DALHOUSIE UNIVERSITY

Sprint 2 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 1: User Management and Authentication		
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
AWS Cognito	AWS Cognito Documentation	https://docs.aws.amazon.com/cognito/latest/developerguide/what-is-amazon-cognito.html
How to use AWS Cognito Authentication with Auth0	Medium	https://hyprstack.medium.com/reactjs-aws-cognito-authentication-with-auth0-f3f16094fa8
AWS Lambda	AWS Lambda Documentation	https://docs.aws.amazon.com/lambda/latest/dg/welcome.html
How to create and deploy AWS Lambda functions for authentication	DEV Community Blog	https://dev.to/lauracarballo/serverless-authentication-with-aws-lambda-427m
AWS Cognito with ReactJs for Authentication	Medium	https://medium.com/@adi2308/aws-cognito-with-reactjs-for-authentication-c8916b873ccb
Module 2: Virtual Assistant		
Research Area	Source	URL
Creating a Dialogflow chatbot	Medium	https://medium.com/@dipan.saha/chatbot-development-made-easy-creating-a-simple-bot-with-dialogflow-ade69caac37d
Deciding between Dialogflow ES and CX	Google Cloud Dialogflow	https://cloud.google.com/dialogflow/cx/docs/how/migrate#:~:text=Dialogflow%20CX%20agents%20provide

		%20you,Dialogflow%20ES%20to%20Dialogflow%20CX.
Defining intents and bot responses	Google Cloud Dialogflow	https://cloud.google.com/dialogflow/es/docs/intents-overview
Integrating the Dialogflow chatbot in the frontend code	DEV Community	https://dev.to/ketan_patil/chatbot-for-your-website-using-dialogflow-fc7
Adding clickable navigation button and customizing the ChatBot	Google Cloud Dialogflow	https://cloud.google.com/dialogflow/es/docs/integrations/dialogflow-messenger
Configuring web hook to connect GCP service to AWS lambda	Google Cloud Dialogflow	https://cloud.google.com/dialogflow/es/docs/fulfillment-webhook
Passing user input and intent details as event in lambda	AWS Documentation Amazon Lex V1	https://docs.aws.amazon.com/lex/latest/dg/lambda-input-response-format.html
Fetching user details from DynamoDB and sending it back to the frontend chatbot interface	AWS Documentation Amazon API Gateway	https://docs.aws.amazon.com/apigateway/latest/developerguide/http-api-dynamo-db.html
Storing and retrieving access tokens from local storage	The New Stack	https://thenewstack.io/best-practices-for-storing-access-tokens-in-the-browser/

How we have Implemented:

Module 1: User Management & Authentication Module

Going over various AWS Docs and research articles, we figured out an approach about how to implement Authentication and Authorization in our application.

First, we began with setting up AWS Cognito by setting up User Pool and configuring it to include Custom Parameters like name, role which Cognito should accept while registering a user. Then we set up Email Verification while registering a new user [2].

We have taken name and role as custom attribute. Role 0 (Customer role) and Role 1 (Property Agent).

Second, out of many libraries available to interact with AWS Cognito programmatically, we have selected the amazon-cognito-identity-js npm library, which provides different APIs to interact with AWS Cognito programmatically [1].

Next, we have configured 3-factor Authentication, after user enter Email and Password, he will be redirected to next page where he will be prompted to enter security answer for the question which he selected at the Signup time, and after this step is successfully completed, he will be redirected to the Ceaser Cipher puzzle page. Where user must De-Cipher the Cipher text based on the Shift Key provided at the time of Signup [5][12].

After all these steps are completed, then the user will be able to access the application and an authentication Token will be generated for the user and stored in the local storage of the user's browser for further requests [13].

When successful signup happens, we store the data in the DynamoDB by using Lambda function. And made Lambda function for Security Question/Answer verification and Ceaser Cipher multi-factor authentication [6].

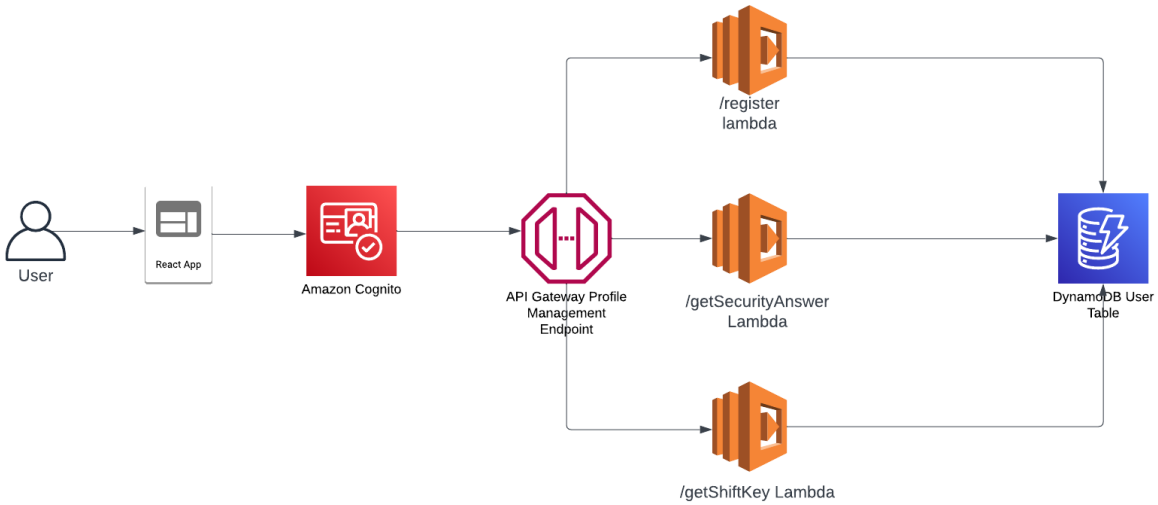

Figure 1: Architecture Diagram of User Management and Authentication Module

Module 2: Virtual Assistance Module

The chatbot was created using Dialogflow ES and integrated into the React frontend using the df-messenger tag, provided by Google as part of the Dialogflow Messenger integration, with parameters for intent, chat title, agent ID, and language code [10]. An API was developed to invoke a Lambda function, which was set as the webhook URL for the Dialogflow bot. This Lambda function acts as an Intent Handler, determining the type of intent and triggering other Lambda functions based on the triggered intent [12].

The chatbot offers comprehensive navigation within the site, achieved by creating multiple navigation intents. Each intent includes customized responses and custom payloads that provide clickable navigation buttons, enhancing user interaction and ease of use. When a user inputs a message, the Dialogflow bot analyzes the text to find the closest matching intent. Upon identification, the corresponding intent is triggered, and the user receives a tailored response, often including a button that redirects to the relevant page. This seamless integration ensures users can efficiently navigate through various site sections, such as the home page, booking details, and authentication page, by merely interacting with the chatbot [9][8].

For room booking details, the Dialogflow bot passes the booking reference code which is used by the IntentHandler Lambda to fetch the corresponding details from the Bookings DynamoDB table [7]. The ticket generation feature involves logic to create a ticket and store it in a DynamoDB table and management of customer concerns [11].

To differentiate between registered users and guests, access tokens are stored in the local browser storage. These tokens are sent to the IntentHandler Lambda, which grants users the appropriate features based on their status. Registered users have access to more features, such as detailed room information and customer support, while guests can use the basic navigation and information features [13].

The implementation involved using AWS services, including Lambda for serverless function execution and DynamoDB for storing and retrieving data. GCP services, specifically Dialogflow ES, were used to build and manage the chatbot [7].

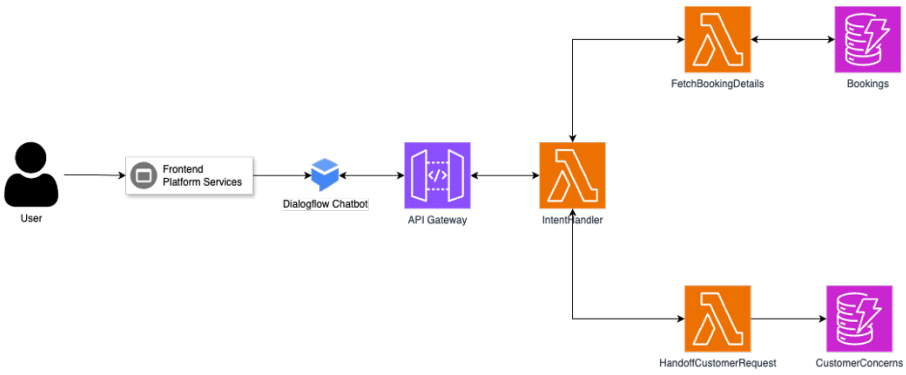

Figure 2: Architecture Diagram of Virtual Assistant Module

Individual Contribution:

Christin Saji:

1. **Virtual Assistant Module with Google Dialogflow:**
    - Developed and integrated the Virtual Assistant Module using Google Dialogflow ES.
    - Created multiple intents for navigation with appropriate responses and custom payloads, enhancing user interaction and site navigation.
    - Configured the chatbot to handle user queries effectively, providing accurate and timely responses.
2. **Frontend Development:**
    - Integrated the Dialogflow Messenger into the React frontend, ensuring seamless communication between users and the chatbot.
    - Customized the styling of the Dialogflow Messenger to align with the overall aesthetic of the application, improving user experience.
    - Enhanced various aspects of the user interface for better usability and visual appeal.
3. **GitLab Board Management:**
    - Actively managed the progress of the Virtual Assistant 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.
4. **Documentation**:
    - Maintained comprehensive meeting logs for Sprint 2, capturing agendas, outcomes of discussions, and links to meeting recordings.

Jay Jagdishbhai Patel:

1. **User Management & Authentication Module with AWS Cognito:**
    - Contributed to developing and integrating the User Management & Authentication Module using AWS Cognito.
    - Implemented user sign-up, sign-in, and multi-factor authentication in frontend.
    - Managed user sessions and secure access to application features.
2. **Lambda Functions:**
    - Created seven Lambda functions to manage room operations and user interactions:
        - **create-room:** Enables Property Agent to create new rooms.

        - **get-all-rooms:** Retrieves a list of all available rooms.
        - **get-room-by-id:** Fetches details of a specific room by its ID.
        - **book-room:** Allows Customer to book a room.
        - **get-booking-by-id:** Retrieves booking details using the booking ID.
        - **get-bookings-by-user:** Lists all bookings made by a specific Customer.
        - **give-feedback-for-room:** Allows Customers to provide feedback for room after their stay ends.
3. **Frontend Development:**
    - Contributed to designing and developing the room management frontend using React.
    - Integrated the Lambda API endpoints with the frontend for real-time data interaction.
4. **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.

Jeet Jani

1. **Virtual Assistant Module:**
    - Created a 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.
    - Integrated this chatbot into the frontend using text-based web demo integration.
    - Created Lambda functions to correctly handle the backend logic for various user requests.
    - Created 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. **Lambda Functions:**
    - Created 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.

- Created 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.
- Created HandoffCustomerSupport function to handle the Customer Support Request Intent. It stores the issue into the DynamoDb table, CustomerConcerns, and notifies the user that the query has been submitted.

3. **Documentation:**
   - Designed architecture diagrams for the Virtual Assistant module and the entire project, including plans for future service integrations.

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.
3. **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:

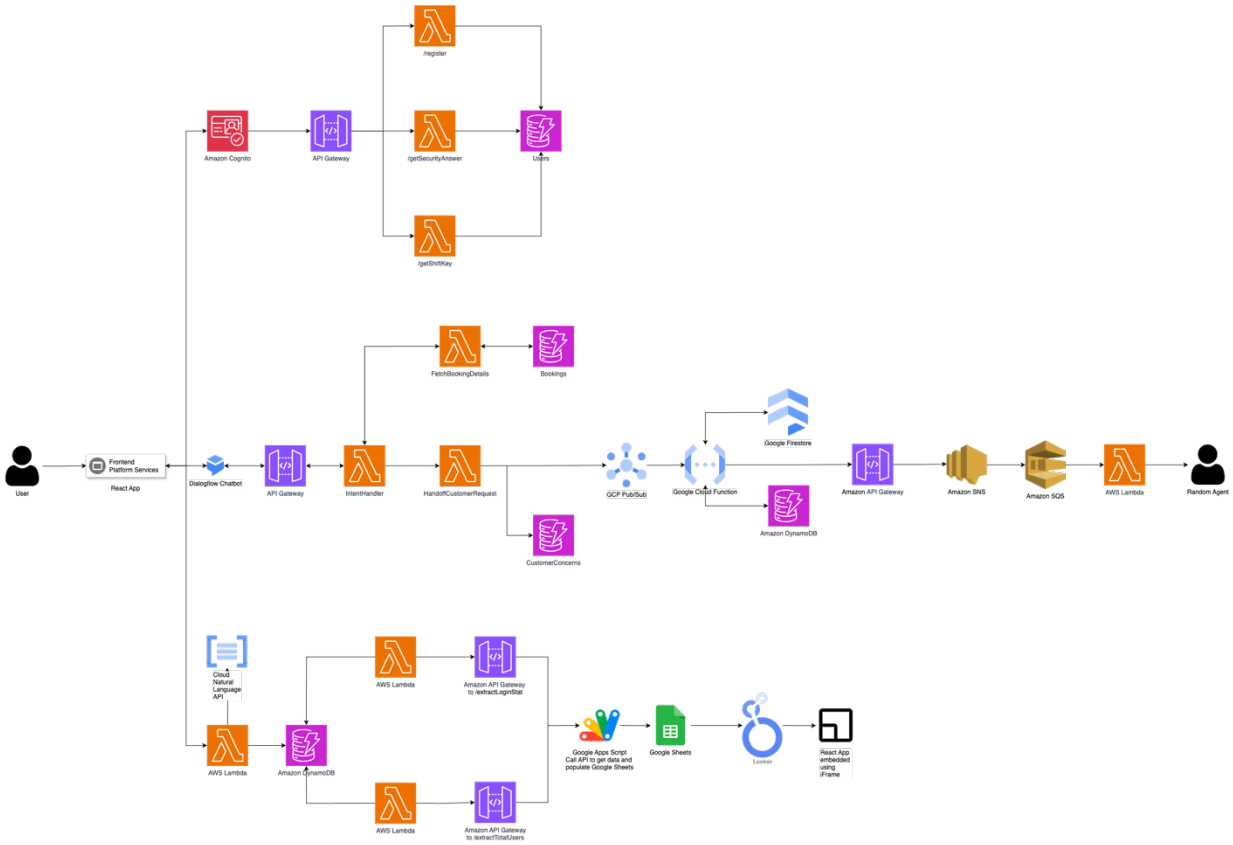

Figure 3: System Architecture Diagram of Application

Pseudocode/ Algorithm for Important Modules' logic:

Module 1: User Management & Authentication Module

Lambda Function: Fetching Security Q/A from DynamoDB

Pseudocode:

```
FUNCTION lambda_handler(event):
    Extract email from event

    IF email is not provided:
        RETURN 400 status code with error message "Email is required"

    Define params for DynamoDB query with TableName "Users" and Key as
email

    TRY:
        Fetch data from DynamoDB using params
        IF data does not contain Item:
            RETURN 404 status code with error message "User not found"

        RETURN 200 status code with security answer from data.Item
    CATCH error:
        RETURN 500 status code with error message "Could not retrieve user"

END FUNCTION lambda_handler
```

Algorithm:

1. Parse the Incoming Event
2. Extract the email from the event.
3. Validate Email
4. If the email is not provided:
    a. Return a 400-status code with an error message "Email is required".
5. Prepare DynamoDB Query Parameters
6. Define the parameters for querying DynamoDB:
    a. Set TableName to "Users".
    b. Set Key to the extracted email.
7. Fetch Data from DynamoDB

8. Use a try-catch block to handle potential errors:
    a. Try Block:
        i. Fetch data from DynamoDB using the defined parameters.
    b. Catch Block:
        i. If an error occurs during the fetch operation:
            1. Return a 500-status code with an error message "Could not retrieve user".
9. Handle Query Response
10. If data does not contain an Item:
    a. Return a 404-status code with an error message "User not found".
11. If data contains an Item:
    a. Return a 200 status code with the security answer from data.Item.
12. Return the Response
13. Based on the outcome of the DynamoDB fetch operation, return the appropriate HTTP status code and message:
    a. 400 if email is not provided.
    b. 404 if user is not found.
    c. 200 with the security answer if user is found.
    d. 500 if there is an error fetching data from DynamoDB.

Lambda Function for getting Shift Key associated with user in DynamoDB

Pseudocode:

```
FUNCTION lambda_handler(event):
  Extract email from event

  IF email is not provided:
    RETURN 400 status code with error message "Email is required"

  Define params for DynamoDB query with TableName "Users" and Key as email

  TRY:
    Fetch data from DynamoDB using params
    IF data does not contain Item:
      RETURN 404 status code with error message "User not found"

    RETURN 200 status code with shiftKey from data.Item
  CATCH error:
```

```
        RETURN 500 status code with error message "Could not retrieve user"

    END FUNCTION lambda_handler
```

Algorithm:

1. Parse the Incoming Event
2. Extract the email from the event.
3. Validate Email
4. If the email is not provided:
    a. Return a 400 status code with an error message "Email is required".
5. Prepare DynamoDB Query Parameters
6. Define the parameters for querying DynamoDB:
    a. Set TableName to "Users".
    b. Set Key to the extracted email.
7. Fetch Data from DynamoDB
8. Use a try-catch block to handle potential errors:
    a. Try Block:
        i. Fetch data from DynamoDB using the defined parameters.
    b. Catch Block:
        i. If an error occurs during the fetch operation:
            1. Return a 500 status code with an error message "Could not retrieve user".
9. Handle Query Response
10. If data does not contain an Item:
    a. Return a 404 status code with an error message "User not found".
11. If data contains an Item:
    a. Return a 200 status code with the shiftKey from data.Item.
12. Return the Response
13. Based on the outcome of the DynamoDB fetch operation, return the appropriate HTTP status code and message:
    a. 400 if email is not provided.
    b. 404 if user is not found.
    c. 200 with the shiftKey if user is found.
    d. 500 if there is an error fetching data from DynamoDB.

Lambda Function for Storing User Details:

Pseudocode:

```
FUNCTION lambda_handler(event):
    Extract email, name, role, securityAnswer, and shiftKey from event

    IF email, name, role, securityAnswer, or shiftKey is not provided:
        RETURN 400 status code with error message "Missing required fields"

    Define params for DynamoDB put operation with TableName "Users" and Item containing email, name, role, securityAnswer, and shiftKey

    TRY:
        Insert item into DynamoDB using params
        RETURN 200 status code with message "User registered successfully"
    CATCH error:
        RETURN 500 status code with error message "Could not register user"

END FUNCTION lambda_handler
```

Algorithm:

1. Parse the Incoming Event
2. Extract email, name, role, securityAnswer, and shiftKey from the event.
3. Validate Required Fields
4. If email, name, role, securityAnswer, or shiftKey is not provided:
    a. Return a 400 status code with an error message "Missing required fields".
5. Prepare DynamoDB Put Parameters
6. Define the parameters for the DynamoDB put operation:
    a. Set TableName to "Users".
    b. Set Item to an object containing email, name, role, securityAnswer, and shiftKey.
7. Insert Item into DynamoDB
8. Use a try-catch block to handle potential errors:
    a. Try Block:
        i. Insert the item into DynamoDB using the defined parameters.
        ii. Return a 200-status code with a message "User registered successfully".

    b. Catch Block:
        i. If an error occurs during the put operation:
            1. Return a 500-status code with an error message "Could not register user".
9. Return the Response
10. Based on the outcome of the DynamoDB put operation, return the appropriate HTTP status code and message:
    a. 400 if any required fields are missing.
    b. 200 if the user is registered successfully.
    c. 500 if there is an error registering the user.

Module 2: Virtual Assistant

IntentHandler

Pseudocode

```
FUNCTION lambda_handler(event):
    TRY:
        IF event.body is a string:
            Parse event.body to JSON
        ELSE:
            Set body to event.body

        Extract parameters and intentName from body.queryResult
        Extract token using optional chaining from
body.originalDetectIntentRequest.payload.data.userRole

        Log event body, parameters, intentName, and token

        IF intentName is 'Booking Info Intent' AND token is not null:
            Extract bookingId from parameters.BookingReferenceCode
            Log bookingId
            Call invokeLambda with 'FetchBookingDetails' and { bookingId }
            Log room number, duration of stay, and message to user from response
        ELSE IF intentName is 'Customer Support Request Intent' AND token is not
null:
            Extract issue from parameters.Issue or set to empty string
            Extract bookingReferenceCode from parameters.BookingReferenceCode or
set to empty string
            Call invokeLambda with 'HandoffSupportRequest' and { Issue: issue,
BookingReferenceCode: bookingReferenceCode }
```

Log response from HandoffSupportRequest

ELSE:

Log lack of necessary permissions and token

Set responseMessage to 'Sorry, you do not have the necessary permissions for this request.'

RETURN createResponse with responseMessage.fulfillmentText

CATCH error:

Log error

RETURN 500 status code with error message 'There was an error processing your request. Please try again later.'

END FUNCTION lambda_handler

FUNCTION invokeLambda(functionName, parameters):

Define params with FunctionName as functionName, InvocationType as 'RequestResponse', and Payload as JSON string of parameters

TRY:

Invoke Lambda with params

Parse data.Payload to JSON

RETURN responsePayload

CATCH error:

Log error

RETURN error message 'There was an error processing your request. Please try again later.'

END FUNCTION invokeLambda

FUNCTION createResponse(fulfillmentText):

RETURN 200 status code with body containing fulfillmentText

END FUNCTION createResponse

Algorithm

1. Initialize Variables
2. Parse Request Body
   a. Check if `event.body` is a string.
   b. If true, parse `event.body` to JSON.
   c. Else, set `body` to `event.body`.
3. Extract Parameters and Intent Name
   a. Extract `parameters` from `body.queryResult.parameters`.
   b. Extract `intentName` from `body.queryResult.intent.displayName`.
4. Extract Token
   a. Extract `token` using optional chaining from `body.originalDetectIntentRequest.payload.data.userRole`.
5. Log Request Details
   a. Log `event body`.
   b. Log `parameters`.
   c. Log `intentName`.
   d. Log `token`.
6. Handle Intent
   a. If `intentName` is 'Booking Info Intent' AND `token` is not null:
      i. Extract `bookingId` from `parameters.BookingReferenceCode`.

ii. Log `bookingId`.

iii. Call `invokeLambda` with 'FetchBookingDetails' and `{ bookingId }`.

iv. Log `room number`, `duration of stay`, and `message to user` from response.

b. Else if `intentName` is 'Customer Support Request Intent' AND `token` is not null:

i. Extract `issue` from `parameters.Issue` or set to an empty string.

ii. Extract `bookingReferenceCode` from `parameters.BookingReferenceCode` or set to an empty string.

iii. Call `invokeLambda` with 'HandoffSupportRequest' and `{ Issue: issue, BookingReferenceCode: bookingReferenceCode }`.

iv. Log `response from HandoffSupportRequest`.

c. Else:

i. Log lack of necessary permissions and `token`.

ii. Set `responseMessage` to 'Sorry, you do not have the necessary permissions for this request.'.

7. Return Response

a. Return `createResponse` with `responseMessage.fulfillmentText`.

8. Handle Errors

a. In the catch block:

i. Log error.

ii. Return 500 status code with error message 'There was an error processing your request. Please try again later.'.

9. Function invokeLambda

a. Define `params` with:

i. `FunctionName` set to `functionName`.

ii. `InvocationType` set to 'RequestResponse'.

iii. `Payload` set to JSON string of `parameters`.

b. Try:

i. Invoke Lambda with `params`.

ii. Parse `data.Payload` to JSON.

iii. Return `responsePayload`.

c. Catch error:

i. Log error.

ii. Return error message 'There was an error processing your request. Please try again later.'.

10. Function createResponse

a. Return 200 status code with body containing `fulfillmentText`.

FetchBookingDetails

Pseudocode

```
FUNCTION lambda_handler(event):
    TRY:
        Extract parameters from event
        Extract bookingId from parameters

        Log bookingId received

        Call fetchBookingDetails with bookingId

        IF bookingDetails is not null:
            Extract roomNumber from bookingDetails
            Convert startDate and endDate from bookingDetails to Date objects
            Calculate duration as the number of days between startDate and endDate

            Log booking found
```

Log room number and duration

Create responseMessage with room number and duration

Log responseMessage

RETURN roomNumber, duration, and fulfillmentText as responseMessage

ELSE:

Create responseMessage as 'No such booking found.'

Log booking not found

Log responseMessage

RETURN fulfillmentText as responseMessage

CATCH error:

Log error

RETURN fulfillmentText as error message 'There was an error processing your booking details request. Please try again later.'

END FUNCTION lambda_handler

FUNCTION fetchBookingDetails(bookingId):

Define params with TableName as 'Bookings' and Key containing bookingId

TRY:

Fetch data from DynamoDB with params

Log fetched data

RETURN data.Item

CATCH error:

Log error

RETURN null

END FUNCTION fetchBookingDetails

Algorithm

1. Initialize Variables
2. Extract Parameters

   a. Extract parameters from event.

   b. Extract bookingId from parameters.
3. Log Booking ID

   a. Log bookingId received.
4. Fetch Booking Details

   a. Call fetchBookingDetails with bookingId.

   b. If bookingDetails is not null:

      i. Extract roomNumber from bookingDetails.

      ii. Convert startDate and endDate from bookingDetails to Date objects.

      iii. Calculate duration as the number of days between startDate and endDate plus one.

      iv. Log booking found.

      v. Log room number and duration.

      vi. Create responseMessage with room number and duration. vii. Log responseMessage.

      viii. Return roomNumber, duration, and fulfillmentText as responseMessage.

   c. Else:

      i. Create responseMessage as 'No such booking found.'

ii. Log booking not found.

iii. Log responseMessage.

iv. Return fulfillmentText as responseMessage.

5. Handle Errors

a. In the catch block:

i. Log error.

ii. Return fulfillmentText as error message 'There was an error processing your booking details request. Please try again later.'

6. Define Parameters
a. Define params with TableName as 'Bookings' and Key containing bookingId.

7. Fetch Data

a. Try:

i. Fetch data from DynamoDB with params.

ii. Log fetched data.

iii. Return data.Item.

b. Catch error:

i. Log error.

ii. Return null.

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.'.

Test Cases:

Lambda Test Create Room Event:

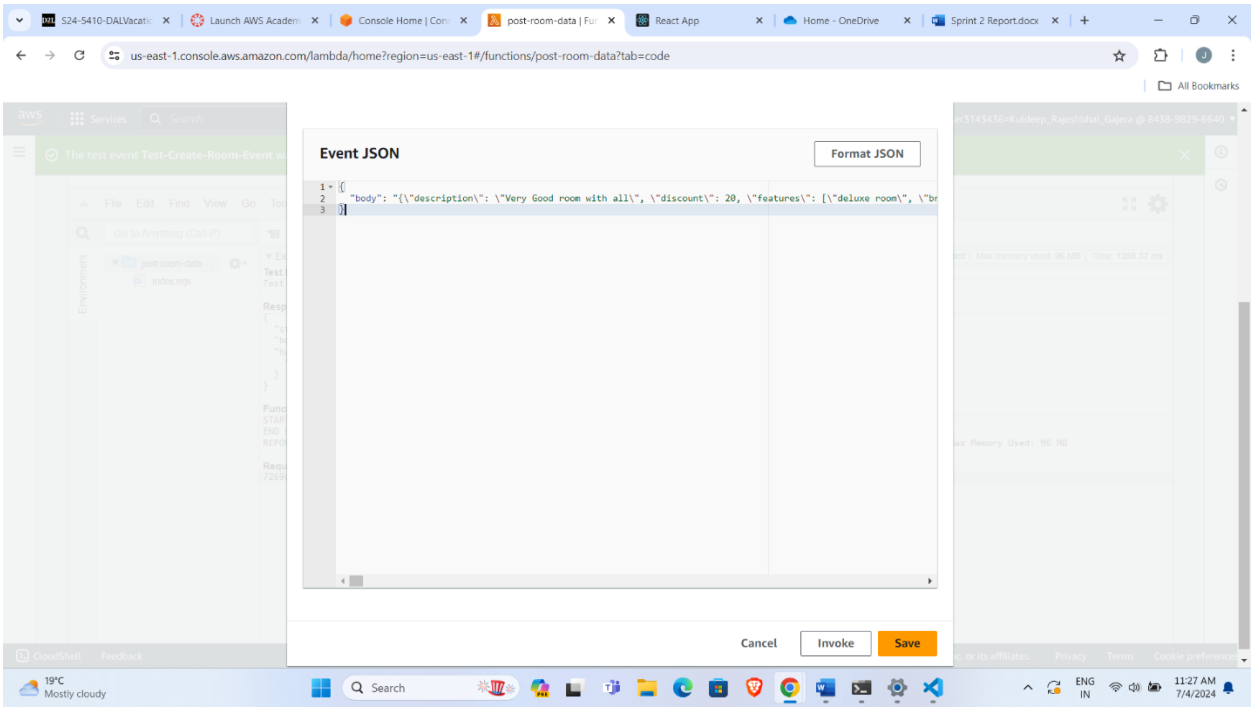

Figure 4: Setup and execution of the Create Room test in AWS Lambda.

