DALHOUSIE UNIVERSITY

CSCI-5410 SERVERLESS DATA PROCESSING

PROJECT REPORT

GROUP – 03

Group Members

Riya Patel	ry818465@dal.ca	B00930901
Dheeraj Bhat	dh210086@dal.ca	B00928874
Arihant Dugar	ar968345@dal.ca	B00917961
Rashmi Goplani	rs216961@dal.ca	B00950827
Preeti Sharma	pr233584@dal.ca	B00929862
Gauravsinh Bharatsinh Solanki	gr441293@dal.ca	B00932065

Course Instructor

Dr. Saurabh Dey

1. Abstract:

The project is a technical proposal for **a Table Reservation App** tailored for restaurants in Halifax, Nova Scotia. It leverages Google Cloud Platform (GCP) and Amazon Web Services (AWS) serverless architecture to ensure scalability, security, and cost-effectiveness. The app will consist of three main components: Customer App, Partner App, and Admin App. It will have key layers including Frontend, Backend Services, Database, Authentication, and APIs. The project will involve the development of various modules for each app, such as Sign Up & Login Module, List Restaurants, Book, edit, delete, view a reservation, Chatbot, Notifications, and Visualization. The data storage and security will be ensured through Google Firestore with role-based access control, encryption of Personally Identifiable Information (PII), and Firebase Authentication. The development environment will include the use of version control systems like Git for collaborative development and Continuous Integration/Continuous Deployment (CI/CD) pipelines for deployment.

2. Feature Components:

a. Customer App

1. Sign Up & Login Module – (Rashmi Goplani)

1.1 Planning

Objective: The Sign up and Log in module of the application allows the user to easily sign up and login to our application with one of the two methods:

1. Sign up/Login using email and password.
    a. Users can create an account and log in using their email and a password.
    b. This method is a standard form of authentication and is commonly used for user accounts.
2. Sign up/Login using identity providers - Google sign-in
    a. Users can use their Google credentials to sign in.
    b. Leveraging identity providers, such as Google, enhances user convenience by eliminating the need to remember multiple login credentials.

To implement this module, we planned to use Firestore, which is a NoSQL database and allows authenticating the users using email id and password or with identity providers such as Google Sign-in.

1.2 Implementation

To implement the Sign-Up and Login feature, I configured the Firestore [8] database and successfully integrated it into our project code. Within the Firestore console, I enabled two sign-in providers: Email/Password and Google. I also created a userDetails collection to store the user details and the user type as ‘user.’

User Sign-up (Email/Password)

- Capture user input from the registration form (email and password).
- Used **createUserWithEmailAndPassword** function to create a new user account. This function securely stores the user's provided email and password in Firebase Authentication. [12]

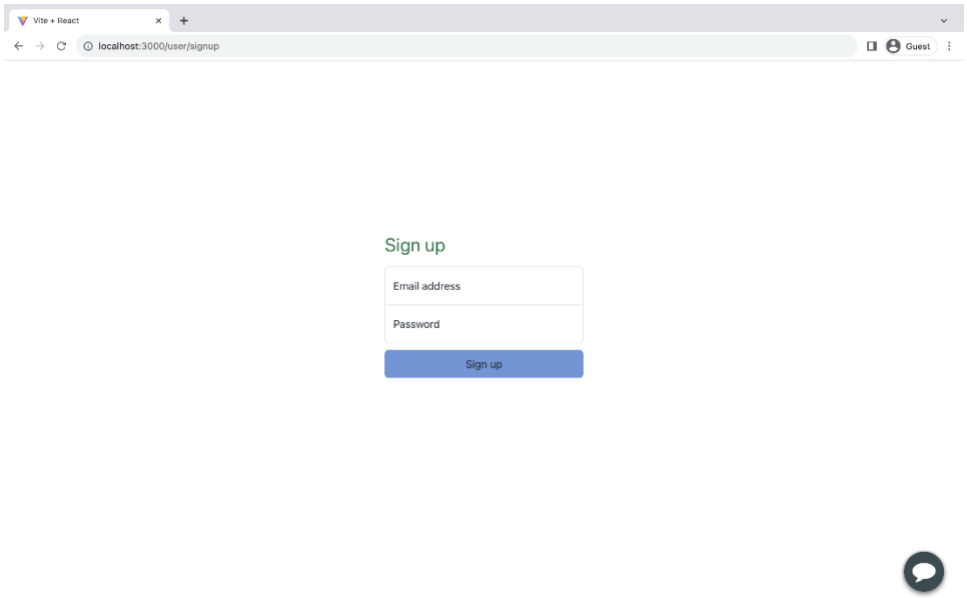


Figure 1User Sign-up (Email/Password)

User Log-in (Email/Password)

- Capture user input from the registration form (email and password).
- Used **signInWithEmailAndPassword** function to authenticate the user. his Firebase Authentication function checks the entered credentials against those stored in the database, granting access upon successful verification. [12]

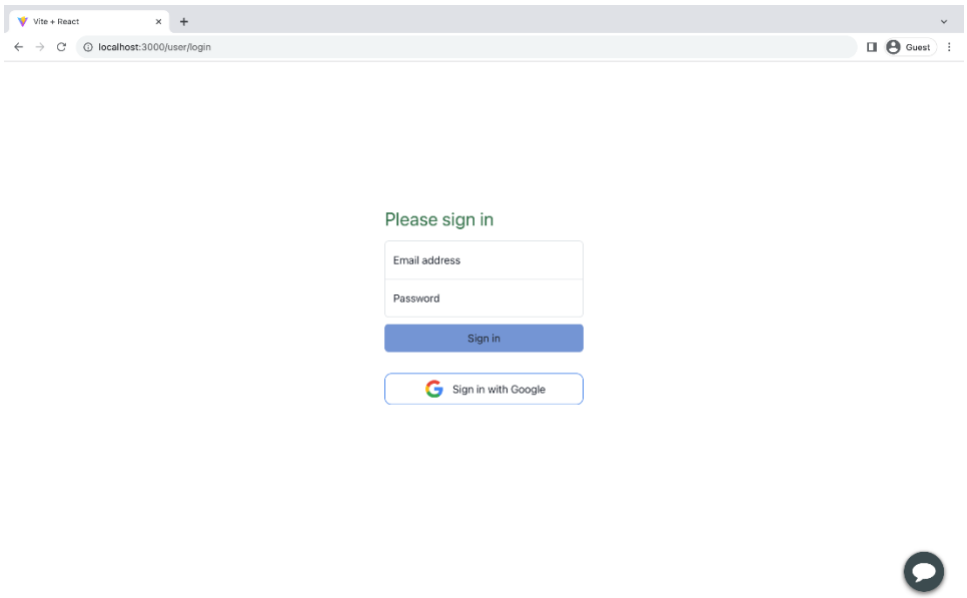


Figure 2User Login (Email/Password)

User Login (Google Sign-in)

- Implemented the function **signInWithGoogle** to handle Google Sign-In. This function is responsible for handling the Google Sign-In process and is typically triggered when users opt for Google as their authentication method. [12]
- Used **signInWithPopup** function with the Google provider to initiate the Google Sign-In process. Paired with the Google provider, this initiates a pop-up window prompting users to sign in with their Google credentials. Upon successful authentication, users gain access to the application [12]

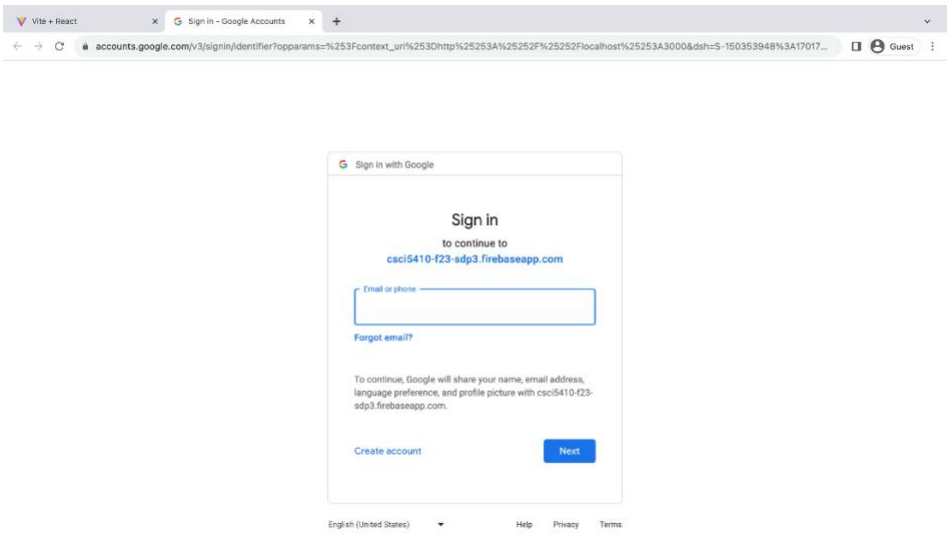


Figure 3 User Login using Google.

As soon as a user sign's up on the application, the user details are store in the userDetails collection with the user type - user.

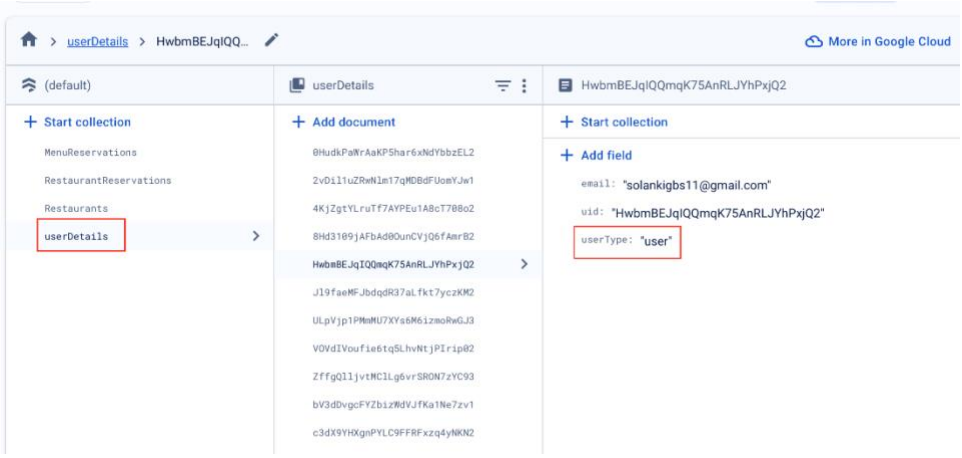


Figure 4 User details stored in userDetails Collection.

On Successful login, a toast message for successful login is shown on top right of the screen.

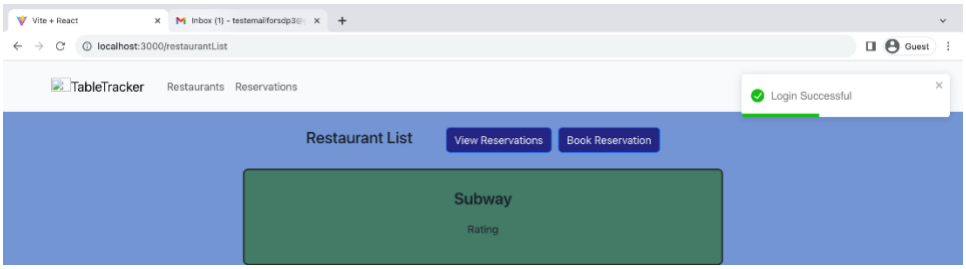


Figure 5 Toast to show successful login.

Frontend Integration

- Implemented frontend components allowing users to Sign up or login using email/password or google.
- Integrate API requests from the frontend to trigger the Firebase Authentication functions to authenticate the user and allow applications access.

User Authentication on navigating routes:

- Whenever any user accesses a particular route, it is first checked if the user is logged in.
- If the user is not logged in, the user is redirected to the login page.
- If the user is logged in, the user is allowed to access that route.

1.3 Test Cases / Use Cases

Table :1 Test Cases for User Sign in/Sign up

TC_No	Test Case	Result
TC_User_Signup_1	Enter email with incorrect format	Displays error message - 'Please include an '@' in the email address. 'adcc' is missing an '@''
TC_User_Signup_2	Submit empty form	Displays error message - 'Please enter email and password'
TC_User_Signup_3	Submit email and password in correct format	User is created and redirected to login page
TC_User_Login_1	Enter incorrect credentials	Displays error message - 'Invalid Credentials'
TC_User_Login_2	Submit empty form for login	Displays error message - 'Please enter email and password'
TC_User_Login_3	Submit the correct login credentials	User redirected to Restaurant List page

TC_ User_ Login_4	Click on Sign-in with Google	Popup opens to sign in using google credentials
TC_ User_ Login_5	Enter invalid google credentials	Google gives invalid credentials error and user remains on the login page on our app
TC_ User_ Login_6	User enters valid google credentials	Popup is closed and user is redirected to Restaurant List page

1.4 Services Used

Firestore – To allow user authentication and store user information [8]

2. List Restaurants – (Preeti Sharma)

2.1 Planning

Objective: The List Restaurant feature allows users to view a list of restaurants available for reservation.

Feature: Listing the names of all the restaurants highlighting their essential details like name, phone number, address, opening hours and closing hours.
At the top of the list of restaurants, users can also take necessary actions regarding their reservations.

Each restaurant also has a view menu, book and add review buttons. The menu button takes the user to the menu of the restaurant, the book button helps the user make a reservation and the review button allows the user to give a review to the restaurant.

Workflow:

- **User Access:**
  - User opens the customer app by signing up and logging in.
- **Restaurant List Display:**
  - The application retrieves a list of restaurants.
  - The list is displayed on the customer's screen displaying the name of the restaurants followed by their rating.
- **Restaurant Page:**
  - Restaurant page shows the restaurant's information on the left like name, phone number and address.
  - It also allows customers to make a reservation by clicking on the Book button which will redirect customers to the reservations page.
  - The right side of the page includes the rating, maximum tables, opening and closing hours and the reviews.
- **Reviews:**
  - The users can rate the restaurant in stars out of 5.
  - They can write a review showcasing their experience and individual opinions.

- They can submit these and will be able to see the real-time update of the review and rating.

2.2 Implementation

Lambda for List Restaurants:

- AWS Lambda Setup:
  - Lambda functions to handle restaurant list.
  - Integrate Lambda functions with Firestore to list restaurants.

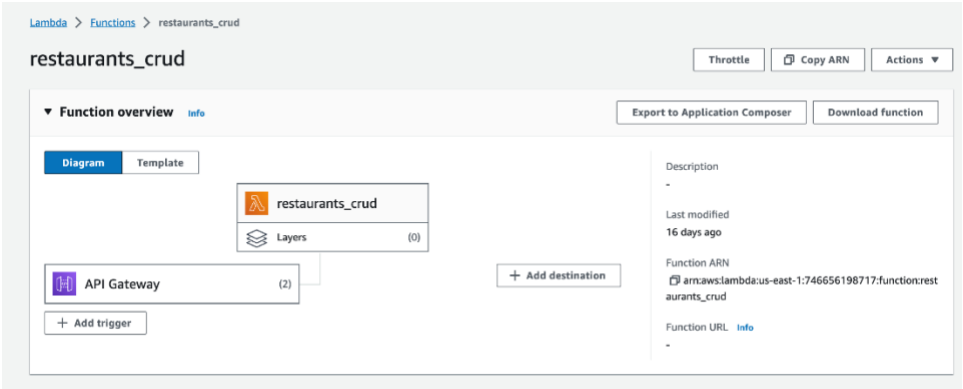


Figure 6: Lambda for list restaurants. [17]

```
function RestaurantList() {
  const isMobile = useMediaQuery({ query: "(max-width: 1080px)" });
  const [restaurants, setRestaurants] = useState(null);
  const navigate = useNavigate();
  const [restaurantDiscount, setRestaurantDiscount] = useState({});

  useEffect(() => {
    const fetchData = async () => {
      const data = await getRestaurants();

      const promises = data.map(async (restaurant) => {
        const url = `${config.Menu.getApiUrl}/${restaurant.restaurant_id}`;
        try {
          const response = await axios.get(url);
          return {
            id: restaurant.restaurant_id,
            discount: response.data.discount,
          };
        } catch (err) {
          console.log(
            `Unable to fetch data for menu of restaurant ${restaurant.restaurant_id}`
          );
          return { id: restaurant.restaurant_id, discount: null };
        }
      });
```

Figure 7:: code of Lambda for list restaurant.

- API Gateway Setup:
  - Configure API Gateway endpoints to trigger the respective Lambda functions for listing restaurants.

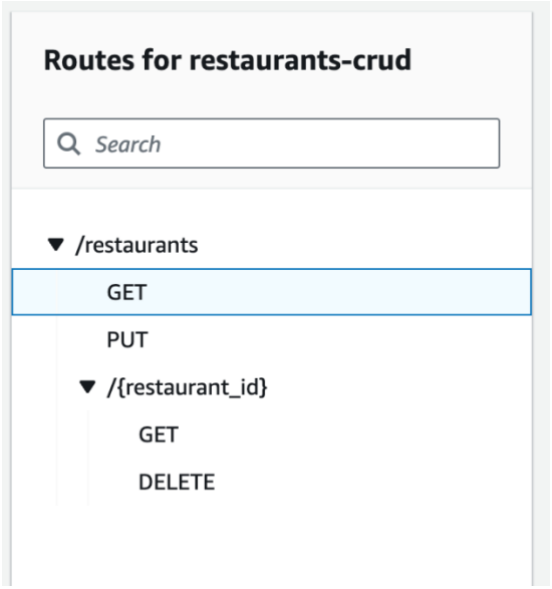


Figure 8: API gateway routes setup for listing restaurants.

- Frontend Integration
  - Implemented frontend components allowing users to view the list of restaurants.
  - Integrate API requests from the frontend to trigger the Lambda function via API Gateway.

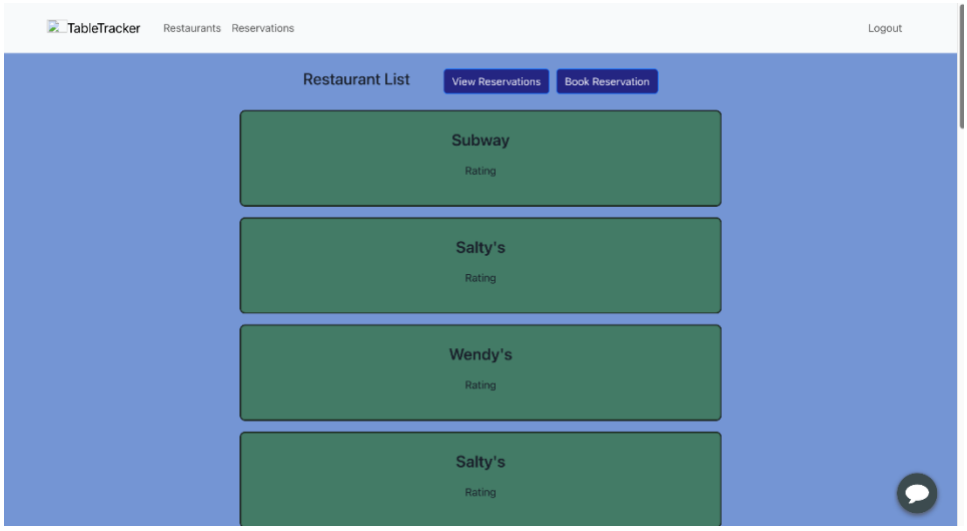


Figure 9: List of restaurants.

2.3 Test Cases / Use Cases

Table: 2 Restaurant List Test Cases:

Test Case ID	Description	Steps to Execute	Expected Result	Pass/Fail
TC_LIST_001	Successful viewing of restaurant list.	1. Signup for the application. 2. Login with the credentials.	Customer can view the list of restaurants.	Pass

		3. The homepage shows the list of restaurants.		
TC_LIST_002	Viewing the restaurant page.	1. Customer attempts to open one of the restaurant pages. 2. All the information about the restaurant loads.	System presents customer with options to book, view menu or give a review to the restaurant.	Pass
TC_LIST_003	Giving review to the restaurant.	1. Input the rating from 0-5 in the stars. 2. Write the review for the restaurant. Click on submit review.	System updates the rating and adds a review to the list of reviews of restaurants.	Pass

2.4 Services Used

AWS Lambda: Code for different functionalities is executed for serverless computing.

Firestore: Used to store restaurant data in real-time database format.

DynamoDB: Used to manage restaurant data as a NoSQL database.

API Gateway: Coordinates API interactions and acts as an entry point.

S3 (Simple Storage Service): A list of images of restaurants is stored and served via S3 (Simple Storage Service).

3. Book, edit, delete and view restaurant reservations (Dheeraj)

3.1 Planning

The functionalities include allowing users to create, modify, delete restaurant reservations. The system architecture employs AWS Lambda for serverless operations, Firestore for data storage, API Gateway for handling API requests, and S3 for lambda code.

Restaurant Reservations structure:

```
{
        "id": "1ntsJQZfX1pe5fp5OqbQ",
        "reservation_date": "2023-12-01T16:00:00.000Z",
        "user_id": "4KjZgtYLruTf7AYPEu1A8cT708o2",
        "restaurant_id": " 17859696-19aa-4ce7-b711-9e8bb8def56e",
        "required_capacity": 5
}
```

Workflow:

- ***Restaurant* Reservation *creation*:**
  - User fills reservation details via the *web* application.
  - Web application triggers the Rest API to store restaurant reservation details.
  - Lambda function triggered via API Gateway*,* verifies, and stores the restaurant reservation details in Firestore.
- **Editing and Deleting Reservations:**
  - Users can edit or delete *restaurant* reservations through the frontend. *They can only edit a reservation one hour before the reservation date.*
  - Web application triggers the Rest API to edit/delete the restaurant reservation.
  - Lambda function triggered via API Gateway*, updates,* or removes *the restaurant* reservation *in the* Firestore *collection.*
- **Viewing *Restaurant* Reservation*s*:**
  - Users request to view *their reservations.*
  - Web application triggers the Rest API to fetch restaurant reservation(s).
  - Lambda function triggered via API Gateway, fetches the restaurant reservations belonging to the user from the Firestore collection. The UI displays the reservations to the user.
- **Integration Testing:**
  - Test scenarios to validate end-to-end functionality, ensuring smooth interactions between Lambda, Firestore, API Gateway, and S3.
- **Security Measures:**
  - Implement authentication and authorization mechanisms (e.g., AWS Cognito, IAM) to *control user access to reservations and menus.*
  - Ensure encryption and secure access policies for data stored in Firestore, and S3.

The *planning outlines the essential components, data structures, workflow, and potential considerations required to implement the mentioned functionalities using the specified technologies.*

3.2 Implementation

Create Restaurant Reservations

- AWS Lambda Setup:
  - Create Lambda functions to handle restaurant reservation creation requests [9].
  - Integrate Lambda functions with Firestore to store restaurant reservation details [8].

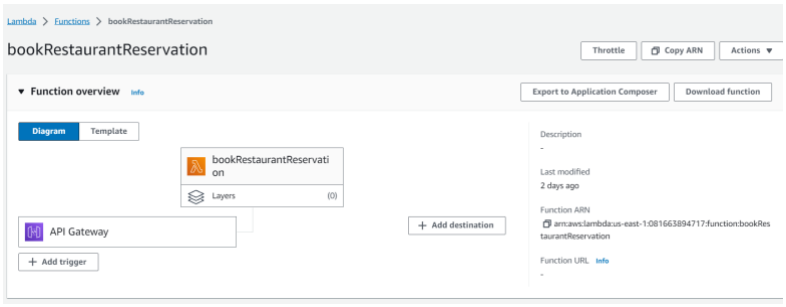


Figure 10: Lambda for create restaurant reservation.

```
admin.initializeApp({
  credential: admin.credential.cert(serviceAccount),
  databaseURL: "https://csci5410-f23-sdp3.firebaseio.com", // Firebase project URL
});

const db = admin.firestore();

exports.handler = async (event) => {
  try {
    const reservationDetails = JSON.parse(event.body);
    const { restaurantId, reservationDate, requiredCapacity, userId } =
      reservationDetails;

    const newReservationDate = new Date(reservationDate);

    const restaurantDoc = db.collection("Restaurants").doc(restaurantId);
    const response = await restaurantDoc.get();
    if (
      newReservationDate >= openingDate &&
      newReservationDate <= closingDate
    ) {
      const reservationsDocs = db.collection("RestaurantReservations");
      const addedReservation = await reservationsDocs.add({
        restaurant_id: restaurantId,
        reservation_date:
          admin.firestore.Timestamp.fromDate(newReservationDate),
        required_capacity: requiredCapacity,
        user_id: userId,
      });

      return Responses._200({
        message: "Reservation successful",
        reservation_id: addedReservation.id,
      });
    } else {
      return Responses._400({
        message: "Reservation time is outside the restaurant's opening hours",
      });
    }
  } catch (error) {
    console.log(error);
    return Responses._400({
      message: "Error booking restaurant reservations",
    });
  }
};
```

Figure 11:: code of Lambda for create restaurant reservation.

- API Gateway Setup:
  - Configure API Gateway endpoints to trigger the respective Lambda functions for restaurant reservation creation. [10]

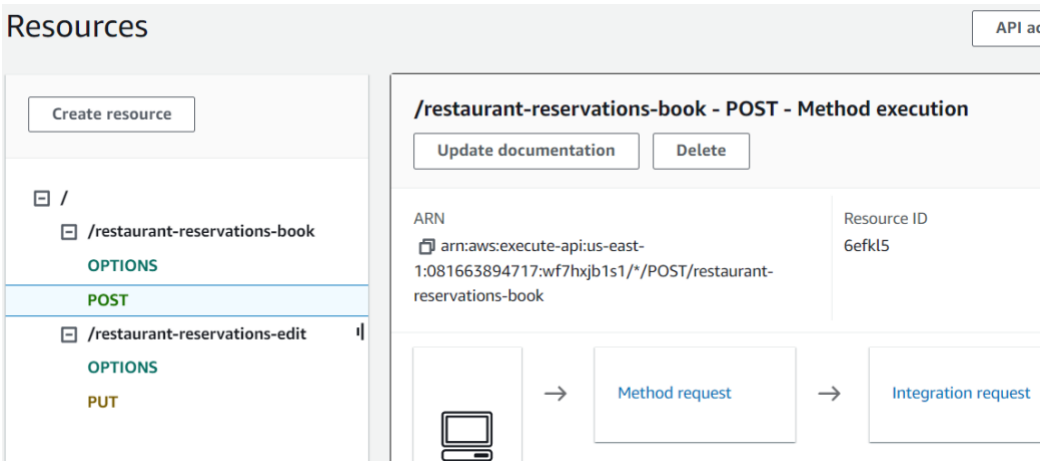


Figure 12: API gateway for create restaurant reservation.

- Frontend Integration
  - Implemented frontend components allowing users to input and submit restaurant reservation details.
  - Integrate API requests from the frontend to trigger the Lambda function via API Gateway.

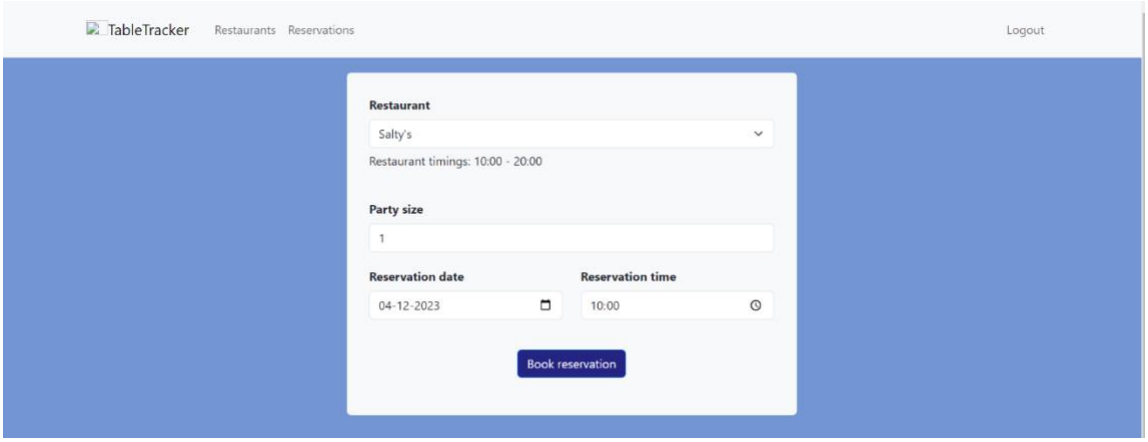


Figure 13: Create restaurant reservation form.

Edit Restaurant Reservations

- AWS Lambda Setup:
  - Create Lambda functions to handle restaurant reservation update requests. [9].
  - Integrate Lambda functions with Firestore to update restaurant reservation details. [8]
  - The customer can update a restaurant any time before 1 hour from the reservation.

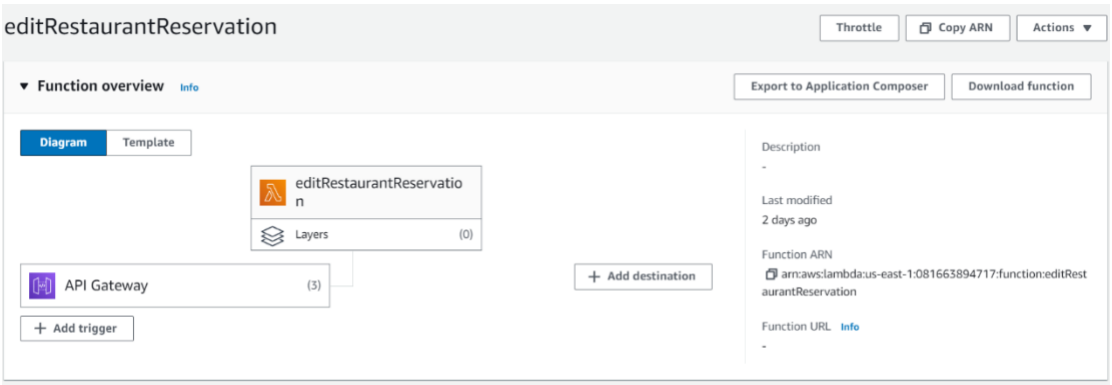


Figure 14: lambda for edit restaurant reservations [17]

```
admin.initializeApp({
  credential: admin.credential.cert(serviceAccount),
});

const db = admin.firestore();

exports.handler = async (event) => {
  try {
    const reservationDetails = JSON.parse(event.body);
    const {
      reservationId,
      restaurantId,
      reservationDate,
      requiredCapacity,
      userId,
      isApproved,
    } = reservationDetails;

---

        await reservationDocRef.update(updatedReservation);

        return Responses._200({
          message: "Reservation edited successfully",
        });
      } else {
        return Responses._400({
          message: "Reservation time is outside the restaurant's opening hours",
        });
      }
    } else {
      return Responses._400({
        message:
          "Reservations can only be edited 1 hour before the reservation time.",
      });
    }
  } catch (error) {
    console.log(error);
    return Responses._400({
      message: "Error booking restaurant reservations",
      error: error.message,
    });
  }
};
```

Figure 15: code of Lambda for edit restaurant reservation.

- API Gateway Setup:
  - Configure API Gateway endpoints to trigger the respective Lambda functions for edit restaurant reservation. [D3]

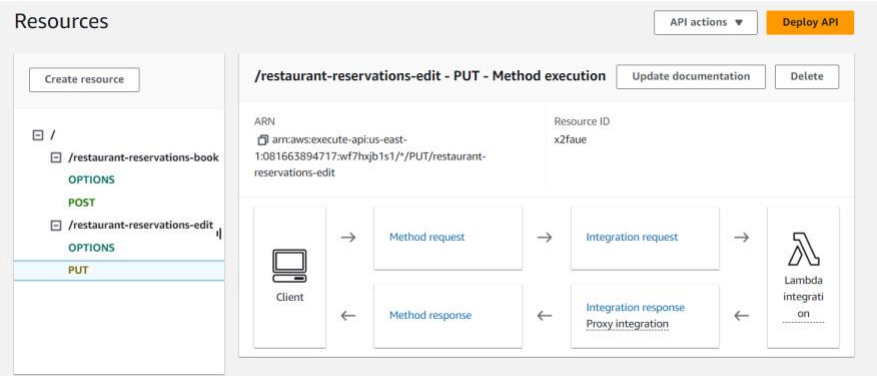


Figure 16: API gateway for edit restaurant reservation.

- Frontend Integration
  - Implemented frontend components allowing users to edit the restaurant reservation details.
  - Integrate API requests from the frontend to trigger the Lambda function via API Gateway.

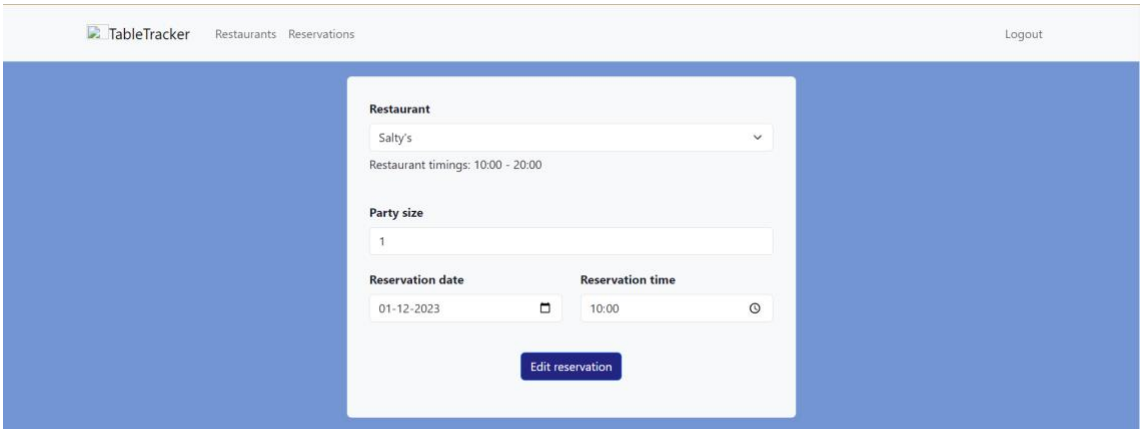


Figure 17: Edit restaurant reservation form.

Delete Restaurant Reservations

- AWS Lambda Setup:
  - Create Lambda functions to handle restaurant reservation delete requests. [9]
  - Integrate Lambda functions with Firestore to delete restaurant reservation details. [8]

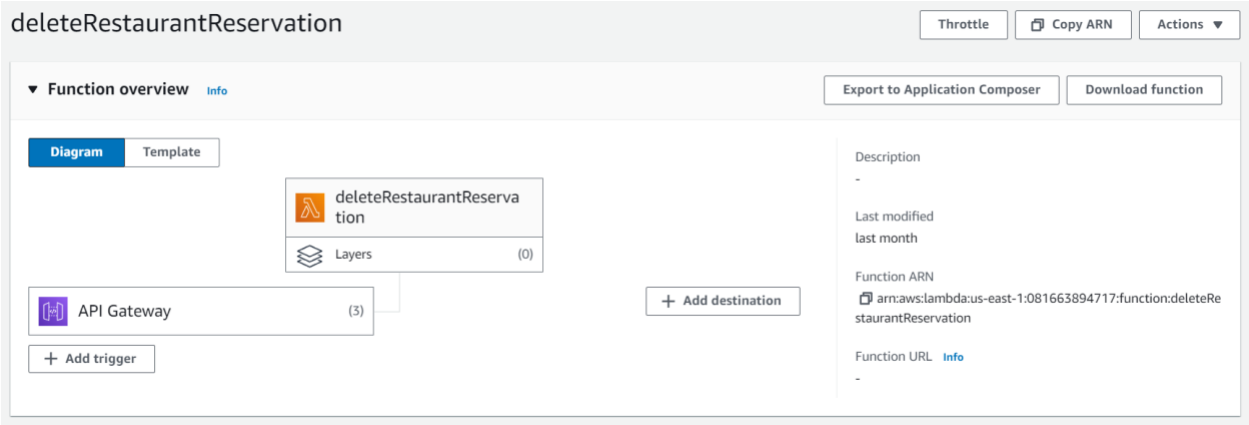


Figure 18: lambda for delete restaurant reservations. [17]

```
var admin = require("firebase-admin");
const axios = require("axios");
var serviceAccount = require("./sdp3-firestore.json");
const Responses = require("./ApiResponses");

admin.initializeApp({
  credential: admin.credential.cert(serviceAccount),
});

const db = admin.firestore();

exports.handler = async (event) => {
  try {
    const reservationId = event.pathParameters.id;

    const reservationDocRef = db
      .collection("RestaurantReservations")
      .doc(reservationId);
    const reservation = await reservationDocRef.get();

    if (!reservation.exists) {
      return Responses._400({
        message: "Restaurant reservation does not exist",
      });
    }

    await reservationDocRef.delete();
    return Responses._200({
      message: "Reservation successfully deleted!",
    });
  } catch (error) {
    console.log(error);
    return Responses._400({
      message: "Error deleting restaurant reservations",
      error: error.message,
    });
  }
};
```

Figure 19: code of Lambda for delete restaurant reservation.

- API Gateway Setup:
  - Configure API Gateway endpoints to trigger the respective Lambda functions for delete restaurant reservation. [10].

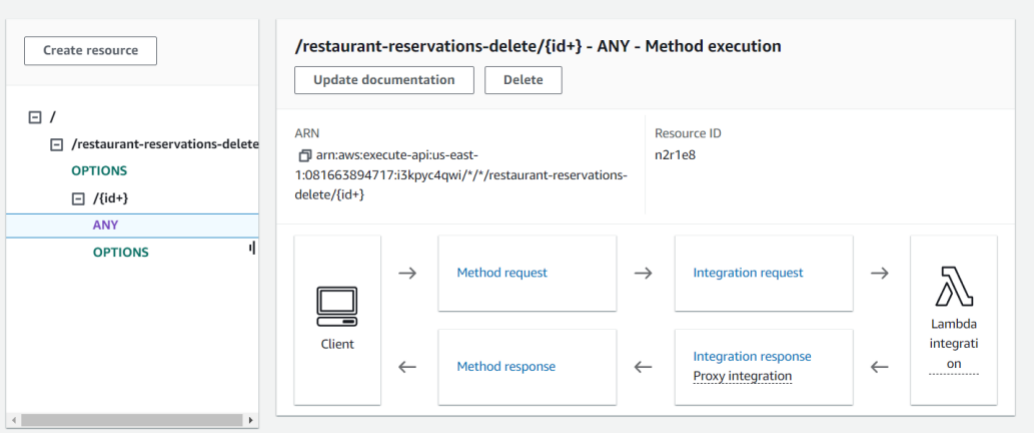


Figure 20: API gateway for delete restaurant reservation.

- Frontend Integration
  - Implemented frontend components allowing users to delete restaurant reservations.
  - Integrate API requests from the frontend to trigger the Lambda function via API Gateway.

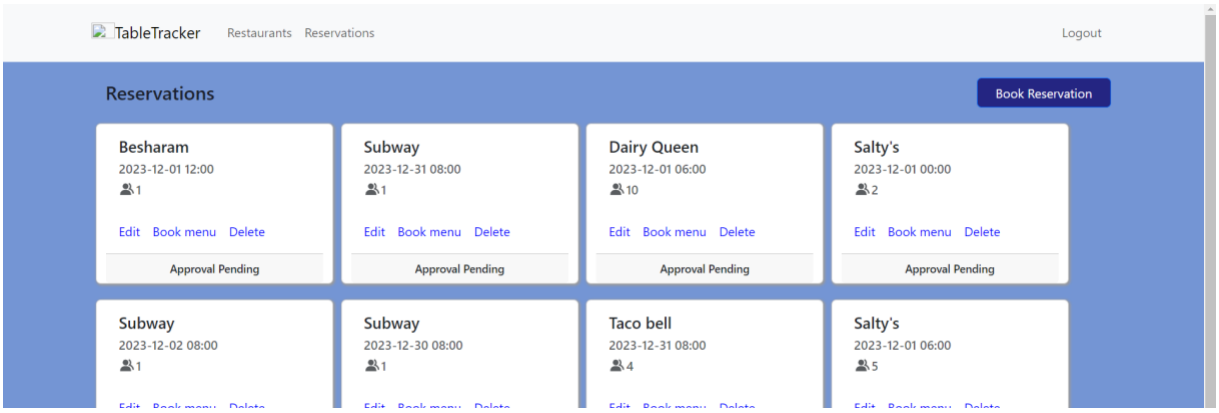


Figure 21: delete restaurant reservation UI.

View Restaurant Reservations

- AWS Lambda Setup:
  - Create Lambda functions to fetch restaurant reservations for a customer. [9]
  - Integrate Lambda functions with Firestore to fetch all restaurant reservations for a customer. [8].

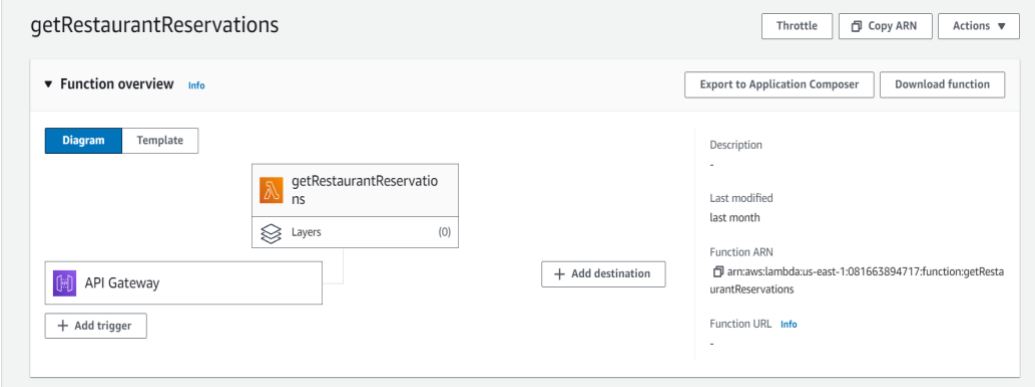


Figure 22: lambda for view restaurant reservations. [17]

```
const admin = require("firebase-admin");
const Responses = require("./ApiResponses");
const serviceAccount = require("./sdp3-firestore.json"); // file path for service account credentials

admin.initializeApp({
  credential: admin.credential.cert(serviceAccount),
  databaseURL: "https://csci5410-f23-sdp3.firebaseio.com", // Firebase project URL
});

exports.handler = async (event, context) => {
  try {
    const db = admin.firestore();
    const userId = event.pathParameters.userId ?? "U1";
    const dbCollection = db.collection("RestaurantReservations");

    const reservationsDocs = await dbCollection
      .where("user_id", "==", userId)
      .get();

    if (reservationsDocs.empty) {
      return Responses._400({
        message: "No reservations found for this user.",
      });
    }

    const reservations = reservationsDocs.docs.map((document) => ({
      id: document.id,
      ...document.data(),
      reservation_date: document.data().reservation_date.toDate(),
    }));

    return Responses._200({
      message: "Fetched reservations successfully",
      data: [...reservations],
    });
  } catch (error) {
    console.log(error);
    return Responses._400({
      message: "Error fetching restaurant reservations",
      error: error.message,
    });
  }
};
```

Figure 23: code of Lambda for view restaurant reservations.

