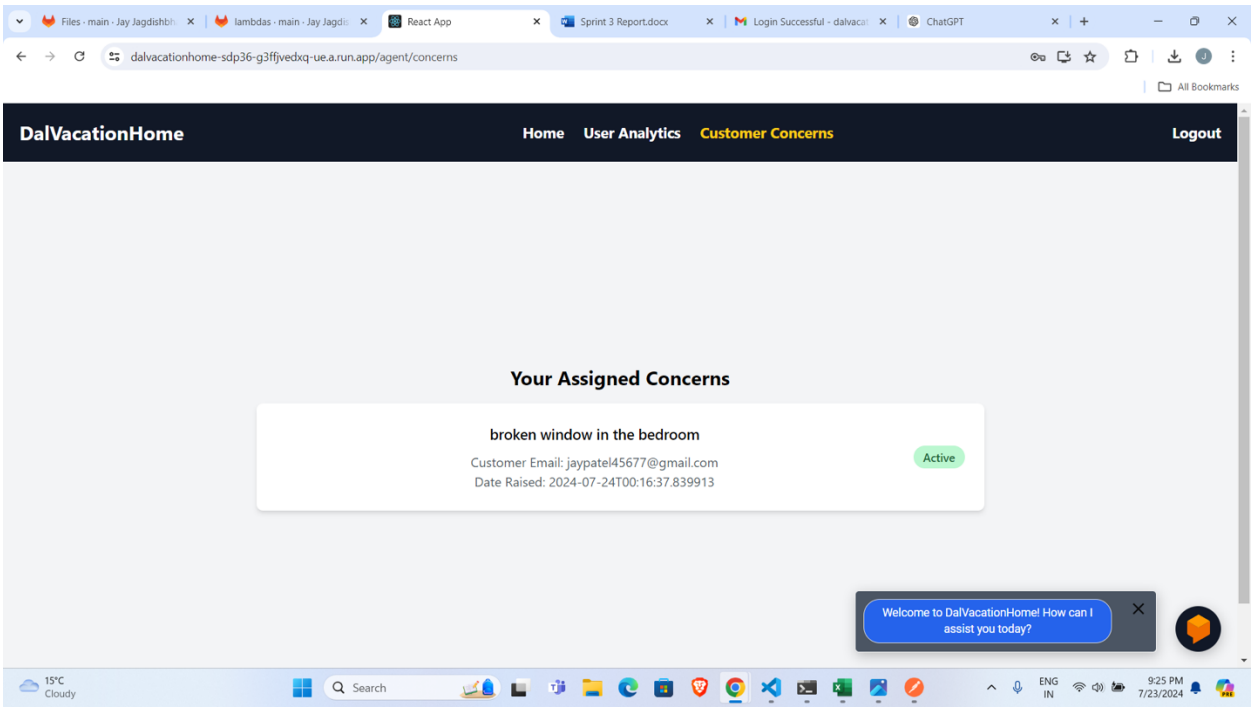

Figure 56: ticket concern active between customer and property agent.

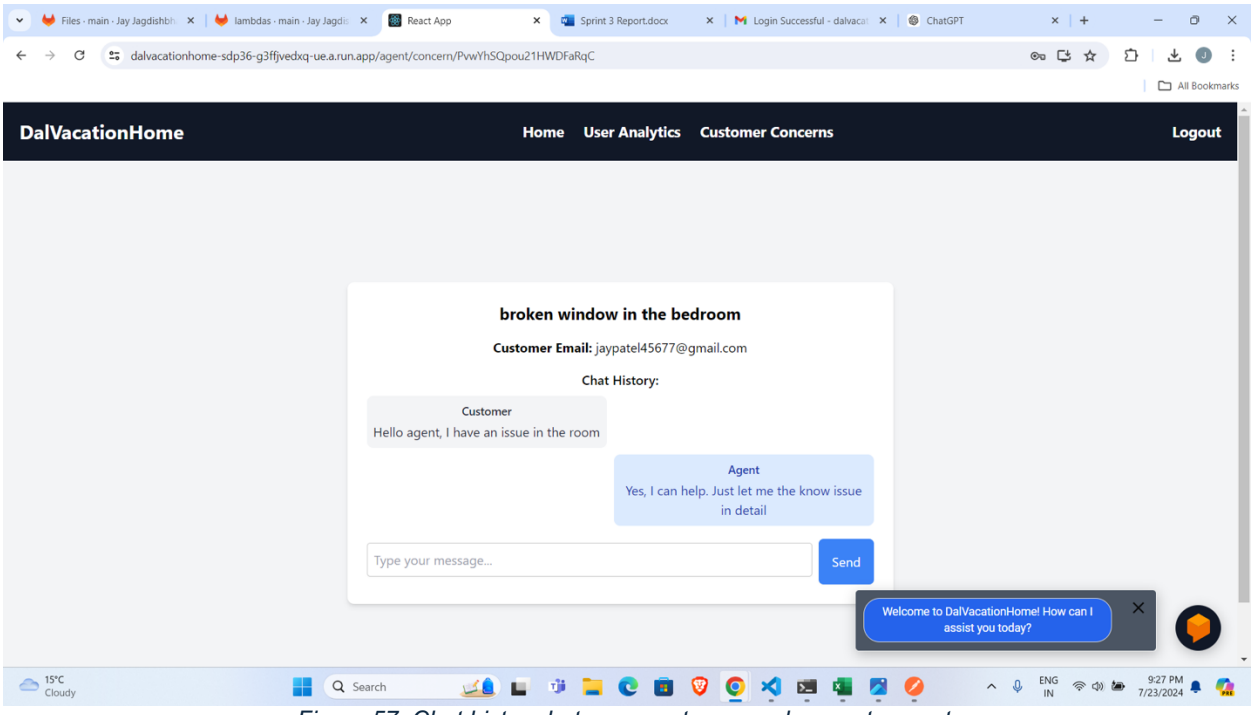

Figure 57: Chat history between customer and property agent.