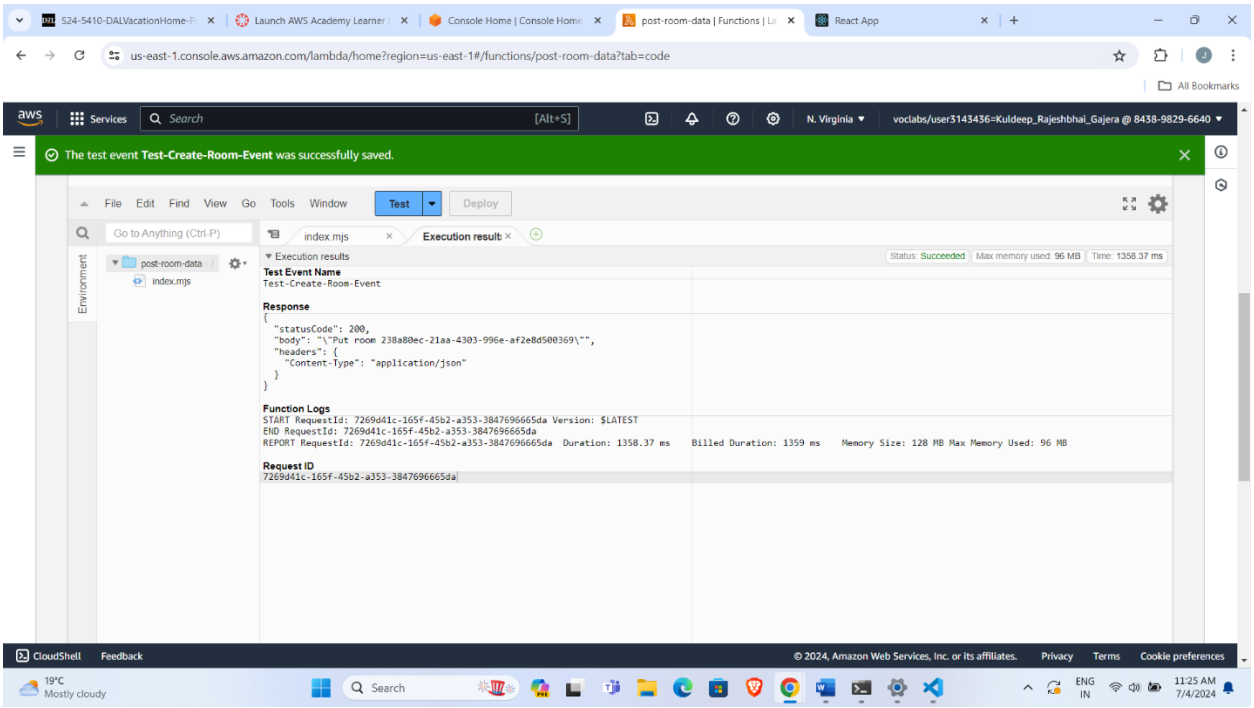

Figure 5: CloudWatch log showing the results of the Create Room test execution.

Lambda Test Get Room by ID Event:

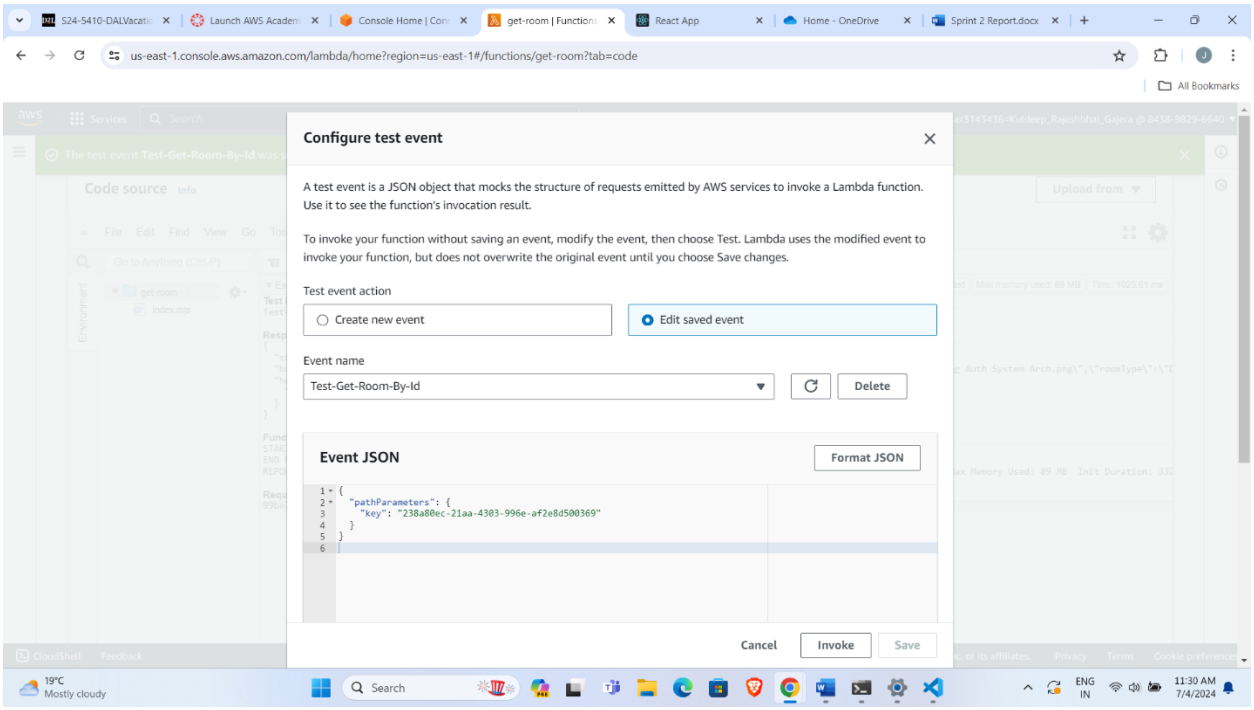

Figure 6: Configuring and executing the Get Room By Id test in AWS Lambda.

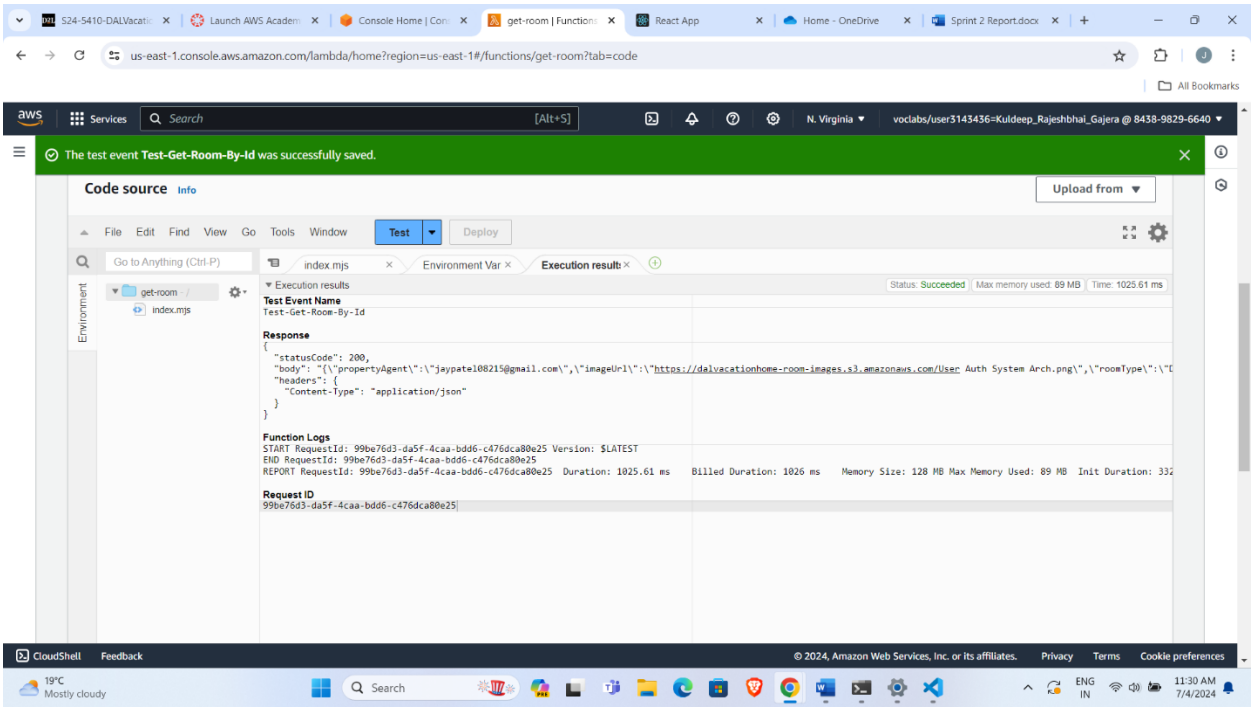

Figure 7: Execution log from CloudWatch for the Get Room By Id test.

Lambda Test Get All Rooms Event:

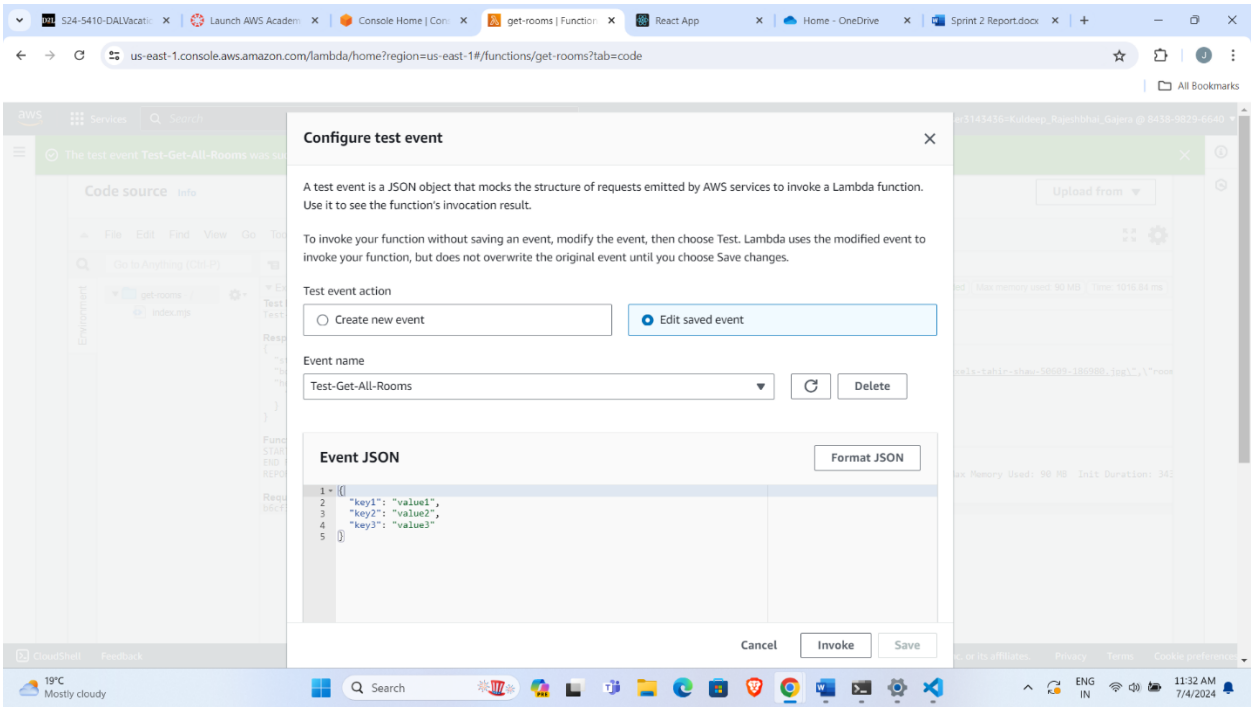

Figure 8: Setting up and executing the Get All Rooms test in AWS Lambda.

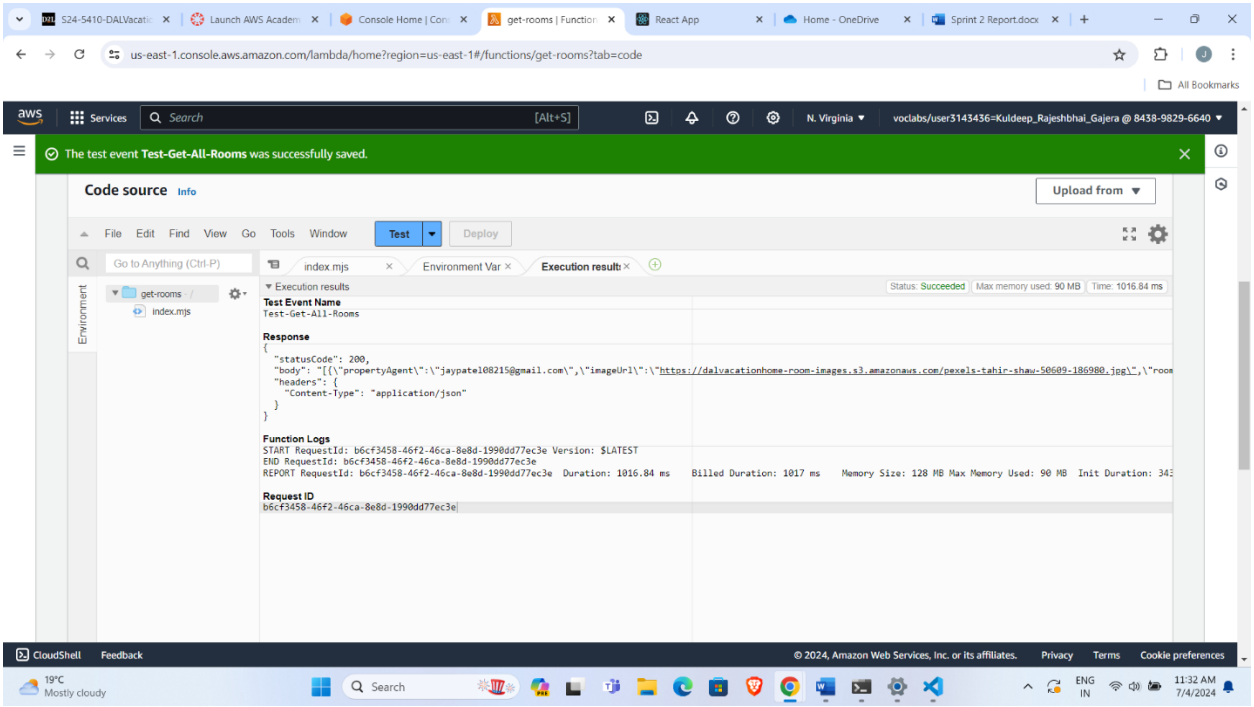

Figure 9: CloudWatch log details from the Get All Rooms test execution.

Lambda Test Booking Room Event:

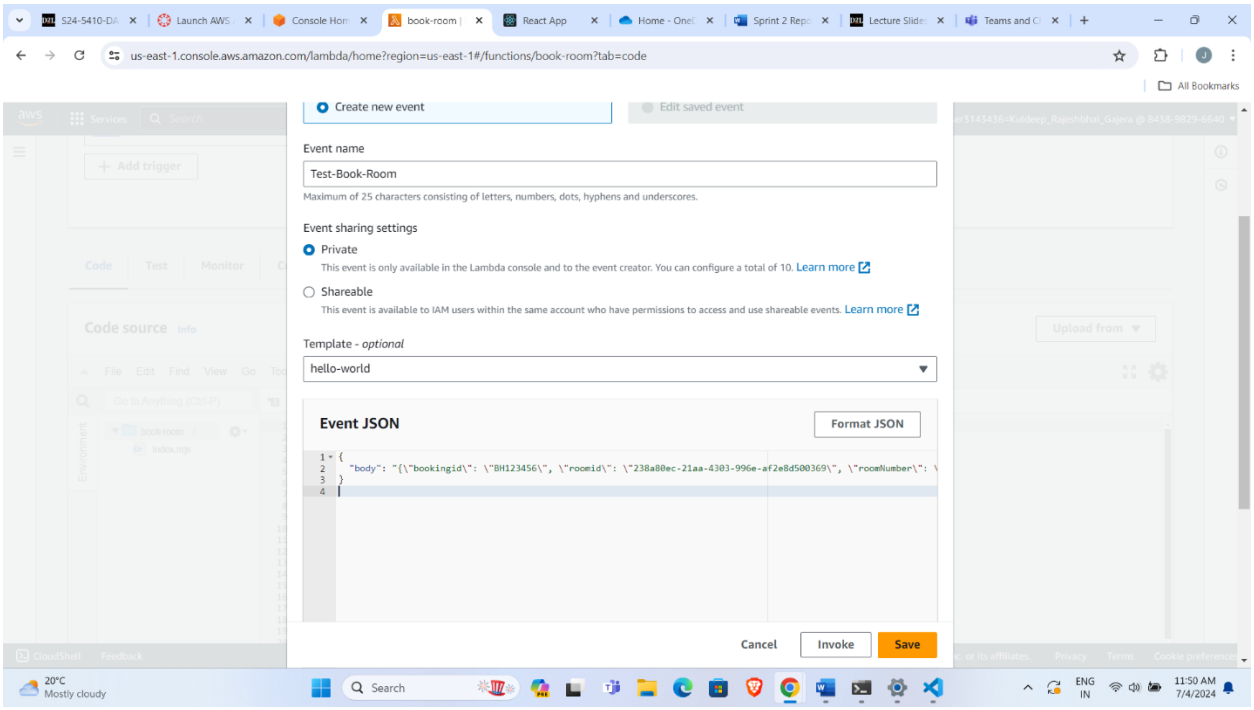

Figure 10: Initiating and running the Booking Room test in AWS Lambda.

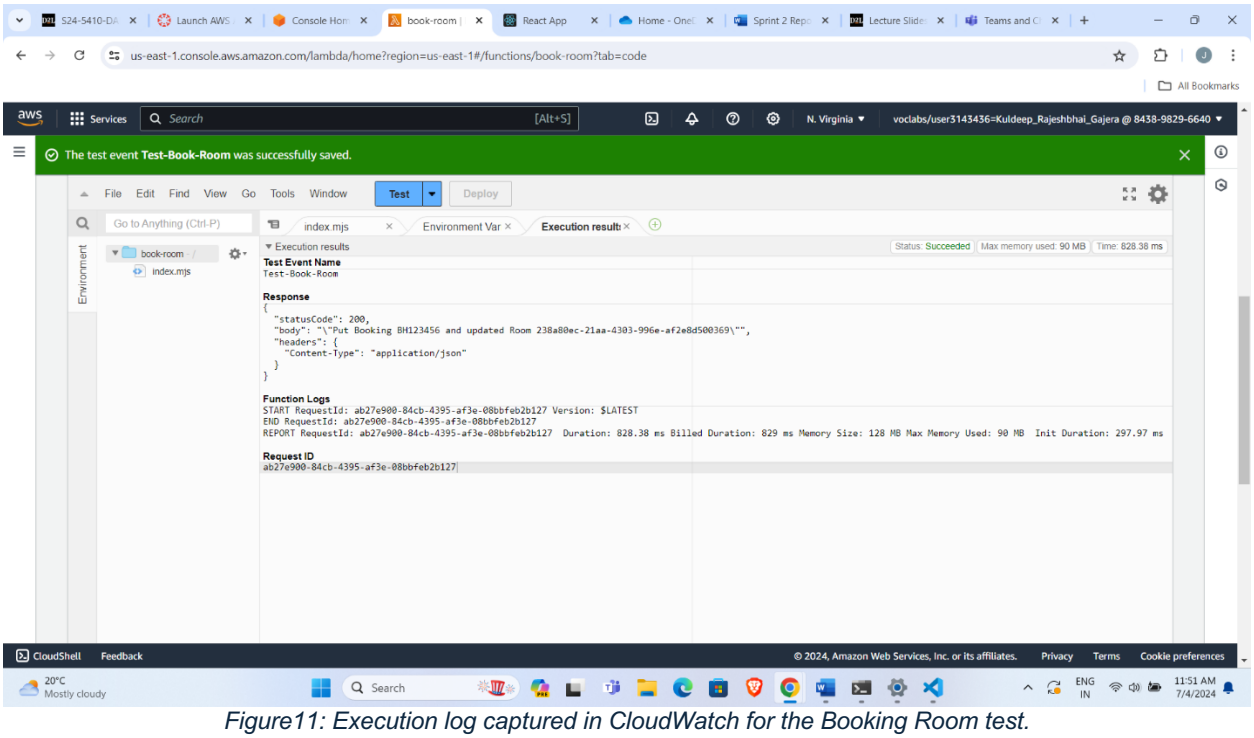

Figure11: Execution log captured in CloudWatch for the Booking Room test.

Lambda Test Get Booking by ID Event:

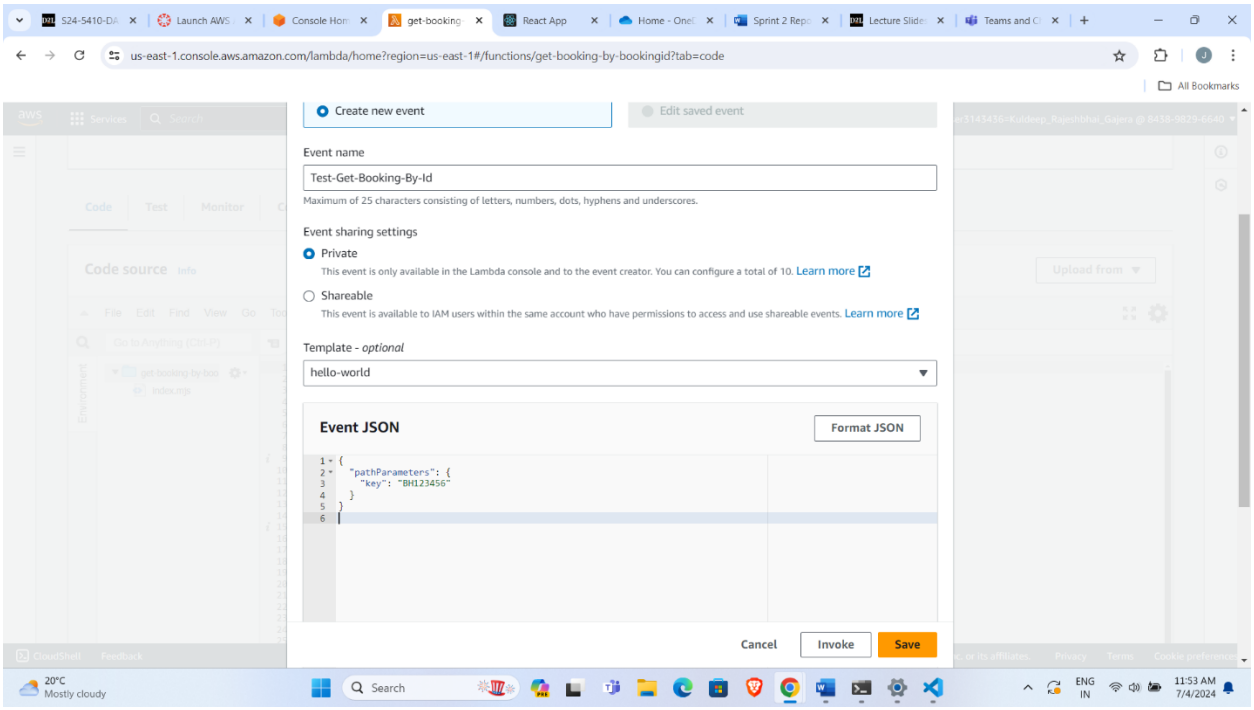

Figure 12: Configuring and launching the Get Booking By Id test in AWS Lambda.

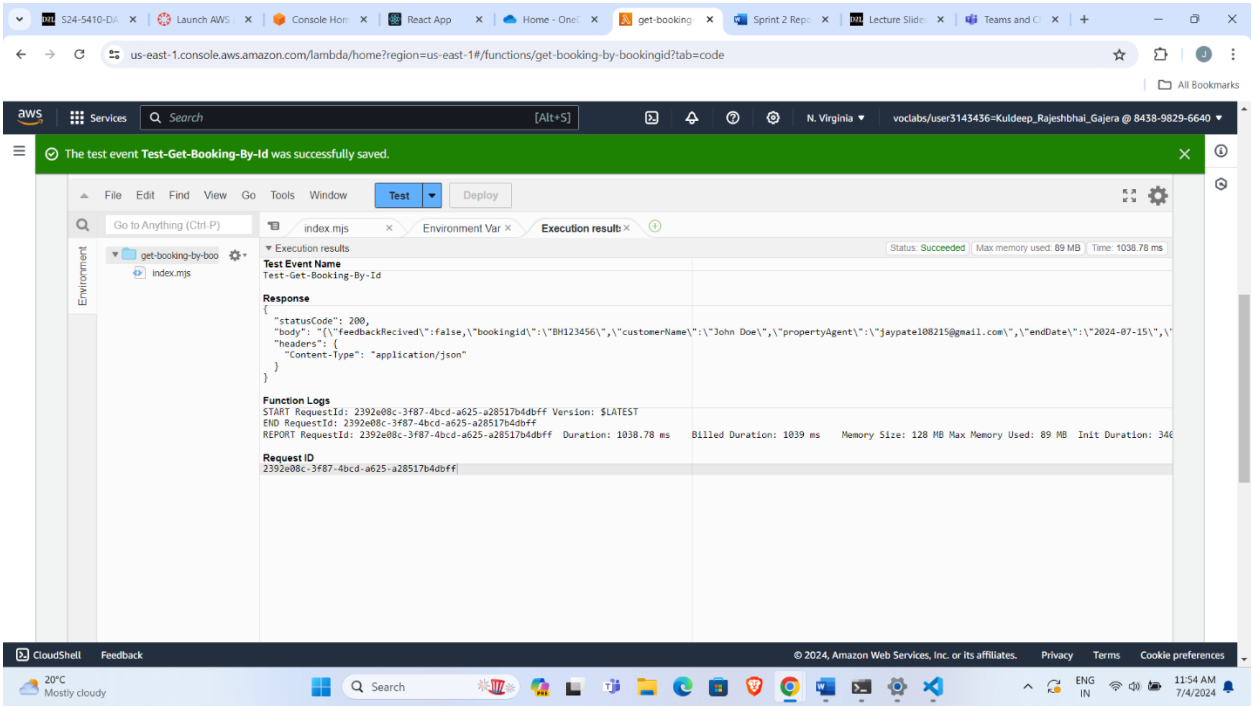

Figure 13: CloudWatch log displaying results from the Get Booking By Id test.

Lambda Test Get Bookings by User:

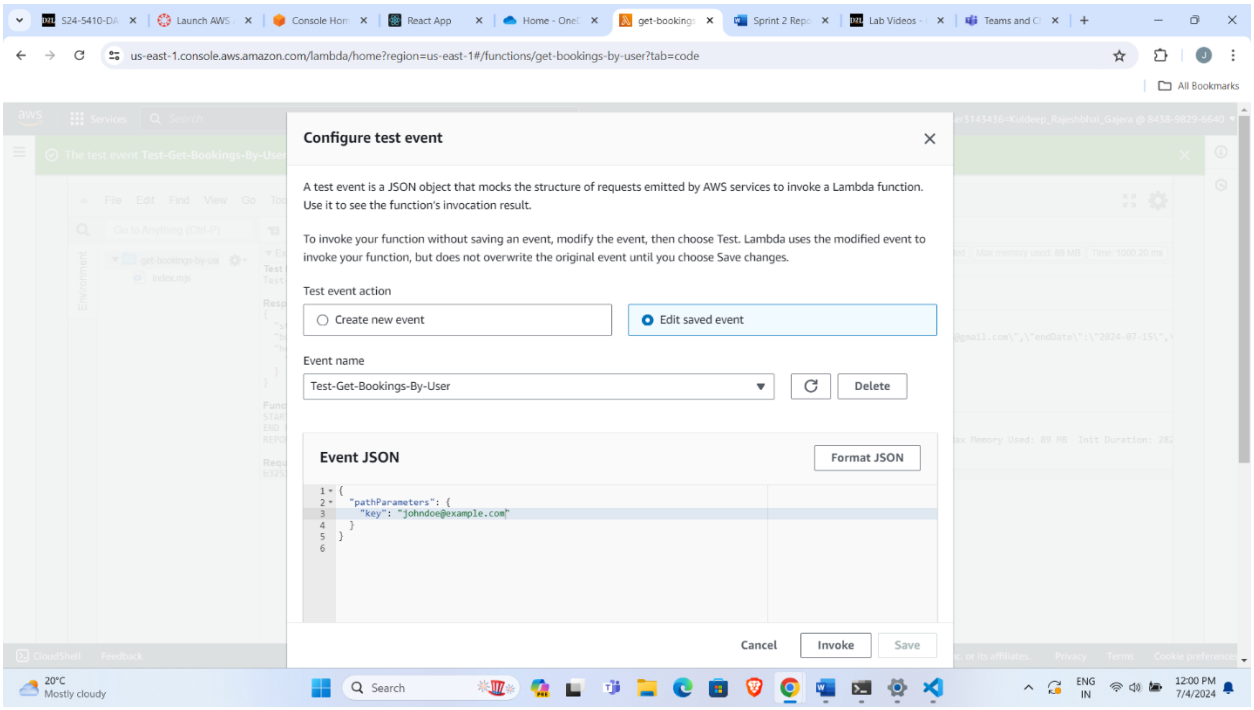

Figure 14: Setting up and executing the Get Bookings by User test in AWS Lambda.

