DALHOUSIE UNIVERSITY

CSCI-5410 SERVERLESS DATA PROCESSING

Project Final Report

GROUP-10

GROUP MEMBERS:

Mugdha Anil Agharkar (B00872601)

Jay Bhagvanbhai Sonani (B00891984)

Ridham Ghanshyambhai Kathiriya (B00893712)

Vivekkumar Patel (B00896765)

Rahul Kherajani (B00884966)

Pankti Vyas (B00886309)

Course Instructor: Dr. Saurabh Dey

TABLE OF CONTENTS

1. Project Overview

The **DALSoft5410** is developing cloud-based hotel reservation system **Serverless B&B**. As a company who delivers serverless solutions, we are following **multi-cloud** deployment and **backend-as-a-service (BaaS)** approach to reduce the development efforts and operational cost of the application. This strategy was proposed to address the issue with a server-hosted solution, which is that we must manually manage and setup the backend service, which is impractical due to resource constraints. There will be two types of users who will be using the application, the guest users, and the authorized users. Authorized users will be able to access additional services, but guest users will only be able to access a restricted number of services. The customers will be able to sign up and sign in inside the application. Customers will be able to reserve hotel rooms and place food orders through the kitchen services. The invoice will be created by the hotel administration based on the length of stay and the amount of food consumed. The user will receive virtual support for tasks like basic site navigation and managing reservations and food orders. Customers can also choose to use an additional tour operator service, which allows them to request trips and has them advised depending on their stay frequency and activities. [1]

For our multi cloud deployment architecture, we will use **Amazon Web Services (AWS)** and **Google Cloud Platform (GCP)** as cloud providers. We have decided to develop our frontend using **React.js** framework for easing and reducing the development efforts and for the serverless backend, we will be using **Node.js** and **Python** for creating lambdas for different microservices. The development process will follow agile methodology for rapid delivery of application in incremental stages.

2. Technology Requirements

The below technologies and tools are used in the project

Table 1: Table displaying tools and technologies used in this project.

Technologies & Tools	
Tools	Visual Studio Code, PyCharm
Serverless Cloud Providers	AWS, GCP
Programming Language	Python3.9, NodeJS
Frontend Framework	ReactJS
GitLab	Version Control System

3. Project Implementation

We have used below AWS and GCP cloud services for each module [1].

Overview

1. User Management Module

Tasks	Cloud Service used for task
Registration	AWS Cognito

2. User Authentication Module

Tasks	Cloud Service used for task
ID-Password	AWS Cognito
Question-Answer	GCP Firestore and Cloud function
Caesar Cipher	GCP Cloud Function

3. Online Support Module

Tasks	Cloud Service used for task
Interactive AI Chat Bot	AWS Lex and AWS Lambda

4. Message Passing Module

Tasks	Cloud Service used for task
Message Passing through Pub/Sub	GCP Pub/Sub, Firebase Cloud Messaging

5. Machine Learning

Tasks	Cloud Service used for task
Tour recommendations	Google Cloud, Vertex AI, Auto-ML, (Tabuler data classification model)
Polarity Score from customer's feedback	Google Cloud, Vertex AI, Auto-ML, (Sentiment Analysis model)

6. Web Application Building and Hosting

Tasks	Cloud Service used for task
Application Hosting	AWS Amplify

7. Other Essential Module

Tasks	Cloud Service used for task
Report Generation	GCP Cloud Function
Testing	AWS Lambda
Visualization	GCP Data Studio

The detailed explanation of each module is as follows:

Module 1: User Management Module

The entry point to the Serverless B&B application are the registration steps. Users will be prompted to enter details such as first name, last name, email, and password using a registration form.

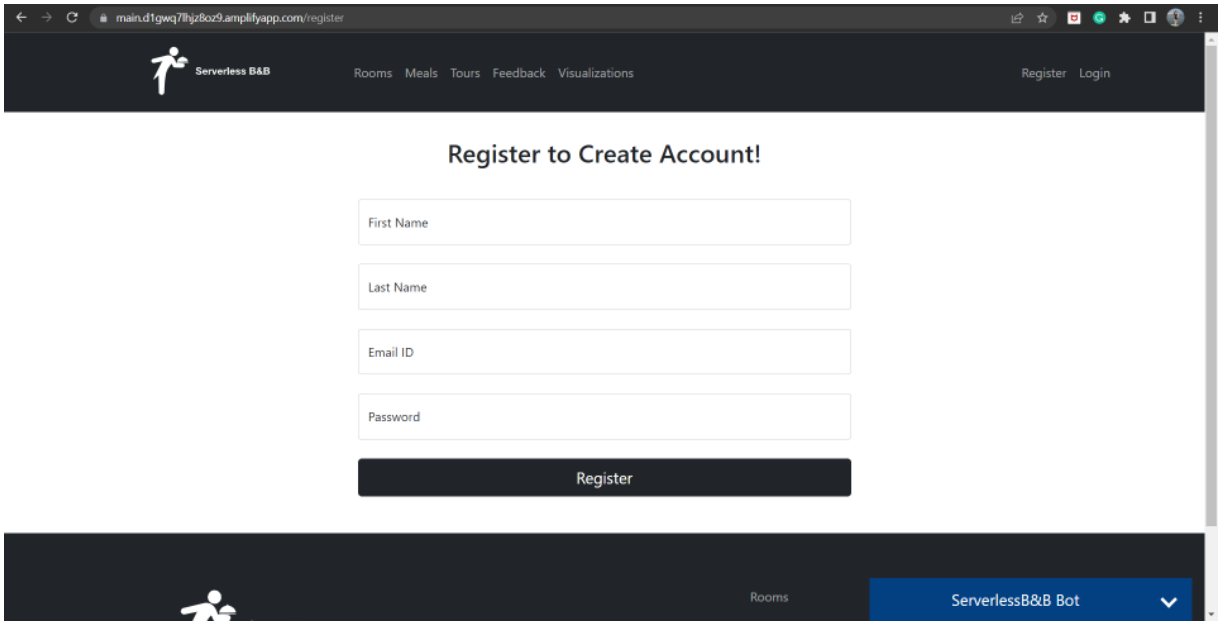

Figure 1: Registration Form.

User inputs will be validated for null values, invalid email format, email already exists or password length and special character requirements.

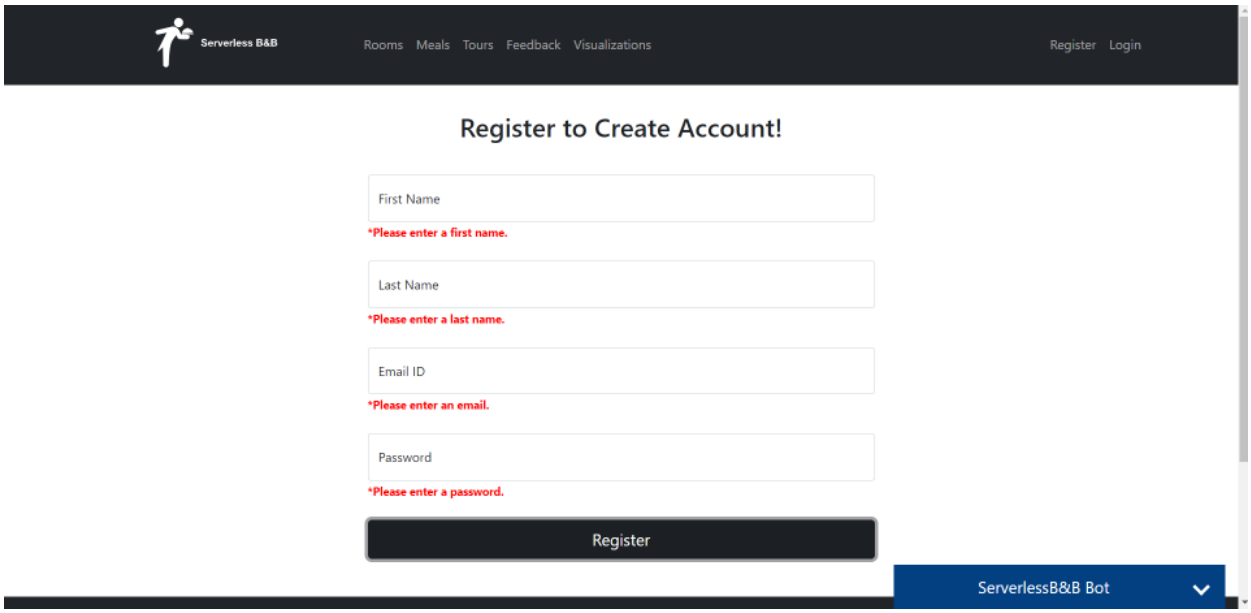

Figure 2: Null values validation in Registration form.

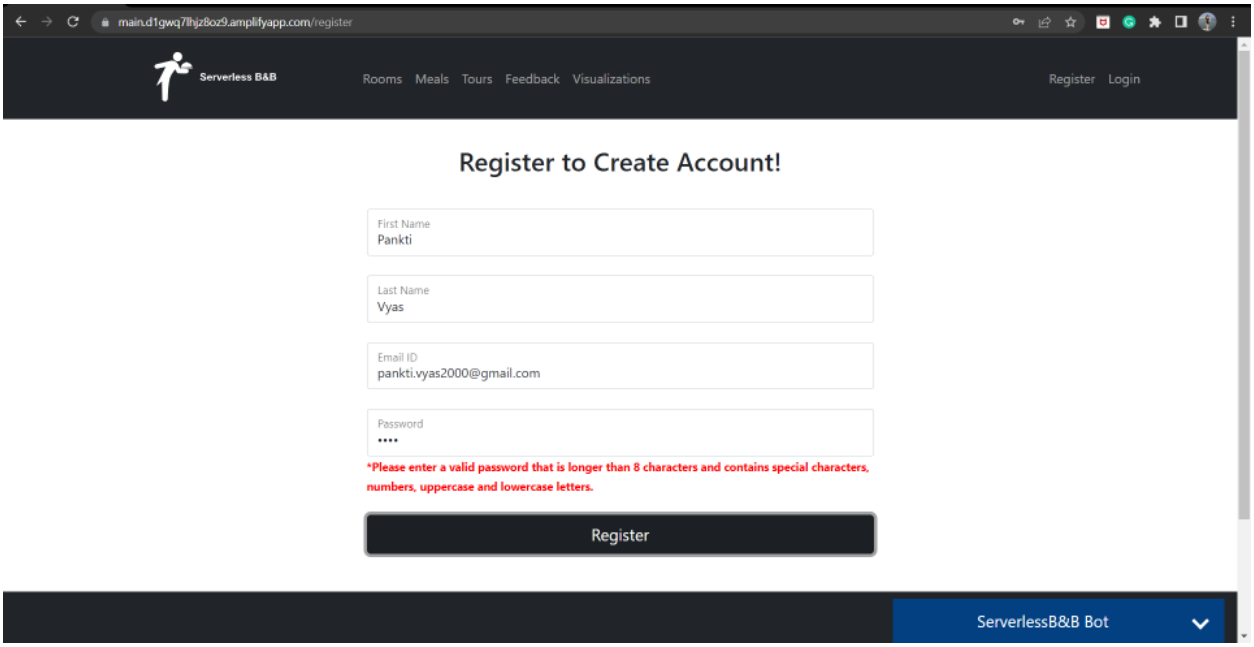

Figure 3: Password constraints validation in Registration form.

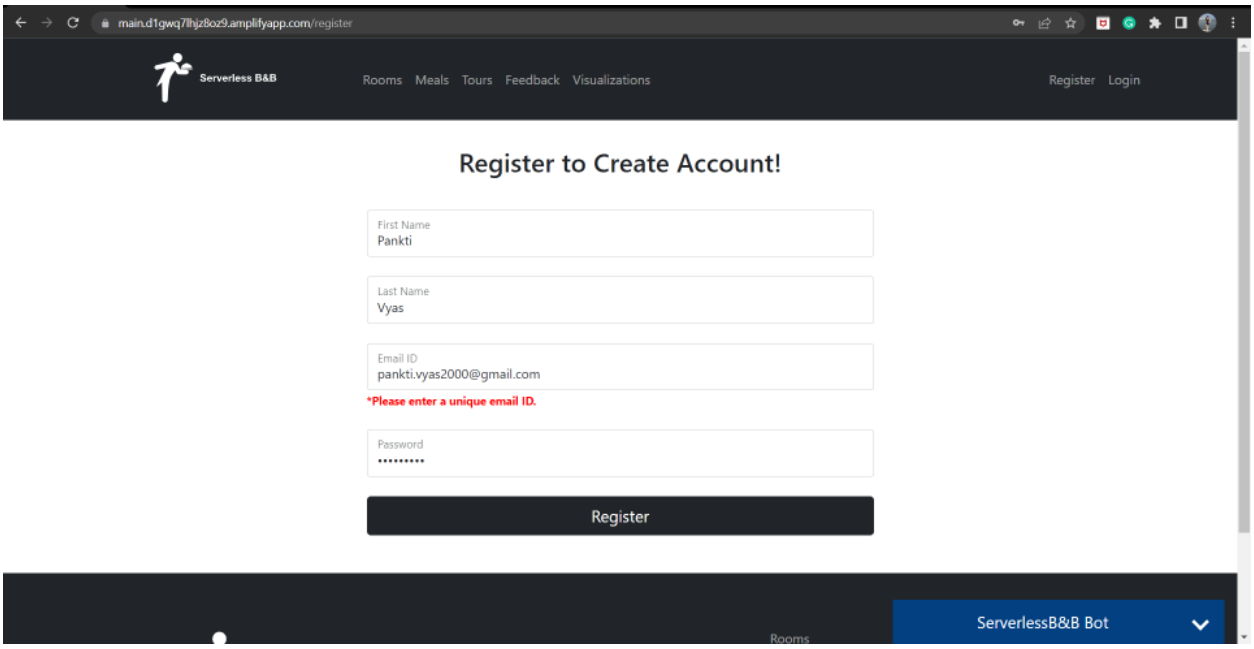

Figure 4: Unique email validation in Registration form.

Once the inputs are validated and user clicks on the Register button, an OTP will be sent to the users' email address as well as a new user will be created in AWS Cognito userpool with 'unconfirmed' status. Once the user validates the received otp in the Serverless B&B application, the status of user in Cognito userpool changes to 'confirmed'.

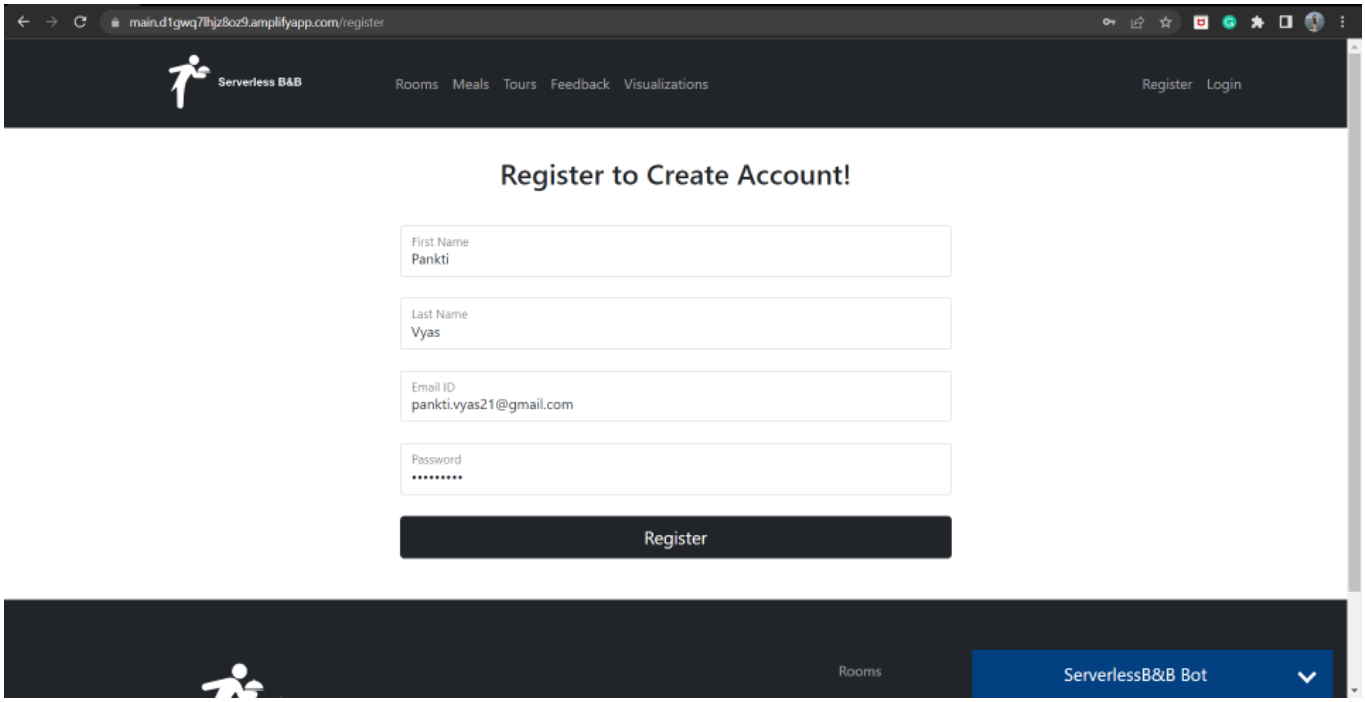

Figure 5: Registration form filled.

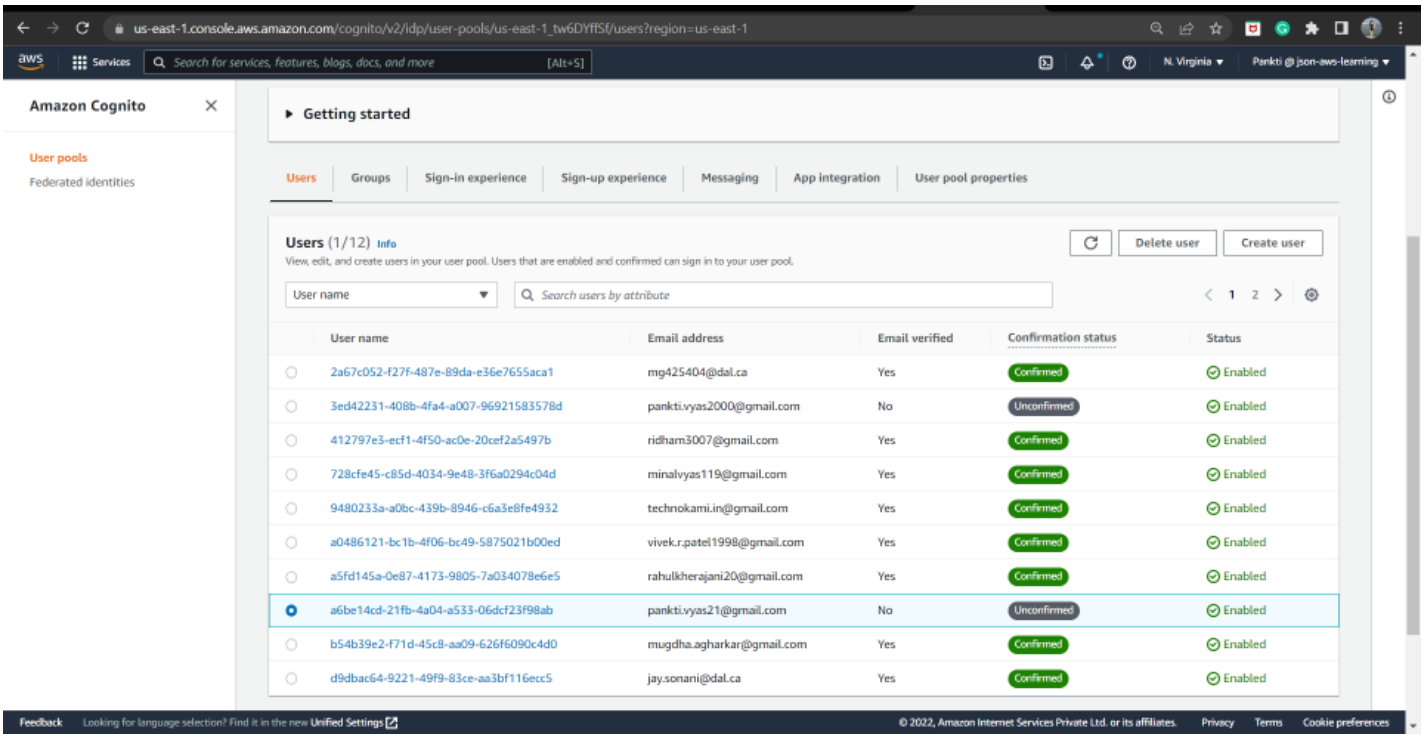

Figure 6: User created in Cognito userpool with 'Unconfirmed' status.

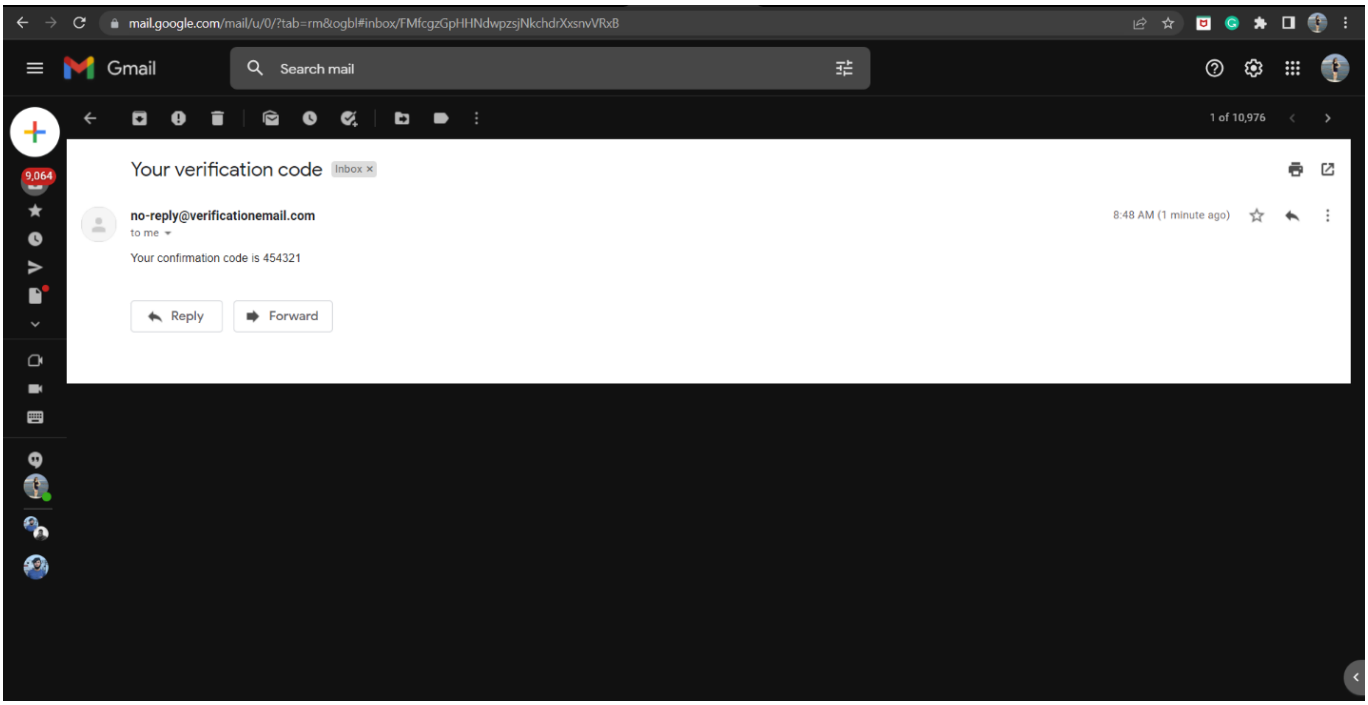

Figure 7: OTP received in user's email inbox.

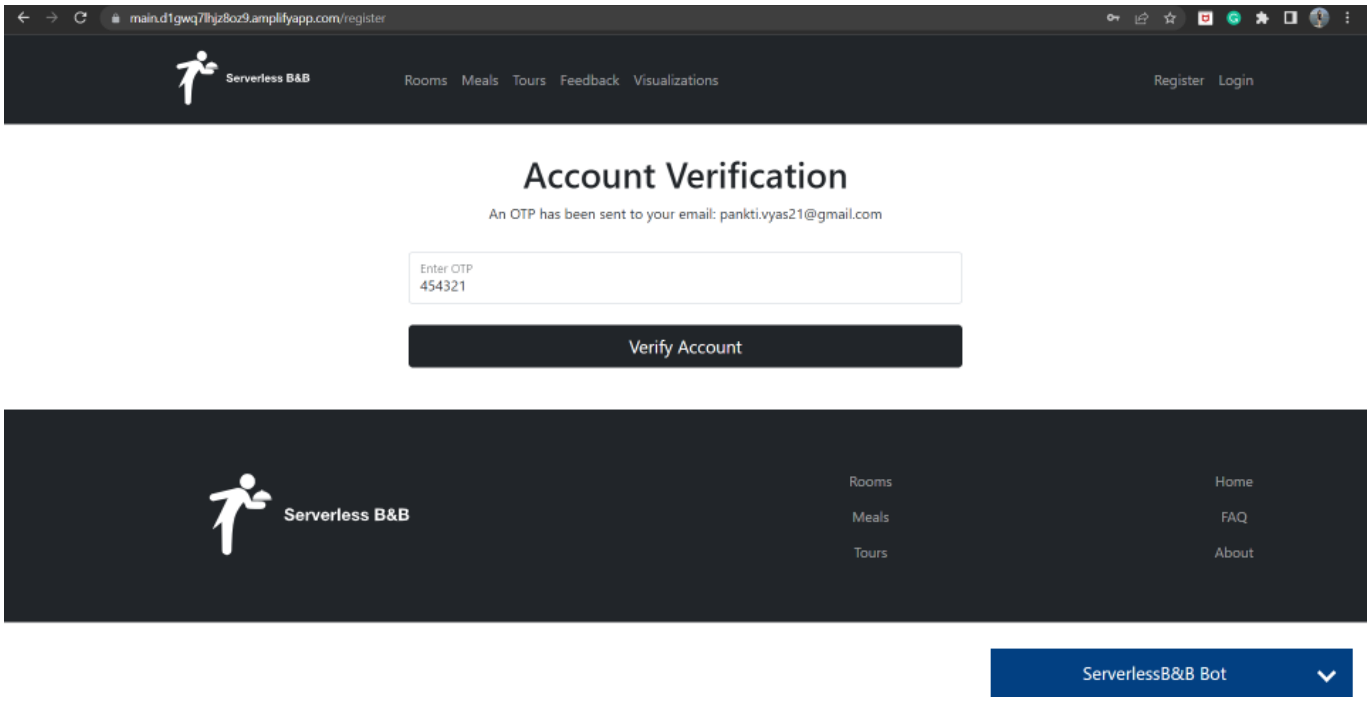

Figure 8: OTP entered in the application.

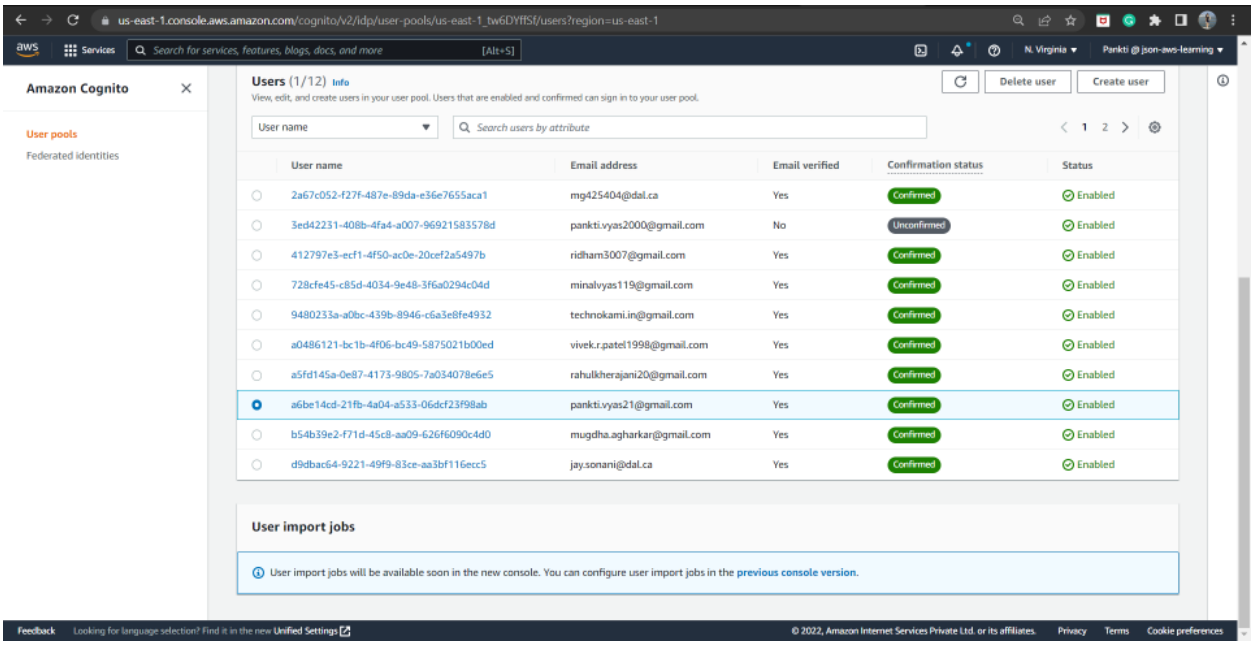

Figure 9: User status changes to 'Confirmed' after otp validation.

After the otp is validated, user will be shown a dropdown of 5 security question. User can select any one of them and enter the answer in the text box. The selected question, answer and user's email will be sent to a cloud function 'store-security-question' as parameters in a post request body. The cloud function 'store-security-question' adds these parameters to a 'security-questions' collection in GCP Firestore.