- API Gateway Setup:
  - Configure API Gateway endpoints to trigger the respective Lambda functions for view restaurant reservation. [10].

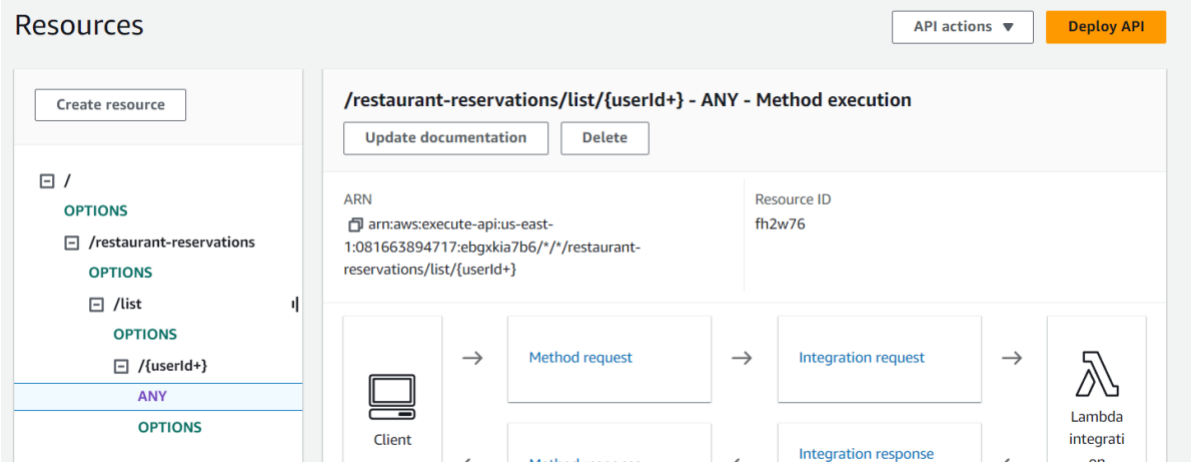


Figure 24: API gateway for view restaurant reservations.

- Frontend Integration
  - Implemented frontend components to show users all their restaurant reservations.
  - Integrate API requests from the frontend to trigger the Lambda function via API Gateway.

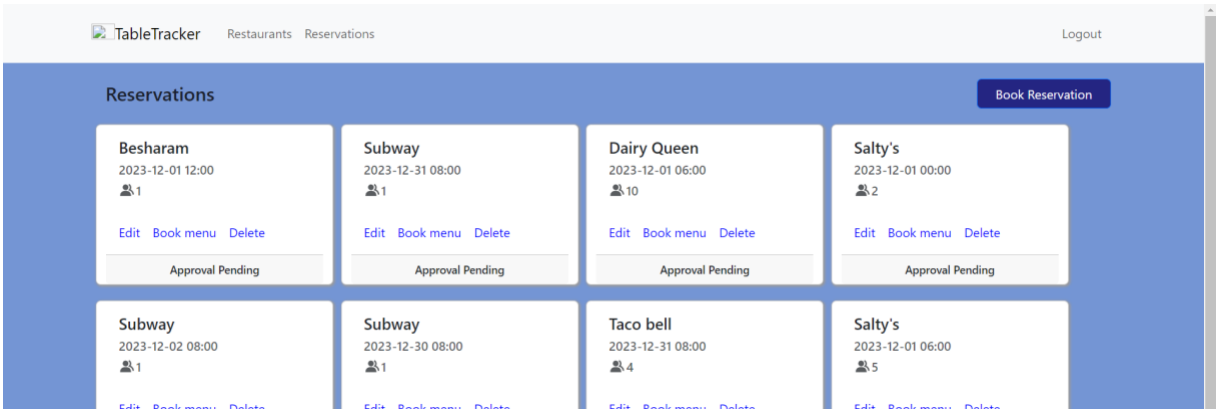


Figure 25: Get restaurant reservations.

3.3 Test Cases / Use Cases

Table: 3 Booking Test Cases:

Test Case ID	Description	Steps to Execute	Expected Result	Pass/Fail
TC_REST_BOOK_001	Successful restaurant reservation creation	1. Navigate to Book Reservation' form. 2. Fill in required details. 3. Submit reservation request.	New restaurant reservation is created successfully.	Pass
TC_REST_BOOK_002	Booking restaurant with incomplete	1. Attempt to book without filling	System prompts to fill in all	Pass

	information	mandatory fields.	mandatory fields.	
TC_REST_BOOK_003	Booking with date/time before opening time or a past date	1. Invalid date/time input	System displays an error message for invalid date/time.	Pass

Table: 4 Editing and Deleting Test Cases

Test Case ID	Description	Steps to Execute	Expected Result	Pass/Fail
TC_REST_RES_EDIT_001	Successful reservation modification	1. Access 'View Reservations'. 2. Choose 'Edit'. 3. Modify reservation details. 4. Save changes.	Reservation details are updated successfully.	Pass
TC_REST_RES_EDIT_002	Deletion of an existing reservation	1. Access 'View Reservations'. 2. Choose Delete.	Reservation is successfully deleted.	Pass
TC_REST_RES_EDIT_003	Editing reservation more than 1 hour before reservation date	1. Attempt to modify reservation.	System prohibits modification.	Pass

Table: 5 Viewing Reservations Test Cases

Test Case ID	Description	Steps to Execute	Expected Result	Pass/Fail
TC_REST_RES_001	Viewing reservations of the user	1. Navigate to reservations page. 2. Check displayed reservations.	All reservations created by user shown	Pass
TC_REST_RES_002	Viewing Reservations without an existing reservation	1. Navigate to reservations page. 2. Check displayed reservations.	System displays empty page with button to create reservation	Pass

3.4 Services Used

- **AWS Lambda:** Used for serverless computing to execute code for various functionalities. [9]
- **Firestore:** Utilized as a real-time database for storing restaurant reservation-related data. [8]
- **API Gateway:** Serves as an entry point for APIs and manages their interactions. [10]
- **S3 (Simple Storage Service):** Used to store zip of the lambda code files.

4. Book, edit, delete, view menu for a reservation – (Arihant Dugar)

4.1 Planning

The functionalities include allowing users to create, modify, delete reservations, and view associated menus. The system architecture employs AWS Lambda for serverless operations, Firestore for data storage, DynamoDB for additional data management, API Gateway for handling API requests, and S3 for storing menu item images.

Menu Items Structure:


```
{
  "id": "3c6fc9e1-aef2-4f51-b34f-e21c5e53d642",
  "discount": 0,
  "items": [
    {
      "id": 1,
      "availability": true,
      "description": "Pizza",
      "discount": 0,
      "img": "https://menu-items-bucket-csci5410.s3.us-east-2.amazonaws.com/pizza.png",
      "name": "Test",
      "price": "20"
    }
  ],
  "restaurantId": "3c6fc9e1-aef2-4f51-b34f-e21c5e53d642"
}
```


Menu Reservation Structure:


```
{
  "id": "582a3ac3-c193-47d6-aeda-
3cf75ed9d09dqTH6TyoSoXNFY3H6b69C4KjZgtYLruTf7AYPEu1A8cT708o2",
  "items": [
    {
      "id": "1",
      "quantity": 2
    },
    {
      "id": "2",
      "quantity": 2
    }
  ],
  "reservationId": "qTH6TyoSoXNFY3H6b69C",
  "restaurantId": "582a3ac3-c193-47d6-aeda-3cf75ed9d09d",
  "userId": "4KjZgtYLruTf7AYPEu1A8cT708o2"
}
```

Workflow:

- **Reservation Booking:**
  - User fills reservation details via the application.
  - Lambda function triggered via API Gateway verifies and stores the reservation details in Firestore and DynamoDB.
- **Editing and Deleting Reservations:**
  - Users can edit or delete reservations through the frontend.
  - Lambda function triggered via API Gateway updates or removes reservation data from Firestore and DynamoDB accordingly.
- **Viewing Menu for Reservation:**
  - Users request to view menus associated with their reservations.
  - API Gateway fetches menu details (from Firestore) linked to the reservation and displays them in the application.
- **Integration Testing:**
  - Test scenarios to validate end-to-end functionality, ensuring smooth interactions between Lambda, Firestore, DynamoDB, API Gateway, and S3.
- **Security Measures:**
  - Implement authentication and authorization mechanisms (e.g., AWS Cognito, IAM) to control user access to reservations and menus.
  - Ensure encryption and secure access policies for data stored in Firestore, DynamoDB, and S3.

The planning outlines the essential components, data structures, workflow, and potential considerations required to implement the mentioned functionalities using the specified technologies.

4.2 Implementation

Create Menu Reservation:

- AWS Lambda Setup:
  - Create Lambda functions to handle reservation creation requests.
  - Integrate Lambda functions with Firestore and DynamoDB to store reservation details.

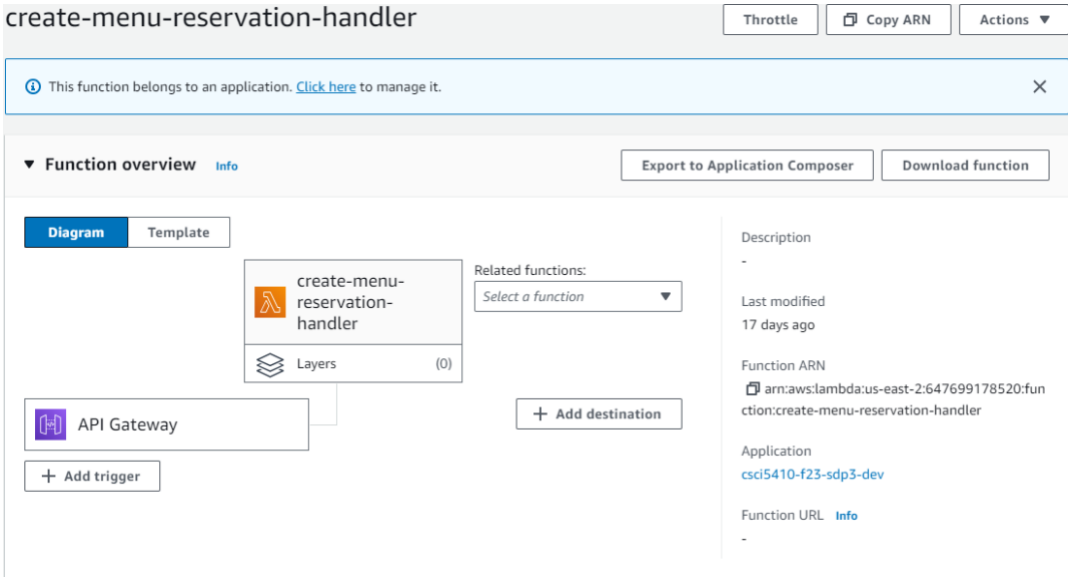
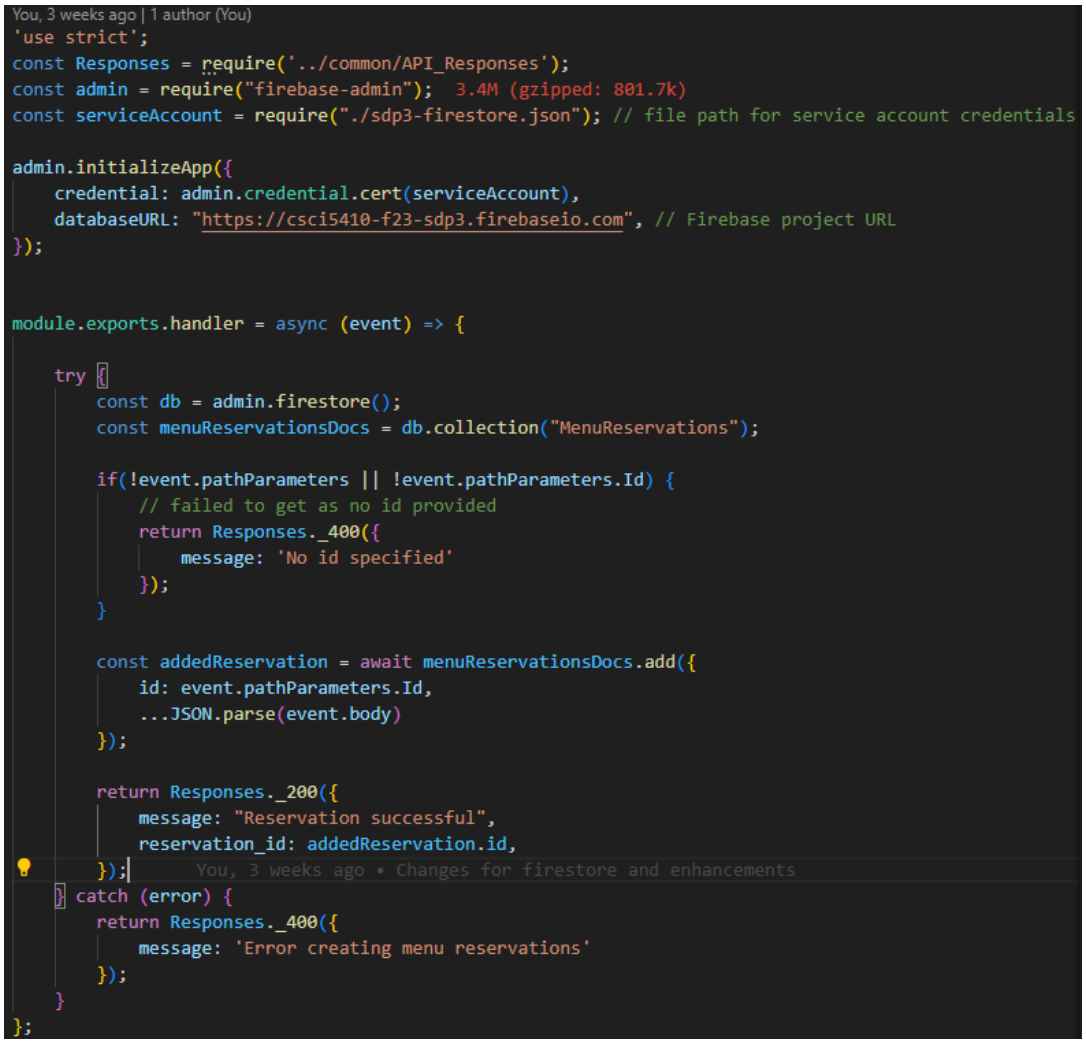


Figure 26 Lambda Function for Create Menu Reservation [17]



```
You, 3 weeks ago | 1 author (You)
'use strict';
const Responses = require('../common/API_Responses');
const admin = require("firebase-admin");  3.4M (gzipped: 801.7k)
const serviceAccount = require("./sdp3-firestore.json"); // file path for service account credentials

admin.initializeApp({
    credential: admin.credential.cert(serviceAccount),
    databaseURL: "https://csci5410-f23-sdp3.firebaseio.com", // Firebase project URL
});


module.exports.handler = async (event) => {

    try {
        const db = admin.firestore();
        const menuReservationsDocs = db.collection("MenuReservations");

        if(!event.pathParameters || !event.pathParameters.Id) {
            // failed to get as no id provided
            return Responses._400({
                message: 'No id specified'
            });
        }

        const addedReservation = await menuReservationsDocs.add({
            id: event.pathParameters.Id,
            ...JSON.parse(event.body)
        });

        return Responses._200({
            message: "Reservation successful",
            reservation_id: addedReservation.id,
        });       You, 3 weeks ago • Changes for firestore and enhancements
    } catch (error) {
        return Responses._400({
            message: 'Error creating menu reservations'
        });
    }
};
```

Figure 27 Lamda Function Code for Create Menu Reservation

- API Gateway Configuration:
  - Configure API Gateway endpoints to trigger the respective Lambda functions for reservation creation.

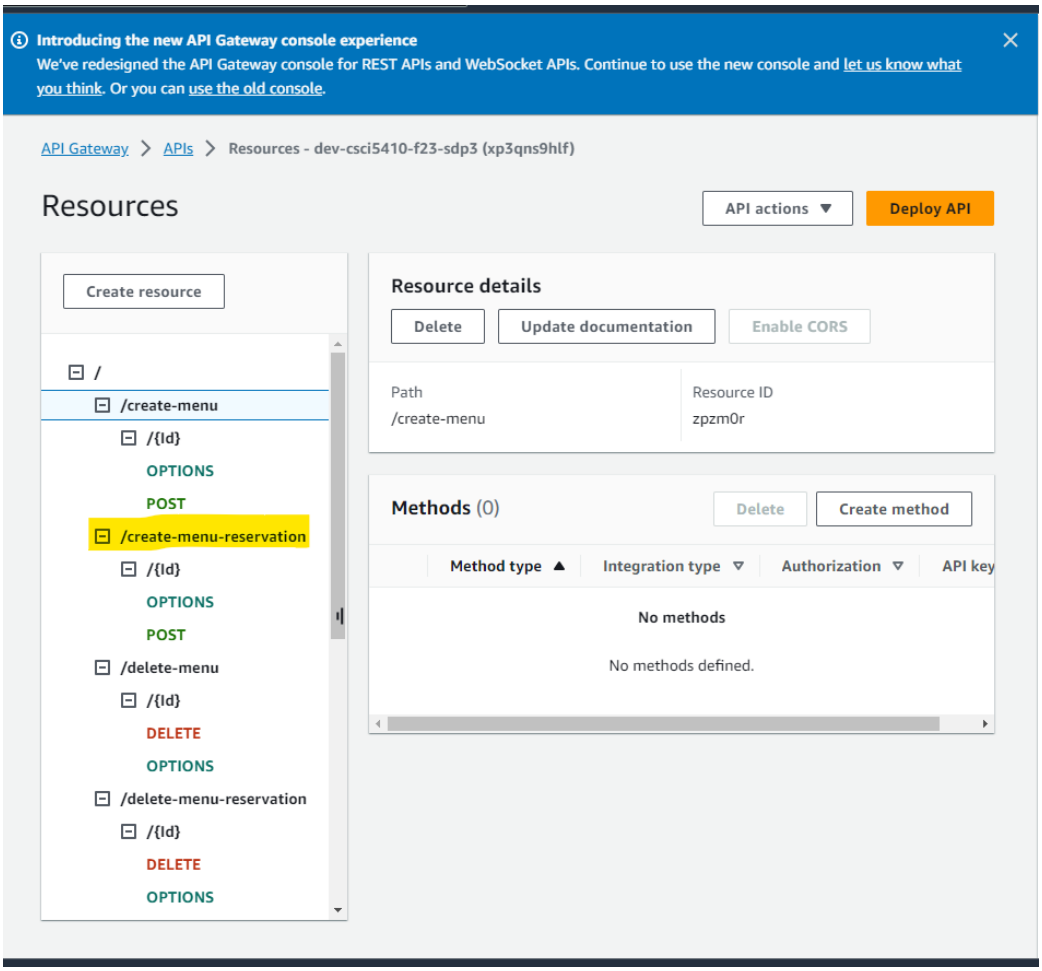


Figure 28 API endpoints in API Gateway for create menu reservation.

- Frontend Integration:
  - Implement frontend components allowing users to input and submit reservation details.
  - Integrate API requests from the frontend to trigger the Lambda function via API Gateway.

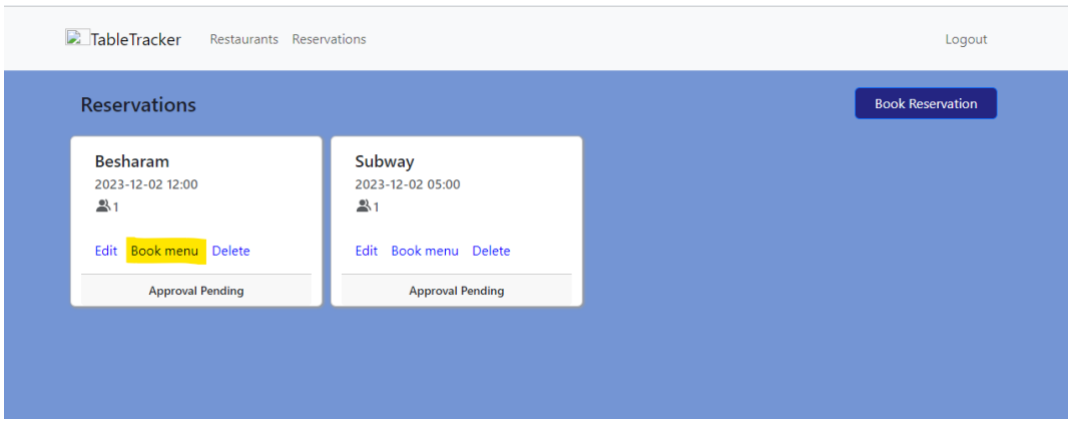


Figure 29 UI for booking menu from reservations dashboard.

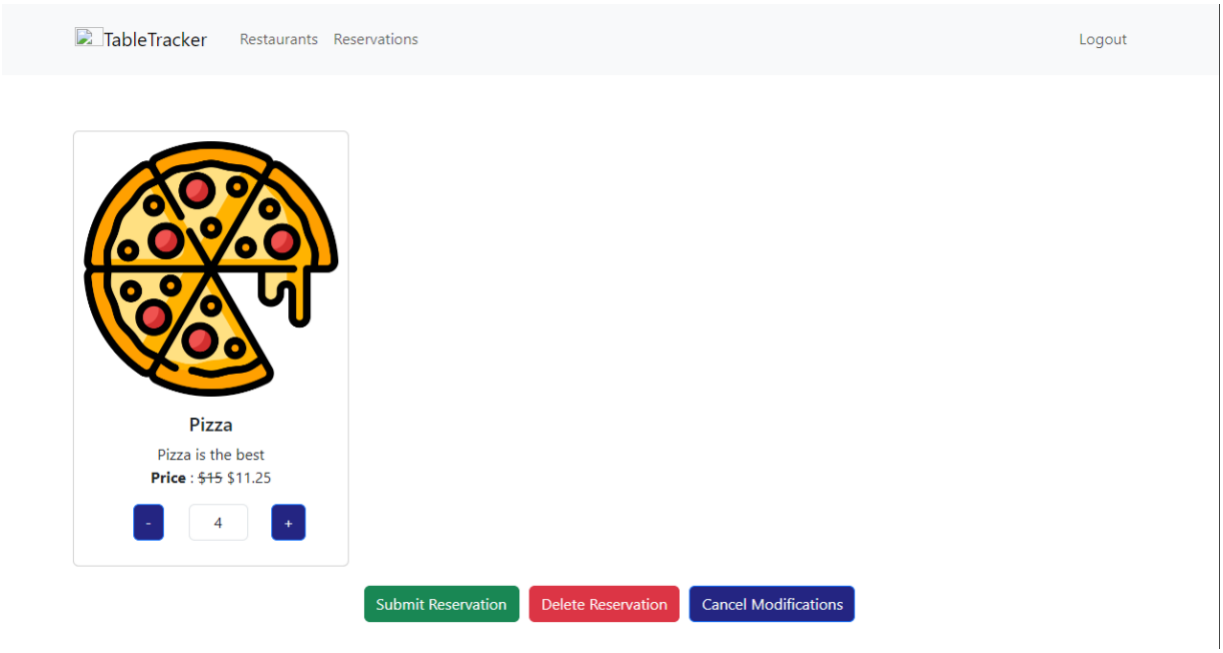


Figure 30 UI Design for creating a menu reservation.

Editing and Deleting Reservations:

- Lambda Function:
  - Create new Lambda functions to handle reservation update and deletion operations.

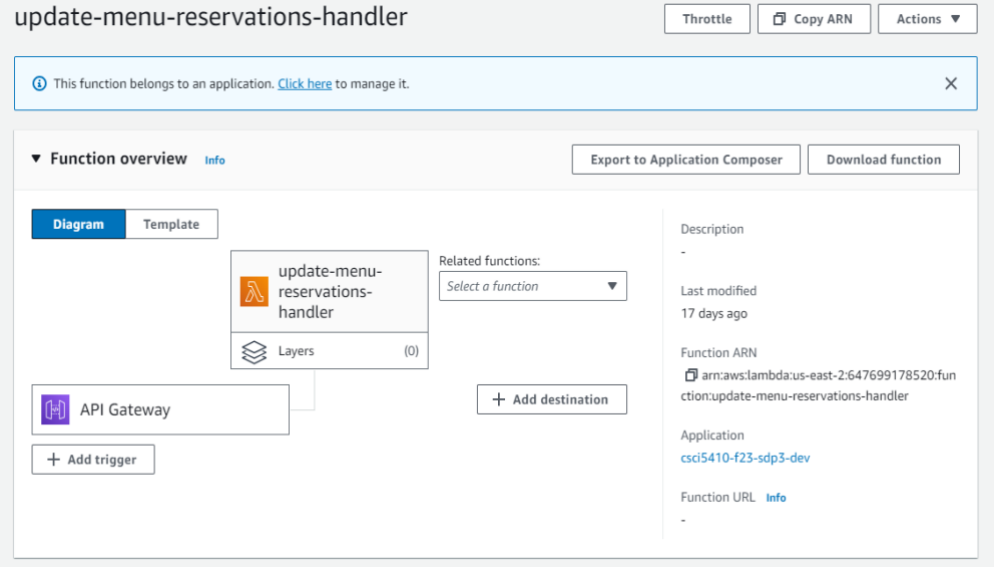


Figure 31 Lambda function for updating menu reservation. [17]

```
You, 3 weeks ago | 1 author (You)
'use strict';
const Responses = require('../common/API_Responses');
const admin = require("firebase-admin");  3.4M (gzipped: 801.7k)
const serviceAccount = require("./sdp3-firestore.json"); // file path for service account credentials

admin.initializeApp({
    credential: admin.credential.cert(serviceAccount),
    databaseURL: "https://csci5410-f23-sdp3.firebaseio.com", // Firebase project URL
});

module.exports.handler = async (event) => {

    try {
        const db = admin.firestore();

        if(!event.pathParameters || !event.pathParameters.Id) {
            // failed to get as no id provided
            return Responses._400({
                message: 'No id specified'
            });
        }

        const collectionRef = db.collection('MenuReservations');

        // Get the document to update
        const querySnapshot = await collectionRef.where('id', '==', event.pathParameters.Id).get();

        querySnapshot.forEach(async (doc) => {
            const docId = doc.id;
            const data = doc.data();       You, 3 weeks ago • Changes for firestore and enhancements
            data.items = JSON.parse(event.body);

            // Update the document with the modified data
            await collectionRef.doc(docId).set(data);
        });

        // Return a success response
        return {
            statusCode: 200, body: 'Menu reservation updated successfully'
        };
    } catch (error) {
        console.log(error);
        return Responses._400({
            message: "Error updating menu reservation",
            error: error.message,
        });
```

Figure 32 Lambda function code for updating menu reservation.

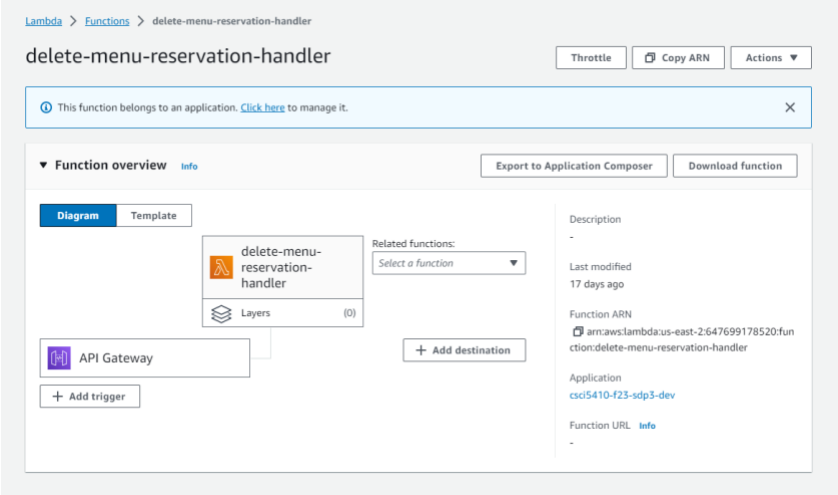


Figure 33 Lambda function for deleting menu reservation. [17]

```
You, 3 weeks ago | 1 author (You)
'use strict';
const Responses = require('../common/API_Responses');
const admin = require("firebase-admin");  3.4M (gzipped: 801.7k)
const serviceAccount = require("./sdp3-firestore.json"); // file path for service account credentials

admin.initializeApp({
    credential: admin.credential.cert(serviceAccount),
    databaseURL: "https://csci5410-f23-sdp3.firebaseio.com", // Firebase project URL
});
      You, 2 months ago • Added changes for create and delete menu reservat…
module.exports.handler = async (event) => {

    if(!event.pathParameters || !event.pathParameters.Id) {
        // failed to get as no id provided
        return Responses._400({
            message: 'No id specified'
        });
    }

    const db = admin.firestore();
    const collectionRef = db.collection('MenuReservations');

    try {
        const querySnapshot = await collectionRef.where('id', '==', event.pathParameters.Id).get();

        if (querySnapshot.empty) {
            return Responses._400({
                message: 'Menu reservation not found'
            });
        }

        // Delete the first document found with the specified child ID
        const documentSnapshot = querySnapshot.docs[0];
        await documentSnapshot.ref.delete();

        return Responses._200({message: 'Menu reservation deleted successfully'})
    } catch (error) {
        return Responses._400({message: 'Error deleting menu reservation'});
    }
};
```

Figure 34 Lambda function code for deleting menu reservation.

- API Gateway Configuration:
  - Configure API Gateway endpoints to trigger the respective Lambda functions for reservation update and delete.

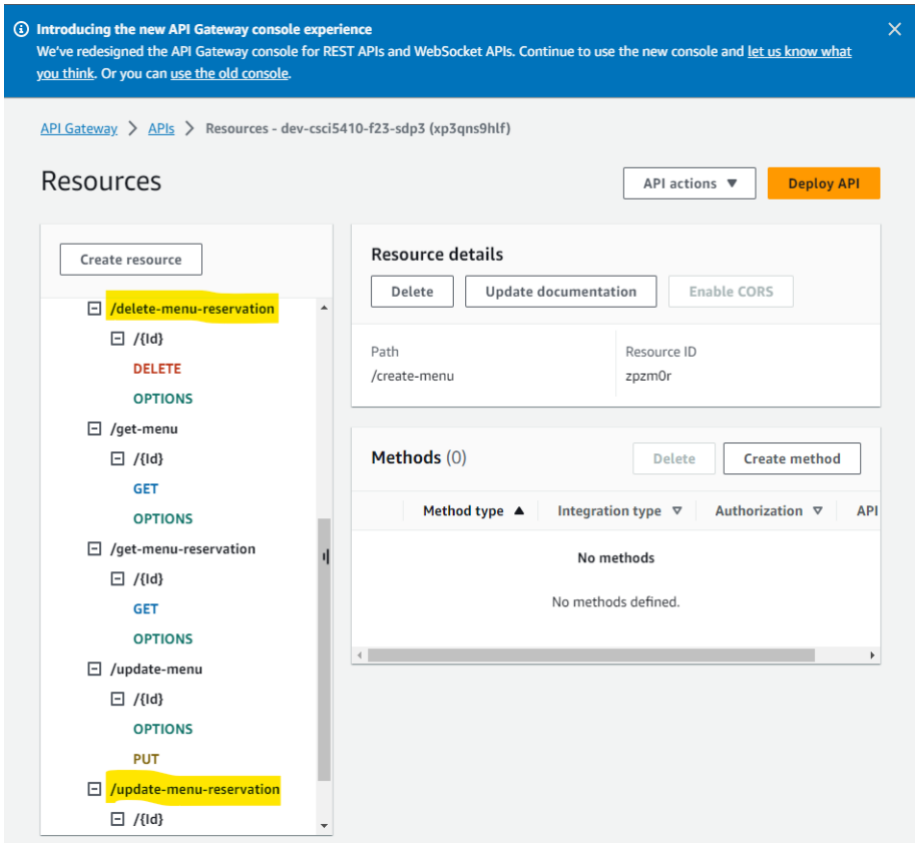


Figure 35 API endpoints in API gateway for Delete and Update menu reservations.

- Frontend Implementation:
  - Develop UI components enabling users to modify or delete existing reservations.
  - Create frontend logic to interact with the corresponding API endpoints for editing or deleting reservations.

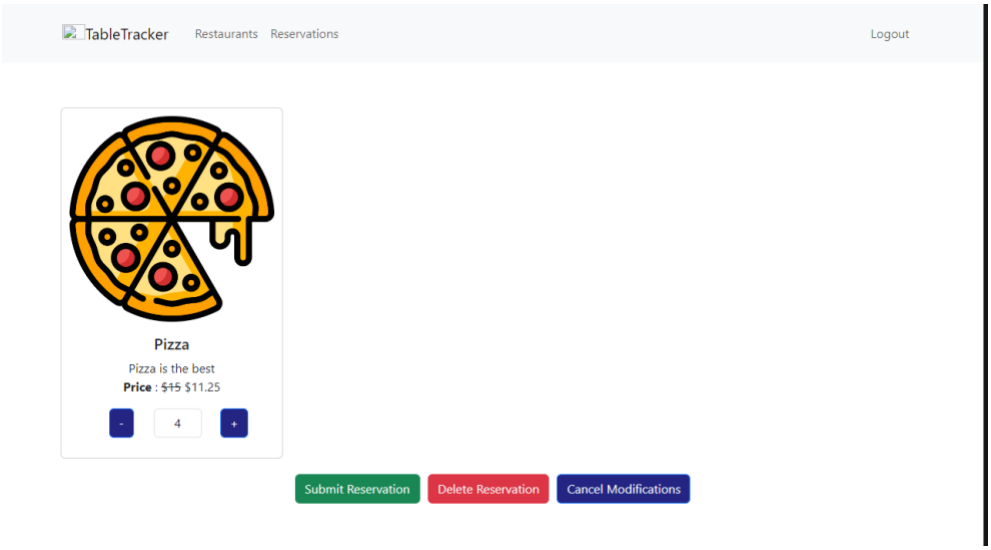


Figure 36 UI for updating and deleting menu reservations.

Viewing Menu for Reservation:

- Firestore Query Setup:
  - Construct queries in Lambda functions to fetch menu details linked to a specific reservation.

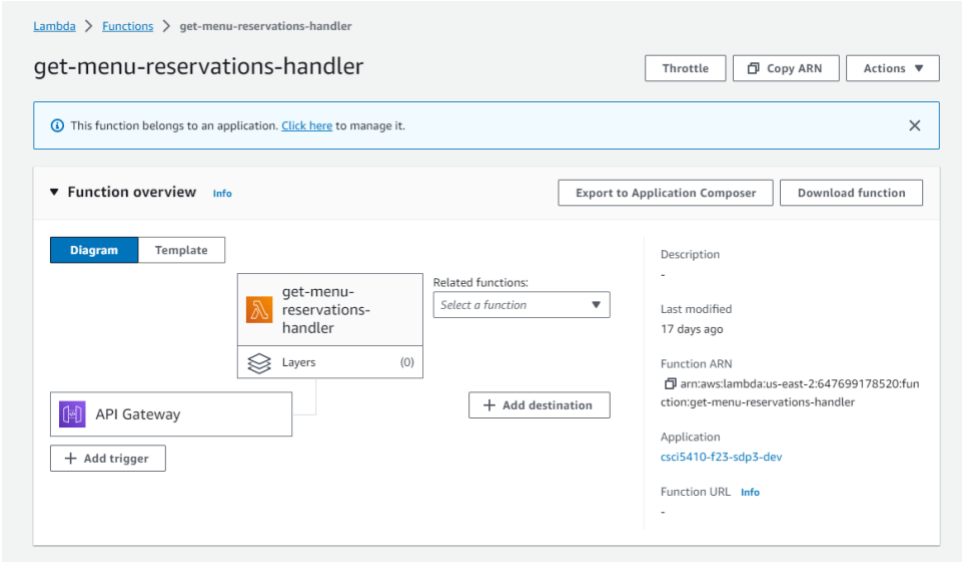


Figure 37 Lambda function to fetch menu reservations. [17]

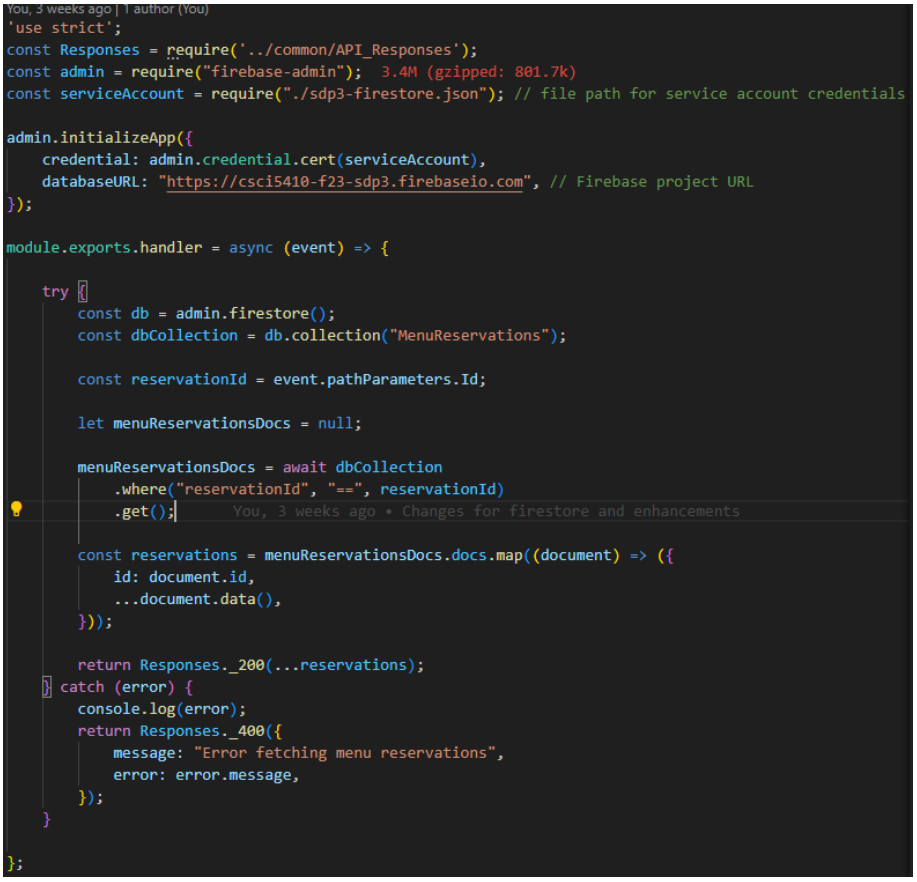


Figure 38 Lambda function code for fetching menu reservations.

- API Gateway Configuration:
  - Configure API Gateway endpoints to trigger the respective Lambda functions for fetching reservations.

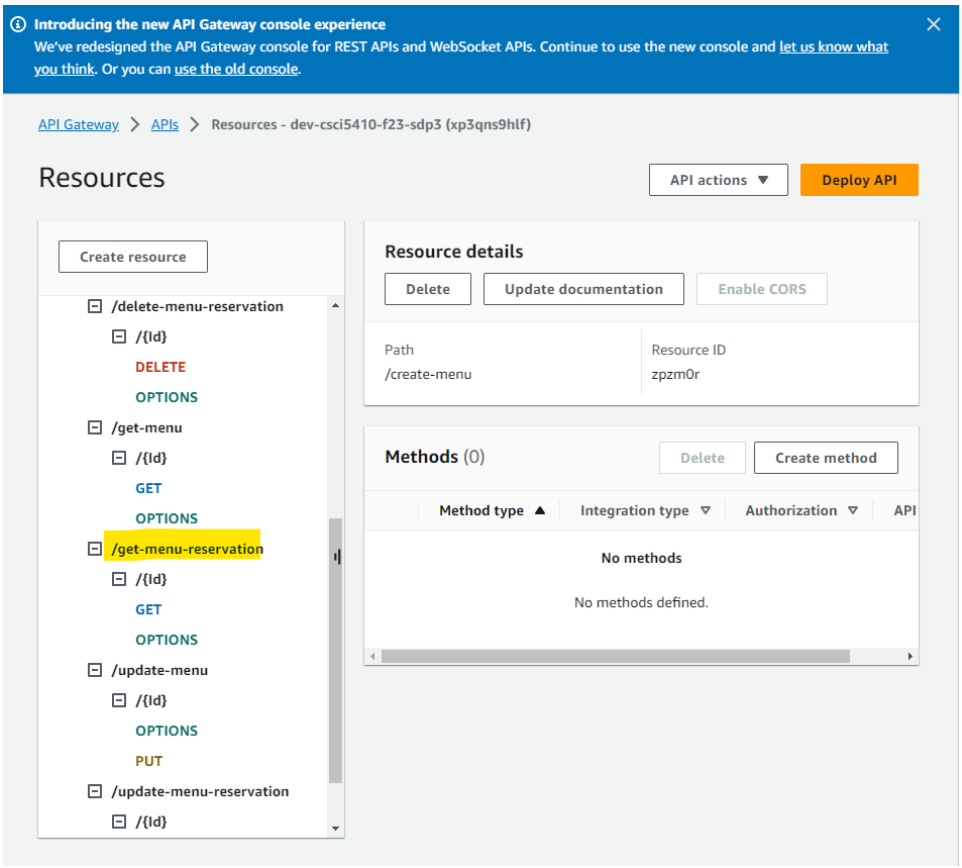


Figure 39 API endpoints in API gateway for fetching menu reservations.

- Frontend Integration:
  - Implement frontend functionalities to display menus associated with user reservations.
  - Develop logic to fetch menu data from the API and display it in the application interface.