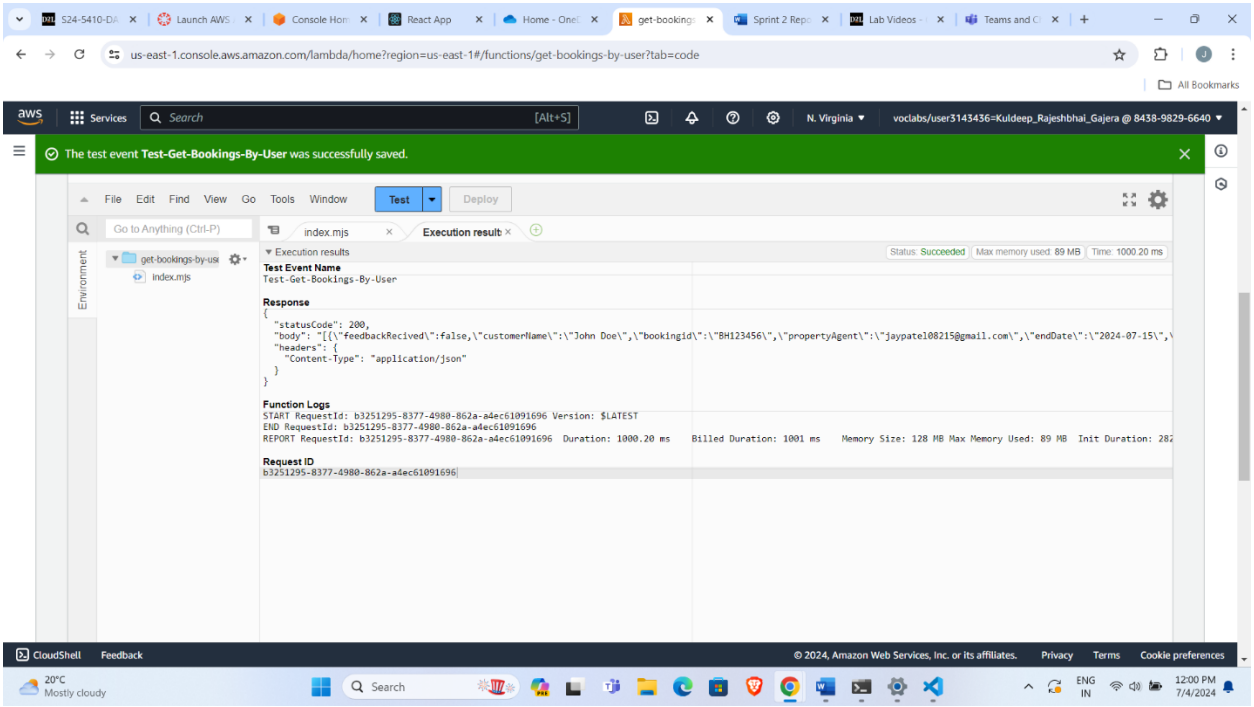

Figure 15: CloudWatch log outcomes from the Get Bookings by User test.

Lambda Test Give Feedback for Room Event:

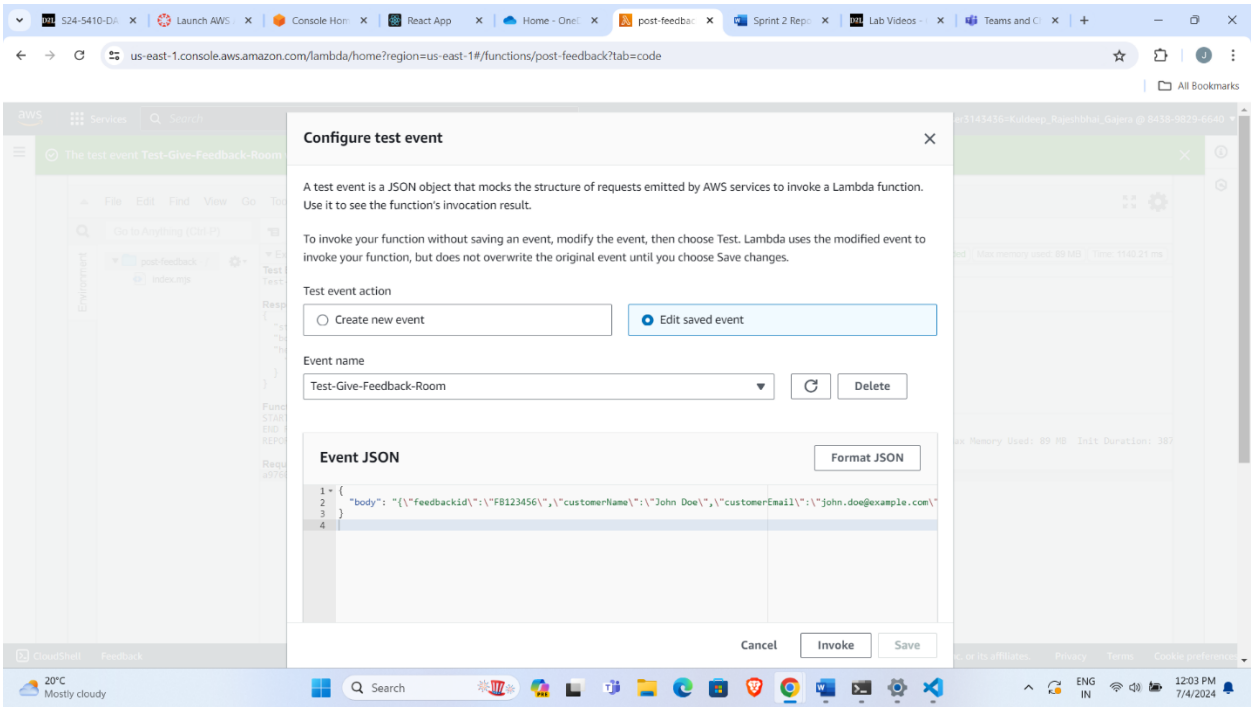

Figure 16: Initiating and performing the Give Feedback for Room test in AWS Lambda.

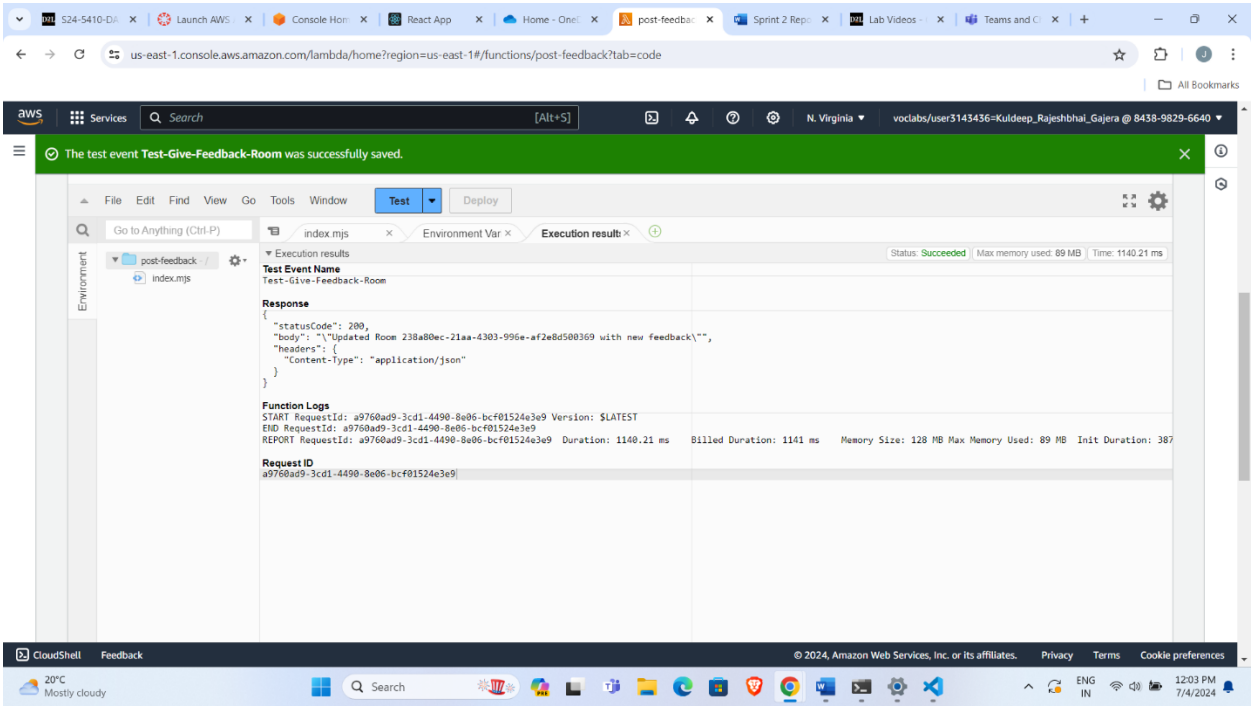

Figure 17: Execution log from CloudWatch for the Give Feedback for Room test.

Lambda Test Storing User Details:

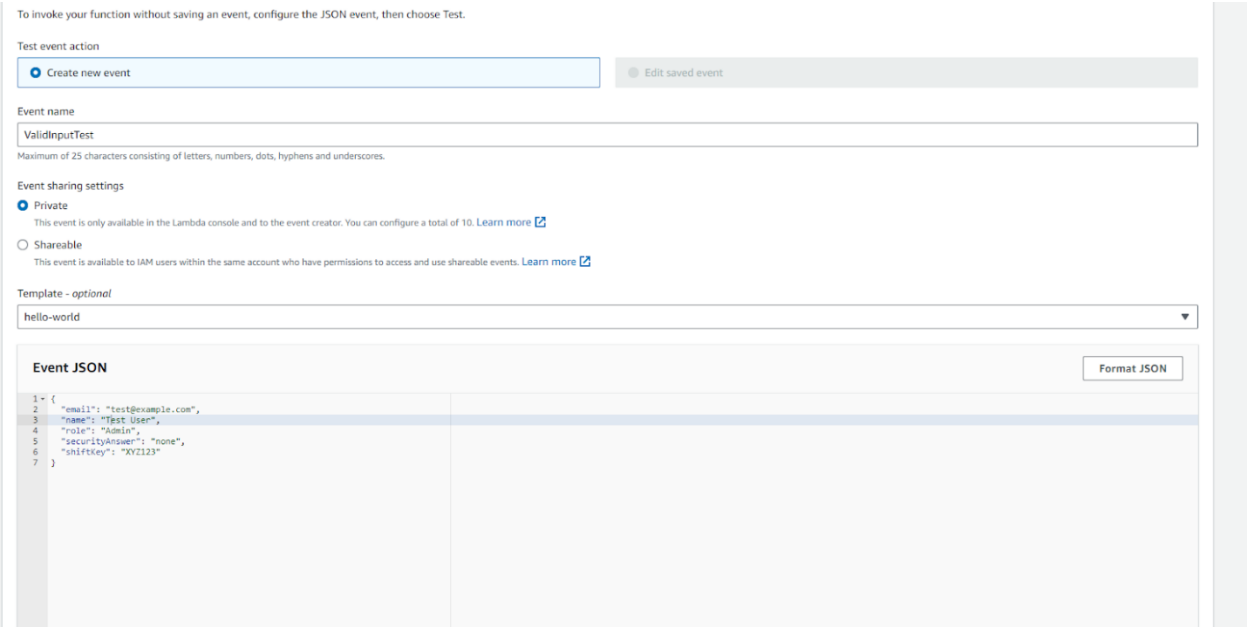

Figure 18: Setting up and executing the Storing User Details test in AWS Lambda.

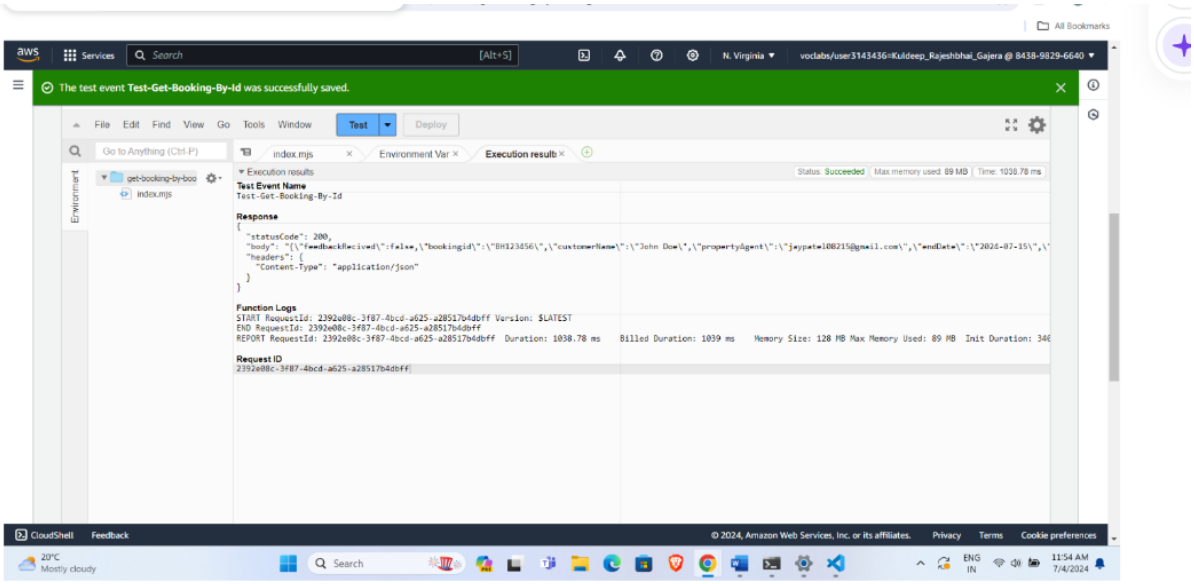

Figure 19: CloudWatch log displaying the Storing User Details test results.

Lambda Test Retrieving Shift Key of a User:

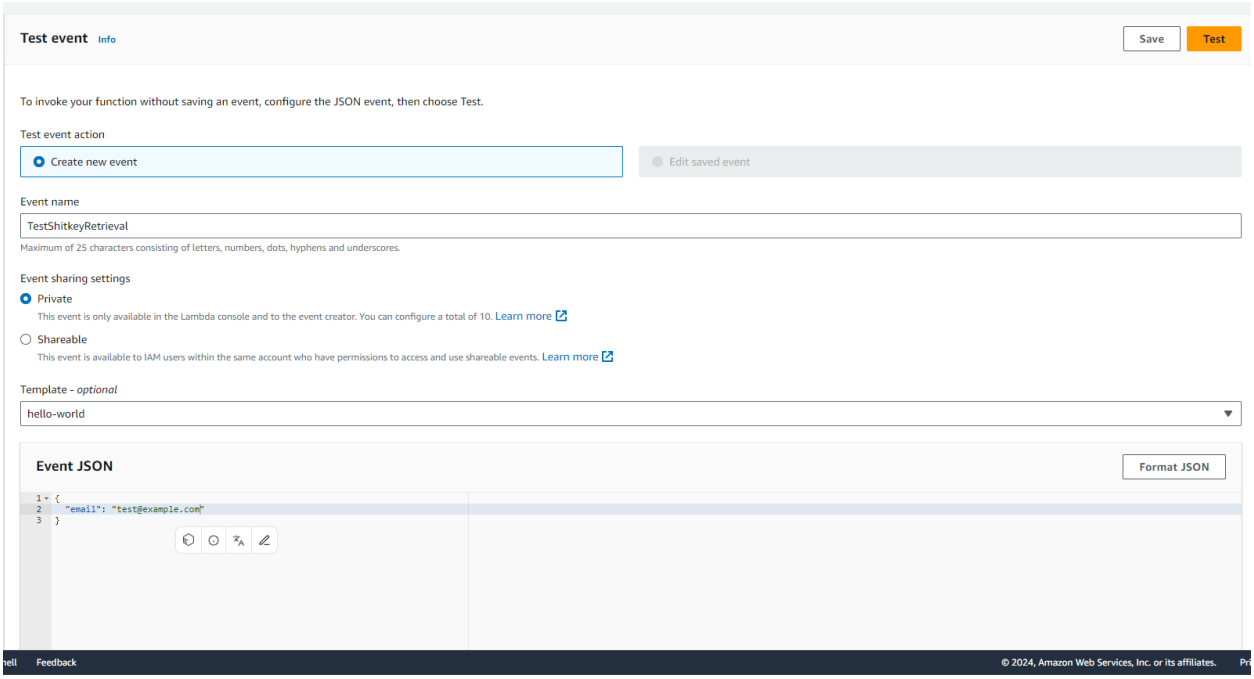

Figure 20: Preparing and executing the Retrieving Shift Key of a User test in AWS Lambda.

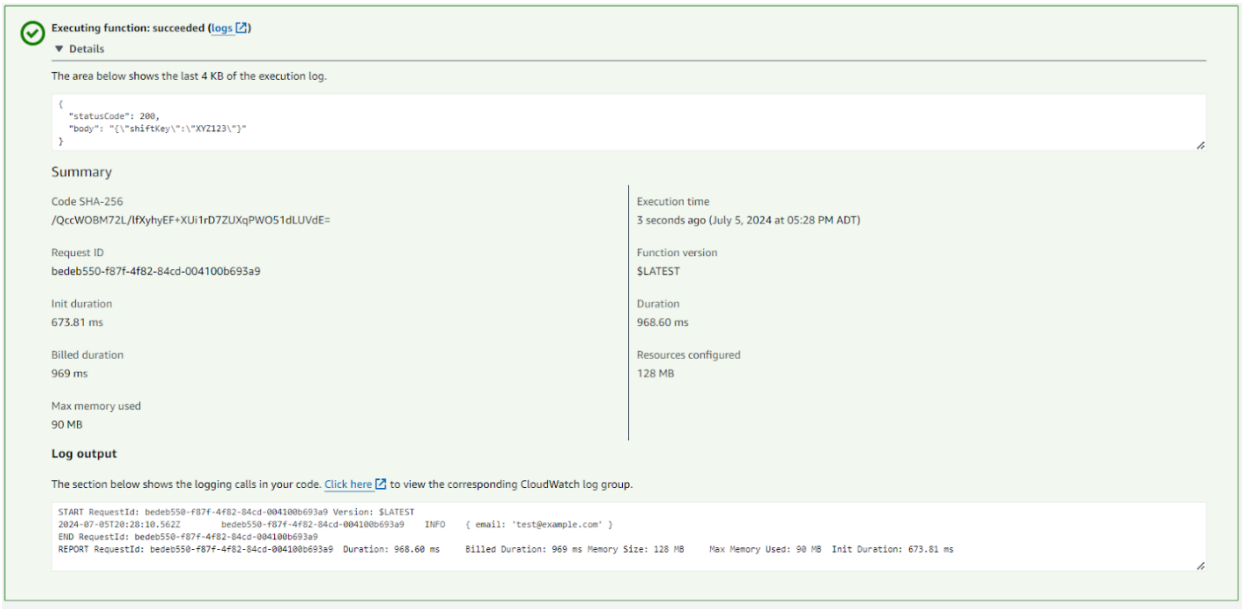

Figure 21: CloudWatch log details from the Retrieving Shift Key of a User test.

Lambda Test Retrieving Security Answer to User's Security Question:

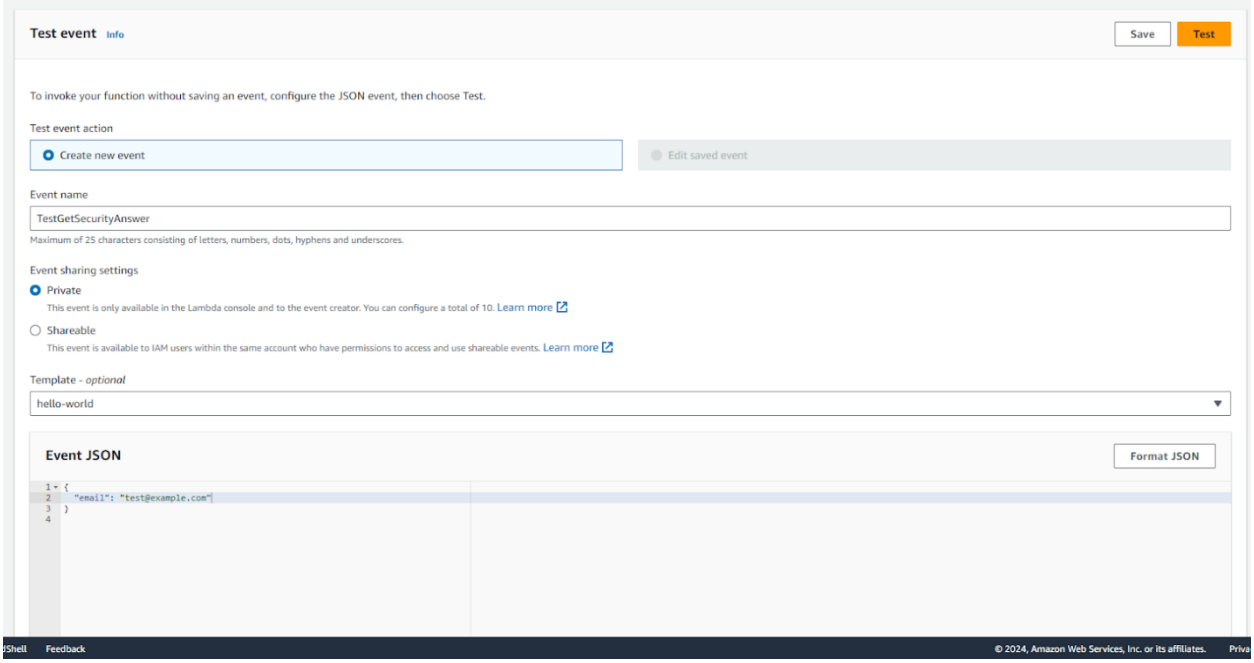

Figure 22: Configuring the test for Retrieving Security Answer to User's Security Question in AWS Lambda

.

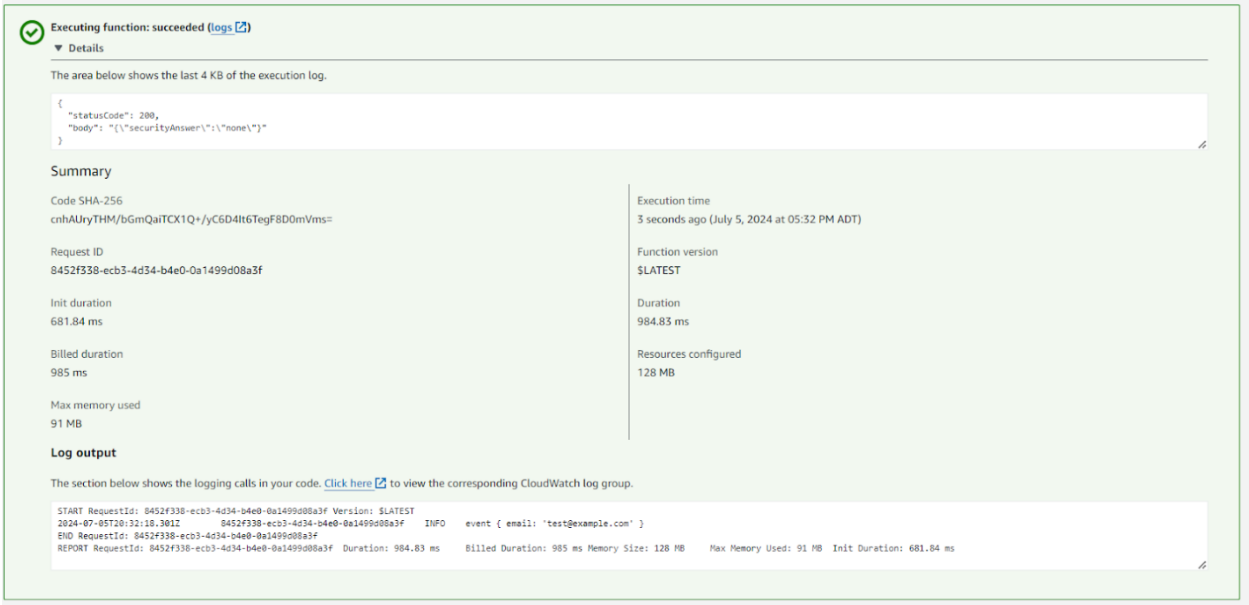

Figure 23: CloudWatch log output from the Retrieving Security Answer test.

Lambda Test Deleting a Room:

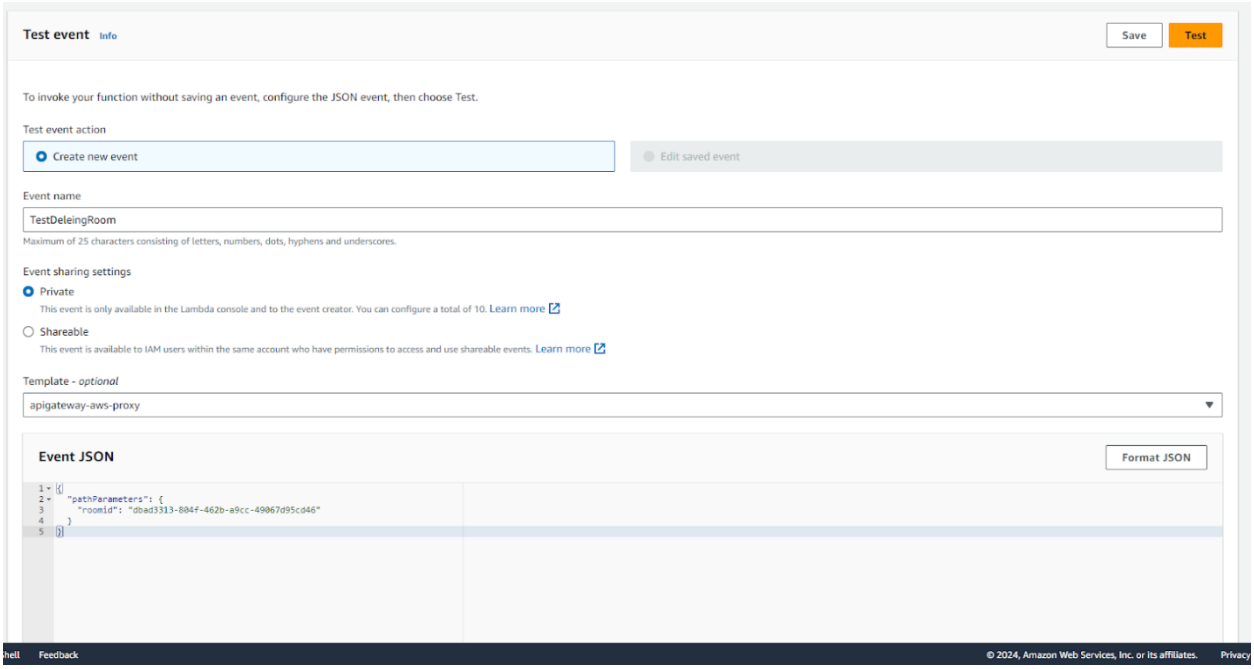

Figure 24: Setting up and executing the Deleting a Room test in AWS Lambda.

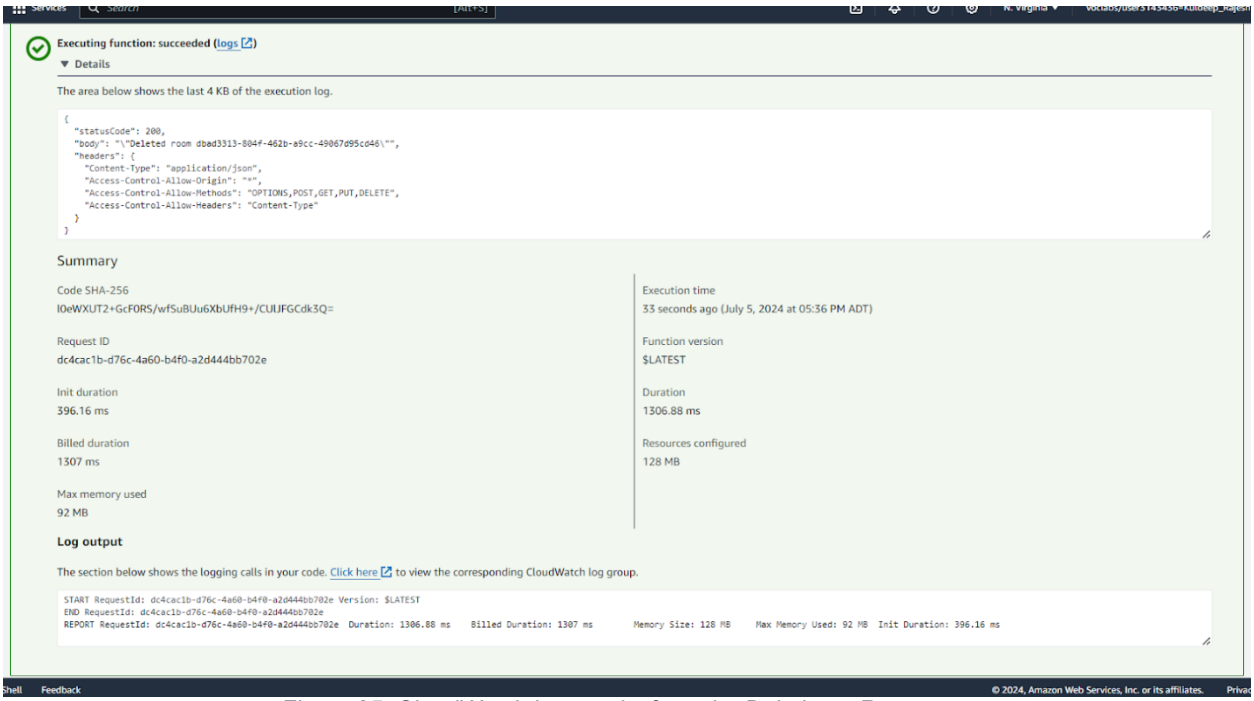

Figure 25: CloudWatch log results from the Deleting a Room test.

Lambda Test Updating Room Details:

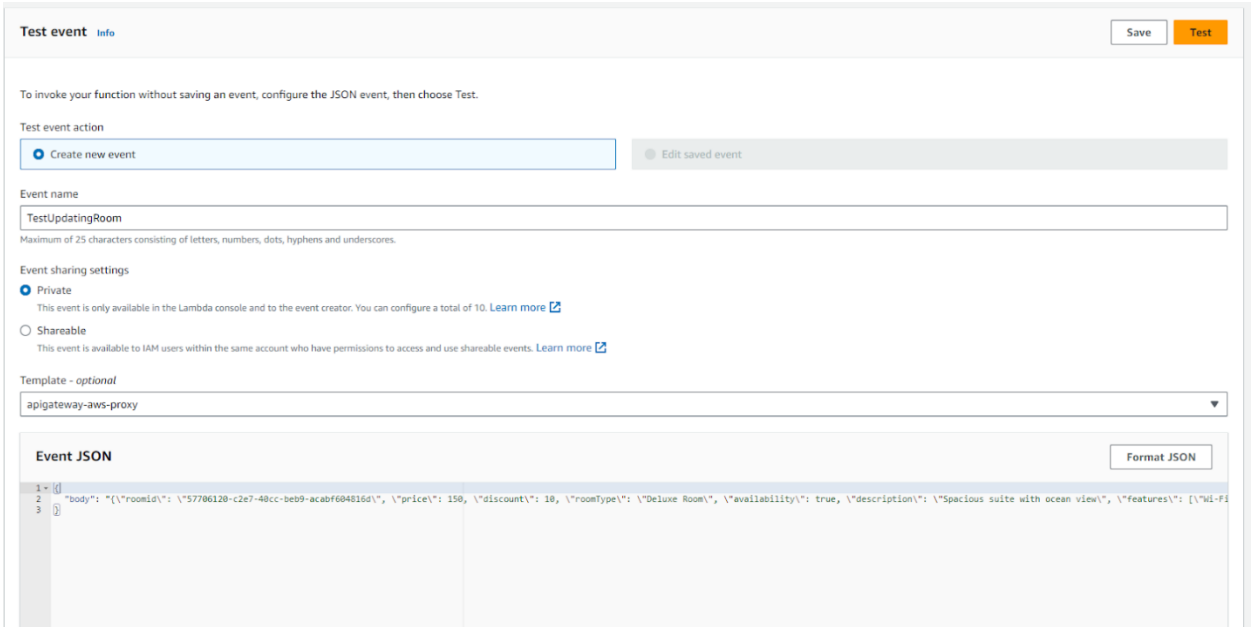

Figure 26: Executing the Updating Room Details test in AWS Lambda.