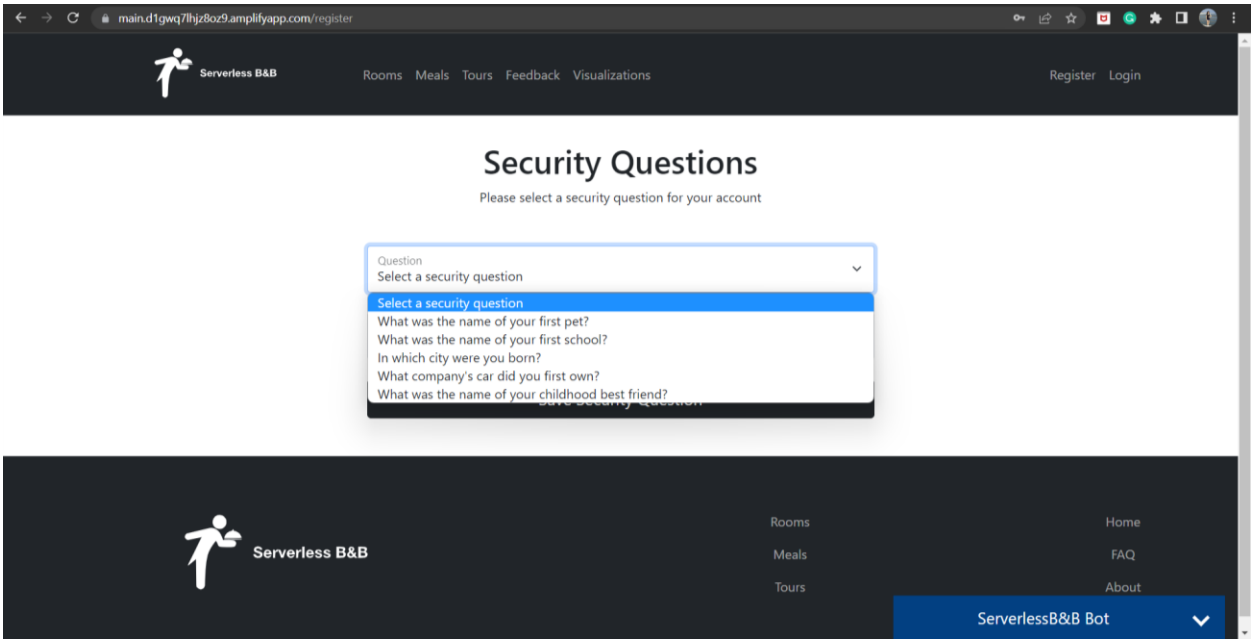

Figure 10: Security Questions dropdown.

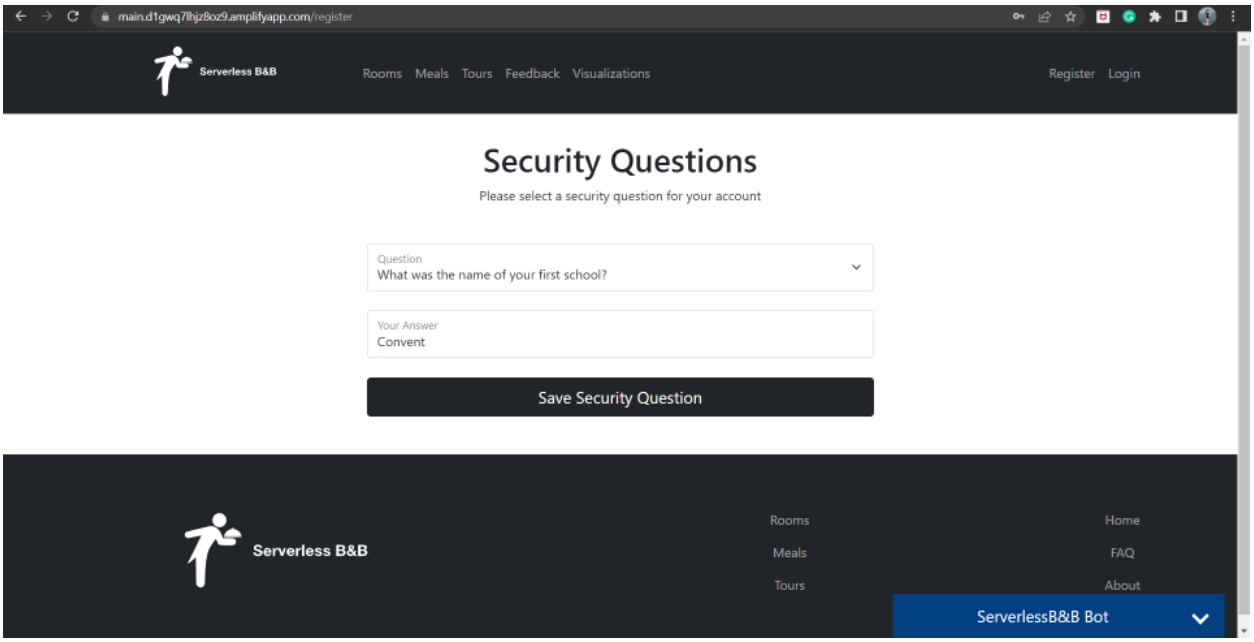

Figure 11: Security Question selected, and answer entered.

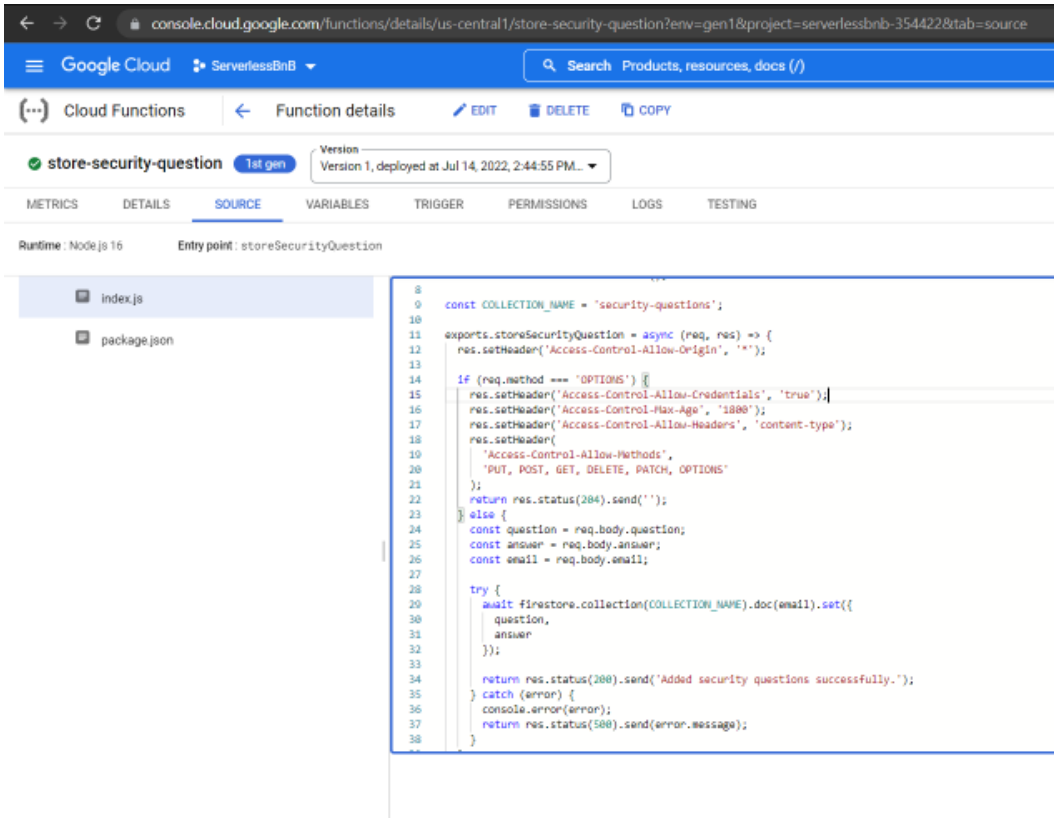

Figure 12: store-security-question cloud function.

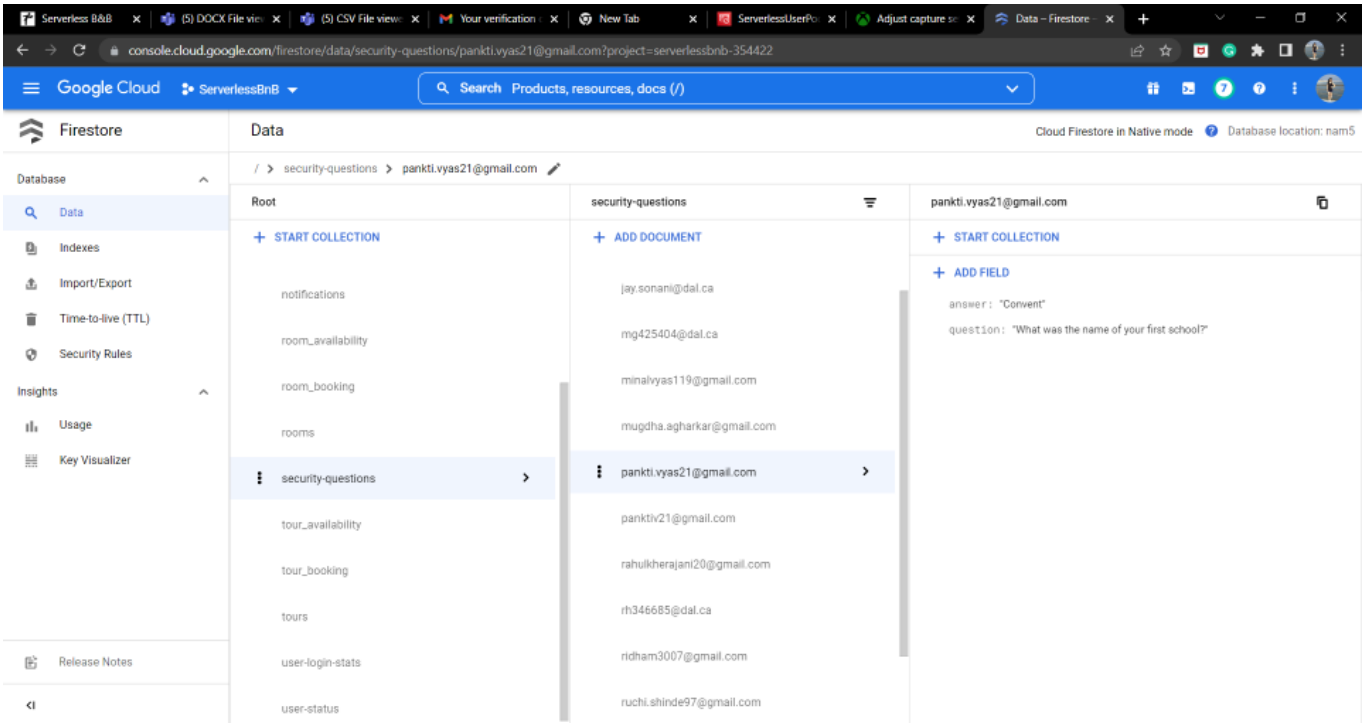

Figure 13: Document added to 'security-questions' collection.

For the final registration step, user is asked to enter a Caesar cipher key between 0 and 25. This key will be used on login to authenticate the user. The entered key and user's email will be sent to a cloud function 'store-cipher-key' as parameters in a post request body. The cloud function 'store-cipher key' creates a document in the 'caeser-keys' collection in GCP Firestore with users email and adds the key to the document.

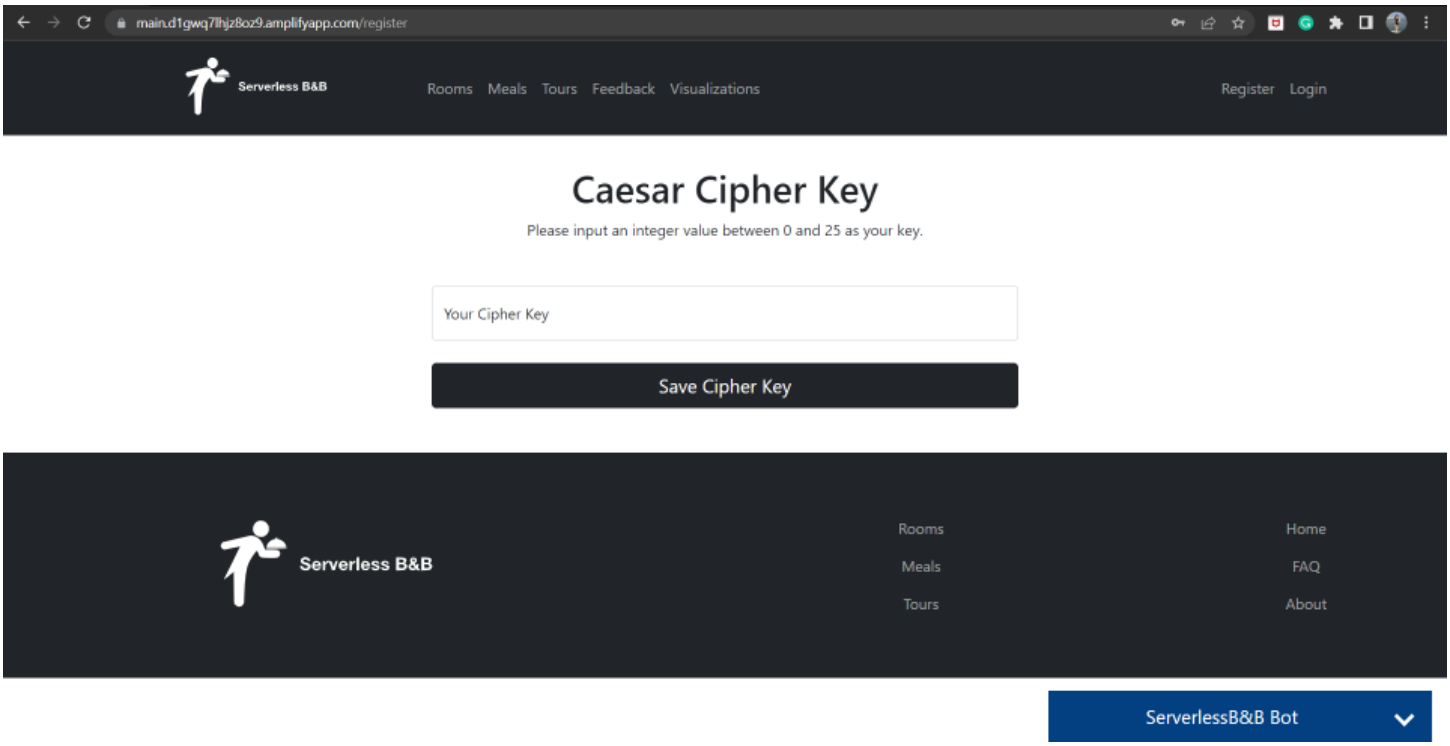

Figure 14: Cipher key input.

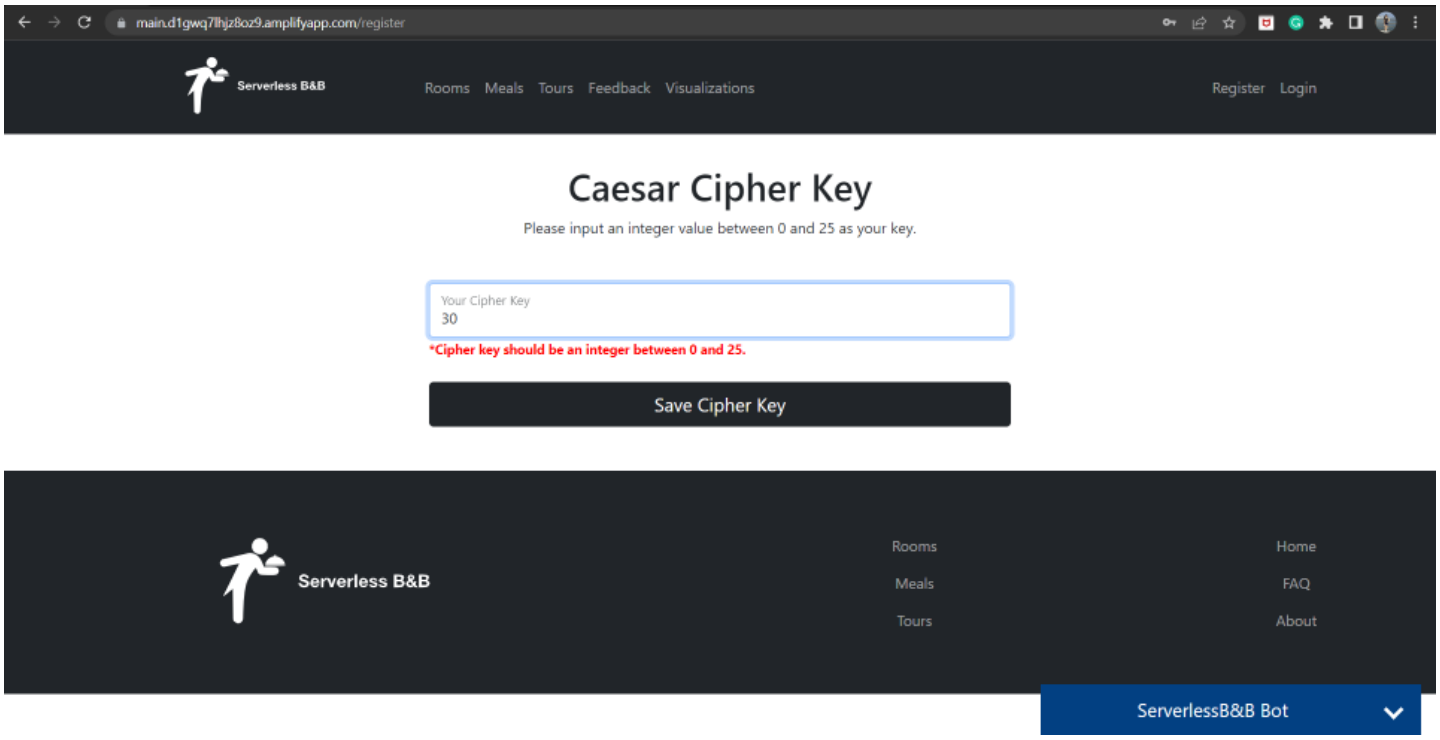

Figure 15: Cipher key validation for an integer between 0 and 25.

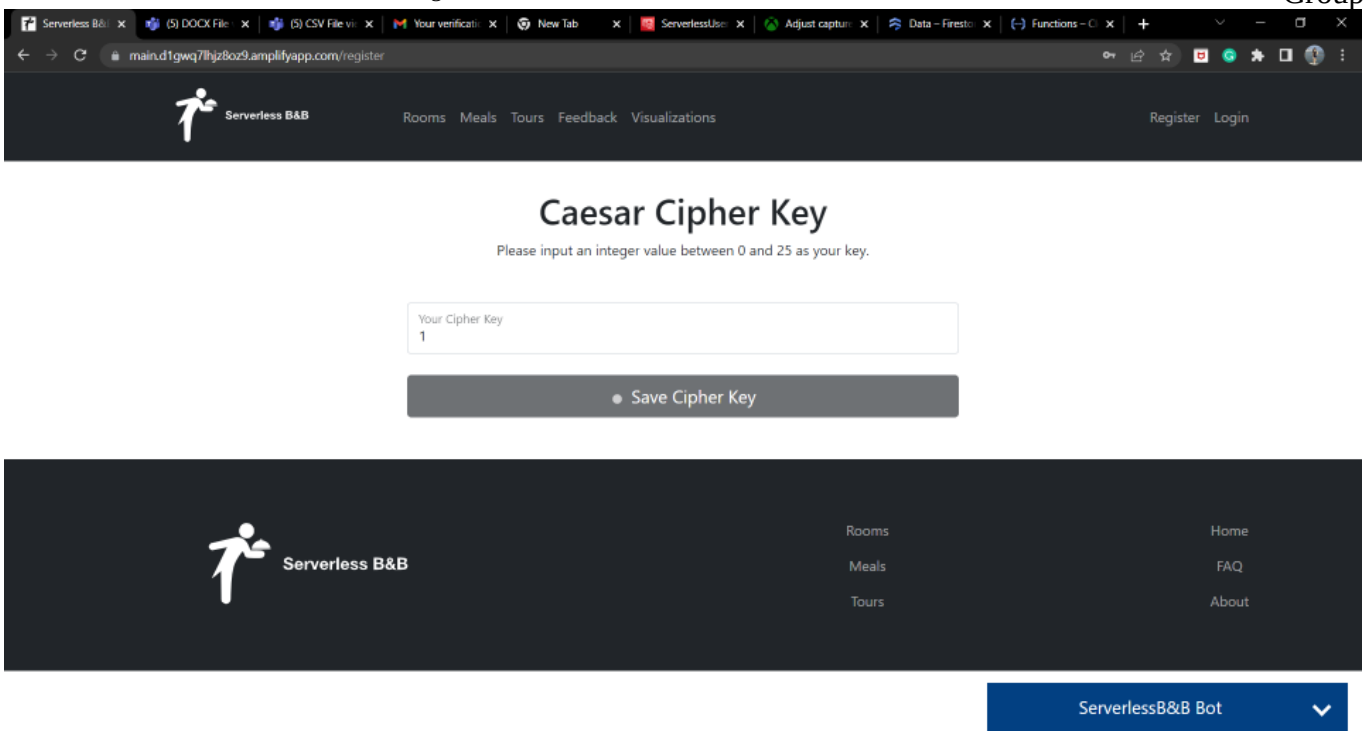

Figure 16: A valid cipher key entered.

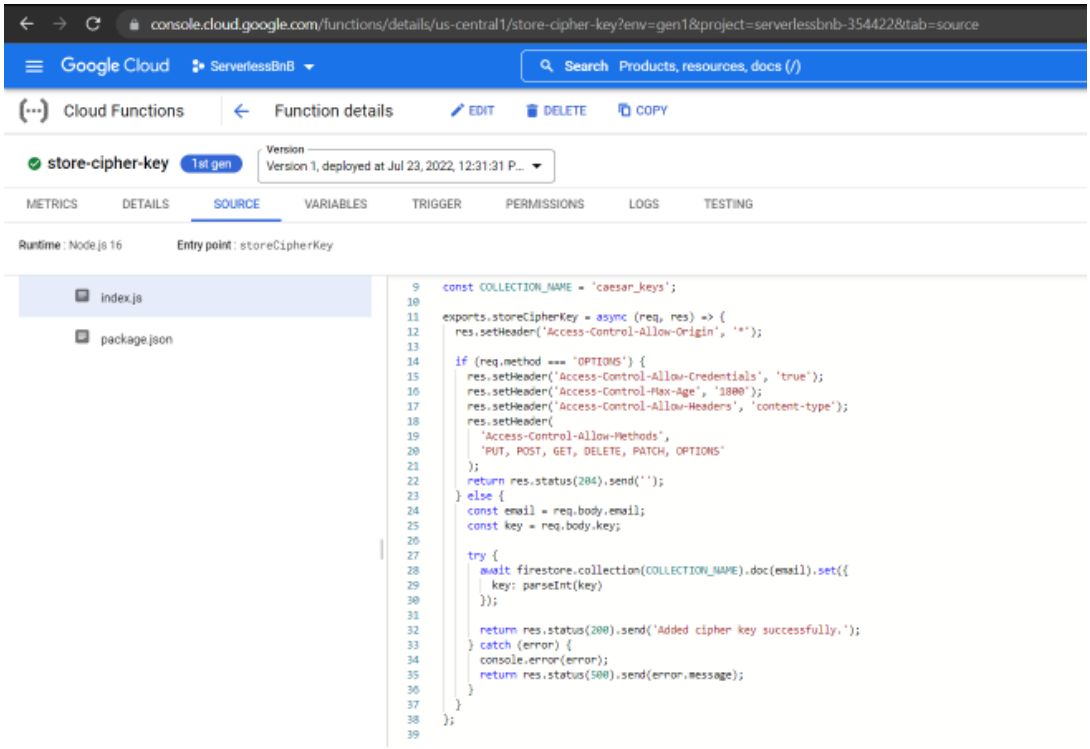

Figure 17: 'store-cipher-key' cloud function.

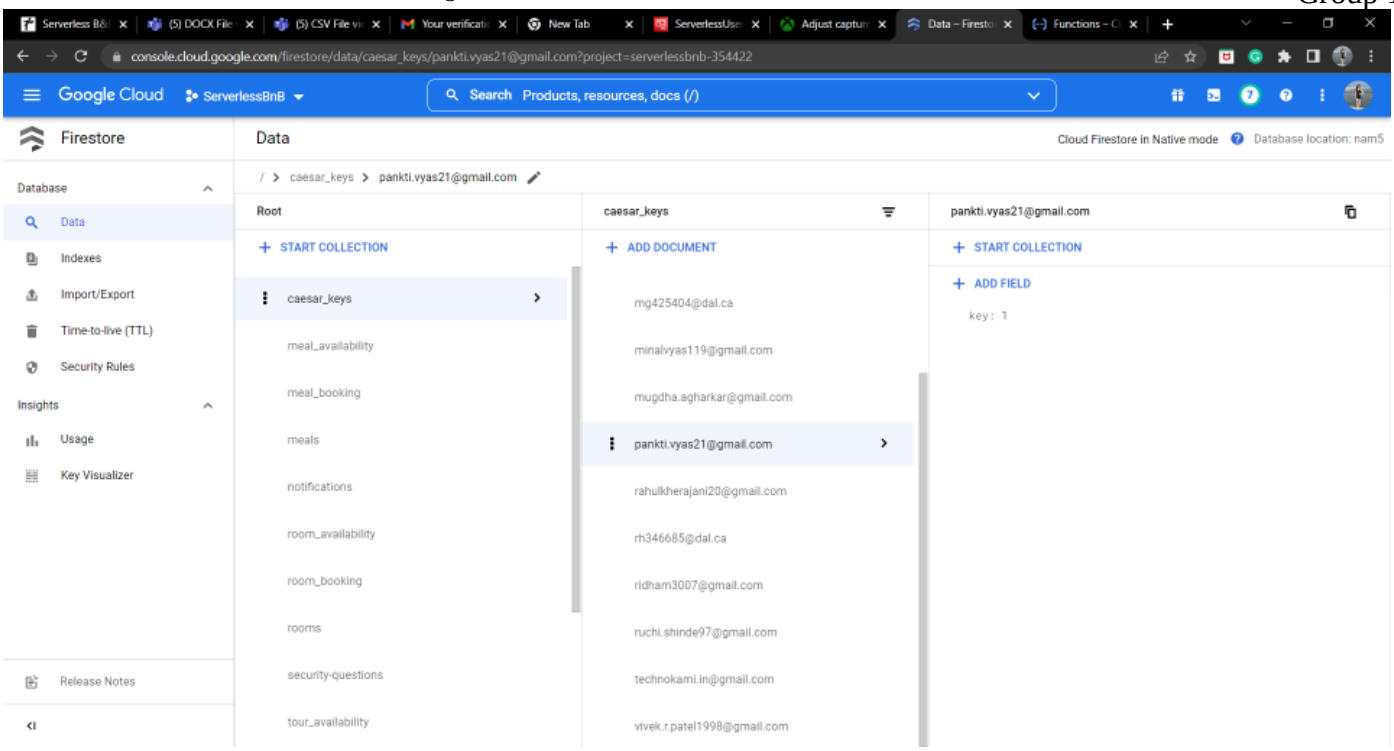

Figure 18: Document added to 'Caesar keys' collection.

Now the registration steps are complete, user is asked to Login and provided with a link to the login page.

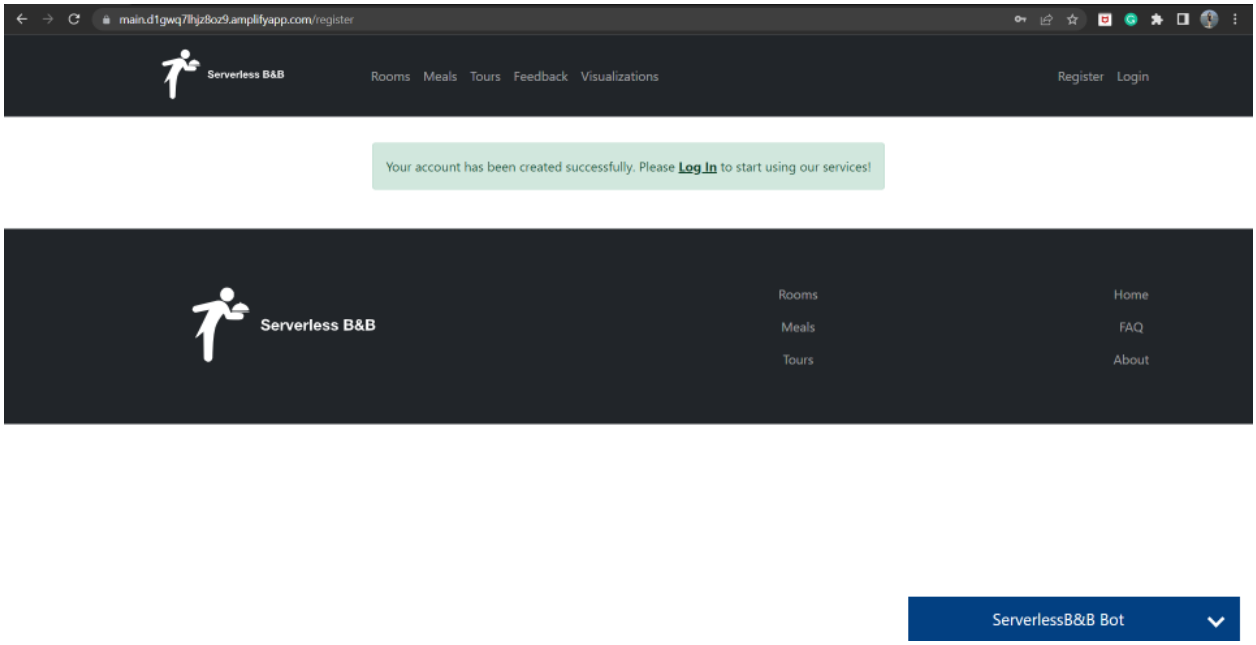

Figure 19: Registration complete.

Pseudo Code

Registration

1. User is asked to input first name, last name, email, and password.
2. These attributes are validated.
3. New user with unconfirmed status is created in the user pool using 'amazon-cognito-identity-js' library.
4. User is sent an OTP to email.
5. User is asked to input the OTP.
6. OTP is validated and user status is confirmed.
7. User is asked to choose one security question from a dropdown and answer it.
8. The question and answer are then stored to firestore collection with the help of a cloud function.
9. User is requested for an integer key between 0 and 25.
10. This cipher key is stored to firestore with the help of a cloud function.
11. Registration process is complete.

Module 2: Authentication Module

For login, user must go through 3 steps. The first step would be the credentials validation followed by 2 factor authentication. Two factor authentication consists of security question answer authentication and Caesar cipher authentication.

Initially, user will be asked to input the email and password. These credentials will then be validated with the help of 'amazon-cognito-identity-js' library.