4.3 Test Cases / Use Cases

Table:6 Booking Test Cases

Test Case ID	Description	Steps to Execute	Expected Result	Pass/Fail
TC_BOOK_001	Successful reservation creation	1. Navigate to 'Make a Reservation' section.	New reservation is created successfully.	Pass

		2. Fill in required details. 3. Submit reservation request.		
TC_BOOK_002	Booking with incomplete information	1. Attempt to book without filling mandatory fields.	System prompts to fill in all mandatory fields.	Pass
TC_BOOK_003	Booking with invalid date/time	1. Input past date/time for reservation.	System displays an error message for invalid date/time.	Pass

Table: 7 Editing and Deleting Test Cases

Test Case ID	Description	Steps to Execute	Expected Result	Pass/Fail
TC_EDIT_001	Successful reservation modification	1. Access 'My Reservations'. 2. Choose 'Edit'. 3. Modify reservation details. 4. Save changes.	Reservation details are updated successfully.	Pass
TC_EDIT_002	Deletion of an existing reservation	1. Access 'My Reservations'. 2. Choose 'Cancel'. 3. Confirm cancellation.	Reservation is successfully deleted.	Pass
TC_EDIT_003	Editing other user's reservation	1. Attempt to modify another user's reservation.	System prohibits unauthorized modification.	Pass

Table: 8 Viewing Menu Test Cases

Test Case ID	Description	Steps to Execute	Expected Result	Pass/Fail
TC_MENU_001	Accessing menu for an existing reservation	1. Click on 'View Menu' for an existing reservation. 2. Check displayed menu items.	Correct menu items associated with the reservation are shown.	Pass
TC_MENU_002	Viewing menu without an existing	1. Attempt to view the menu without an active	System displays a message to make a reservation	Pass

	reservation	reservation.	first.	

4.4 Services Used

- **AWS Lambda:** Used for serverless computing to execute code for various functionalities.
- **Firestore:** Utilized as a real-time database for storing reservation-related data.
- **DynamoDB**: Employed as a NoSQL database to manage menu-related information.
- **API Gateway:** Serves as an entry point for APIs and manages their interactions.
- **S3 (Simple Storage Service):** Used to store and serve menu item images associated with reservations.

5. Chatbot – (Gauravsinh Bharatsinh Solanki)

5.1 Planning

Objective: Develop a customer-facing chatbot application for table reservations, providing an intuitive and efficient user experience.

Requirements Gathering: Held brainstorming sessions with the team to determine feature sets.

Sprint Planning:

Divided the project into two major sprints focusing on foundational and advanced features.

Defined sprint goals, with an emphasis on producing a minimum viable product (MVP) in the first sprint.

Design:

Created wireframes for the chatbot interface to visualize the customer interaction flow.

Developed a data model outlining how customer data would be handled and stored.

5.2 Implementation

Sprint 1 for Customer App: Foundation Building

Implemented core chatbot functionalities including greeting users, presenting restaurant options, and taking basic reservation details.

Set up the AWS environment, including AWS Lex for the chatbot and AWS Lambda for backend processing.

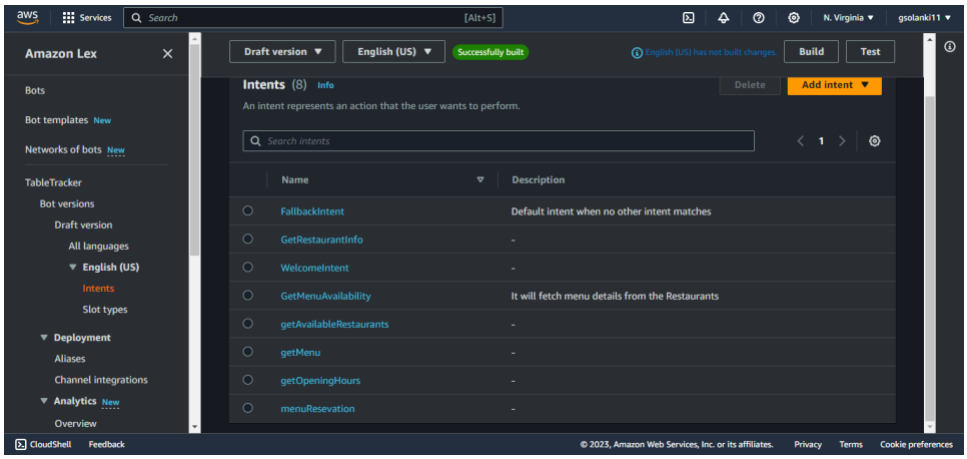


Figure 40 Intent Implementation for the Customer Application.

Sprint 2 for Customer App: Advanced Features

Added complex features such as querying reservation status, modifying reservations, and handling cancellations.

Enhanced the chatbot's natural language processing abilities for a more conversational user experience.

Backend Integration:

Developed **11 Lambda** functions for various backend tasks such as fetching restaurant details, processing reservations, and managing user queries.

Ensured that Lambda functions were stateless and optimized for performance.

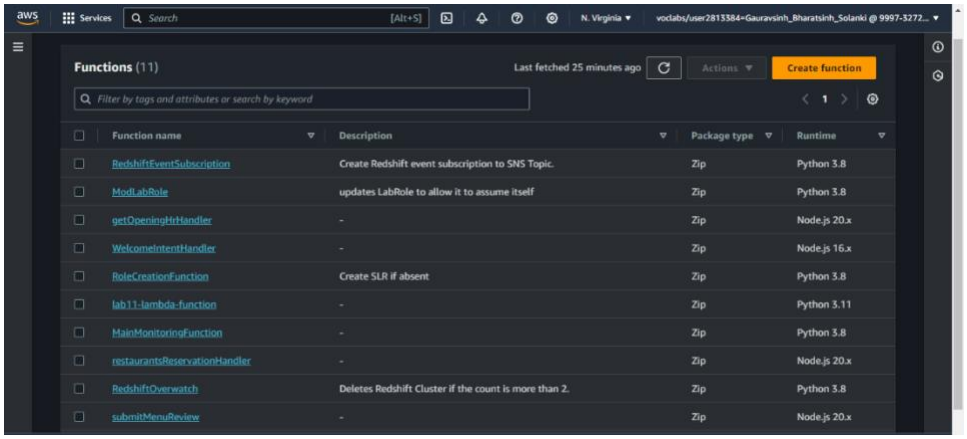


Figure 41 Lambda for the Customer Application.

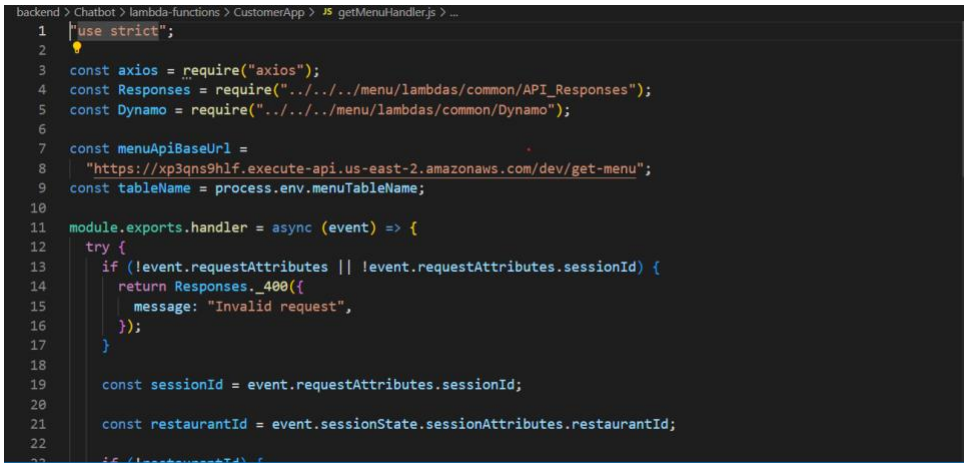


Figure 42 Basic Implementation of One of the Lambda Function. [1]

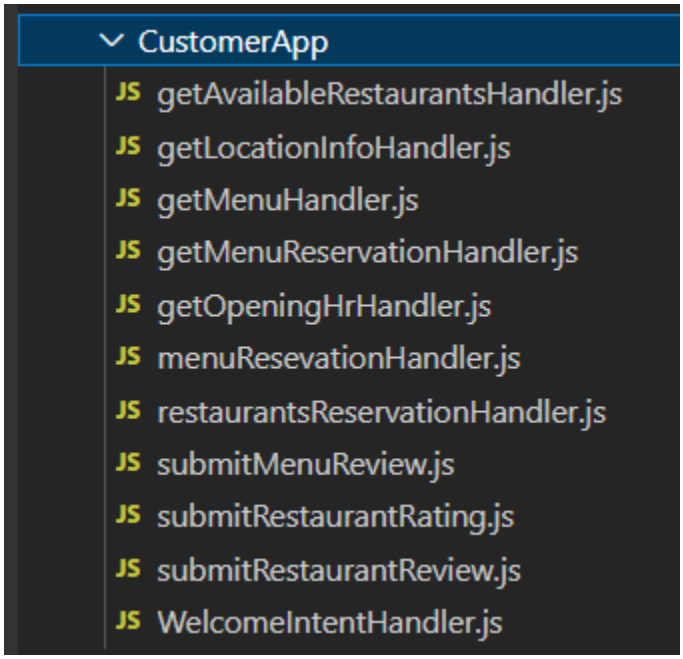


Figure 43 Total 12 Lambdas for the Customer Application.

Frontend Development:

Integrated the chatbot within a React-based frontend application.

I am using Kommunicate Chatbot which is third party frontend boiler plate provider for the chatbot. [7]

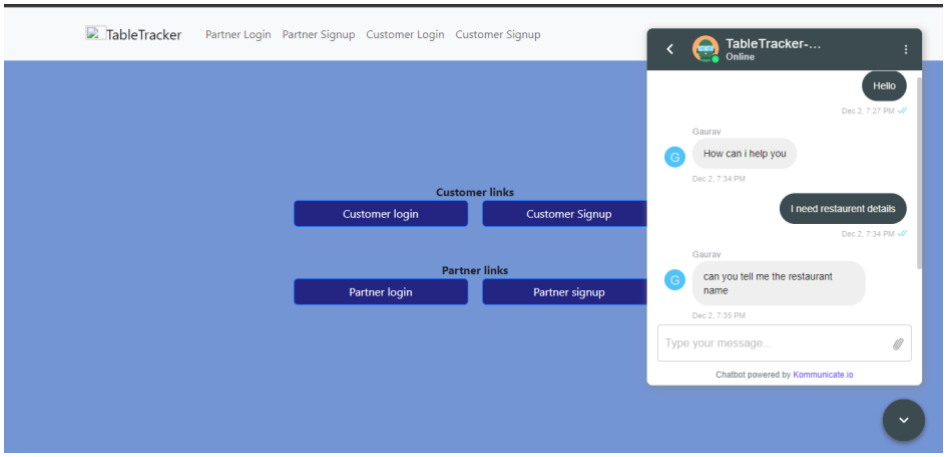


Figure 44 Frontend Integration Chatbot

Ensured responsive design for cross-platform compatibility.

Challenges Faced:

Overcoming the complexities of date and time parsing for reservations across different time zones.

Ensuring the chatbot could handle unexpected user inputs gracefully.

5.3 Test Cases / Use Cases

Test Case: Querying Restaurant Availability

Objective: To validate that the chatbot correctly provides available times for a specified date at a chosen restaurant.

Input: User says, "What times are available at Italian Bistro on Friday?"

Expected Result: The chatbot should respond with available time slots for the upcoming Friday at Italian Bistro.

Test Case: Booking a Reservation

Objective: To ensure that the chatbot can handle the reservation booking process end-to-end.

Input: User provides a specific date, time, and party size for a booking at Italian Bistro.

Expected Result: The chatbot should confirm the reservation and provide a reservation ID.

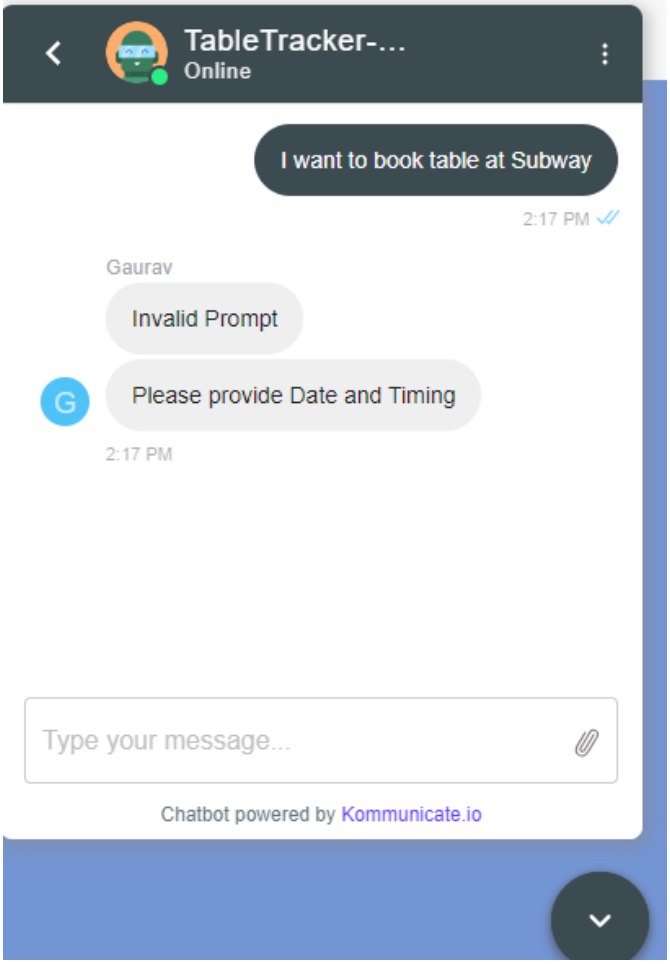


Figure 45 Demo testing

Test Case: Handling Invalid Input

Objective: To test the chatbot's ability to handle unexpected or invalid input during the interaction.

Input: User enters an invalid date format or past date for a reservation.

Expected Result: The chatbot should prompt the user to provide a valid future date.

Test Case: Canceling a Reservation

Objective: To confirm that the chatbot processes cancellations correctly.

Input: User requests to cancel a reservation, providing the correct reservation ID.

Expected Result: The chatbot should acknowledge the cancellation and update the reservation status in the database.

Test Case: Editing a Reservation

Objective: To verify that the chatbot supports editing an existing reservation's details.

Input: User asks to change the time for a specific reservation ID.

Expected Result: The chatbot should update the reservation details and confirm the changes with the user.

Use Cases:

Use Case: Discovering Restaurants

Scenario: A user wants to find a restaurant for dining out on the weekend.

Path: The user interacts with the chatbot to explore restaurant options, filter by cuisine, and view restaurant details, including ratings and reviews.

Use Case: Immediate Reservation Needs

Scenario: A user decides on a last-minute dinner plan and needs to book a table immediately.

Path: The user asks the chatbot for the next available slot at a nearby restaurant and completes the booking process through the chatbot interface.

Use Case: Group Dining Planning

Scenario: A user is planning a birthday dinner for a large group and needs to find a suitable venue.

Path: The user engages with the chatbot to find restaurants that can accommodate large parties, inquiries about group amenities, and books a table for the event.

Use Case: Reservation Management

Scenario: A user needs to manage their upcoming reservations, including confirming times and making changes.

Path: The user reviews their upcoming reservations through the chatbot, confirms the details, and adjusts the booking as needed.

Use Case: Feedback Submission

Scenario: After dining, a user wants to leave feedback about their experience.

Path: The user interacts with the chatbot to submit a review and rating for the restaurant they visited.

5.4 Services Used

AWS Lex 2.0:

Leveraged for building conversational interfaces for the chatbot.

AWS Lambda:

Utilized for creating serverless backend functions that the chatbot interacts with for processing user requests.

Amazon DynamoDB & Firebase Firestore:

DynamoDB was used for storing static content, while Firestore was used for dynamic content requiring real-time updates.

Additional AWS Services:

Amazon API Gateway for RESTful API management.

Development Tools:

Utilized Node.js for writing backend Lambda functions.

Employed the Serverless Framework for deploying and managing AWS Lambda functions.

Version Control:

Used Git for version control, with GitHub as the central repository for code sharing and collaboration.

6. Notifications – (Riya Patel)

6.1 Planning

The proposed architecture aims to use AWS Lambda functions to interact with Firebase and trigger Amazon SNS for email notifications. The key features are as follows:

Hourly Offers and Restaurant Openings: AWS Lambda functions are scheduled to run every hour using CloudWatch Events or EventBridge. These functions query Firebase for recent offers or newly opened restaurants, identify target customers based on preferences or location, and trigger SNS to send notifications about the latest offers or new restaurant openings. 30 Minutes Before Successful Reservation: A timer or an EventBridge event is set to trigger a Lambda function 30 minutes before a successful reservation. This function retrieves reservation details from Firebase, identifies the associated customer, and triggers SNS to send a notification 30 minutes before the reservation time. Menu Item or Reservation Changes: A mechanism is implemented to detect changes to menu items or reservations. Upon detecting changes, a Lambda function is triggered which identifies affected customers, fetches updated information from Firebase, and triggers SNS to notify them about the changes. Restaurant Closure Notification: A mechanism is implemented to detect unexpected restaurant closures. Upon detecting a closure event, a Lambda function is triggered which identifies customers with reservations at the closed restaurant, fetches additional details from Firebase if needed, and triggers SNS to notify affected customers about the closure, providing relevant information.

6.2 Implementation

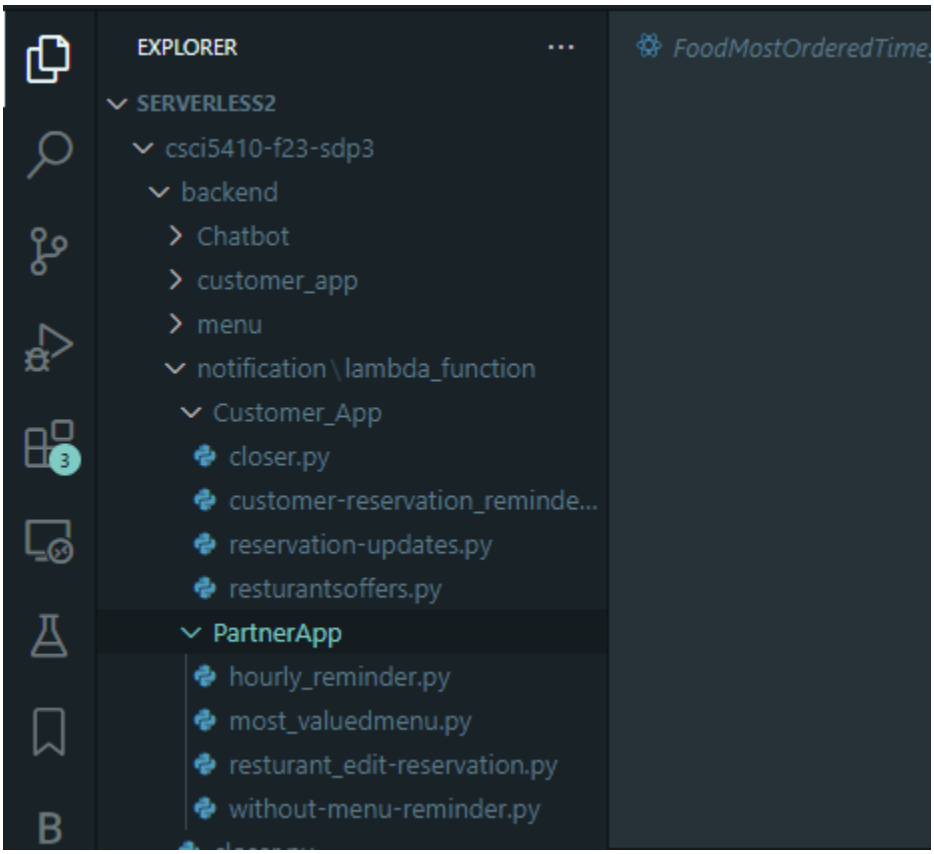


Figure 46 Working Directory for lambda.

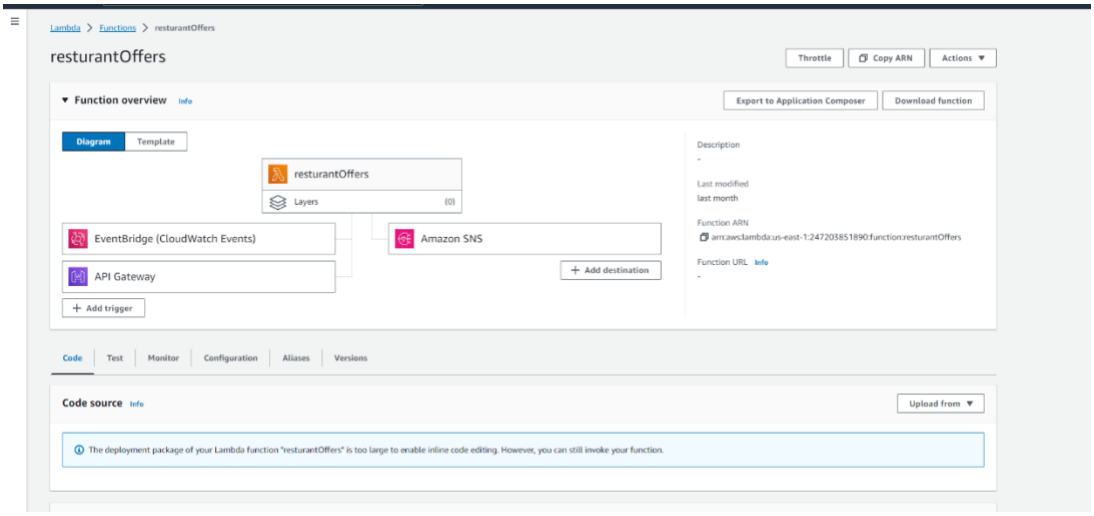


Figure 47 Restaurants' offer & Subscription Lambda [17]

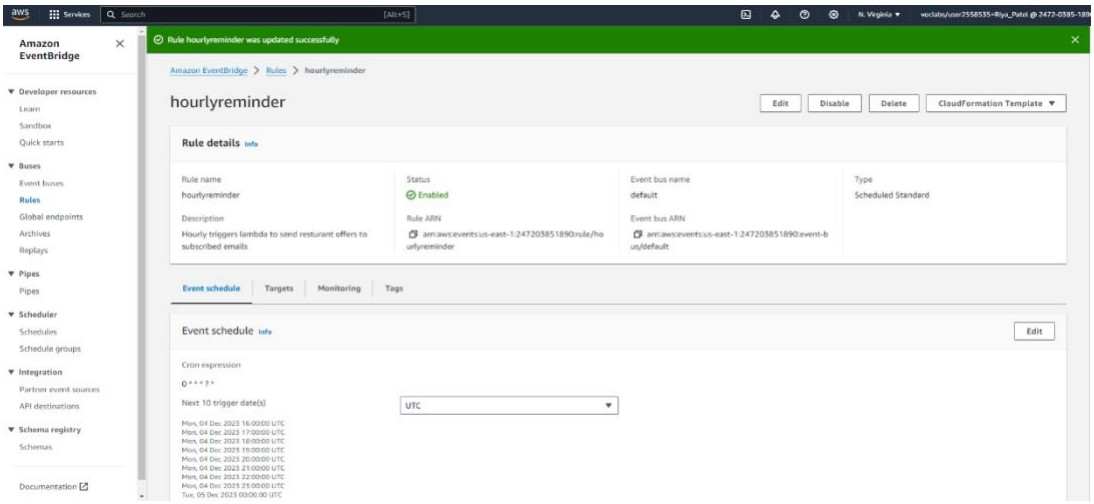


Figure 48 EventBridge for lambda triggering.

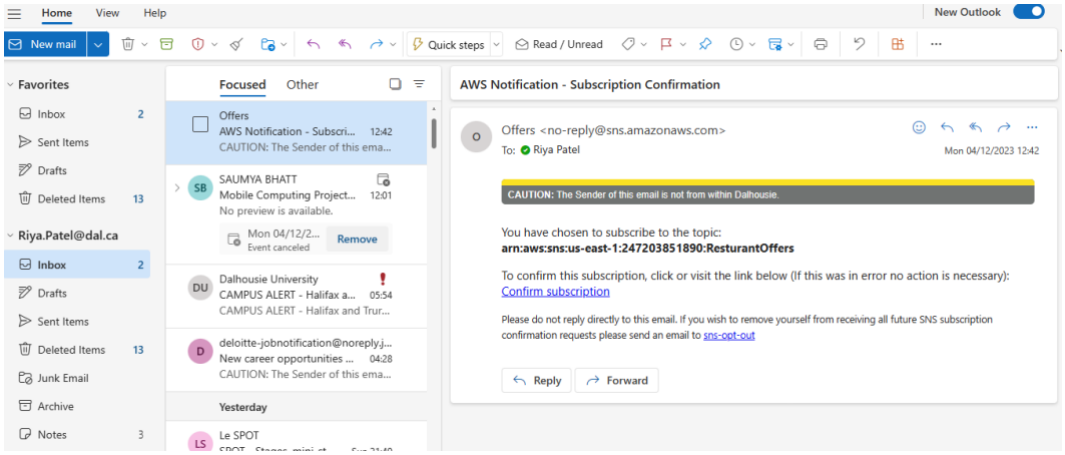


Figure 49 Subscription Email

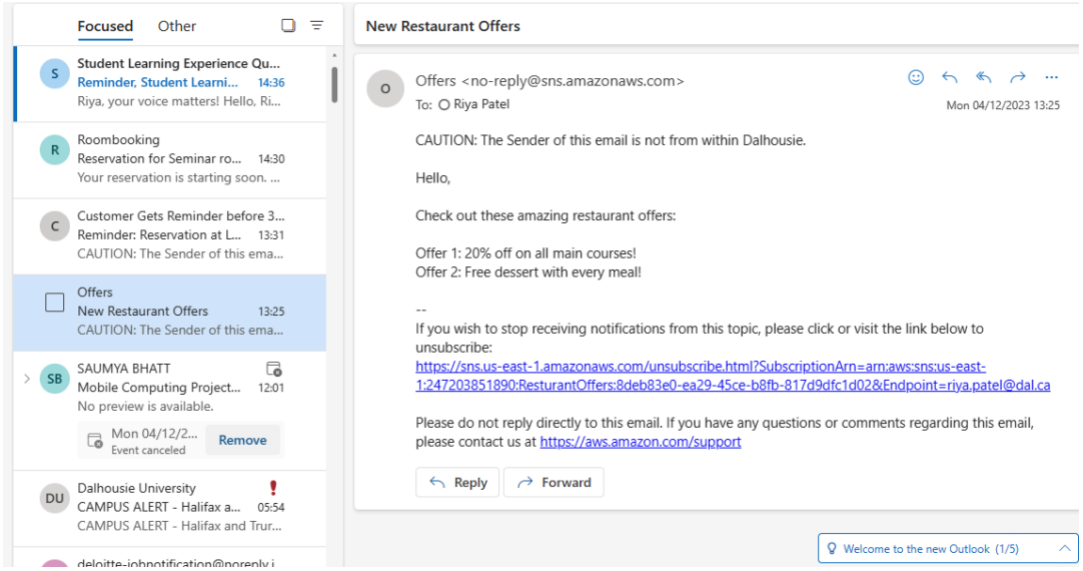


Figure 50 restaurant's offer email.

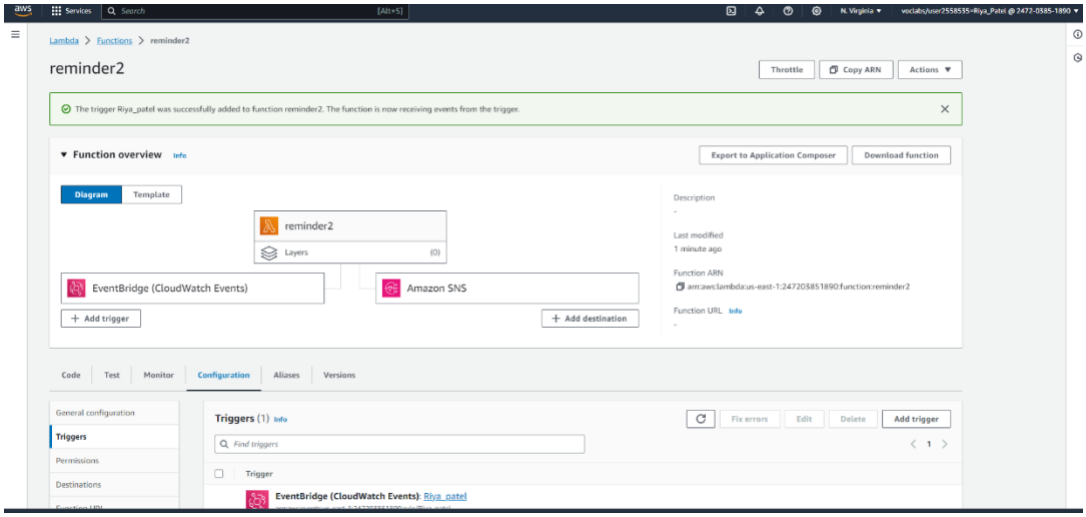


Figure 51 Reservation Reminder Lambda [17]

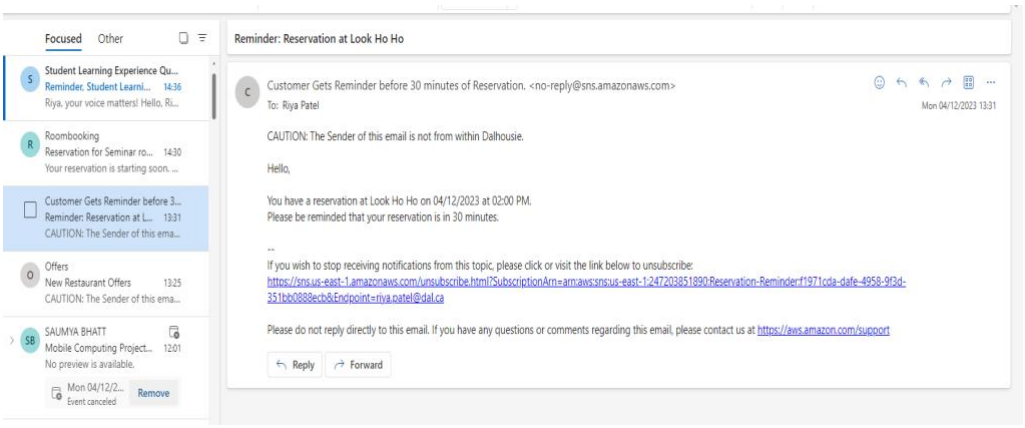


Figure 52 Reservation Reminder Email

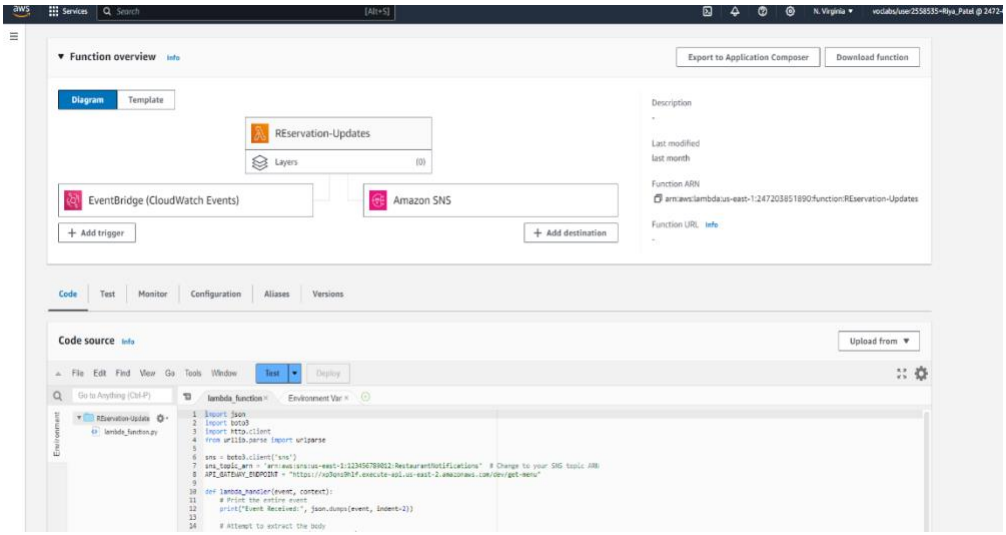


Figure 53 Reservation Updates Lambda [17]

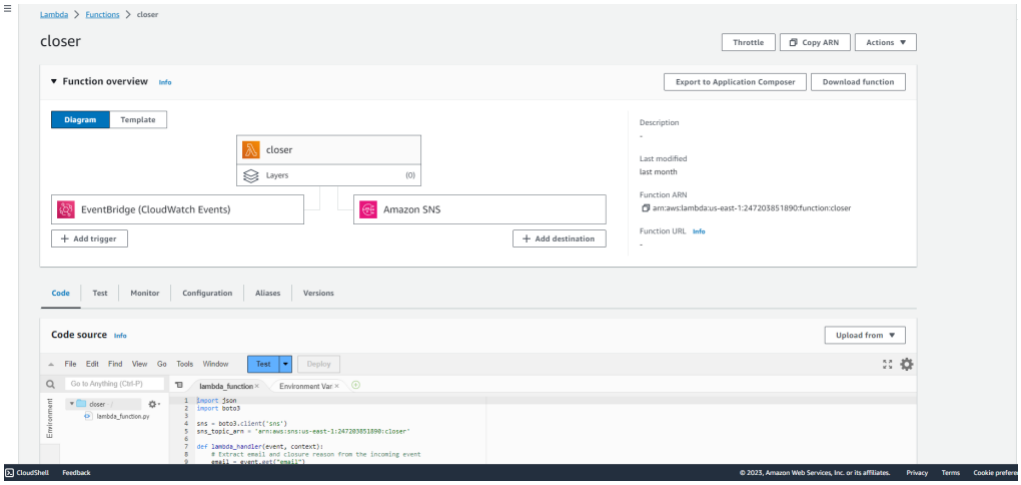


Figure 54 Sudden Closer lambda [17]

6.3 Test Cases

Table: 9 Test Cases for Notification in Customer App

Test Case ID	Description	Input	Output
TC_1	Hourly Offers and Restaurant Openings Trigger	Simulate hourly CloudWatch Event or EventBridge trigger	- Lambda function is triggered. Query Firebase for recent offers and newly opened restaurants. Determine target customers. Verify SNS notifications are sent.
TC_2	30 Minutes Before Successful Reservation Trigger	Simulate EventBridge event scheduled 30 minutes before reservation	- Lambda function is triggered. Retrieve reservation details from Firebase. Identify associated customer. Verify SNS sends notification 30 minutes before reservation.
TC_3	Menu Item or Reservation Change Trigger	Simulate change in menu items or reservations	- Lambda function is triggered. Identify affected customers. Fetch updated information from Firebase. Verify SNS sends notifications about the changes.
TC_4	Restaurant Closure Trigger	Simulate detection of unexpected restaurant closure	- Lambda function is triggered. Identify customers with reservations at the closed restaurant. Fetch additional details from Firebase. Verify SNS sends notifications about closure.

6.4 Services Used

- **AWS Lambda**: Used for serverless execution of backend logic in response to various triggers, such as scheduled events or changes in data.
- **Firebase:** A mobile and web application development platform by Google. In this context, it is used to store and retrieve data related to offers, restaurant openings, reservations, menu items, and closures.
- **Amazon SNS (Simple Notification Service):** A fully managed messaging service for sending notifications to a distributed set of recipients. Used here to send email notifications to customers based on different events.
- **EventBridge:** A serverless event bus service by AWS. It allows you to connect different applications using events. In this context, it is used to trigger Lambda functions at specific intervals or in response to specific events.
- **CloudWatch Events**: A service that enables you to schedule Lambda functions to run at specified intervals. Used here to trigger the hourly event for sending notifications.

- **API Gateway**: To get uid and other data from Frontend.

b. Partner App

7. Sign Up & Login Module – (Rashmi Goplani)

7.1 Planning

Objective: The Sign up and Log in module of the application allows the partner to easily sign up and login to our application to register their restaurant by using one of the two methods:

1. Sign up/Login using email and password
    a. Users can create an account and log in using their restaurant email and a password.
    b. This method is a standard form of authentication and is commonly used for user accounts.
2. Sign up/Login using identity providers - Google sign-in
    a. Users can use their restaurant's Google credentials to sign in.
    b. Leveraging identity providers, such as Google, enhances user convenience by eliminating the need to remember multiple login credentials.

To implement this module, we planned to use Firestore as we did in the Customer App

7.2 Implementation

To implement the Sign-Up and Login feature, I used the Firestore database which was previously used while building the customer app. The user type that was stored in the userDetails table was 'partner.'

Partner Sign-up (Email/Password)

- Capture user input from the registration form (email and password).
- Used **createUserWithEmailAndPassword** function to create a new partner account. This function securely stores the partner's provided email and password in Firebase Authentication. [12]

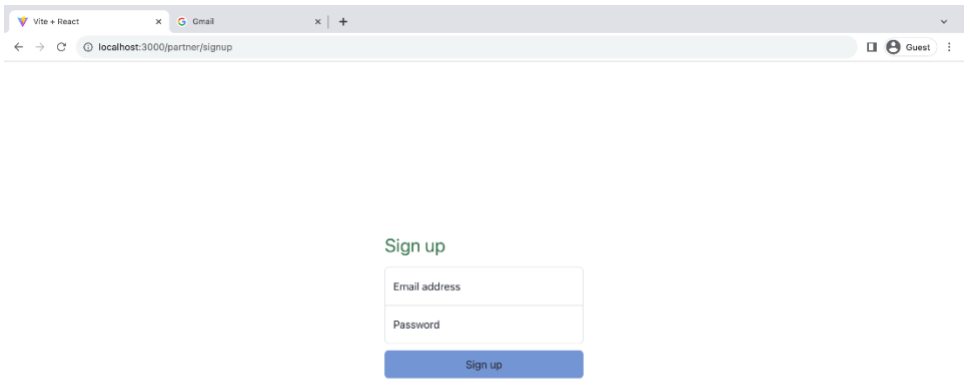


Figure 55 Partner Sign-up (Email/Password)

User Log-in (Email/Password)

- Capture user input from the registration form (email and password).
- Used **signInWithEmailAndPassword** function to authenticate the user. his Firebase Authentication function checks the entered credentials against those stored in the database, granting access upon successful verification. [R1]

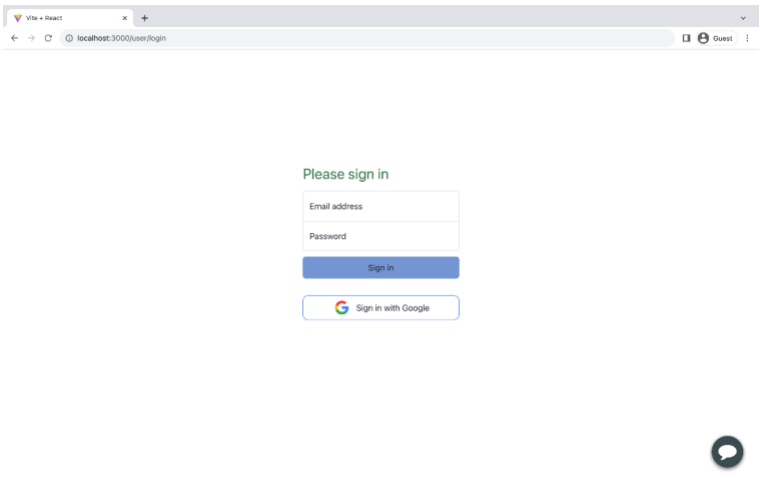


Figure 56 Partner Login (Email/Password)

User Login (Google Sign-in)

- Implemented the function **signInWithGoogle** to handle Google Sign-In. This function is responsible for handling the Google Sign-In process and is typically triggered when users opt for Google as their authentication method. [12]
- Used **signInWithPopup** function with the Google provider to initiate the Google Sign-In process. Paired with the Google provider, this initiates a pop-up window prompting users to sign in with their Google credentials. Upon successful authentication, users gain access to the application [12]

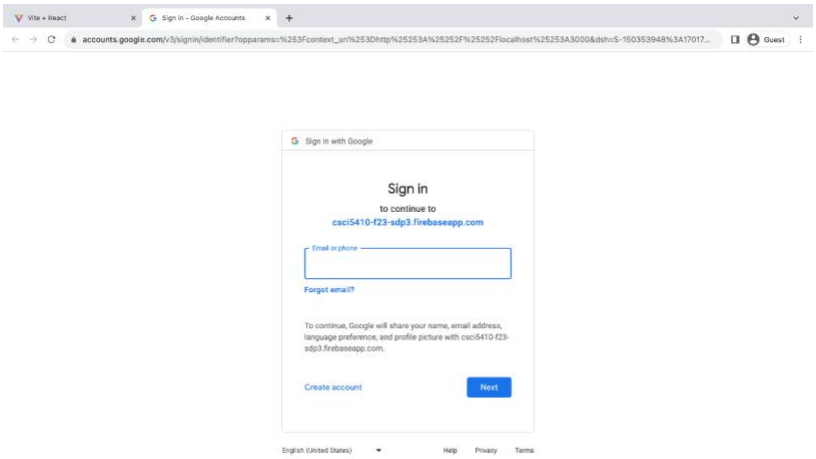


Figure 57 Partner Login using Google.

As soon as a partner sign's up on the application, the user details are store in the userDetails collection with the user type - partner.

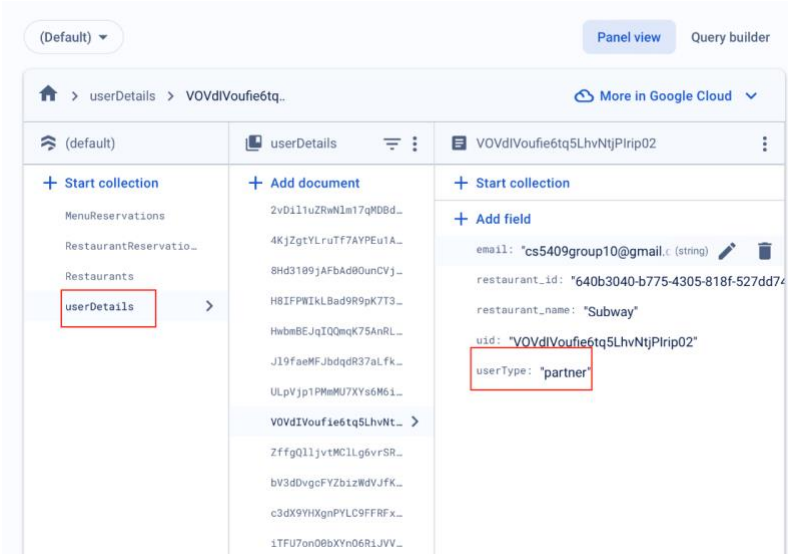


Figure 58 Partner details stored in userDetails Collection.