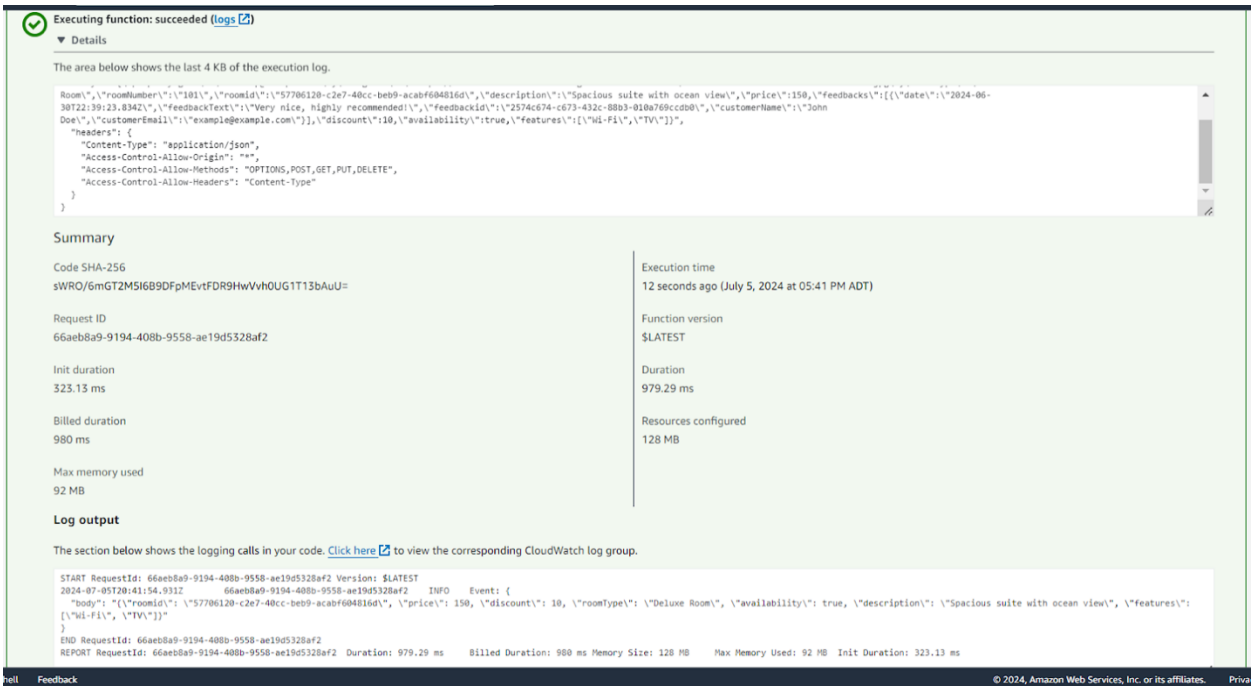

Figure 27: CloudWatch log showcasing the results from the Updating Room Details test.

IntentHandler:

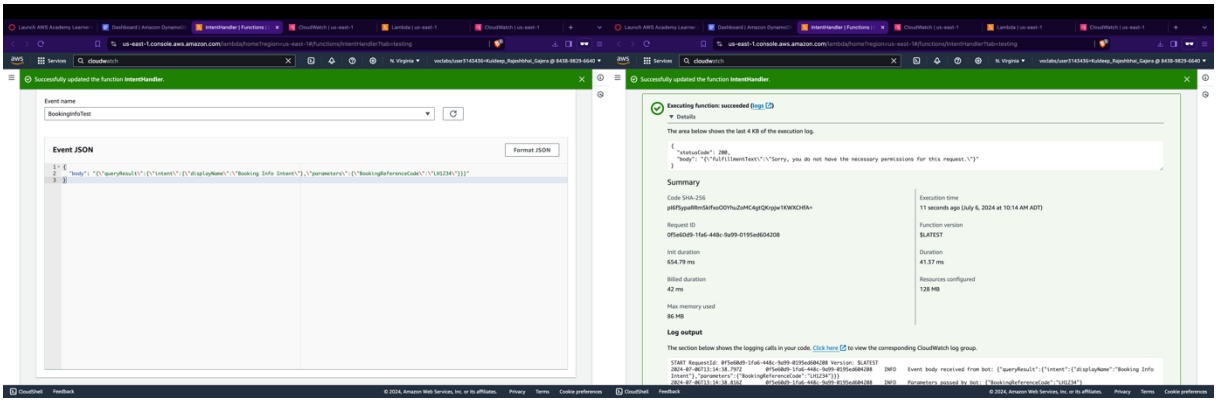

Figure 28: Setting up and executing the IntentHandler test, with results shown in the CloudWatch logs.

FetchBookingDetails

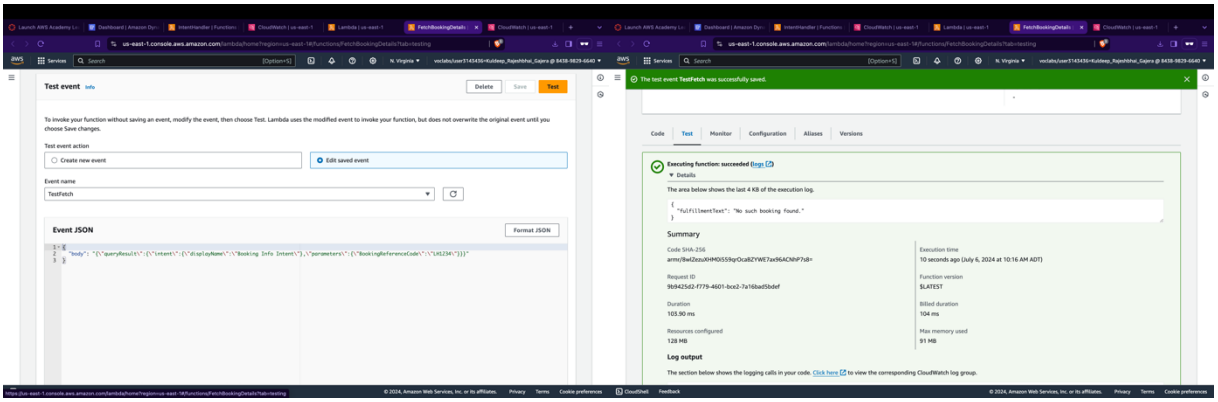

Figure 29: Running the FetchBookingDetails test and displaying execution logs in CloudWatch.

HandoffCusotmerSupport

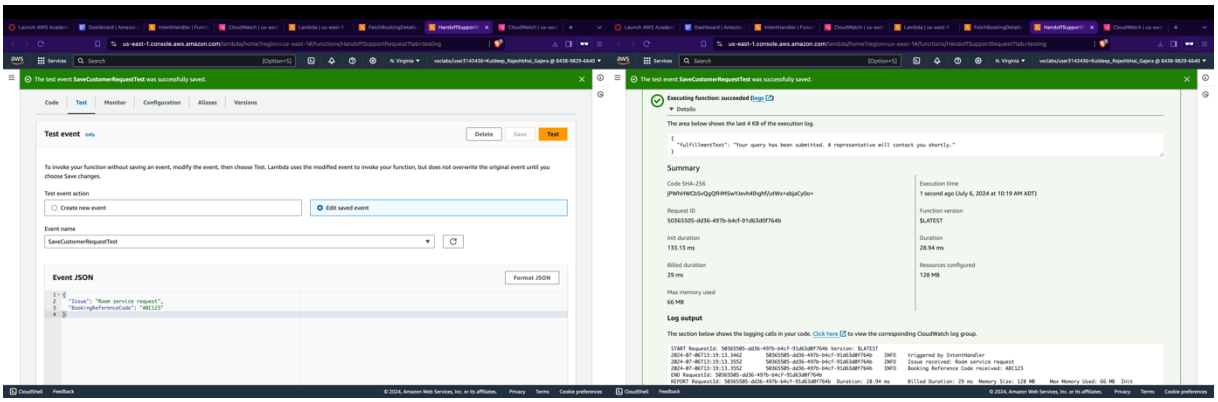

Figure 30: Executing the HandoffCustomerSupport test with corresponding CloudWatch log outcomes.

API Testing:

API Testing Create Room:

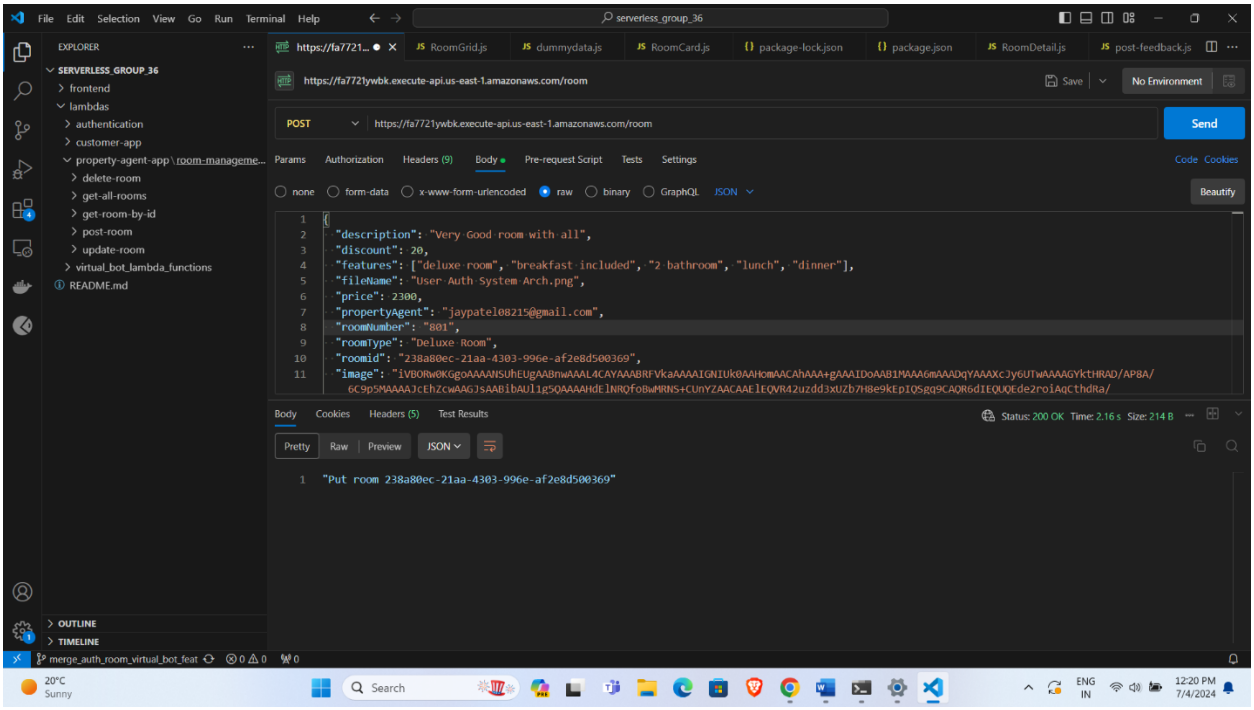

Figure 31: API call execution to create a room, highlighting the request and response.

API Test Get Room by roomId:

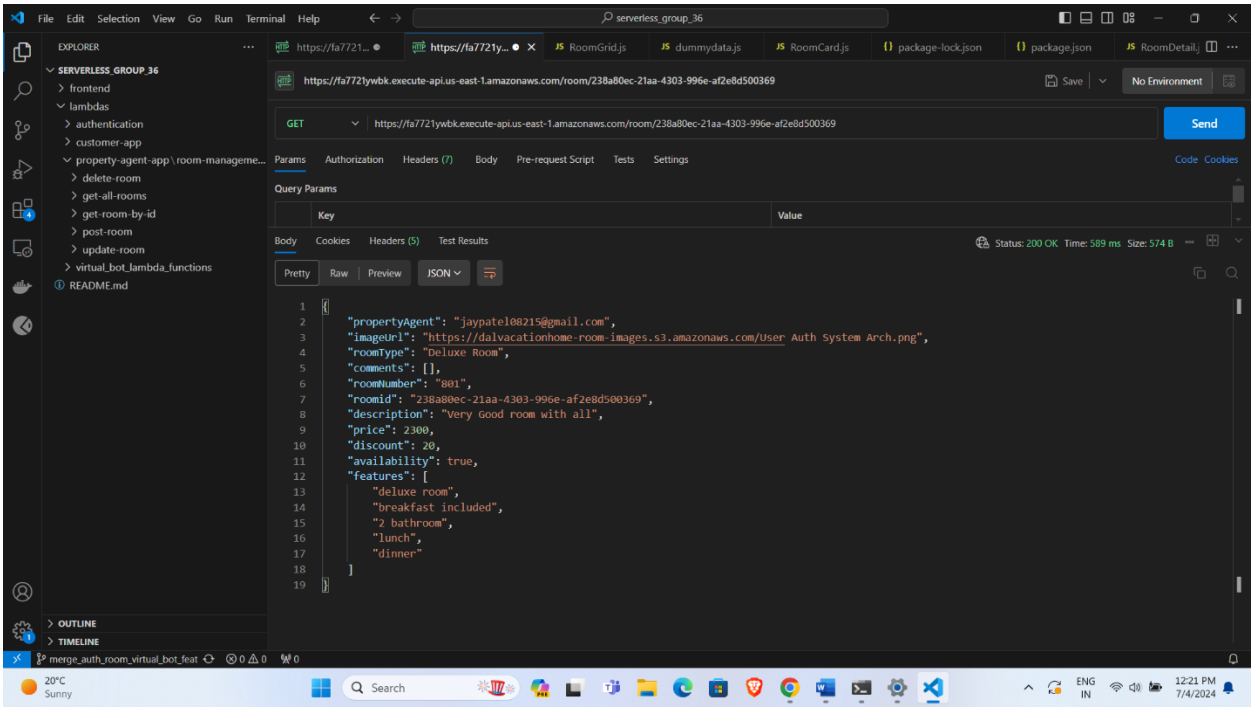

Figure 32: API interaction for retrieving a room by roomId, displaying request and response details.

API Test Get All Rooms:

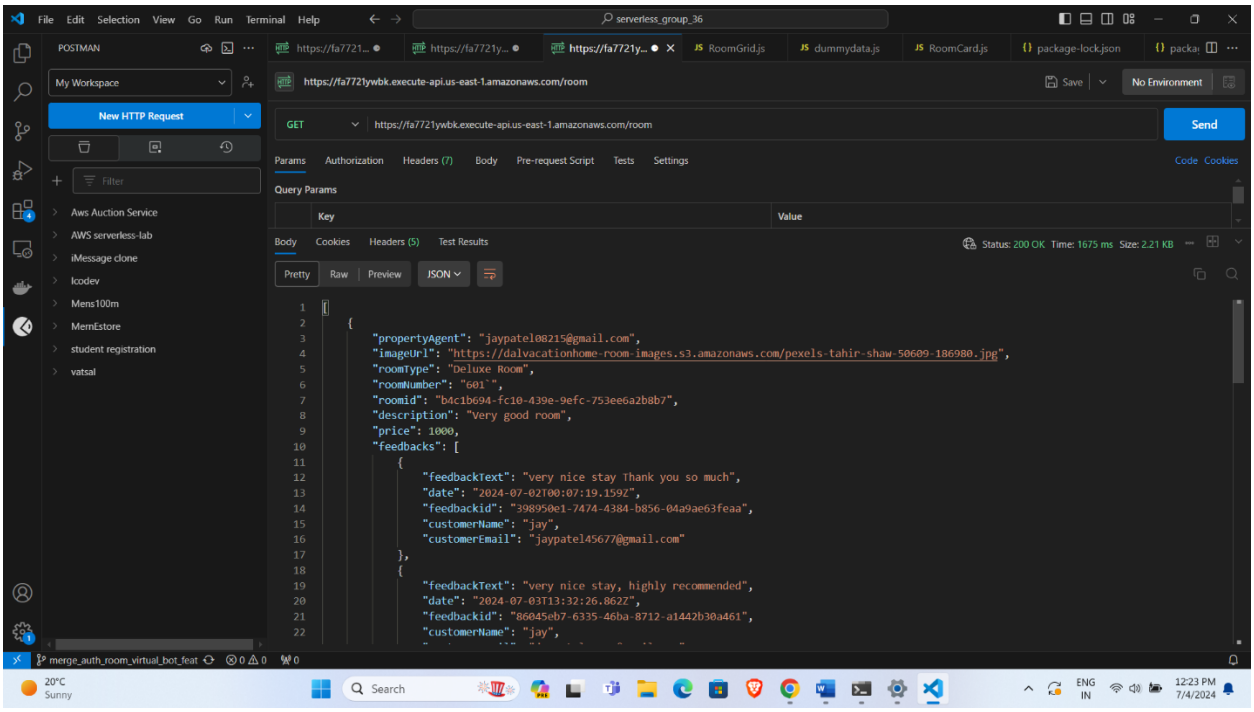

Figure 33: API request and response for fetching all rooms shown.

API Testing Booking Room:

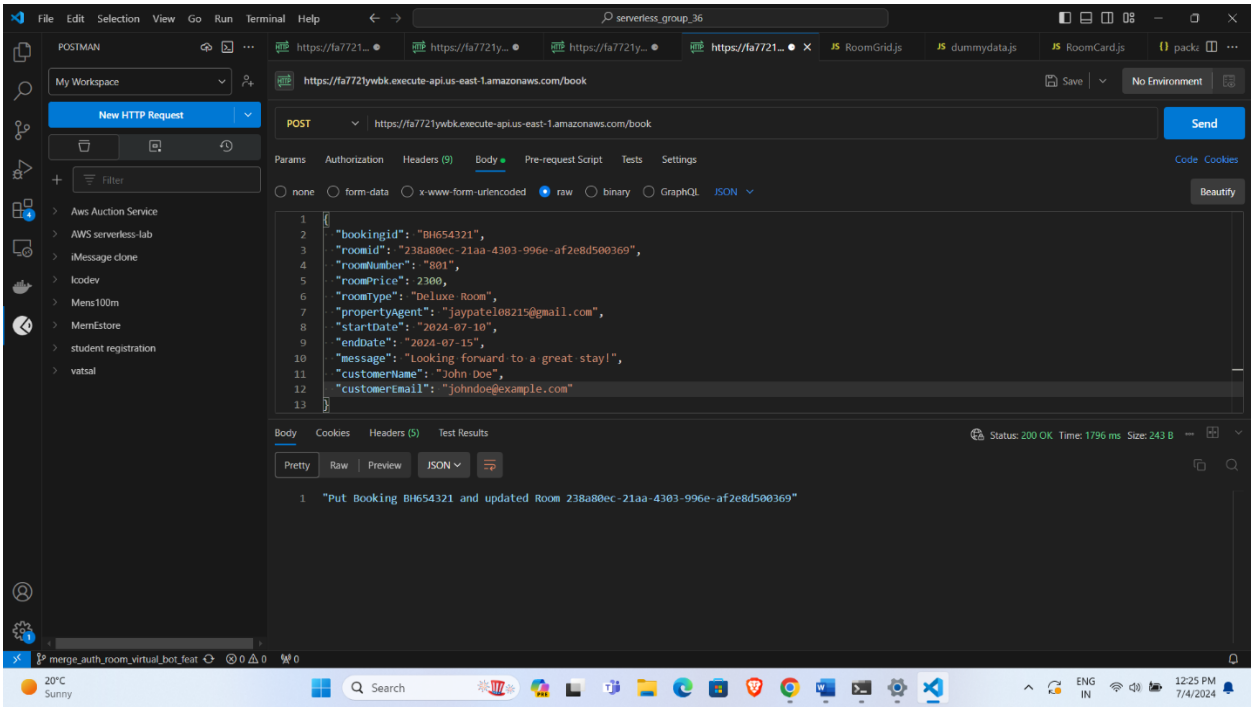

Figure 34: Executing an API request to book a room, with response displayed.

API Testing Get Booking by BookingId:

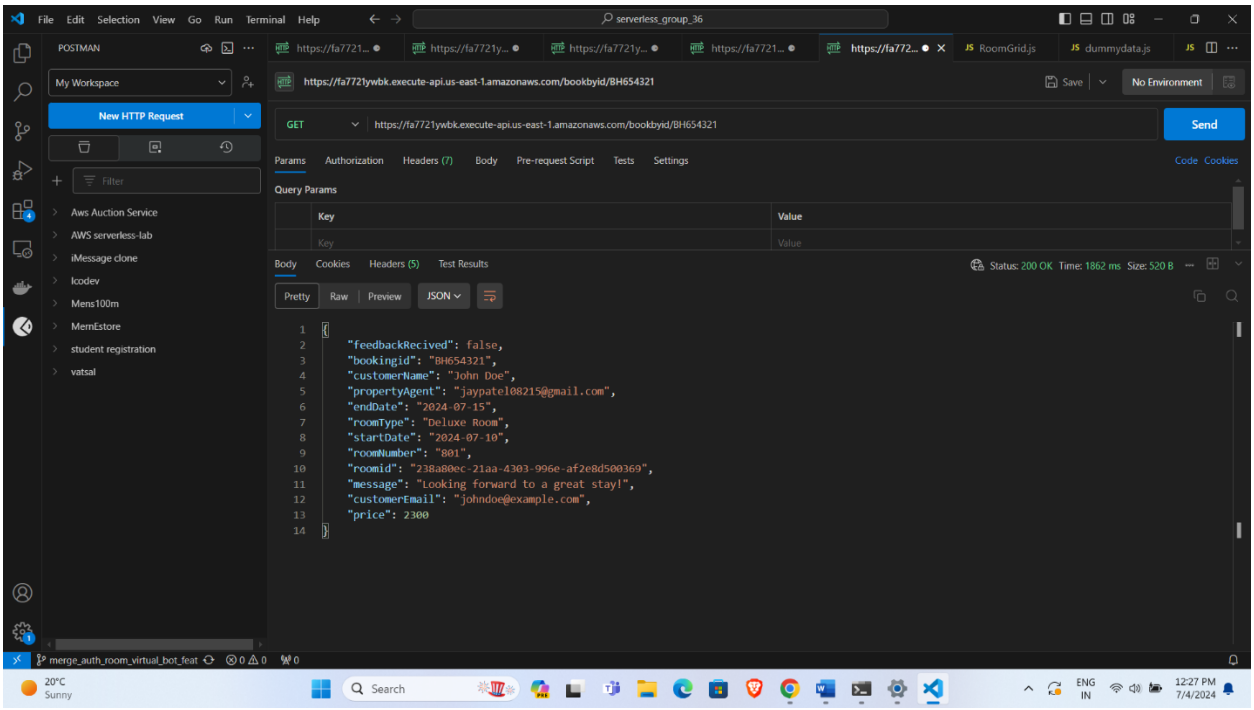

Figure 35: API interaction to retrieve booking details by BookingId, showing request and response.

API Testing Get Bookings by Customer:

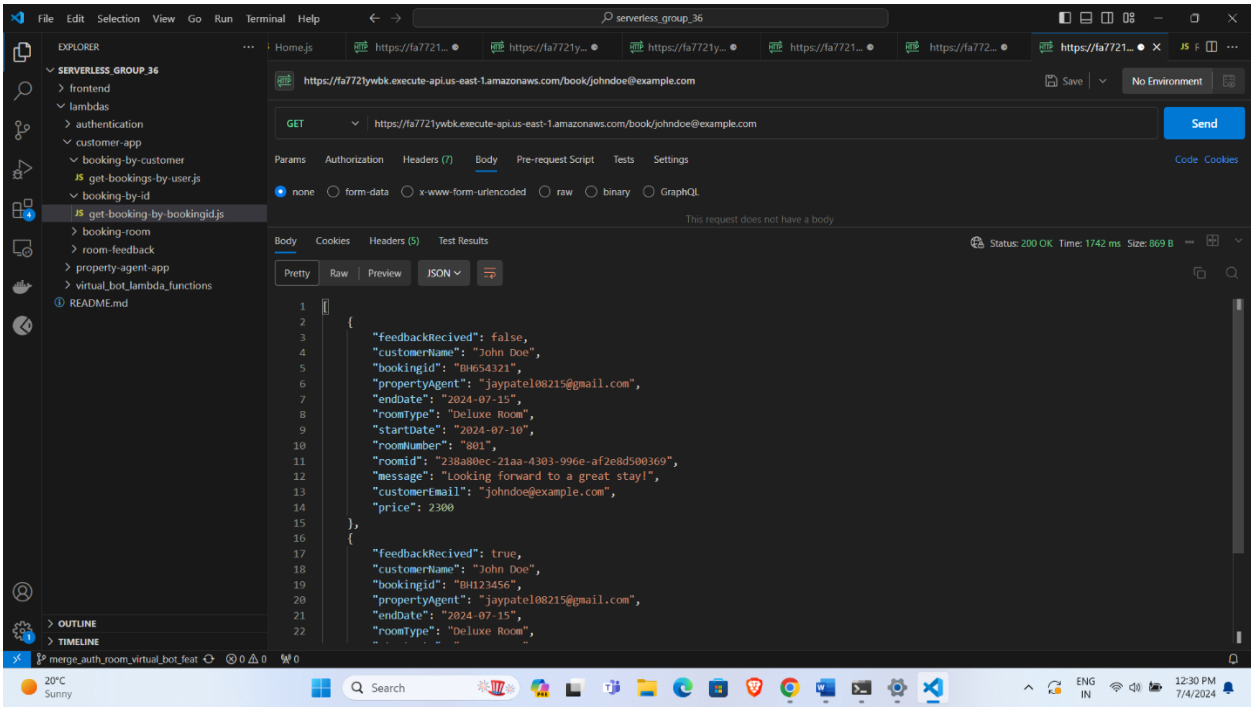

Figure 36: API request to get bookings by a customer, showcasing the response.

API Testing Give Feedback for Room:

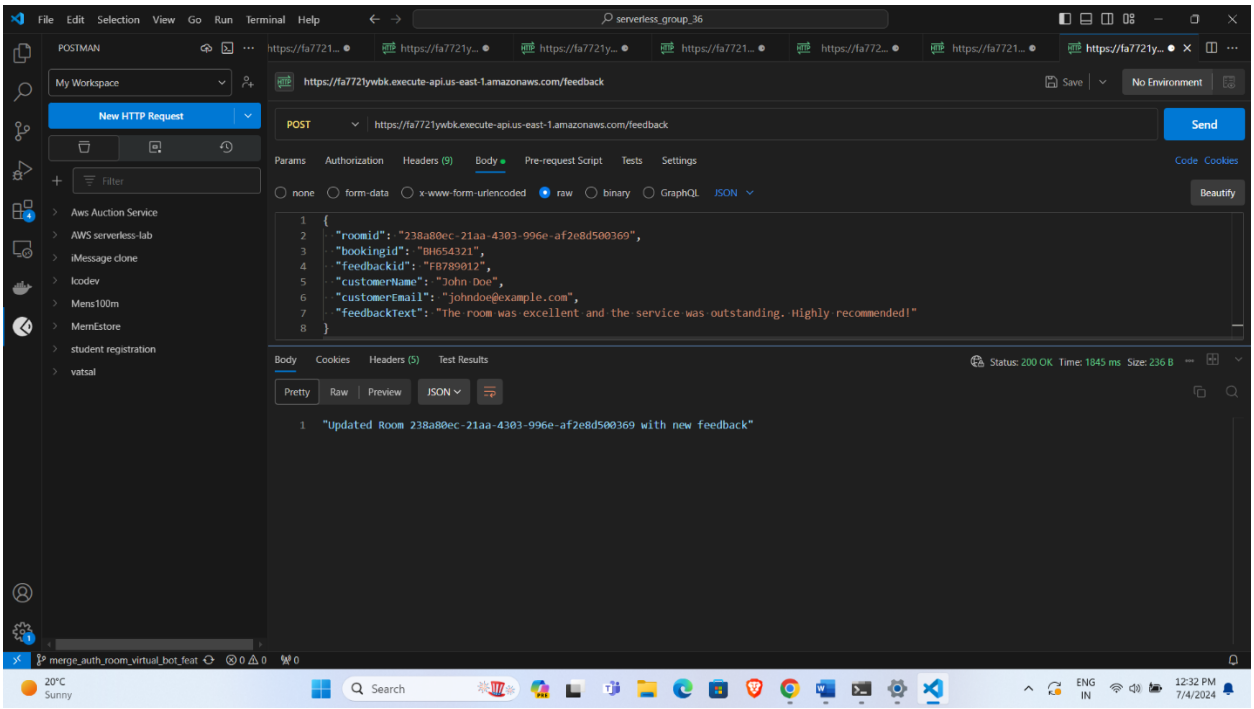

Figure 37: API process for submitting feedback for a room, including the response.