Customer Side updated chat:

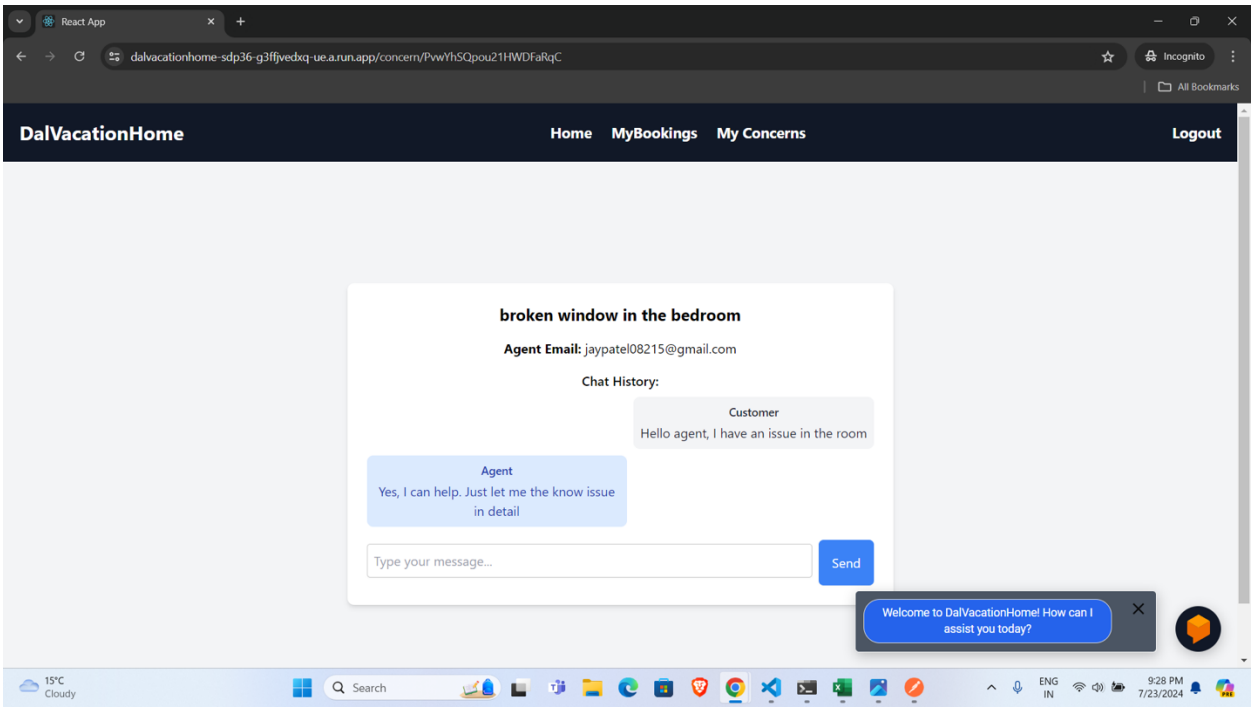

Figure 58: Property agent giving response to customer's query.

Meeting Logs:

Agenda	Outcome of discussion	Link to meeting recordings
• Share status of each module. • Discuss completed and pending parts. • Identify issues in the chatbot. • Prepare for the upcoming meeting with Head TA. • Plan live meeting for module presentations.	• Shared status of each module. • Identified completed and pending parts. • Discussed and identified issues in the chatbot that need fixing. • Prepared for the meeting with Head TA. • Decided to hold a live meeting where everyone will describe their module's functionality.	Call with Serverless Team 36-20240715_204120-Meeting Recording.mp4
• Explain each module for better understanding. • Demonstrate module setups.	• Each member explained their module, enhancing team understanding. • Demonstrated module setups, facilitating better comprehension of workflows. • Prepared for the meeting with the TA.	Call with Serverless Team 36-20240716_192449-Meeting Recording.mp4
• Discuss SNS & SQS module. • Plan and start the report. • Share access to final architecture. • Identify remaining project tasks. • Plan final demo video recording. • Finalize Sprint 3 report. • Identify and plan to fix bugs in the application.	• Discussed the workings of the SNS & SQS module. • Planned the report structure and initiated the writing process. • Shared access to the final architecture for module modifications. • Identified remaining project tasks • Planned the final demo video recording and scheduled a meeting to finalize the Sprint 3 report. • Identified bugs in the application and planned necessary fixes.	Meeting in Serverless Team 36-20240722_192536-Meeting Recording.mp4

Chat Screenshots:

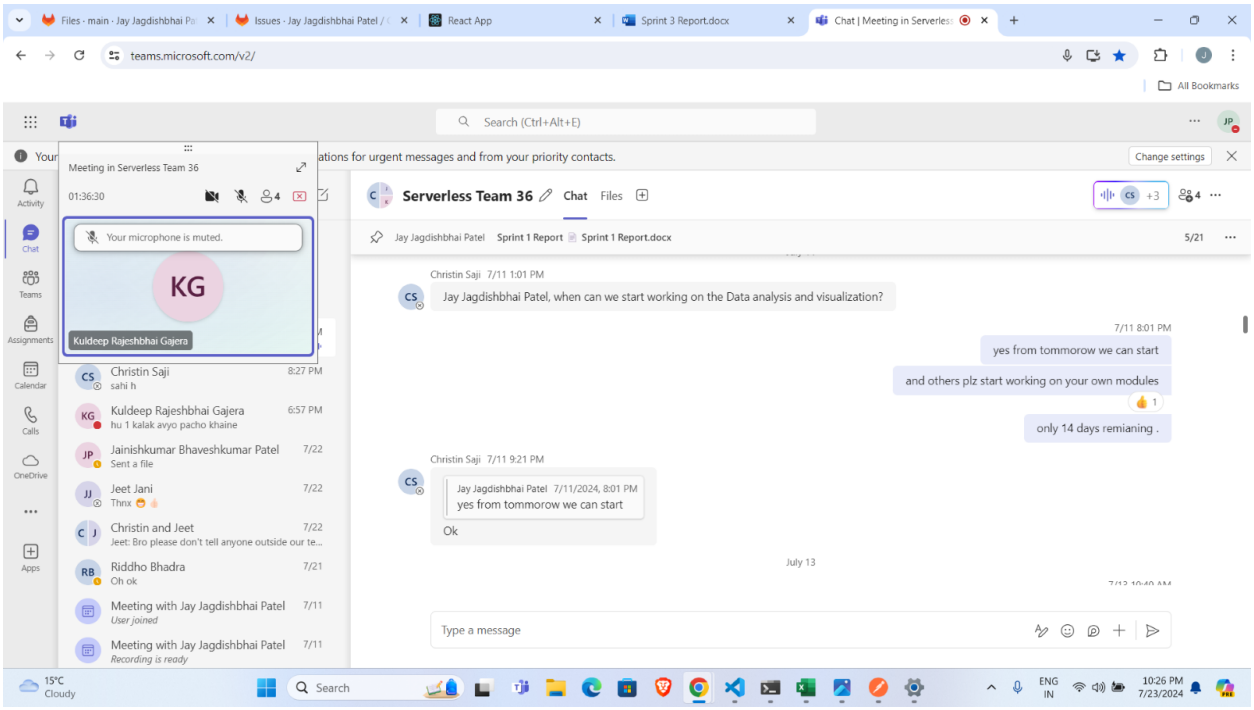

Figure 59: Team members discussing project updates and strategies in a team chat.