Frontend Integration

- Implemented frontend components allowing partner to Sign up or login using email/password or google.
- Integrate API requests from the frontend to trigger the Firebase Authentication functions to authenticate the partner and allow applications access.

User Authentication on navigating routes:

- Whenever any partner accesses a particular route, it is first checked if the partner is logged in.
- If the partner is not logged in, the partner is redirected to the login page.
- If the partner is logged in, the partner is allowed to access that route.

7.3 Test Cases / Use Cases

Table: 10 Test Cases for Partners' Sign In / Sign Up

TC_No	Test Case	Result
TC_Partner_Signup_1	Enter email with incorrect format	Displays error message - 'Please include an '@' in the email address. 'adcc' is missing an '@''
TC_Partner_ Signup_2	Submit empty form	Displays error message - 'Please enter email and password'
TC_Partner_ Signup_3	Submit email and password in correct format	User is created and redirected to login page
TC_Partner_ Login_1	Enter incorrect credentials	Displays error message - 'Invalid Credentials'
TC_Partner_ Login_2	Submit empty form for login	Displays error message - 'Please enter email and password'
TC_Partner_ Login_3	Submit the correct login credentials	Partner redirected to Restaurant List page
TC_Partner_ Login_4	Click on Sign-in with Google	Popup opens to sign in using google credentials
TC_Partner_ Login_5	Enter invalid google credentials	Google gives invalid credentials error and user remains on the login page on our app
TC_Partner_ Login_6	Partner enters valid google credentials	Popup is closed and partner is redirected to Restaurant List page

7.4 Services Used

Firestore – To allow partner authentication and store partner information [8]

8. Restaurant details – (Preeti Sharma)

8.1 Planning

Objective: The partner can sign-up and then sign-in to create their restaurant.

Feature: Listing the name of the restaurant and highlighting its essential details like name, phone number, address, opening hours and closing hours.

Partner's restaurant has a view menu and edit restaurant details buttons. The menu button takes the partner to the menu page of the restaurant and the edit restaurant button takes them to the form to edit the details.

Workflow:

- **Partner Access:**
  - Partner opens the partner app by signing up and logging in.
- **Register Restaurant Form:**
  - The application takes the partner to the restaurant registration form.
  - The form is displayed on the partner's screen asking them to fill the essential details like name, address, social media handles, maximum number of tables, and opening and closing hours of the restaurant.
  - Partner can enter these details and become a part of the application.
- **Dashboard:**
  - Partner is taken to the Dashboard of the partner app.
  - They can see the Monthly trends, weekly trends, and daily views.
- **Restaurant Page:**
  - Restaurant page shows the restaurant's information on the left like name, phone number and address.
  - The right side of the page includes the rating, maximum tables, opening and closing hours and the reviews.
- **Buttons:**
  - View Menu button redirects the partner to the view menu page to view and edit the menu details.
  - Edit Restaurant details button opens the edit restaurant form where the partner can update their restaurant details.

8.2 Implementation

- AWS Lambda Setup:
  - Lambda functions to handle create and update restaurant.
  - Integrate Lambda functions with Firestore to create and update restaurant.

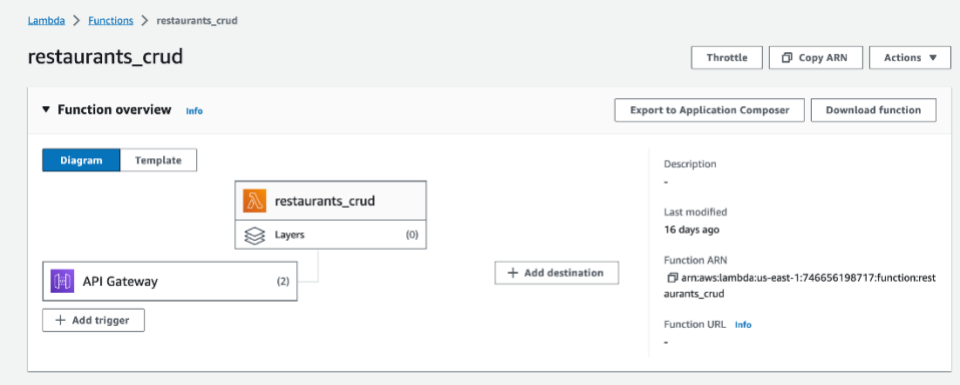


Figure 59: Lambda for restaurant CRUD operations.[17]

```
try {
    switch (event.routeKey) {
        case "DELETE /restaurants/{restaurant_id}":
            await dynamo.send(
                new DeleteCommand({
                    TableName: tableName,
                    Key: {
                        restaurant_id: event.pathParameters.restaurant_id,
                    },
                })
            );
            body = { restaurant_id: event.pathParameters.restaurant_id };
            break;
        case "GET /restaurants/{restaurant_id}":
            body = await dynamo.send(
                new GetCommand({
                    TableName: tableName,
                    Key: {
                        restaurant_id: event.pathParameters.restaurant_id,
                    },
                })
            );
            body = body.Item;
            break;
        case "GET /restaurants":
            body = await dynamo.send(
                new ScanCommand({ TableName: tableName })
            );
            body = body.Items;
            break;
        case "PUT /restaurants":
            let requestJSON = JSON.parse(event.body);
            let ruuid = requestJSON.restaurant_id ?? uuid();
            let muuid = requestJSON.menu_id ?? uuid();
            await dynamo.send(
                new PutCommand({
                    TableName: tableName,
```

Figure 60 Code for Lambda for restaurant CRUD operations.

- API Gateway Setup:
  - Configure API Gateway endpoints to trigger the respective Lambda functions for restaurants.

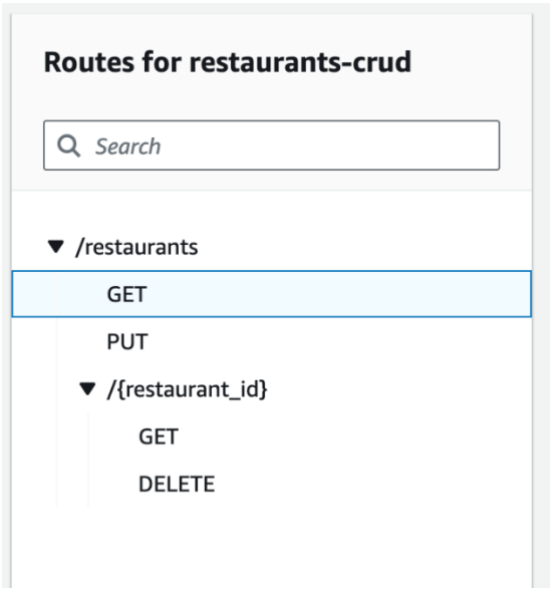


Figure 61: API gateway routes setup for CRUD operations on restaurants.

- Frontend Integration
  - Implemented frontend components allowing partners to create and edit their restaurant.
  - Integrate API requests from the frontend to trigger the Lambda function via API Gateway.

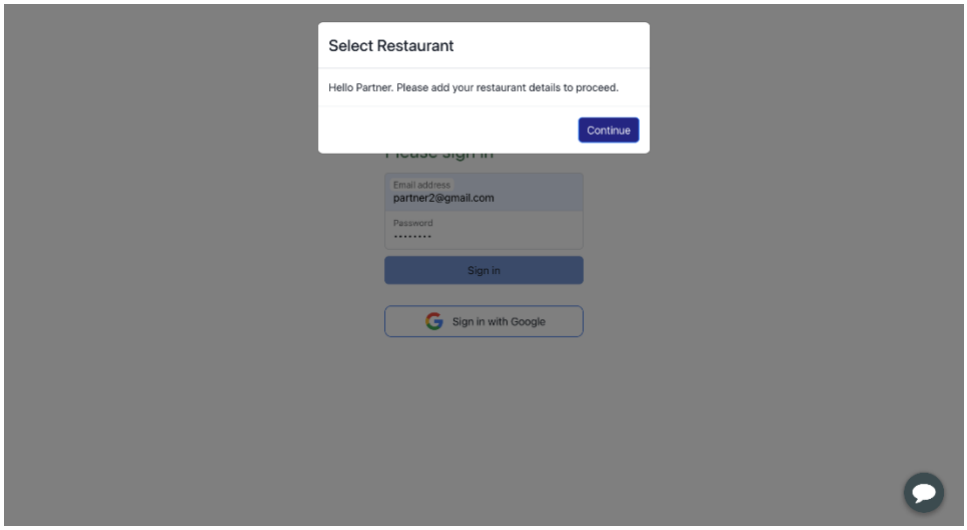


Figure 62: Welcoming partner upon successful login

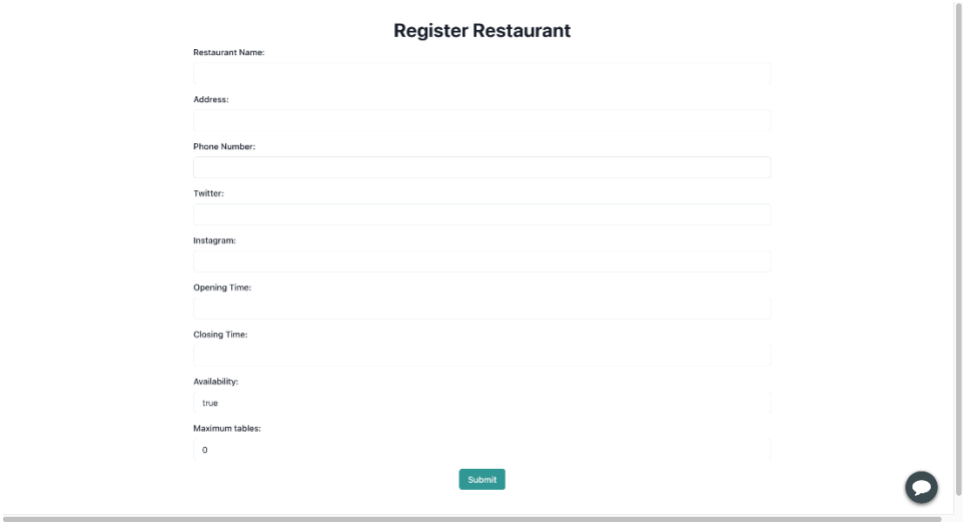


Figure 63: Restaurant Registration form

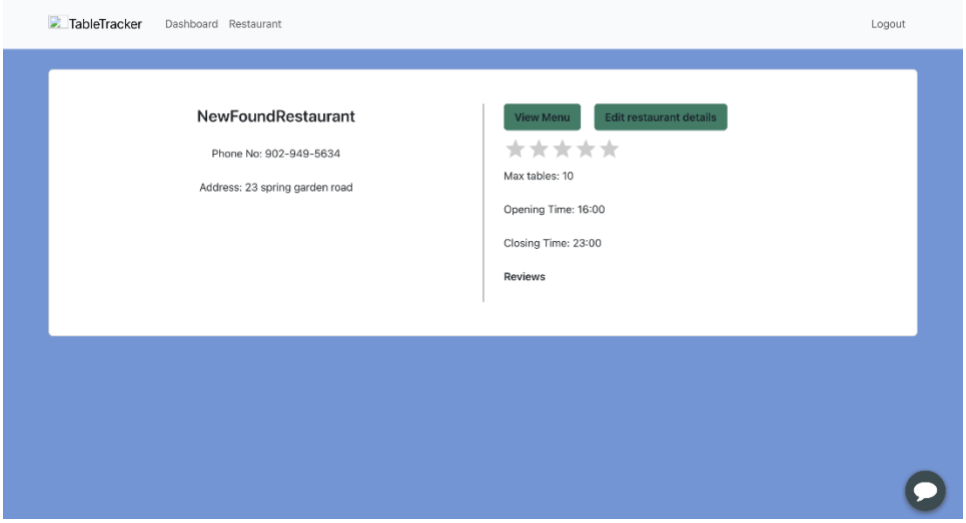


Figure 64: Partner's restaurant page

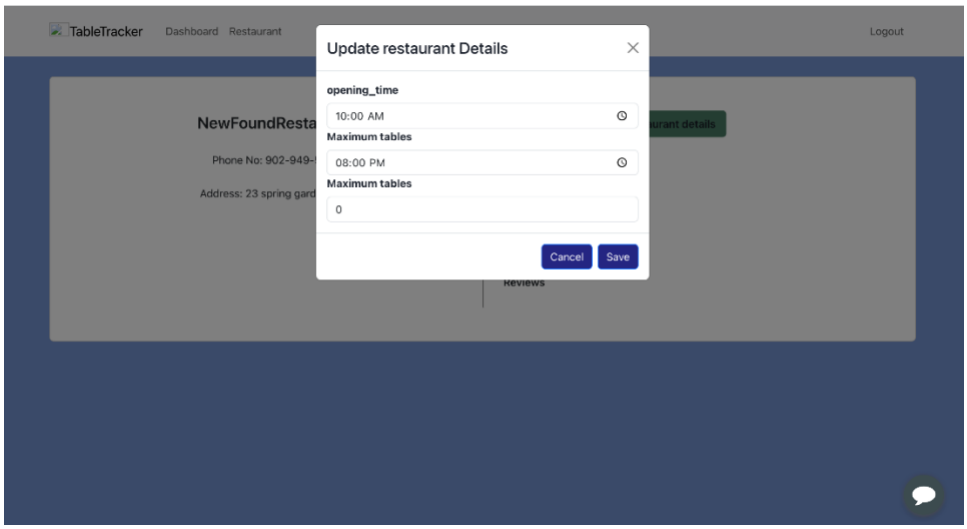


Figure 65: Update restaurant details.

8.3 Test Cases / Use Cases

Table: 11 Restaurant List Test Cases

Test Case ID	Description	Steps to Execute	Expected Result	Pass/Fail
TC_REST_DET_001	Successful login and submission of the restaurant registration form	1. Signup for the application. 2. Login with the credentials. 3. Enters the restaurant details in the form.	Partner successfully submits the form.	Pass
TC_ REST_DET _002	Viewing the restaurant page.	1. Partner opens the restaurant page 2. All the information about the restaurant loads.	System presents the partner with options to view menu and edit restaurant details.	Pass
TC_ REST_DET _003	Updating restaurant details	1. Partner clicks on the edit restaurant details button. 2. Partner updates the details and clicks on save.	System updates restaurant details and shows on the restaurant page.	Pass

8.4 Services Used

AWS Lambda: Code for different functionalities is executed for serverless computing.

Firestore: Used to store restaurant data in real-time database format.

DynamoDB: Used to manage restaurant data as a NoSQL database.

API Gateway: Coordinates API interactions and acts as an entry point.

S3 (Simple Storage Service): A list of images of restaurants is stored and served via S3 (Simple Storage Service).

9. view, edit, and delete a reservation – (Dheeraj Bhat)

9.1 Planning

Workflow:

- **Restaurant Reservation approval/rejection**:
  - Partner can approve/reject a customer's reservation detail via the web application.
  - Web application triggers the Rest API to approve/reject a customer's restaurant reservation.
  - Lambda function triggered via API Gateway, verifies, and updates the restaurant reservation details in Firestore.
- **Editing and Deleting Reservations:**
  - Partners can edit or delete their restaurant's reservations through the frontend. They can only edit a reservation one hour before the reservation date.
  - Web application triggers the Rest API to edit/delete the restaurant reservation.
  - Lambda function triggered via API Gateway, updates, or removes the restaurant reservation in the Firestore collection.
- **Viewing Restaurant Reservations:**
  - Partners request to view the reservations for their restaurant.
  - Web application triggers the Rest API to fetch restaurant reservation(s).
  - Lambda function triggered via API Gateway, fetches the restaurant reservations belonging to the partner's restaurant from the Firestore collection. The UI displays the reservations to the partner.
- **Integration Testing:**
  - Test scenarios to validate end-to-end functionality, ensuring smooth interactions between Lambda, Firestore, API Gateway, and S3.
- **Security Measures:**
  - Implement authentication and authorization mechanisms (e.g., AWS Cognito, IAM) to *control user access to reservations and menus.*
  - Ensure encryption and secure access policies for data stored in Firestore, and S3.

The *planning outlines the essential components, data structures, workflow, and potential considerations required to implement the mentioned functionalities using the specified technologies.*

9.2 Implementation

Approve Restaurant Reservation

- AWS Lambda Setup:
  - Create Lambda functions to handle restaurant reservation approval requests. [9]
  - Integrate Lambda functions with Firestore to update restaurant reservation details. [8].

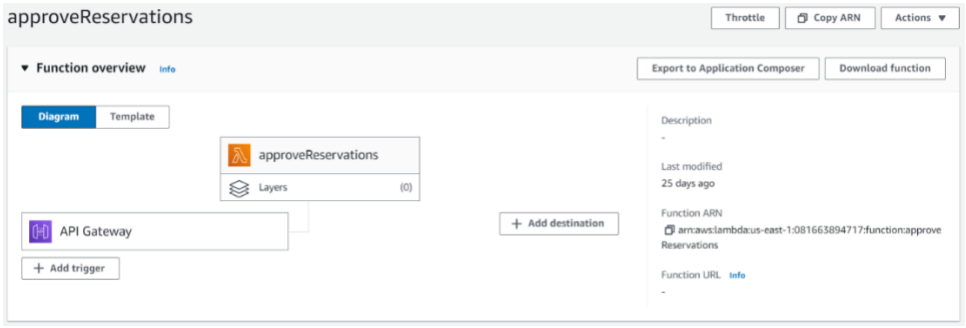


Figure 66: Lambda for approve restaurant reservation.

```
var admin = require("firebase-admin");
var serviceAccount = require("./sdp3-firestore.json");
const Responses = require("./ApiResponses");

admin.initializeApp({
  credential: admin.credential.cert(serviceAccount),
});

const db = admin.firestore();

exports.handler = async (event) => {
  try {
    const reservationDetails = JSON.parse(event.body);
    const { reservationId } = reservationDetails;

    const reservationDocRef = db
      .collection("RestaurantReservations")
      .doc(reservationId);
    const reservation = await reservationDocRef.get();

    if (!reservation.exists) {
      return Responses._400({
        message: "Restaurant reservation does not exist",
      });
    }

    const updatedReservation = { ...reservation.data(), isApproved: true };

    await reservationDocRef.update(updatedReservation);

    return Responses._200({
      message: "Reservation approved",
    });
  } catch (error) {
    console.log(error);
    return Responses._400({
      message: "Error editing restaurant reservations",
      error: error.message,
    });
  }
};
```

Figure 67: Code of Lambda for approve restaurant reservation.

- API Gateway Setup:
  - Configure API Gateway endpoints to trigger the respective Lambda functions for restaurant reservation approval. [10].

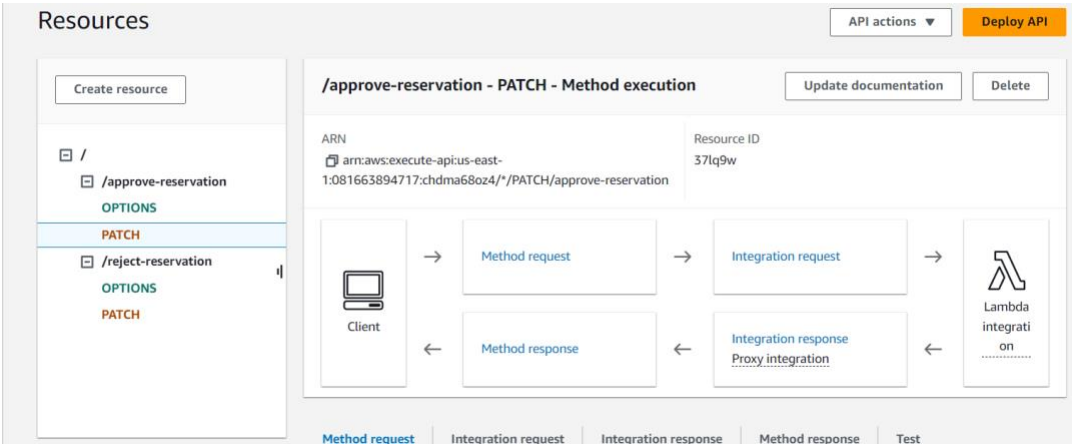


Figure 68: API gateway for approve restaurant reservation.

- Frontend Integration
  - Implemented frontend components allowing users to approve restaurant reservation details.
  - Integrate API requests from the frontend to trigger the Lambda function via API Gateway.

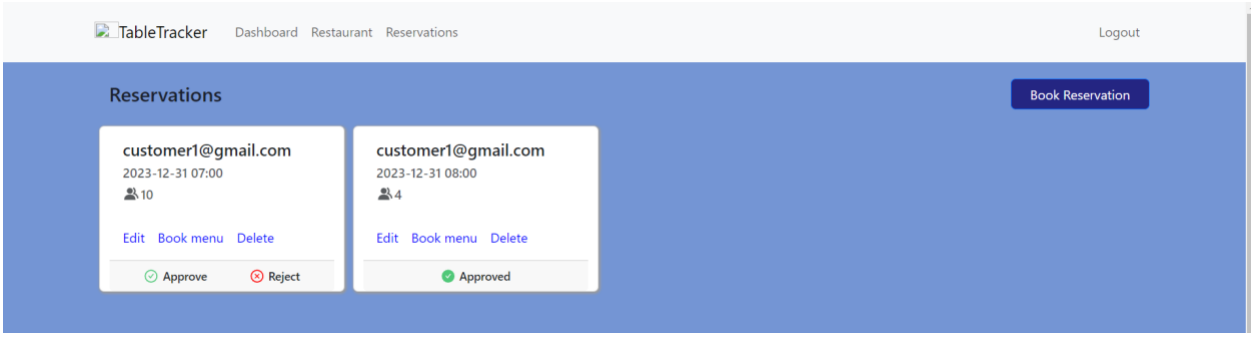


Figure 69: Approve restaurant reservation UI.

Reject Restaurant Reservation

- AWS Lambda Setup:
  - Create Lambda functions to handle restaurant reservation rejection requests. [9].
  - Integrate Lambda functions with Firestore to update restaurant reservation details. [8].

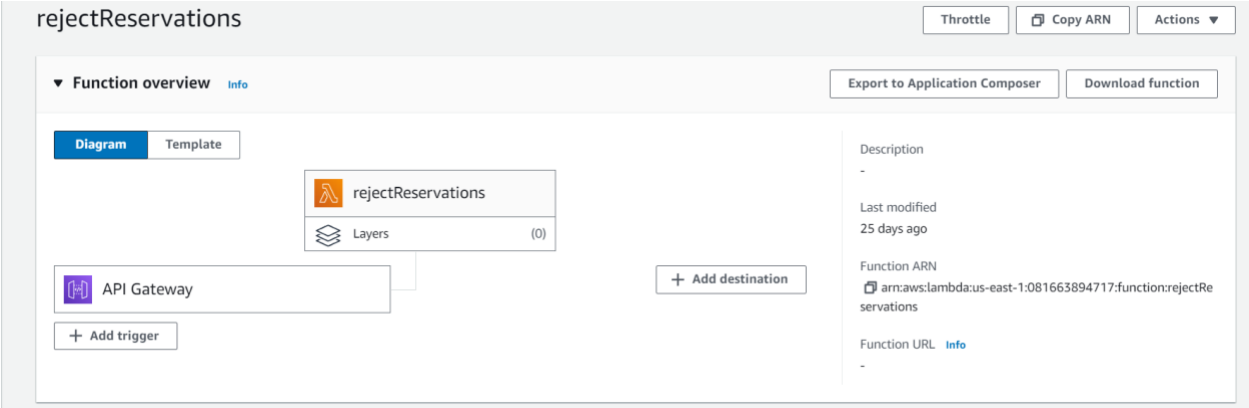


Figure 70: Lambda for reject restaurant reservation.

```
var admin = require("firebase-admin");
const axios = require("axios");
var serviceAccount = require("./sdp3-firestore.json");
const Responses = require("./ApiResponses");

admin.initializeApp({
  credential: admin.credential.cert(serviceAccount),
});

const db = admin.firestore();

exports.handler = async (event) => {
  try {
    const reservationDetails = JSON.parse(event.body);
    const { reservationId } = reservationDetails;

    const reservationDocRef = db
      .collection("RestaurantReservations")
      .doc(reservationId);
    const reservation = await reservationDocRef.get();

    if (!reservation.exists) {
      return Responses._400({
        message: "Restaurant reservation does not exist",
      });
    }

    const updatedReservation = { ...reservation.data(), isApproved: false };

    await reservationDocRef.update(updatedReservation);

    return Responses._200({
      message: "Reservation rejected",
    });
  } catch (error) {
    console.log(error);
    return Responses._400({
      message: "Error editing restaurant reservations",
      error: error.message,
    });
  }
};
```

Figure 71: Code of Lambda for reject restaurant reservation.

- API Gateway Setup:
  - Configure API Gateway endpoints to trigger the respective Lambda functions for restaurant reservation rejection. [10].

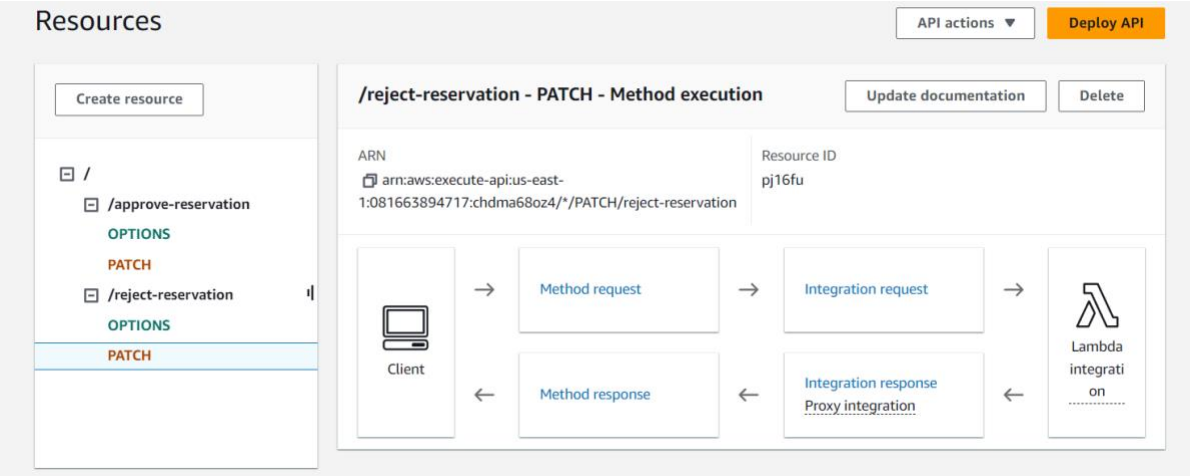


Figure 72: API gateway for reject restaurant reservation.

- Frontend Integration
  - Implemented frontend components allowing users to reject restaurant reservation details.
  - Integrate API requests from the frontend to trigger the Lambda function via API Gateway.

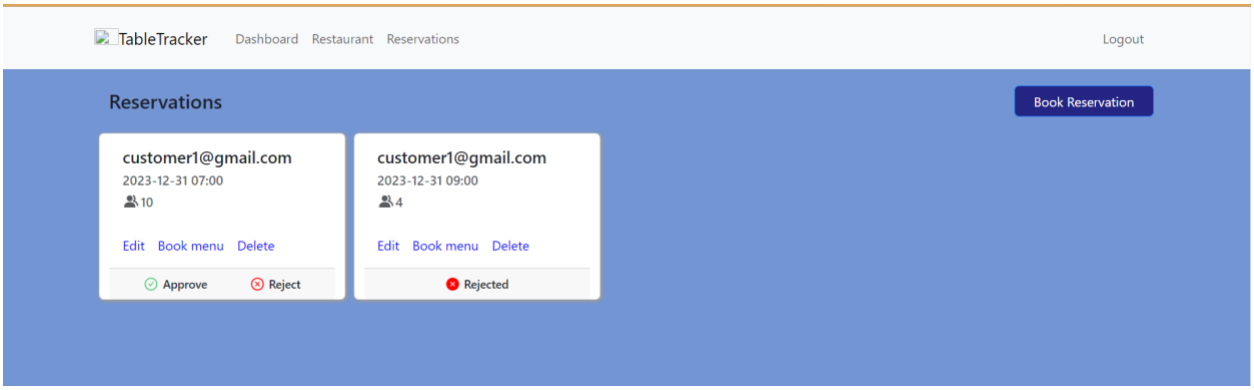


Figure 73: Reject restaurant reservation UI.

Edit Restaurant Reservations

- AWS Lambda Setup:
  - Create Lambda functions to handle restaurant reservation update requests. [9].
  - Integrate Lambda functions with Firestore to update restaurant reservation details. [8].
  - The customer can update a restaurant any time before 1 hour from the reservation.

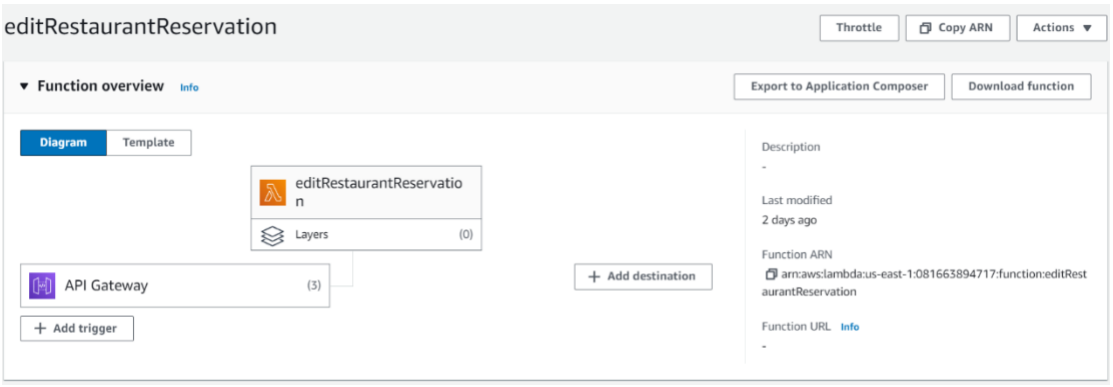


Figure 74: lambda for edit restaurant reservations

```
admin.initializeApp({
  credential: admin.credential.cert(serviceAccount),
});

const db = admin.firestore();

exports.handler = async (event) => {
  try {
    const reservationDetails = JSON.parse(event.body);
    const {
      reservationId,
      restaurantId,
      reservationDate,
      requiredCapacity,
      userId,
      isApproved,
    } = reservationDetails;

---

        await reservationDocRef.update(updatedReservation);

        return Responses._200({
          message: "Reservation edited successfully",
        });
      } else {
        return Responses._400({
          message: "Reservation time is outside the restaurant's opening hours",
        });
      }
    } else {
      return Responses._400({
        message:
          "Reservations can only be edited 1 hour before the reservation time.",
      });
    }
  } catch (error) {
    console.log(error);
    return Responses._400({
      message: "Error booking restaurant reservations",
      error: error.message,
    });
  }
};
```

Figure 75: code of Lambda for edit restaurant reservation.

- API Gateway Setup:
  - Configure API Gateway endpoints to trigger the respective Lambda functions for edit restaurant reservation. [10].

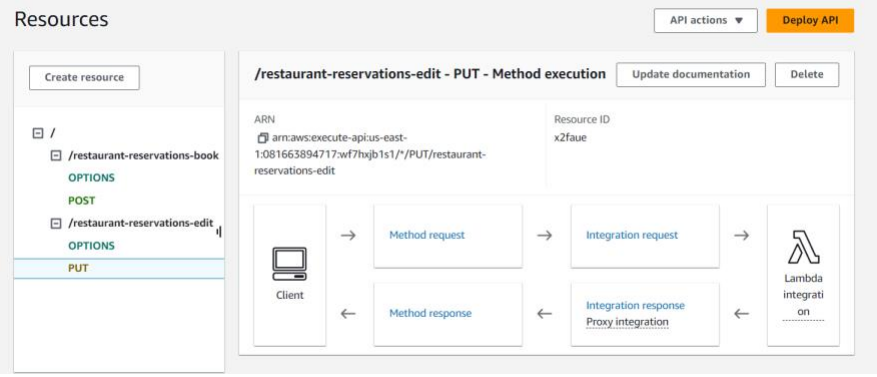


Figure 76: API gateway for edit restaurant reservation.

- Frontend Integration
  - Implemented frontend components allowing users to edit the restaurant reservation details.
  - Integrate API requests from the frontend to trigger the Lambda function via API Gateway.

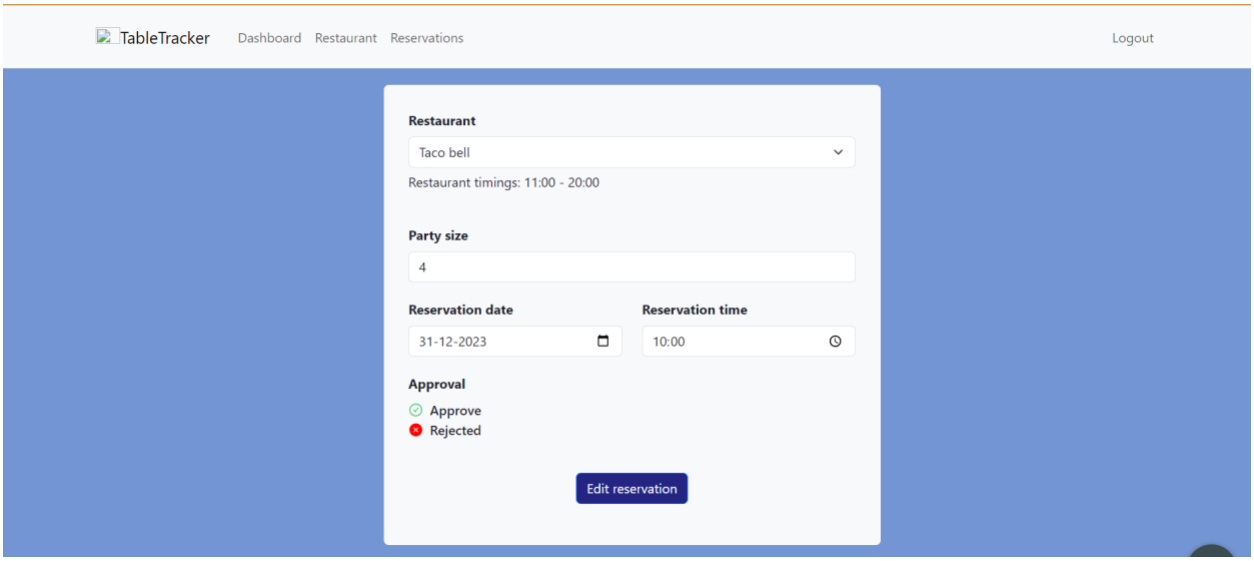


Figure 77: Partner Edit restaurant reservation form.

Delete Restaurant Reservations

- AWS Lambda Setup:
  - Create Lambda functions to handle restaurant reservation delete requests. [9].
  - Integrate Lambda functions with Firestore to delete restaurant reservation details. [8]

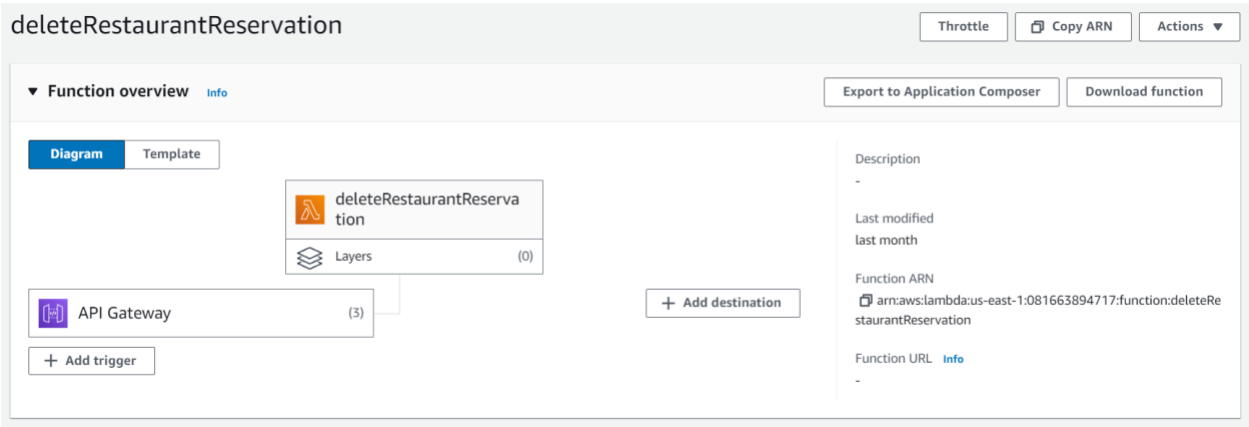


Figure 78: lambda for delete restaurant reservations.

```
var admin = require("firebase-admin");
const axios = require("axios");
var serviceAccount = require("./sdp3-firestore.json");
const Responses = require("./ApiResponses");

admin.initializeApp({
  credential: admin.credential.cert(serviceAccount),
});

const db = admin.firestore();

exports.handler = async (event) => {
  try {
    const reservationId = event.pathParameters.id;

    const reservationDocRef = db
      .collection("RestaurantReservations")
      .doc(reservationId);
    const reservation = await reservationDocRef.get();

    if (!reservation.exists) {
      return Responses._400({
        message: "Restaurant reservation does not exist",
      });
    }

    await reservationDocRef.delete();
    return Responses._200({
      message: "Reservation successfully deleted!",
    });
  } catch (error) {
    console.log(error);
    return Responses._400({
      message: "Error deleting restaurant reservations",
      error: error.message,
    });
  }
};
```

Figure 79: code of Lambda for delete restaurant reservation.

- API Gateway Setup:
  - Configure API Gateway endpoints to trigger the respective Lambda functions for delete restaurant reservation. [10].

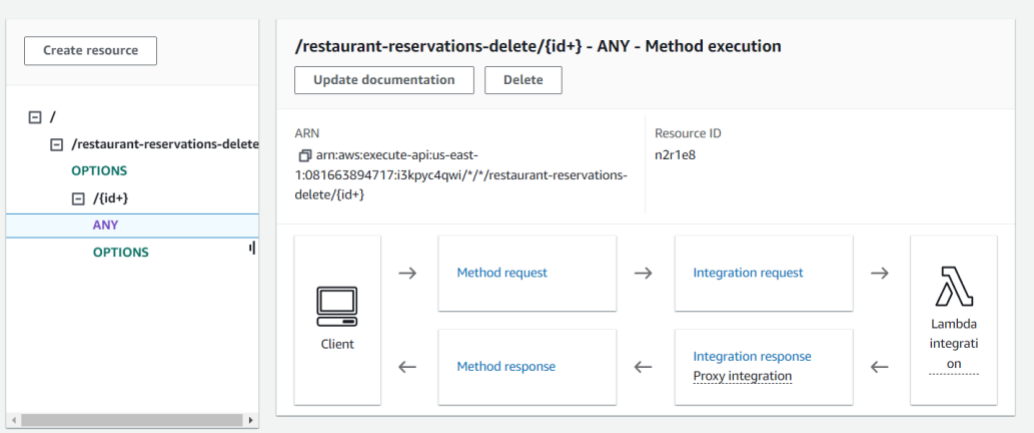


Figure 80: API gateway for delete restaurant reservation.

- Frontend Integration
  - Implemented frontend components allowing partner to delete restaurant reservations.
  - Integrate API requests from the frontend to trigger the Lambda function via API Gateway.

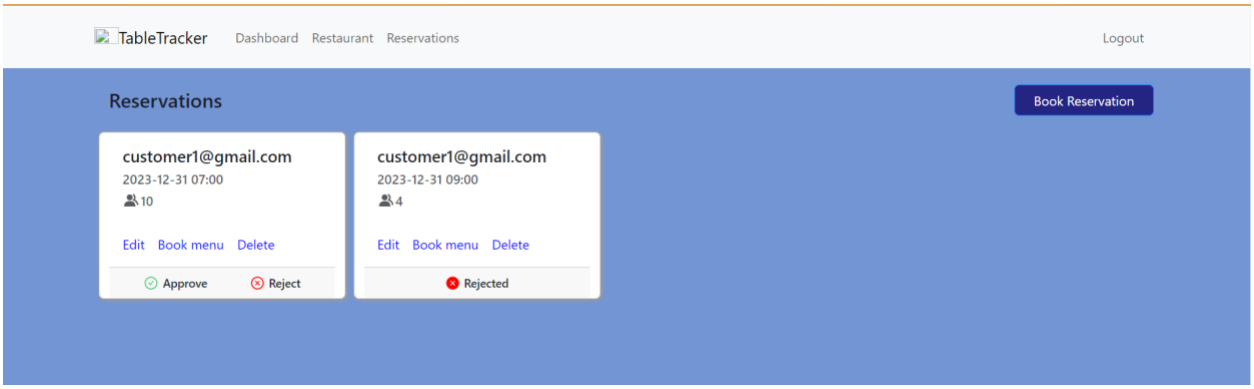


Figure 81: partner delete restaurant reservation UI.