API Testing Storing User Registration Details in DynamoDB:

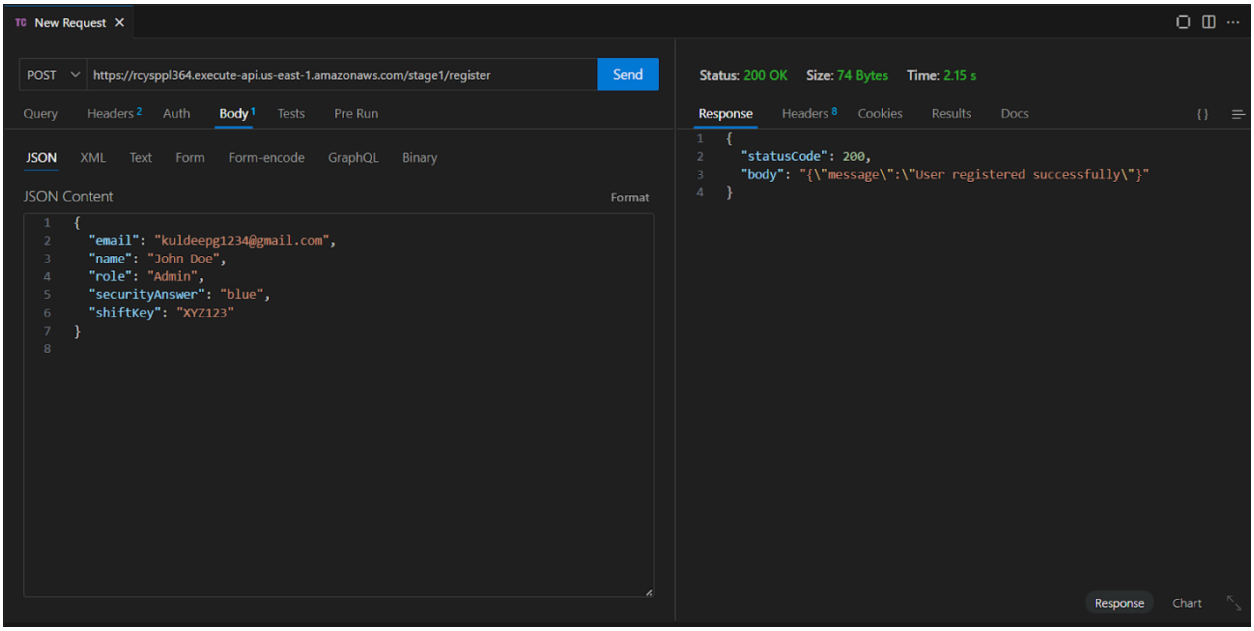

Figure 38: Executing an API request to store user registration details in DynamoDB, with the response shown.

API Testing Getting Shift key of User from DynamoDB:

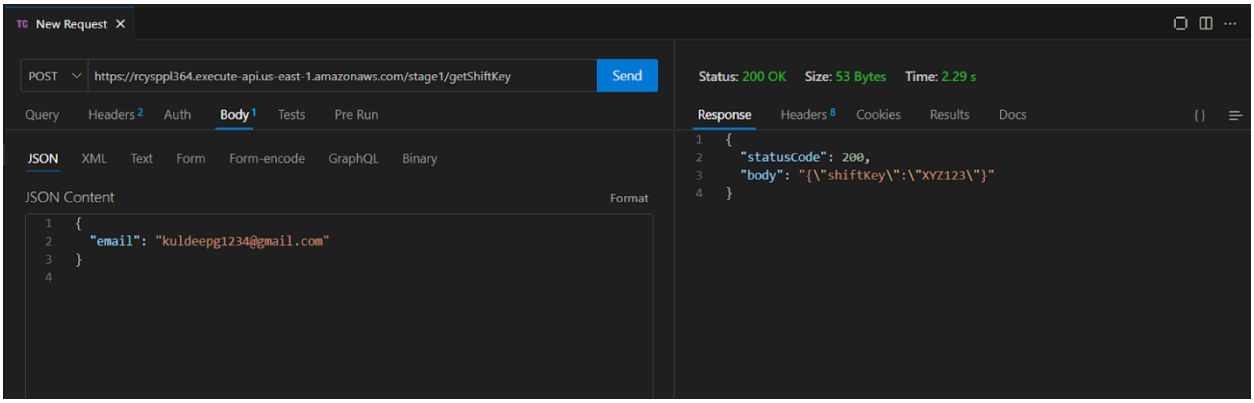

Figure 39: API request and response for retrieving a user's shift key from DynamoDB.

API Testing Getting Security Answer to User's Security Question from DynamoDB

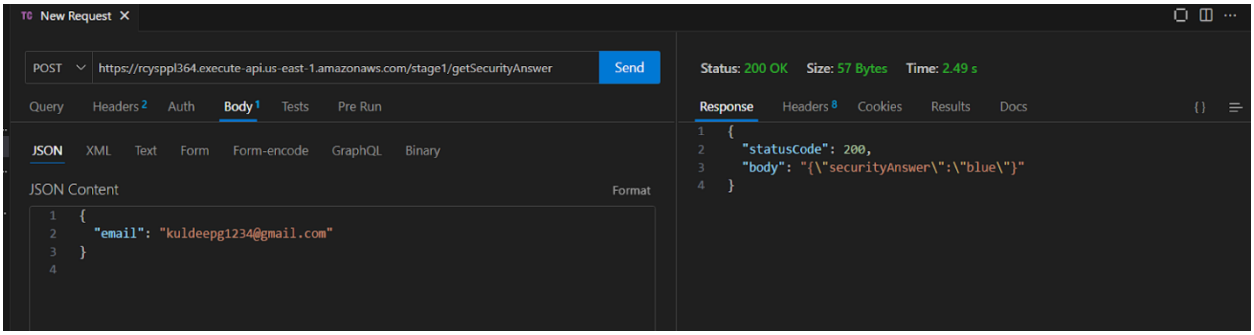

Figure 40: API interaction for getting a user's security answer from DynamoDB, displaying details.

API Testing Deleting a Room and Image Associated with it from S3:

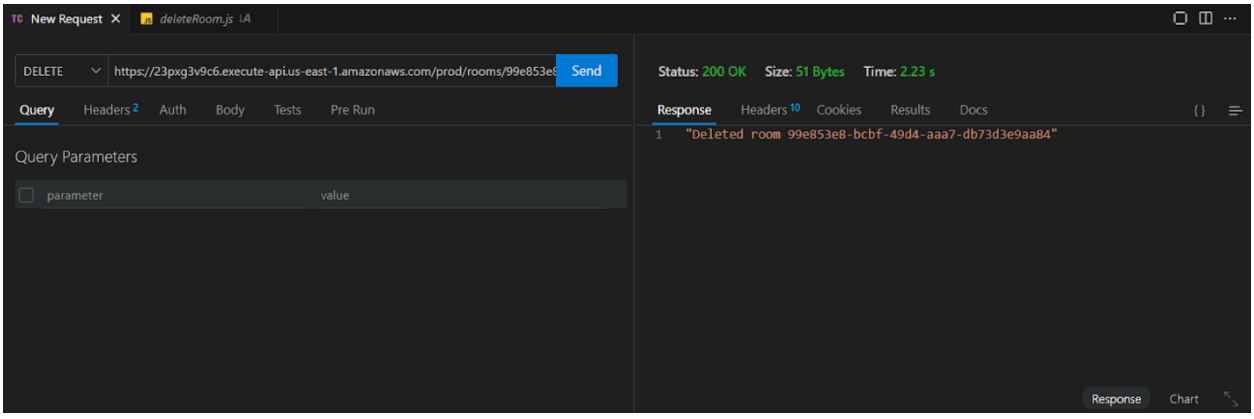

Figure 41: API call to delete a room and its associated image from S3, response shown.

API Testing Updating Room Details and Optionally Image into S3:

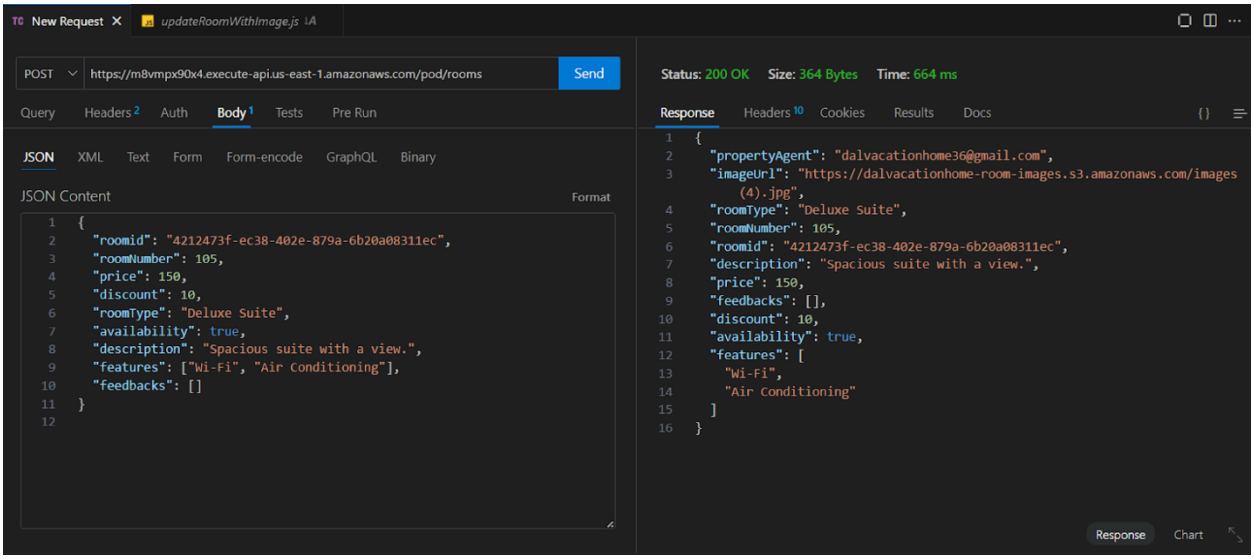

Figure 42: API request to update room details and optionally upload an image into S3, displaying the response.

Evidence of testing for completed modules:

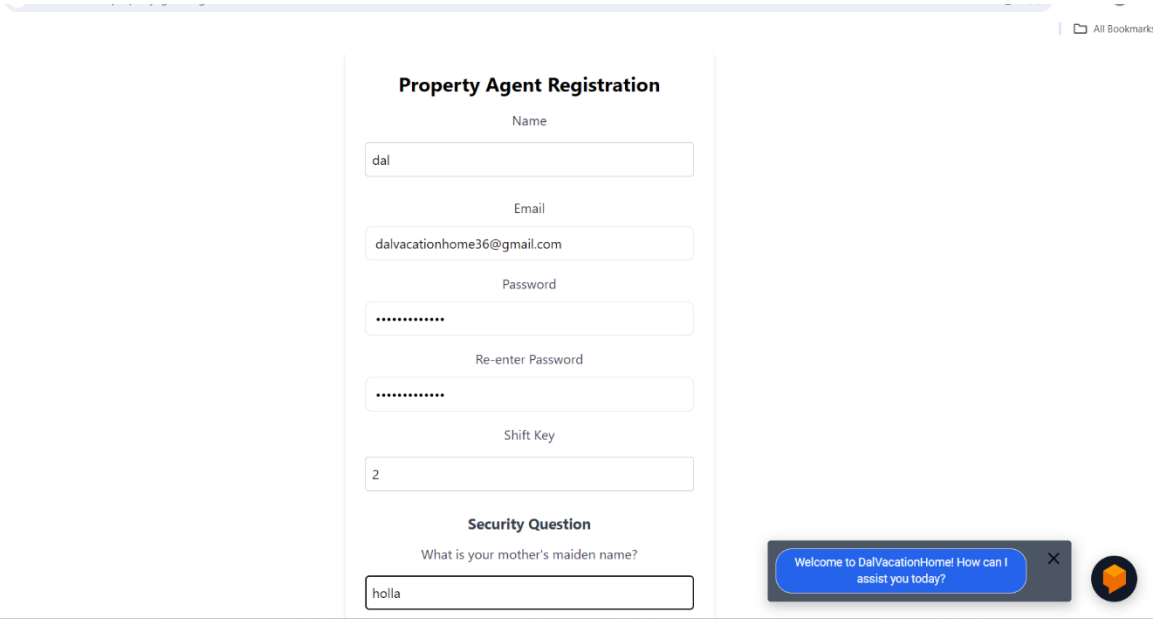

Figure 43: Screenshot of the Property Agent registration form, including fields for shift key and security question.

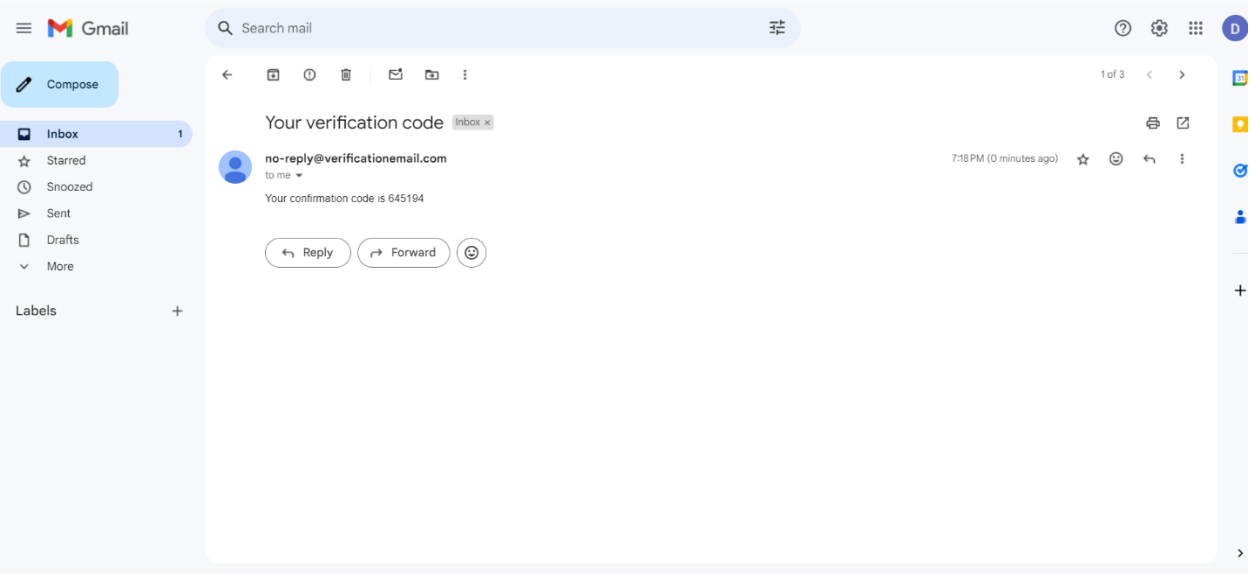

Figure 44: Display of the verification code received via email during registration.

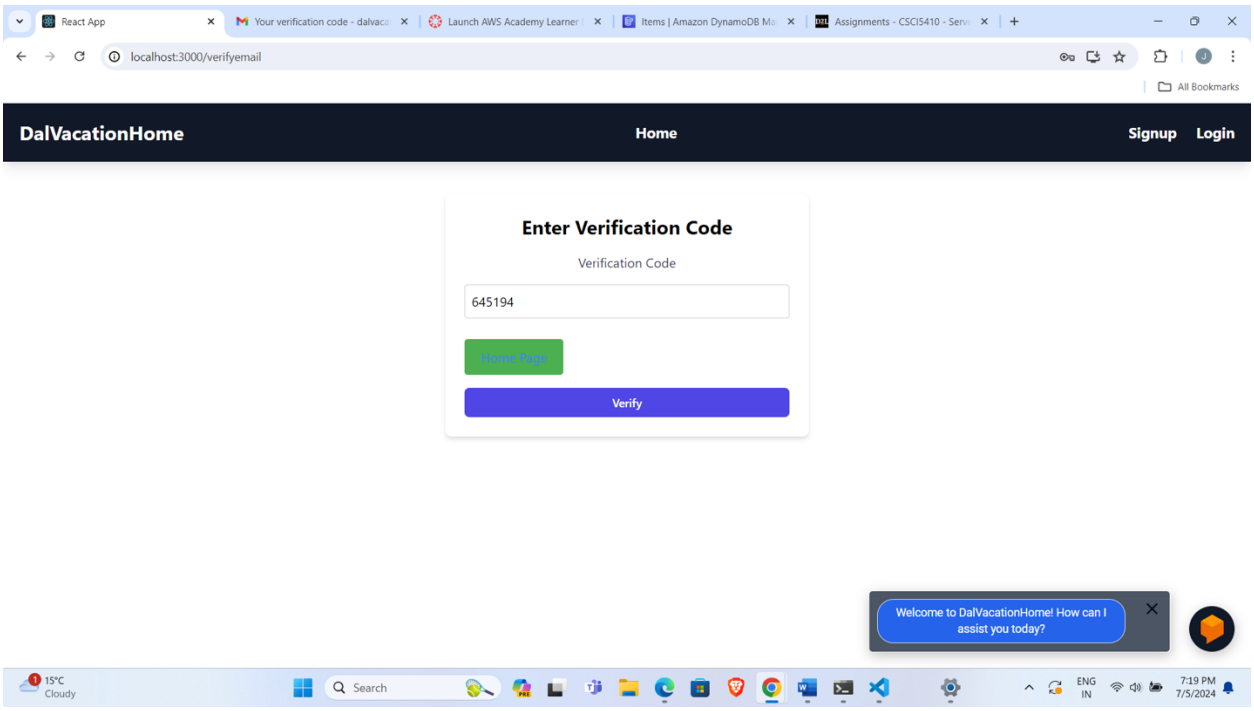

Figure 45: Screenshot showing the verification process of entering the received code.

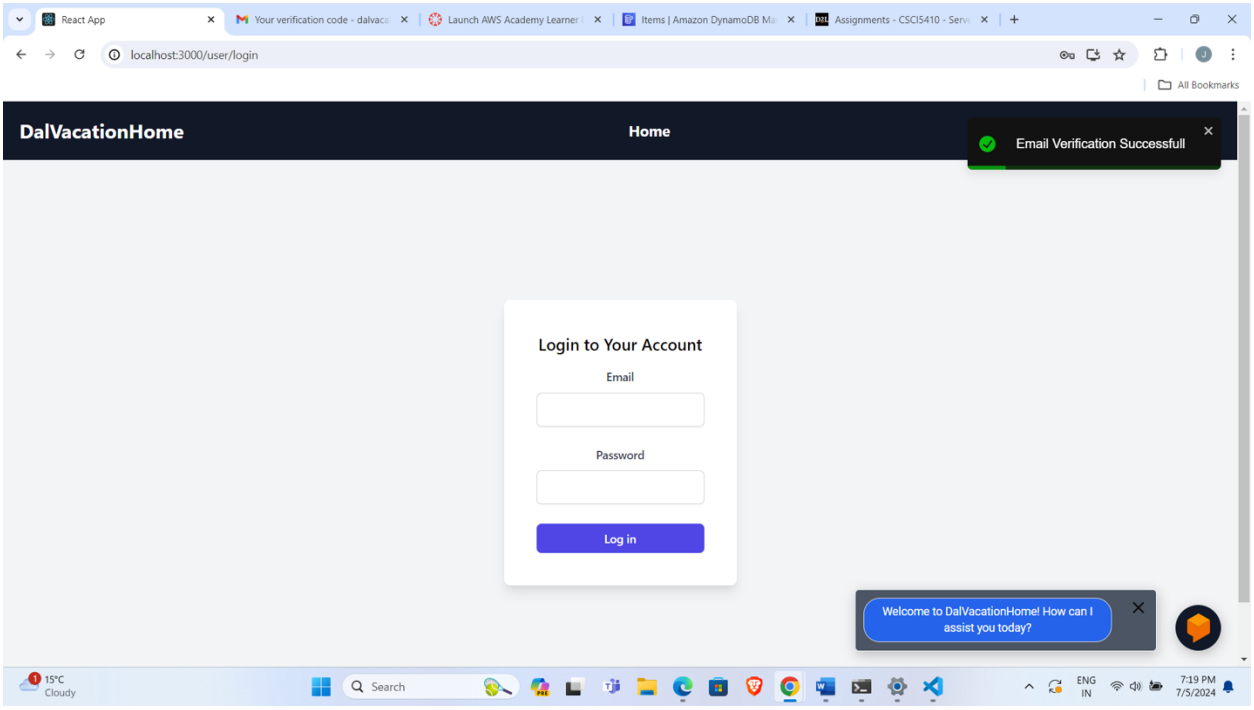

Figure 46: Login screen capture, featuring fields for email and password.

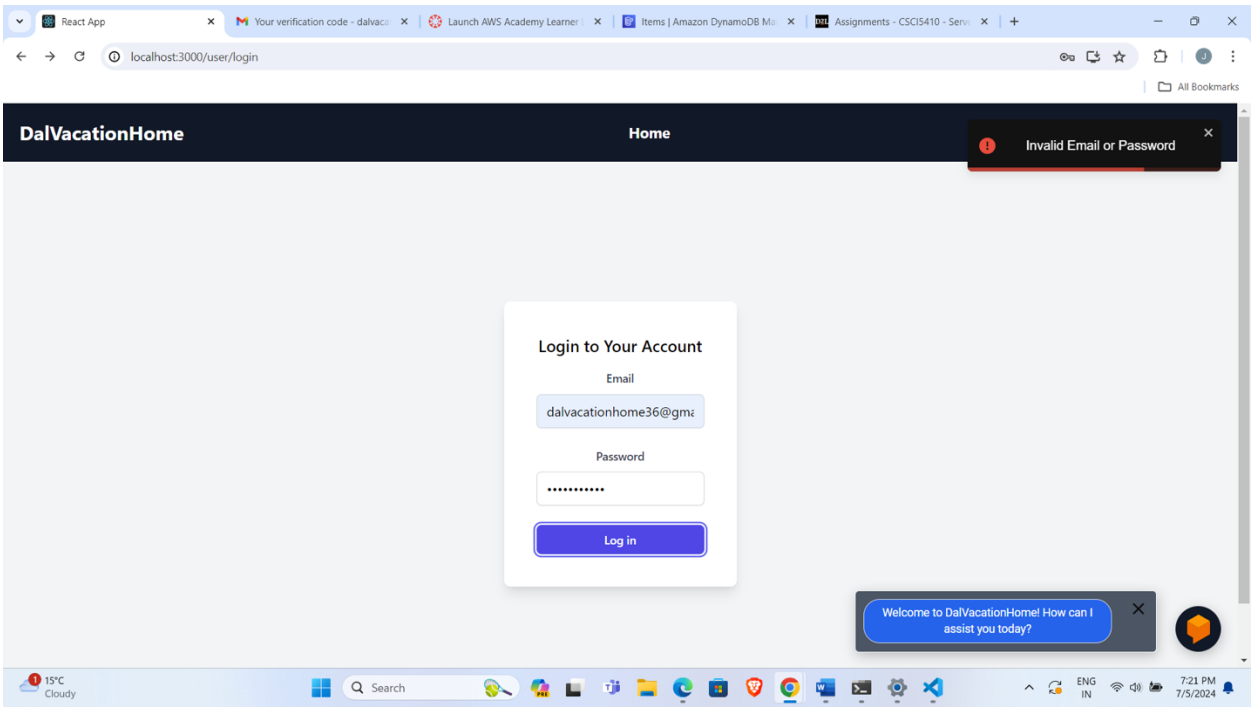

Figure 47: Attempt to log in showing the screen during the process.

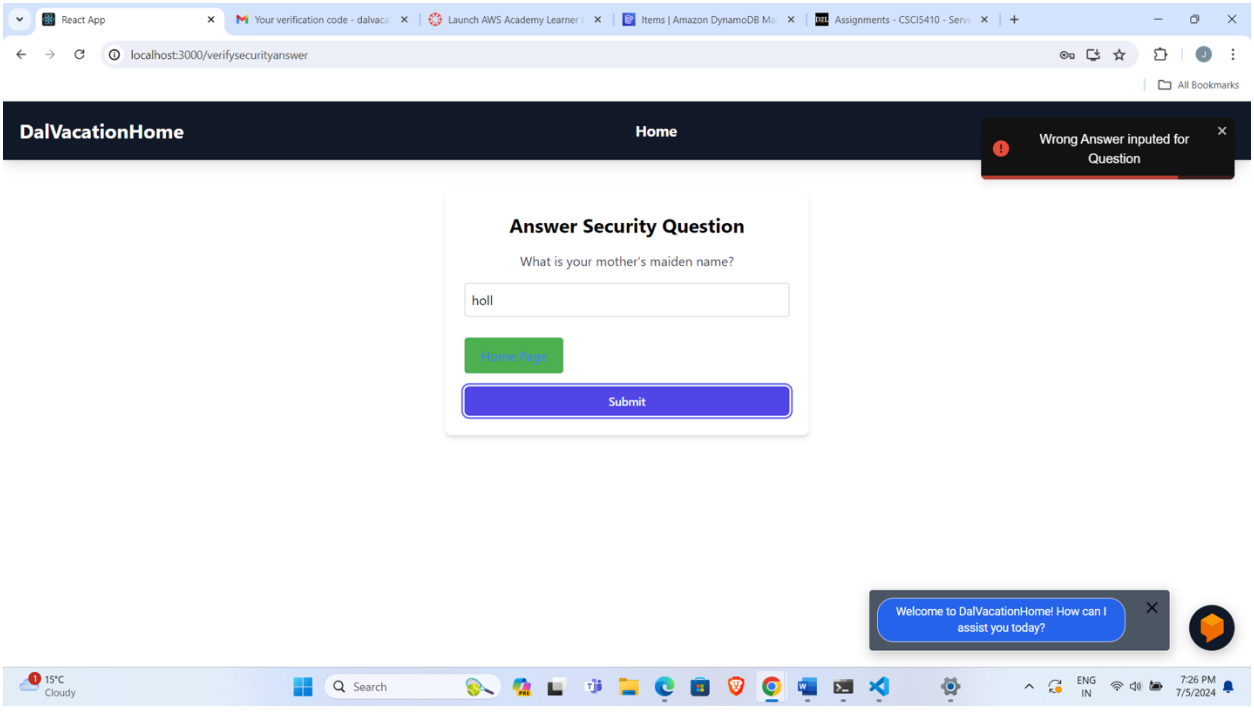

Figure 48: Security question prompt during login for additional verification.

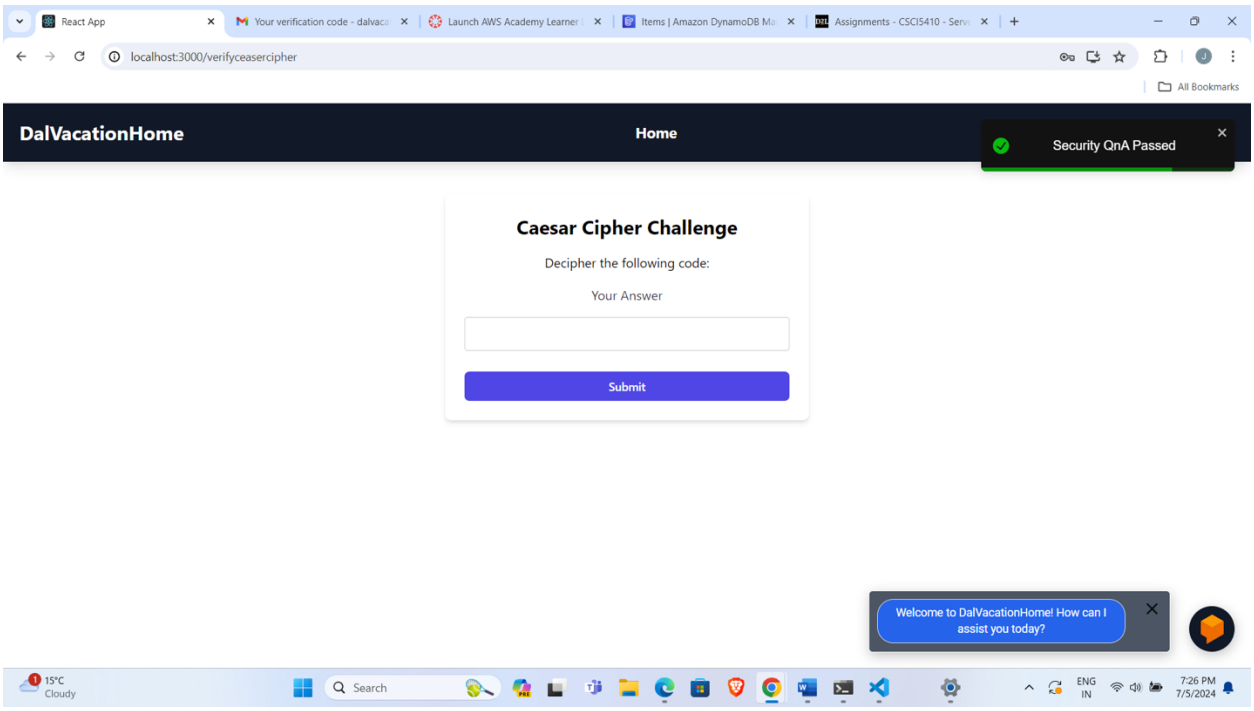

Figure 49: Caesar cipher challenge presented to the user as a login verification step.