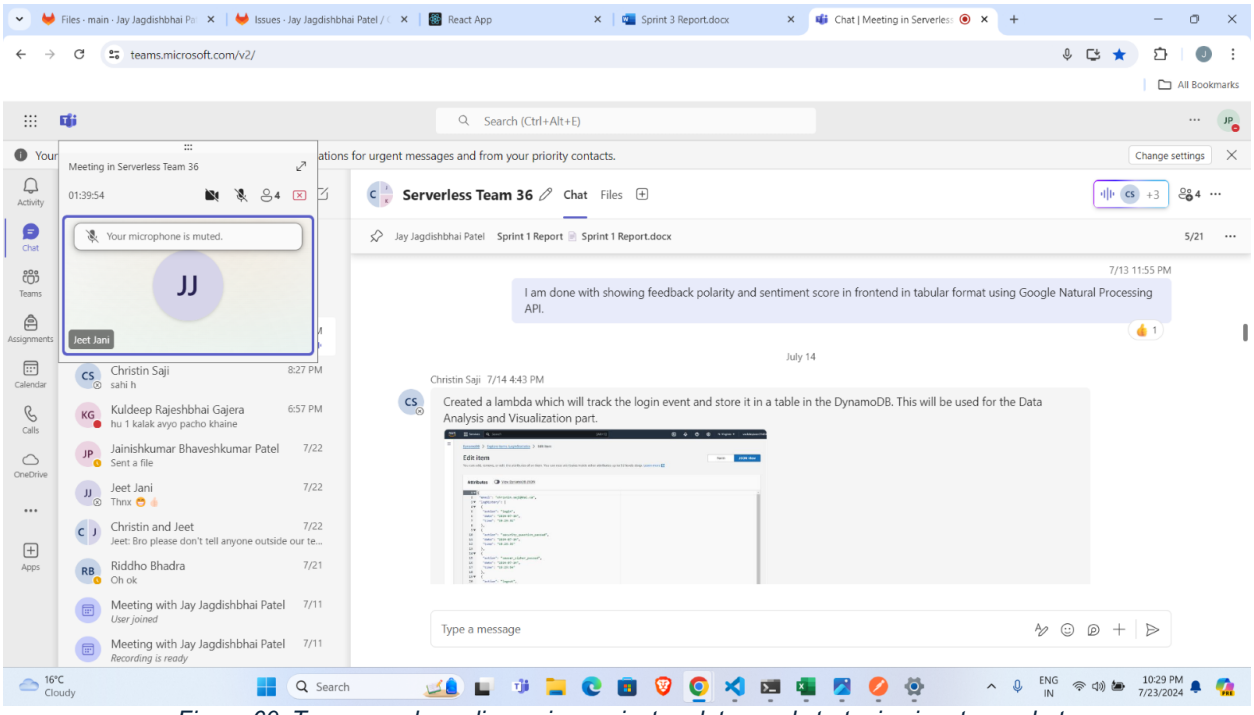

Figure 60: Team members discussing project updates and strategies in a team chat.

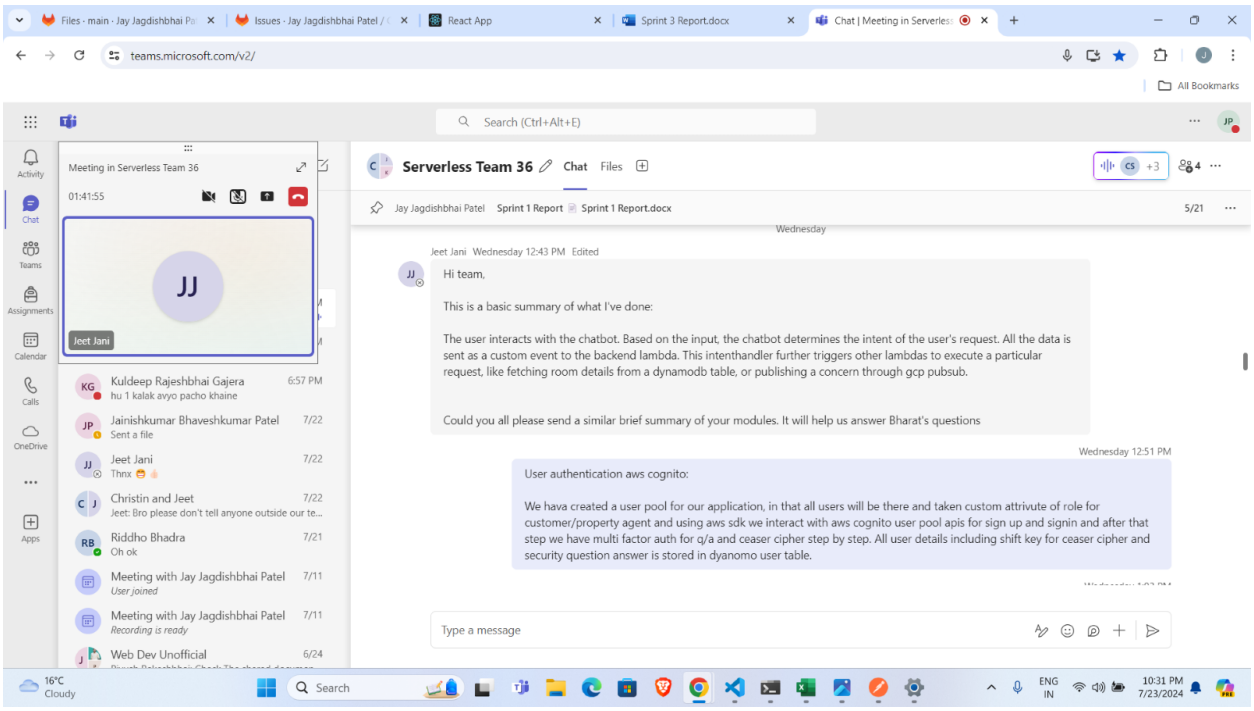

Figure 61: Team members discussing project updates and strategies in a team chat.