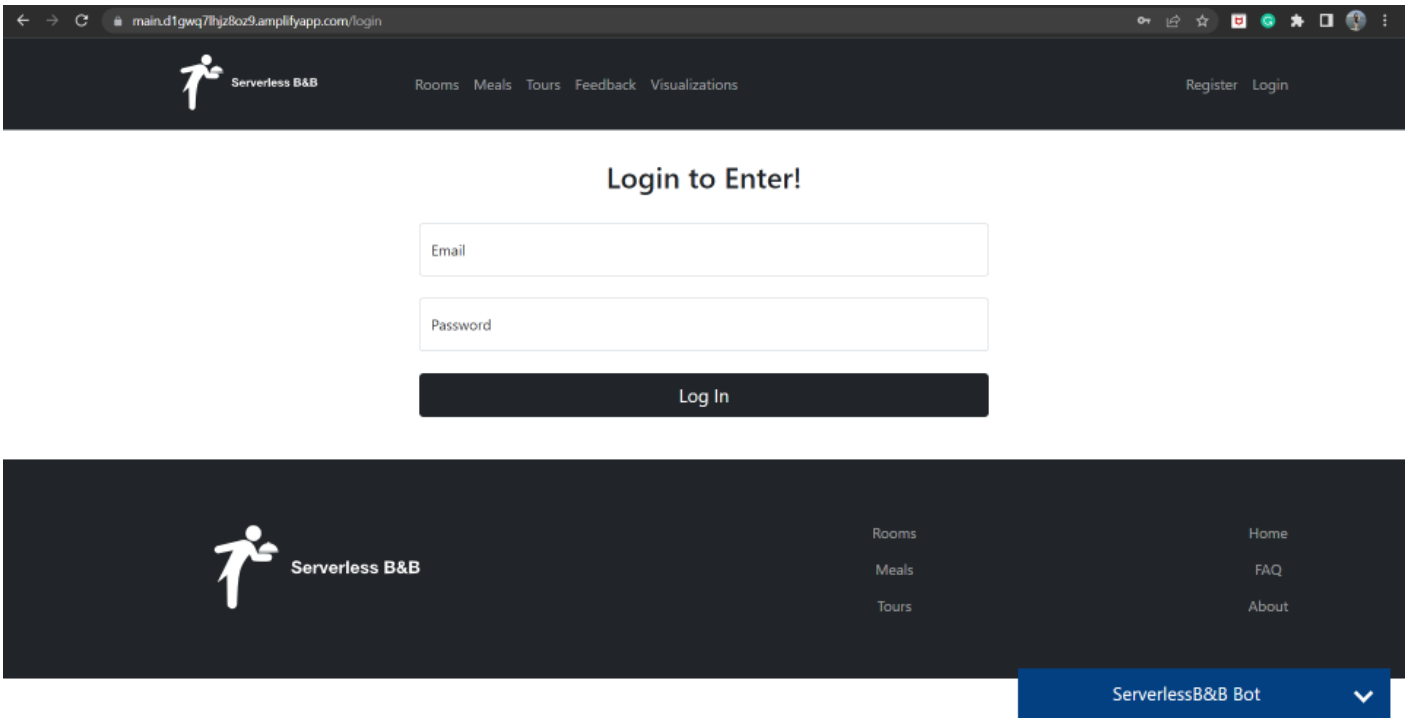

Figure 20: Login form.

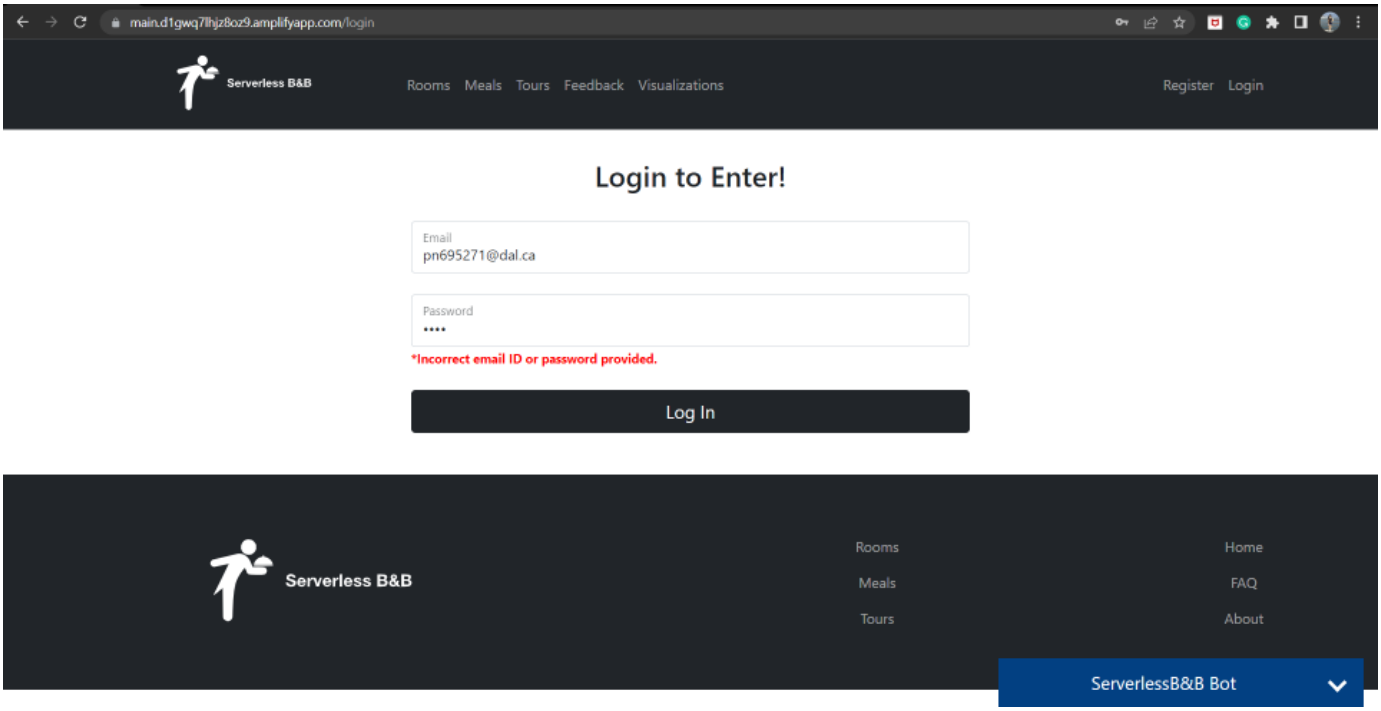

Figure 21: Invalid credentials error.

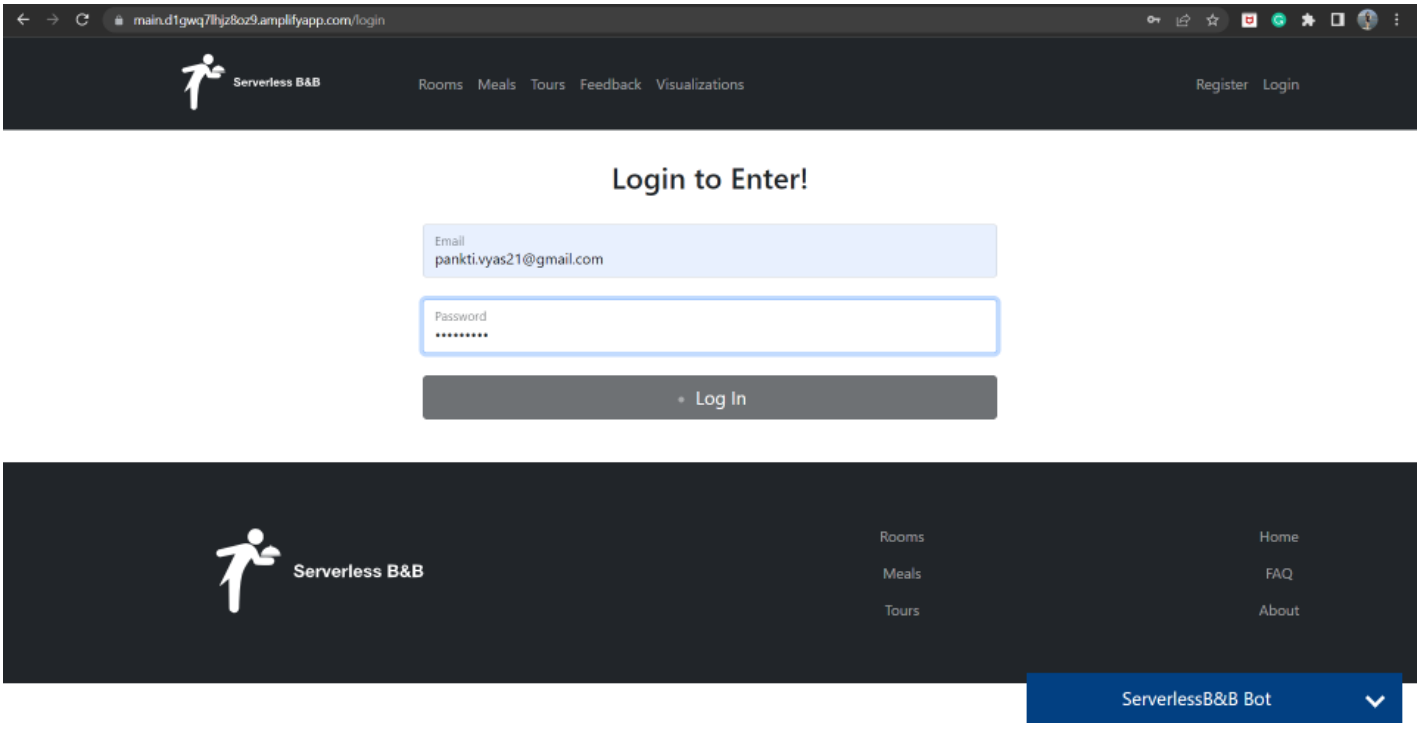

Figure 22: Valid credentials added to Login form.

Now, the security question that user selected while registering needs to be accessed from the 'security-questions' collection in Firestore. This is done by 'get-security-question' cloud function. It gets the user's email in the post request and retrieves the security question from the document with the same email in the 'security-questions' collection and returns the question in the response. This question is then displayed to the user and the answer is requested.

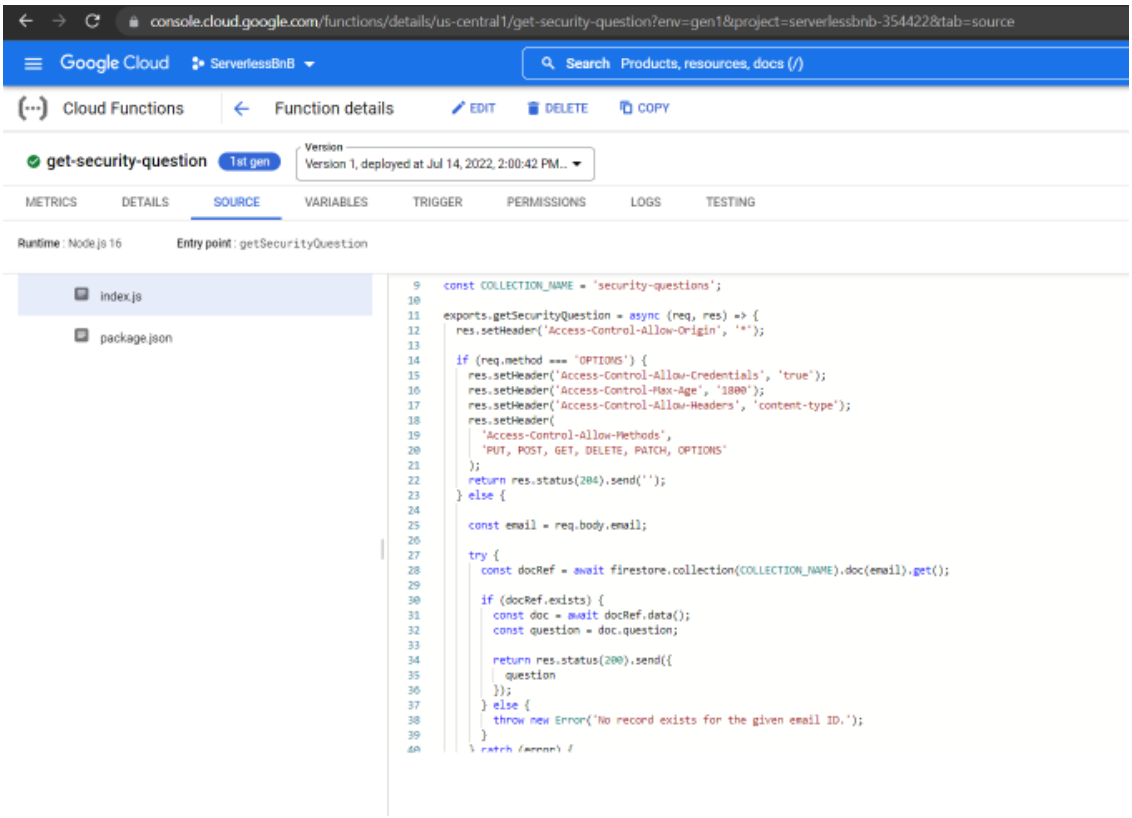

Figure 23: 'get-security-question' cloud function.

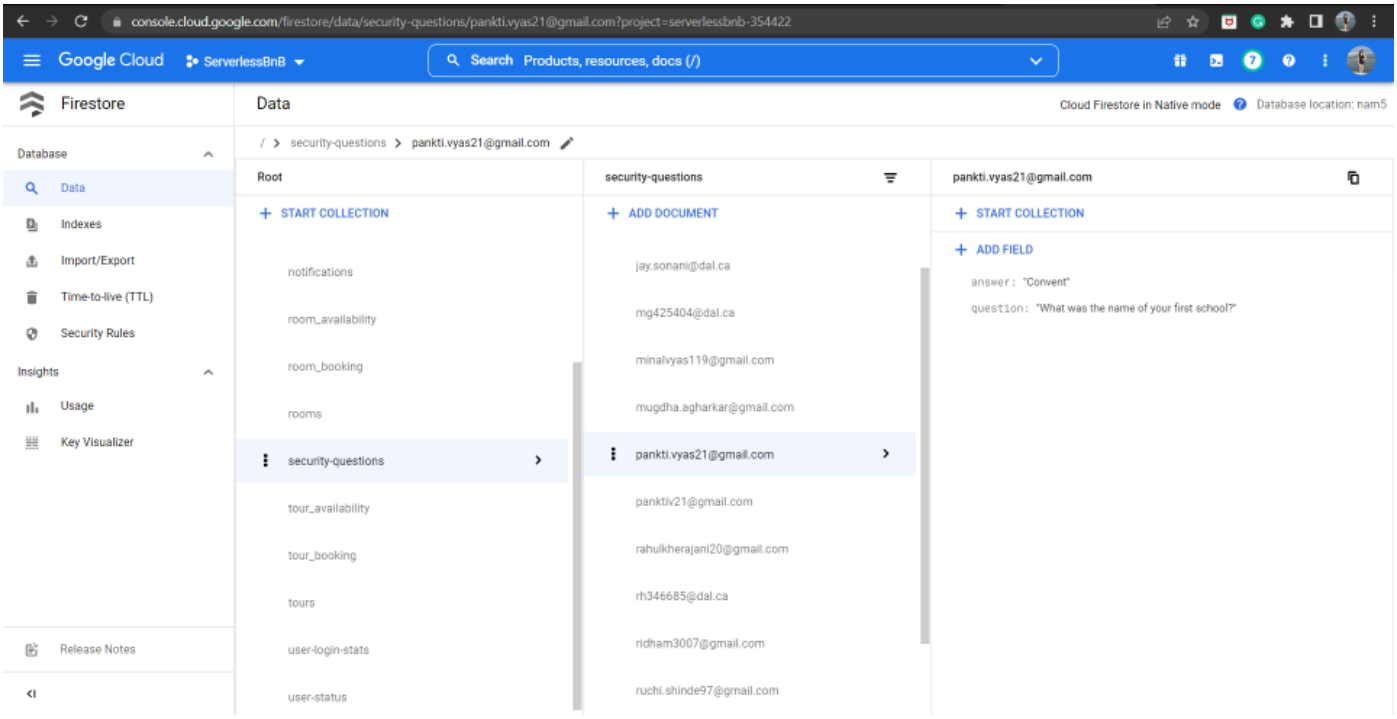

Figure 24: Security question is accessed from 'security-questions' collection.

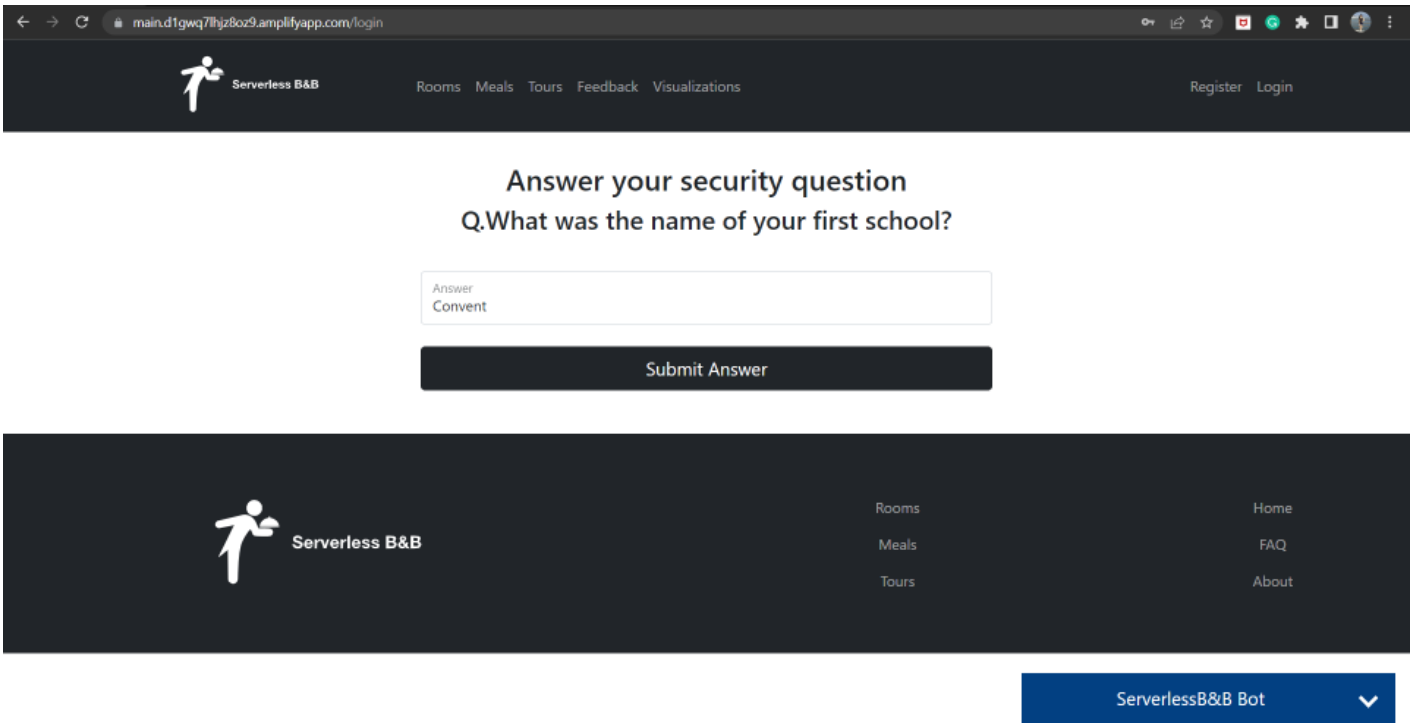

Figure 25: Security question displayed and answer for the security question is required to be entered.

Once the answer to the security question is entered it is passed to the 'validate-security-question-answer' cloud function within a post request body. This answer is then matched with the correct answer stored in the 'security-questions' collection which was stored while registering. If the answer is matched a success response is returned or else an error is thrown.

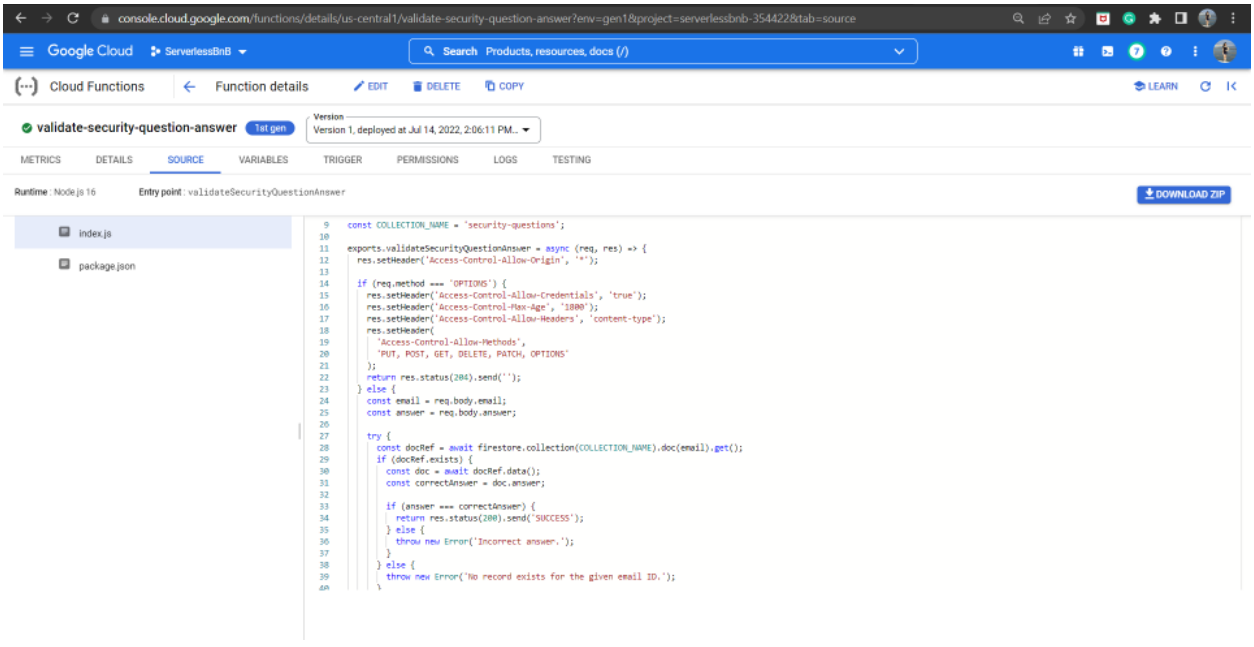

Figure 26: 'validate-security-question-answer' cloud function.

In this final step, the user must use their key to encrypt a given plain text. This plain text is a five-character long alphabetic string randomly generated on the frontend. After entering the cipher text, when user hits the Submit button, three fields will be sent to API hosted on GCP API Gateway. The plain text, the cipher text and the user

id. This API will trigger the Cloud function “caesar-cipher". This cloud function will first of all, retrieve all three fields from the request. Then, using the user id, it will fetch the key for that user from the “caesar_keys” collection on Firebase. Using that key, a correct cipher text is generated from the plain text. This generated cipher text is then compared with the one that user has entered, retrieved from the request. A True or False as a “success” flag is sent back in the response along with the correct generated cipher text.

Testing

Invalid Input

Figure 27 shows the screenshot when the wrong answer entered in the cipher text field

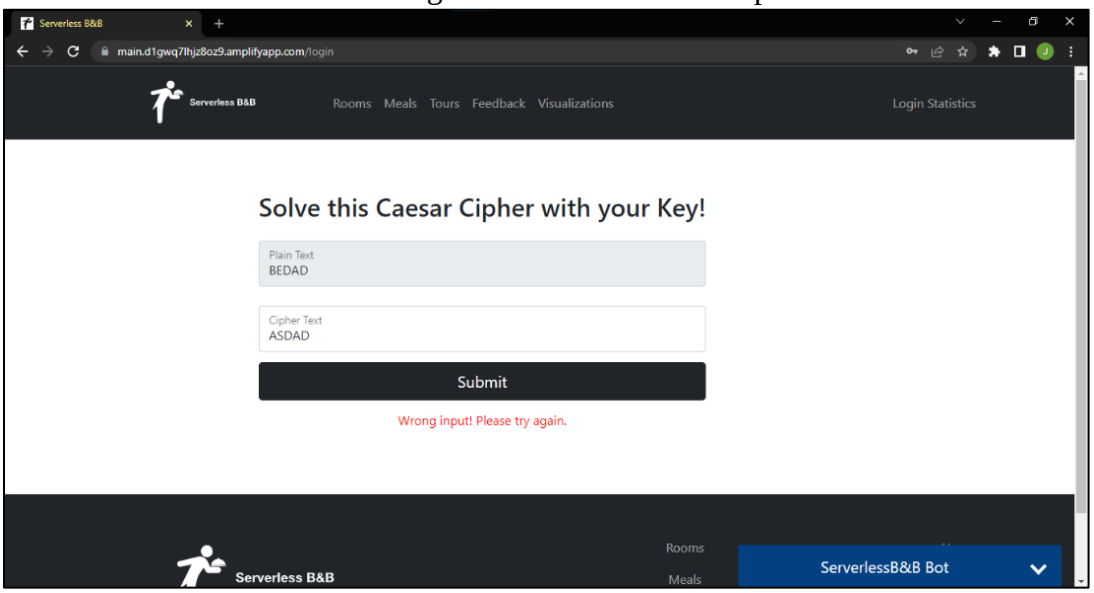

Figure 27: Invalid cipher input.

Figure 28 shows the landing screen upon entering the correct cipher text in the third step of the authentication.

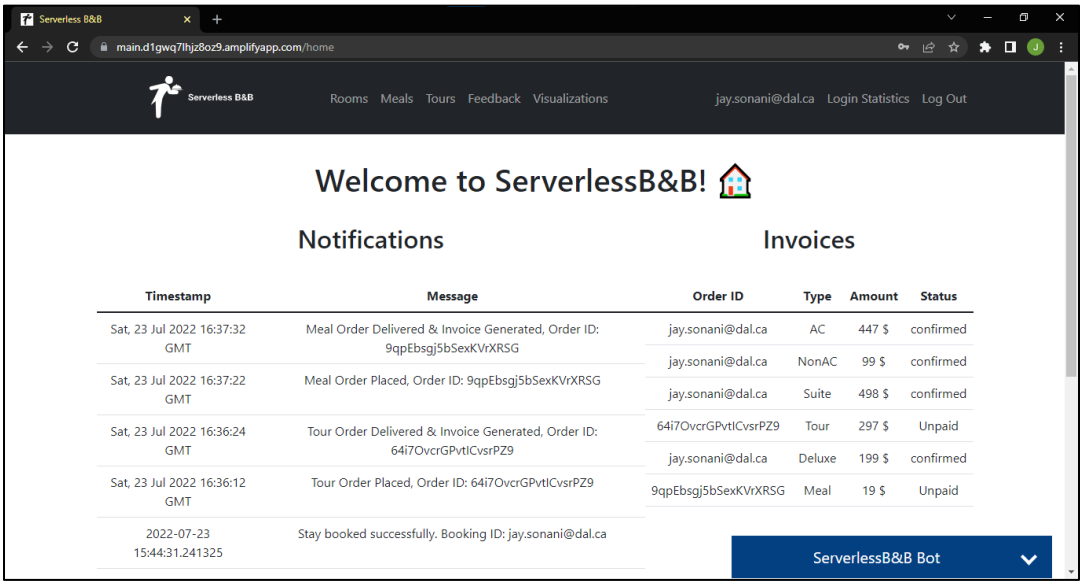

Figure 28: Login successful after 3-step authentication.

To perform this operation, we have created a collection "caesar_keys" in Firebase database. This collection will contain one document per user. The id of this document is user's user id, i.e., email. This document will have one key-value pair as "key" and its value as a number. Figure 29 shows the "caesar_keys" collection and keys stored inside the collection for different users on Firebase.

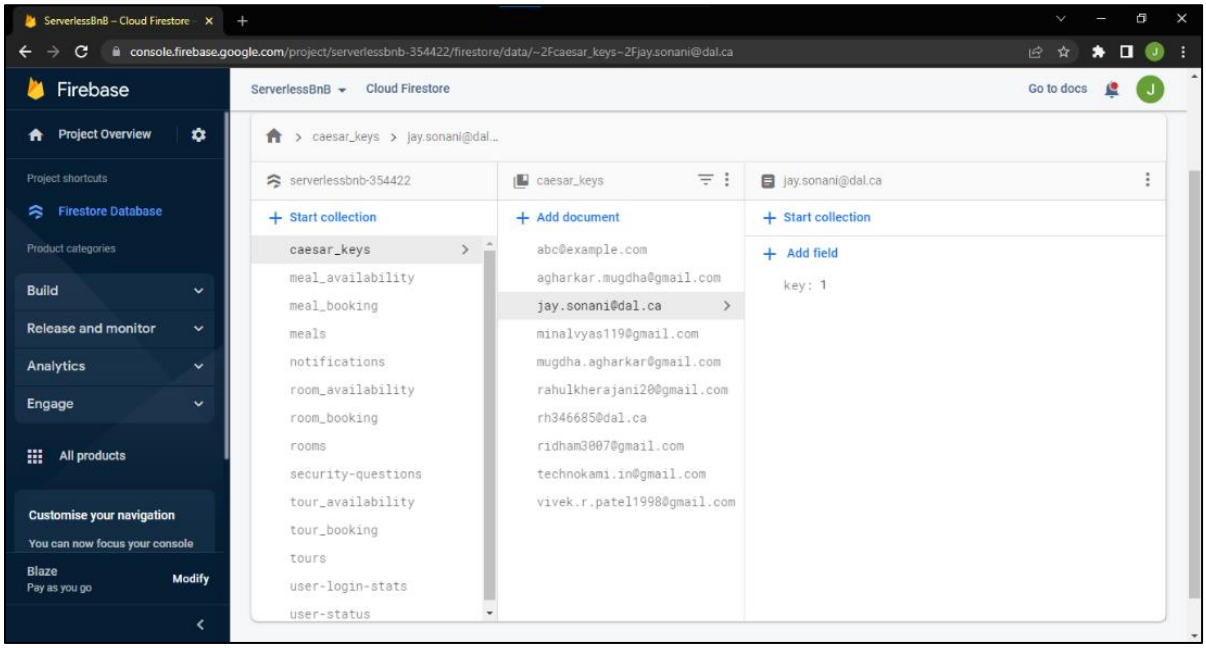

Figure 29: Firebase "caesar_keys" collection.

From the frontend, a POST http request is sent to GCP API Gateway on the end point /caesar-cipher. A cloud function trigger URL with other necessary configurations are mentioned in the config.yaml file for API Gateway. Figure 30 shows the config file containing the caesar-cipher configuration.