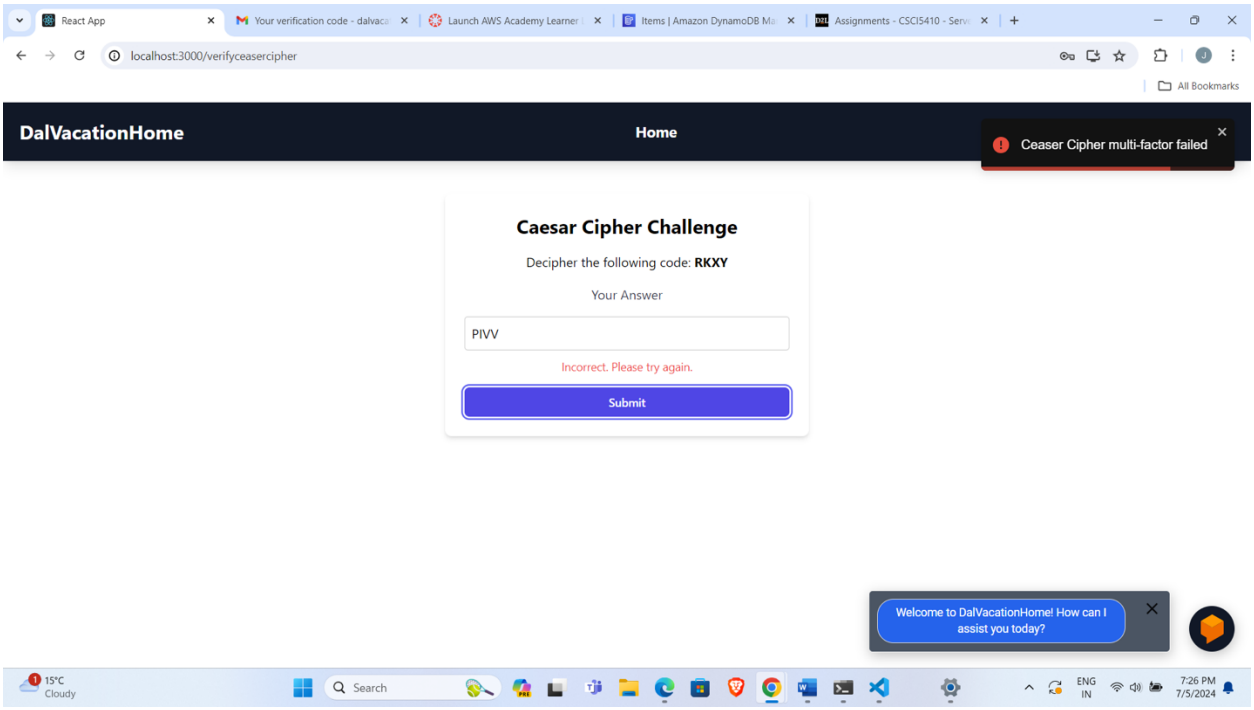

Figure 50: Process of verifying the decrypted text for login authentication.

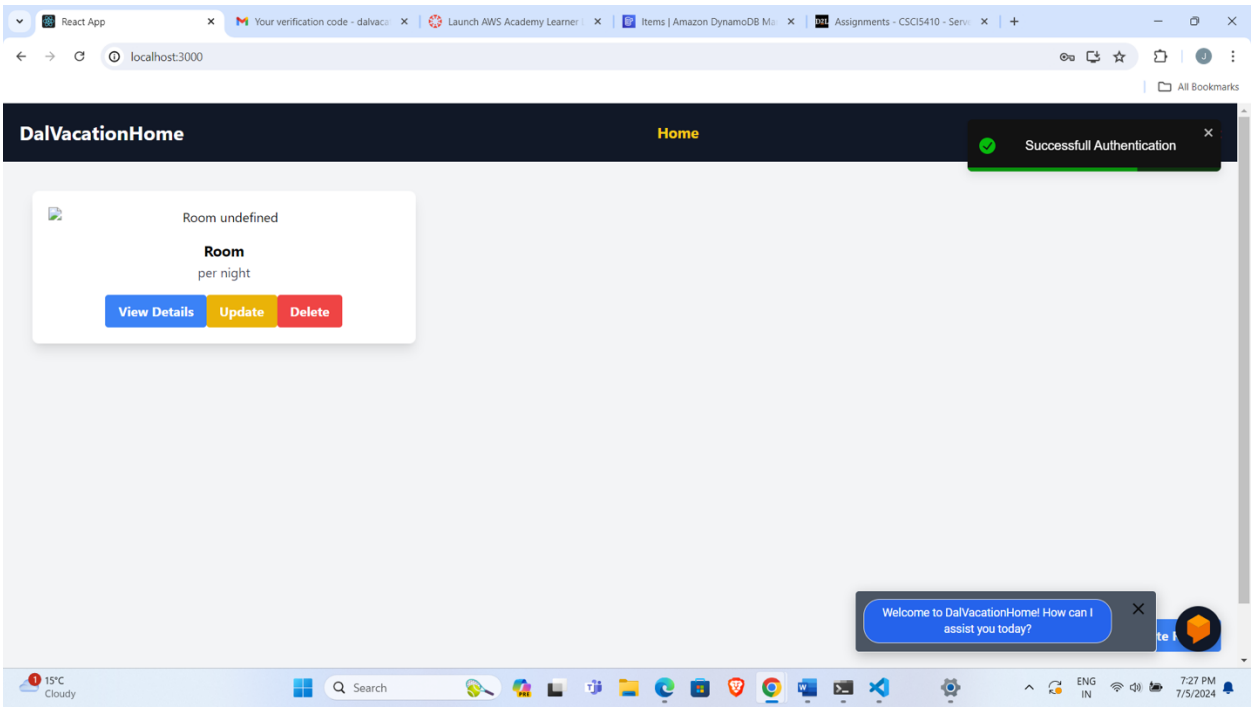

Figure 51: Homepage of DalVacationHome displaying available rooms and the chatbot interface.

General Module: Room Management functionality for application:

Property Agent Side:

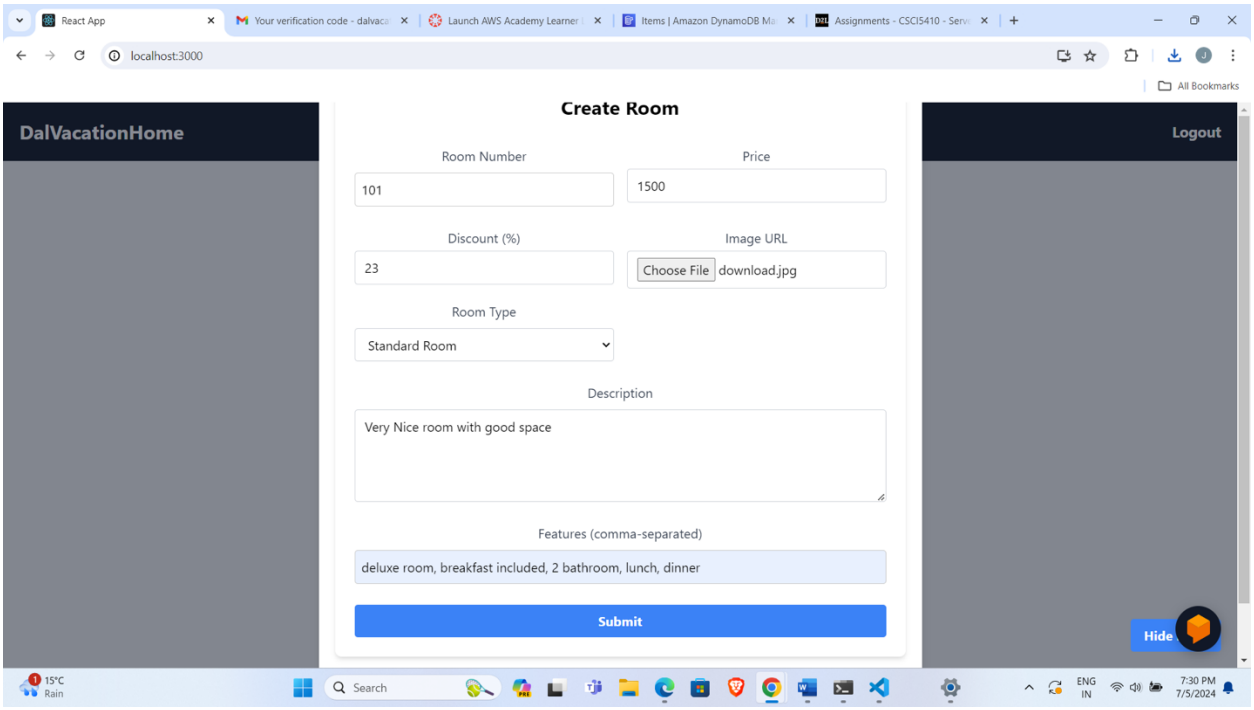

Figure 52: Create Room form used by property agents to list new rooms.

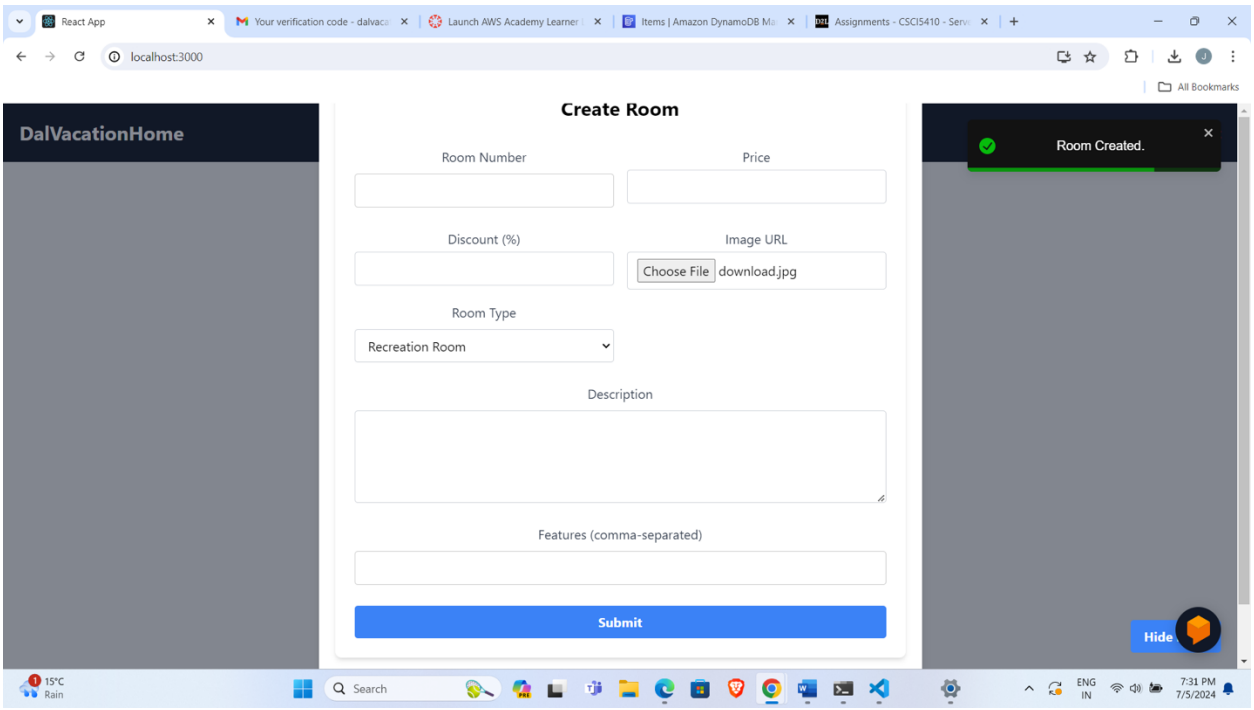

Figure 53: Detailed view of the Create Room form with fields for room details and image upload.

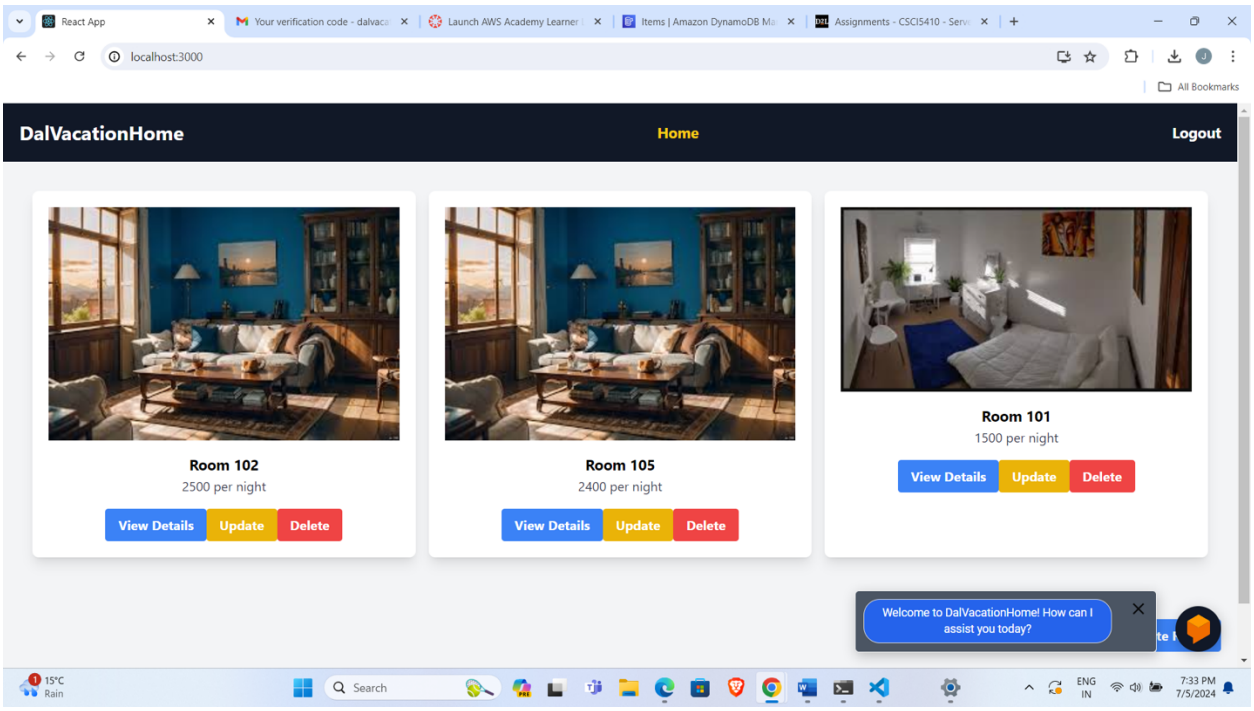

Figure 54: Overview of room listings with options to view details, update, and delete.

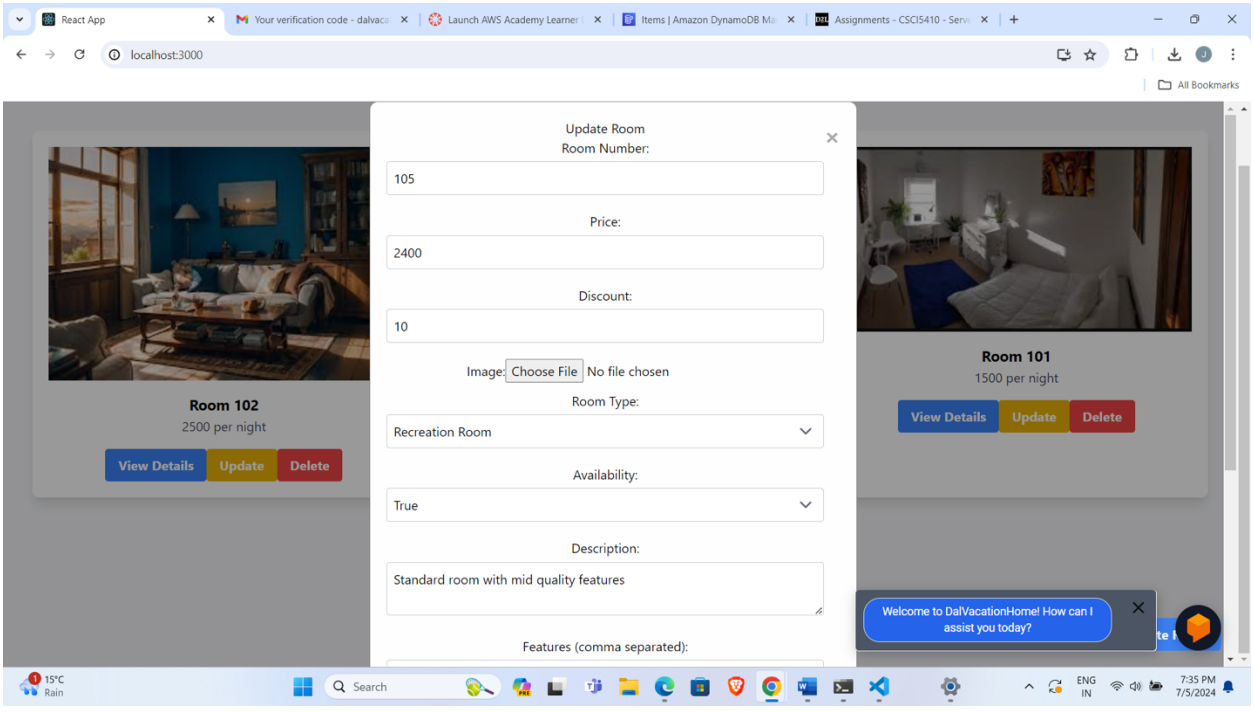

Figure 55: Update Room form pre-filled with existing room details for modification.

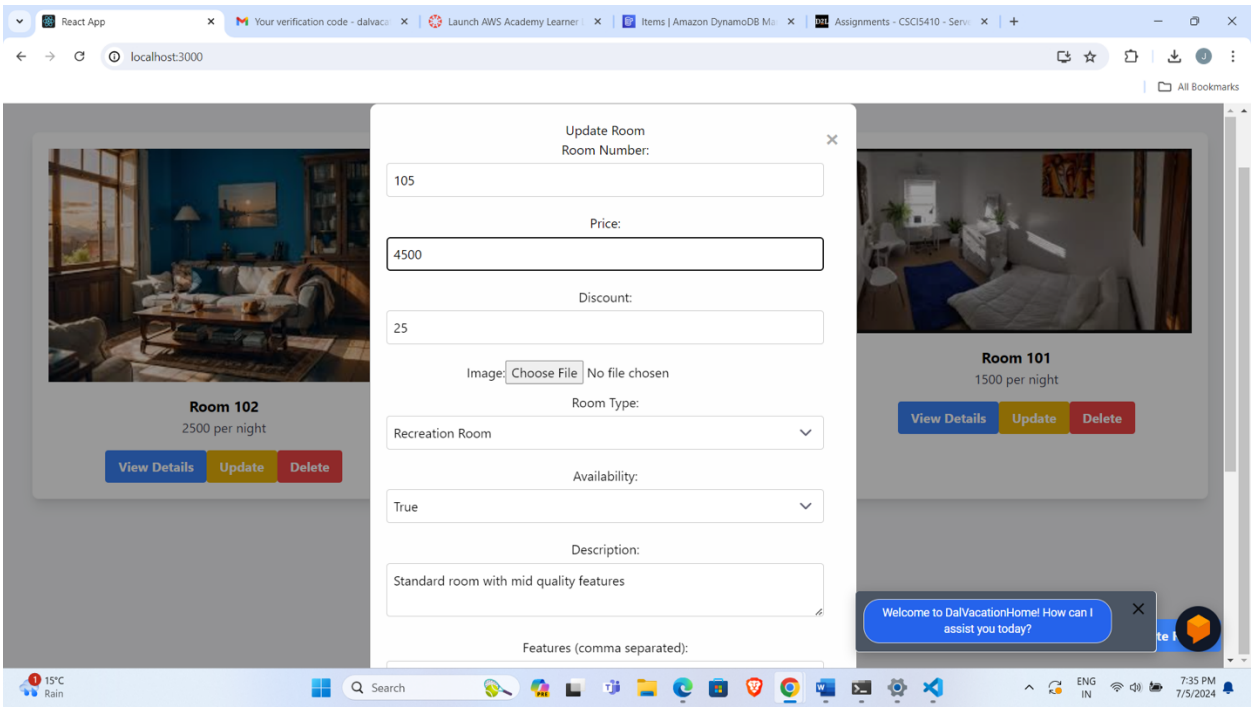

Figure 56: Process of temporarily removing an image from a room's listing.

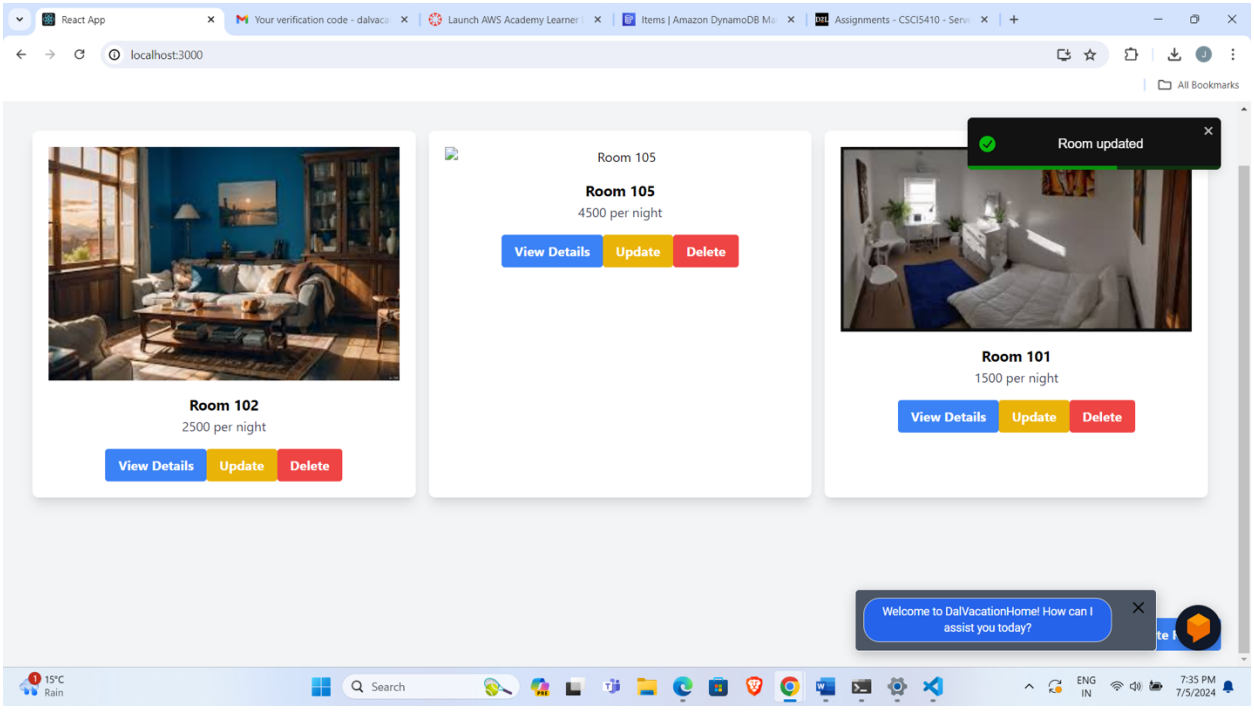

Figure 57: Updated room details shown without the previously attached image.

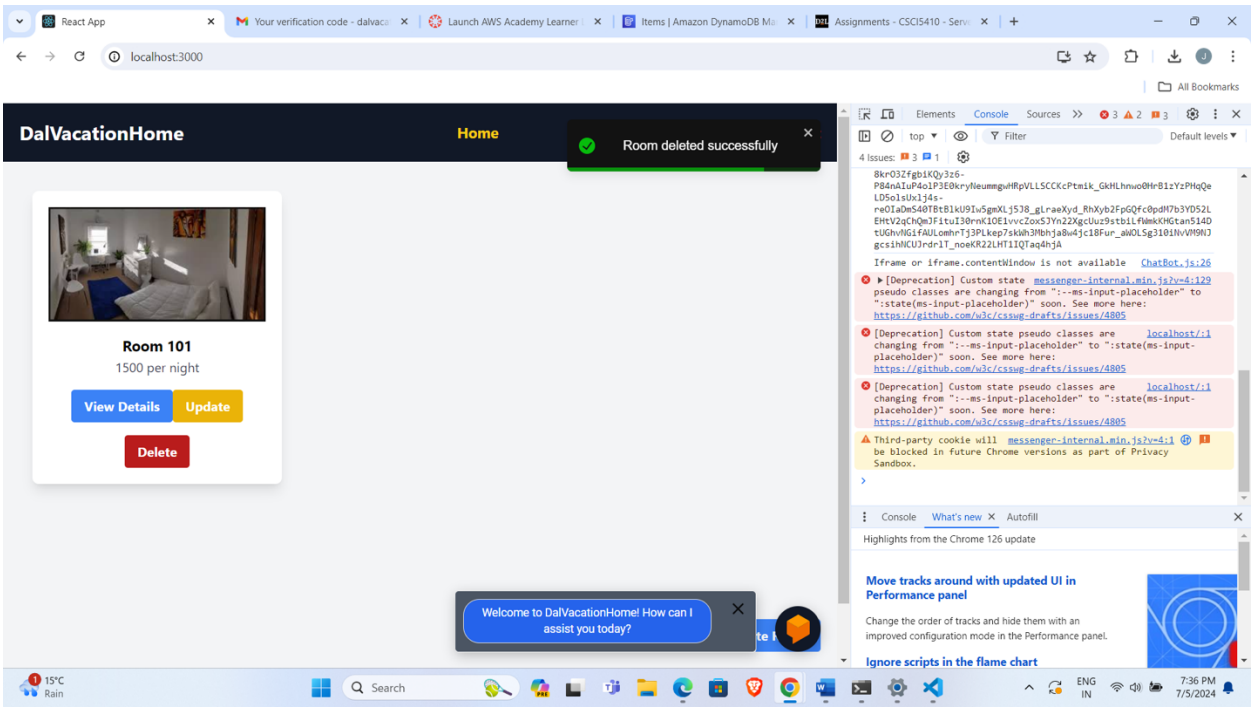

Figure 58: Screenshot of the deletion process for a room.

Customer Side:

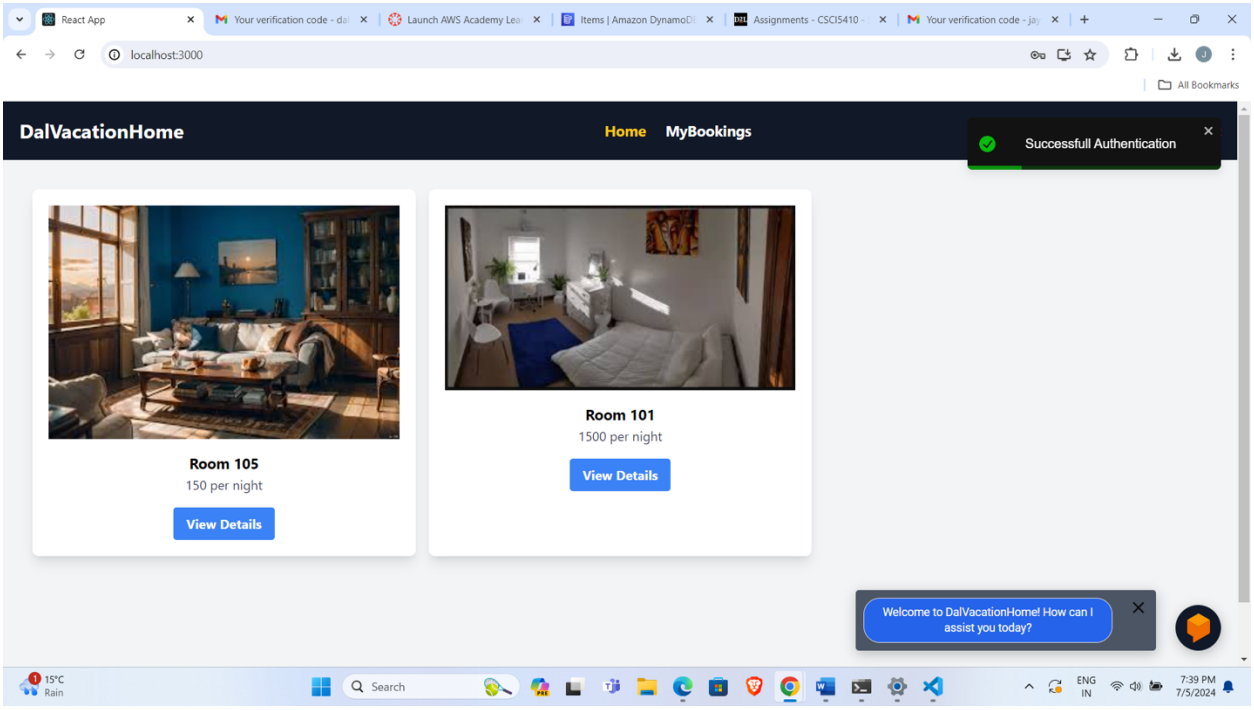

Figure 59: Customer view showing available rooms for booking.

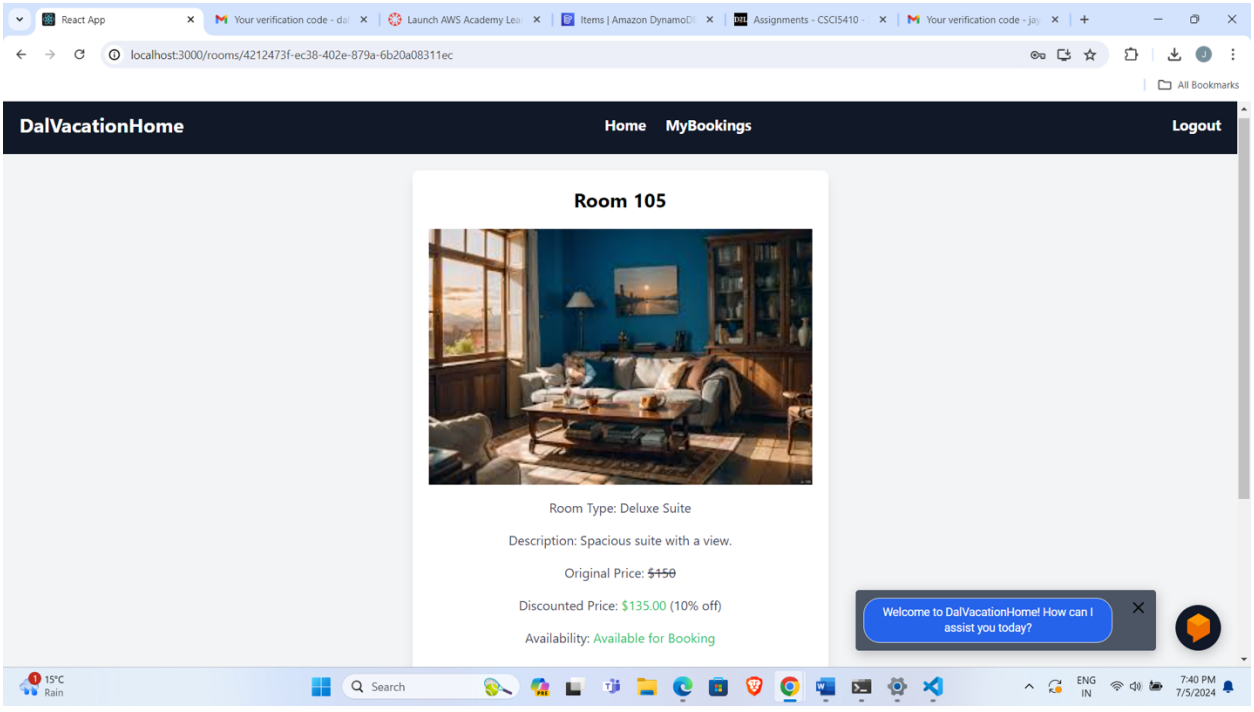

Figure 60: Detailed view of a room from the customer's perspective.