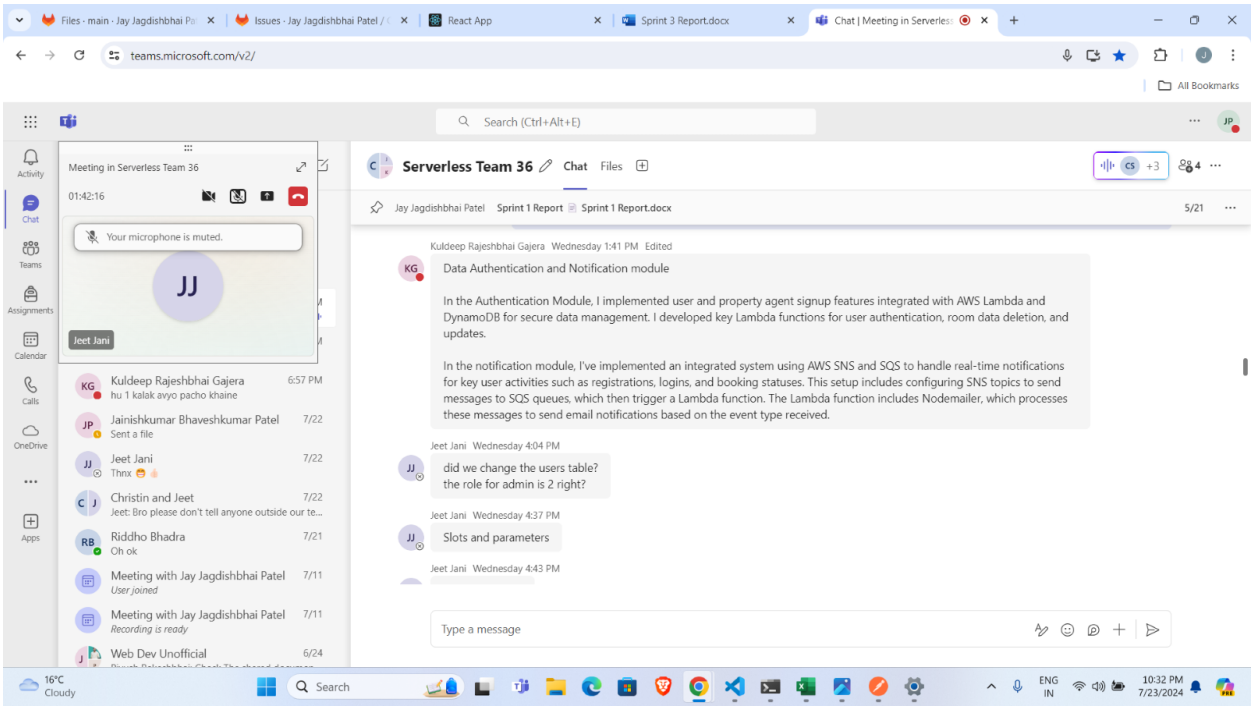

Figure 62: Team members discussing project updates and strategies in a team chat.

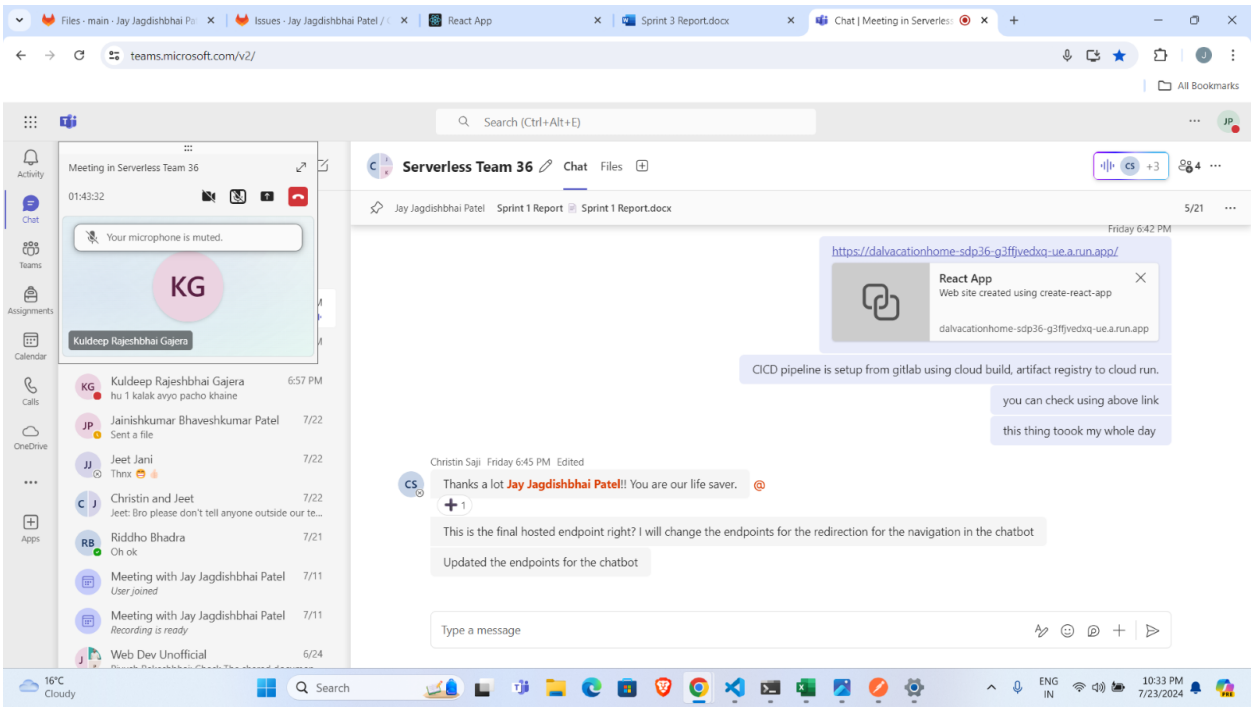

Figure 63: Team members discussing project updates and strategies in a team chat.