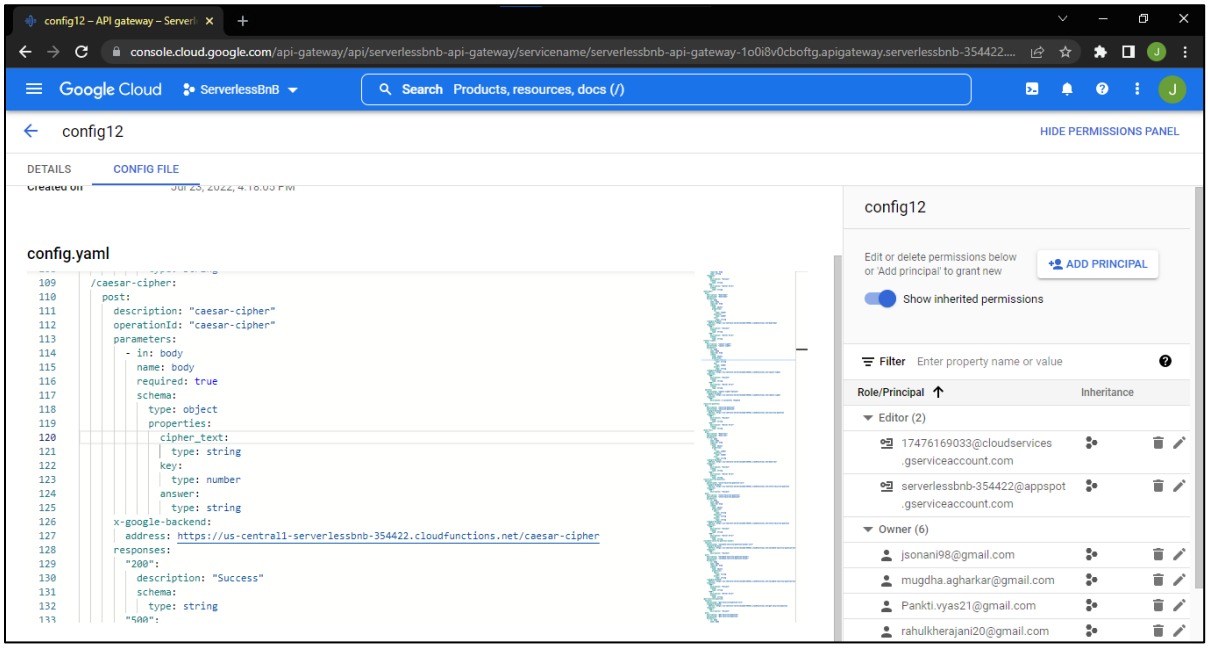

Figure 30: Caesar-cipher configuration for API Gateway.

Using configuration shown in the Figure 30, the caesar-cipher cloud function will be invoked. Figure 31 shows the cloud function caesar-cipher that will be invoked via API Gateway

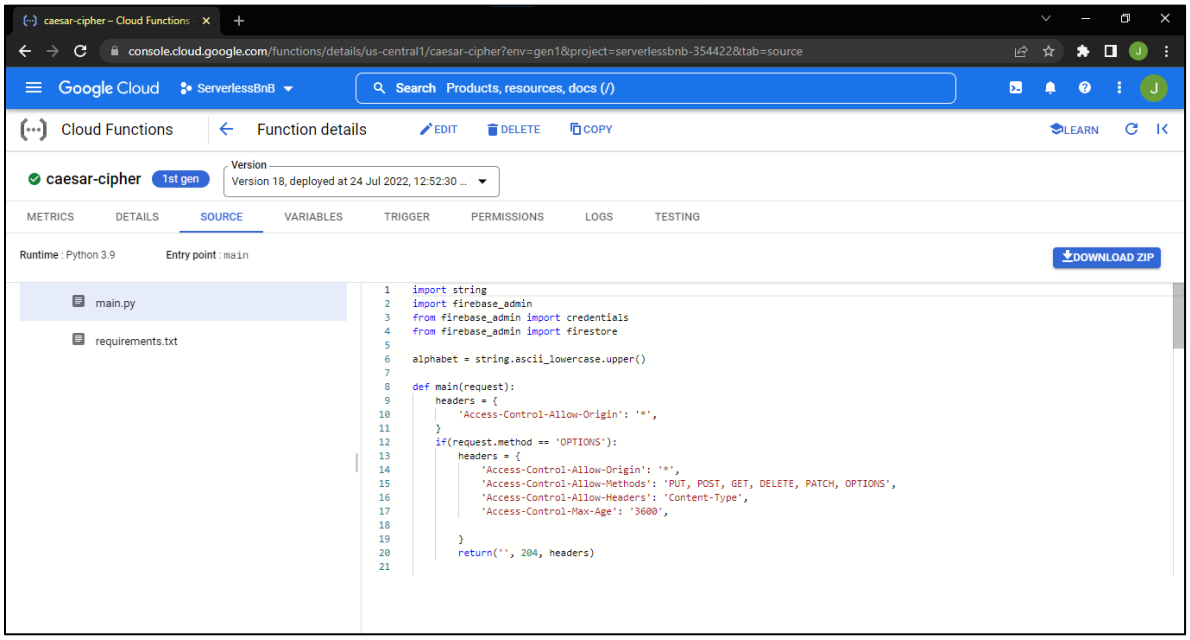

Figure 31: Cloud function "caesar-cipher".

Pseudo Code

- User is prompted with the plain text.
- User uses their key to encrypt the text.
- User enters the encrypted cipher text.
- User hits the submit button.
- Cipher text, plain text and user Id are sent to API.
- Actual cipher text is generated using the key from the database.
- Cipher text user entered is compared with the actual generated cipher text.
- If they match, returned success as True in the response.
- If they don't match, returned success as False in the response.

Once the user is logged in, the email appears on the top right of the navbar. This indicated that the user is authenticated and the session is currently active. On successful login user status is set to 'true' in the 'user-status' collection. This is done using the 'set-user-status' cloud function which gets the email and status as parameters in a post request and updates the status to Firestore.

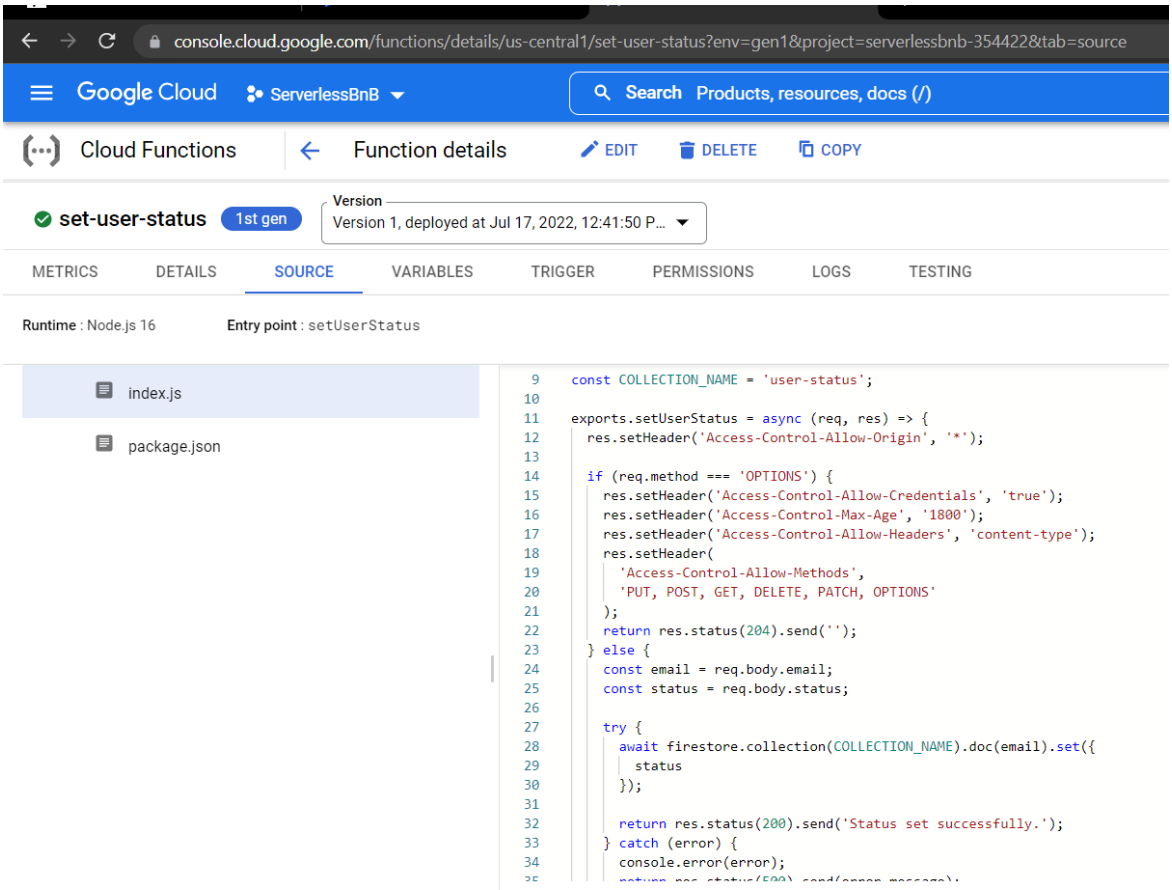

Figure 32: 'set-user-status' collection.

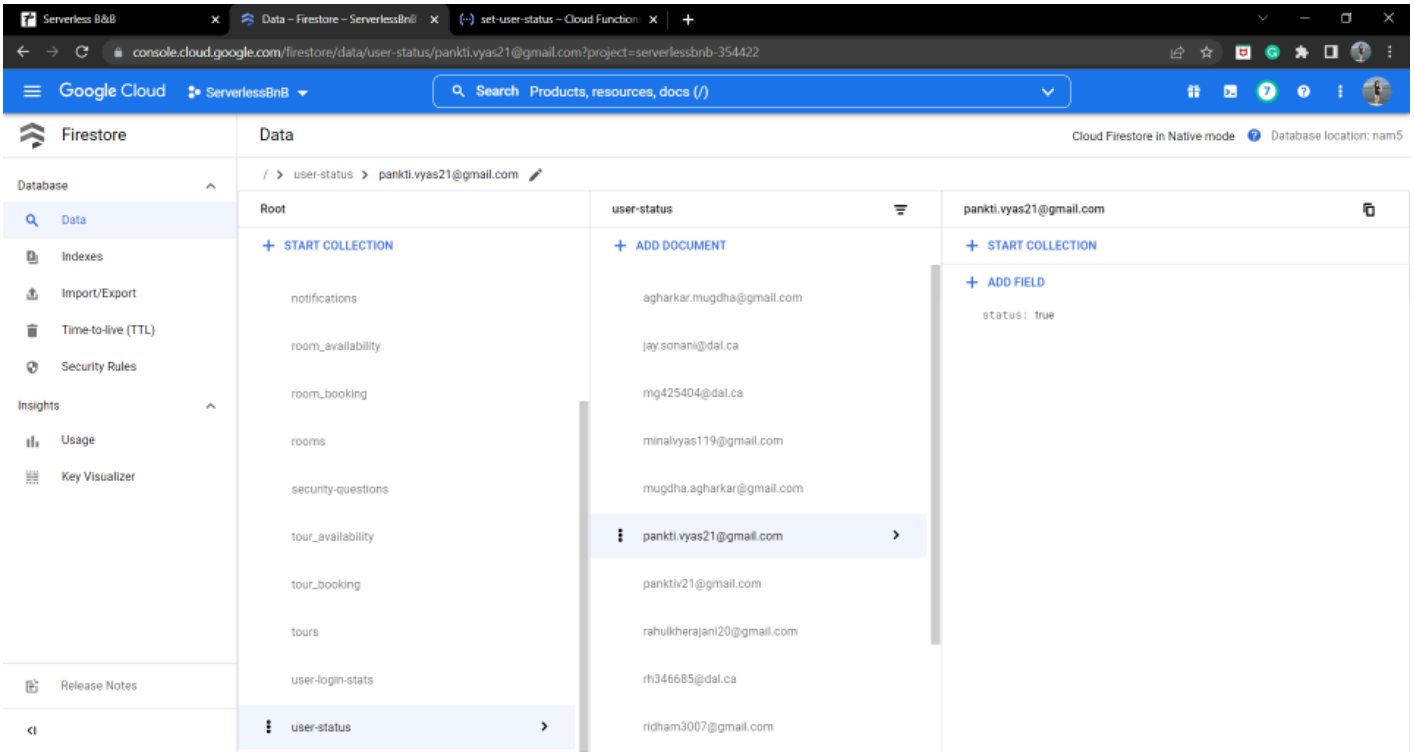

Figure 33: Status set to 'true' currently authenticated and logged in user.

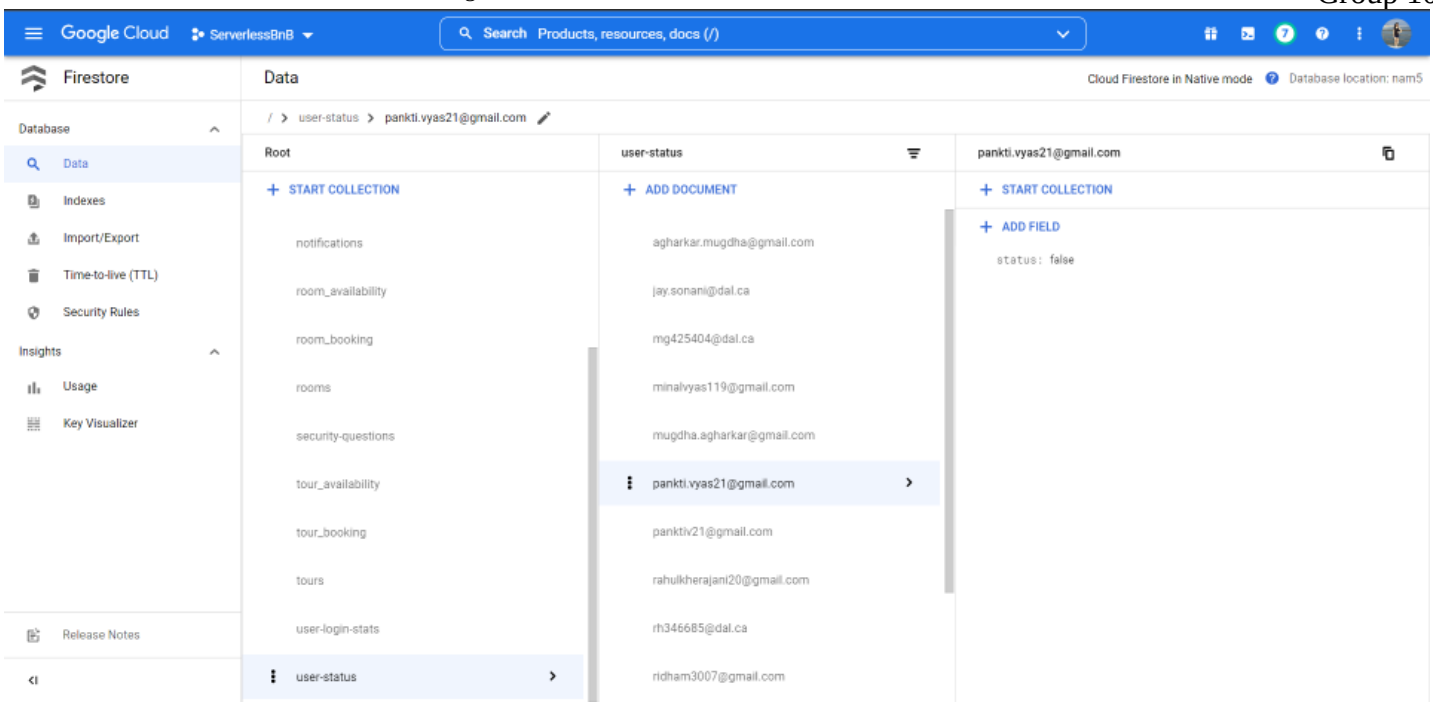

Figure 34: Status set to 'false' after logout.

Pseudo Code

Login

1. User is asked to input email and password.
2. These attributes are validated.
3. Credentials are validated.
4. Security question that user entered while registration is retrieved from the firestore collection with the help of a cloud function.
5. User is asked to answer the question.
6. Answer is validated with the answer stored in firestore 'security-questions' collection by a cloud function 'validate-security-question-answer'.
7. The user is then asked to cipher a random string which is then validated.
8. User is now authenticated and logged in.

Module 3: Online Support Module

The online support module is a chat bot which responds to the user's queries. It provides online virtual assistance for both users, the guest users and the logged in users. The chat bot is implemented using two AWS services that are **AWS Lex** and **AWS Lambda**. For the front end of ServerlessBnB chat bot I have used react library **react-lex-plus** which provides readymade UI for lex. The chat bot provides mainly three functionalities such as basic site navigation, checking room availability and service bookings. The service booking includes services such as room booking, ordering a meal and tour booking. The guest users only have access to basic site navigation and room availability service whereas logged in users have access to all these services. For creating the chat bot, I have used AWS Lex service. The chat bot name is ServerlessBnB. Figures 35 and 36 display the configuration of ServerlessBnB chat bot.

Figure 35: ServerlessBnB chat bot on AWS Lex console

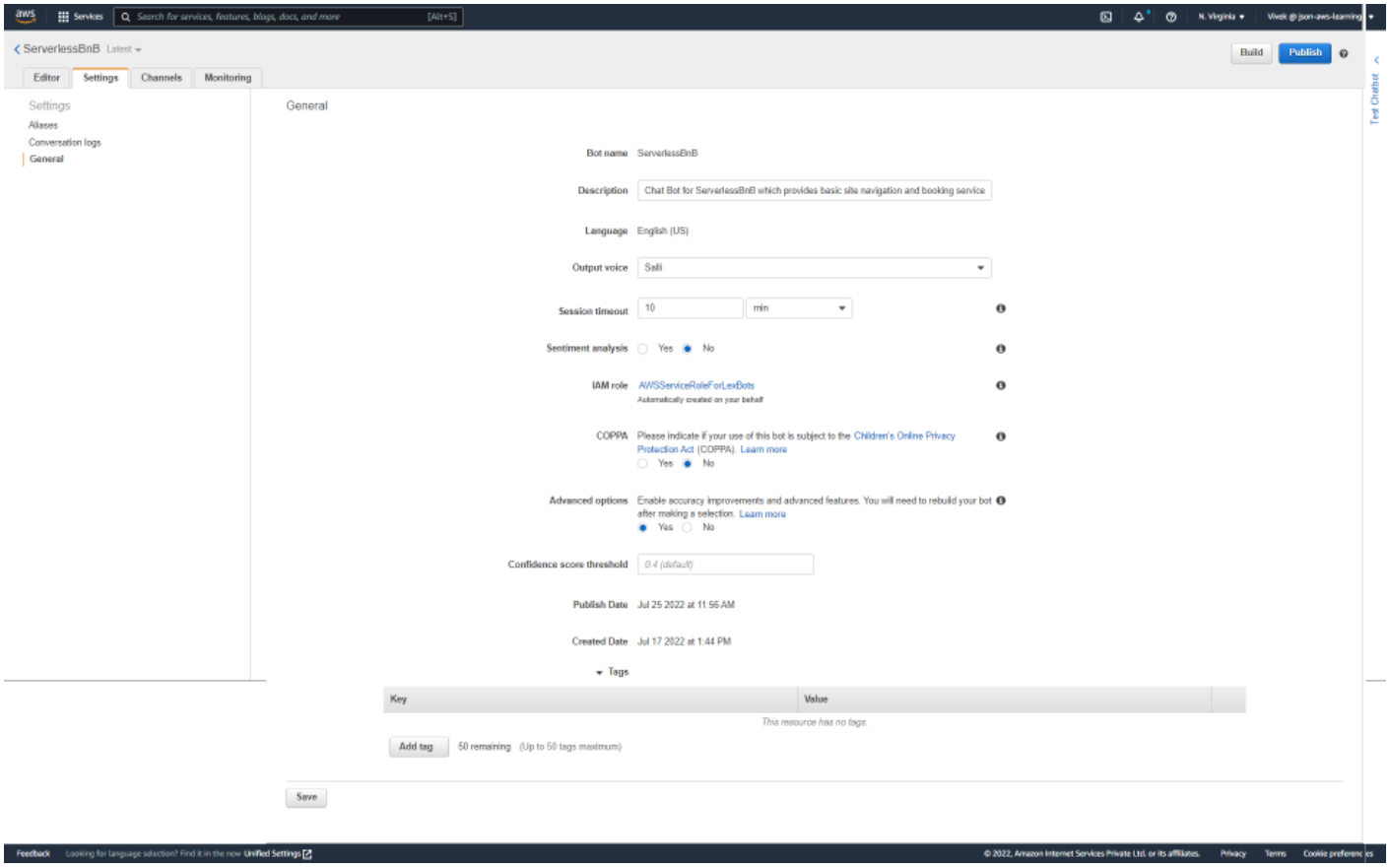

Figure 36: Configuration of ServerlessBnB bot.

As displayed in the Table-2, the chat bot has its internal components such as Intents, Utterances, Slots, and Code Hook. The intent is a specific action or task that the user wishes to complete. I have created total 5 intents for this chat bot. **BookRoom** intent is responsible for room booking, **BookTour** intent handles tour booking, **OrderMeal** intent takes care of meal orders, **RoomAvailability** intent allows to check for available rooms and lastly Navigation intent allows basic site navigation. All these intents are actions that ServelessBnB users want to perform which will be handles automatically by **ServelessBnB** bot using intents. Utterances are textual representations of what a user should type or say to trigger an intent to perform a specific action or task. Slots are the values or pieces of data that the bot intent requires to complete its task. User is usually prompted to provide slot values, or he can provide slot values in utterances itself, e.g., for a OrderMeal intent bot is likely to need slots for meal type and meal quantity. The bot can prompt user to ask for a slot value; for example, "What type of meal you want to order? Our most popular options are: 1. Veg and 2. Non-veg." or user can provide slot value in utterance itself; for example, "I want to order 2 Veg meals"; in this case the slot values meal type and meal quantity will be automatically retrieved by the bot. I have used built-in slot types as well as custom slot types for

fulfilling the intents. The built-in slot types which I have used are **AMAZON.DATE**, **AMAZON.NUMBER** and **AMAZON.AlphaNumeric**. For some of the data items I have used custom slot types such as **RoomTypes**, **TourTypes** and **MealTypes**. These slot values will be used as an enumeration, with the user's value being resolved to the slot value only if it is the same as one of the slot values or a synonym. [9]

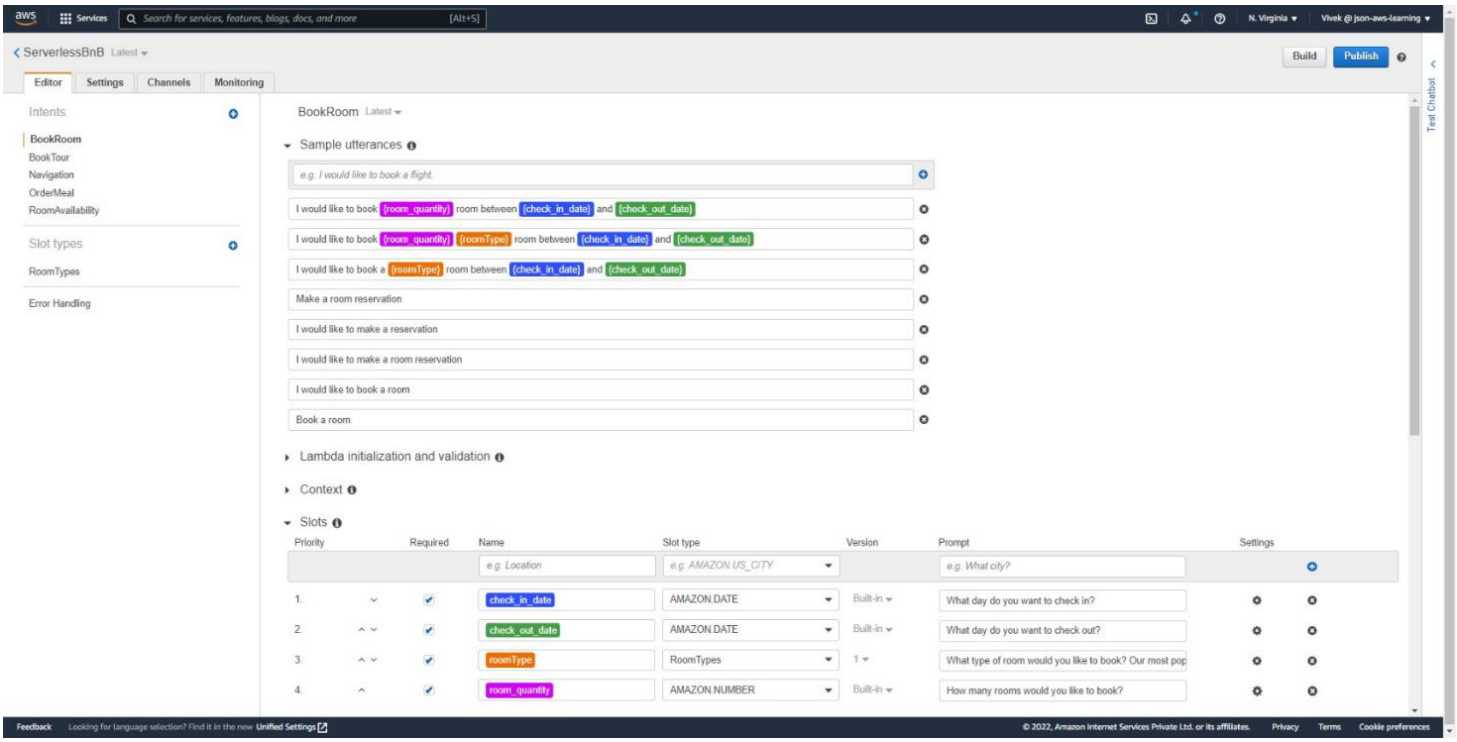

Figure 37: Internal components of AWS Lex ServerlessBnB chat bot.

Table 22: Internal components of ServerlessBnB chat bot.

Intent	Purpose	Slots
BookRoom	For booking a room	Check-in Date, Check-out Date, Room Type and Room Quantity
BookTour	For booking a tour	Tour Type and Number of People
OrderMeal	For ordering a meal	Meal Type and Meal Quantity
RoomAvailibility	For checking room availability	Check-in Date and Check-out Date
Navigation	For website navigation	Webpage

For fulfilment of each intent, I have created a Code Hook which in turn triggers AWS Lambda function which handles the logic of fulfilment of the intent based on its type. For this, I have created **ServerlessBnBLex** lambda function as displayed in the Figure-38. Whenever, user wants to perform particular action he will simply interact with the chat bot by providing sample utterances and based on those utterances particular intent will be called and it will ask for the required slot values needed for the fulfilment of the slot based on the task to the user wants to perform. After taking slot values the intent will then trigger the ServerlessBnBLex lambda function which will receive event from the lex. The event contains information about the intent such as intent name, session attributes and slot values. Based on the name of the intent the function for handling the task associated with the intent will be called and after executing the business logic the response will be returned to the lex to display it to the user. For example, if user wants to order a meal, then the chat bot first takes slot values such as meal type and meal

quantity from the user and then passes all the intent information along with the slot values to the ServerlessBnBLex Lambda function as an event. The intent name will be retrieved from the event. If the intent name matches with the "OrderMeal" then the order_meal () function will be called which carry out necessary, POST request to the **APIGateway** to invoke a **GCP CloudFunction** responsible for database operations for

ordering a meal. Then based on the response of the request the response is created for Lex user and will be returned to the user to get displayed in the chat bot UI.

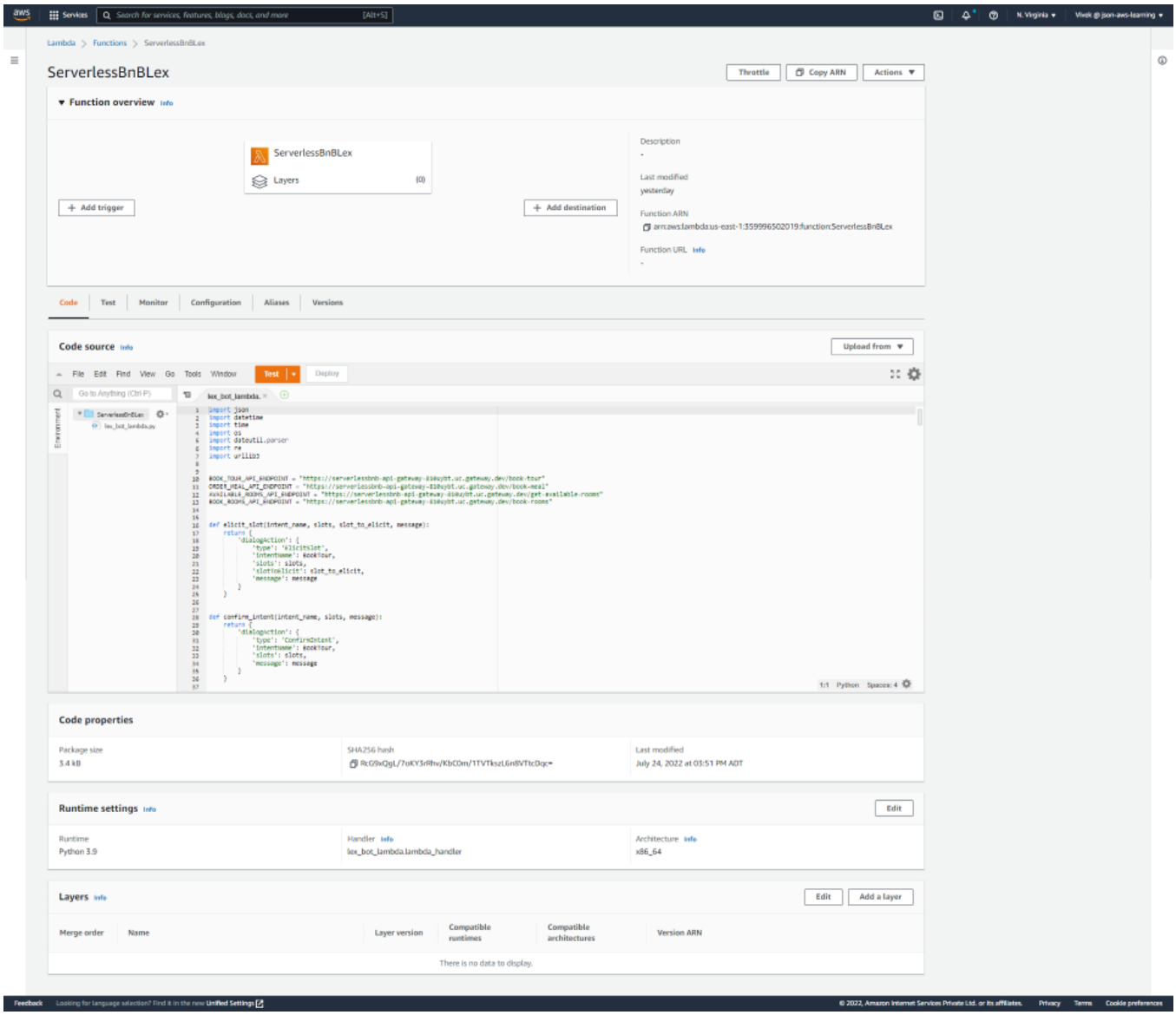

Figure 38: Lambda function that handles fulfilment of Lex Intent.