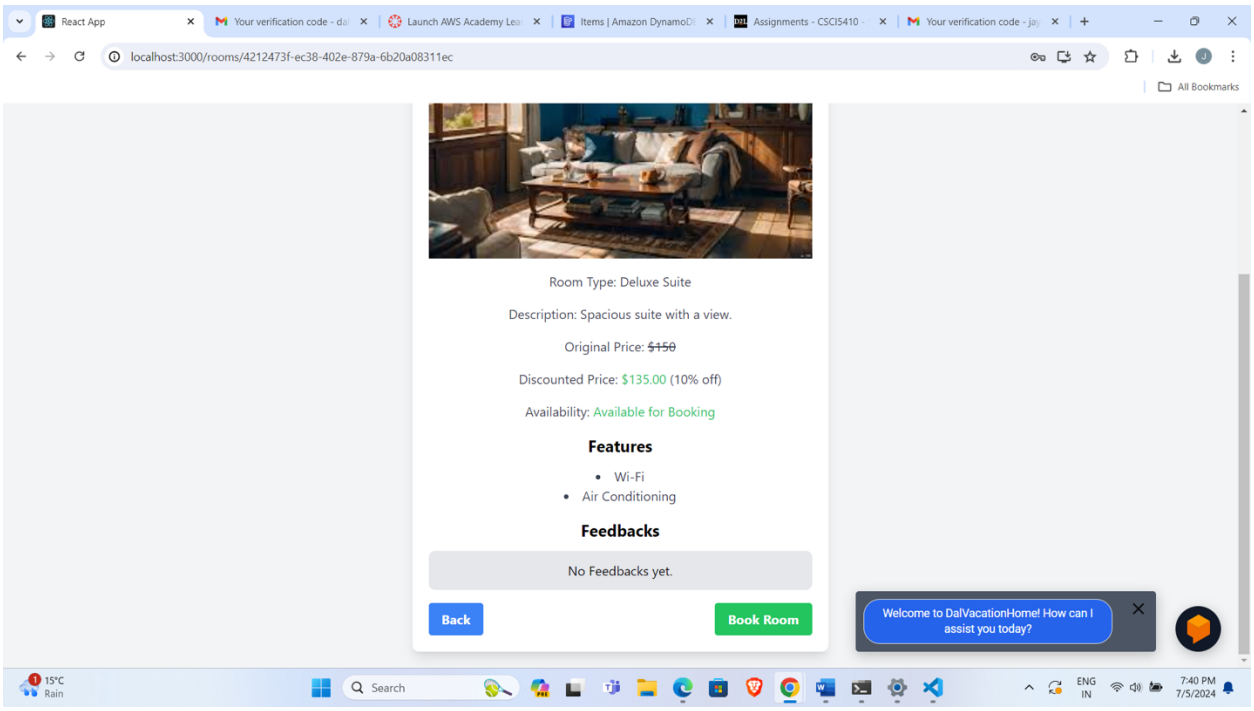

Figure 61: Room details display for a customer with the option to book the room.

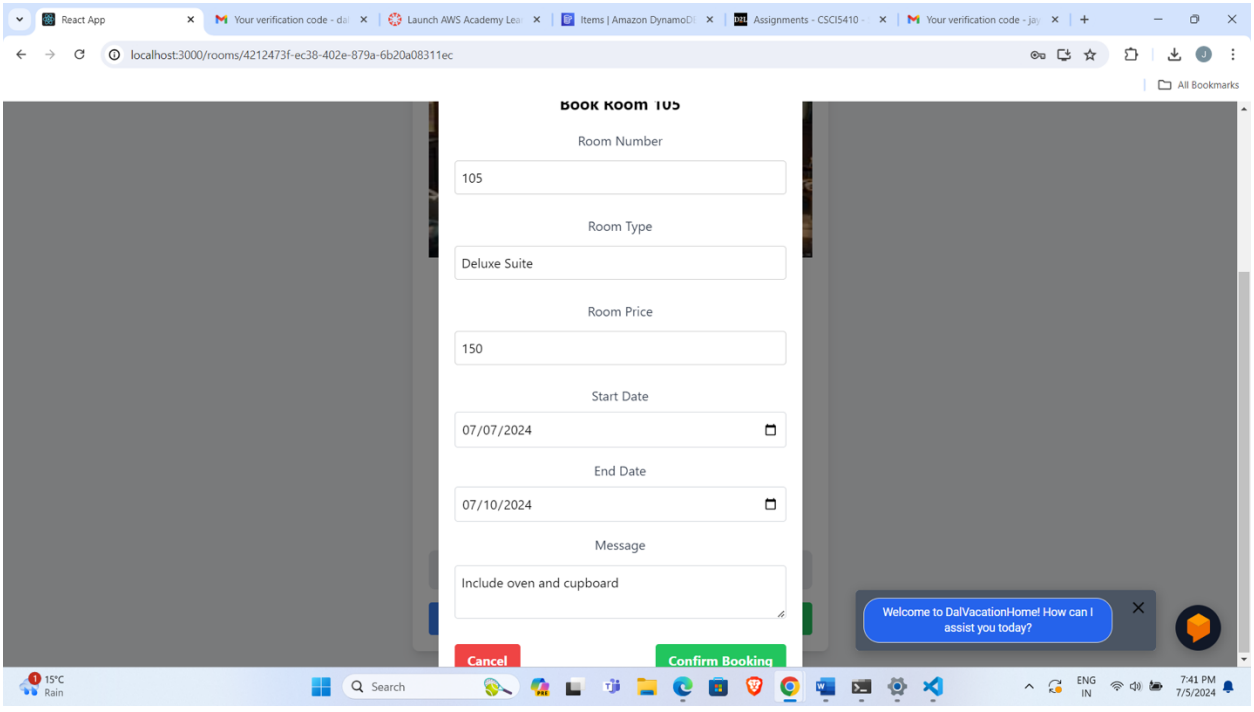

Figure 62: Booking form as filled out by a customer to reserve a room.

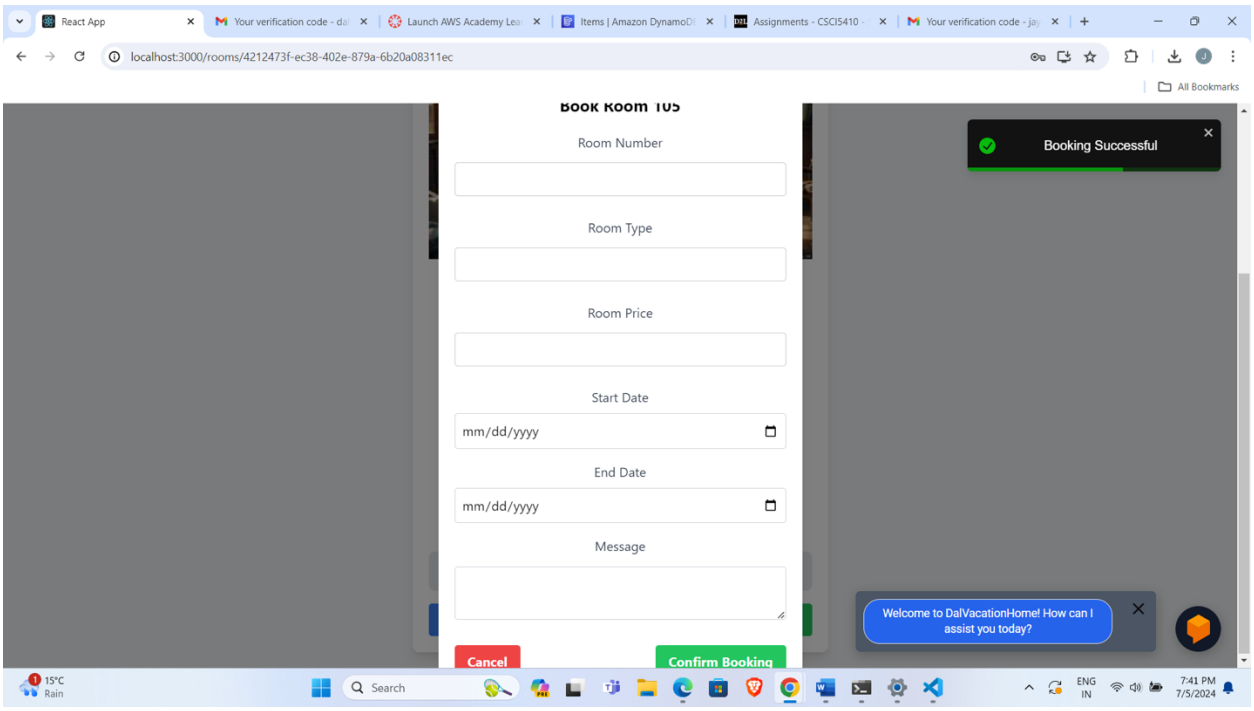

Figure 63: Confirmation screen showing successful room booking.

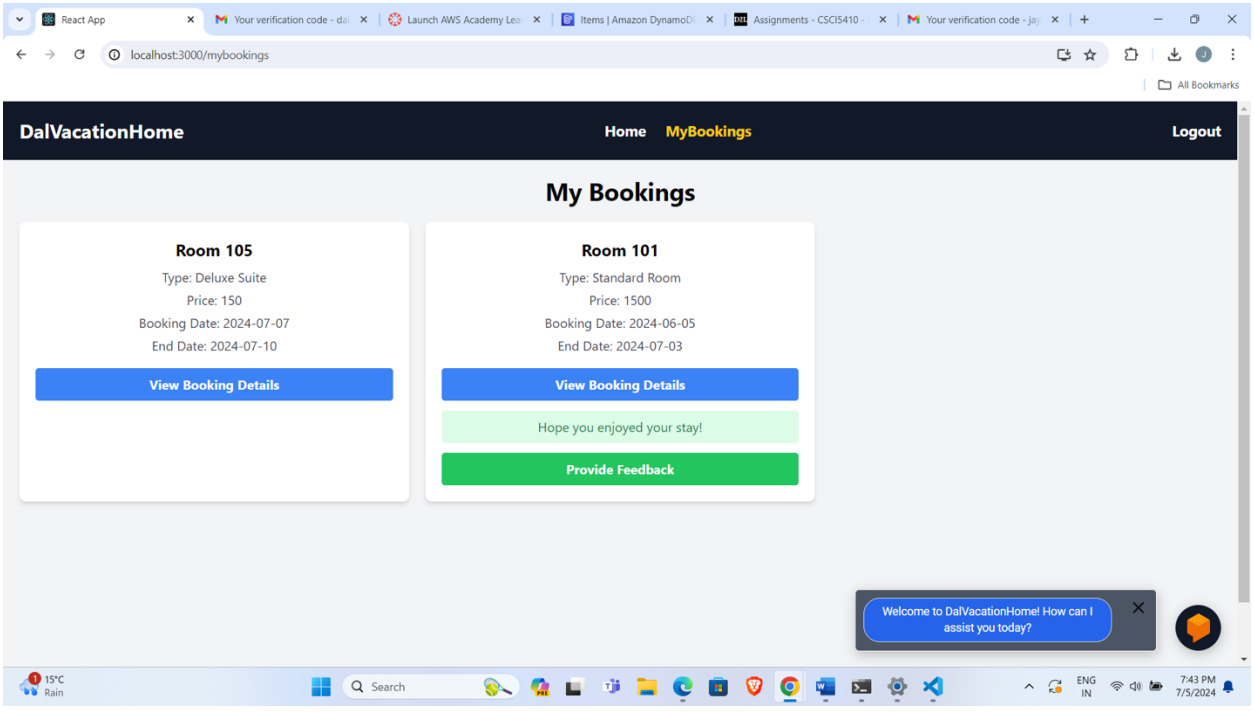

Figure 64: MyBookings page displaying all of a customer's room bookings.

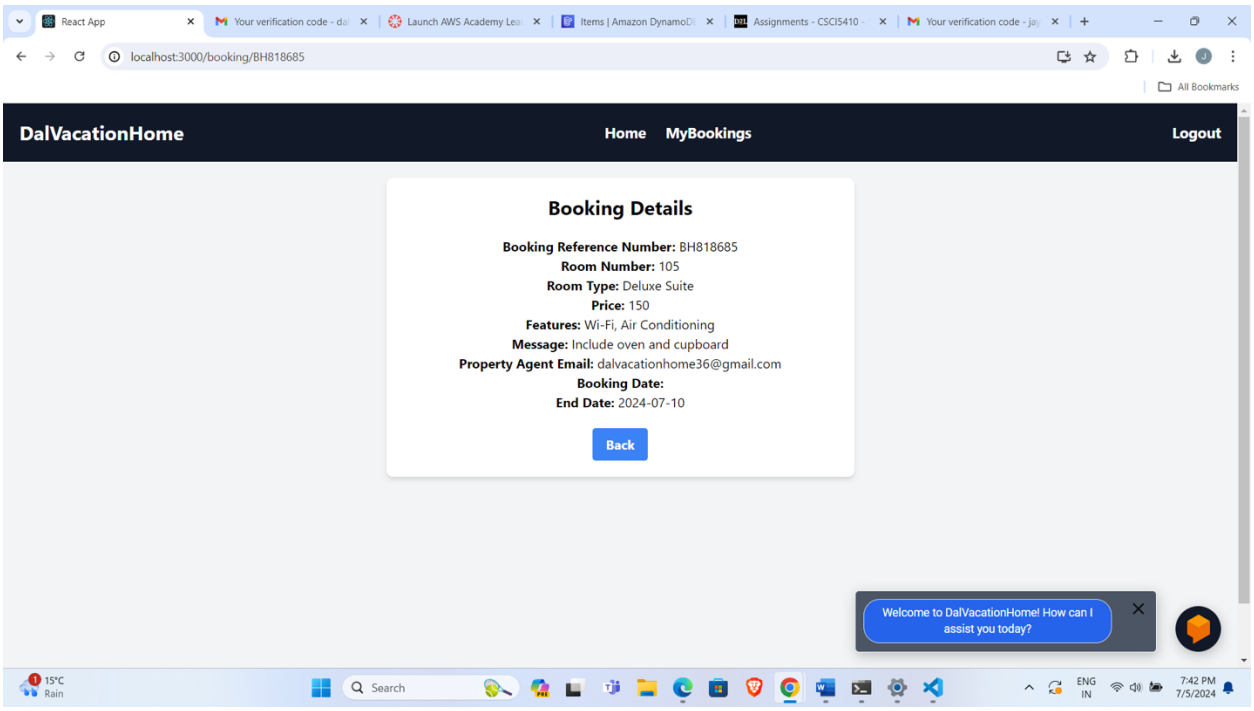

Figure 65: Detailed view of a specific booking from the customer's MyBookings page.

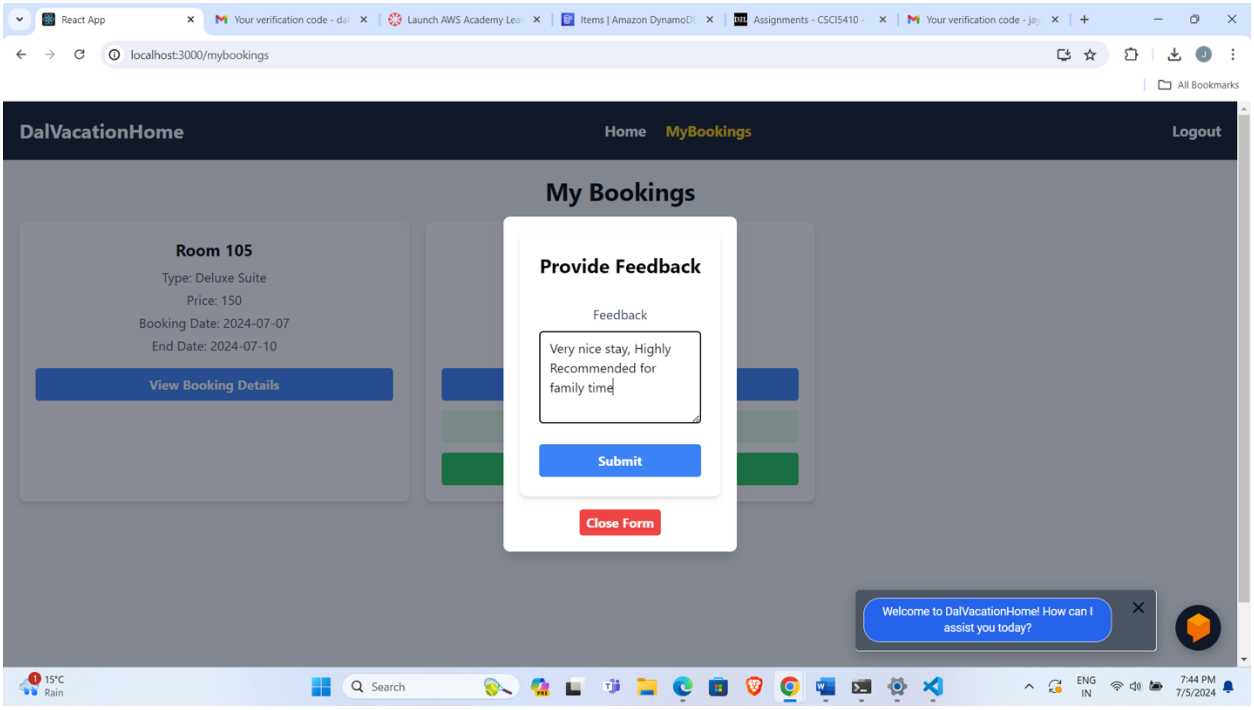

Figure 66: Process of a customer providing feedback for a room.

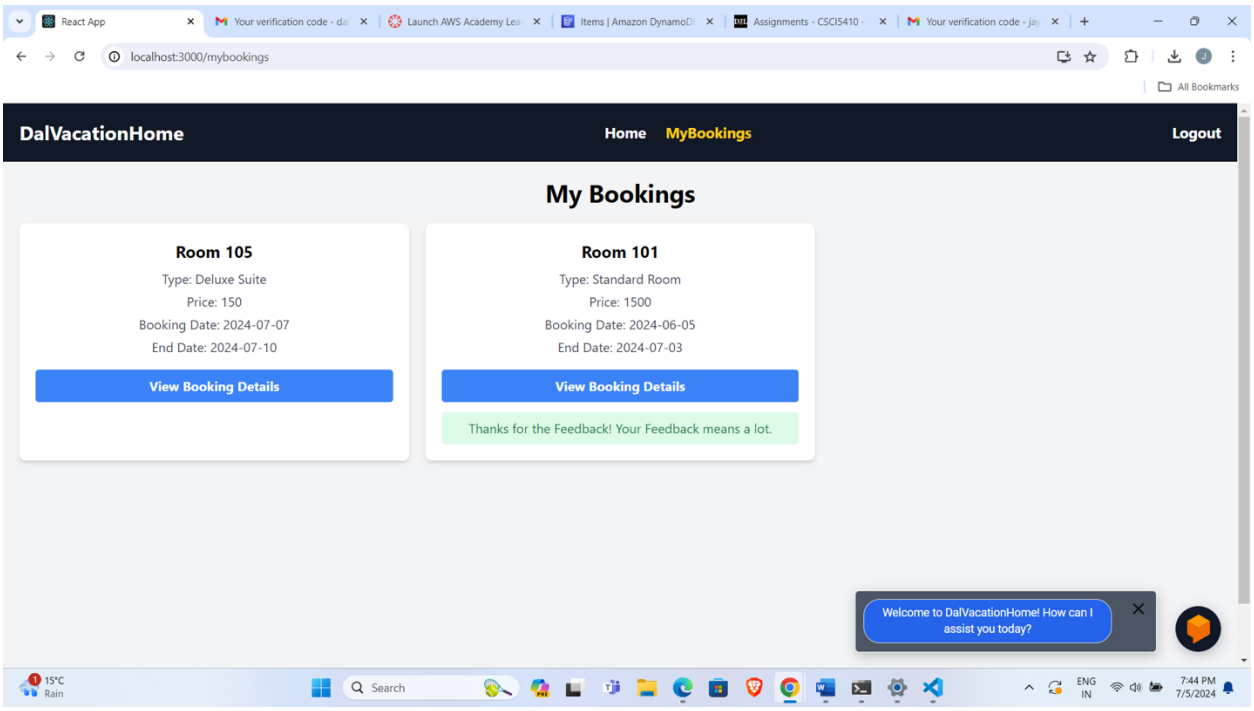

Figure 67: Feedback submission confirmation and display on the room's page.

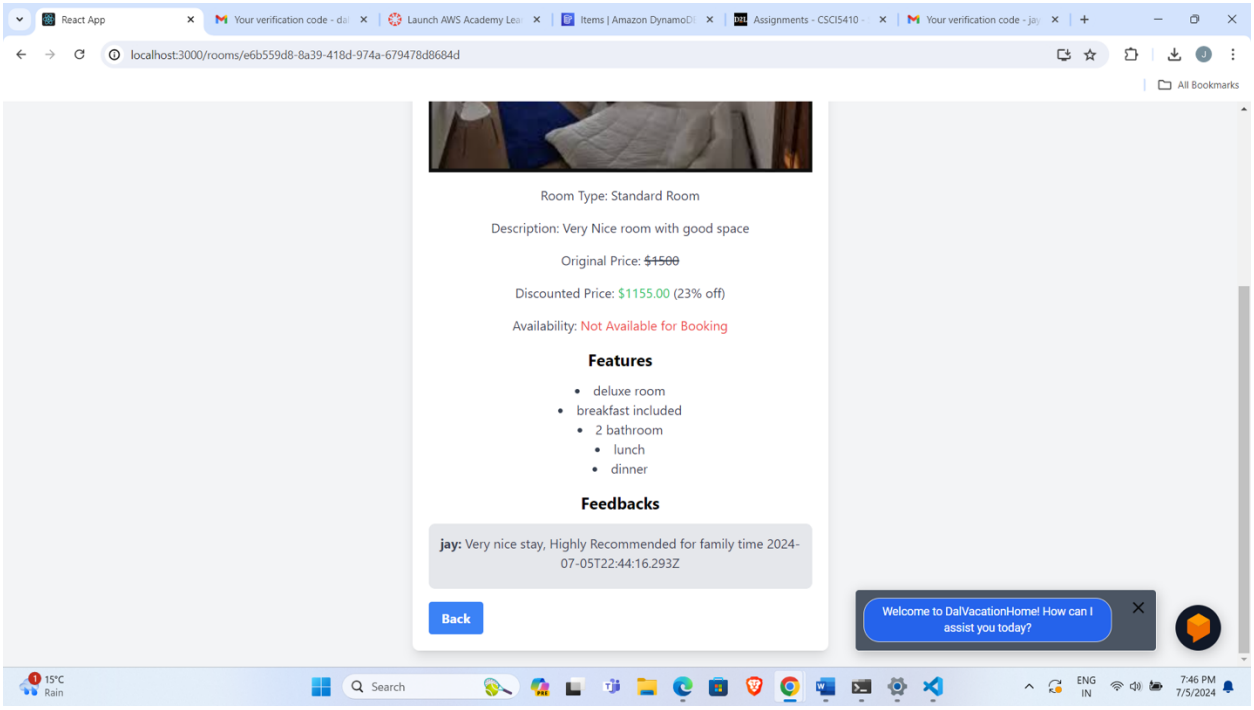

Figure 68: Room details page showing customer feedback.

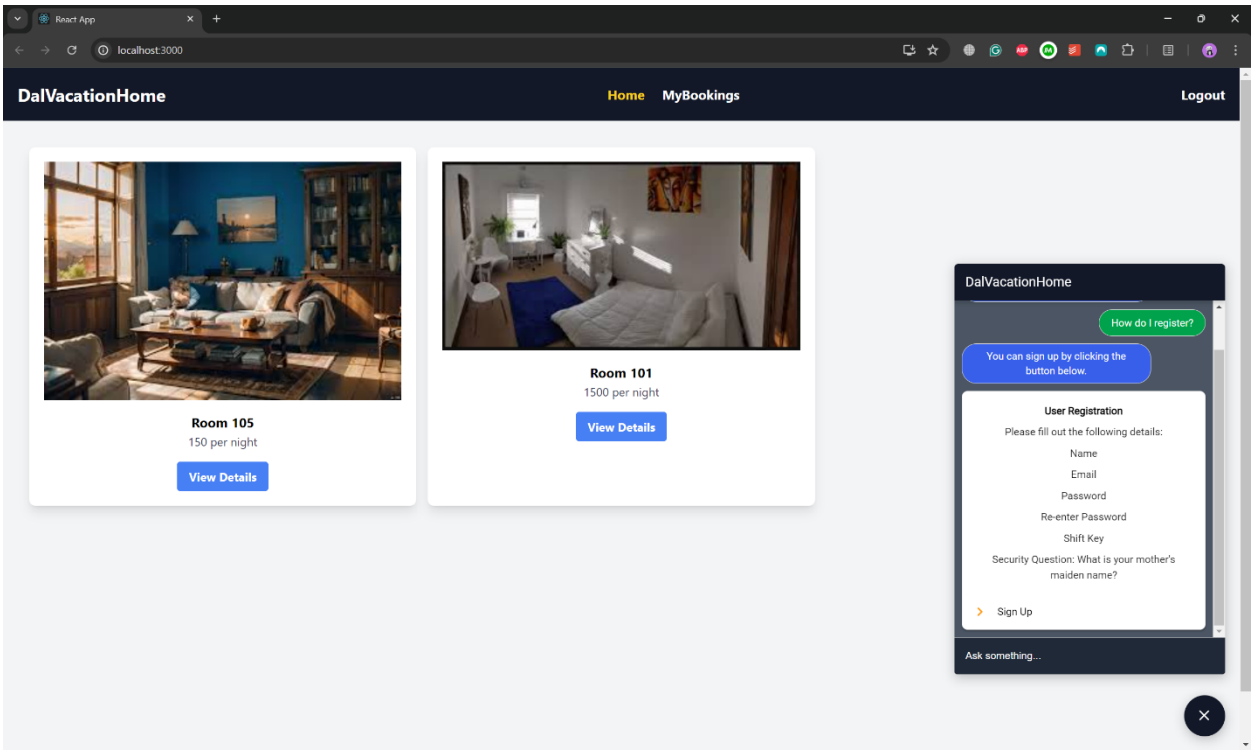

Figure 69: Chatbot interface assisting users with application functionalities (how to register).

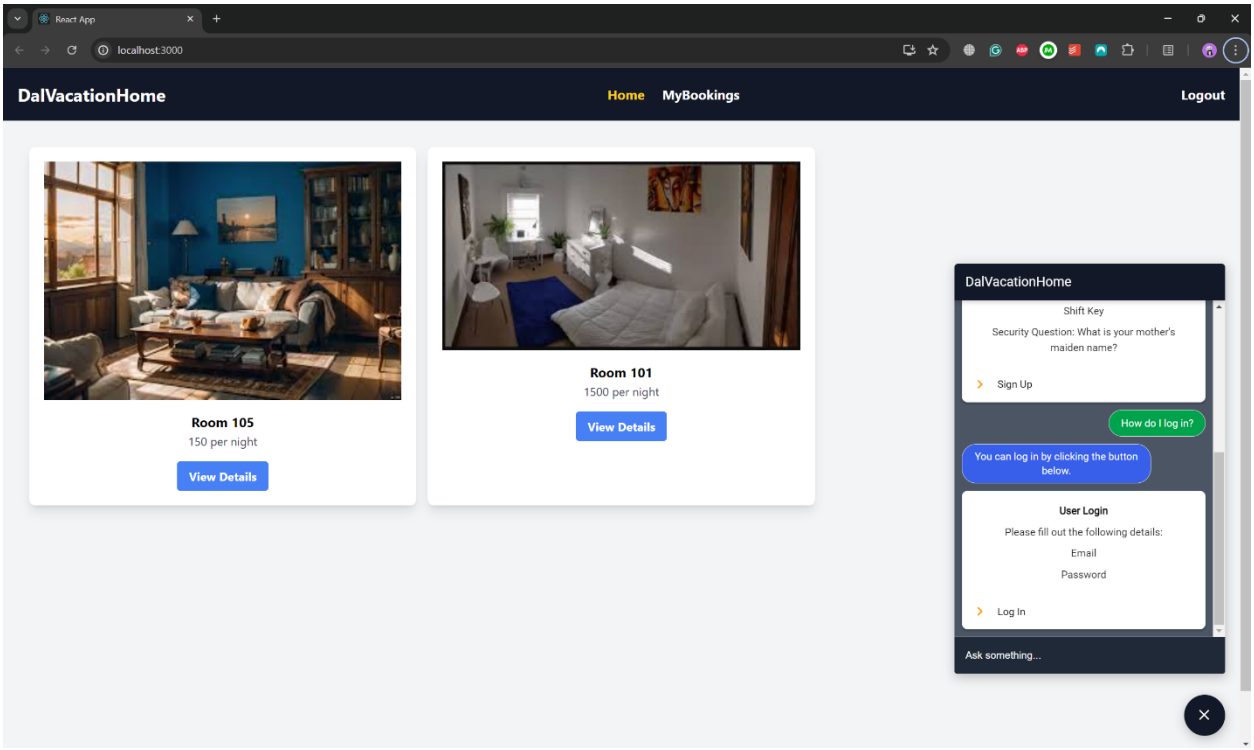

Figure 70: Chatbot interface assisting users with application functionalities (how to login).