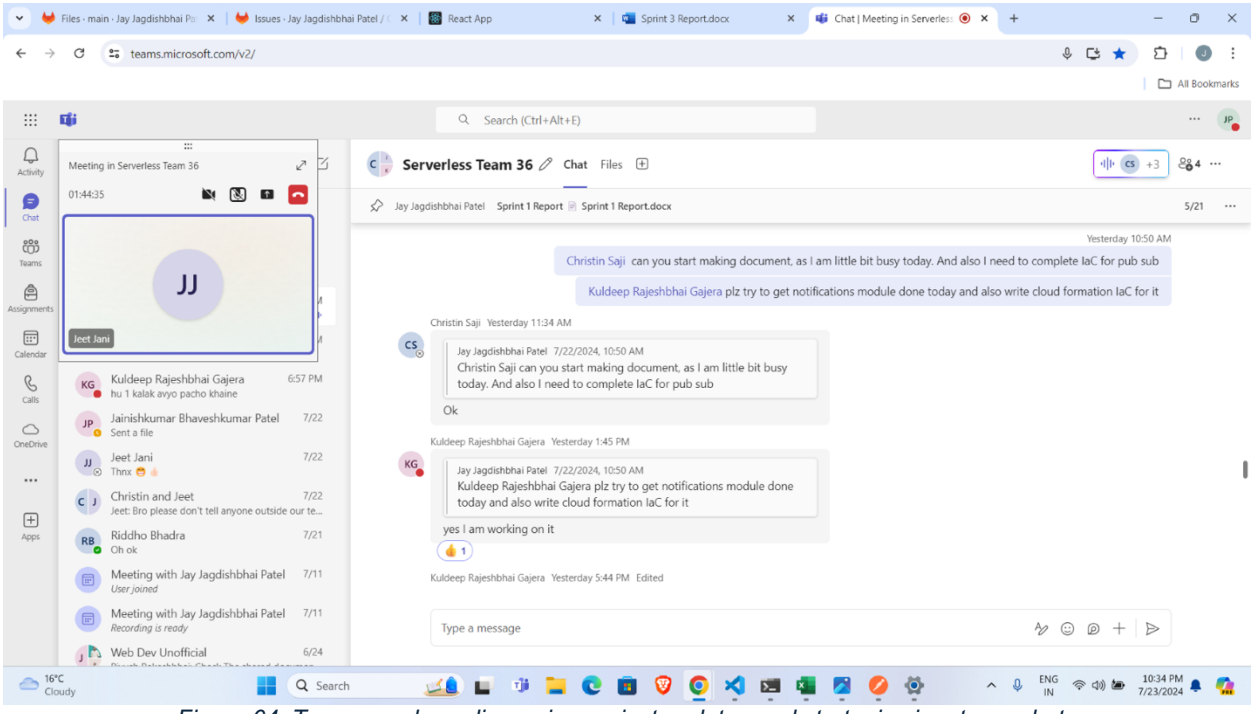

Figure 64: Team members discussing project updates and strategies in a team chat.

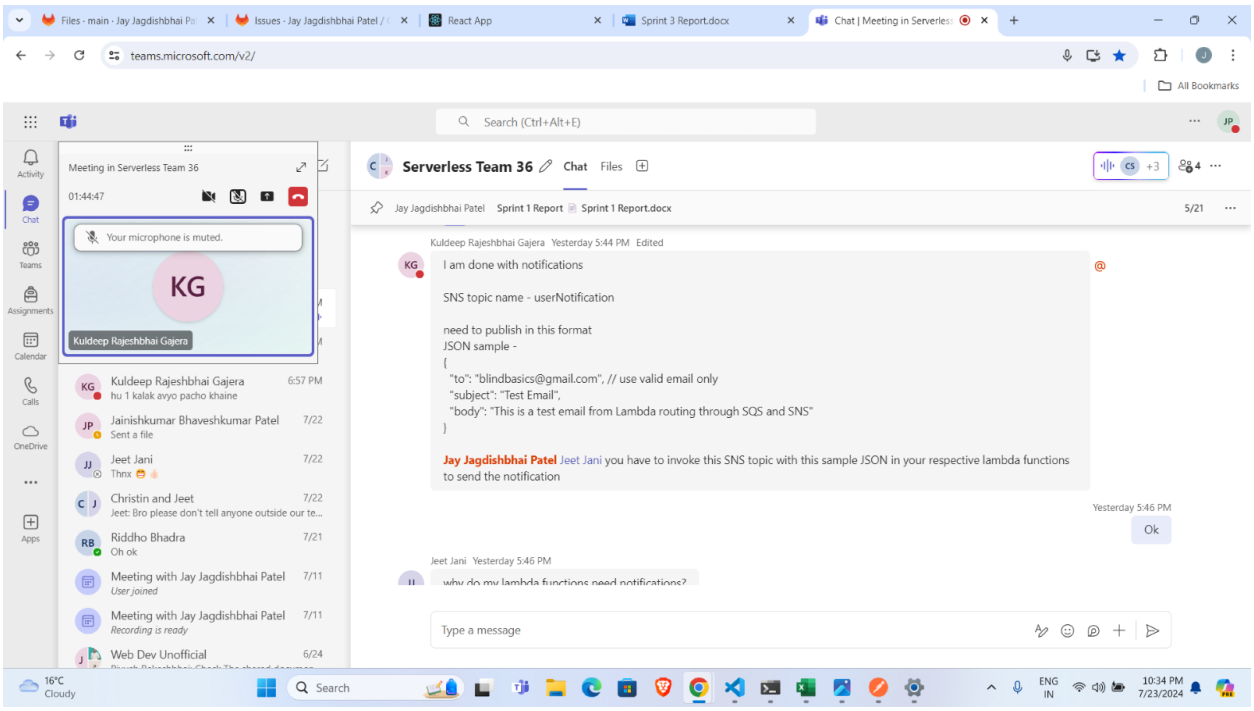

Figure 65: Team members discussing project updates and strategies in a team chat.

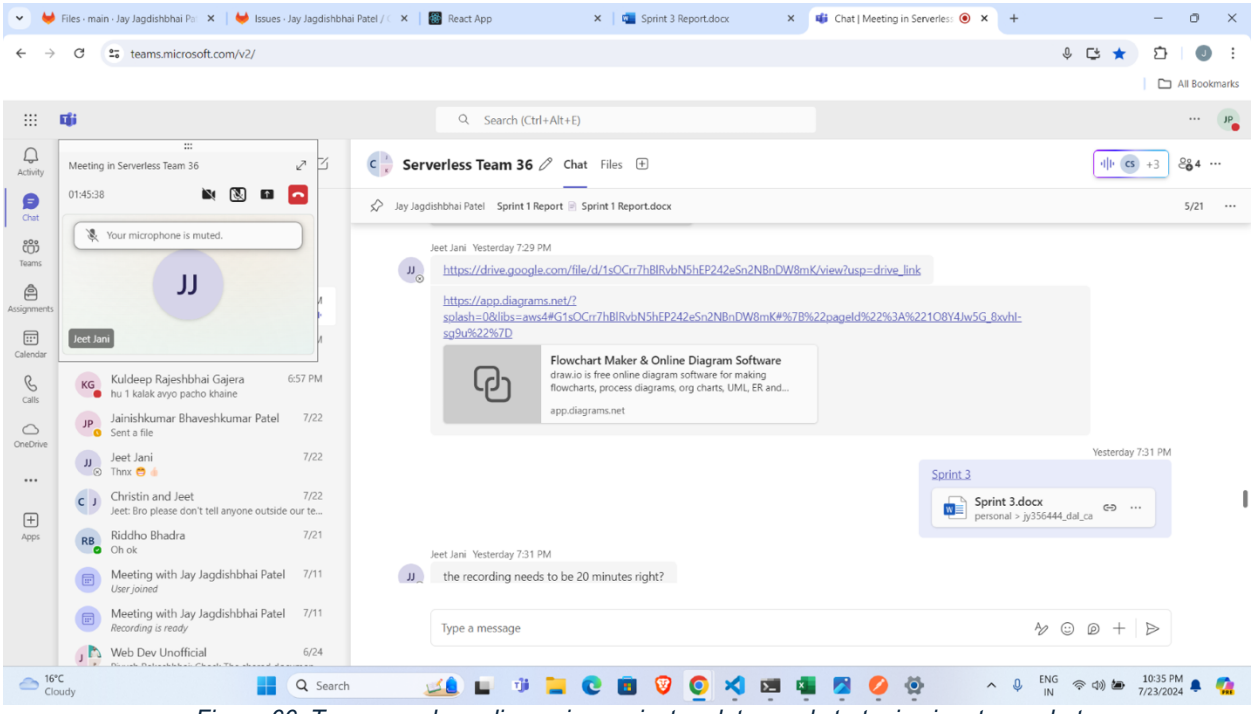

Figure 66: Team members discussing project updates and strategies in a team chat.