Flowchart

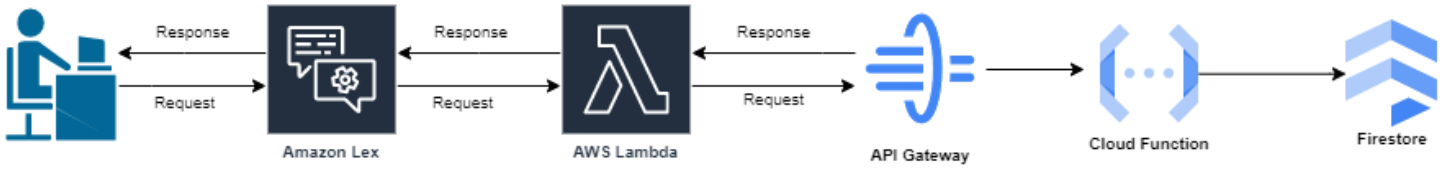

Figure 39: Flowchart of Online Support Module.

Test Cases

- Figure-40 shows the test case for a situation when guest user attempts to use service which are only available for logged in user. Guest user is asked to login first in order to use the service.

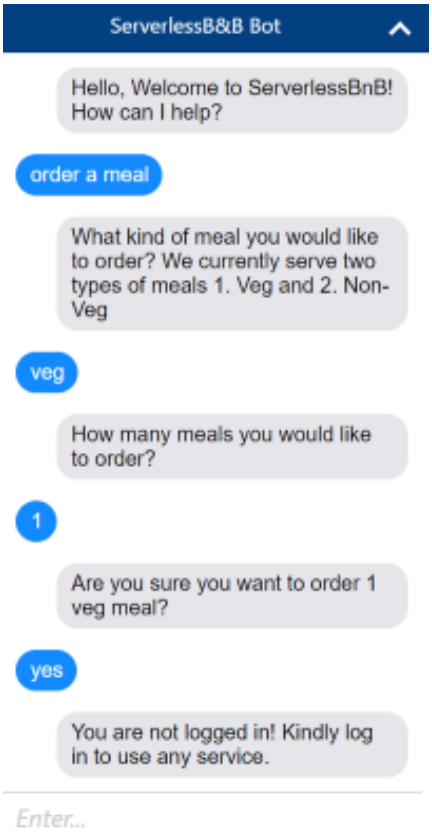

Figure 40: If user is guest user and attempts to use any booking or ordering service through chat bot then he is asked to login first.

- Figure-41 displays screenshot of interaction between user and the ServerlessBnB bot and it involves use of basic site navigation functionality provided through the chat bot.

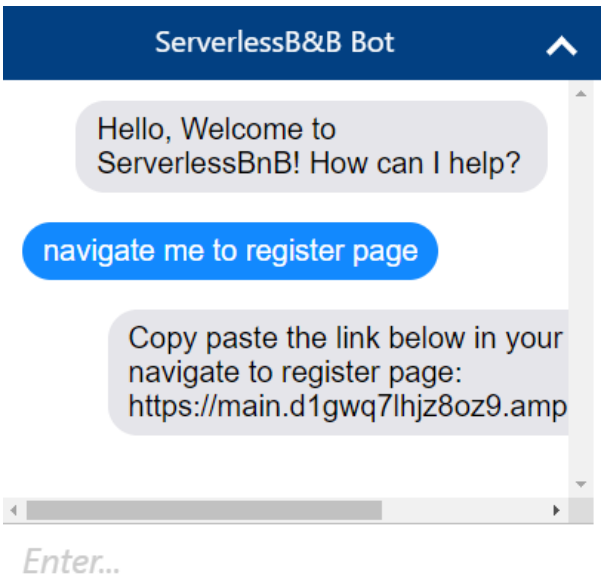

Figure 41: Website navigation request to chat bot and its response.

- Figure-42 displays interaction between the guest user and the chat bot, and it involves room availability checking service. The guest user provides possible check-in and check-out date of his stay to check for the availability of the room. The chat bot displays availability in response if any.

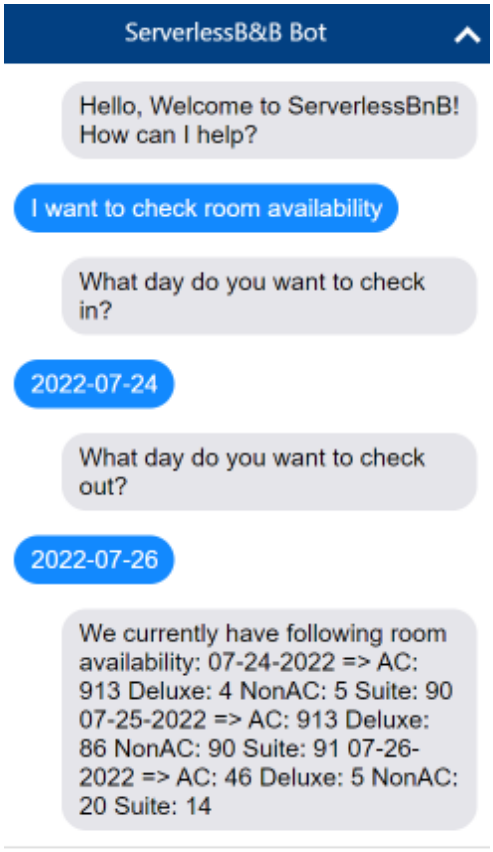

Figure 42: Guest user checking room availability.

- Figure-43 displays logged in user placing successful meal order through the chat bot. If the sufficient meal quantity is available for the requested meal, then the chat bot places the order and in response displays order id to the user.

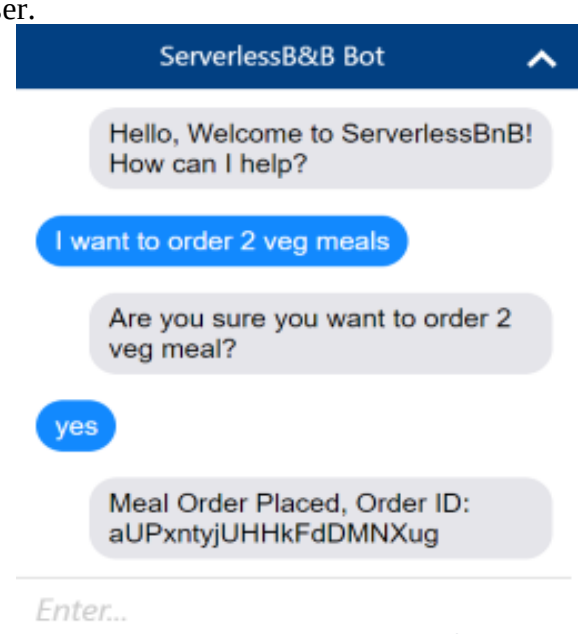

Figure 43: Logged in user placing successful meal order.

- Figure-44 displays response of Lex chat bot in case when logged in user attempts to order excessive quantity of any meal type.

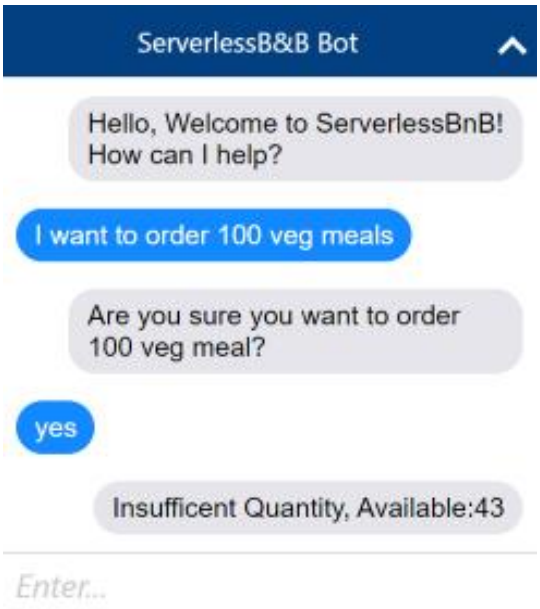

Figure 44: Logged in user booking for excessive quantity for meal order.

- Figure-45 displays response from chat bot when user denied the meal order confirmation. It can be seen that the bot prompts user to reenter the slot value if user provides invalid value for the slot.

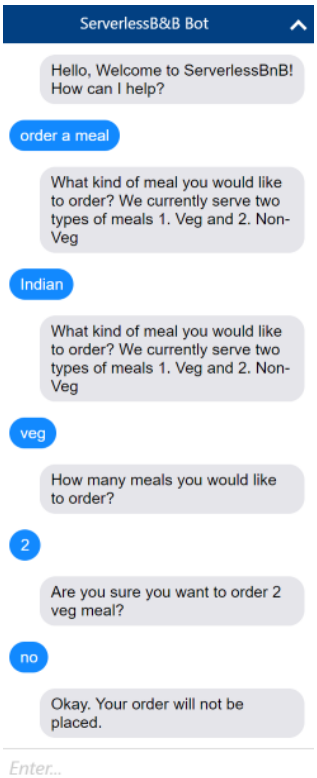

Figure 45: Chat bot response when user denied confirmation of meal order.

- Figure-46 displays user's interaction with chat bot for booking a tour. The bot response with Order ID if sufficient slots are available for a requested tour type otherwise it will respond with available slots for the requested tour types as displayed in Figure-47.

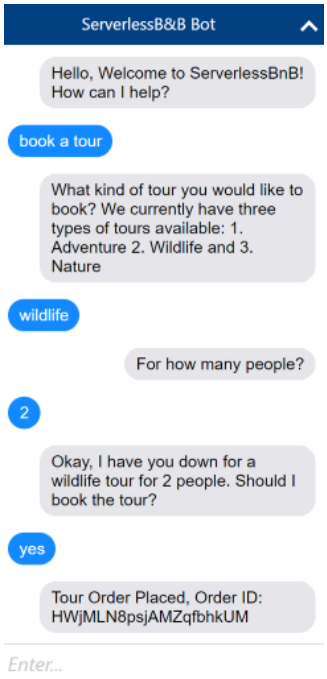

Figure 46: User and chat bot interaction during successful tour booking.

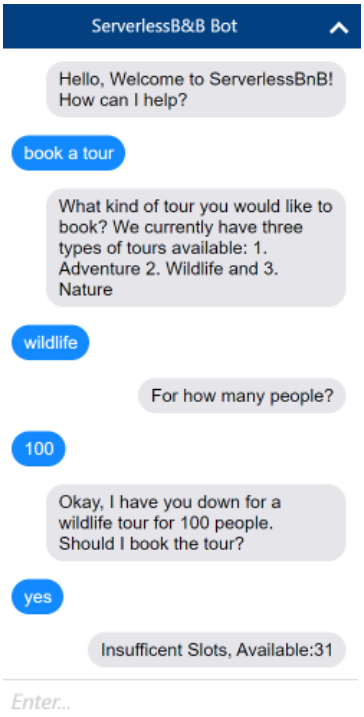

Figure 47: Chat bot response for excessive number of slots requested by user for tour.

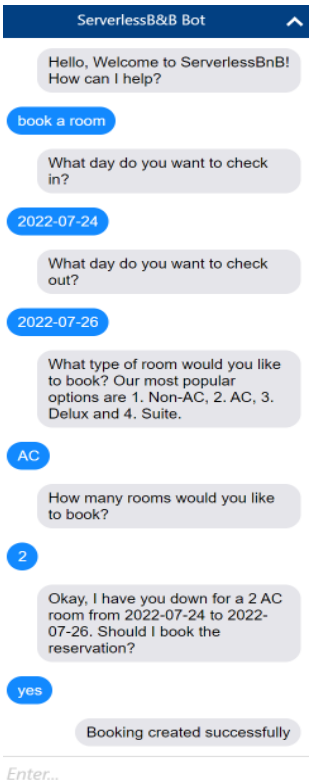

Figure 48: Chat bot booking rooms when user provides enough rooms for booking.

- Figure-48 displays interaction between user and the chat bot for room booking service. Chat bot responds with booking successful message when number of rooms requested by a user are available for user's check-in and check-out dates otherwise it displays rooms not available message as displayed in Figure-49.

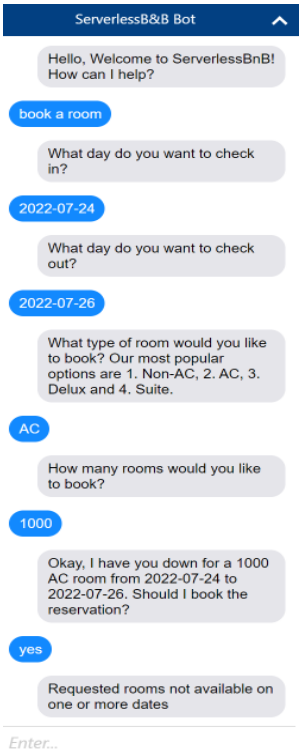

Figure 49: Chat bot response when the user requests excessive quantity of rooms during the booking.

Pseudo Code

Site Navigation

1. User provides utterance to chat bot; for example: “Navigate me to registration page”.
2. Based on the utterance the chat bot calls Navigation intent and passes the intent details to the SeverlessBnBLex lambda.
3. The lambda function retrieved intent details from the event and calls navigation method if the intent name matches.
4. The navigation method maps webpage with its URL using get_navigation_link() method and return the response to the user along with requested navigation URL.

Room Availability

1. User provides utterance to chat bot; for example: “I want to check room availability”.
2. Based on the utterance the chat bot calls RoomAvailability intent, the intent prompts user to provide slot values required for the fulfilment of the intent and passes the intent details to the triggered ServerlessBnBLex lambda.
3. The lambda function then retrieves intent details from the event and calls get_available_rooms() method.
4. The get_available_rooms() method gets the slot values such as check-in date and check-out date from the intent details.
5. Preprocesses the date values and makes a POST request to the API Endpoint that checks room availability.
6. If rooms are available between the provided dates, then the response is return to the user though lex and it contains available rooms of all types for those dates.
7. Otherwise, the lambda function return no rooms available message to the lex to display it to the user.

Order Meal

1. User provides utterance to chat bot; for example: “I want to order a meal”.
2. Based on the utterance the chat bot calls OrderMeal intent, the intent prompts user to provide slot values required for the fulfilment of the intent and passes the intent details to the triggered ServelessBnBLex lambda.
3. The lambda function retrieves session attributes of the logged in user from the intent details and if the session attribute is present then processed further, otherwise it sends a response to the user through lex asking him to login to use calling any booking service.
4. The lambda function then retrieves intent details from event and calls order_meal() function.
5. The order_meal() function gets the slot values from intent details such as Meal Type and Meal Quantity and makes a POST request to the API Endpoint which handles meal orders.
6. If enough quantity is available for the requested meal, then response is return by the lambda to user containing the order successful message and the order ID.
7. Otherwise, the lambda function returns Insufficient quantity available and available quantity of requested meal to the user through Lex.

Book Tour

1. User provides utterance to chat bot; for example: “I want to book a tour”.
2. Based on the utterance the chat bot calls BookTour intent, the intent prompts user to provide slot values required for the fulfilment of the intent and passes the intent details to the triggered ServelessBnBLex lambda.
3. The lambda function retrieves session attributes of the logged in user from the intent details and if the session attribute is present then processed further, otherwise it sends a response to the user through lex asking him to login to use calling any booking service.
4. The lambda function then retrieves intent details from event and calls book_tour() function.
5. The book_tour() function gets the slot values from intent details such as Tour Type and Number of People and makes a POST request to the API Endpoint which handles tour booking.
6. If enough slots are available for the requested tour type, then response is return by the lambda to user containing the Tour booking successful message and the order ID.
7. Otherwise, the lambda function returns Insufficient slots available and available quantity of requested tour to the user through Lex.

Book Room

1. User provides utterance to chat bot; for example: “I want to book a room”.
2. Based on the utterance the chat bot calls BookRoom intent, the intent prompts user to provide slot values required for the fulfilment of the intent and passes the intent details to the triggered ServelessBnBLex lambda.
3. The lambda function retrieves session attributes of the logged in user from the intent details and if the session attribute is present then processed further, otherwise it sends a response to the user through lex asking him to login to use calling any booking service.
4. The lambda function then retrieves intent details from event and calls book_room() function.
5. The book_room() function gets the slot values from intent details such as Room Type, Check-in date, Check-out date and Number of Rooms and makes a POST request to the API Endpoint which handles room booking.
6. If enough rooms are available for the requested room type between provided check-in and check-out date, then response is return by the lambda to user containing the Room booking successful message.
7. Otherwise, the lambda function returns Insufficient rooms available to the user through Lex.

Module 4: Message Passing Module

In this module, Hotel, Kitchen, Tour & Notification microservices communicate with each other. The communication is asynchronous i.e., the services interacting with each other need not be synchronously sending requests and responses as in the case of REST APIs, rather we make use of **Google Pub/Sub** topics which transfer data from publishers to subscribers, with latencies of order of 100 milliseconds [4]. Pub/Sub messaging enables the use of event driven architectures which are horizontally scalable.

In our Serverless B&B we have used **Google Cloud Functions** & **Google Pub/Sub** to enable service to service communication among microservices. We have also used **Firebase Cloud Messaging** to enable service to user communication.

Following scenarios have been considered:

1. **Meal Order (Kitchen):**

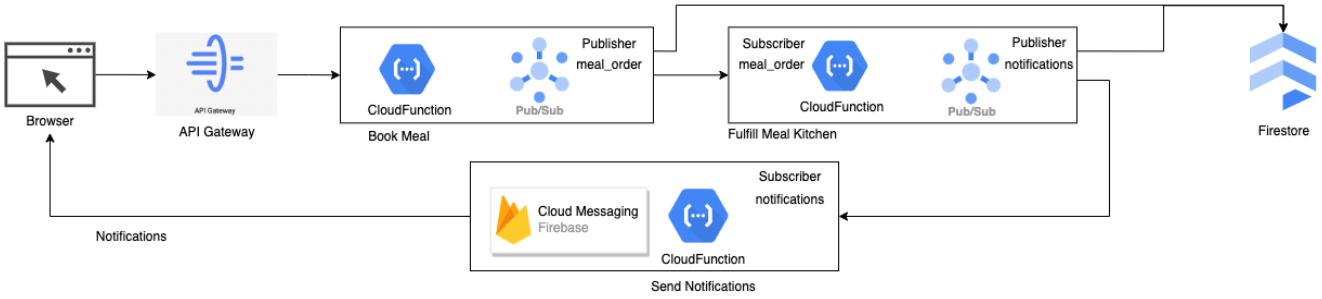

Figure 50: Flow chart for Meal Booking using Google Pub/Sub.

Customer places a meal order from the frontend. Meal Order is received as a HTTP POST request to API gateway which executes a cloud function "book-meal". This cloud function acts as a hotel service which creates a meal order in database and communicates the order details through a pub/sub topic "meal_order". A message published on "meal_order" topic triggers a new cloud function "fulfill-meal-kitchen". This cloud function acts as kitchen service which receives order details, fulfills that order in the database, creates invoice for that order in database and publish a notification message to a pub/sub topic "notifications". A message published on "notifications" topic triggers a new cloud function "send-notification". This cloud function acts a notification service which receives a notification message and extracts the user details from it. Then it retrieves the corresponding token for that user from firestore and sends an order fulfillment notification to the frontend app using firebase cloud messaging.

Pub/Sub Topics:

a. **meal_order**

Subscription: gcf-fulfill-meal-kitchen-us-central1-meal_order

This topic is used for service-to-service communication between hotel and kitchen service. It receives food order from "book-meal" cloud function and triggers a "fulfill-meal-kitchen" cloud function.

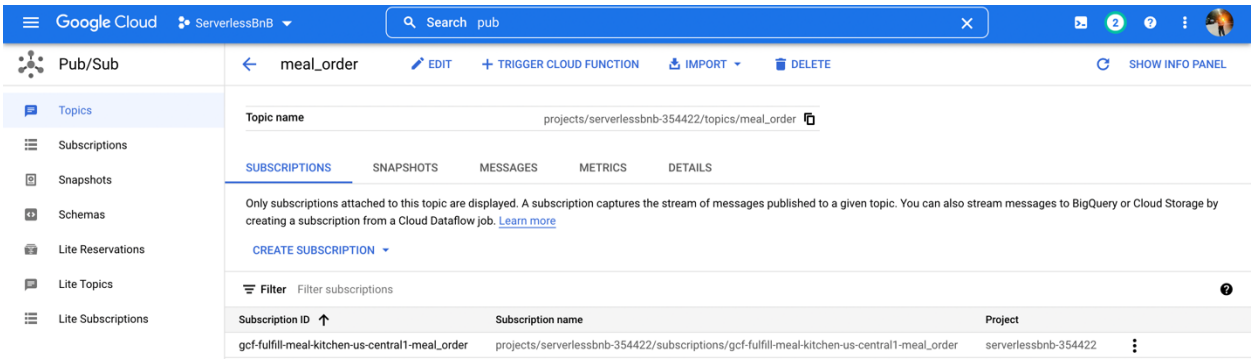

Figure 51: "meal_order" pub/sub topic created for meal order.

b. Notifications

Subscription: gcf-send-notifications-us-central1-notifications

This topic is used for service-to-service communication between kitchen and notification service. It receives notification message from "fulfill-meal-kitchen" cloud function and triggers a "send-notifications" cloud function.

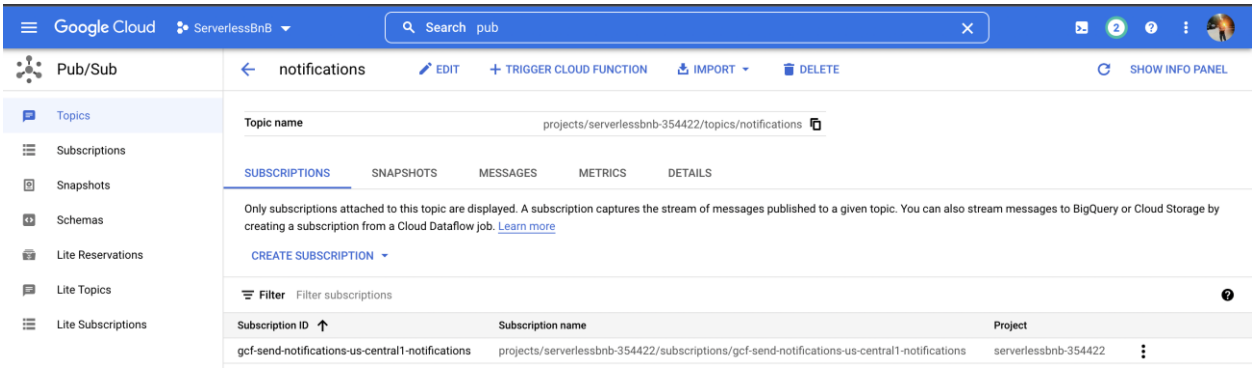

Figure 52: "notifications" pub/sub topic created for sending notifications.

Cloud Functions:

a. book-meal

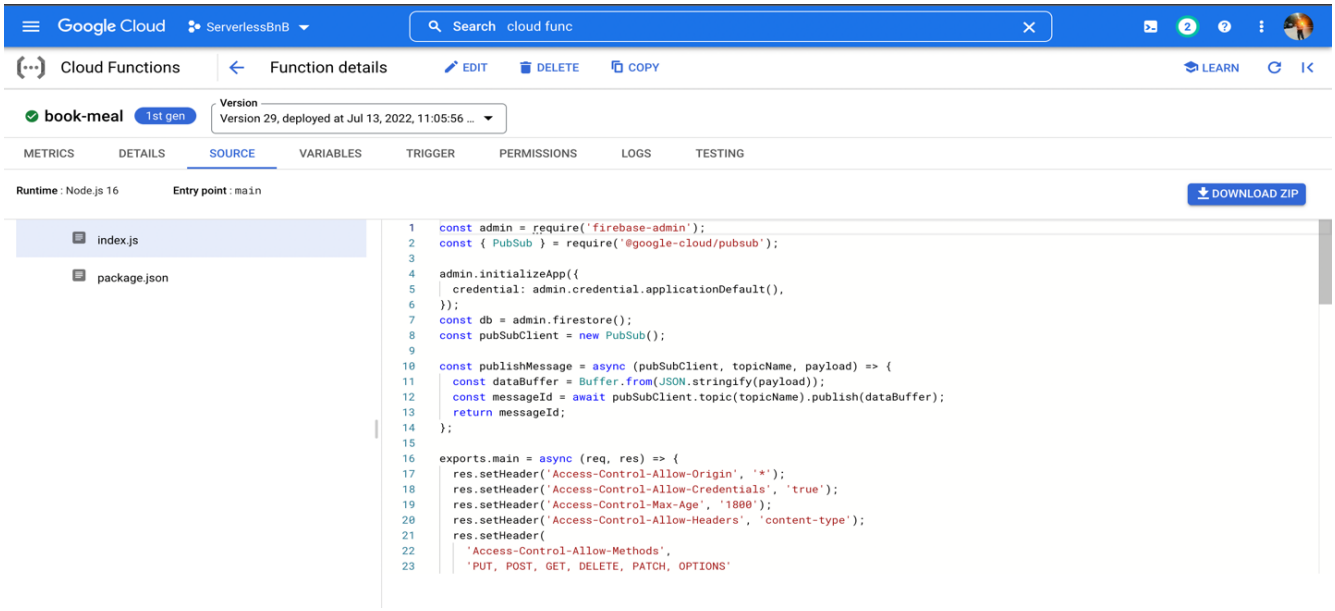

Figure 53: "book-meal" cloud function created to receive meal orders.

Pseudo Code:

- receives meal order details from frontend
- checks availability of meal type for current day.
- If available, creates a meal order in database and sends confirmation; If availability is not enough; it sends an error message back.
- If the order is created, it publishes a message with order details to "meal_order" pub sub topic. A message published to "meal_order" topic triggers "fulfill-meal-kitchen" cloud function.

b. fulfill-meal-kitchen

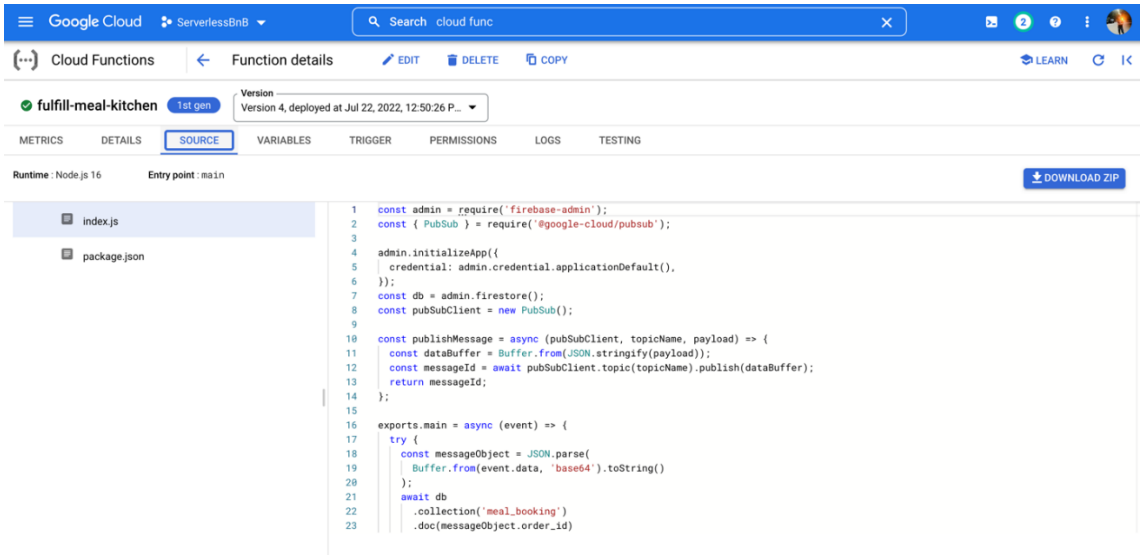

Figure 54: "fulfill-meal-kitchen" cloud function created to fulfill orders.

Pseudo Code:

- receives meal order details from "meal_order" topic.
- Changes the status of order from "Placed" to "Delivered" status.
- Creates an invoice for that meal order.
- Publishes a notification message to "notifications" topic which triggers a cloud function "send-notification"

c. send-notifications

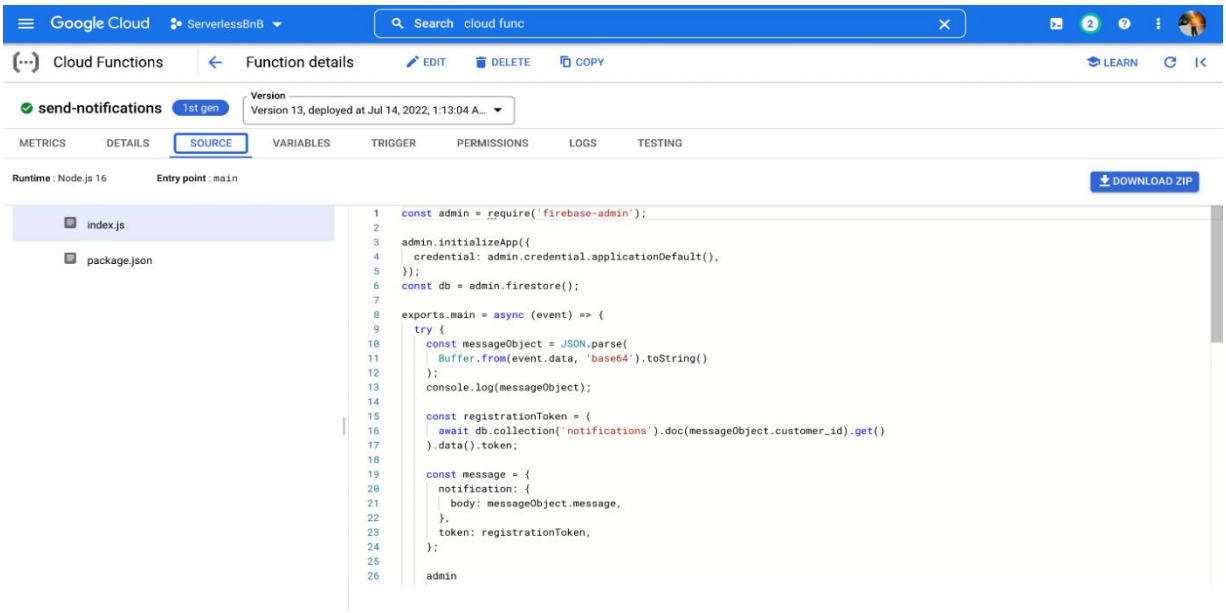

Figure 55: "send-notifications" cloud function created to send notifications to frontend.

Pseudo Code:

- receives notification details from "notification" pub sub topic.
- Retrieves customer id from the message and then retrieves token for the corresponding customer.
- Transmits an order confirmation message to frontend using firebase cloud messaging.