View Restaurant Reservations

- AWS Lambda Setup:
  - Create Lambda functions to fetch restaurant reservations for a partner's restaurant. [9].
  - Integrate Lambda functions with Firestore to fetch all restaurant reservations for the partner's restaurant. [8]

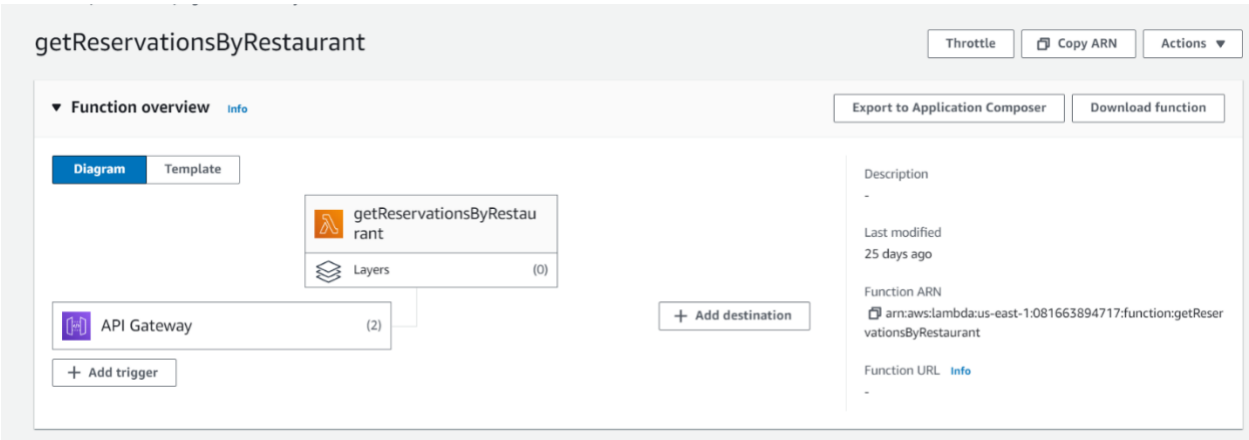


Figure 82: lambda for view restaurant reservations by restaurant id.

```
const admin = require("firebase-admin");
const Responses = require("./ApiResponses");
const serviceAccount = require("./sdp3-firestore.json"); // file path for service account credentials

admin.initializeApp({
  credential: admin.credential.cert(serviceAccount),
  databaseURL: "https://csci5410-f23-sdp3.firebaseio.com", // Firebase project URL
});

exports.handler = async (event, context) => {
  try {
    const db = admin.firestore();
    const userId = event.pathParameters.userId ?? "U1";
    const dbCollection = db.collection("RestaurantReservations");

    const reservationsDocs = await dbCollection
      .where("user_id", "==", userId)
      .get();

    if (reservationsDocs.empty) {
      return Responses._400({
        message: "No reservations found for this user.",
      });
    }

    const reservations = reservationsDocs.docs.map((document) => ({
      id: document.id,
      ...document.data(),
      reservation_date: document.data().reservation_date.toDate(),
    }));

    return Responses._200({
      message: "Fetched reservations successfully",
      data: [...reservations],
    });
  } catch (error) {
    console.log(error);
    return Responses._400({
      message: "Error fetching restaurant reservations",
      error: error.message,
    });
  }
};
```

Figure 83: code of Lambda for view partners restaurant reservations

- API Gateway Setup:
  - Configure API Gateway endpoints to trigger the respective Lambda functions for view restaurant reservation. [10]

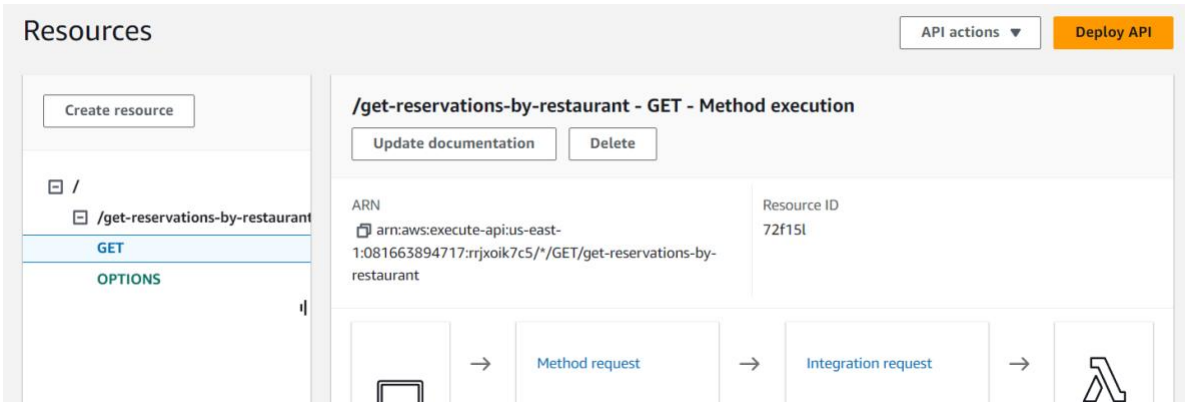


Figure 84: API gateway for view partner restaurant reservations.

- Frontend Integration
  - Implemented frontend components to show partners all their restaurant reservations.
  - Integrate API requests from the frontend to trigger the Lambda function via API Gateway.

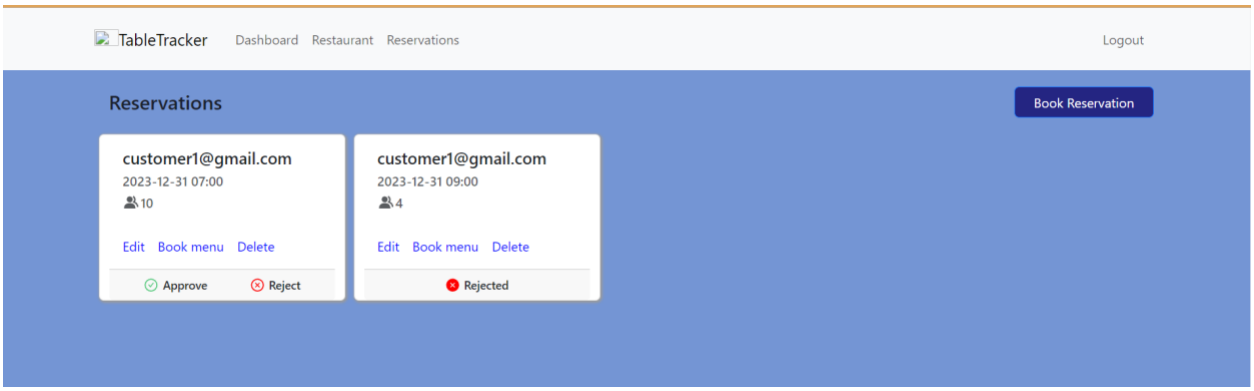


Figure 85: Get partner restaurant reservations UI.

9.3 Test Cases / Use Cases

Table: 12 Approval/Rejection Test Cases:

Test Case ID	Description	Steps to Execute	Expected Result	Pass/Fail
TC_REST_BOOK_001	Successful restaurant reservation approval.	1. Navigate to Reservations page. 2. Approve a reservation	Reservation is updated successfully	Pass
TC_REST_BOOK_002	Successful restaurant reservation rejection.	1. Navigate to Reservations page. 2. Reject a reservation	Reservation is updated successfully	Pass

Table: 13 Editing and Deleting Test Cases

Test Case ID	Description	Steps to Execute	Expected Result	Pass/Fail
TC_REST_RES_EDIT_001	Successful reservation modification	1. Access ‘Reservations'. 2. Choose 'Edit'. 3. Modify reservation details. 4. Save changes.	Reservation details are updated successfully.	Pass
TC_REST_RES_EDIT_002	Deletion of an existing reservation	1. Access ‘Reservations'. 2. Choose Delete.	Reservation is successfully deleted.	Pass
TC_REST_RES_EDIT_003	Editing reservation more than 1 hour before reservation date	1. Attempt to modify reservation.	System prohibits modification.	Pass

Table: 14 Viewing Reservations Test Cases

Test Case ID	Description	Steps to Execute	Expected Result	Pass/Fail
TC_REST_RES_001	Viewing reservations of the partner	1. Navigate to reservations page. 2. Check displayed reservations.	All reservations for partners restaurant shown	Pass
TC_REST_RES_002	Viewing Reservations without an existing reservation	1. Navigate to reservations page. 2. Check displayed reservations.	System displays empty page.	Pass

9.4 Services Used

- **AWS Lambda:** Used for serverless computing to execute code for various functionalities. [9].
- **Firestore:** Utilized as a real-time database for storing restaurant reservation-related data. [8]
- **API Gateway:** Serves as an entry point for APIs and manages their interactions. [10].
- **S3 (Simple Storage Service):** Used to store zip of the lambda code files.

10. Edit, delete, view menu – (Arihant Dugar)

10.1 Planning

The primary goal is to enable restaurants to efficiently manage their menus by providing essential functionalities such as editing, deleting, and viewing menu items. This involves empowering restaurants to add pricing details, including offering discounts on individual menu items or the entire menu. Customers should witness discounted prices when selecting items from participating restaurants, with clear indications displayed in banners for restaurants providing such offers. Additionally, restaurants are expected to have granular control, allowing them to update, create,

read, and delete menu items' availability status. The aim is to offer a user-friendly interface for restaurants to manage their menus effectively while enhancing the customer experience by displaying discounted prices for eligible menu items or entire menus.

Restaurant Menu Structure:


```
{
 "id": "3c6fc9e1-aef2-4f51-b34f-e21c5e53d642",
 "discount": 0,
 "items": [
  {
   "id": 1,
   "availability": true,
   "description": "Pizza",
   "discount": 0,
   "img": "https://menu-items-bucket-csci5410.s3.us-east-2.amazonaws.com/pizza.png",
   "name": "Test",
   "price": "20"
  }
 ],
 "restaurantId": "3c6fc9e1-aef2-4f51-b34f-e21c5e53d642"
}
```


Workflow:

- **Menu Editing:**
  - **View Menu:** Restaurants can access their menu to review all listed items and their details.
  - **Edit Menu Items:** Ability to modify existing menu items, including descriptions, prices, and availability.
  - **Add Pricing Information:** Enable restaurants to set and update prices for menu items.
- **Offer Management:**
  - **Offer Creation:** Restaurants can create offers, either for individual menu items or the entire menu.
  - **Offer Presentation:** Display slashed prices on items participating in offers, allowing customers to see discounted rates.
  - **Banner Notifications:** Highlight restaurants offering deals on the menu, specifying if the offer applies to the entire menu.
- **Availability Control:**
  - **Update Availability:** Modify the availability status of menu items—enabling/disabling item listings.
  - **Create New Items:** Ability to add new items to the menu with associated details.

- **Delete Menu Items:** Capability to remove items no longer offered from the menu.

The planning outlines the essential components, data structures, workflow, and potential considerations required to implement the mentioned functionalities using the specified technologies.

10.2 Implementation

1. **Menu Management Interface:**
    a. Development: Create an intuitive and user-friendly interface allowing restaurant partners to manage their menus efficiently.
    b. Functionality: Enable features for editing, deleting, and viewing menu items through a user-friendly interface.

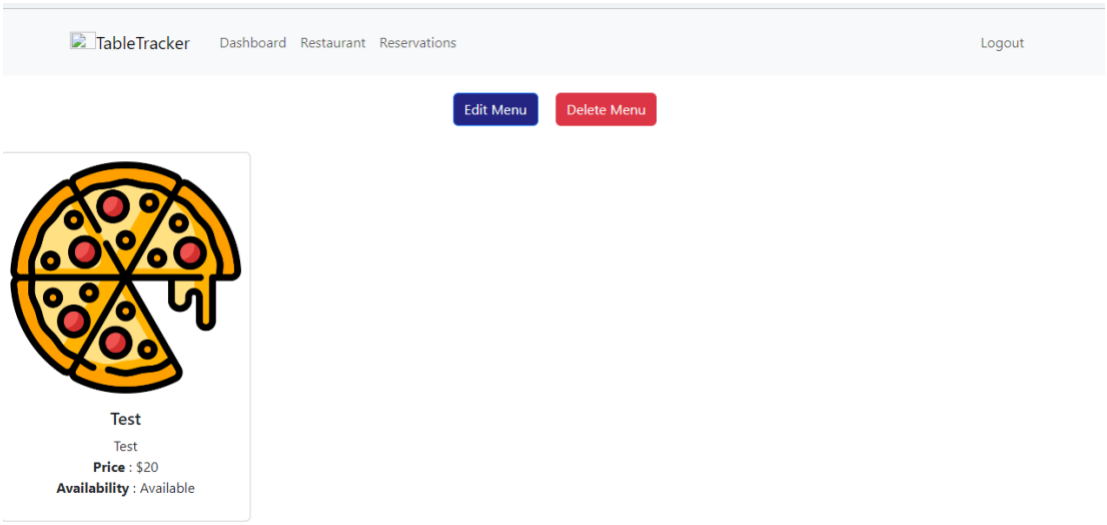


Figure 86 UI for menu management in partner application.

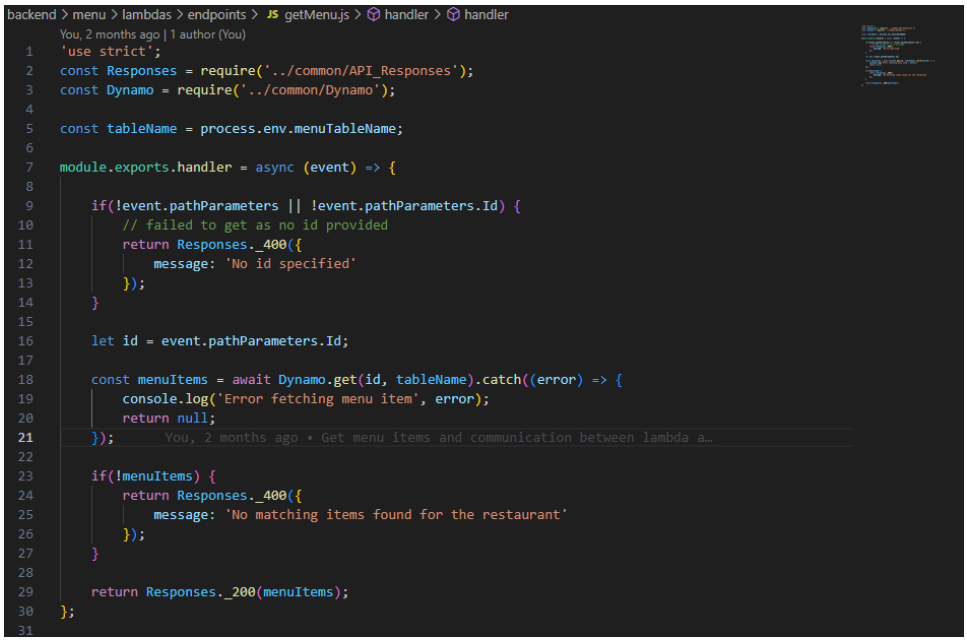


Figure 87 Fetch menu for the restaurant.

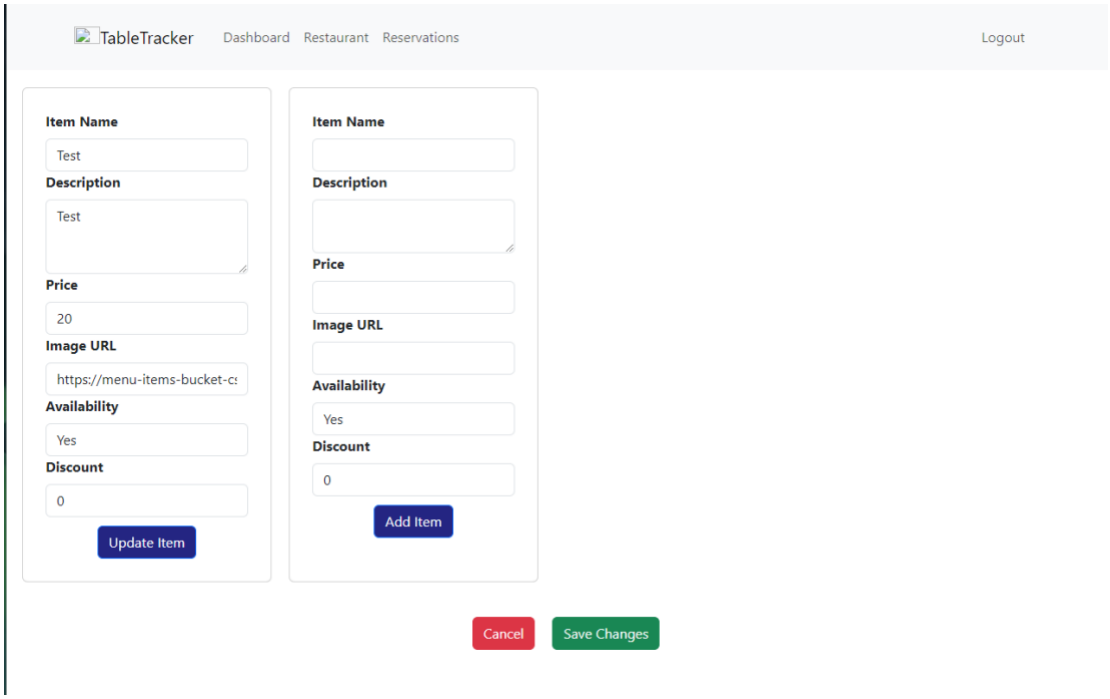


Figure 88 UI for updating or modifying menu items for partner.

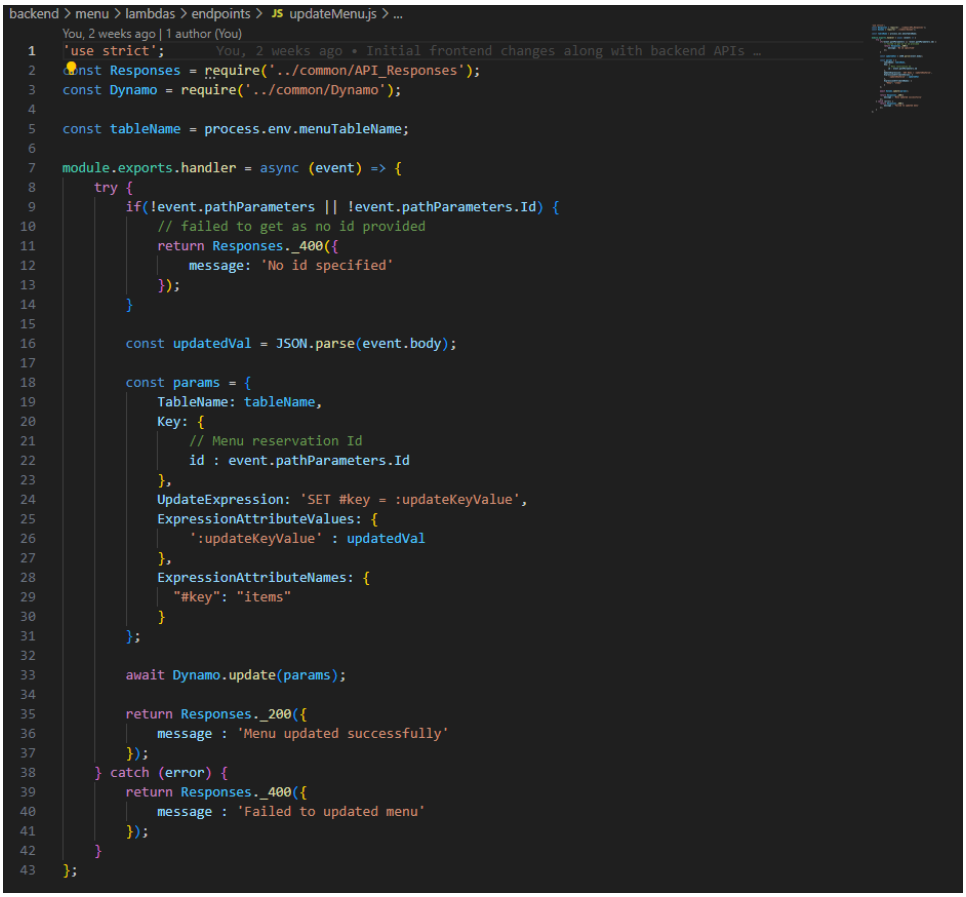


Figure 89 Lambda code to update a menu for a partner restaurant.

Figure 90 Lambda code to delete a menu for a partner restaurant.

2. **Price Addition for Menu Items:**
    a. Interface Update: Integrate a section in the menu interface allowing restaurant owners to add and update prices for menu items.
    b. Validation: Implement validation checks to ensure accuracy in price entries and enforce data consistency.

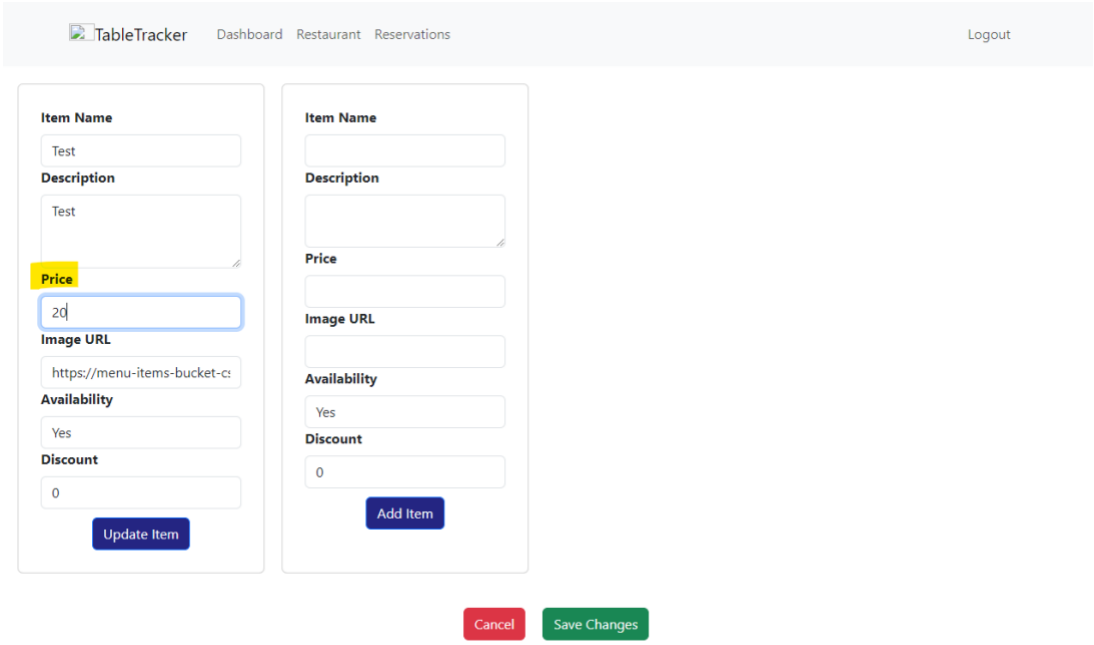


Figure 91UI to demonstrate price addition or modification for a menu item.

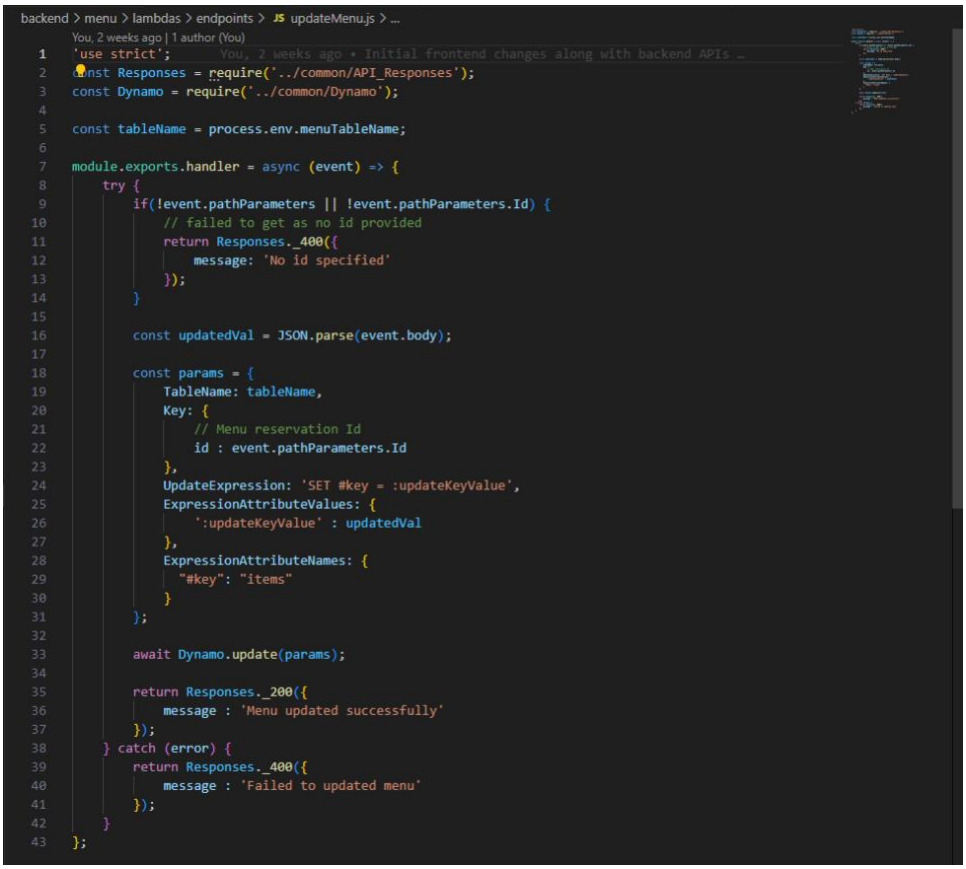


Figure 92 Lambda code to update the menu for a partner restaurant.

Offer Management:

c. Offer Creation: Develop functionality for restaurants to create and manage offers for menu items.
d. Offer Types: Support two primary offer types: individual menu item-based offers and entire menu offers.
e. Offer Visibility: Ensure that customers can view offers by displaying slashed prices for items with active offers and banner displays for menu-wide offers.

```
{
 "id": "R1",
 "discount": 25,
 "items": [
  {
   "id": 1,
   "availability": true,
   "description": "Pizza is the best",
   "discount": "0",
   "img": "https://menu-items-bucket-csci5410.s3.us-east-2.amazonaws.com/pizza.png",
   "name": "Pizza",
   "price": "15"
  },
  {
   "id": 2,
   "availability": false,
   "description": "Pasta is second best",
   "discount": "0",
   "img": "https://menu-items-bucket-csci5410.s3.us-east-2.amazonaws.com/pasta.png",
   "name": "Pasta",
   "price": "10"
  }
 ],
 "restaurantId": "R1"
```


Figure 93 Individual and Entire menu discount data structure for a restaurant

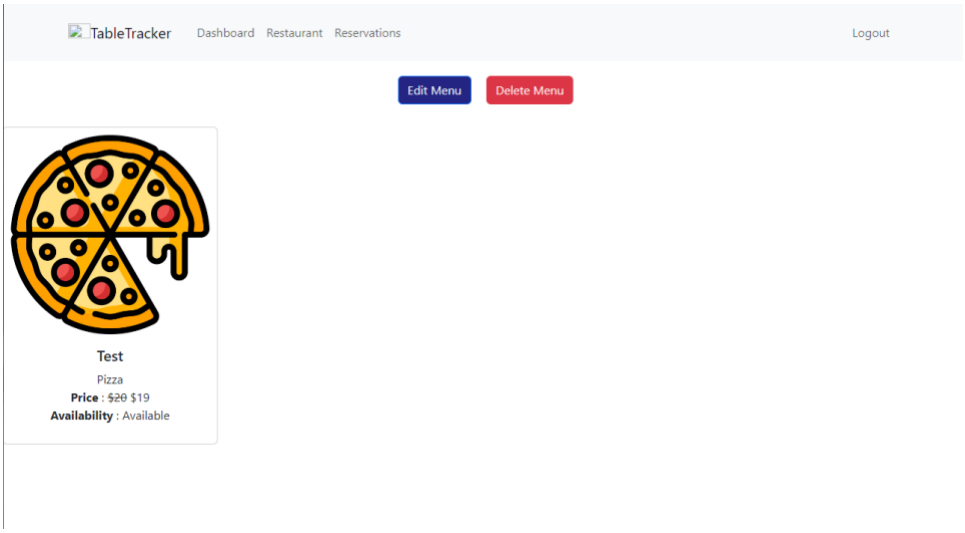


Figure 94 View menu with updated or discounted price based on the discount provided by a partner.

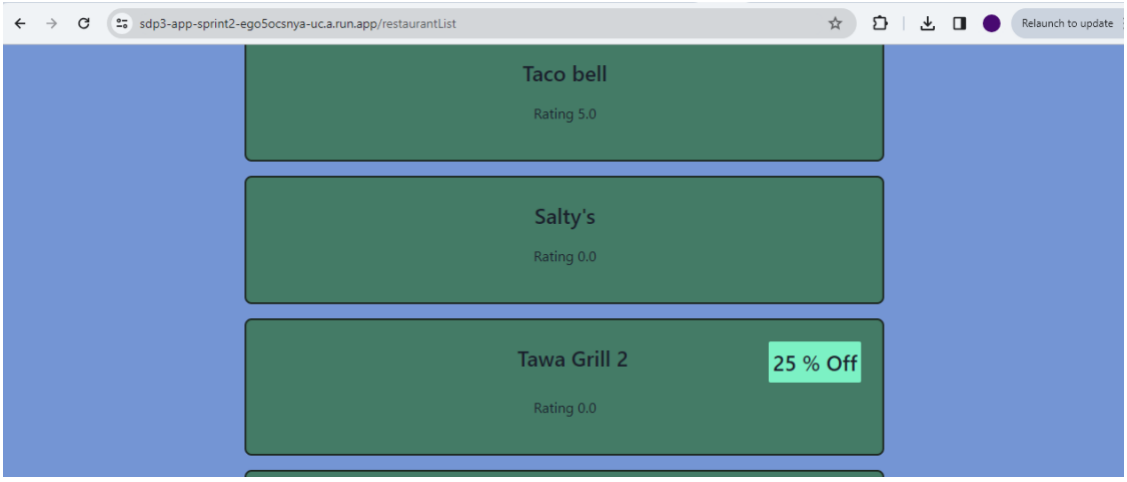


Figure 95 Restaurants with offer banner that has discount on whole menu items.

```
{
 "id": "R1",
 "discount": 25,
 "items": [
  {
   "id": 1,
   "availability": true,
   "description": "Pizza is the best",
   "discount": "0",
   "img": "https://menu-items-bucket-csci5410.s3.us-east-2.amazonaws.com/pizza.png",
   "name": "Pizza",
   "price": "15"
  },
  {
   "id": 2,
   "availability": false,
   "description": "Pasta is second best",
   "discount": "0",
   "img": "https://menu-items-bucket-csci5410.s3.us-east-2.amazonaws.com/pasta.png",
   "name": "Pasta",
   "price": "10"
  }
 ],
 "restaurantId": "R1"
```


Figure 96 Restaurants data structure for discount on whole menu.

3. **Availability Management:**
    - CRUD Operations: CRUD functionality for managing the availability of menu items.
    - Update Mechanism: Implement an update mechanism allowing restaurants to modify, create, read, and delete the availability status of menu items.

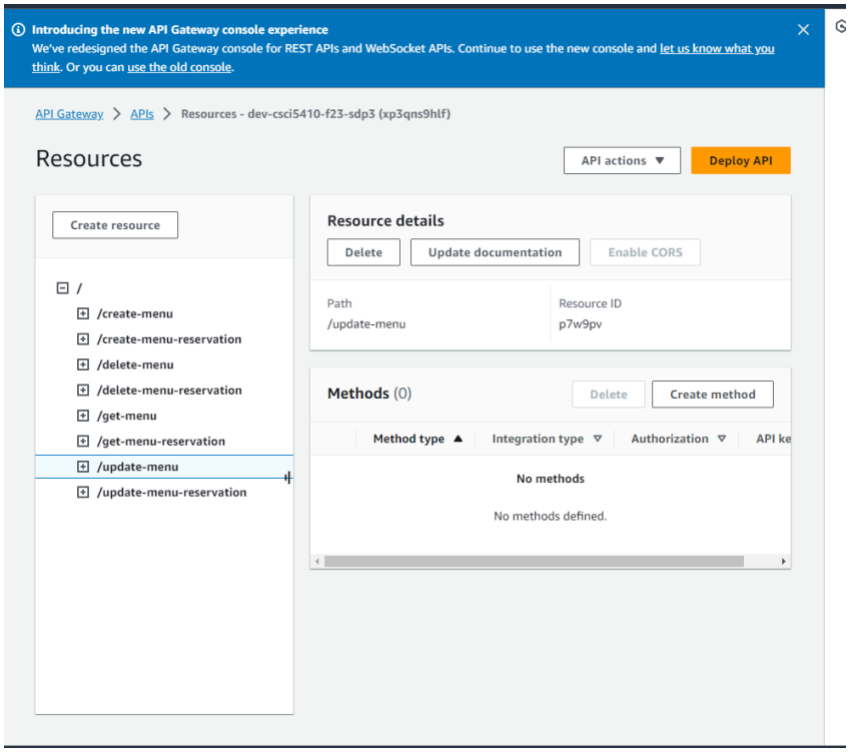


Figure 97 API Gateway displaying all API endpoints for the menu management.

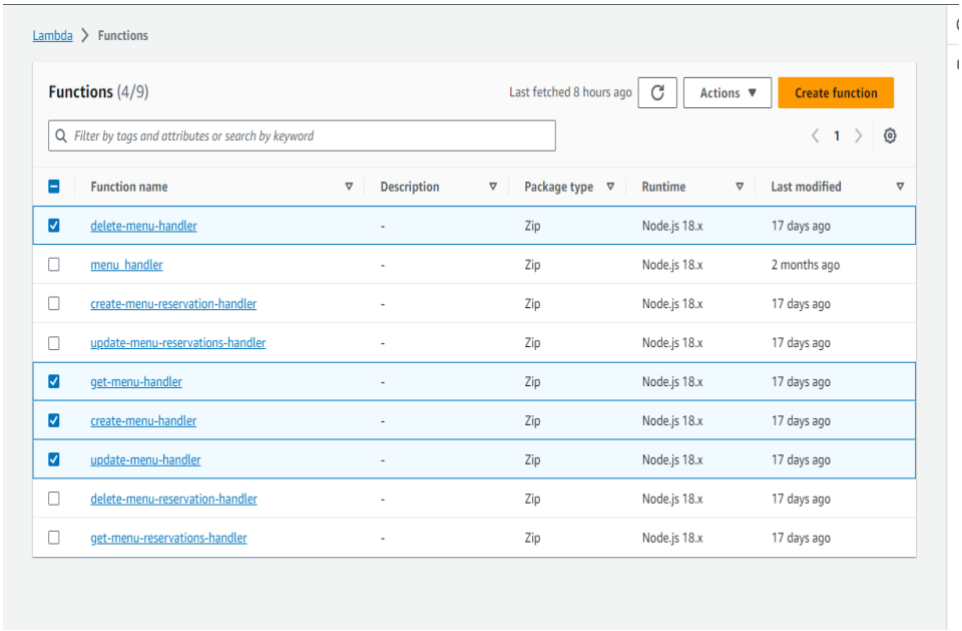


Figure 98 Lambda functions for menu management. [17]

10.3 Test Cases / Use Cases

Table: 15 Edit, delete, view menu

Test Case ID	Description	Steps	Expected Result
TC_EDVM_01	Editing Menu Items	1. Log in as a restaurant owner. 2. Navigate to the menu editing section. 3. Select an existing menu item and modify its name or description. 4. Save changes.	The edited menu item details are updated and reflect the changes on the menu display.
TC_EDVM_02	Deleting Menu Items	1. Log in as a restaurant owner. 2. Access the menu management interface. 3. Choose a menu item to delete. 4. Confirm the deletion action.	The selected menu item is removed from the menu and is no longer displayed.
TC_EDVM_03	Viewing Menu	1. Access the customer application. 2. Navigate to the	The customer can view the restaurant's menu with all available items,

		menu section of the chosen restaurant.	prices, and descriptions.
TC_EDVM_04	Adding Price to Menu Items	1. Log in as a restaurant owner. 2. Create a new menu item and add a price. 3. Save the newly added item.	The new menu item with the specified price is successfully added and visible on the menu.
TC_EDVM_05	Providing Offers on Menu Items	1. Log in as a restaurant owner. 2. Access the offer management section. 3. Select a menu item to offer a discount. 4. Apply a discount offer to the chosen item.	Customers viewing the menu item notice a slashed price reflecting the applied offer.
TC_EDVM_06	Entire Menu Offer Display	1. Log in as a restaurant owner. 2. Set a discount offer for the entire menu.	A banner displaying the offer is visible on the customer application when accessing the restaurant's menu.
TC_EDVM_07	Offers Validation	1. Apply an offer on a single menu item. 2. Apply an offer for the entire menu.	Offers are correctly applied and displayed according to their respective types (individual or entire menu).
TC_EDVM_08	Availability Management	1. Log in as a restaurant owner. 2. Navigate to the availability management section. 3. Modify the availability status of a menu item.	The availability status is updated accordingly and reflected on the menu.

10.4 Services Used

- **AWS Lambda:** Used for serverless computing to execute code for various functionalities.
- **Firestore:** Utilized as a real-time database for storing reservation-related data.
- **DynamoDB**: Employed as a NoSQL database to manage menu-related information.
- **API Gateway:** Serves as an entry point for APIs and manages their interactions.

- **S3 (Simple Storage Service):** Used to store and serve menu item images associated with reservations.

11. Holistic View – (Dheeraj Bhat)

11.1 Planning

The functionalities include a holistic view restaurant data for the partners. This includes a daily, weekly, and monthly trend of reservations for a restaurant. The system architecture employs AWS Lambda for serverless operations, Firestore for data storage, API Gateway for handling API requests, and S3 for lambda code.

Workflow:

- ***Monthly trend of reservations***
  - Partner is a shown a bar graph showing the average reservations for every month.
  - Lambda function triggered via *API Gateway and fetches* the restaurant reservation *trend from the* Firestore.
- ***Weekly trend of reservations***
  - Partner is a shown a bar graph showing the average reservations for every day of a week.
  - Lambda function triggered via *API Gateway and fetches* the restaurant reservation *trend from the* Firestore.
- ***Daily trend of reservations***
  - Partner is a shown a bar graph showing the number reservations by date.
  - Lambda function triggered via *API Gateway and fetches* the restaurant reservation *trend from the* Firestore.
- **Integration Testing:**
  - Test scenarios to validate end-to-end functionality, ensuring smooth interactions between Lambda, Firestore, API Gateway, and S3.
- **Security Measures:**
  - Implement authentication and authorization mechanisms (e.g., AWS Cognito, IAM) to *control user access to reservations and menus.*
  - Ensure encryption and secure access policies for data stored in Firestore, and S3.

The *planning outlines the essential components, data structures, workflow, and potential considerations required to implement the mentioned functionalities using the specified technologies.*

11.2 Implementation

Get Holistic view

- AWS Lambda Setup:

- Create Lambda functions to fetch holistic view data for the partners restaurant reservation. The lambda accepts 3 types of views: Daily, Weekly and Monthly. The holistic view shows 2 metrics: no of reservations and the number of tables. [D2].
- Integrate Lambda functions with Firestore to fetch the holistic view data. [D1].

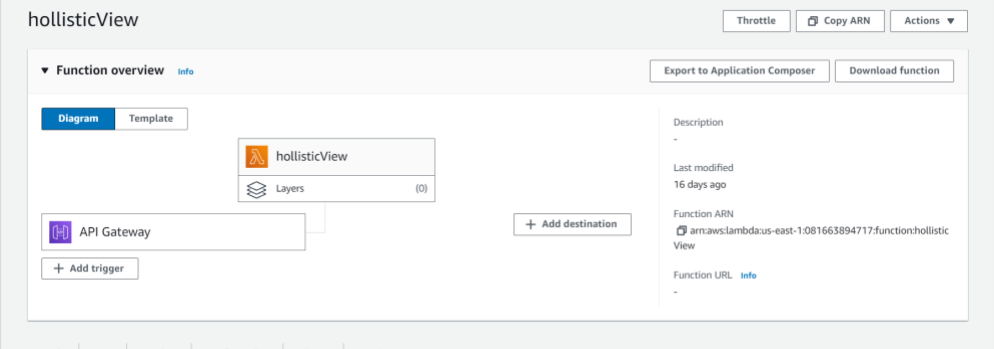


Figure 99: Lambda for holistic view. [17]

```
var admin = require("firebase-admin");
var serviceAccount = require("./sdp3-firestore.json");
const Responses = require("./ApiResponses");

admin.initializeApp({
  credential: admin.credential.cert(serviceAccount),
});

const db = admin.firestore();

exports.handler = async (event) => {
  try {
    const { restaurantId, view } = event.queryStringParameters;

    const reservations = await db
      .collection("RestaurantReservations")
      .where("restaurant_id", "==", restaurantId)
      .get();

    const aggregatedData = aggregateReservations(reservations.docs, view);

    return Responses._200({
      data: [...aggregatedData],
      message: "Retrieved hollistic data successfully",
    });
  } catch (error) {
    console.log(error);
    return Responses._400({
      message: "Error retrieving hollistic data",
      error: error.message,
    });
  }
};
```

Figure 100: Code of Lambda for holistic view.

- API Gateway Setup:
  - Configure API Gateway endpoints to trigger the respective Lambda functions for holistic view. [10].

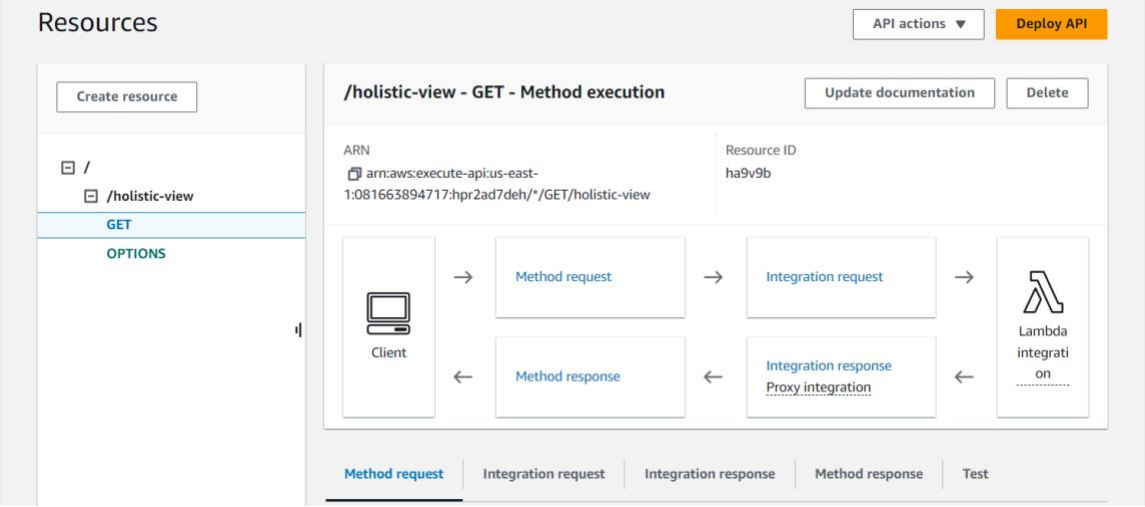


Figure 101: API gateway for holistic view.

- Frontend Integration
    - Implemented frontend components allowing users to approve restaurant reservation details.
    - Integrate API requests from the frontend to trigger the Lambda function via API Gateway.