Project Management (Sprint 3) Gitlab Board:

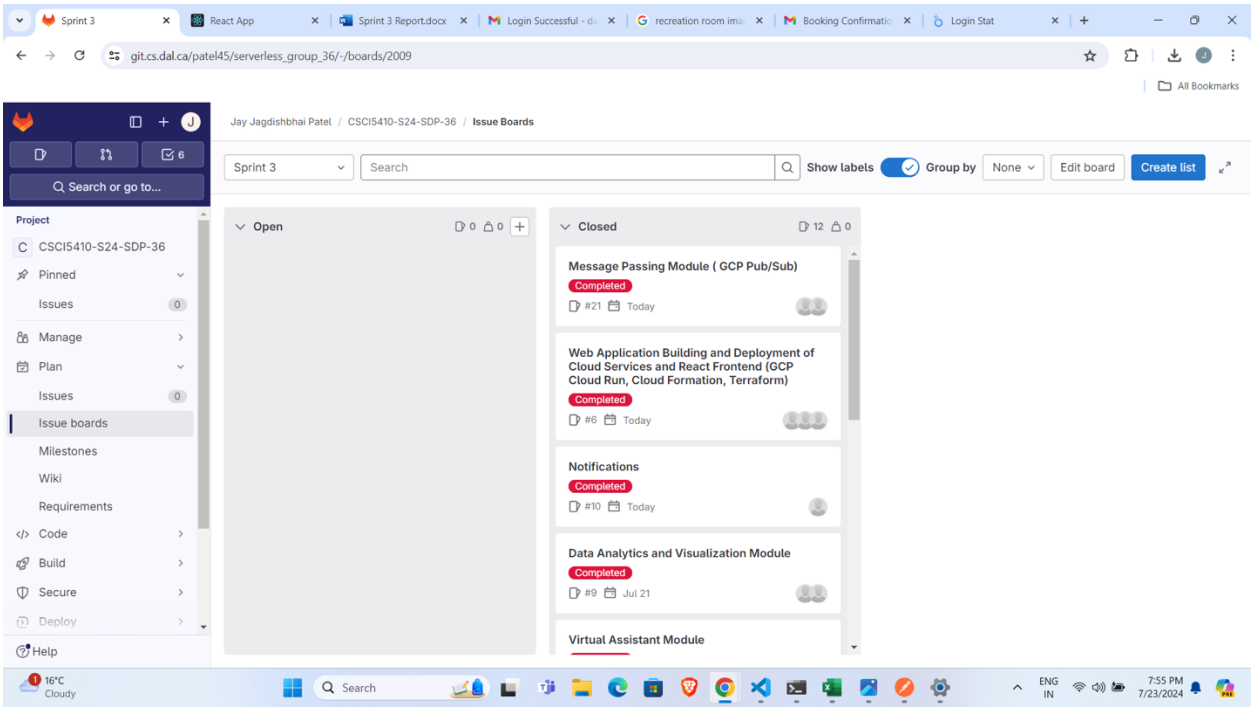

Figure 67: Gitlab Board for Sprint 3 (all issues are closed and completed).

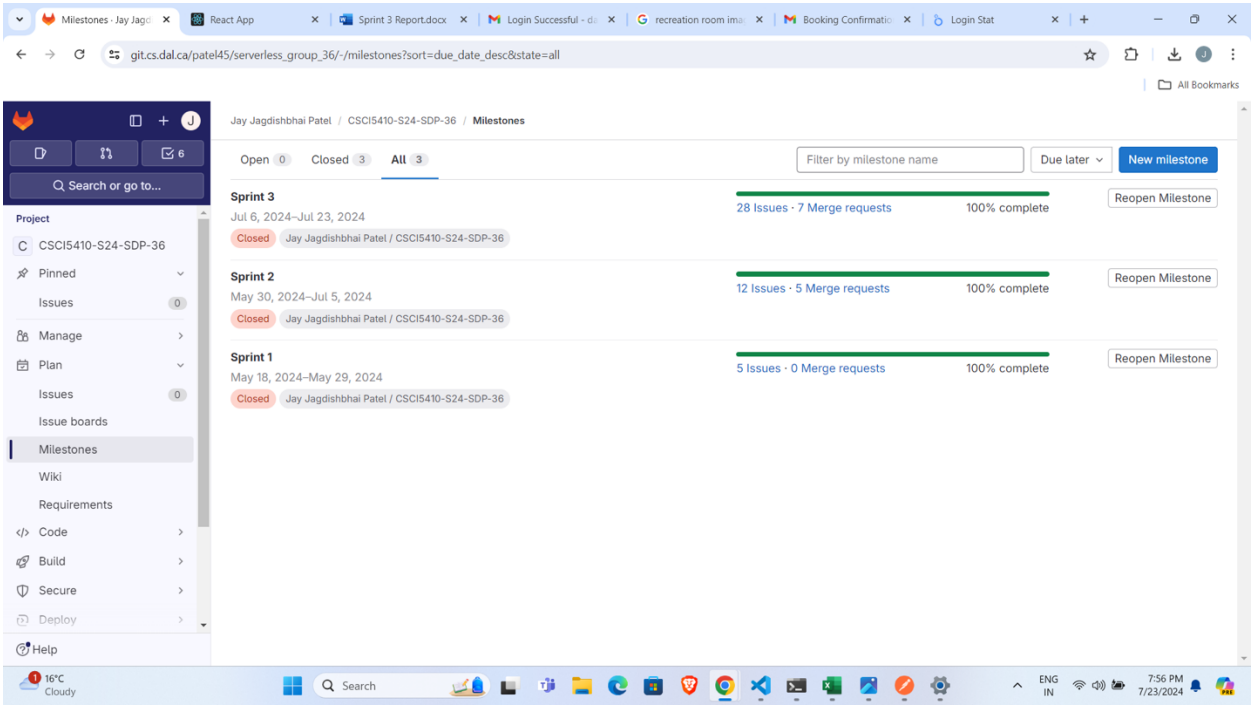

Figure 68: Gitlab Board: All Milestone (Sprints) are completed on time and closed.