2. Tour Order (Tour Operator):

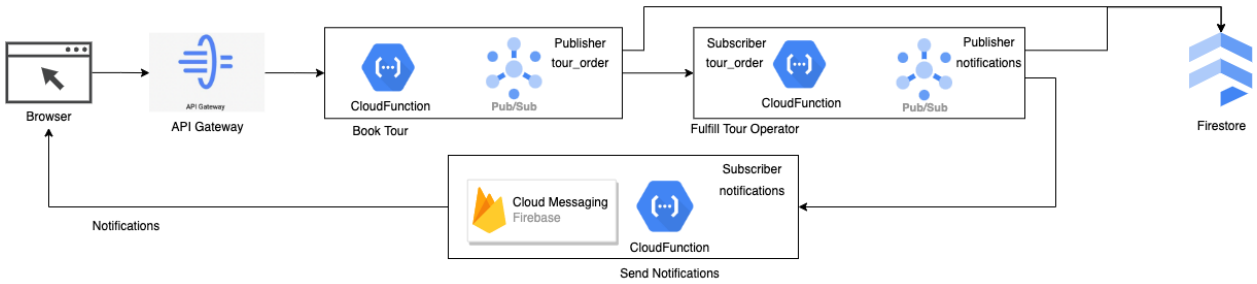

Figure 56: Flow chart for tour booking using Google Pub/Sub.

Customer places a tour order from the frontend. Tour Order is received as a HTTP POST request to API gateway which executes a cloud function "book-tour". This cloud function acts as a hotel service which creates a tour order in database and communicates the order details through a pub/sub topic "tour_order". A message published on "tour_order" topic triggers a new cloud function "fulfill-tour-operator". This cloud function acts as tour service which receives order details, fulfills that order in the database, creates invoice for that order in database and publishes a notification message to a pub/sub topic "notifications". A message published on "notifications" topic triggers a new cloud function "send-notification". This cloud function acts a notification service which receives a notification message and extracts the user details from it. Then it retrieves the corresponding token for that user from firestore and sends an order fulfillment notification to the frontend app using firebase cloud messaging.

Pub/Sub Topics:

a. tour_order

Subscription: gcf-fulfill-tour-operator-us-central1-tour_order

This topic is used for service-to-service communication between hotel and tour service. It receives tour order from "book-tour" cloud function and triggers a "fulfill-tour-operator" cloud function.

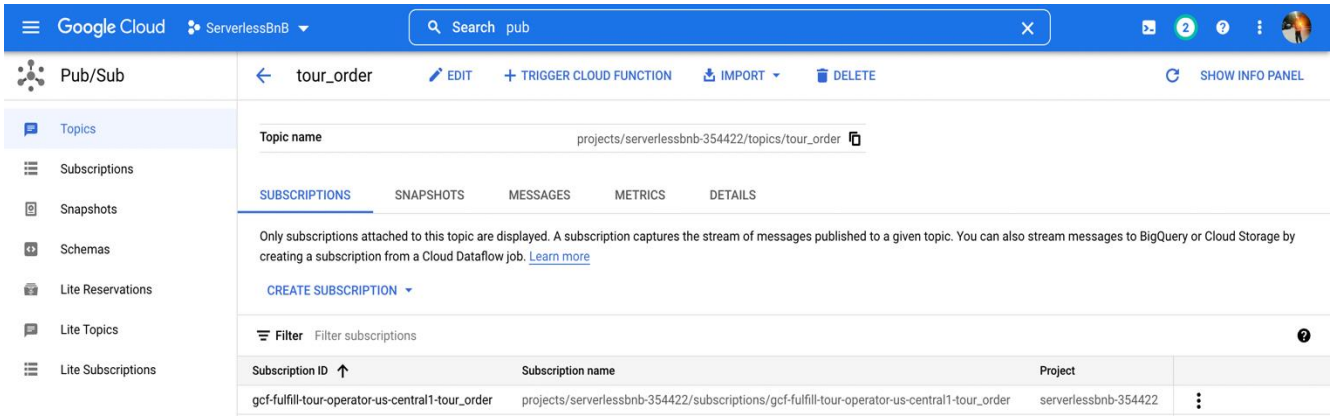

Figure 57: "tour_order" topic created to communicate tour orders.

c. notifications

Subscription: gcf-send-notifications-us-central1-notifications

This topic is used for service-to-service communication between tour and notification service. It receives notification message from "fulfill-tour-operator" cloud function and triggers a "send-notifications" cloud function.

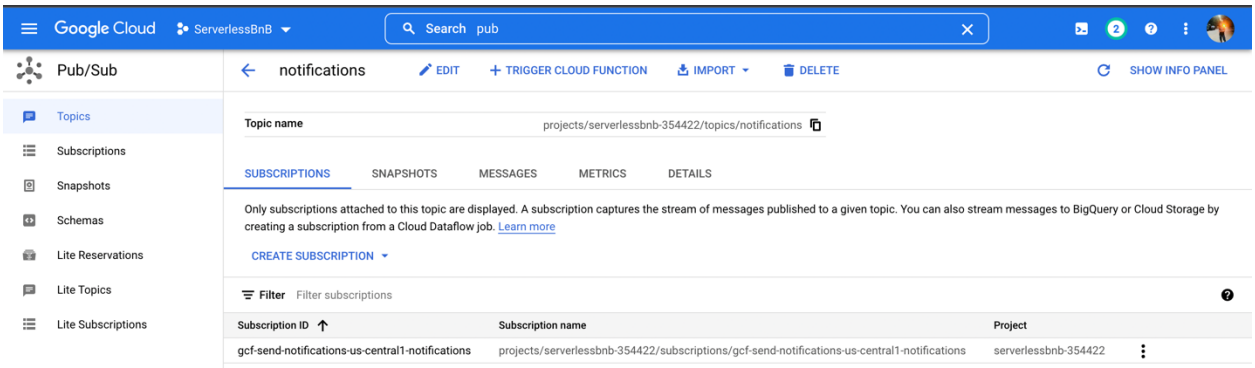

Figure 58: "notifications" topic created for sending notifications.

Cloud Functions:

a. book-tour

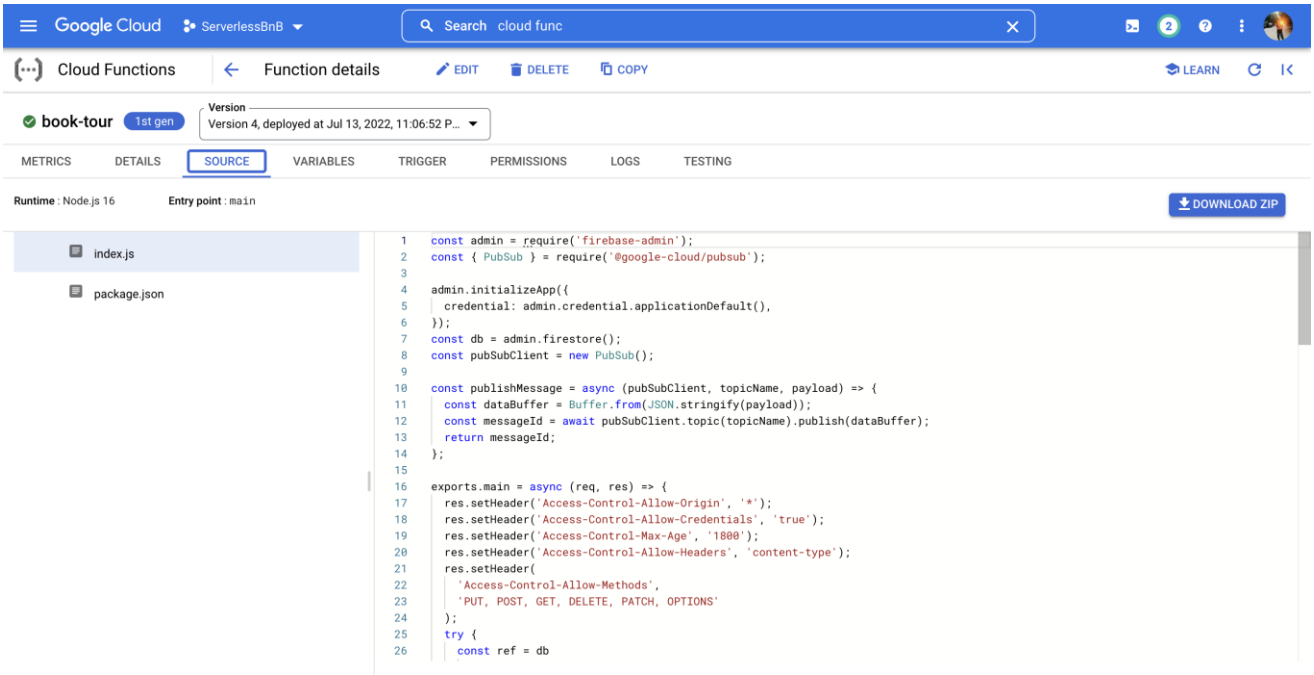

Figure 59: "book-tour" cloud function created to receive tour orders from user.

Pseudo Code:

- receives tour order details from frontend
- checks availability of tour type for current day.
- If available, creates a tour order in database and sends confirmation; If availability is not enough; it sends an error message back.
- If the order is created, it publishes a message with order details to “tour_order” pub sub topic. A message published to “tour_order” topic triggers “fulfill-tour-operator” cloud function.

b. fulfill-tour-operator

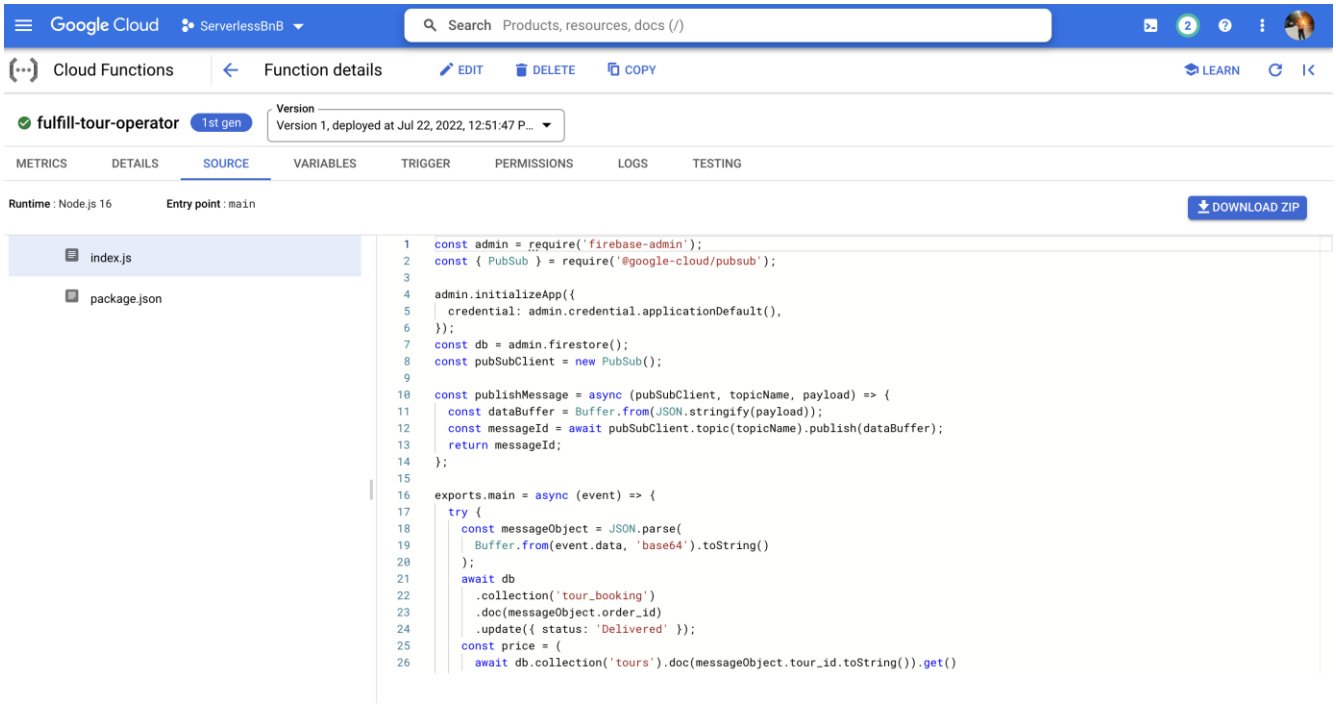

Figure 60: "fulfill-tour-operator" cloud function created to fulfill tour orders.

Pseudo Code:

- receives tour order details from “tour_order” topic.
- Changes the status of order from “Placed” to “Delivered” status.
- Creates an invoice for that tour order.
- Publishes a notification message to “notifications” topic which triggers a cloud function “send-notification”

c. send-notifications

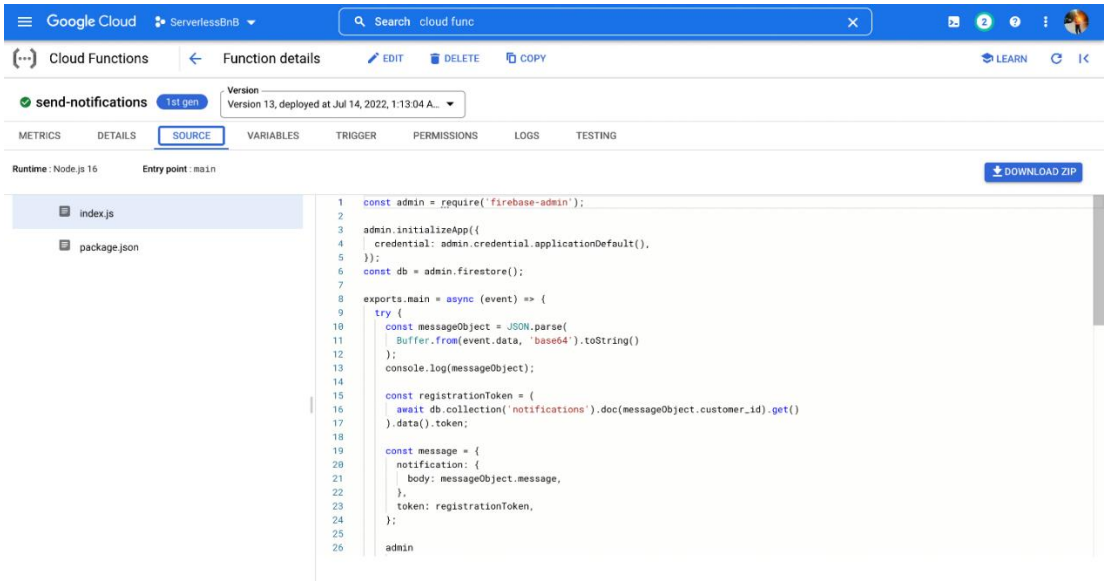

Figure 61: "send-notifications" cloud function created to send notifications to the frontend.

Pseudo Code:

- receives notification details from "notification" pub sub topic.
- Retrieves customer id from the message and then retrieves token for the corresponding customer.
- Transmits an order confirmation message to frontend using firebase cloud messaging.

Firebase Cloud Messaging:

This service is used for service to user communication. It transmits a message from "send-notifications" to a specific user token for frontend of the app. Frontend of the application is running a service worker which listens for any incoming notifications to the app.

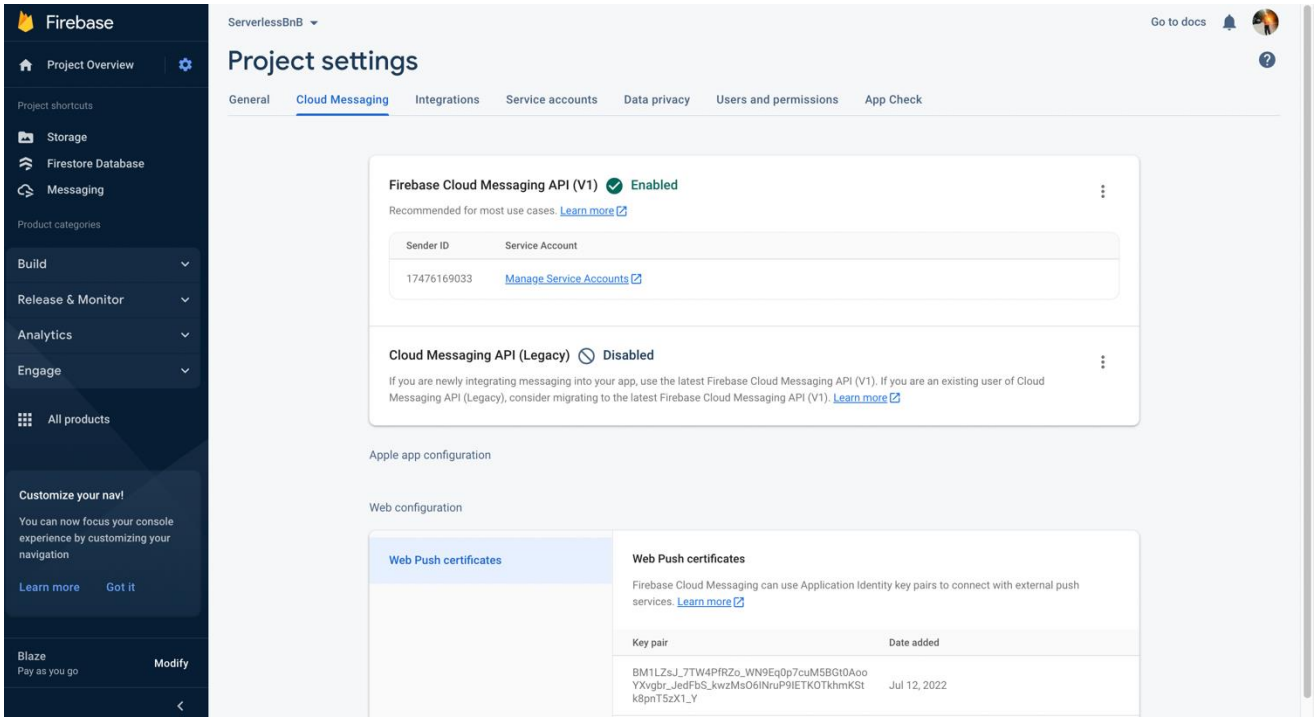

Figure 62: "Firebase cloud messaging" configuration to send notifications to the frontend.

Test Cases - Kitchen, Tour & Notification Service through Pub/Sub

1. Meal Order Insufficient Quantity

a. User Places a Meal Order from the frontend. The Quantity selected on the order is more than the availability for that meal type on current day.

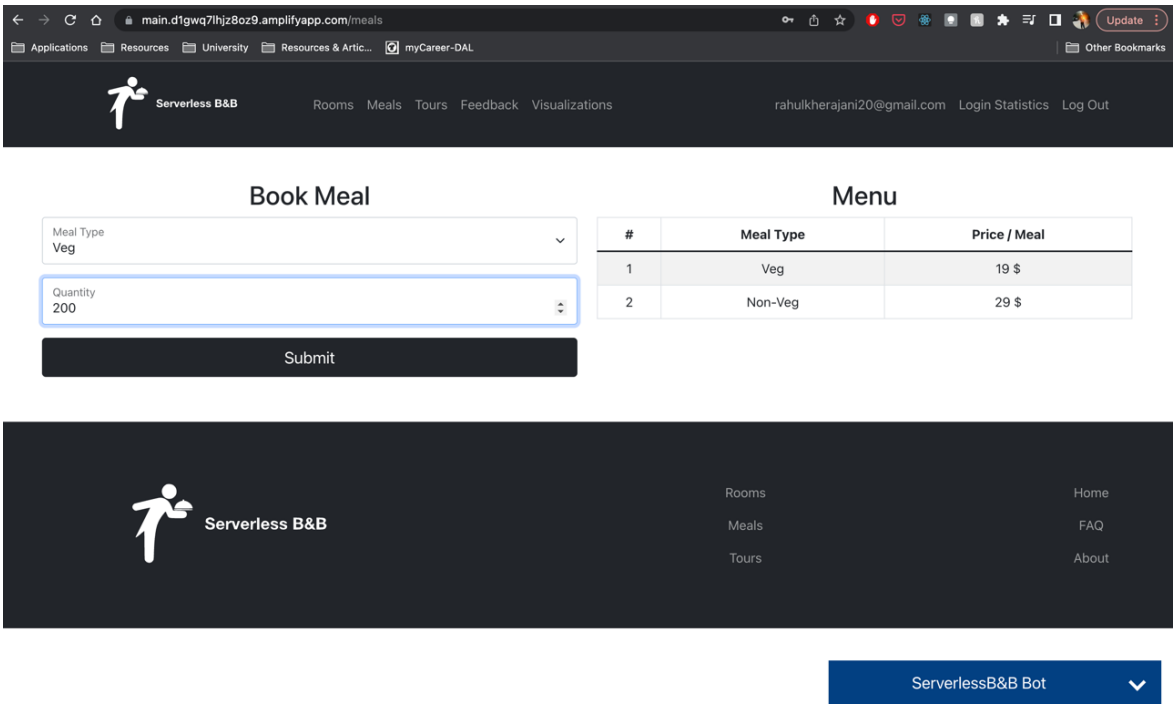

Figure 63: Meal Order placed with insufficient quantity.

b. Insufficient Quantity notification is sent to the user.

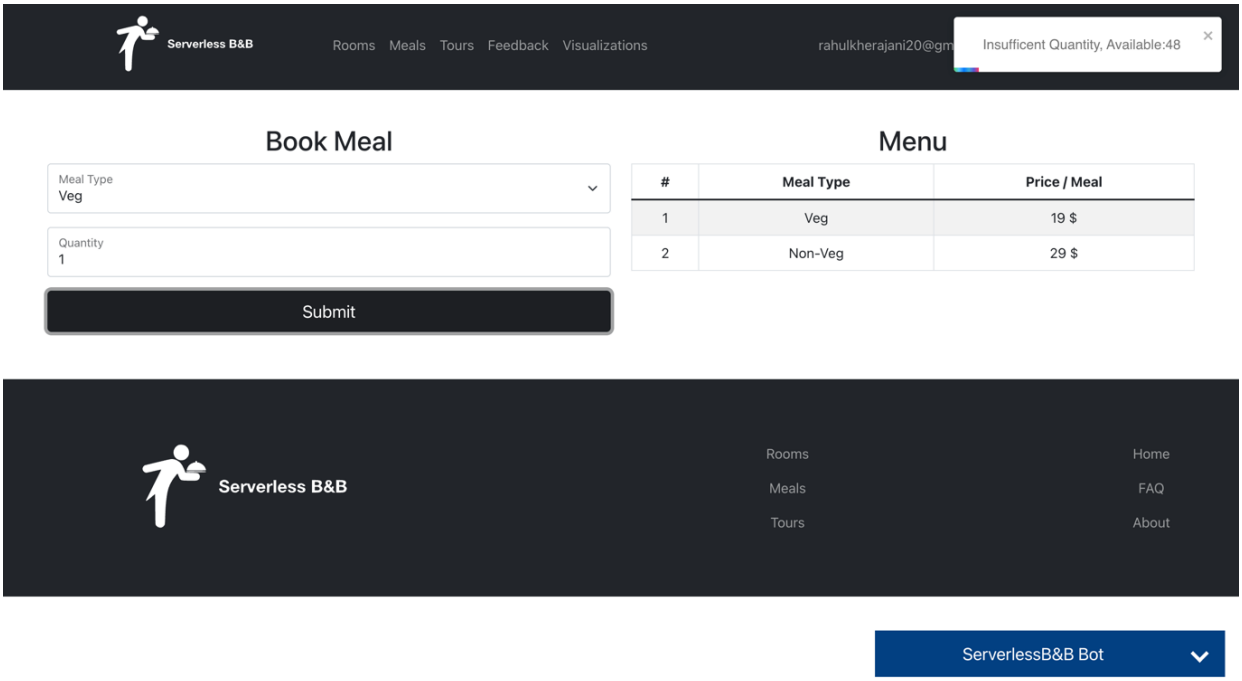

Figure 64: Insufficient quantity message sent to the user.

2. Meal Order Successful

a. User Places a Meal Order from the frontend with valid quantity. A meal order is booked with status 'Placed'

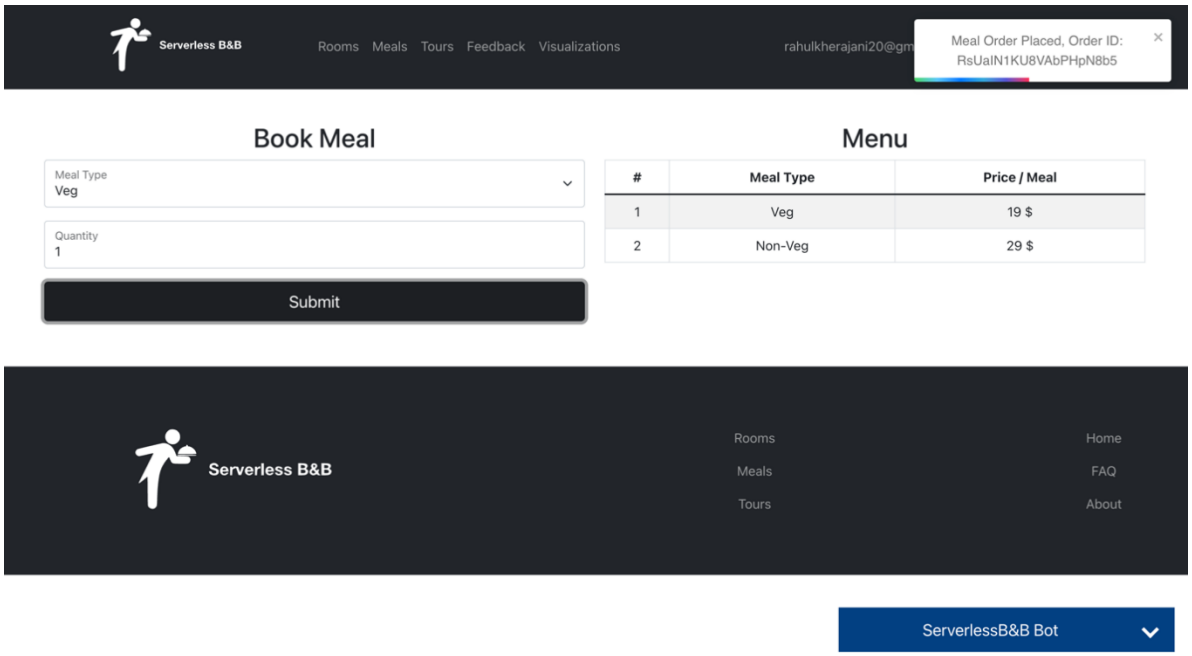

Figure 65: Meal order placed with valid quantity

b. Order is fulfilled through pub/sub triggered kitchen service and notification is sent back to frontend by notification service triggered through pub/sub.

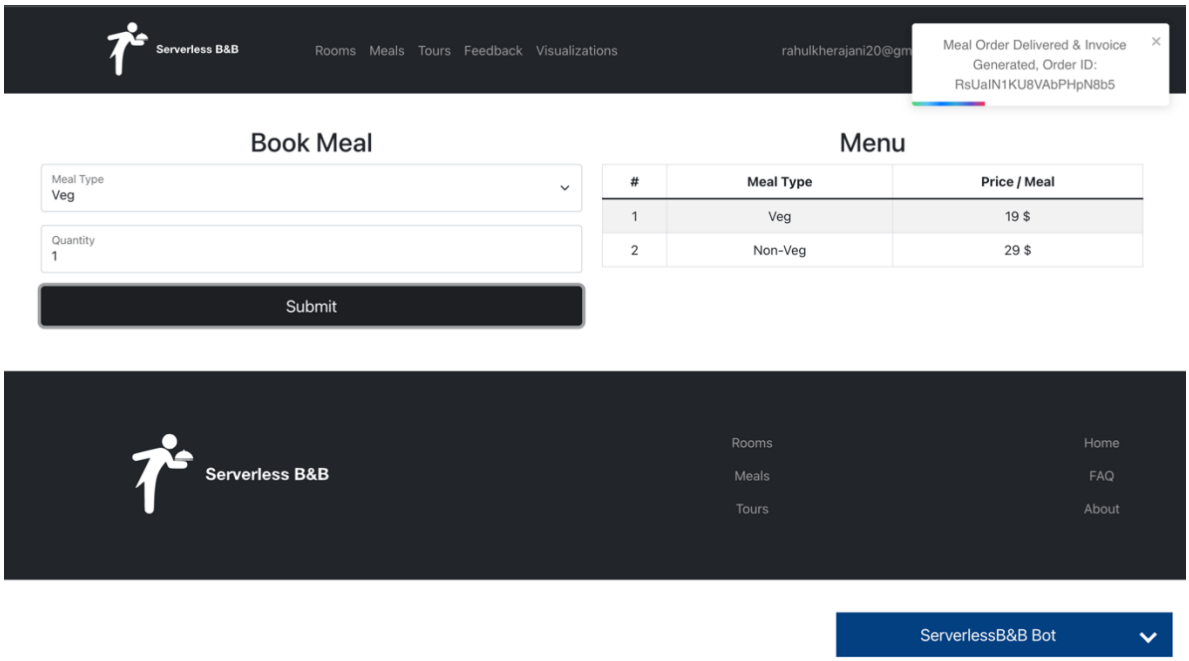

Figure 66: Meal order fulfilled through pub/sub.

3. Tour Order Insufficient Quantity

a. User Places a Tour Order from the frontend. The Quantity selected on the order is more than the availability for that meal type on current day.

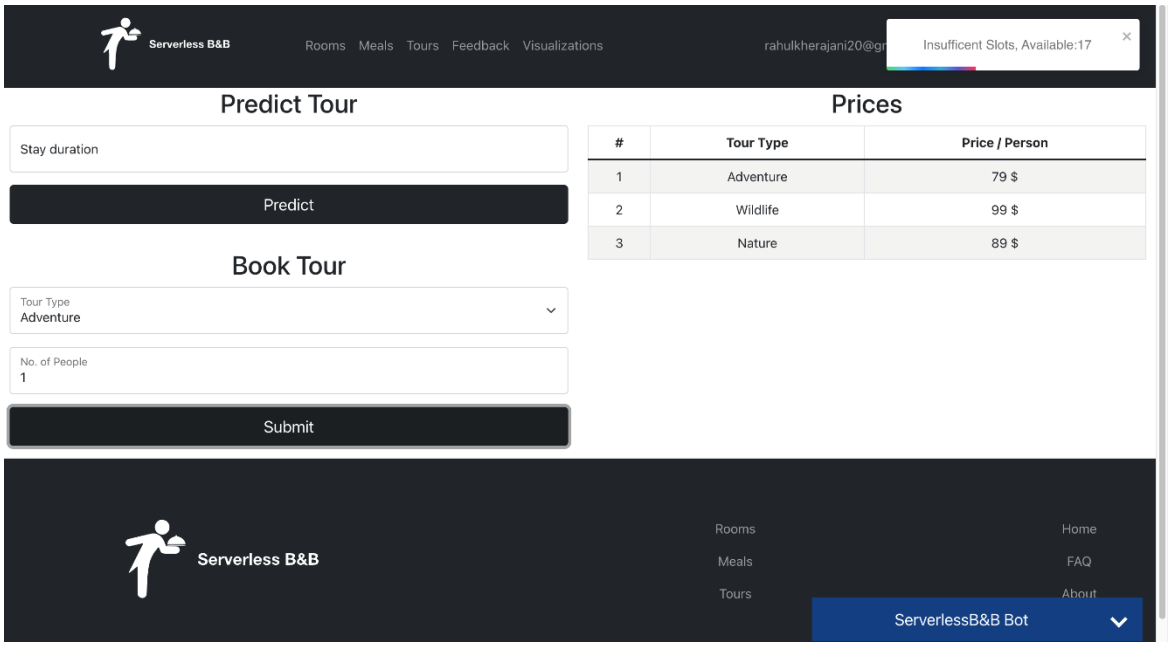

Figure 67: Meal order placed with insufficient available quantity.

b. Insufficient Quantity notification is sent to the user.