Figure 102: Monthly and weekly trend holistic view UI

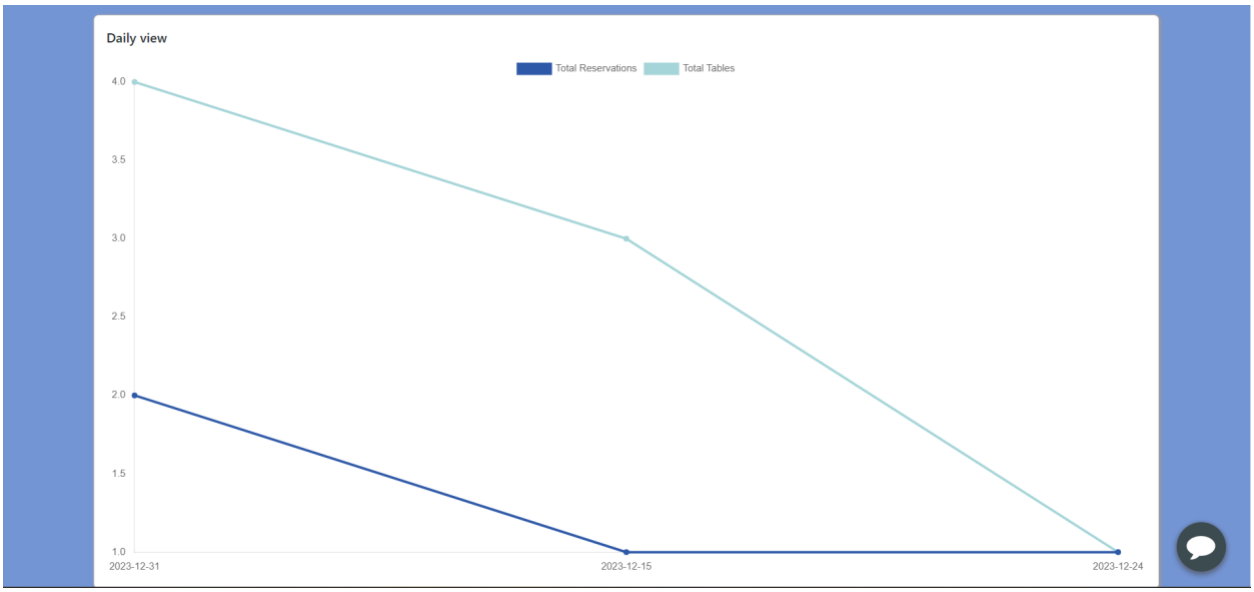


Figure 103: Daily trend holistic view UI

11.3 Test Cases / Use Cases

Table: 16 Holistic view Test Cases

Test Case ID	Description	Steps to Execute	Expected Result	Pass/Fail
TC_HOLISTIC_001	Successful monthly trend view	Navigate to Dashboard page.	Monthly trend holistic view graph shown.	Pass
TC_HOLISTIC_002	Successful weekly trend view	Navigate to Dashboard page.	Weekly trend holistic view graph shown.	Pass
TC_HOLISTIC_003	Successful daily trend view	Navigate to Dashboard page.	Daily trend holistic view graph shown.	Pass

11.4 Services Used

- **AWS Lambda:** Used for serverless computing to execute code for various functionalities. [9].
- **Firestore:** Utilized as a real-time database for storing restaurant reservation-related data. [8]
- **API Gateway:** Serves as an entry point for APIs and manages their interactions. [10].
- **S3 (Simple Storage Service):** Used to store zip of the lambda code files.

12. Chatbot – (Gauravsinh Bharatsinh Solanki)

12.1 Planning

Objective: To construct a partner-side chatbot application that allows restaurant partners to manage reservations and operational details seamlessly.

Intent Development:

ViewReservationIntent: Allows partners to view all current reservations, filterable by date, time, and status.

UpdateRestaurantInfoIntent: Enables partners to update restaurant details, such as location, contact information, and operational hours.

ManageMenuIntent: Permits partners to add, remove, or edit menu items, including pricing and item descriptions.

ReservationManagementIntent: Facilitates partners in managing bookings, including confirming, rescheduling, or canceling reservations.

12.2 Implementation

Sprint Execution:

Implemented serverless AWS Lambda functions to perform background tasks like reservation synchronization, data aggregation, and operational updates.

Developed a robust API layer using Amazon API Gateway to facilitate secure and scalable interactions between the chatbot and backend services.

Technical Solutions:

Employed Amazon DynamoDB for high-performance data storage, ensuring fast access to reservation and restaurant information.

Integrated AWS Lex for constructing the conversational logic required for partners to interact with the system through natural language processing.

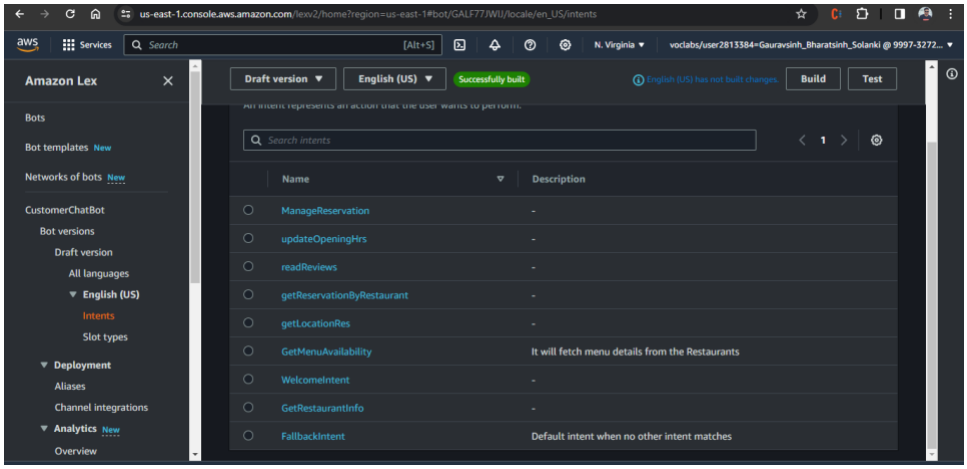


Figure 104 Intents for the Partner Application Chatbot.

Feature Development:

Enabled dynamic management of restaurant information, including updating opening hours and menu details through conversational commands.

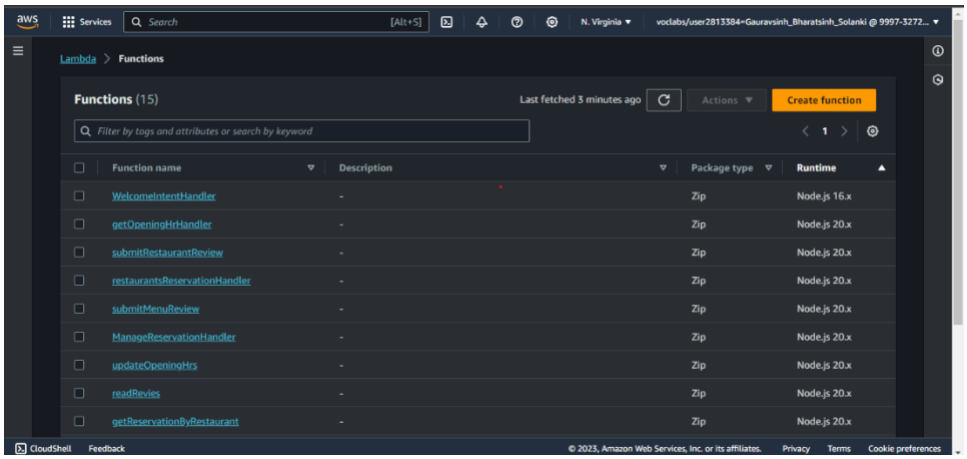


Figure 105 Lambda Functions Integration.

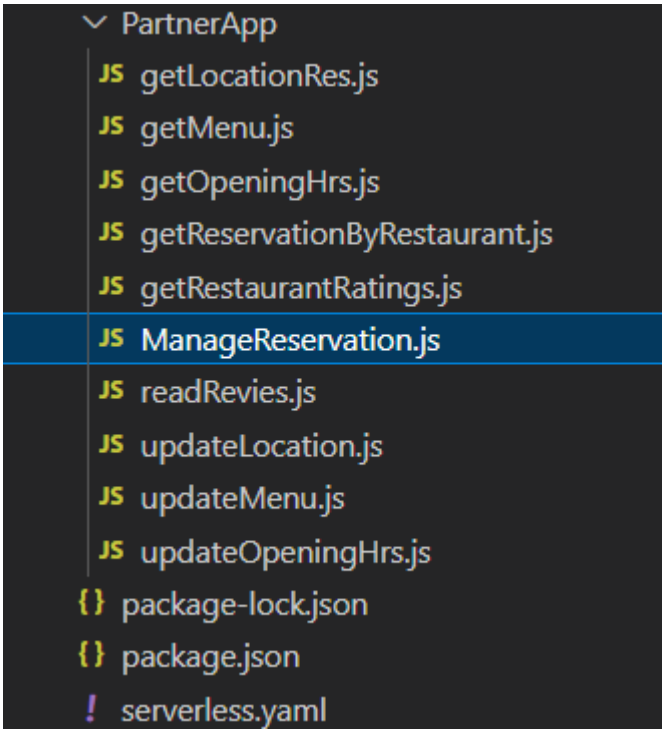


Figure 106 Total 10 Lambdas for the Partner Application

Facilitated real-time reservation management, allowing partners to view, cancel, and modify bookings via the chatbot interface.

Challenges Overcome:

Addressed the complexity of handling concurrent reservation modifications, ensuring data integrity and consistency.

Implemented advanced error handling strategies to gracefully manage potential system failures or user input anomalies.

12.3 Test Cases / Use Cases

Test Case: Update Opening Hours

Objective: Ensure that the chatbot accurately processes requests to update restaurant opening hours.

Procedure: Simulate a chatbot interaction where the partner requests to change opening hours and verify the update in the database.

Expected Outcome: The system reflects the new opening hours accurately and acknowledges the update to the partner.

Test Case: Reservation Conflict Resolution

Objective: Validate the system's ability to handle double-booking conflicts.

Procedure: Create two overlapping reservations and assess the chatbot's response for conflict resolution.

Expected Outcome: The chatbot should notify the partner of the conflict and provide options to resolve it.

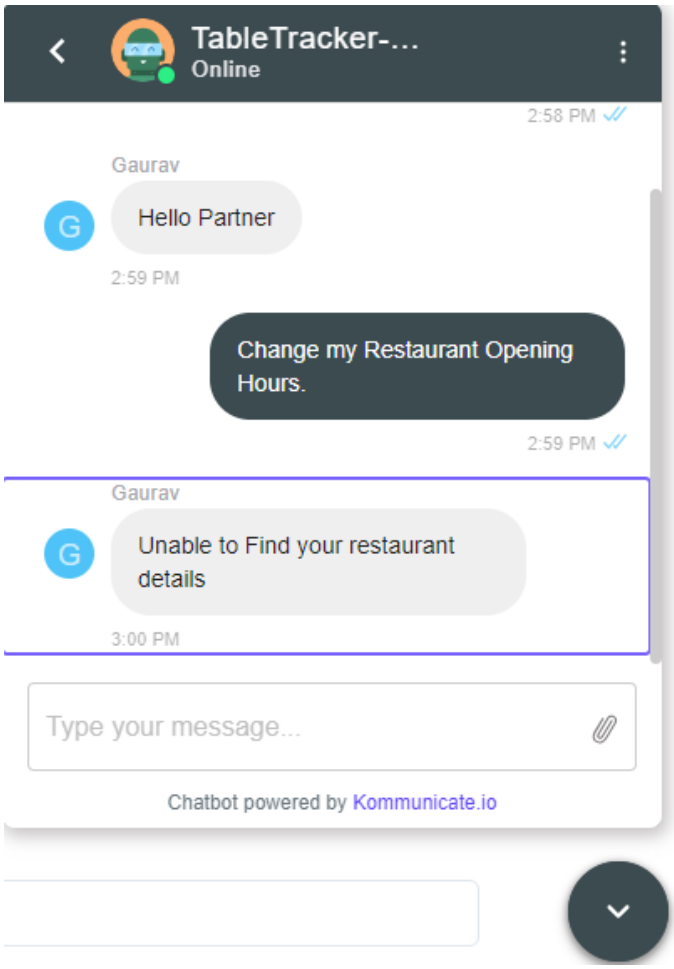


Figure 107 Partner trying to update hours before registering restaurant.

Test Case: Menu Item Modification

Objective: Test the chatbot's capability to modify menu items and reflect changes in real-time.

Procedure: Initiate a menu item change through the chatbot and confirm the update across the system.

Expected Outcome: The menu should be updated promptly, and the partner should receive confirmation of the change.

Use Cases:

Use Case: Real-Time Booking Overview

Scenario: A partner needs to get an updated list of all reservations for the day.

Functionality: The chatbot presents an up-to-date booking schedule upon request, including any recent changes or cancellations.

Use Case: Handling Customer Reviews

Scenario: A partner wants to review and respond to customer feedback collected through the chatbot.

Functionality: The chatbot aggregates customer reviews and ratings, allowing the partner to view and interact with customer feedback.

Use Case: Analytics and Reporting

Scenario: A partner requires insights into dining trends and reservation patterns.

Functionality: The chatbot compiles analytics on customer behavior, popular dining times, and frequently booked tables, presenting the data in an accessible format.

12.4 Services Used

Amazon Web Services:

AWS Lex: Crafted interactive chat experiences for partners using natural language understanding.

AWS Lambda: Deployed multiple backend functions for transaction processing and data manipulation.

Amazon DynamoDB: Utilized for its low-latency and scalable database capabilities, holding reservation data, partner profiles, and operational details.

Amazon API Gateway: Provided a secure, scalable entry point for the chatbot to communicate with backend services.

Development Frameworks:

Used Node.js for creating scalable server-side applications and Lambda functions.

Implemented unit testing frameworks such as Jest for automated testing of individual components.

Applied the Serverless Framework to streamline the deployment of cloud resources and Lambda functions.

Integration and Monitoring:

Integrated with Amazon CloudWatch for real-time monitoring of system performance and operational health.

13. Notifications – (Riya Patel)

13.1 Planning

To implement the proposed features for restaurant notifications, the following steps can be taken:

New Reservations Notification:

- Use Firebase to track changes in reservation data.
- Trigger a Lambda function upon reservation changes.
- Send a notification to the respective restaurant using SNS.

1 Hour Notification for Reservations with Menu:

- Utilize Firebase to identify reservations with associated menus.
- Schedule an Event Bridge event or use a timer to trigger a Lambda function.
- Send a notification to the restaurant using SNS.

10 Minutes Notification for Reservations without Menu:

- Use Firebase to identify reservations without associated menus.
- Schedule an Event Bridge event or use a timer to trigger a Lambda function.
- Send a notification to the restaurant using SNS.

Overbooked Tables and Top Menu Items Notification:

- Every 4 hours, identify overbooked tables and top three menu items.
- Trigger a Lambda function using Event Bridge or a scheduled event.
- Send a comprehensive notification to the restaurant using SNS.

These steps can be implemented using a combination of Firebase for data tracking, AWS Lambda functions for event handling, Event Bridge, or timers for scheduling, and SNS for sending notifications.

13.2 Implementation

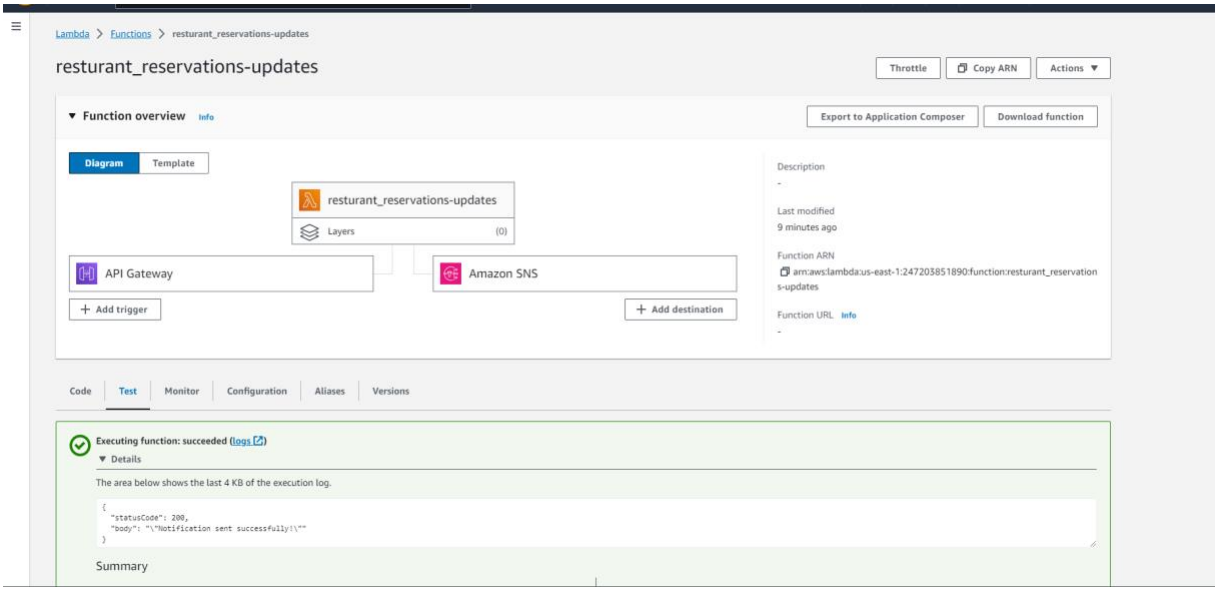


Figure 108 New reservation Updates for restaurants' Lambda.[17]

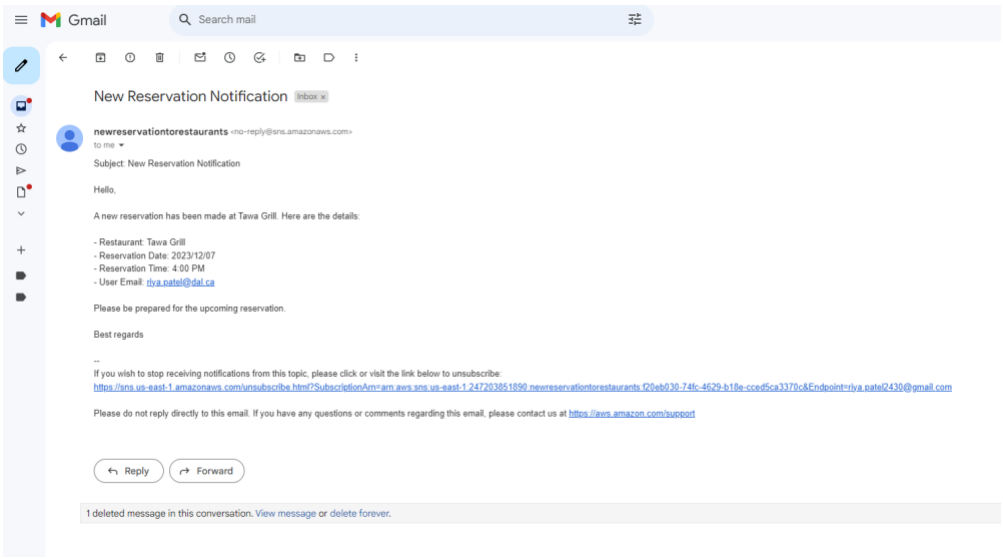


Figure 109 New reservation Updates for restaurants' email.

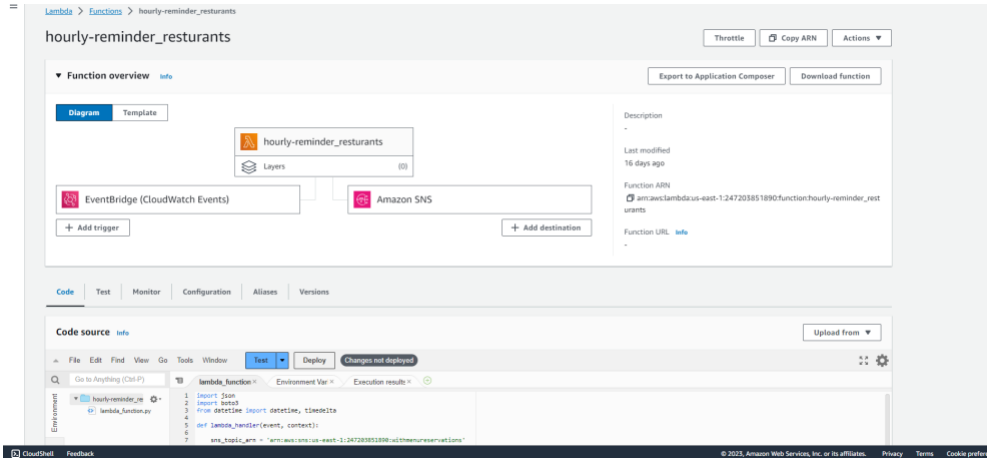


Figure 110 Reservation Reminder before 1 hour to Restaurant [17]

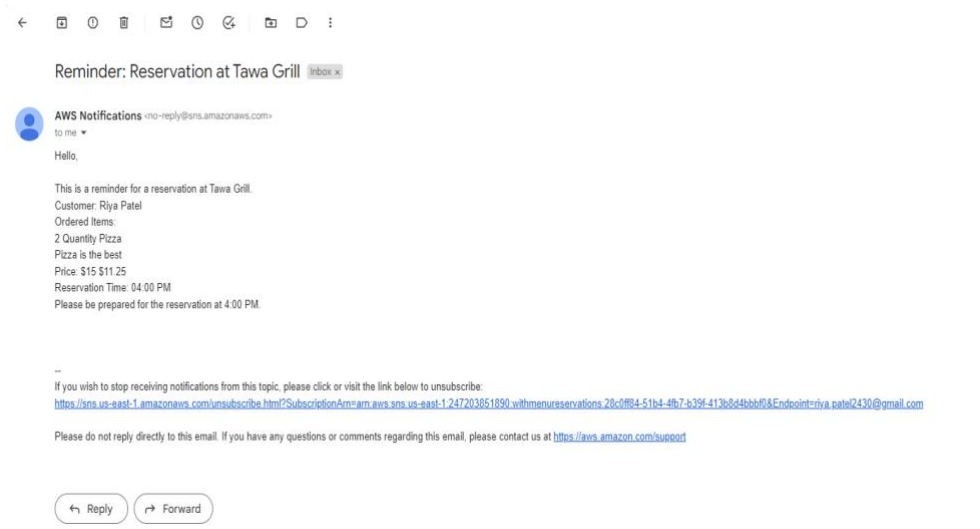


Figure 111 Reservation Reminder before 1 hour to Restaurant Email

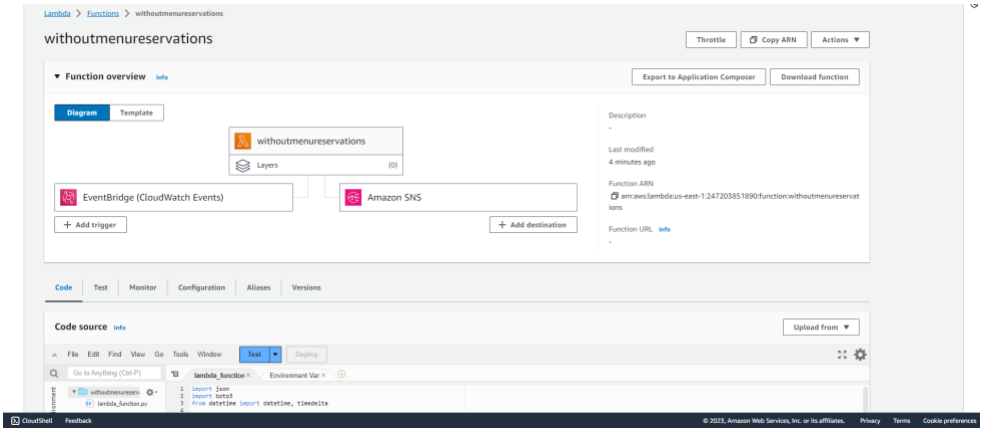


Figure 112 without menu reservation reminder lambda. [17]

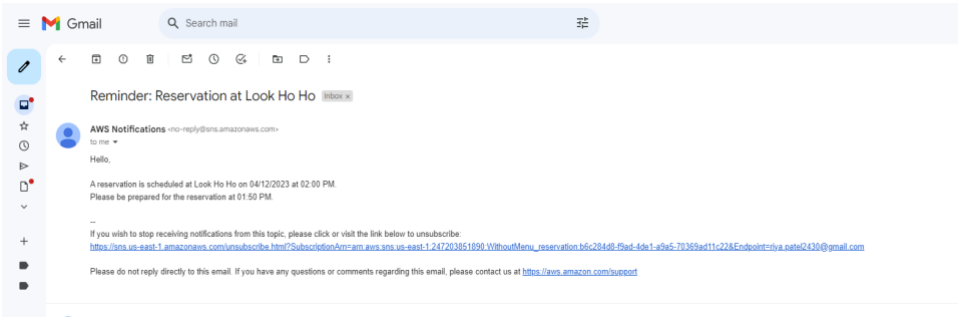


Figure 113 without menu reservation reminder email.

13.3 Test Cases / Use Cases

Table: 17 Test Cases for Notifications in Partners' App

Test Case ID	Description	Input	Expected Output
TC_1	New Reservations Notification	Simulate new reservation bookings, modifications, deletions	- Lambda function is triggered. Notifications are sent to the respective restaurants for new reservation events.
TC_2	1 Hour Notification for Reservations with Menu	Simulate reservations with associated menus	- Trigger the scheduled event. Lambda function is triggered. Notifications are sent to the restaurants 1 hour in advance for reservations with menus.
TC_3	10 Minutes Notification for Reservations without Menu	Simulate reservations without associated menus	- Trigger the scheduled event. Lambda function is triggered. Notifications are sent to the restaurants 10 minutes before reservations without menus.
TC_4	Overbooked Tables and Top Menu Items Notification	Simulate overbooked tables and record bookings	- Trigger the scheduled event. Lambda function is triggered. Identify overbooked tables and top menu items. Notifications are sent to the restaurants with relevant information.

13.4 Services Used

- **AWS Lambda**: Used for serverless execution of backend logic in response to various triggers, such as scheduled events or changes in data.
- **Firebase:** A mobile and web application development platform by Google. In this context, it is used to store and retrieve data related to offers, restaurant openings, reservations, menu items, and closures.
- **Amazon SNS (Simple Notification Service):** A fully managed messaging service for sending notifications to a distributed set of recipients. Used here to send email notifications to customers based on different events.
- **Event Bridge:** A serverless event bus service by AWS. It allows you to connect different applications using events. In this context, it is used to trigger Lambda functions at specific intervals or in response to specific events.
- **CloudWatch Events**: A service that enables you to schedule Lambda functions to run at specified intervals. Used here to trigger the hourly event for sending notifications.
- **API Gateway**: To get uid and other data from Frontend.

c. Admin App

1. Visualisations

a. Planning

- **Firebase Data Integration:**
  Ensure that the relevant data related to customer orders is stored in Firebase. This might include information about orders, customers, and order quantities.
- **BigQuery Integration:**
  Export or synchronize the Firebase data to BigQuery. This can be achieved using tools like Firebase Data Transfer or other integration methods.
- **BigQuery Query for Visualization:**
  Write a SQL query in BigQuery to extract the necessary data for visualizing the top 10 customers who have ordered the most. This query may involve aggregating order quantities per customer and ordering them in descending order.
- **Looker Dataset Configuration:**
  Create a dataset in Looker Studio that connects to the BigQuery table containing the customer order data. Define dimensions and measures based on the fields you need for the visualization.
- **LookML Model Development:**
  Use LookML (Looker Modeling Language) to define the relationships, dimensions, and measures in Looker. This step involves writing LookML code to model the data from BigQuery within Looker.
- **Looker Dashboard and Reports:**
  Create a Looker dashboard that includes a report for visualizing the top 10 customers who have ordered the most. This report should use the LookML model and dataset you have configured.
- Embed Looker Report in HTML App:
  Generate an embed link or URL for the Looker report. Looker provides embedding options that allow you to embed reports in external applications.
- **Integrate Looker Embed Code in Admin App:**
  Embed the Looker report in your admin app's HTML using the Looker embedded view code. Looker provides SDKs and documentation for embedding reports in various web applications.
- **Dynamic Data Updating:**
  Ensure that the embedded Looker view dynamically fetches the latest data from BigQuery. Looker Studio typically handles this automatically, but you may need to configure data refresh settings if necessary.

b. Services Used

- **Looker Studio:**
  Looker Studio is a platform for business intelligence and data exploration. It provides tools for creating and embedding interactive data visualizations. In this case, Looker Studio is used to build reports and embed visualizations in the admin app.
- **BigQuery:**

BigQuery is a fully managed, serverless data warehouse by Google Cloud. It is used for querying and analyzing large datasets. In this context, BigQuery is used to store and query data related to customer orders.

- **Firebase:**
  Firebase is a mobile and web application development platform by Google. It includes a NoSQL database and other services. Firebase is used to store and retrieve data related to customer orders and other app functionalities.

• The top 10 restaurants that have the most orders – (Arihant Dugar)

I. Implementation

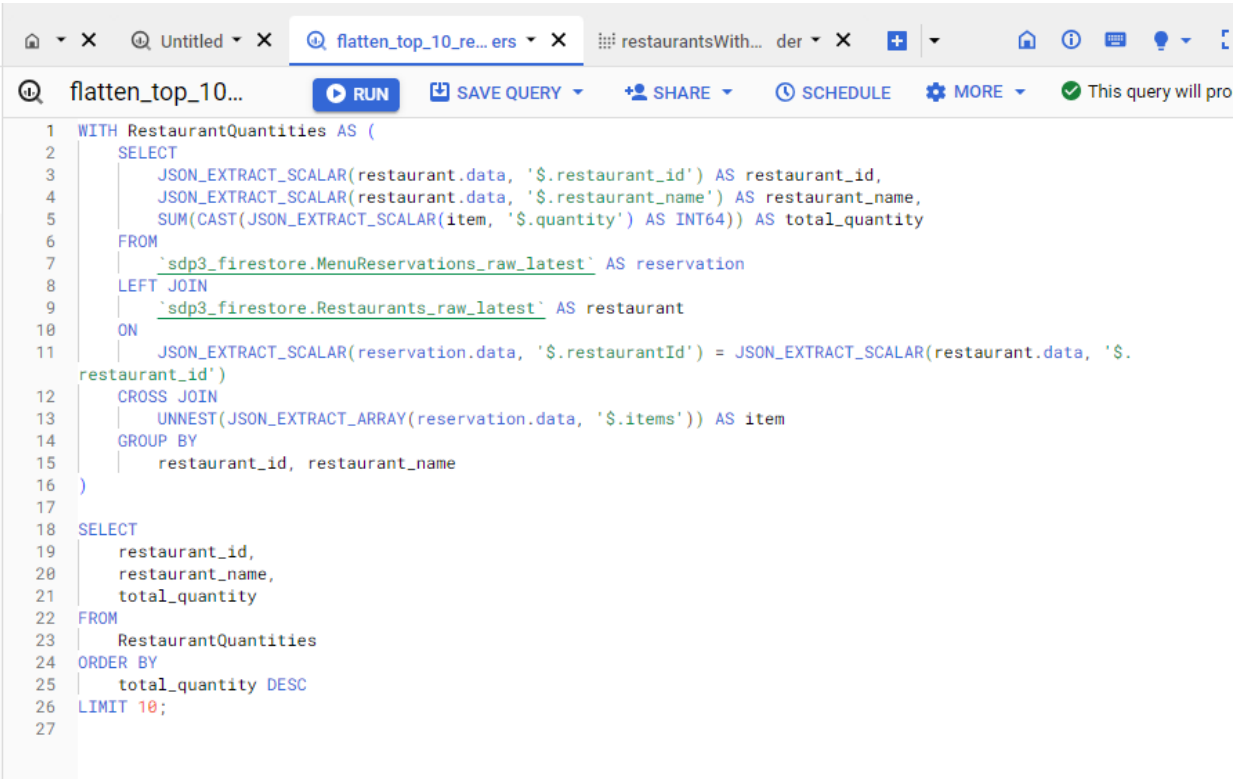


Figure 114 Query The top restaurants that have the most orders. [18]

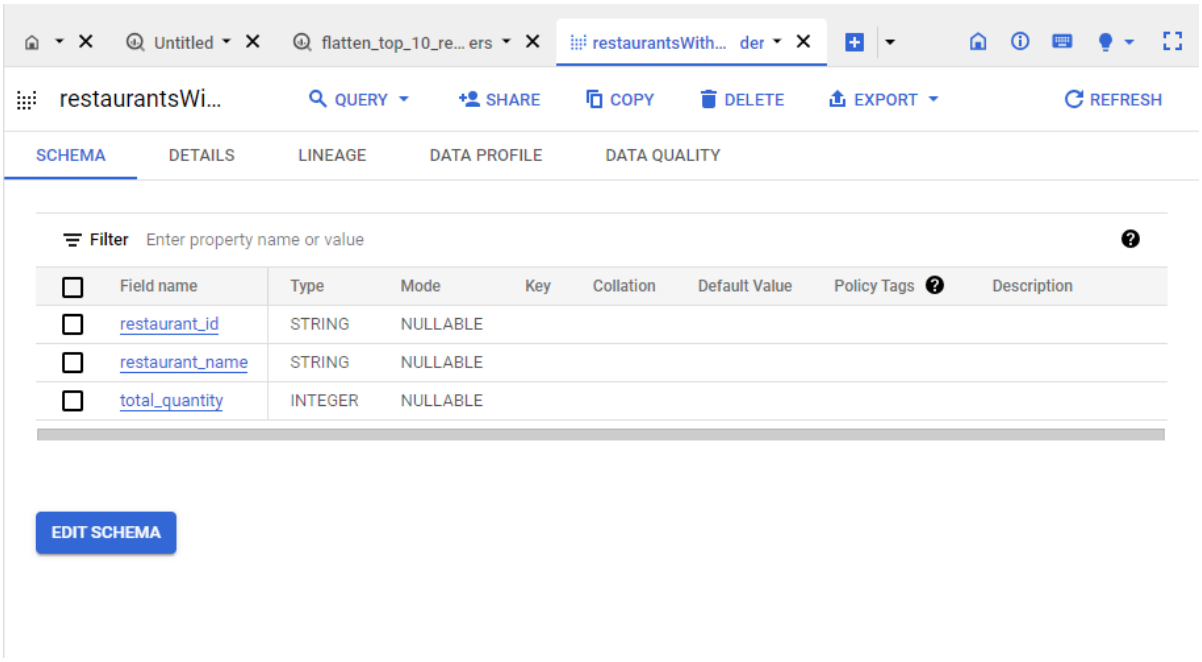


Figure 115 Schema for Restaurants with most orders [18]

	Restaurant Id	Restaurant	Total orders
1.	c06cbc14-abed-4694-8adc-47313db03b1c	Westcliff Restaurant	8
2.	2fdc3ca3-e087-49c3-acf4-3eabf71f5181	Wendy's	6
3.	582a3ac3-c193-47d6-aeda-3cf75ed9d09d	Taco bell	4
4.	17859696-19aa-4ce7-b711-9e8bb8def56e	Salty's	11
5.	af57cd34-6951-4002-8138-2a88f46e5fe4	McDonalds	9
6.	acf6c783-e64a-4905-9585-c37fa9f6348e	Look Ho Ho Restaurant	7
7.	b880c905-ae0a-409e-b9dc-25a84026bee9	Five Guy's	2
8.	9284f24e-0396-44fd-9371-299d0e9e53c1	Dairy Queen	5
9.	aaff3e6f-0eda-4cc9-b5d6-b37ef162fe6a	Cosy Snack Bar	1
10.	R1	Besharam	8

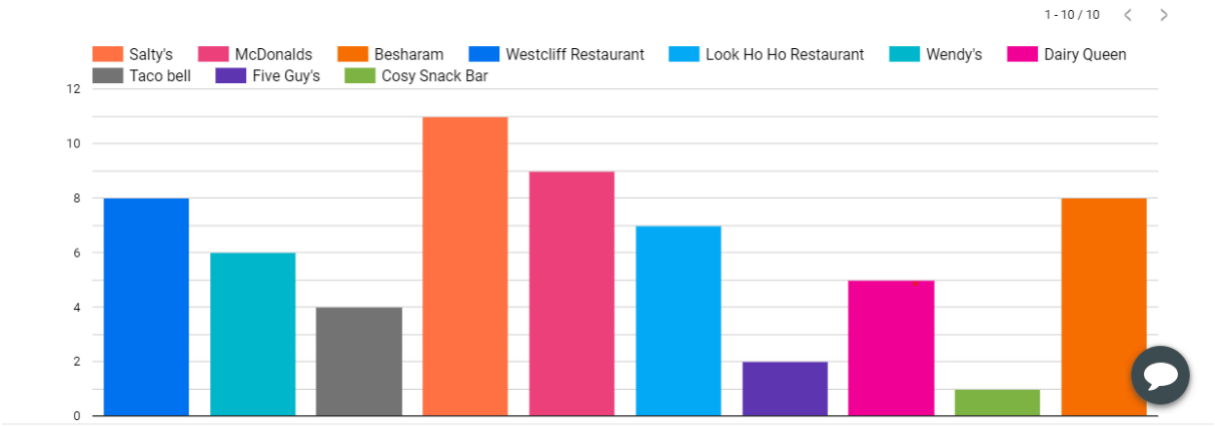


Figure 116 Dashboard look.

II. Test Cases / Use Cases

Objective:

To verify the accuracy of the visualization displaying the top 10 restaurants by the number of orders.

Scenario:

- **Preconditions**: Ensure access to the Looker Studio environment and necessary permissions to view the visualization.
- **Input Data:** Ensure the availability and correctness of the dataset related to restaurant orders.

Test Steps:

Validation of Visualization:

- Open the Looker Studio dashboard containing the "Top 10 Restaurants with Most Orders" visualization.
- Verify that the visualization displays a list of restaurants along with the count of orders for each restaurant.
- Check if the restaurants are listed in descending order based on the number of orders, with the restaurant having the most orders at the top.
- Confirm that the visualization is limited to 10 restaurants as expected.

Interaction and Filtering:

- Apply different filters to the visualization and ensure that the top 10 restaurants dynamically adjust according to the applied filters without any discrepancies.
- Verify the responsiveness of the visualization when interacting with different devices or screen sizes.

Expected Results:

- The visualization should accurately present the top 10 restaurants by the number of orders.
- The list should update dynamically based on applied filters or time frames, maintaining accuracy.

Acceptance Criteria:

- The visualization should consistently display the correct top 10 restaurants, updating dynamically as per any applied filters or time frames.
- The order count for each restaurant should be precise and match the underlying data.
- The visualization should be responsive and adaptable to different screen sizes/devices.

• The top 10 food *items ordered across restaurants – (Preeti Sharma)*

I. Implementation

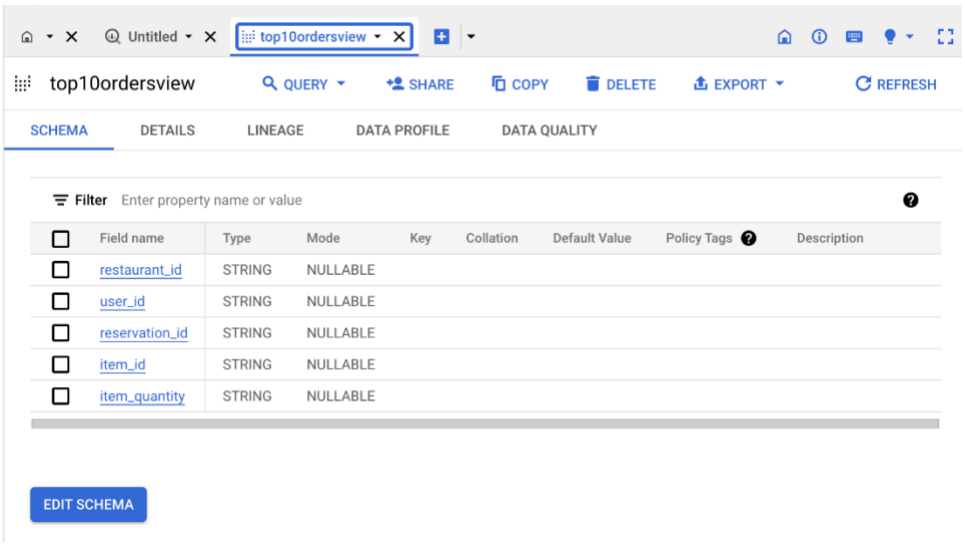


Figure 117 Schema for the top 10 food items ordered across restaurants. [18]

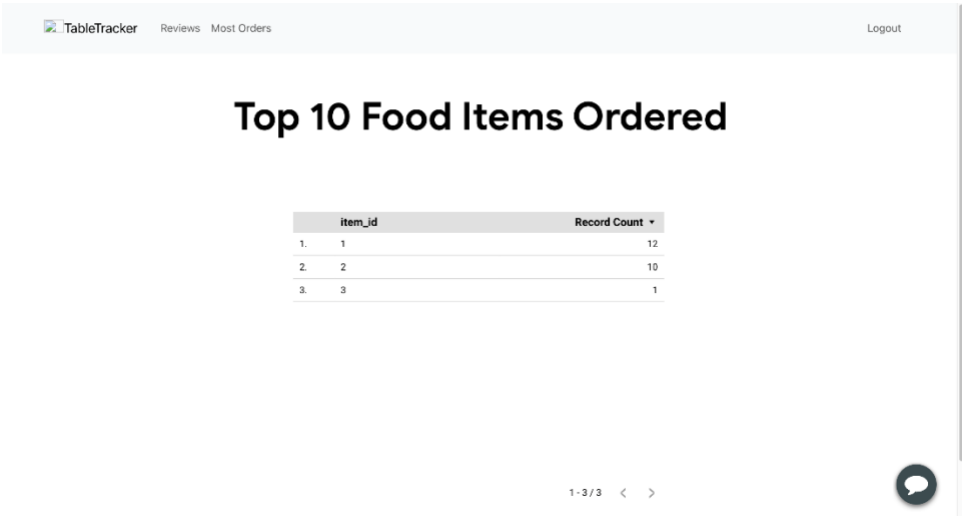


Figure 118 Dashboard for the top 10 food items ordered across restaurants.