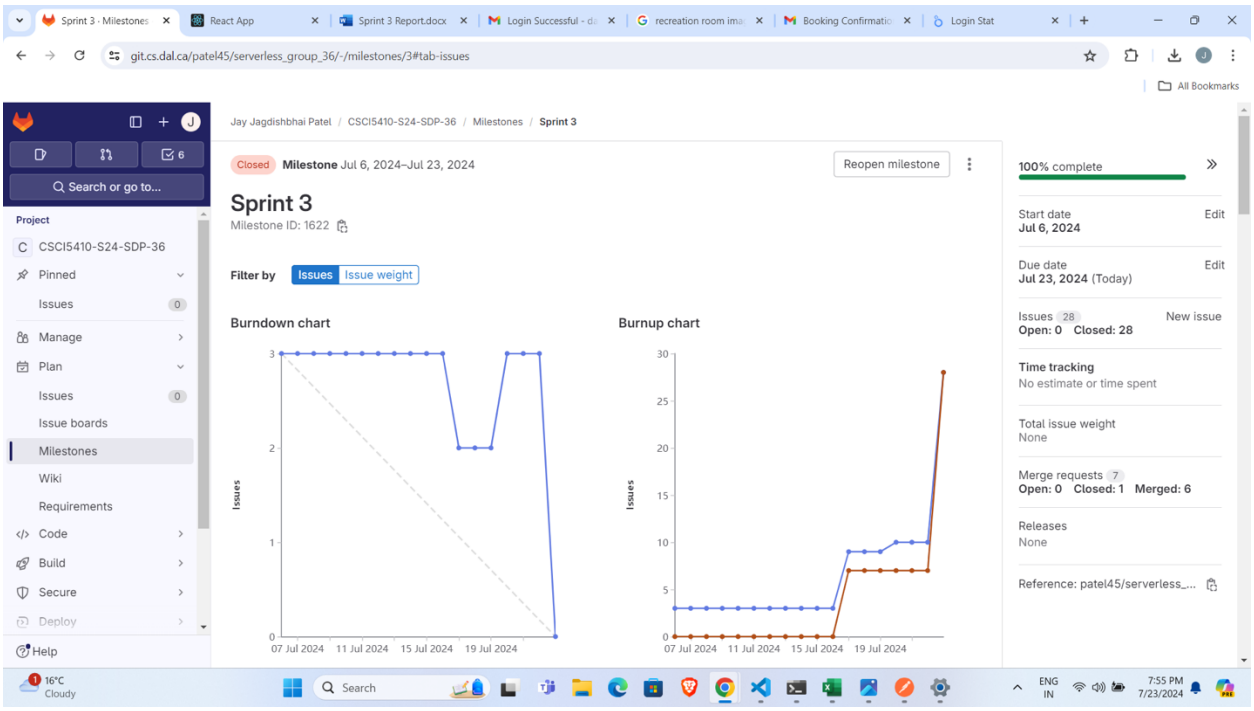

Figure 69: GitLab board and Milestone snapshot displaying Milestone statuses and progress for Sprint 3 of our project.

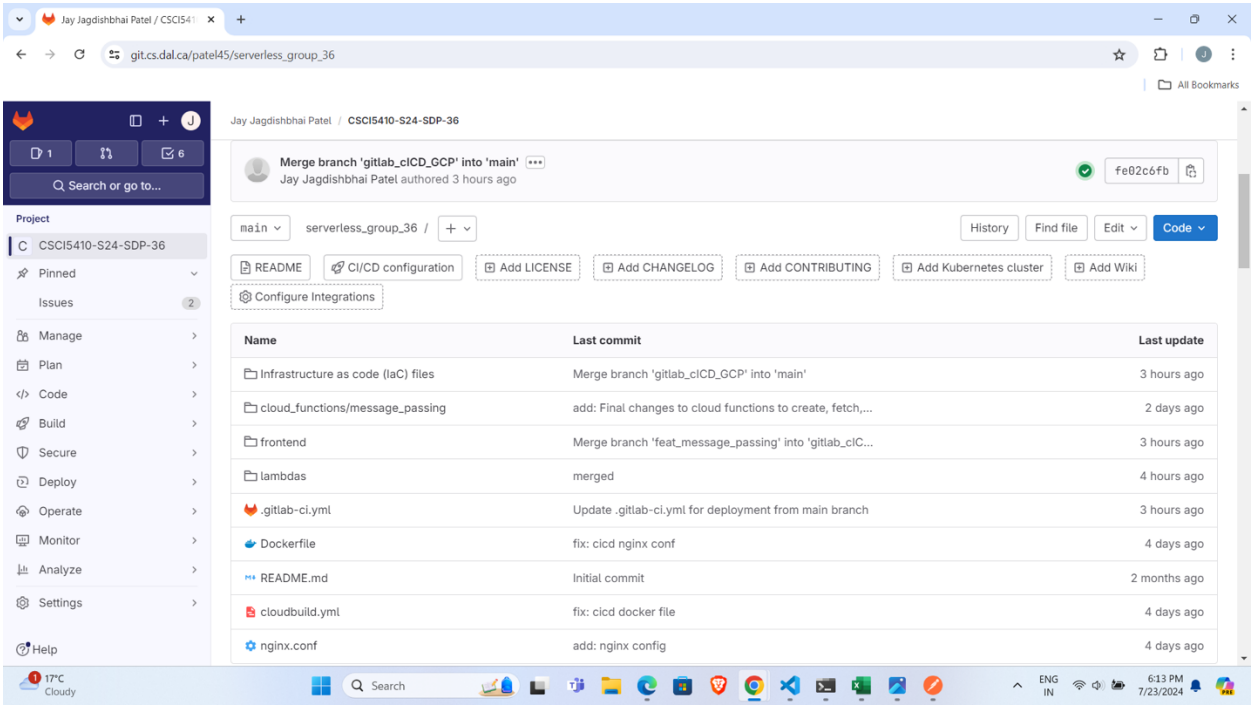

Figure 70: Main branch of our Gitlab Repository and CICD pipeline is working, and application is deployed.