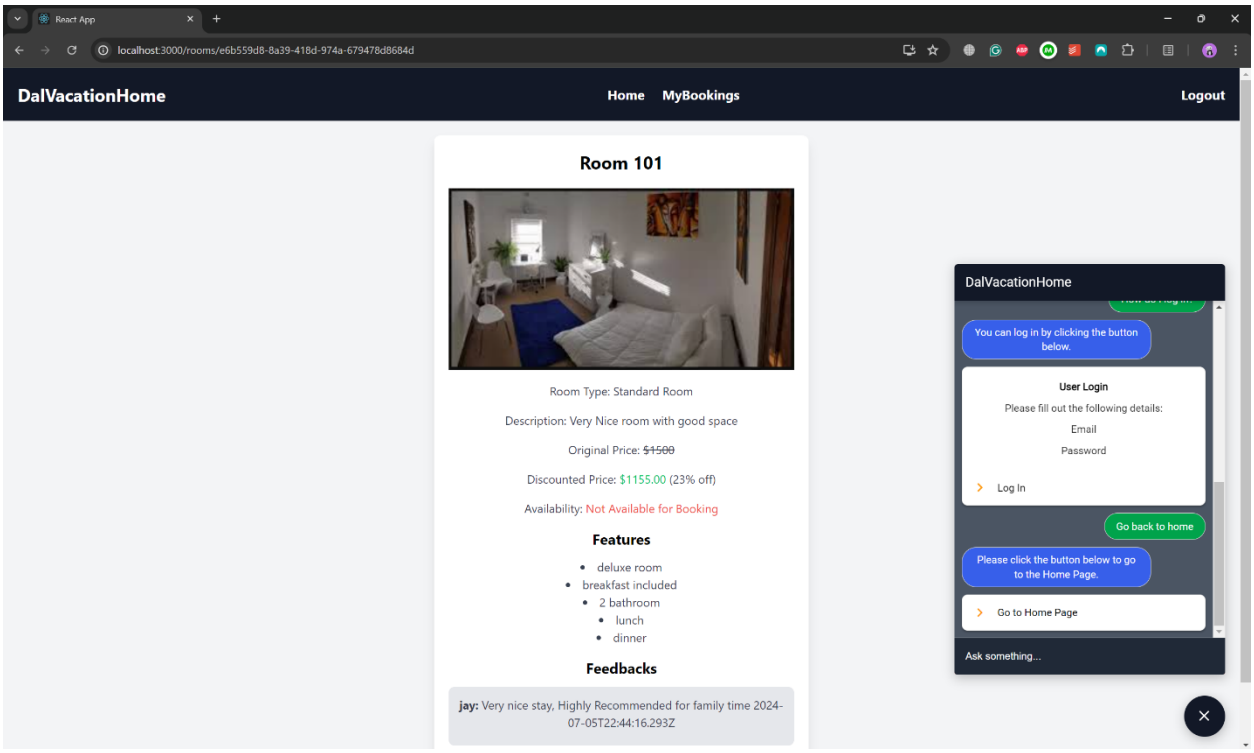

Figure 71: User navigating between different pages of the application via chatbot commands.

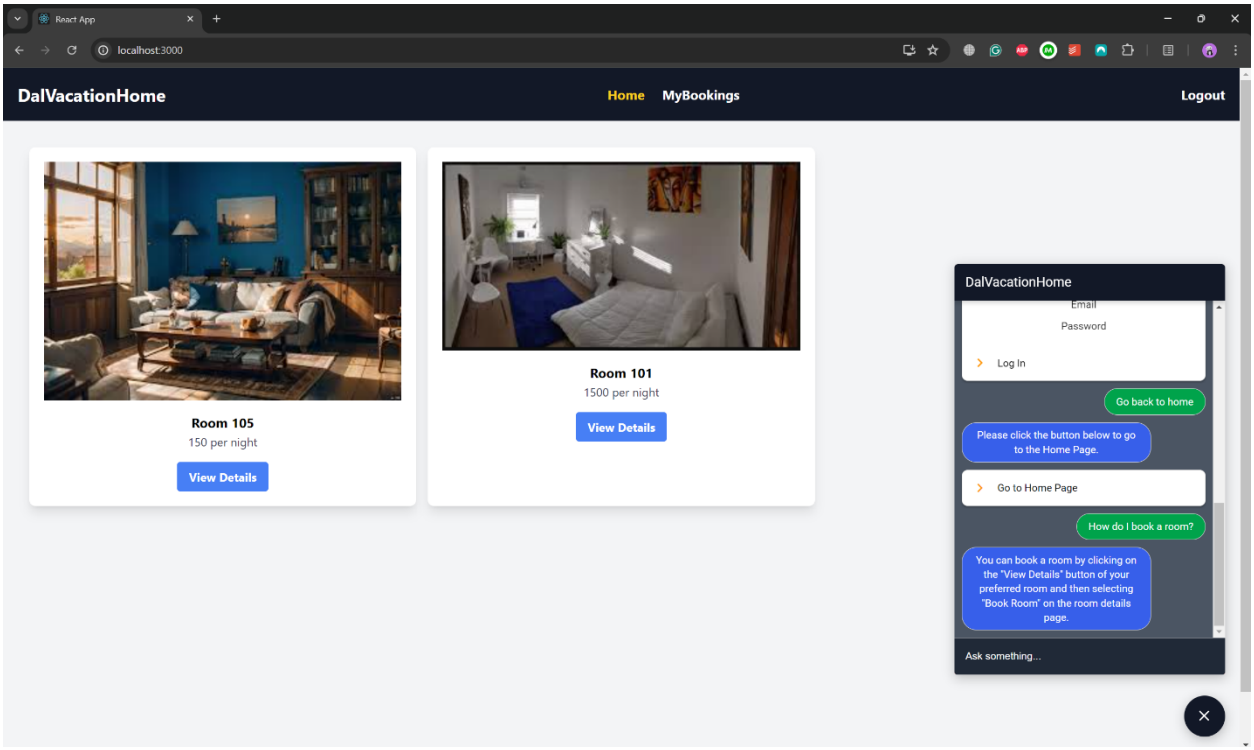

Figure 72: User interaction with the chatbot to ask various questions about the application.

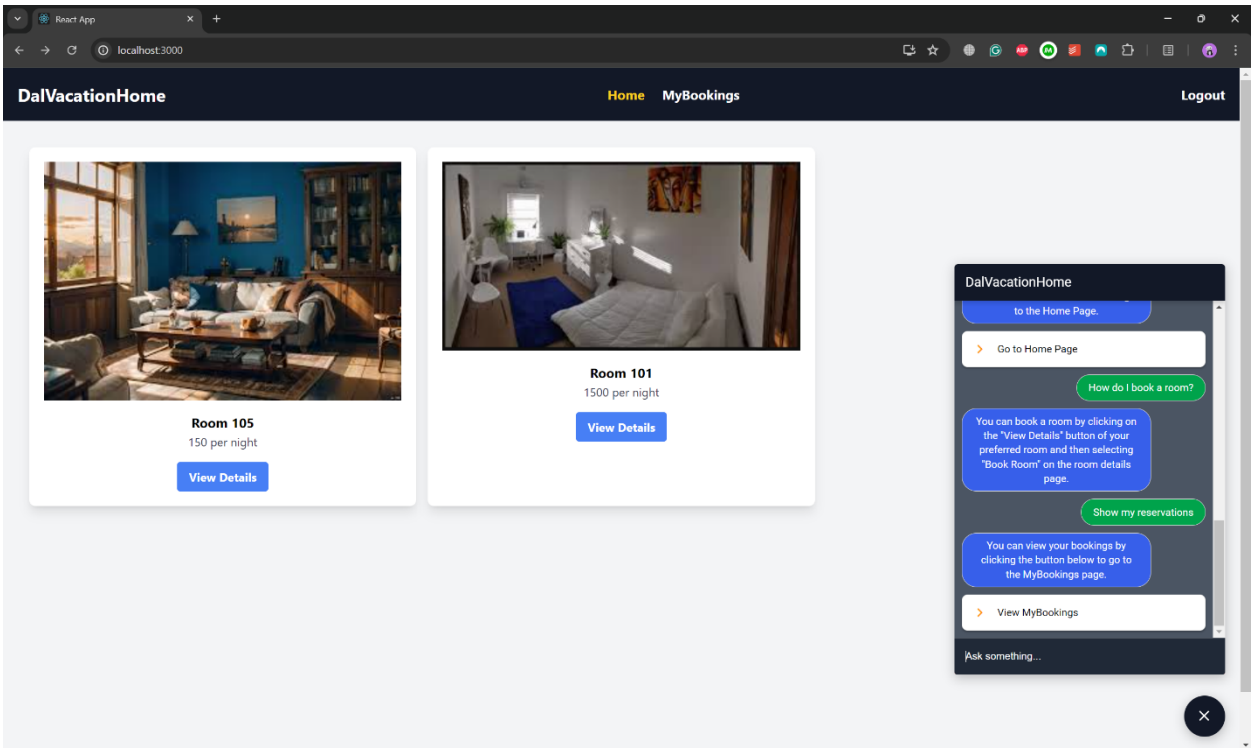

Figure 73: User querying the chatbot about their previous room bookings.

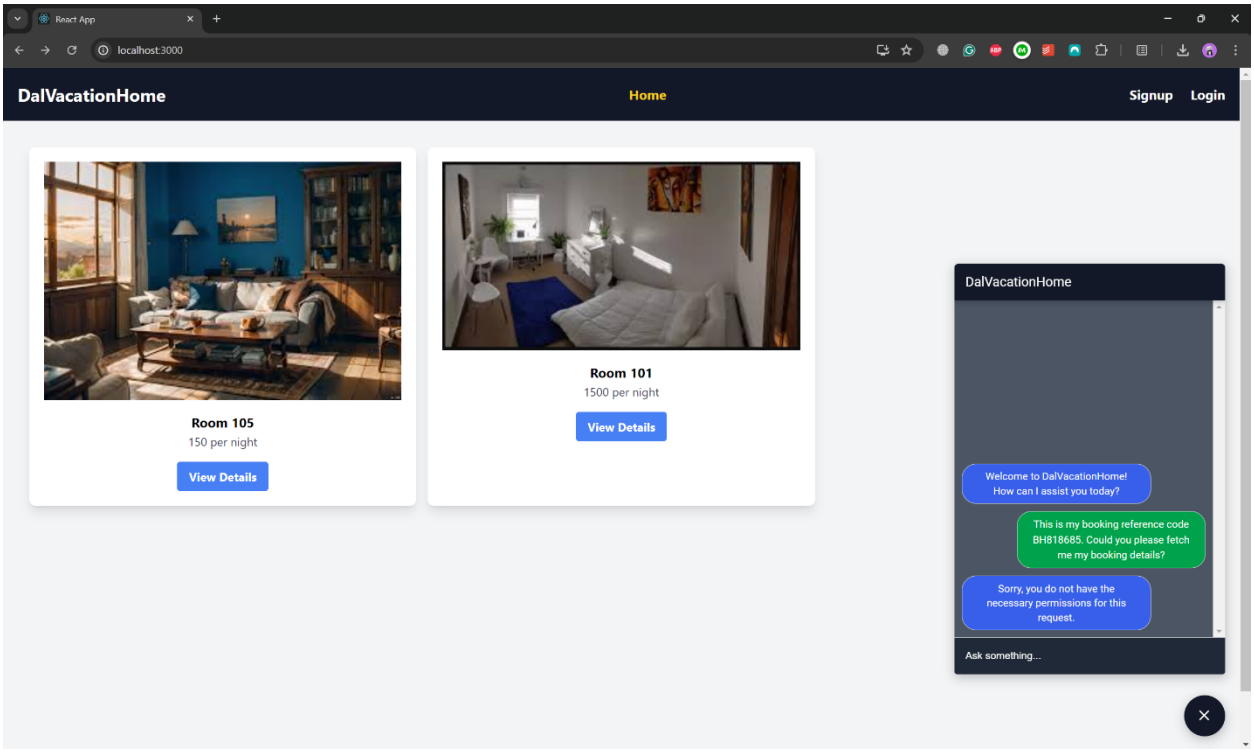

Figure 74: Guest without login inquiring in the chatbot about details of a specific booking.

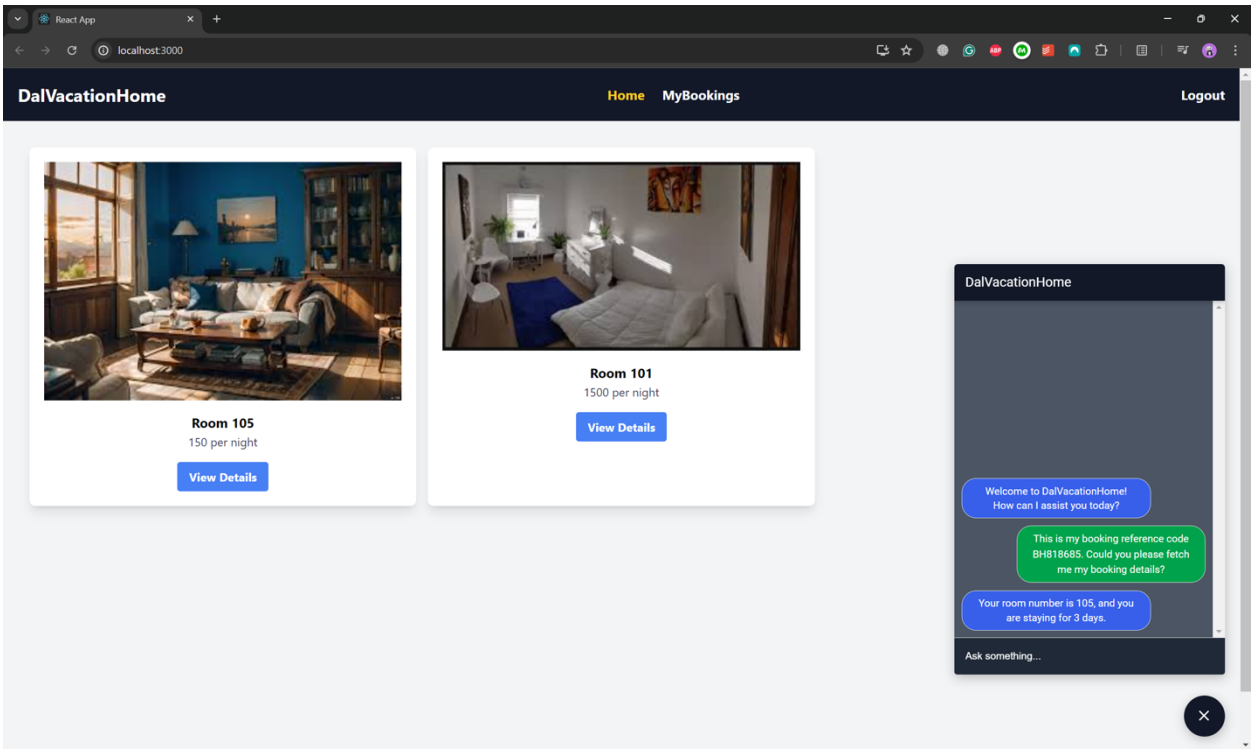

Figure 75: Customer inquiring in the chatbot about details of a specific booking.

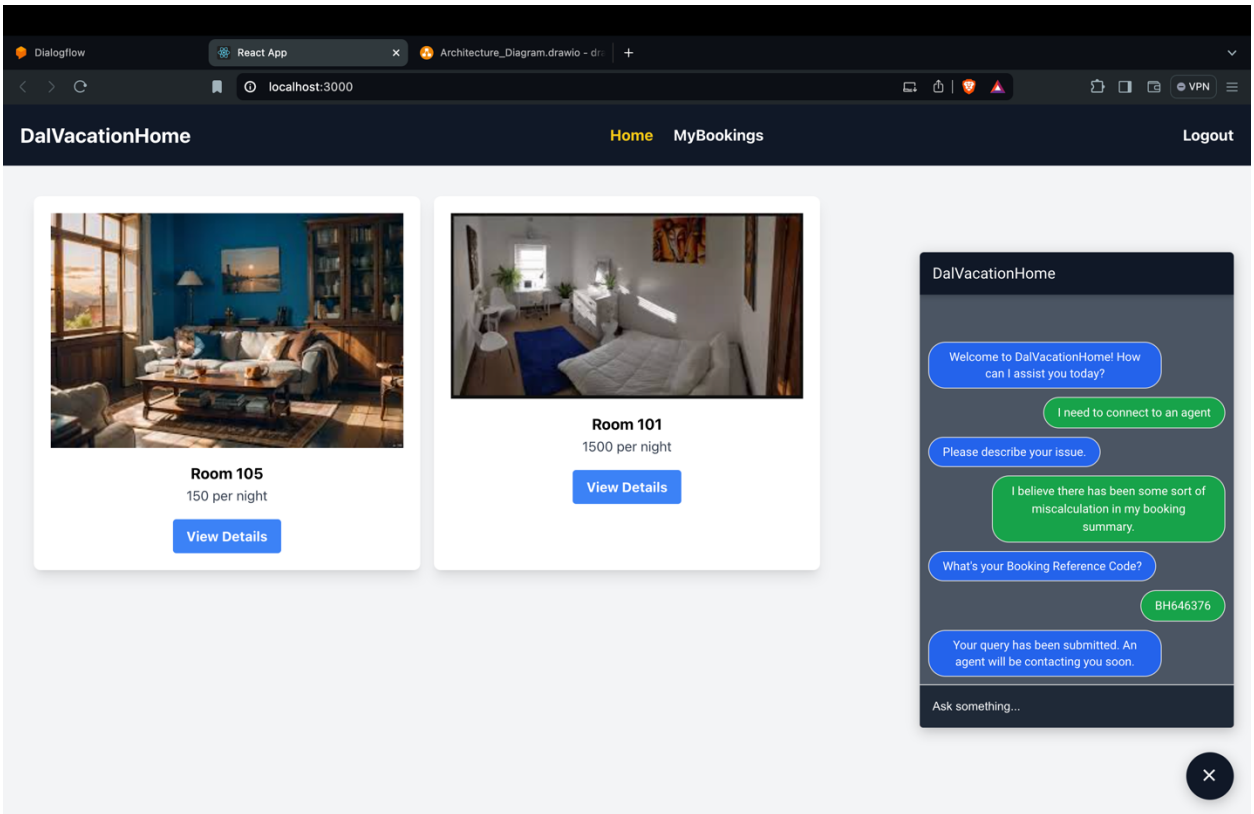

Figure 76: Customer requesting to connect to an agent.

Meeting Logs:

Agenda	Outcome of discussion	Link to meeting recordings
• Discuss how to divide by the module. • Plan regular GitLab commits. • Define project structure.	• Decided to divide the first two modules between two people per module. • Planned to discuss the division of later modules later. • Emphasized the importance of regular commits in GitLab. • Defined the project structure to ensure a consistent approach.	Call with Serverless Team 36-20240604_191112-Meeting Recording.mp4
• Discuss the unavailability of AWS Lex. • Decide on using Dialogflow for the virtual assistant. • Formulate questions for TA. • Assign later modules. • Discuss multi-cloud service allocation.	• Decided to use Dialogflow for the virtual assistant. • Formulated a set of questions to ask the TA. • Divided later modules among team members. • Decided which services should be in which account for multi-cloud implementation. • Assigned tasks for the Authentication and Virtual Assistant modules.	Meeting in Serverless Team 36-20240626_222308-Meeting Recording.mp4
• Discuss remaining tasks for Sprint 2. • Review Sprint 2 report requirements. • Plan bug fixing.	• Assigned remaining tasks for Sprint 2. • Reviewed and discussed how to approach the Sprint 2 report requirements. • Decided to compile all details worked on for the report. • Identified bugs in the application and planned necessary fixes.	Meeting in Serverless Team 36-20240702_181939-Meeting Recording.mp4
• Review and fix Sprint 2 report. • Finalize pending tasks.	• Reviewed the report and identified areas needing fixes. • Discussed what was remaining and completed the pending tasks.	Meeting in Serverless Team 36-20240705_224643-Meeting Recording.mp4

Chat Screenshots:

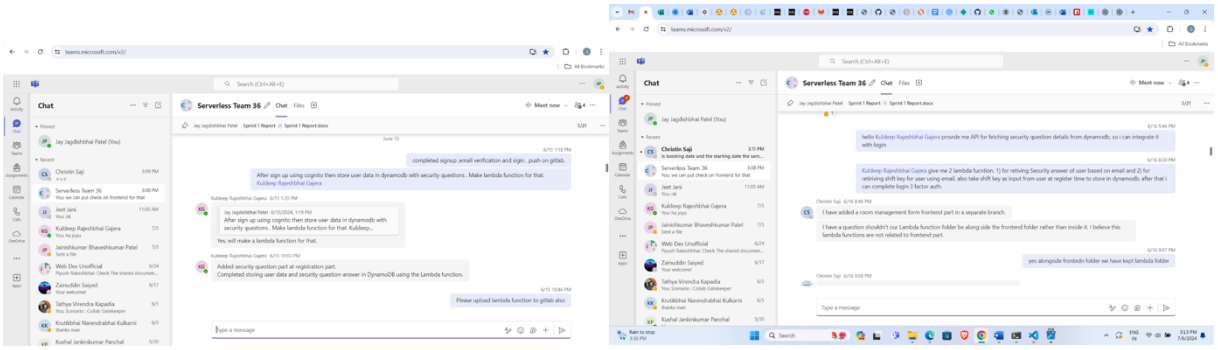

Figure 77: Team members discussing project updates and strategies in a team chat.

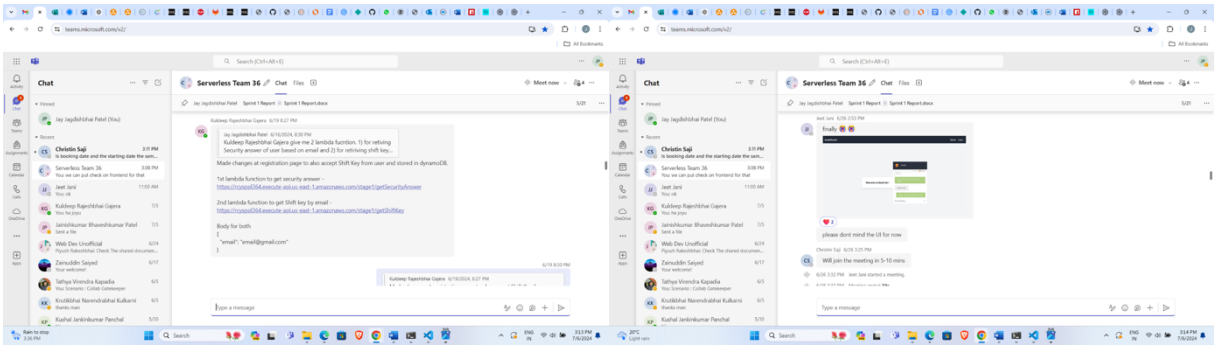

Figure 78: Team members discussing project updates and strategies in a team chat.

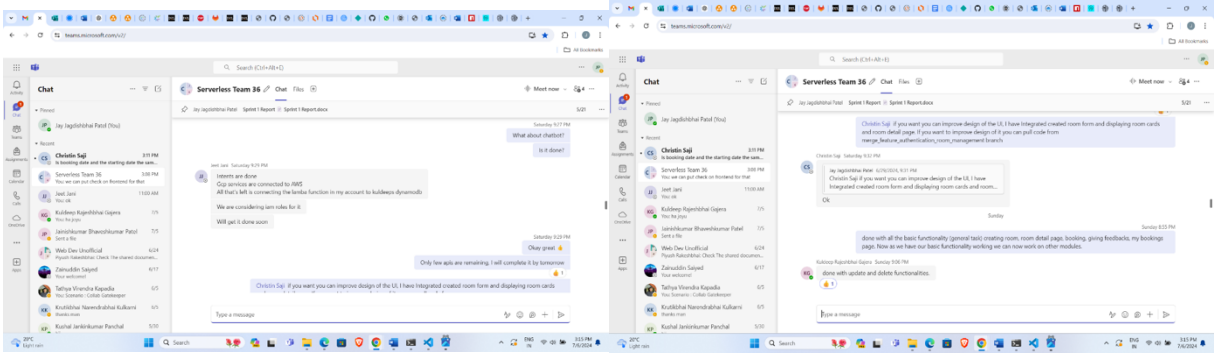

Figure 79: Team members discussing project updates and strategies in a team chat.

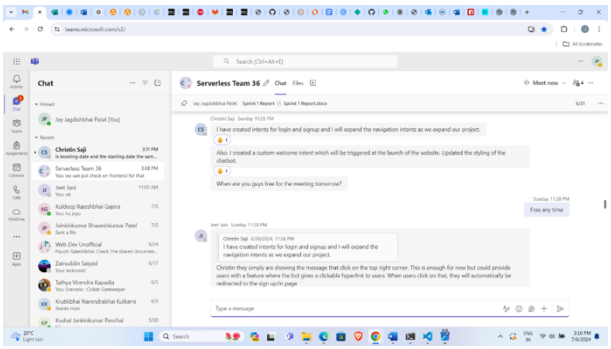

Figure 80: Team members discussing project updates and strategies in a team chat.

Project Management (Sprint 2) Gitlab Board:

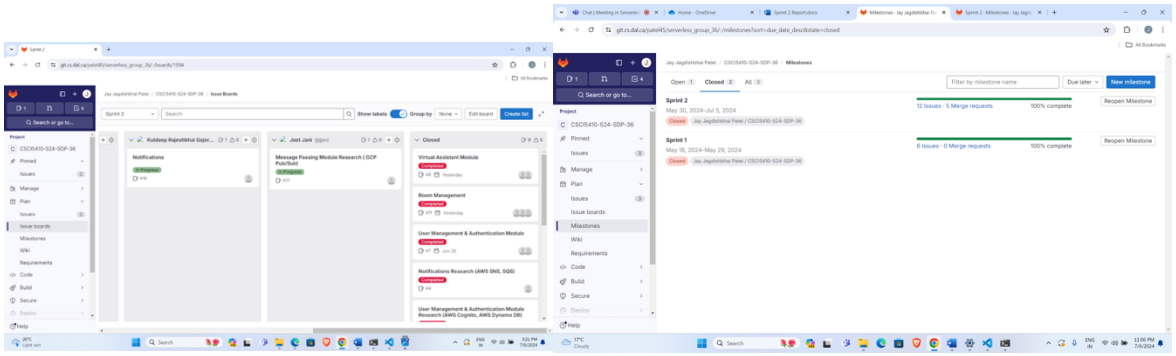

Figure 81: GitLab board and Milestone snapshot displaying Milestone statuses and progress for Sprint 2 of our project.

References

[1] "Amazon Cognito Identity SDK for JavaScript," npm. [Online]. Available: https://www.npmjs.com/package/amazon-cognito-identity-js. [Accessed: June 10 5, 2024].

[2] “What is Amazon Cognito?” *Amazon.com*. [Online]. Available: https://docs.aws.amazon.com/cognito/latest/developerguide/what-is-amazon-cognito.html. [Accessed: June 11, 2024].

[3] M. Mendes, “ReactJs — AWS Cognito Authentication with Auth0,” *Medium*, 21-Nov-2019. [Online]. Available: https://hyprstack.medium.com/reactjs-aws-cognito-authentication-with-auth0-f3f16094fa8. [Accessed: June 13, 2024].

[4] L. Carballo, “Serverless authentication with AWS lambda,” *DEV Community*, 06-Apr-2021. [Online]. Available: https://dev.to/lauracarballo/serverless-authentication-with-aws-lambda-427m. [Accessed: June 13, 2024].

[5] “A. Adi, ‘AWS Cognito with ReactJS for Authentication,’ Medium, Oct. 07, 2020. [Online]. Available: https://medium.com/@adi2308/aws-cognito-with-reactjs-for-authentication-c8916b873ccb. [Accessed: June 14, 2024].”

[6] “D. Saha, ‘Chatbot Development Made Easy: Creating a Simple Bot with Dialogflow,’ Medium, Oct. 22, 2018. [Online]. Available: https://medium.com/@dipan.saha/chatbot-development-made-easy-creating-a-simple-bot-with-dialogflow-ade69caac37d. [Accessed: June 19, 2024].”

[7] “Migrate from Dialogflow ES to CX,” Google Cloud. [Online]. Available: https://cloud.google.com/dialogflow/cx/docs/how/migrate. [Accessed: June 25, 2024].”

[8] “Intents Overview,” Google Cloud. [Online]. Available: https://cloud.google.com/dialogflow/es/docs/intents-overview. [Accessed: June 25, 2024].”

[9] “K. Patil, ‘Chatbot for Your Website Using Dialogflow,’ Dev.to, Jul. 03, 2020. [Online]. Available: https://dev.to/ketan_patil/chatbot-for-your-website-using-dialogflow-fc7. [Accessed: June 26, 2024].”

[10] “Integrating with Dialogflow Messenger,” Google Cloud. [Online]. Available: https://cloud.google.com/dialogflow/es/docs/integrations/dialogflow-messenger. [Accessed: June 26, 2024].”

[11] “Webhook for Fulfillment,” Google Cloud. [Online]. Available: https://cloud.google.com/dialogflow/es/docs/fulfillment-webhook. [Accessed: June 26, 2024].”

[12] “Lambda Input Response Format in Amazon Lex,” AWS Documentation. [Online]. Available: https://docs.aws.amazon.com/lex/latest/dg/lambda-input-response-format.html. [Accessed: June 29, 2024].”

[13] “Best Practices for Storing Access Tokens in the Browser,” The New Stack, Apr. 15, 2020. [Online]. Available: https://thenewstack.io/best-practices-for-storing-access-tokens-in-the-browser/. [Accessed: June 29, 2024].”

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

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