II. Test Cases / Use Cases

Table: 18 Test cases for top 10 food items ordered across restaurants

Test Case ID	Description	Input	Expected Output
TC_1	Data Accuracy	- View the Looker report displaying the top 10 food items ordered across restaurants.	- Data in the Looker report accurately represents the top 10 food items ordered across restaurants, in

		Compare the displayed data with the expected results from BigQuery query.	alignment with the BigQuery query results.
TC_2	Dynamic Data Updating	- Make changes to orders in Firebase. View the Looker report after changes.	- Looker report reflects the changes made to orders in Firebase, demonstrating dynamic data updating functionality.
TC_3	Embedding in Admin App	- View the admin app with the embedded Looker report. Ensure responsive design and seamless integration within the app.	- Looker report is correctly embedded in the admin app's HTML. The embedded view is responsive and fits seamlessly within the app's design.
TC_4	Performance	- Load the Looker embedded view with a large dataset. Measure the time taken for the visualization to load.	- The Looker embedded view performs efficiently, loading the visualization within an acceptable timeframe even with a large dataset.
TC_5	Security	- Access the Looker embedded view with unauthorized credentials. Attempt to manipulate or access unauthorized data.	- Access to the Looker embedded view is restricted to authorized users. Unauthorized attempts to manipulate or access data are prevented by Looker's secure access controls

Objective:

To verify the accuracy of the visualization displaying the top 10 food items ordered across restaurants.

Scenario:

- **Preconditions**: Ensure access to the Looker Studio environment and necessary permissions to view the visualization.
- **Input Data:** Ensure the availability and correctness of the dataset related to restaurant orders.

Test Steps:

Validation of Visualization:

- Open the Looker Studio dashboard containing the "Top 10 Food Items Ordered Across Restaurants" visualization.
- Verify that the visualization displays item id and the count of orders for that item.
- Check if the items are listed in descending order based on the number of orders, with the items having the most orders at the top.
- Confirm that the visualization is limited to 10 items as expected.

Interaction and Filtering:

- Apply different filters to the visualization and ensure that the top 10 items dynamically adjust according to the applied filters without any discrepancies.
- Verify the responsiveness of the visualization when interacting with different devices or screen sizes.

Expected Results:

- The visualization should accurately present the top 10 food items ordered across restaurants.
- The list should update dynamically based on applied filters or time frames, maintaining accuracy.

Acceptance Criteria:

- The visualization should consistently display the correct top 10 items, updating dynamically as per any applied filters or time frames.
- The order count for each item should be precise and match the underlying data.
- The visualization should be responsive and adaptable to different screen sizes/devices.

• The top 10 periods when the food is most ordered – (Rashmi Goplani)

I. Implementation

Top 10 periods food ordered | RUN | SAVE VIEW | MORE | This query will process 54.14 KB when run

```
WITH CombinedData AS (
  SELECT
    JSON_EXTRACT_SCALAR(restaurantReservation.data, '$.restaurant_id') AS restaurant_id,
    JSON_EXTRACT_SCALAR(Restaurants.data, '$.restaurant_name') AS restaurant_name,
    ARRAY_LENGTH(JSON_EXTRACT_ARRAY(menuReservations.data, '$.items')) as items,
    TIMESTAMP_SECONDS(CAST(JSON_EXTRACT_SCALAR(restaurantReservation.data, '$.reservation_date._seconds') AS INT64)) AS reservation_date
  FROM
    `csci5410-f23-sdp3.sdp3_firestore.RestaurantReservations_raw_latest` AS restaurantReservation
  INNER JOIN
    `csci5410-f23-sdp3.sdp3_firestore.MenuReservations_raw_latest` AS menuReservations
    ON
    JSON_EXTRACT_SCALAR(restaurantReservation.data, '$.restaurant_id') = JSON_EXTRACT_SCALAR(menuReservations.data, '$.restaurantId')
  INNER JOIN
    `csci5410-f23-sdp3.sdp3_firestore.Restaurants_raw_latest` AS Restaurants
  ON
    JSON_EXTRACT_SCALAR(restaurantReservation.data, '$.restaurant_id') = JSON_EXTRACT_SCALAR(Restaurants.data, '$.restaurant_id')
  WHERE
    ARRAY_LENGTH(JSON_EXTRACT_ARRAY(menuReservations.data, '$.items')) > 0
)
SELECT
  restaurant_name,
  EXTRACT(HOUR FROM reservation_date) AS hour_of_day,
  COUNT(*) AS number_of_orders
FROM
  CombinedData
GROUP BY
  restaurant_name,
  hour_of_day
ORDER BY
  number_of_orders DESC;
```

Figure 119 Query to flatten the data to extract top 10 periods when food was ordered. [18]

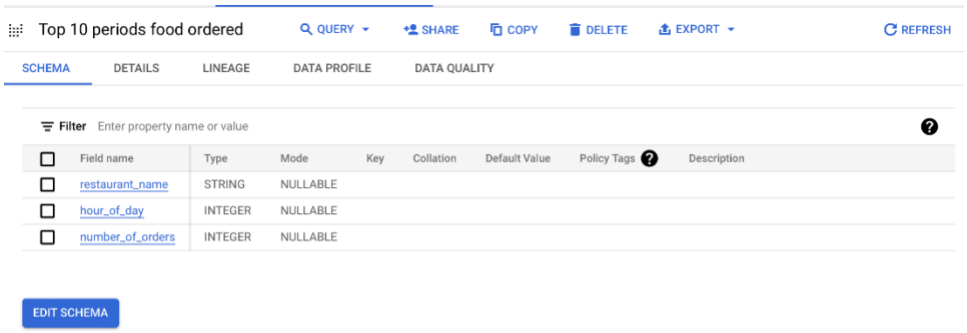


Figure 120 Schema for top 10 periods when food was ordered. [18]

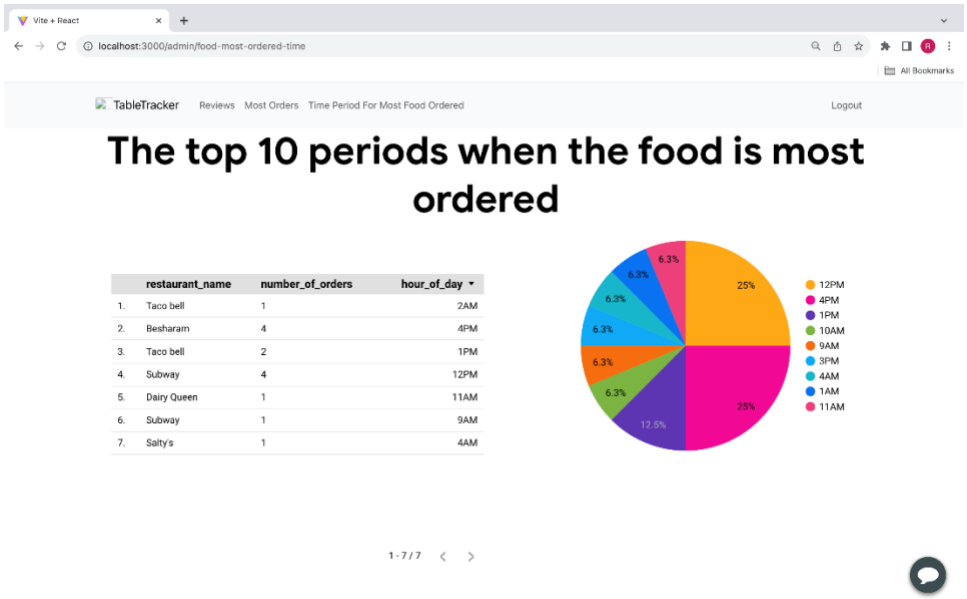


Figure 121 Visualization showing top 10 periods when the food is most ordered.

II. Test Cases / Use Cases

Table: 19 Test Cases for top 10 periods when the food is most ordered

Test Case ID	Description	Input	Expected Output
TC_1	Data Accuracy	- View the Looker report displaying the top 10 periods when the food is most ordered . Compare the displayed data with the expected results from BigQuery query.	- Data in the Looker report accurately represents the top 10 periods when the food is most ordered , in alignment with the BigQuery query results.
TC_2	Dynamic Data Updating	- Make changes to orders in Firebase. View the Looker report after changes.	- Looker report reflects the changes made to orders in Firebase, demonstrating dynamic data updating functionality.
TC_3	Embedding in Admin App	- View the admin app with the embedded Looker report. Ensure responsive design and seamless integration within the app.	- Looker report is correctly embedded in the admin app's HTML. The embedded view is responsive and fits seamlessly within the app's design.
TC_4	Performance	- Load the Looker embedded view with a large dataset. Measure the time taken for the visualization to load.	- The Looker embedded view performs efficiently, loading the visualization within an acceptable timeframe even with a large dataset.
TC_5	Security	- Access the Looker embedded view with unauthorized credentials. Attempt to manipulate or access unauthorized data.	- Access to the Looker embedded view is restricted to authorized users. Unauthorized attempts to manipulate or access data are prevented by Looker's secure access controls

Objective:

To verify the accuracy of the visualization displaying the top 10 periods when the food is ordered most.

Scenario:

- **Preconditions**: Ensure access to the Looker Studio environment and necessary permissions to view the visualization.
- **Input Data:** Ensure the availability and correctness of the dataset related to items ordered.

Test Steps:

Validation of Visualization:

- Open the Looker Studio dashboard containing the " Time Period for Most Food Ordered" visualization.
- Verify that the visualization displays a table and a pie chart indicating the restaurant name, number of orders and which hour of the day when the food was most ordered
- Check if all the restaurants are listed in the table and it shows the correct time.
- Confirm that the visualization is limited to 10 orders as expected.

Expected Results:

- The visualization should accurately present the table and pie chart for the top 10 periods when food was ordered most.

Acceptance Criteria:

- The visualization should consistently display the correct reviews
- The visualization should be responsive and adaptable to different screen sizes/devices

• The top 10 customers who have ordered the most – (Riya Patel)

I. Implementation

Figure 122 Flatten Query for The top 10 customers who have ordered the most. [18]

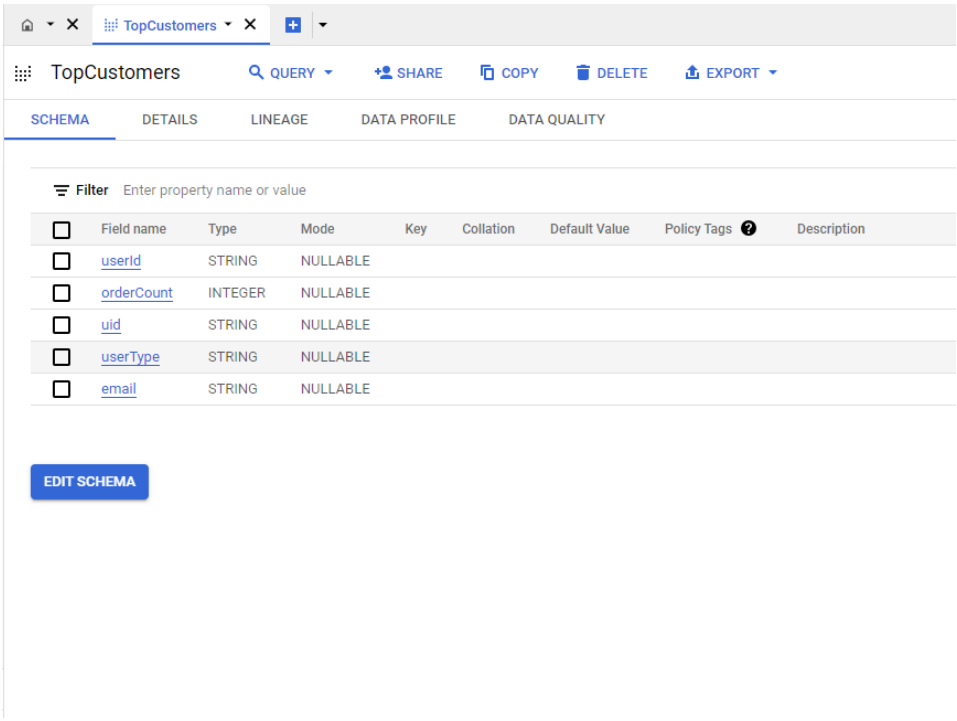


Figure 123 Schema for the top 10 customers who have ordered the most. [18]

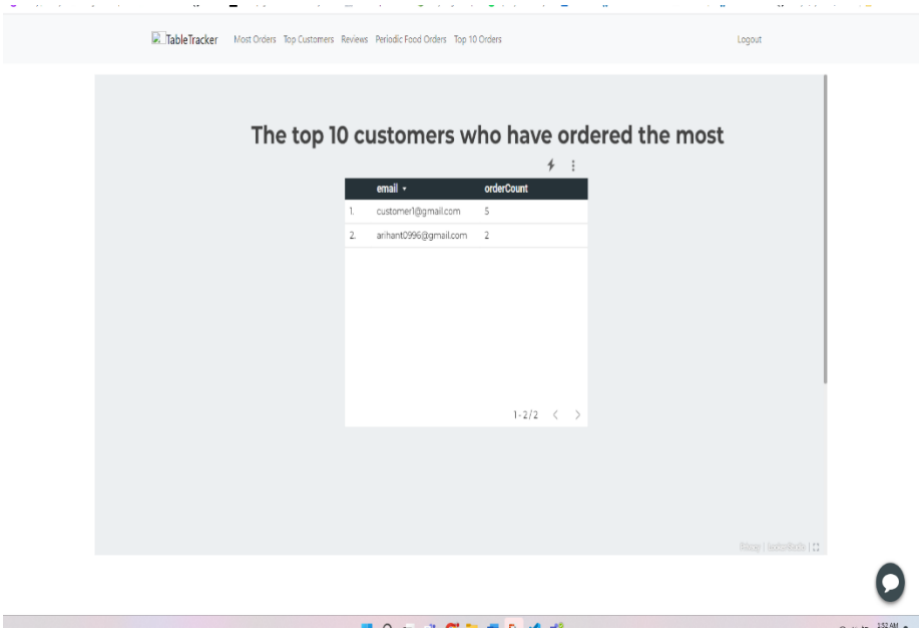


Figure 124 Dashboard for the top 10 customers who have ordered the most.

II. Test Cases / Use Cases

Table: 20 Test Cases top 10 customers who have ordered the most.

Test Case ID	Description	Input	Expected Output
TC_1	Data Accuracy	- View the Looker report displaying the top 10 customers. Compare the displayed data with the expected results from your BigQuery query.	- Data in the Looker report accurately represents the top 10 customers who have ordered the most, in alignment with the BigQuery query results.
TC_2	Dynamic Data Updating	- Make changes to customer orders in Firebase. View the Looker report after changes.	- Looker report reflects the changes made to customer orders in Firebase, demonstrating dynamic data updating functionality.
TC_3	Embedding in Admin App	- View the admin app with the embedded Looker report. Ensure responsive design and seamless integration within the app.	- Looker report is correctly embedded in the admin app's HTML. The embedded view is responsive and fits seamlessly within the app's design.
TC_4	Performance	- Load the Looker embedded view with a large dataset. Measure the time taken for the visualization to load.	- The Looker embedded view performs efficiently, loading the visualization within an acceptable timeframe even with a large dataset.
TC_5	Security	- Access the Looker embedded view with unauthorized credentials. Attempt to manipulate or access unauthorized data.	- Access to the Looker embedded view is restricted to authorized users. Unauthorized attempts to manipulate or access data are prevented by Looker's secure access controls

• Reviews filtered based on restaurant names – (Dheeraj Bhat)

I. Implementation

Objective:

To verify the accuracy of the visualization displaying the reviews filtered on restaurant names.

Scenario:

- **Preconditions**: Ensure access to the Looker Studio environment and necessary permissions to view the visualization.
- **Input Data:** Ensure the availability and correctness of the dataset related to restaurant reviews.

Test Steps:

Validation of Visualization:

- Open the Looker Studio dashboard containing the "Restaurant Reviews" visualization.

- Verify that the visualization displays a dropdown containing the Restaurants and a table with the restaurant reviews for the selected restaurant. If no restaurant is chosen, then all restaurant reviews are shown.
- Check if all the restaurants are listed in the dropdown. Check if all reviews of a given restaurant is shown.
- Confirm that the visualization is limited to 10 restaurants as expected.

Interaction and Filtering:

- Choose different restaurants from the dropdown. The reviews table must get updated with every selection with the right reviews.
- Verify the responsiveness of the visualization when interacting with different devices or screen sizes.

Expected Results:

- The visualization should accurately present the reviews of the chosen restaurant.
- The table should update dynamically based on the selected restaurant.

Acceptance Criteria:

- The visualization should consistently display the correct reviews, updating dynamically as per selected restaurants.
- The visualization should be responsive and adaptable to different screen sizes/devices.

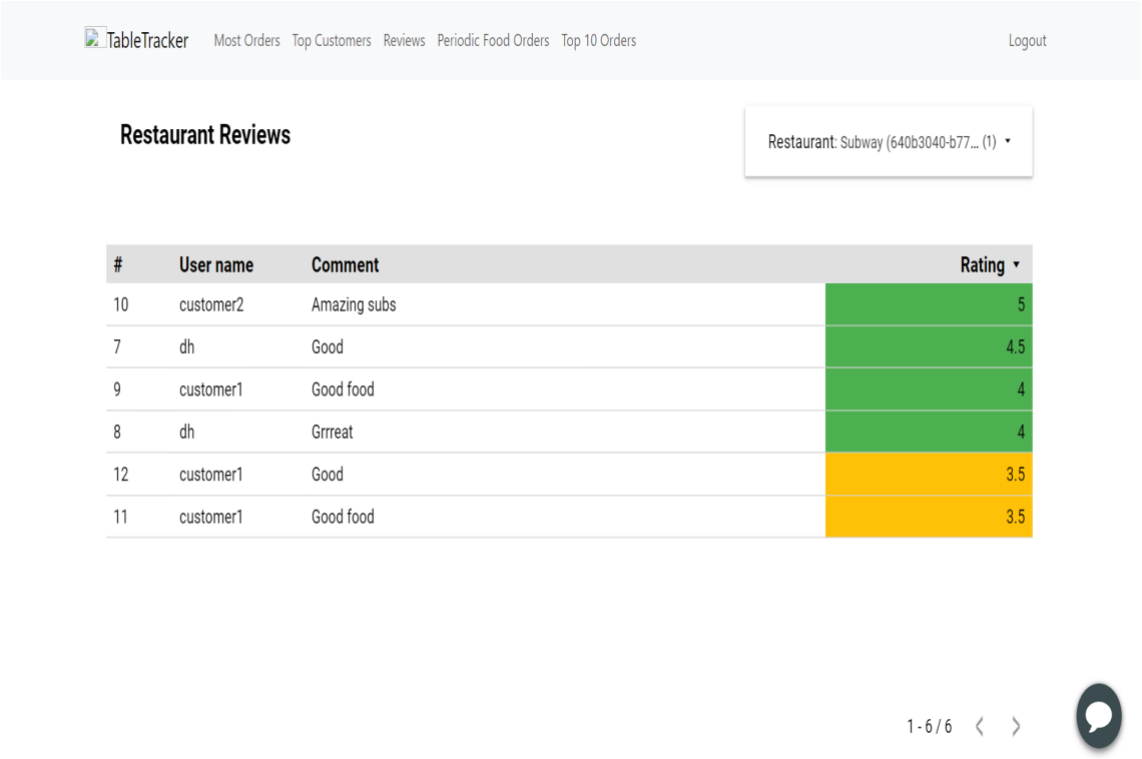


Figure 125: Admin reviews by restaurant looker UI

II. Test Cases / Use Cases

Table: 21 Test Cases for reviews by restaurant looker

Test Case ID	Description	Input	Expected Output
TC_1	Data Accuracy	- View the Looker report displaying the reviews. Compare the displayed data with the expected results from your BigQuery query.	- Data in the Looker report accurately represents the reviews for the selected restaurants, in alignment with the BigQuery query results.
TC_2	Dynamic Data Updating	- Add a review for the selected restaurant. View the Looker report after changes.	- Looker report reflects the changes made to reviews in Firebase, demonstrating dynamic data updating functionality.
TC_3	Embedding in Admin App	- View the admin app with the embedded Looker report. Ensure responsive design and seamless integration within the app.	- Looker report is correctly embedded in the admin app's HTML. The embedded view is responsive and fits seamlessly within the app's design.
TC_4	Performance	- Load the Looker embedded view with a large dataset. Measure the time taken for the visualization to load.	- The Looker embedded view performs efficiently, loading the visualization within an acceptable timeframe even with a large dataset.
TC_5	Security	- Access the Looker embedded view with unauthorized credentials. Attempt to manipulate or access unauthorized data.	- Access to the Looker embedded view is restricted to authorized users. Unauthorized attempts to manipulate or access data are prevented by Looker's secure access controls

3. Application RoadMap

1. Application RoadMap

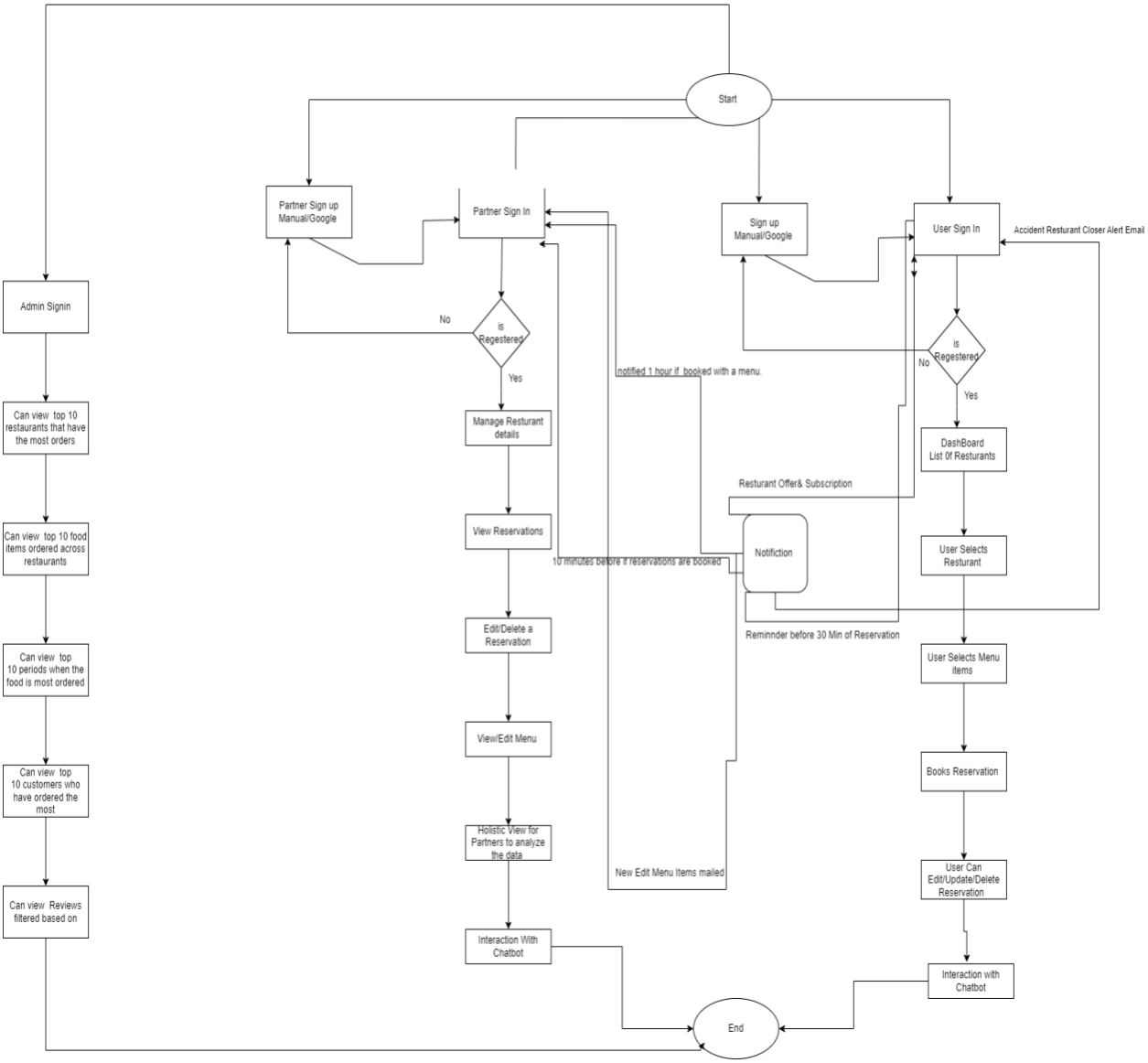


Figure 126 Application RoadMap. [19]

2. User Authentication

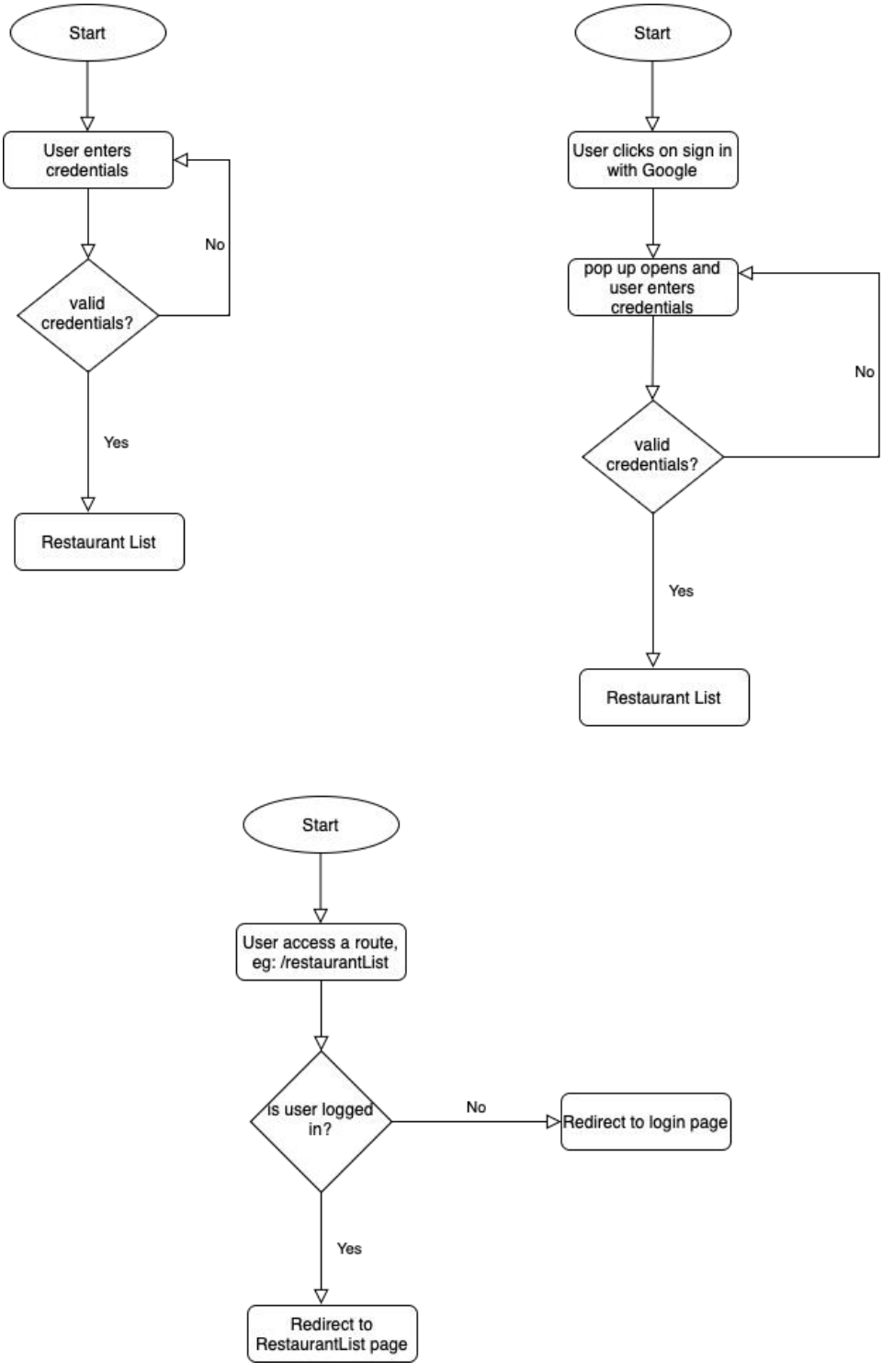


Figure 127 User Authentication Flowchart. [19]

3. List of Restaurants

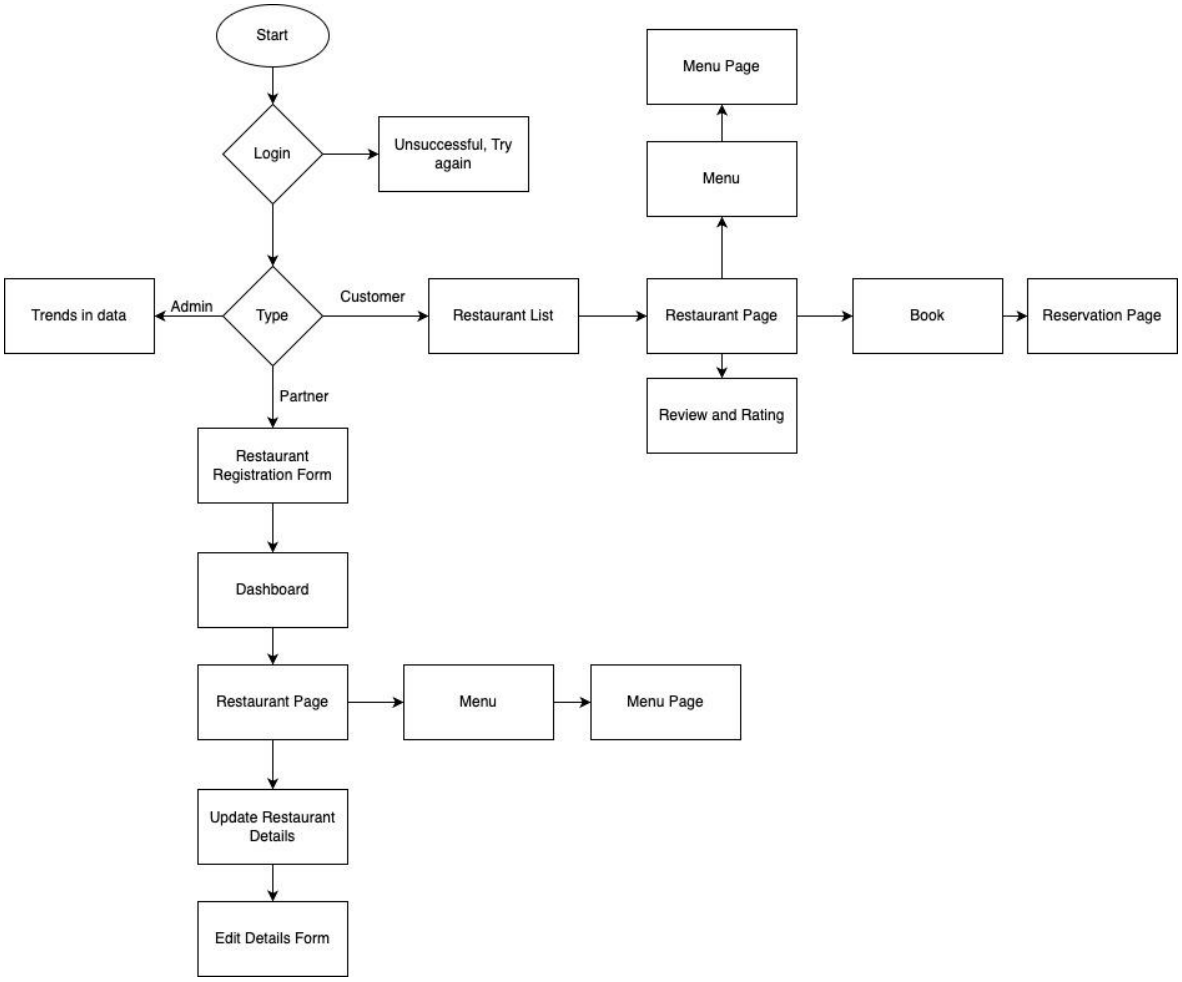


Figure 128 Flow diagram for List of Restaurants. [19]

4. Book, Edit, Delete View Restaurant Reservation

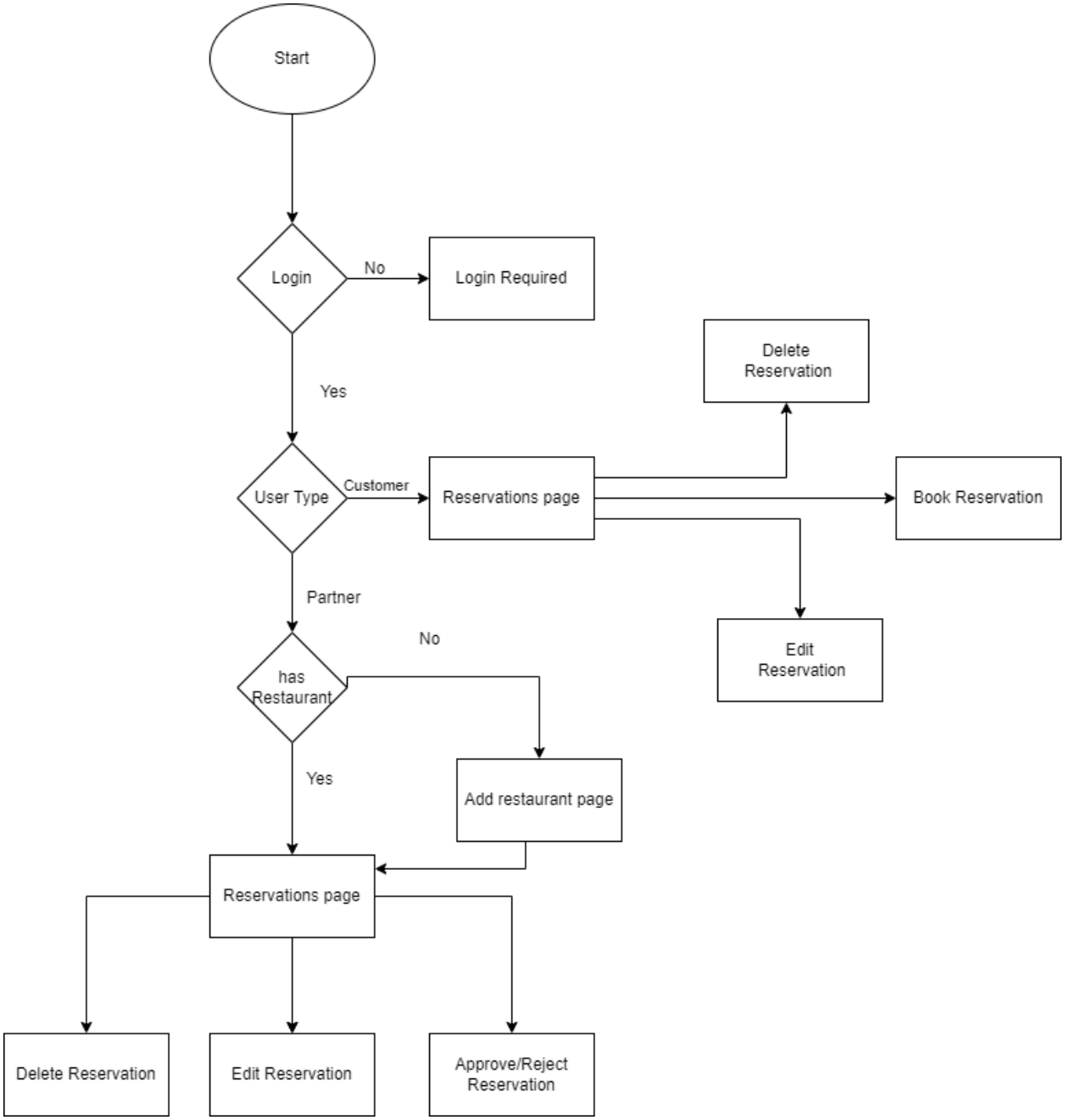


Figure 129 Flow Diagram for Book, Edit, Delete View Restaurant Reservation. [19]

5. Book, Edit, Delete, View Menu for Reservation

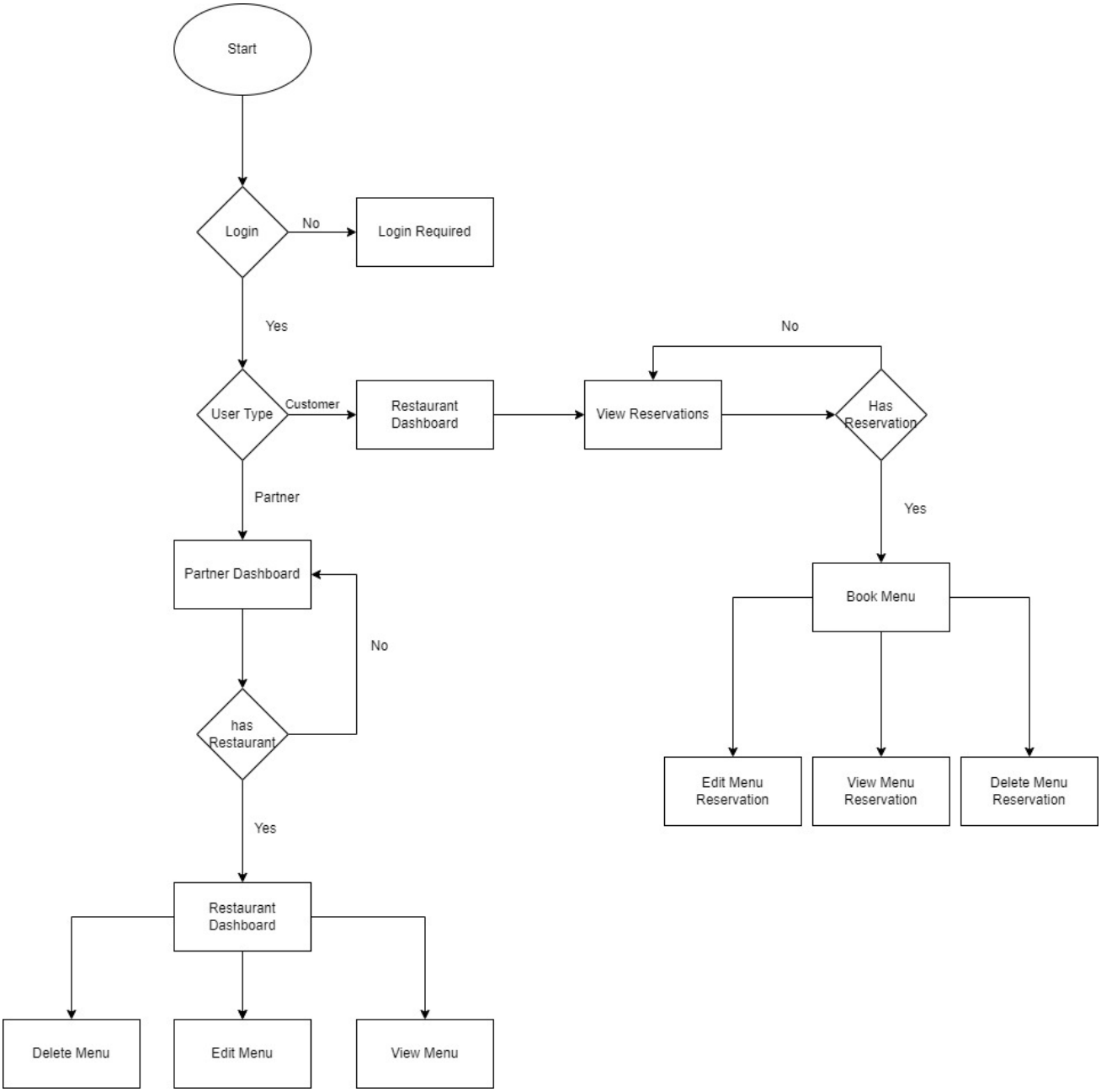


Figure 130 flow diagram for Book, Edit, Delete, View Menu for Reservation. [19]

6. Chatbot

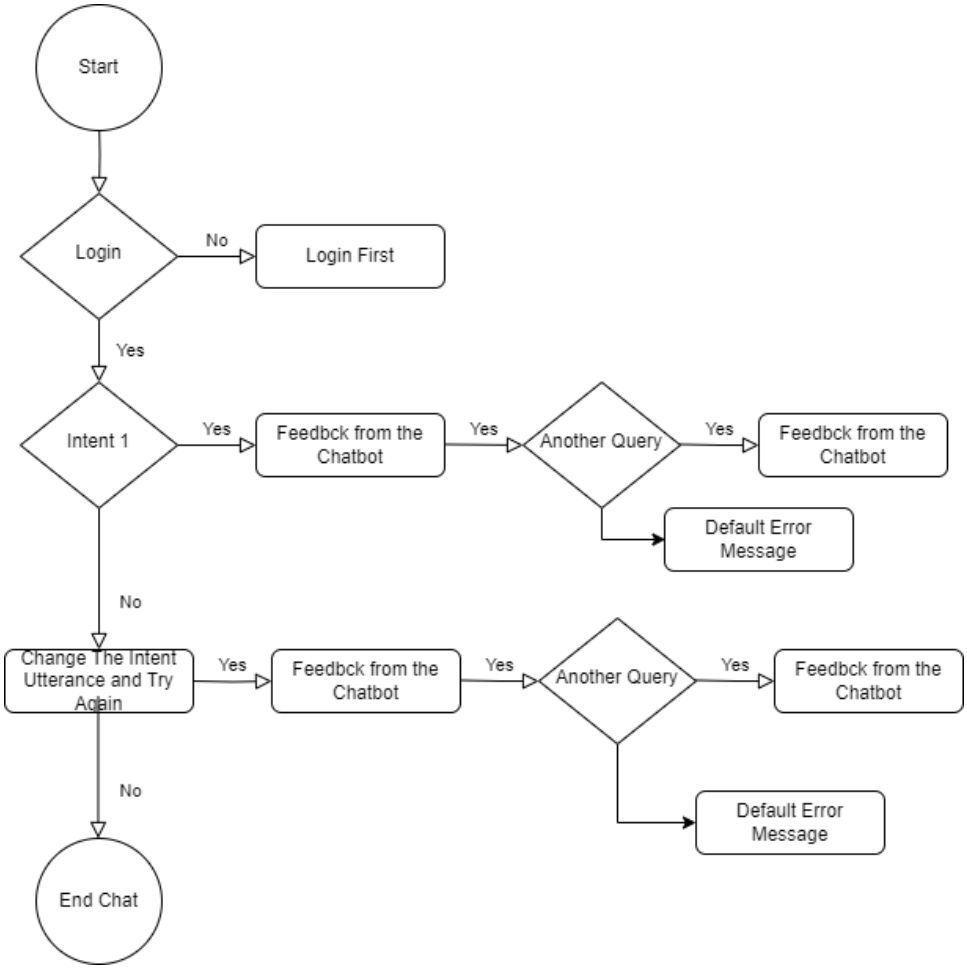


Figure 131 Flow Diagram for Chatbot. [19]