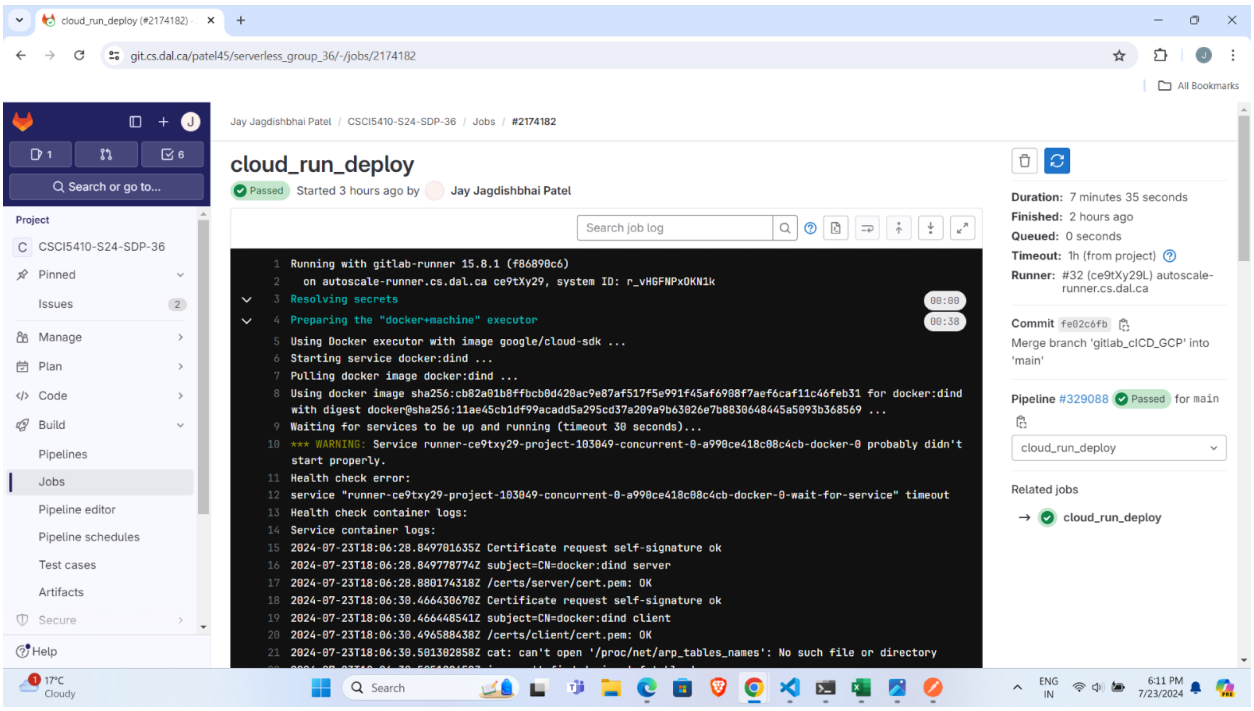

Figure 71: CICD pipeline Job Successed screenshot.

References

[1] "Google Cloud Pub/Sub Documentation," Google Cloud. [Online]. Available: https://cloud.google.com/pubsub/docs/create-topic. [Accessed: July 23, 2024].

[2] "Google Cloud Pub/Sub Documentation," Google Cloud. [Online]. Available: https://cloud.google.com/pubsub/docs/publisher. [Accessed: July 23, 2024].

[3] "Calling Cloud Pub/Sub from Google Cloud Functions," Google Cloud. [Online]. Available: https://cloud.google.com/functions/docs/calling/pubsub#:~:text=If%20you%20are%20deploying%20using%20the%20Google%20Cloud%20console%2C%20you,trigger%20calls%20to%20your%20function. [Accessed: July 23, 2024].

[4] "Extend Cloud Firestore with Cloud Functions," Firebase. [Online]. Available: https://firebase.google.com/docs/firestore/extend-with-functions. [Accessed: July 23, 2024].

[5] A. Prasad, "Building a Real-Time Chat App with React.js and Firebase," freeCodeCamp, May 14, 2020. [Online]. Available: https://www.freecodecamp.org/news/building-a-real-time-chat-app-with-reactjs-and-firebase/. [Accessed: July 23, 2024].

[6] "Amazon Simple Notification Service Developer Guide," AWS. [Online]. Available: https://docs.aws.amazon.com/sns/latest/dg/welcome.html. [Accessed: July 23, 2024].

[7] "Amazon Simple Queue Service Developer Guide," AWS. [Online]. Available: https://docs.aws.amazon.com/AWSSimpleQueueService/latest/SQSDeveloperGuide/welcome.html. [Accessed: July 23, 2024].

[8] "Using Amazon SNS and SQS as Subscribers," AWS. [Online]. Available: https://docs.aws.amazon.com/sns/latest/dg/sns-sqs-as-subscriber.html. [Accessed: July 23, 2024].

[9] A. Gupta, "Build a Serverless Push and SMS Notification Service on AWS," Medium, March 10, 2020. [Online]. Available: https://lo0o0p.medium.com/build-a-serverless-push-and-sms-notification-service-on-aws-f92fdbf7ece3. [Accessed: July 23, 2024].

[10] "Amazon Cognito Identity SDK for JavaScript," npm. [Online]. Available: https://www.npmjs.com/package/amazon-cognito-identity-js. [Accessed: July 23, 2024].

[11] "Analyzing Sentiment," Google Cloud. [Online]. Available: https://cloud.google.com/natural-language/docs/analyzing-sentiment. [Accessed: July 23, 2024].

[12] "Natural Language Basics," Google Cloud. [Online]. Available: https://cloud.google.com/natural-language/docs/basics#interpreting_sentiment_analysis_values. [Accessed: July 23, 2024].

[13] "Node.js Client for Google Cloud Natural Language," npm. [Online]. Available: https://www.npmjs.com/package/@google-cloud/language. [Accessed: July 23, 2024].

[14] "Cognito User Pool Lambda Trigger," AWS. [Online]. Available: https://docs.aws.amazon.com/cognito/latest/developerguide/user-pool-lambda-post-authentication.html. [Accessed: July 23, 2024].

[15] "Batch Update," Google Sheets API. [Online]. Available: https://developers.google.com/sheets/api/guides/batchupdate. [Accessed: July 23, 2024].

[16] "Embedding in Looker Studio," Google Developers. [Online]. Available: https://developers.looker.com/embed/getting-started. [Accessed: July 23, 2024].

[17] "Looker Studio Help," Google Support. [Online]. Available: https://support.google.com/looker-studio/answer/7450249?hl=en#zippy=%2Cin-this-article. [Accessed: July 23, 2024].

[18] "Deploying Builds on Cloud Run," Google Cloud. [Online]. Available: https://cloud.google.com/build/docs/deploying-builds/deploy-cloud-run. [Accessed: July 23, 2024].

[19] "Building Containers," Google Cloud. [Online]. Available: https://cloud.google.com/build/docs/building/build-containers. [Accessed: July 23, 2024].

[20] "Create a GitLab Pipeline to Push to Google Artifact Registry," GitLab Documentation. [Online]. Available: https://docs.gitlab.com/ee/tutorials/create_gitlab_pipeline_push_to_google_artifact_registry/. [Accessed: July 23, 2024].

[21] "Build Your Application," GitLab Documentation. [Online]. Available: https://docs.gitlab.com/ee/topics/build_your_application.html. [Accessed: July 23, 2024].

[22] “Flowchart maker & online diagram software,” *Diagrams.net*. [Online]. Available: https://app.diagrams.net/. [Accessed: July 4, 2024]].

[23] "Thunder Client," Thunder Client. [Online]. Available: https://www.thunderclient.com/. [Accessed: July 1, 2024].