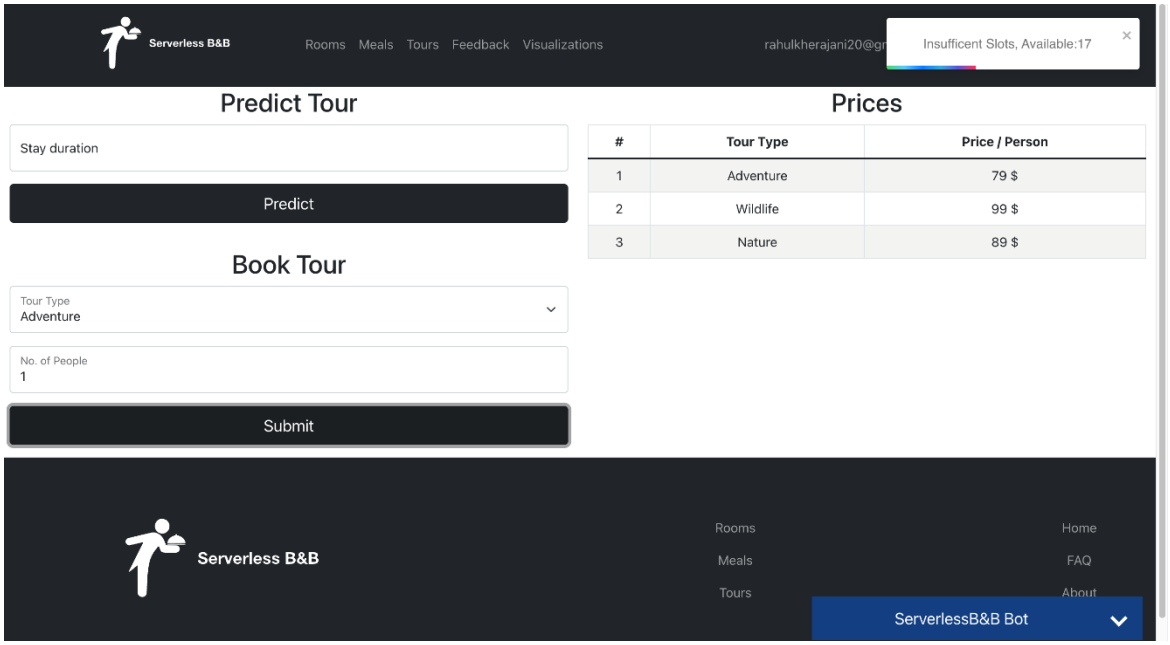

Figure 68: Insufficient quantity notification sent to the user.

4. Tour Order Successful

a. User Places a Tour Order from the frontend with valid quantity. A tour order is booked with status 'Placed'.

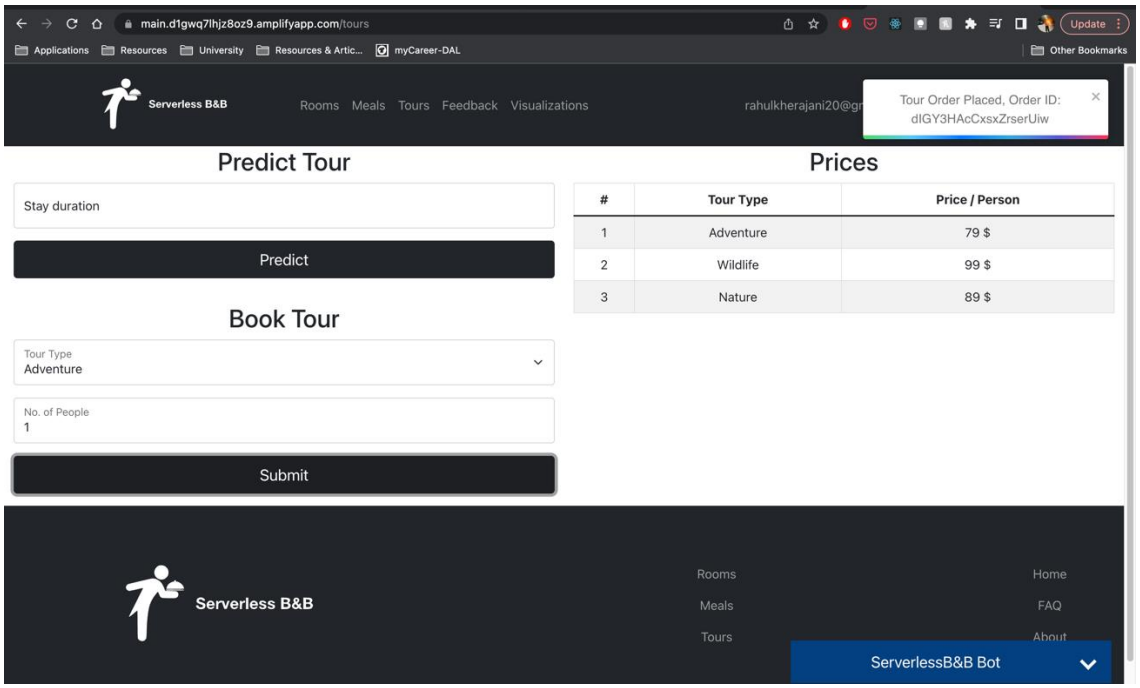

Figure 69: Tour order placed with valid quantity.

b. Order is fulfilled through pub/sub triggered kitchen service and notification is sent back to frontend by notification service triggered through pub/sub.

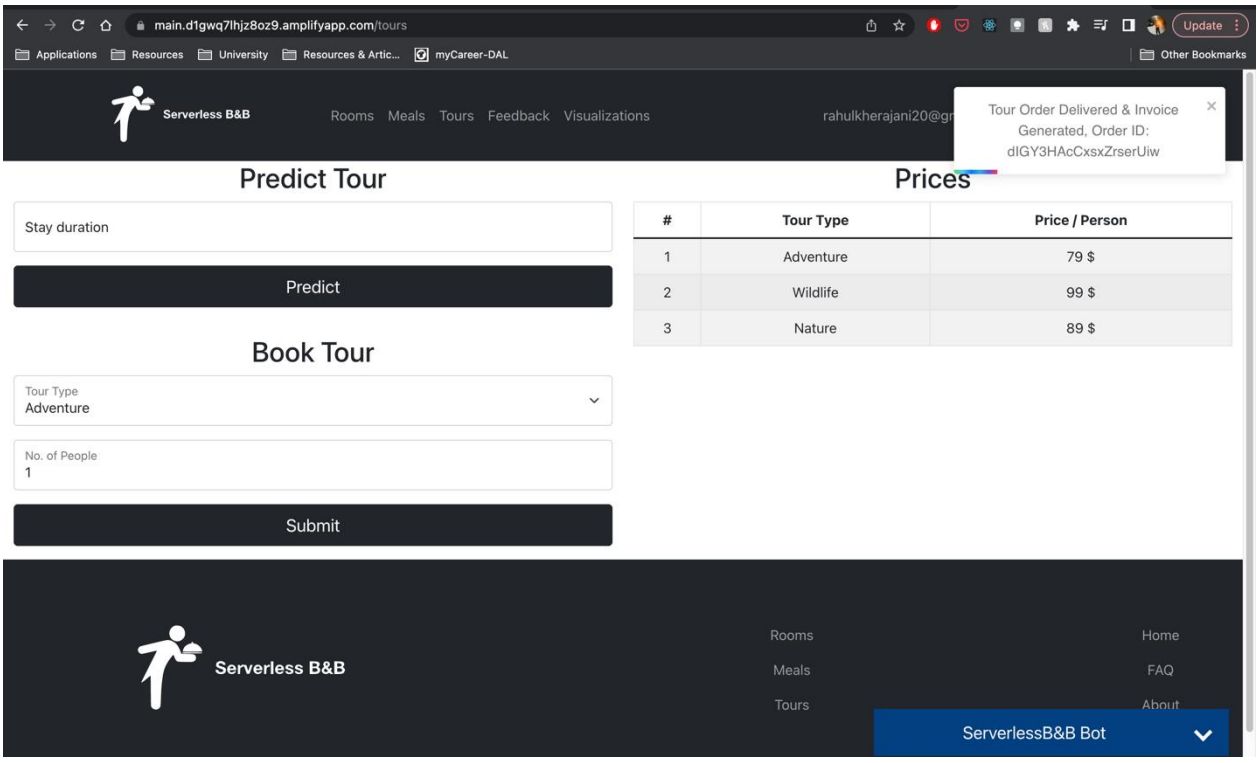

Figure 70: Tour order delivered using Pub/Sub.

Module 5: Machine Learning Module [8]

We have used Vertex AI service of GCP to implement whole ML module.

To identify the similarity of stay duration of customers and propose a tour package:
To implement this task, We have used VertexAI's AutoML service. More specifically, we used Tabular data classification model. Here we trained our model using predefined duration-package data. Then we deployed that model on Auto ML endpoint. Afterwards, using REST API we integrated those endpoints to the application.

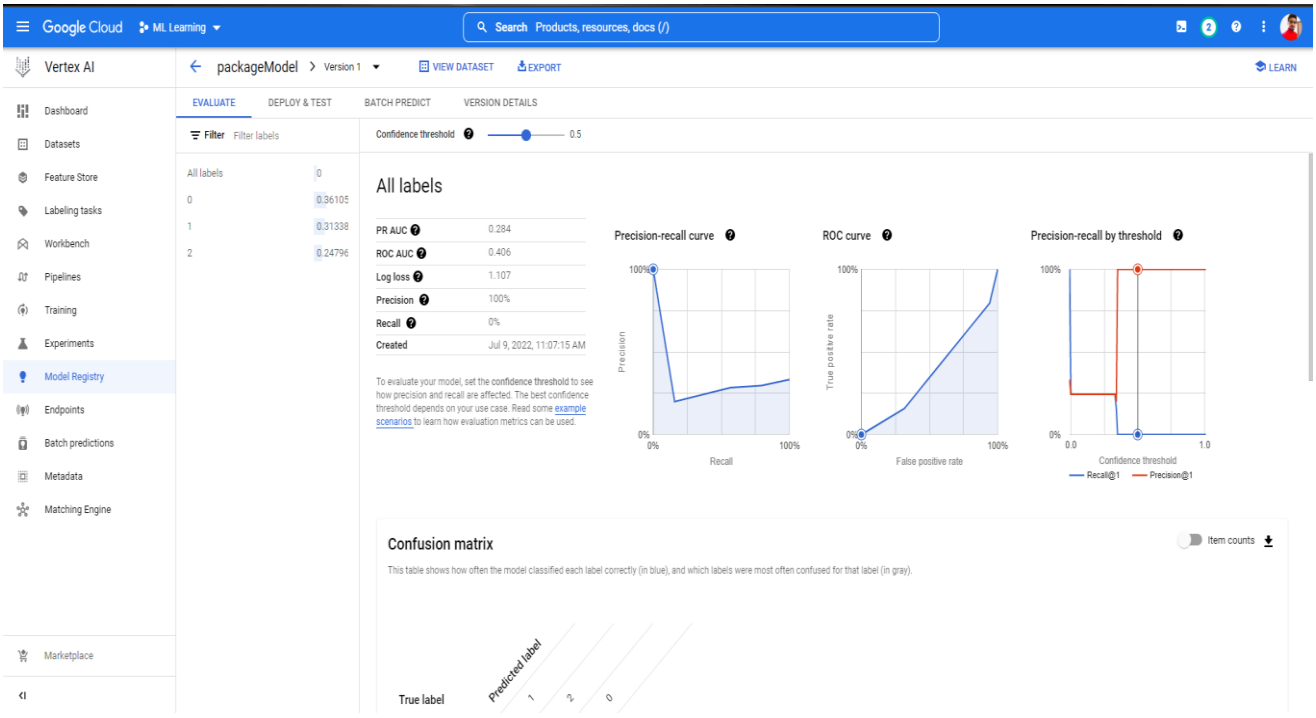

Figure 71: Trained model of package prediction.

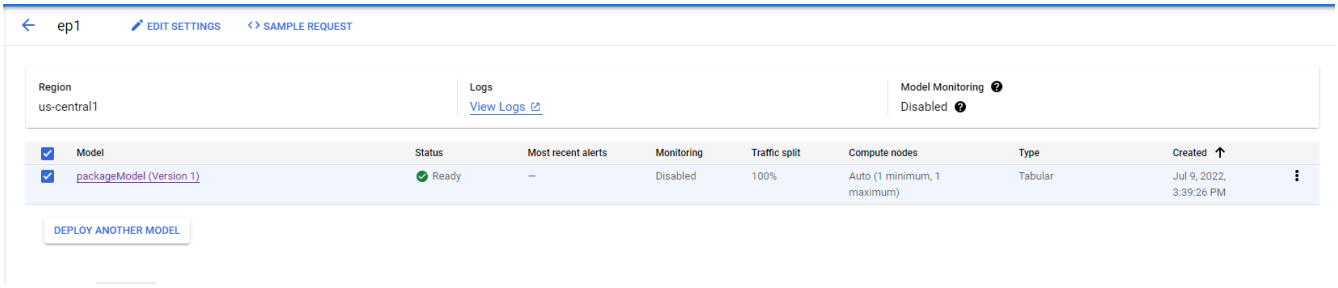

Figure 72: Model deployed on endpoint.

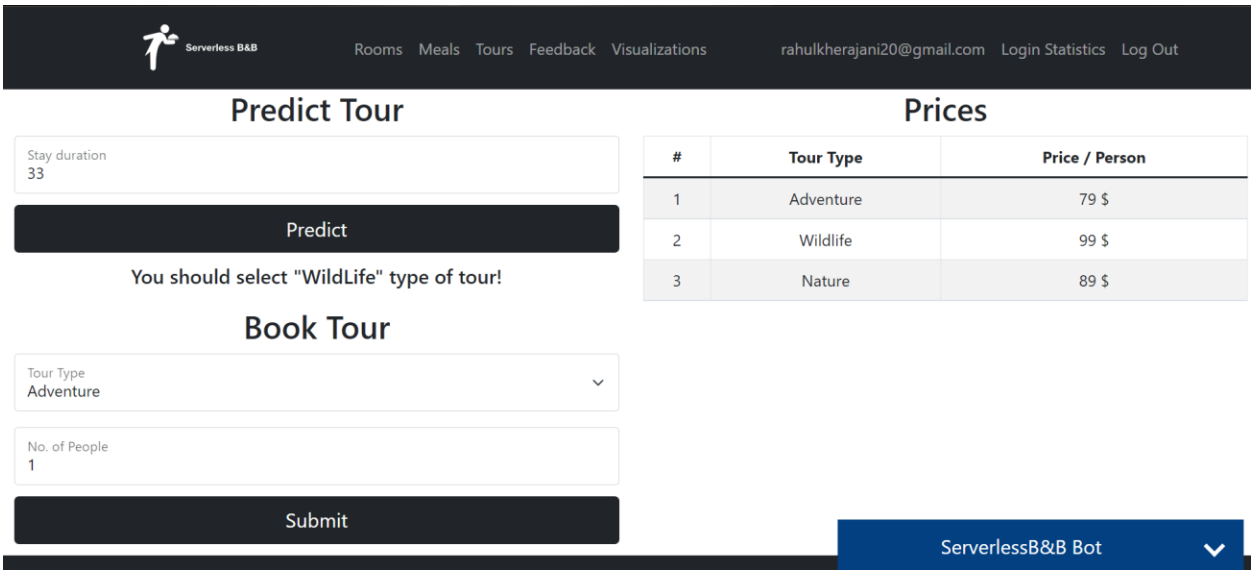

Figure 73: Predict Tour testcase.

On a tour booking page, there is a section named predict tour, where you can predict a tour based on your stay duration. When I entered 33 as my stay duration, the system suggests me to go for Wildlife type of tour.

```
const Bearer_token = "ya29.a0AVA9y1v3743Onieof7tqrTcxVyxts53vI_F_r8XBqncG46q8p4hrDyvl0P0tycScbA
const ProjectId = "ml-learning-352715";
const EndpointId = "8254772661528297472";

const predictPackage = async (duration, Bearer_token, ProjectId, EndpointId) => {
  const res = await axios.post('https://us-central1-aiplatform.googleapis.com/v1/projects/'
    + ProjectId + '/locations/us-central1/endpoints/' + EndpointId + ':predict',
    { "instances": { "duration": duration } },
    {
      headers: {
        'Authorization': 'Bearer ' + Bearer_token,
        'content-type': 'application/json'
      }
    });
  const maxi = await Math.max(...res.data.predictions[0].scores);
  const ind = await res.data.predictions[0].scores.indexOf(maxi);
  console.log(res.data.predictions[0].classes[ind]);
  return res.data.predictions[0].classes[ind];
};
```

Figure 74: Code snippet to predict a package.

To identify the polarity of customer feedback and to add appropriate score:
To implement this task, we have used VertexAI's AutoML service. More specifically, we used Sentiment Analysis model. Here we trained our model using predefined Feed-back dataset. Which has 30 feedbacks (15 for each type of sentiment). Then we deployed that model on Auto ML endpoint. Afterwards, using REST API we integrated those endpoints to the application.

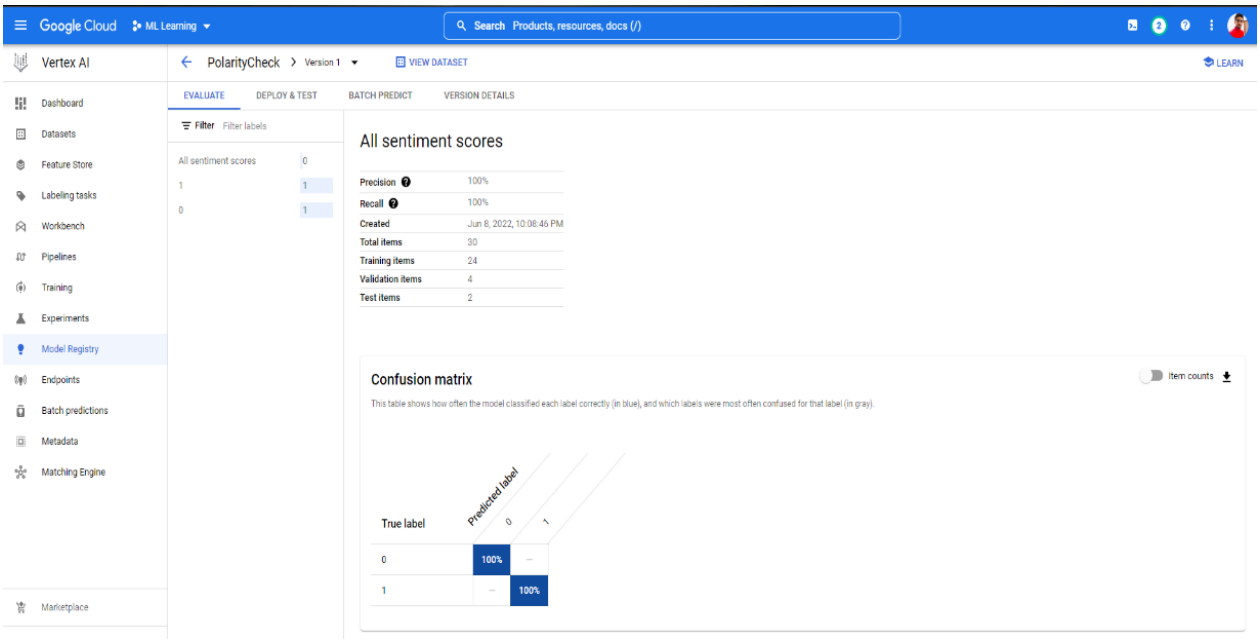

Figure 75: Trained model for sentiment analysis.

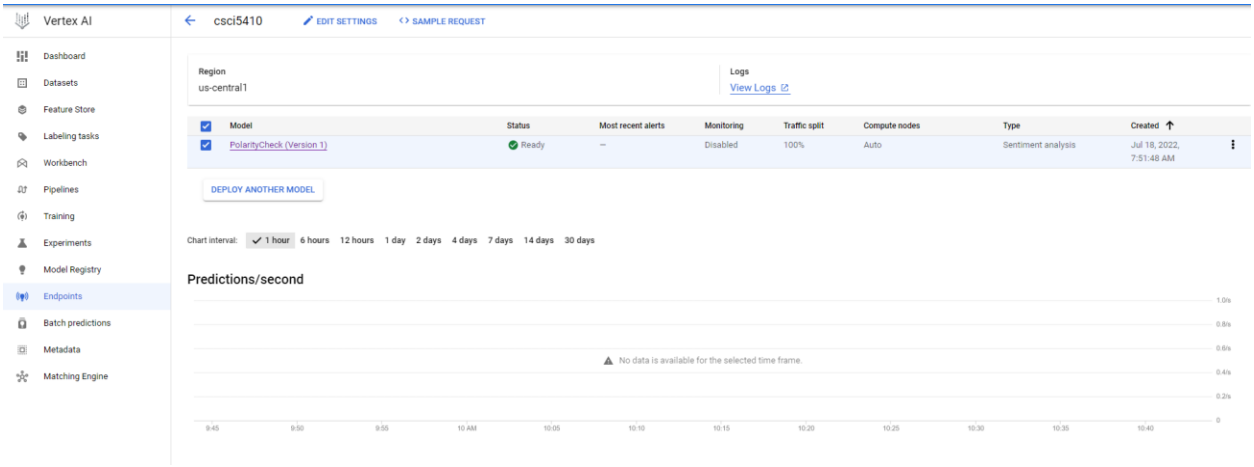

Figure 76: Endpoint for sentiment Analysis model.

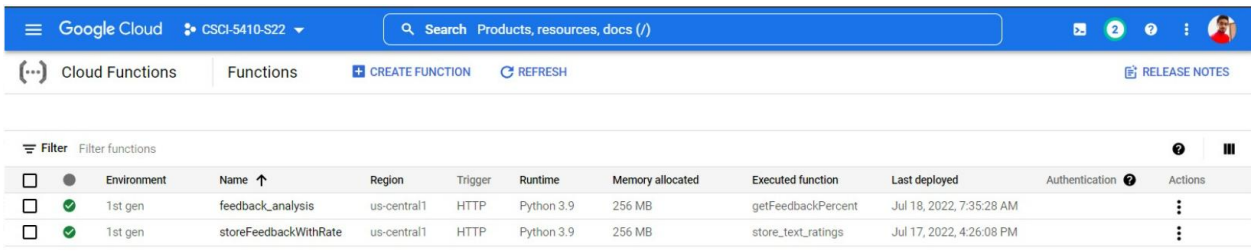

Figure 77: Cloud Functions for sentiment Analysis.

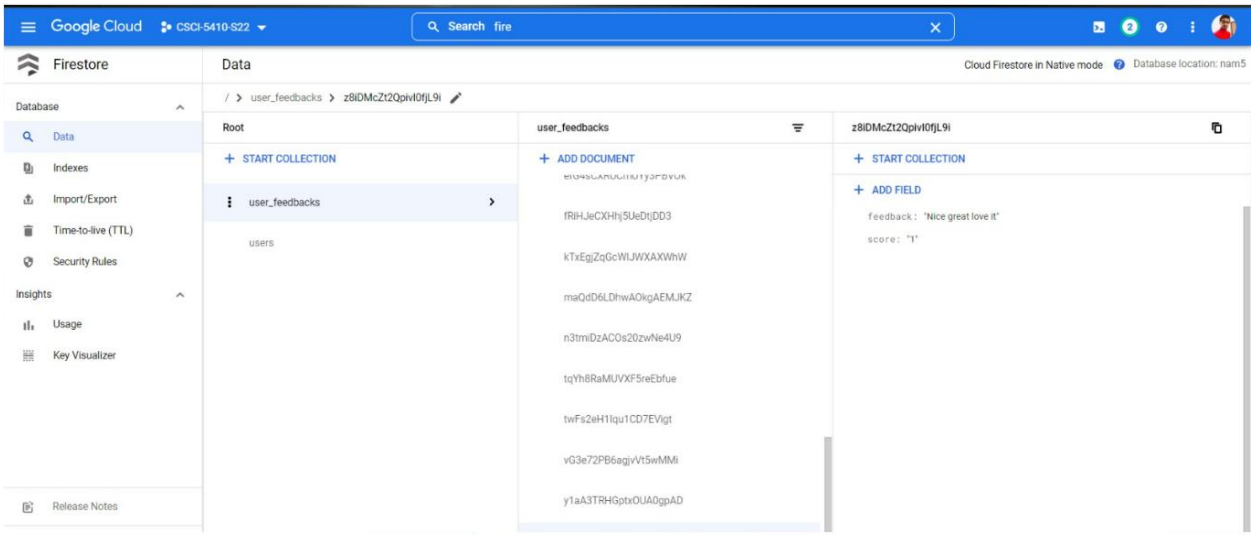

Figure 78: Firestore for sentiment analysis.

```
const Bearer_token = "ya29.a0ARrdaM_QRFr-K4bTHU6Ymanqq-wxPs1R91U_Im_HNIjYkDE8OL2k3g5HdUo0Q4OaXc
const ProjectId = "ml-learning-352715";
const EndpointId = "5946255640035852288";

const getSentiment = async (sentence, Bearer_token, ProjectId, EndpointId) => {
  const res = await axios.post('https://us-central1-aiplatform.googleapis.com/ui/projects/'
    + ProjectId + '/locations/us-central1/endpoints/' + EndpointId + ':predict',
    { "instances": { "mimeType": "text/plain", "content": sentence } },
    {
      headers: {
        'Authorization': 'Bearer ' + Bearer_token,
        'content-type': 'application/json'
      }
    });
  console.log(res.data.predictions[0].sentiment);
  return (res.data.predictions[0].sentiment);
};
```

Figure 79: Code snippet to predict sentiment.

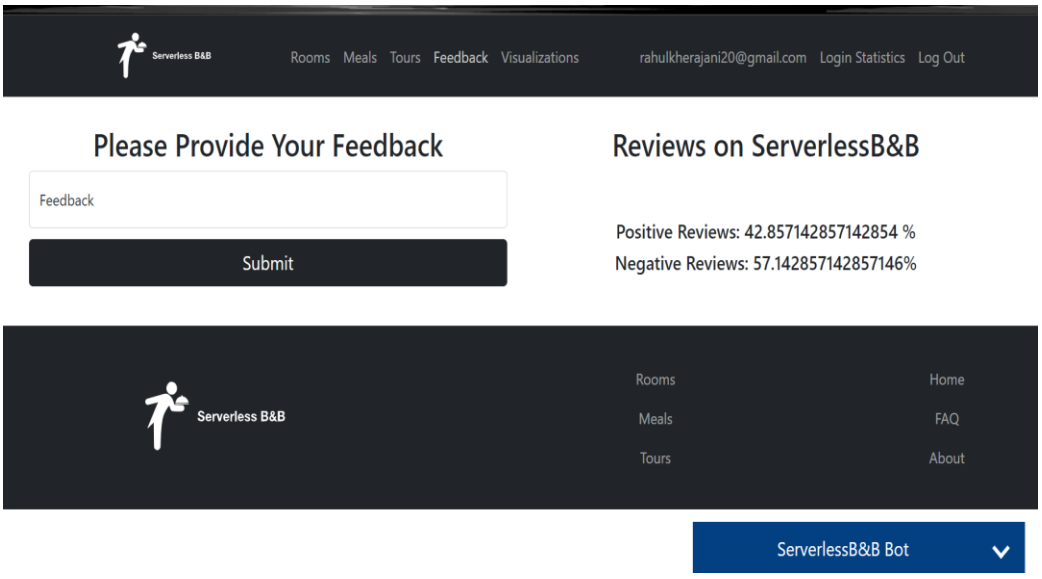

Figure 80: Feedback page.

Here, user provides feedback, and it will be stored in a firestore. Then we analyze those feedbacks and provide the output that how much portion is positive.

Module 6: Web Application building and hosting

As the program's user-facing front-end module, this module serves as the application. Utilizing the ReactJS library, this module was created. We can design reusable components with ReactJS, which enables us to develop interactive applications. It has a large user base, is adaptable, and is simple to manage. Python and NodeJS are used to build and deploy modules in the backend's micro-services architecture. Amplify is used to host the front-end application.

Module 7: Other Essential Modules – Testing, Report Generation, and Visualizations

Room Booking

To book a room, user can navigate to "Rooms" section from the navbar. On the book room page, user can see various types of Rooms available for stay along with their respective price. Mainly, they include Non-AC rooms, AC rooms, Deluxe rooms, and Suites. Figure 81 shows the web screen for booking rooms.

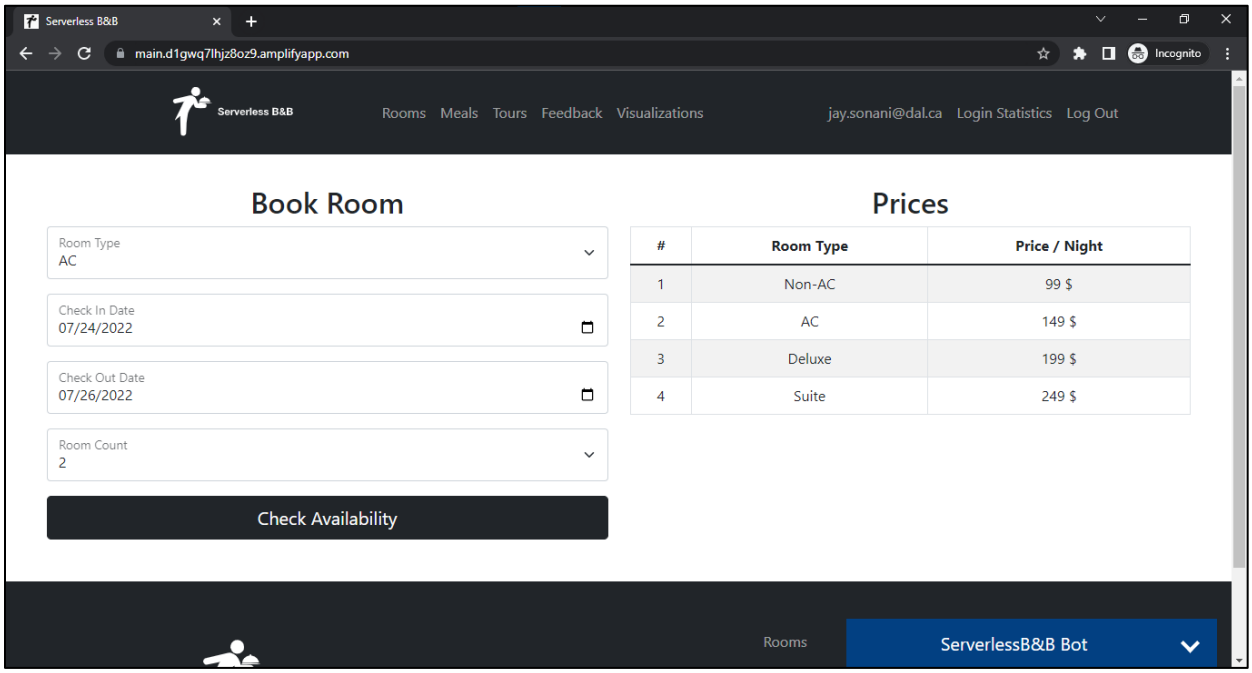

Figure 81: Room booking screen.