7. Visualization

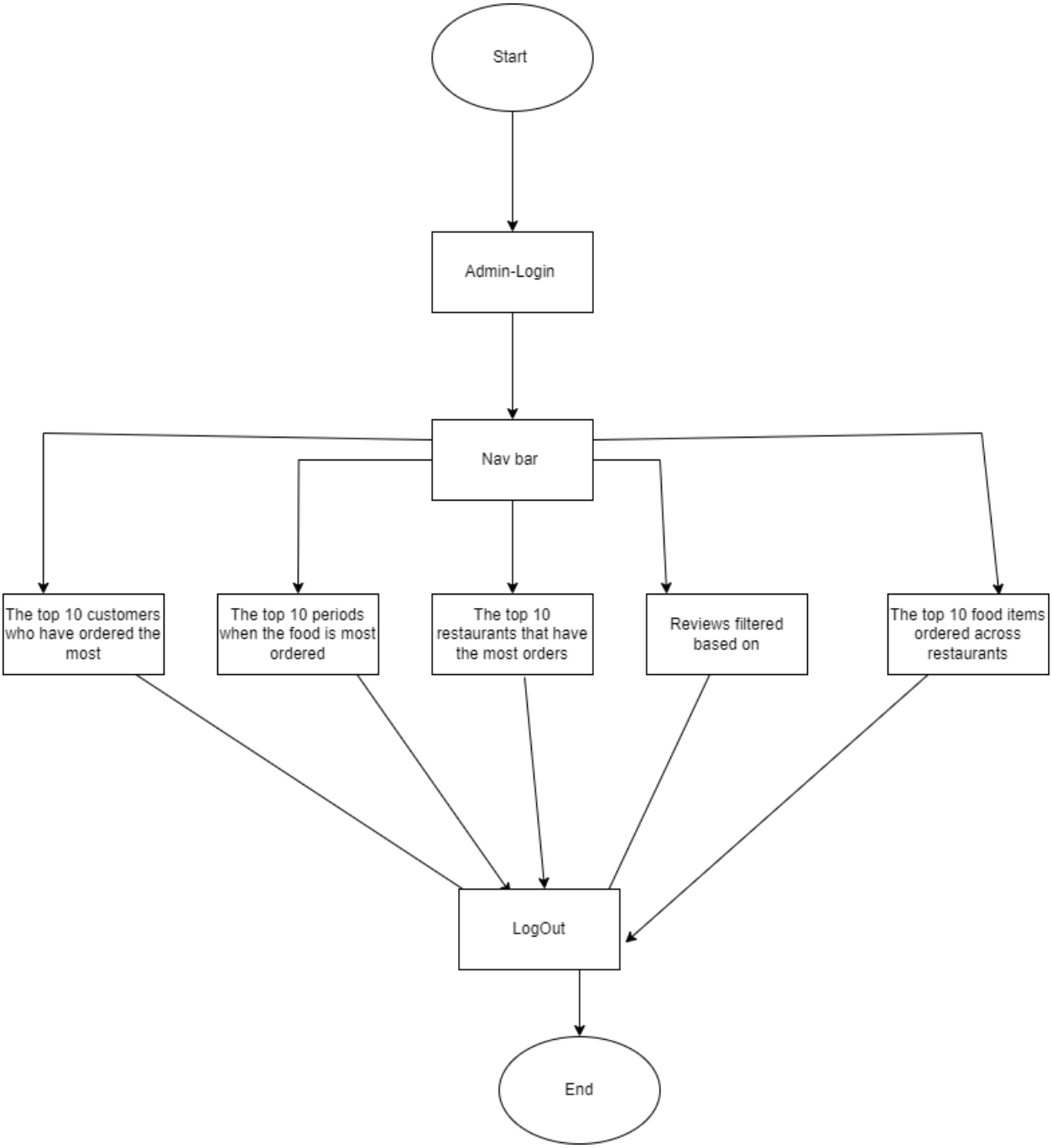


Figure 132 Flow Diagram for Admin App (Visualization). [19]

4. Application Architecture

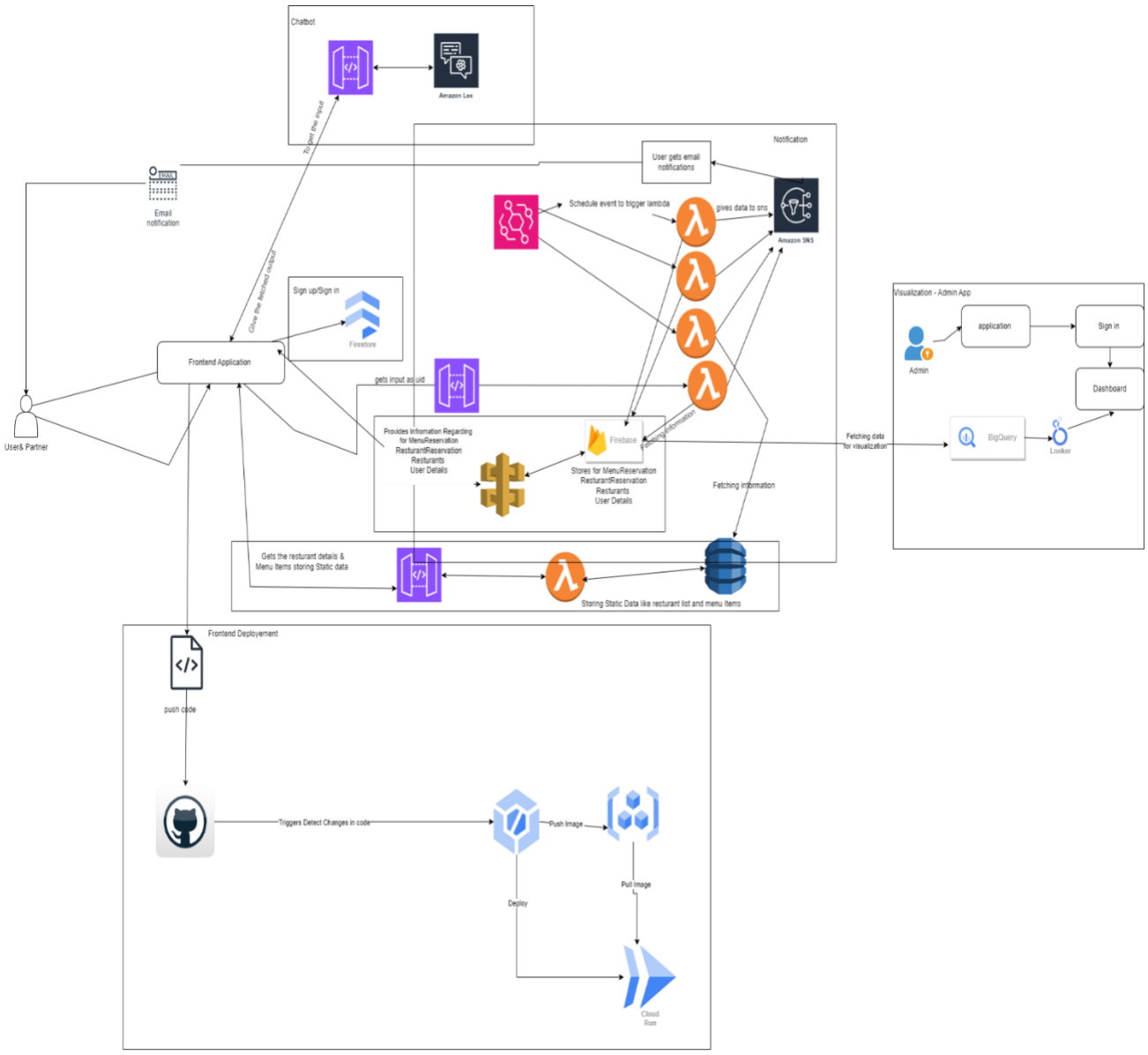


Figure 133 Application Architecture.[19]

5. Worksheet

Sr. No	Name Of Assignee	Customer App	Partner App	Admin App
1.	Rashmi Goplani	Sign Up & Login Module	Sign Up & Login Module	The top 10 periods when the food is most ordered
2.	Preeti Sharma	List Restaurants	Restaurant details	The top 10 food items ordered across restaurants
3.	Dheeraj Bhat	Book, edit, delete, view a reservation & CI/CD	view, edit, and delete a reservation & Holistic View	Reviews filtered based on restaurant names

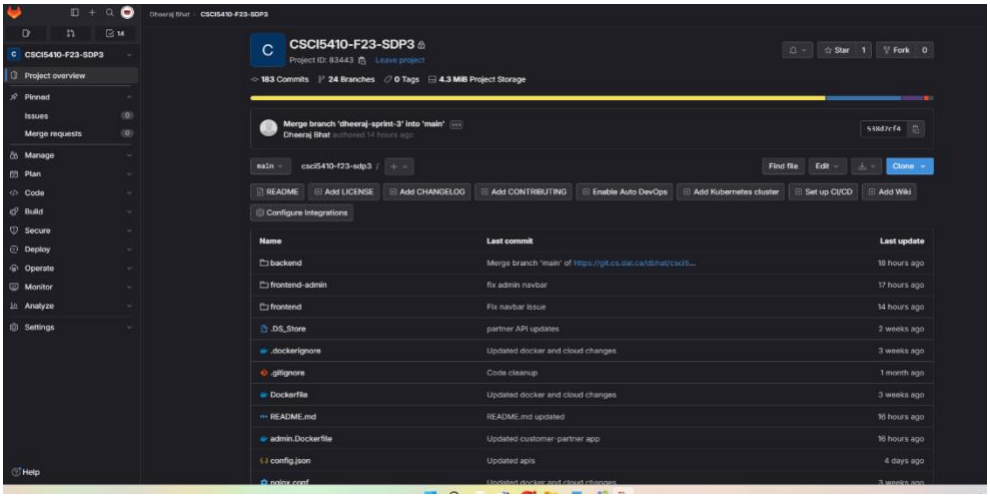
4.	Arihant Dugar	Book, edit, delete, view menu for a reservation	Edit, delete, view menu	The top 10 restaurants that have the most orders
5.	Gauravsinh Bharatsinh Solanki	Chatbot	Chatbot	
6.	Riya Patel	Notifications	Notifications	The top 10 customers who have ordered the most

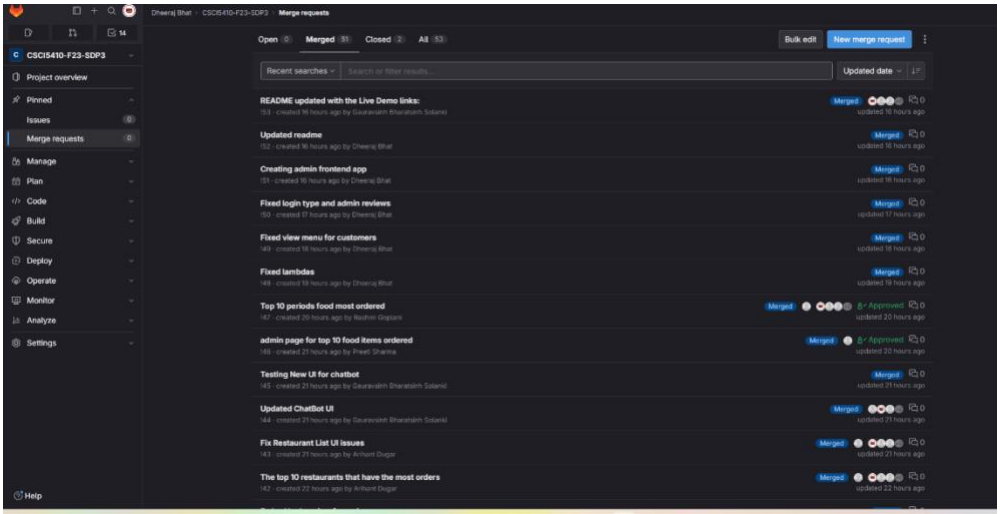
6. Sprint Plan

Sprint	Week	Module	Team Member	Task
1	1	Customer App	Rashmi Goplani	Sign Up & Login Module
1	1	Customer App	Preeti Sharma	List Restaurants
1	2	Customer App	Dheeraj Bhat	Book, edit, delete, view a reservation
1	2	Customer App	Arihant Dugar	Book, edit, delete, view menu for a reservation
1	3	Customer App	Gauravsinh Bharatsinh Solanki	Chatbot
1	3	Customer App	Riya Patel	Notifications
1	4 (From 5th Oct – 27th Oct)	-	-	Sprint Review and Preparation for Sprint 2
2	1	Partner App	Rashmi Goplani	Sign Up & Login Module
2	1	Partner App	Preeti Sharma	Restaurant details
2	2	Partner App	Dheeraj Bhat	View, edit, and delete a reservation
2	2	Partner App	Arihant Dugar	Edit, delete, view menu
2	3	Partner App	Dheeraj Bhat	Holistic View
2	3	Partner App	Gauravsinh Bharatsinh Solanki	Chatbot
2	3	Partner App	Riya Patel	Notifications
2	4 From (28th Oct- 17th Nov)	-	-	Sprint Review and Preparation for Sprint 3
3	1	Admin App	Arihant Dugar	Visualisations
3	2	Admin App	Preeti Sharma	Top 10 food items ordered

				across restaurants
3	2	Admin App	Rashmi Goplani	Top 10 periods when the food is most ordered
3	2	Admin App	Riya Patel	Top 10 customers who have ordered the most
3	2	Admin App	Dheeraj Bhat	Reviews filtered based on restaurant names
3	3 From (18th Nov- 28th Nov)	-	-	Sprint Review and Preparation for Project Demo and Report
Demo		-	-	Finalize Project Demo
Report		-	-	Prepare and Submit Project Report
Final	From (29th Nov- 3RD Dec)	-	-	Final Sprint Review

Screenshots of GitLab commits and merge.





7. Meeting Log

Sprint 1 Meeting Logs:

1. [10/5 11:23 PM]

Subject: Sprint 1 Review and Progress Update

Discussion:

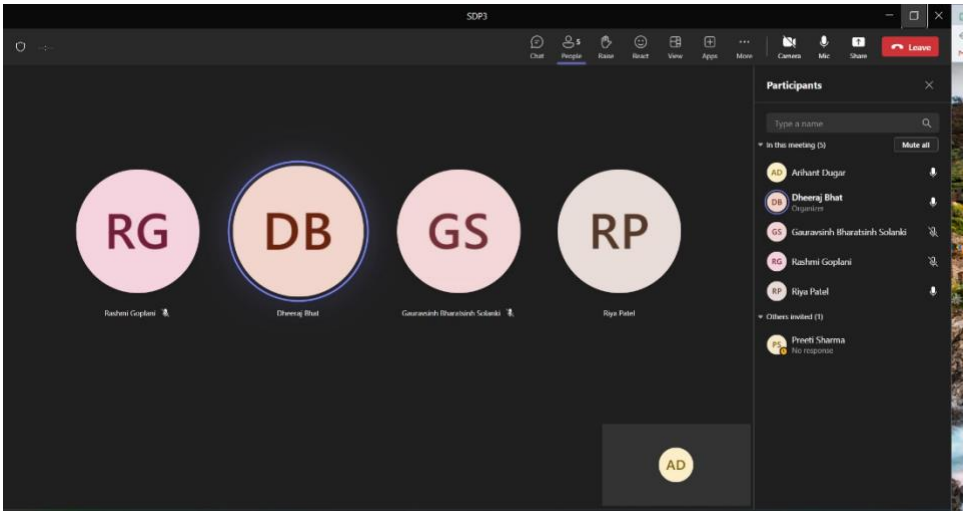
- Review of tasks for Customer App - Sprint 1
- Sign Up & Login Module, List Restaurants, Bookings, Menu Features
- Challenges and Resolutions

2. [10/22 7:14 PM]

Subject: Sprint 1 Completion and Planning for Sprint 2

Discussion:

- Overview of completed features.
- Planning for Sprint 2 - Partner App
- Coordination between Customer and Partner App teams



3. [10/26 9:46 PM]

Subject: Sprint 1 Retrospective

Discussion:

- Lessons learned from Sprint 1
- Improvements for future sprints
- Feedback and Suggestions

Sprint 2 Meeting Logs:

1. [11/6 10:30 PM]

Subject: Sprint 2 Kickoff and Progress Update

Discussion:

- Overview of tasks for Partner App - Sprint 2
- Sign Up & Login, Restaurant Details, Reservations, Menu Features
- Coordination with Customer App team

2. [11/24 11:02 PM]

Subject: Sprint 2 Completion and Sprint 3 Planning

Discussion:

- Recap of Partner App features implemented.
- Sprint 3 tasks for Admin App discussed.
- Any bottlenecks or challenges

Sprint 3 Meeting Logs:

1. [11/25 11:20 PM]

Subject: Sprint 3 Kickoff and Admin App Overview

Discussion:

- Introduction to Admin App tasks
- Integration with Looker Studio and Firebase
- Clarification of requirements

2. [12/3 6:15 PM]

Subject: Sprint 3 Finalization and Project Overview

Discussion:

- Review of Admin App features
- Final preparations for Project Report and Demo
- Deadline reminders and last-minute tasks

Project Report and Demo Meeting:

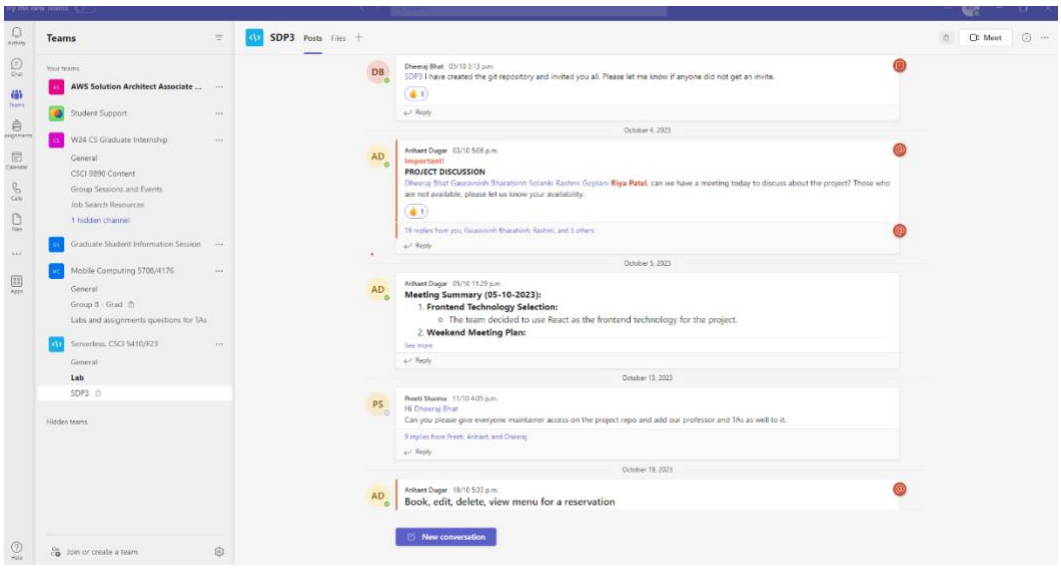
1. [12/03 6:15 PM]

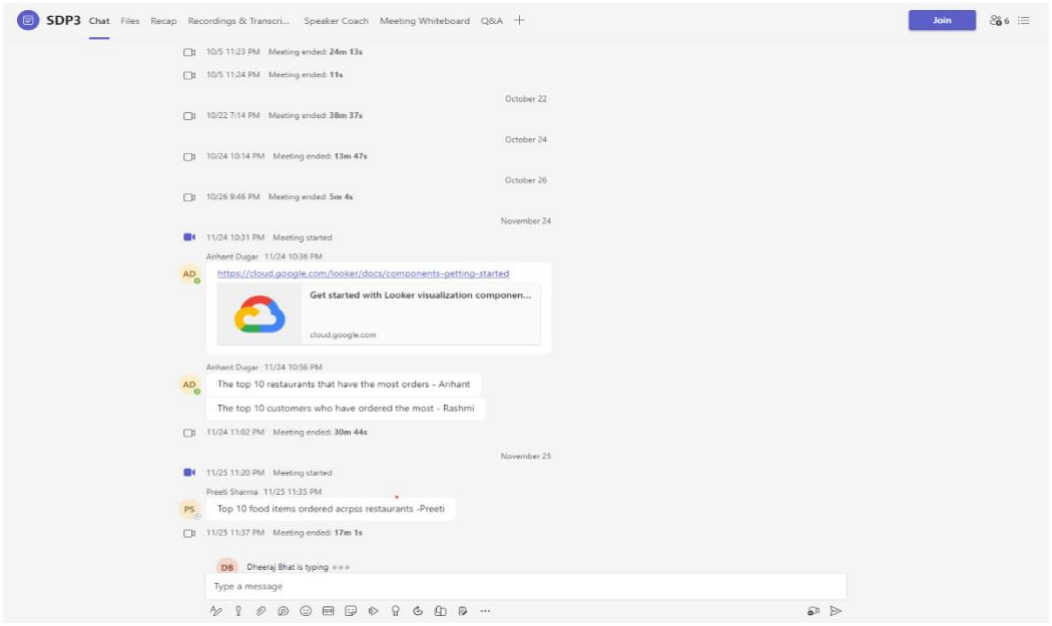
Subject: Project Report and Demo Preparation

Discussion:

- Final review of project milestones
- Demo walkthrough and preparations
- QA and last-minute checks

Screenshot Of Teams Group SPD3





8. Novelty

In a strategic move to streamline our development and deployment processes, we adopted an innovative approach right from the first sprint. Leveraging the power of Cloud Run and Docker, we successfully deployed our project, ensuring a seamless and efficient deployment pipeline. The integration with GitLab allowed us to mirror Docker artifacts, facilitating a smooth transition from code repository to deployment environment. This early deployment not only displays our commitment to agile practices but also establishes a robust foundation for continuous integration and delivery throughout the project lifecycle.

As part of our commitment to meeting project requirements, we meticulously addressed the needs of the admin app. We seamlessly integrated Firebase data, encompassing all collections, by linking it to BigQuery. Employing a well-crafted query, we curated a dataset that became the backbone of insightful visualizations. Leveraging Looker Studio, we connected this dataset to our frontend, generating dynamic reports that not only meet but exceed the expectations outlined for the admin app. This robust integration not only ensures compliance with project specifications but also elevates the user experience by providing rich, real-time insights through Looker Studio's powerful visualization capabilities. For reference see figure below.

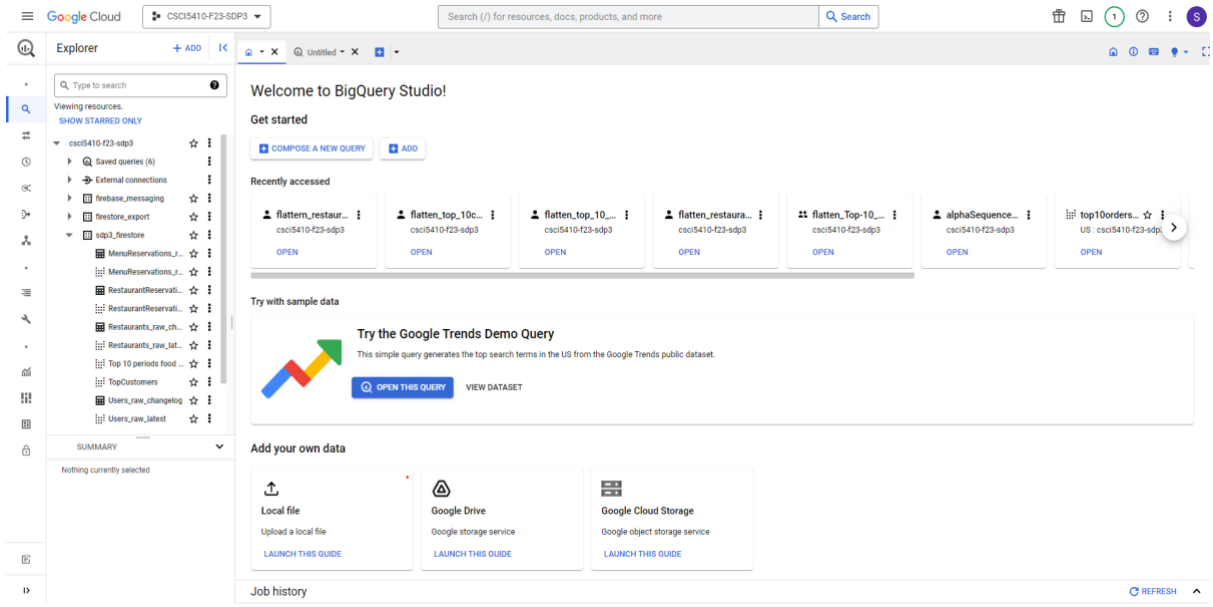


Figure 134 Group 3 BigQuery Linked Dataset from Firebase.[18]

9. Data Storage and Security

For our application, we have implemented robust data storage and security measures to uphold the principles of confidentiality, integrity, and availability within our system. Employing a dual-database strategy, we leverage Google Firestore for dynamic data, enabling real-time synchronization and scalability, while Firebase Authentication ensures secure user access through its authentication mechanisms during login and sign-up processes. Simultaneously, AWS DynamoDB is utilized for storing static data, such as restaurant lists and menu items, contributing to optimized performance and scalability.

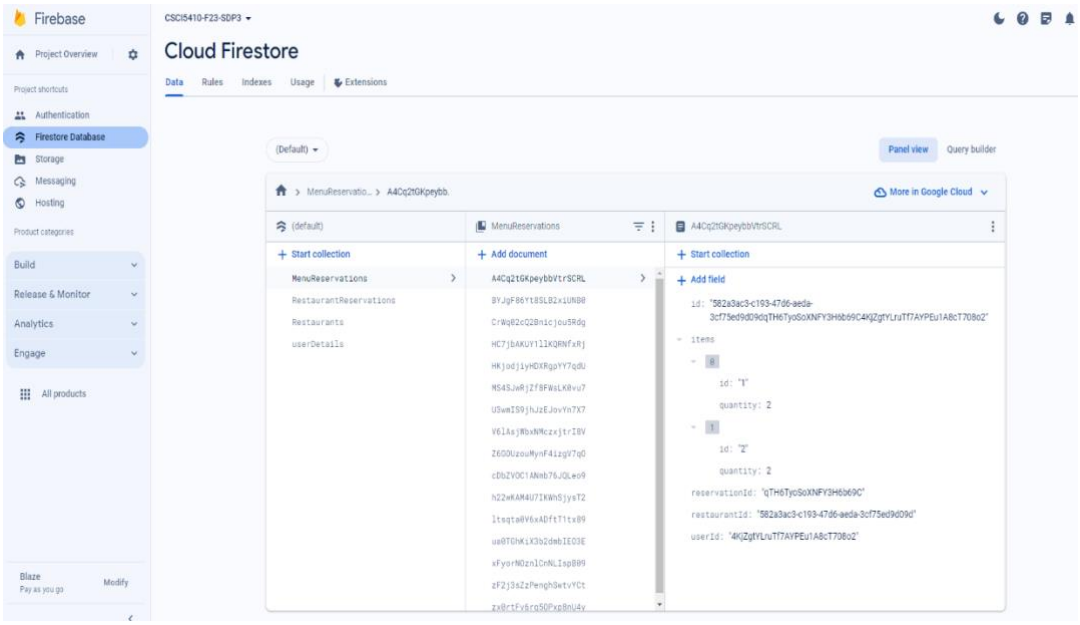


Figure 135 Firebase Database for our group [16].

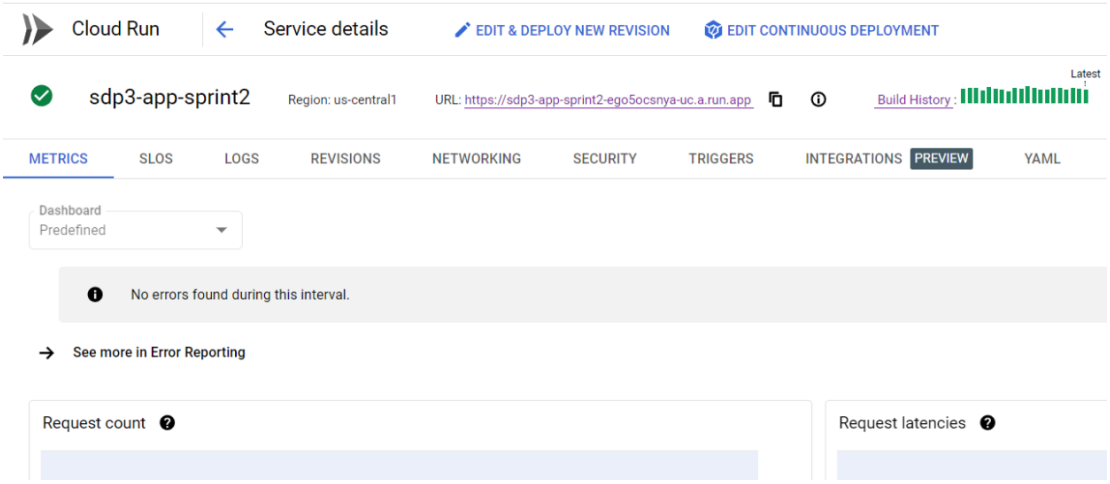
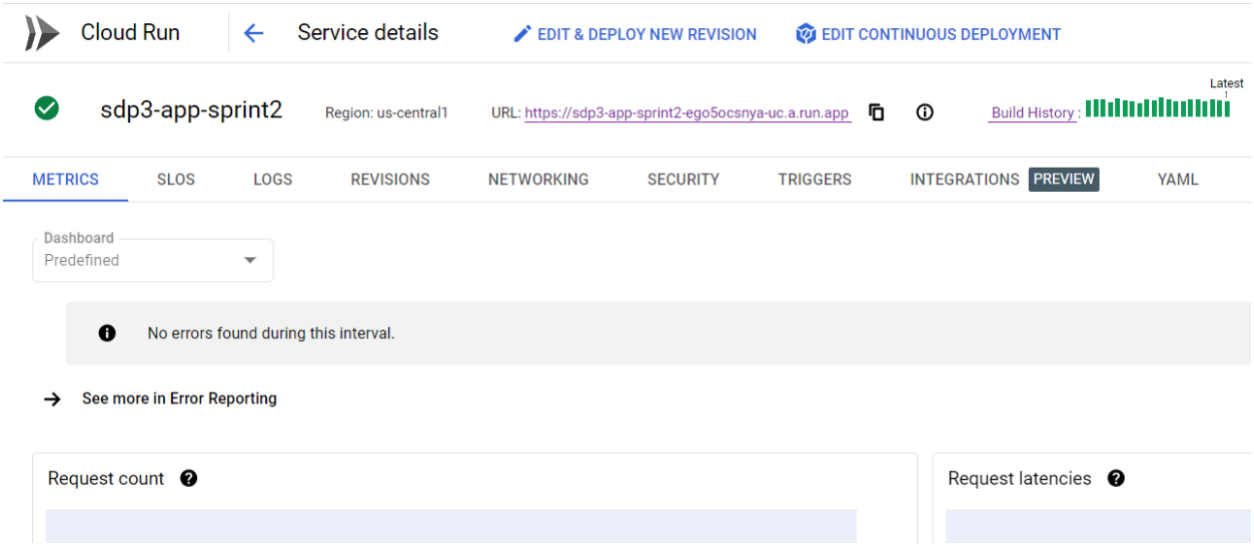


Figure 136 Cloud Run for Sprint 2[16]



Our security framework embraces role-based access control (RBAC) to assign specific roles to users, ensuring that access permissions align with their designated responsibilities. Additionally, we prioritize the encryption of Personally Identifiable Information (PII) to fortify sensitive user data against unauthorized access. Regular security audits are conducted to proactively identify and address vulnerabilities, providing a proactive approach to system security. We have also implemented an incident response plan to promptly and effectively address any security incidents that may arise.

The adoption of roles within our authentication system, including user, partner, and admin roles, contributes to a more fine-grained control over access privileges. This strategic use of Firebase and DynamoDB aligns with industry best practices and provides a solid foundation for maintaining a secure, compliant, and scalable system. As our application evolves, we remain committed to adhering to high-security standards and continuously enhancing our security posture.

10. Development Environment

In our development setup, we have embraced a contemporary and scalable strategy for deploying our frontend using Cloud Run with Docker. This approach allows us to encapsulate our application in a container and smoothly deploy it on Cloud Run, leveraging its serverless and container orchestration capabilities. Our Docker image is securely stored in Google Cloud's Artifact Registry, providing a centralized repository for managing our container images. This setup enhances version control and facilitates efficient image management throughout the development lifecycle.

For continuous deployment (CD), we have established a streamlined process through a mirrored GitHub repository. This arrangement integrates seamlessly with our development workflow, enabling automatic deployment triggers whenever changes are pushed to the GitHub repository. Using Cloud Build or other

CI/CD tools, we have configured a pipeline that automates the build, test, and deployment processes, ensuring quick and reliable delivery of updates to Cloud Run.

This development environment setup allows us to leverage the scalability and efficiency of Cloud Run, [13] the robustness of Docker containers [14], and the seamless integration of GitHub for version control and CD. This strategy aligns with our objective of maintaining an agile and efficient development workflow while utilizing advanced cloud technologies for optimal deployment and scalability.

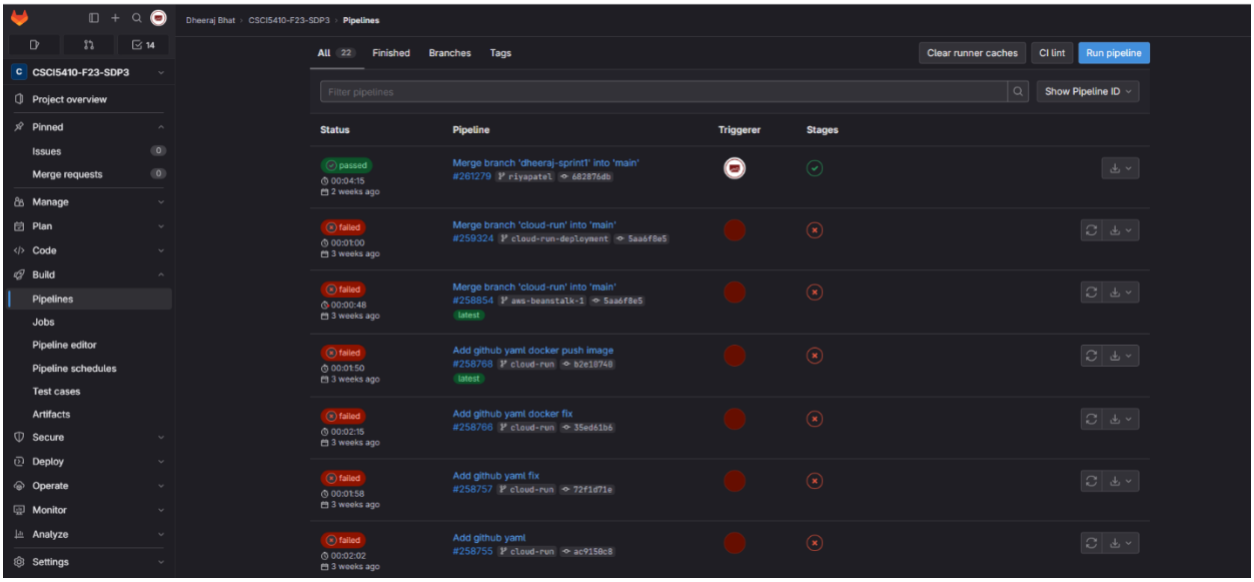


Figure 137 Git Pipeline [15]

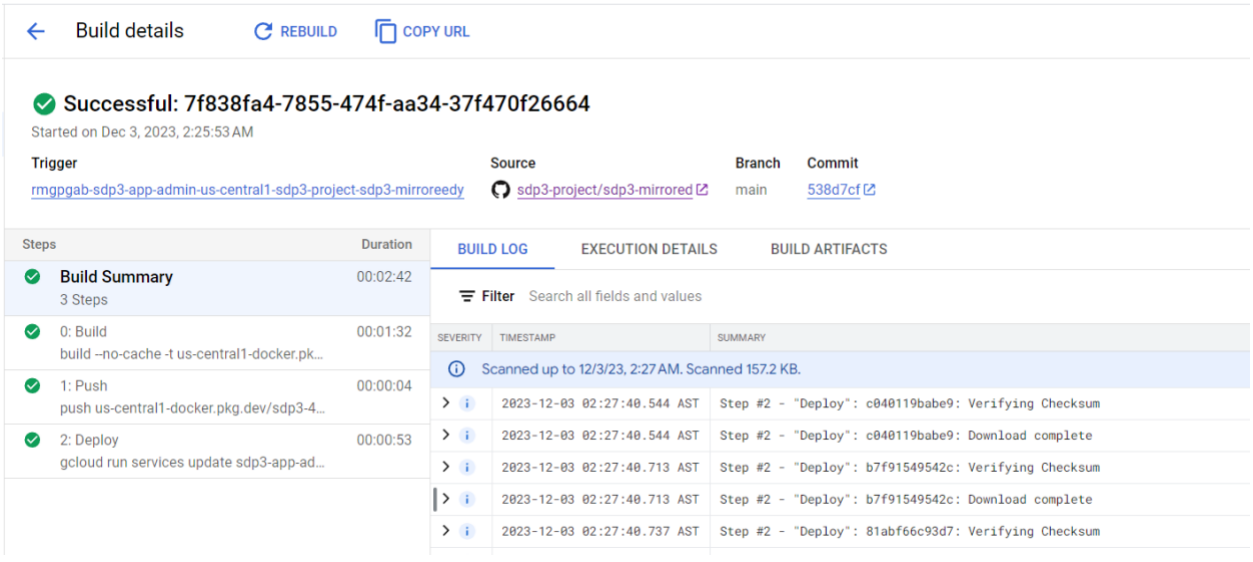


Figure 138 Build Details [16]

11. Demo URL:

- Customer and partner app: https://sdp3-app-sprint2-ego5ocsnya-uc.a.run.app/
- Admin app: https://sdp3-app-admin-ego5ocsnya-uc.a.run.app/
- [admin credential - email: sysadmin@gmail.com / password: password / userType: admin]
- Git repository: https://git.cs.dal.ca/dbhat/csci5410-f23-sdp3.git

12. References:

[1] "How to build an AI Chatbot using Amazon Lex and Lambda, and Integration with ReactJS," *DEV Community*, 15-Mar-2023. [Online]. Available: https://dev.to/onlyoneerin/how-to-build-an-ai-chatbot-using-amazon-lex-and-lambda-and-integration-with-reactjs-592j. [Accessed: 04-Dec-2023].

[2] *Amazon.com*. [Online]. Available: https://docs.aws.amazon.com/lexv2/latest/dg/what-is.html. [Accessed: 04-Dec-2023].

[3] *Amazon.com*. [Online]. Available: https://docs.aws.amazon.com/lexv2/latest/dg/lambda.html. [Accessed: 04-Dec-2023].

[4] "Flowchart maker & online diagram software," *Diagrams.net*. [Online]. Available: https://app.diagrams.net/. [Accessed: 04-Dec-2023].

[5] *Amazon.com*. [Online]. Available: https://docs.aws.amazon.com/lexv2/latest/dg/sample-utterances.html. [Accessed: 04-Dec-2023].

[6] "AWS Lex V2 - Network of Bots - Error: There was an error getting the sentiment for this request. Try your request again," *Stack Overflow*. [Online]. Available: https://stackoverflow.com/questions/75461993/aws-lex-v2-network-of-bots-error-there-was-an-error-getting-the-sentiment-f. [Accessed: 04-Dec-2023].

[7] Kommunicate, "Kommunicate," *Kommunicate.io*. [Online]. Available: https://dashboard.kommunicate.io/. [Accessed: 04-Dec-2023].

[8] "Firestore, Firebase," *Google*. [Online]. Available: https://firebase.google.com/docs/firestore. [Accessed: 04-Dec-2023]

[9] S. Vişan, "AWS Lambda," *Amazon Web Services*. [Online]. Available: ttps://docs.aws.amazon.com/lambda/. [Accessed: 04-Dec-2023]

[10] "Amazon API Gateway Documentation," *Amazon Web Services*. [Online]. Available: https://docs.aws.amazon.com/apigateway/. [Accessed: 04-Dec-2023]

[11] "Cloud run documentation," *Google*. [Online]. Available: https://cloud.google.com/run/docs. [Accessed: 04-Dec-2023]

[12] "Get Started with Firebase Authentication on Websites," *Firebase*. [Online]. Available: https://firebase.google.com/docs/auth/web/start. [Accessed: 04-Dec-2023]

[13] "Cloud Run," Google Cloud. [Online]. Available: https://cloud.google.com/run?hl=en. [Accessed: 03-Dec-2023].

[14] “Artifact Registry overview,” Google Cloud. [Online]. Available: https://cloud.google.com/artifact-registry/docs/overview. [Accessed: 03-Dec-2023].

[15] “Continuous deployment from Git using Cloud Build,” Google Cloud. [Online]. Available: https://cloud.google.com/run/docs/continuous-deployment-with-cloud-build. [Accessed: 03-Dec-2023].

[16] “Deploying to Cloud Run,” Google Cloud. [Online]. Available: https://cloud.google.com/run/docs/deploying. [Accessed: 03-Dec-2023].

[17] “Getting started with Lambda,” *Amazon.com*. [Online]. Available: https://docs.aws.amazon.com/lambda/latest/dg/getting-started.html. [Accessed: 04-Dec-2023].

[18] “BigQuery – CSCI5410-F23-SDP3 – Google Cloud console,” Google.com. [Online]. Available: https://console.cloud.google.com/bigquery?project=csci5410-f23-sdp3&authuser=2&ws=!1m0. [Accessed: 04-Dec-2023].

[19] “Untitled Diagram - draw.io,” Diagrams.net. [Online]. Available: https://app.diagrams.net/. [Accessed: 04-Dec-2023].