Test Cases

Booking room – rooms not available

When user clicks on "Check Availability" button after filling in the required details, /get-available-rooms API will be called using the provided inputs. An alert box with an appropriate message will be displayed with the availability of rooms. Figure 82 shows the alert message when there are no rooms available for the selected dates.

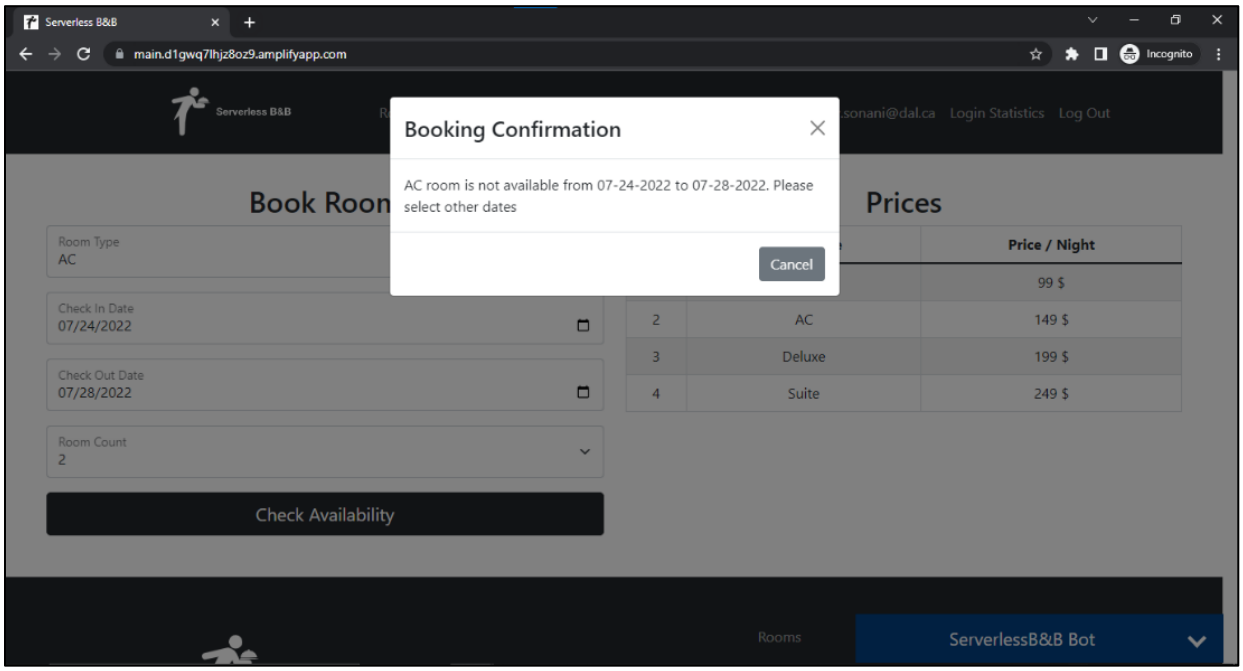

Figure 82: Rooms not available alert box.

Booking room – rooms are available

If rooms are available for the selected date, the system asks for confirmation from the user. Figure 83 shows an alert box asking for confirmation before making the final booking.

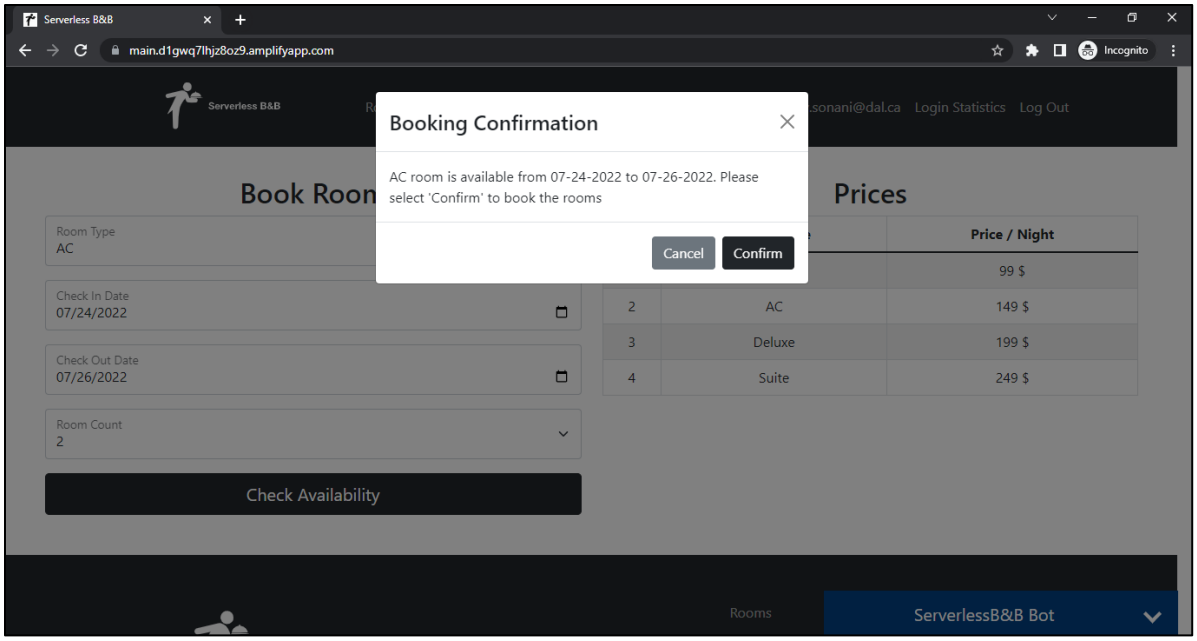

Figure 83: Confirmation alert box.

Once the user confirms the booking, a notification has been pushed and the same can be seen on the home page. Moreover, an invoice for the booking has also been generated. Figure 84 illustrates the logs of notifications and invoices.

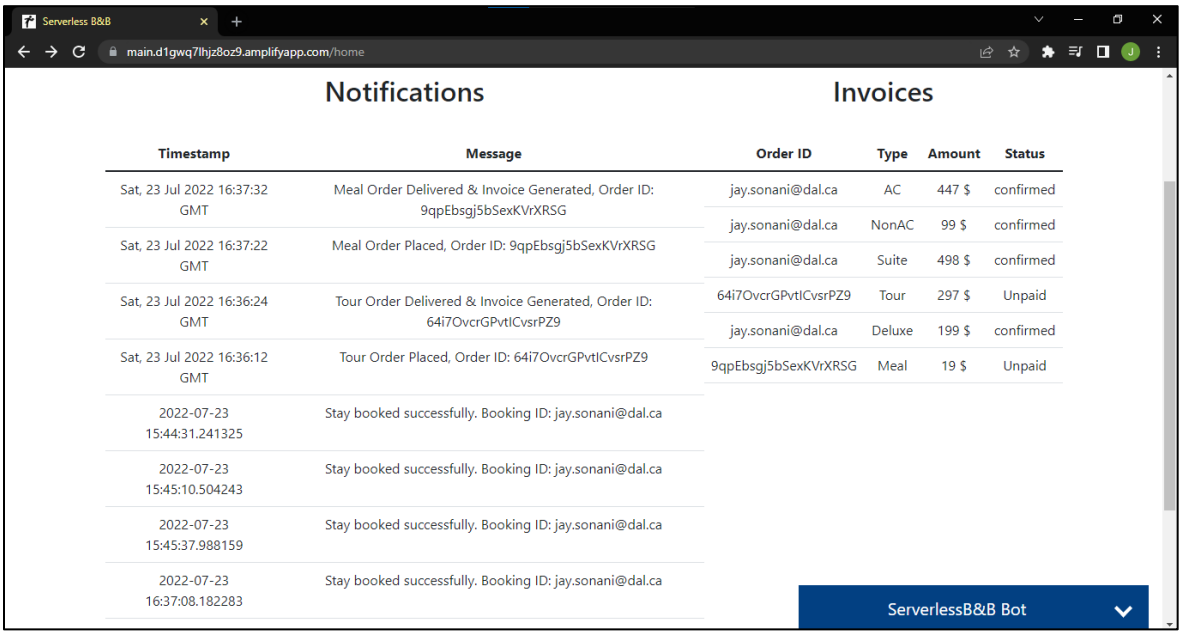

Figure 84: Notifications and Invoices.

To check the availability of rooms in the database, there is a POST API that accepts start and end date and provides the response with all the available rooms with their count between selected dates. Figure 85 shows the end point configured in the API Gateway's config.yaml file.

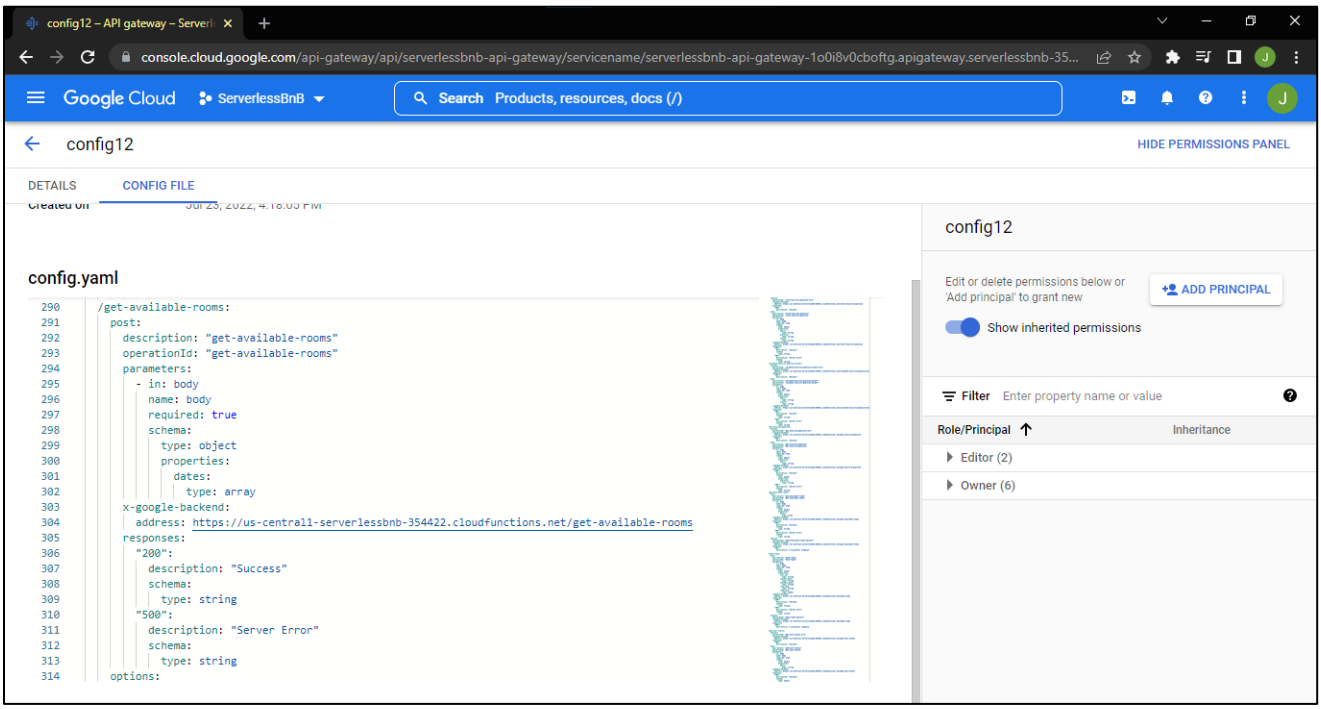

Figure 85: Get-available-rooms configuration for API gateway.

This API invokes the cloud function /get-available-rooms using the HTTP trigger URL of the cloud function. Figure 86 shows the cloud function responsible for providing room availabilities.

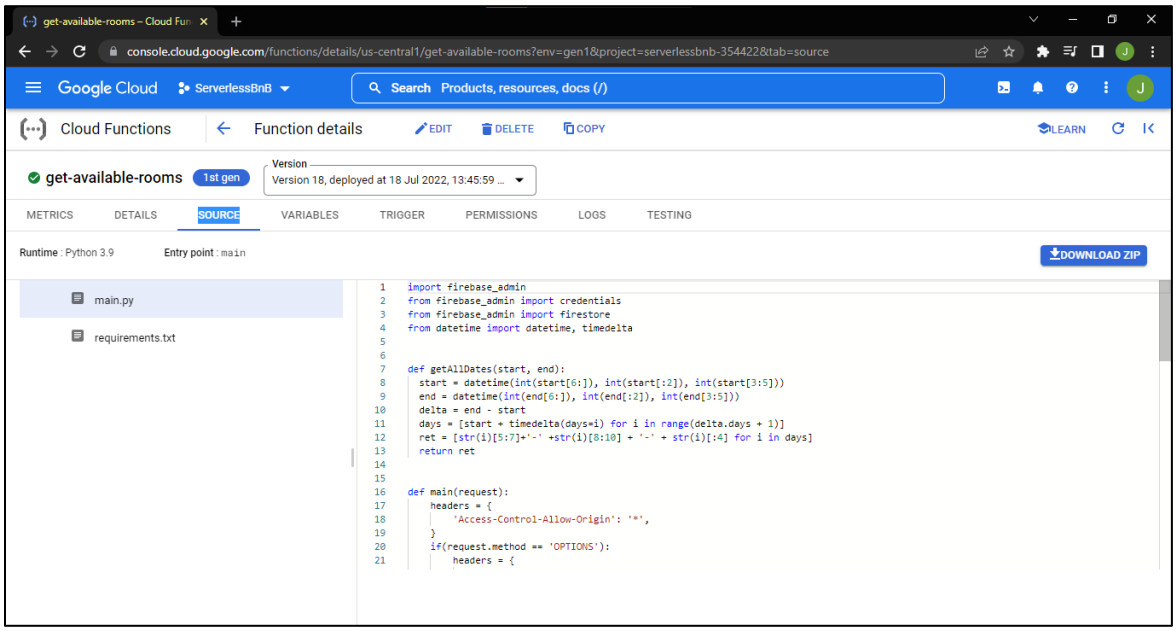

Figure 86: Cloud function "get-available-rooms".

The get-available-rooms cloud function fetches the data from a firebase collection. A collection named "room_availability" is maintained that contains one document per date. These documents store count for all types of the rooms. Figure 87 shows the described firebase collection.

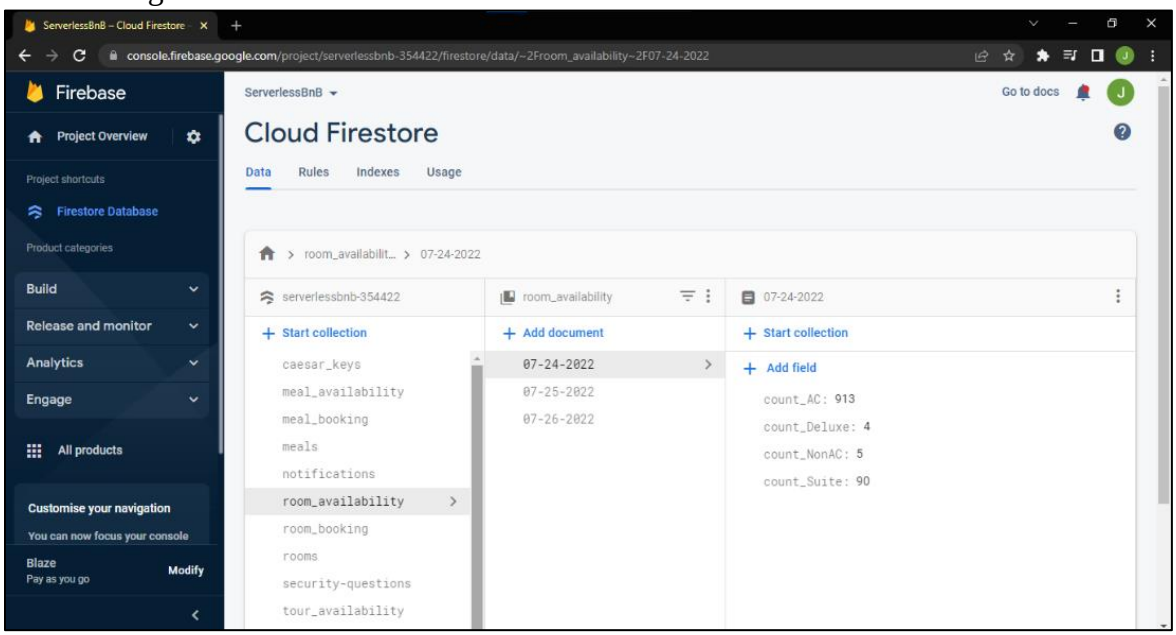

Figure 87: Firebase collection "room_availability".

To book rooms, book-rooms POST API will be called. This API is configured in the GCP API gateway in the configuration file as it can be seen in the Figure 88.

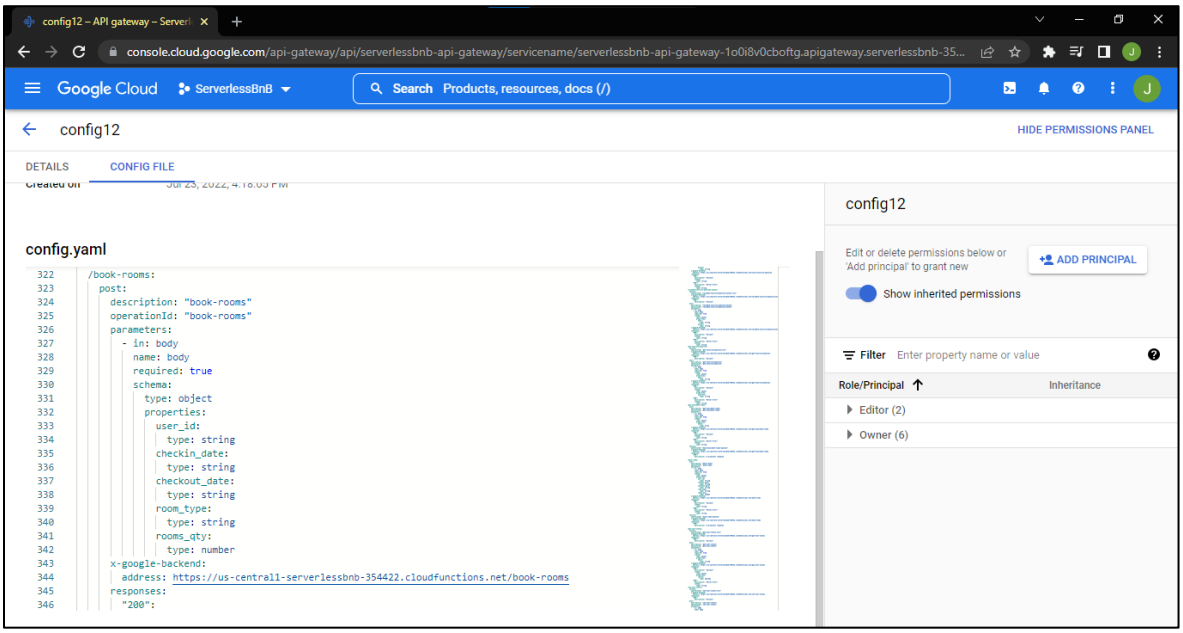

Figure 88: Book-rooms configuration for API gateway.

This API invokes the cloud function "book-rooms" that will first check for the availability of the rooms. If rooms are available on all the requested dates, it will create a booking and store in in "room_booking" firebase collection. Figure 89 and Figure 90 show the cloud function "book_rooms" and the firebase collection "room_booking" respectively.

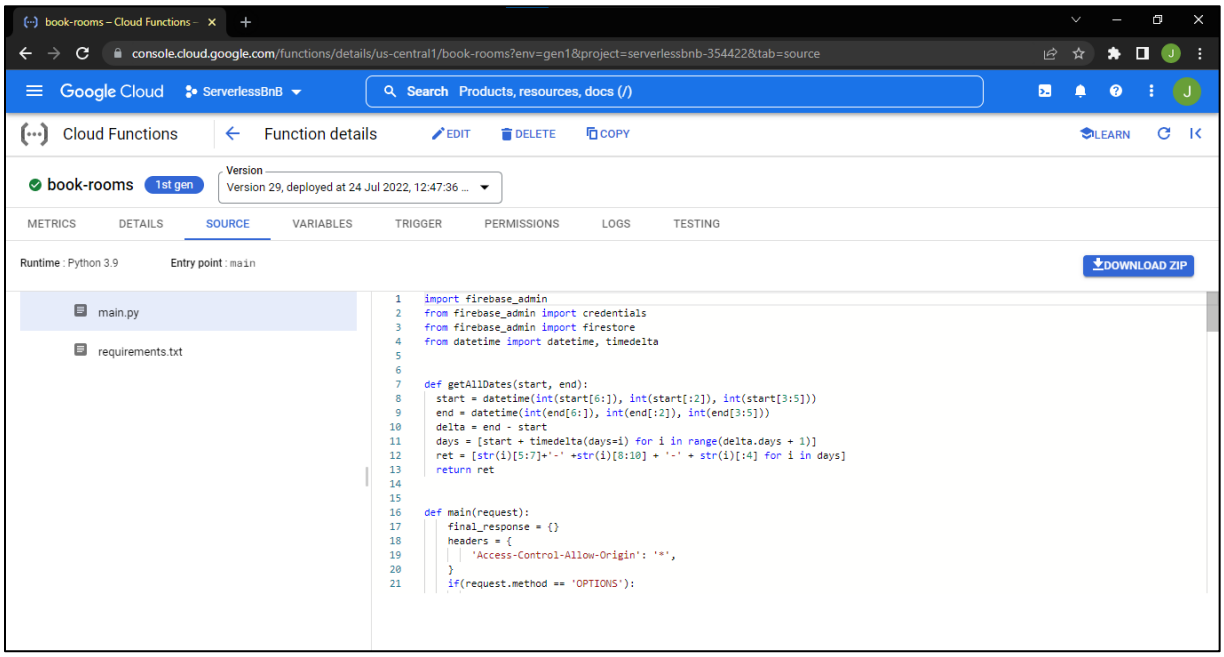

Figure 89: Cloud function "book-rooms".

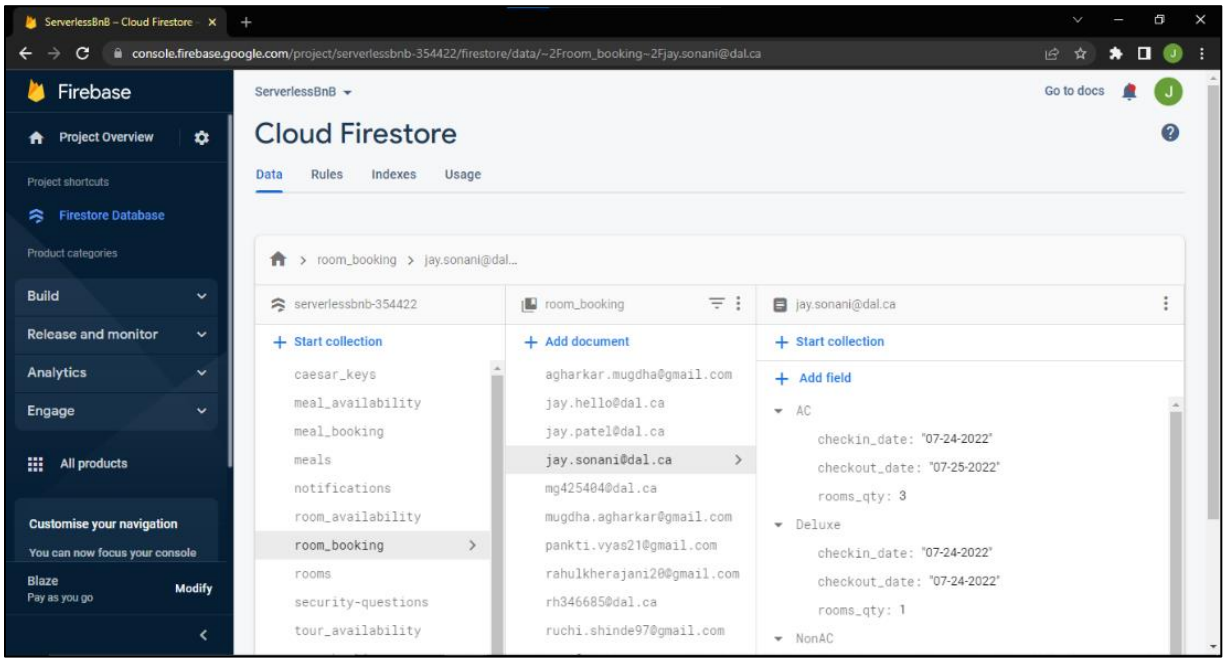

Figure 90: Firebase collection "room_booking".

Pseudo Code

- User navigates to book room section
- User fills the form
- User hits the submit button
- Room availability is checked
- If rooms are not available, appropriate message has been shown
- If rooms are available, a confirmation is asked
- Once the user confirms, rooms are booked
- Notification and invoice are generated for the booking

Report Generation:

In this module, the previous login statistics of the user are displayed to them. The implementation for this involved triggering two cloud functions – one POST API (*update _session_timestamp)* to store the timestamp of login and logout in the Firestore collection (*user_login_stats*), and one GET API (*get_user_session_stats*) to get all the session information of the specified user from the same Firestore collection.

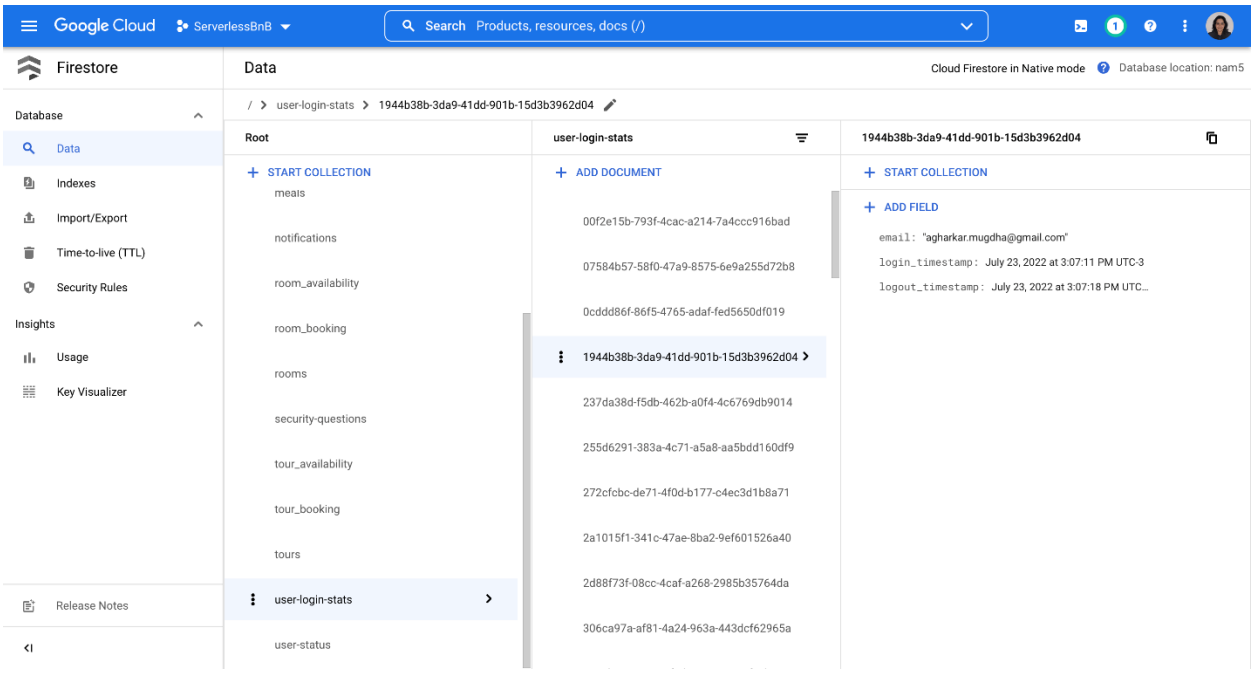

Figure 91: Firestore Collection for storing user login statistics.

When the user logs in, the *update _session_timestamp* is triggered with the email id of the user. It is a cloud function running behind the API Gateway which takes email id as the input and creates a document for that email id to store the login timestamp. When the user logs out, the log out time stamp is updated in the same document.

CaesarCipherPage.jsx | main.py .../update-session-timestamp | LoginForm.jsx | TourBookingPage.

frontend > src > Pages > CaesarCipherPage.jsx > CaesarCipherPage > handleCaesarCipher

```
13    const [plainText, setPlainText] = useState('');
14    const [wrongInputError, setWrongInputError] = useState('');
15    const navigate = useNavigate();
16
17    const handleCaesarCipher = async (event) => {
18      setLoading(true);
19      event.preventDefault();
20      const cipherTextInput = event.target.cipherText.value;
21      try {
22        const response = await axios.post(`/caesar-cipher/`, {
23          plain_text: plainText,
24          user_id: props.userId,
25          cipher_text: cipherTextInput,
26        });
27          if (response.data.success) {
28            document.dispatchEvent(userLogInEvent);
29            localStorage.setItem('cipher-key-verification-status', 'verified');
30            await axios.post('/update_session_timestamp/', {
31              email: props.userId,
32            })      You, 2 days ago • Add user login statistics
33            navigate('/home', { state: { userId: props.userId } });
34          }
35          else {
36            setLoading(false);
37            setWrongInputError("Wrong input! Please try again.")
38          }
39      } catch (error) {
40          setLoading(false);
41          setWrongInputError("Wrong input! Please try again.")
```

Figure 92: API Triggered when user successfully logs in.

AuthService.js | AuthService.js (bdf7883d) ↔ AuthService.js (fe354507) | MealBookingPage.jsx

frontend > src > Services > AuthService.js > logoutUser > currentUser.getSession() callback > currentUser.getUserAt

```
165   currentUser.getSession((error) => {
166     if (error) throw error;
167
168     currentUser.getUserAttributes((error, attributes) => {
169       if (error) throw error;
170
171       const emailAttribute = attributes.find((attribute) => attribute.Name === 'email');
172       const email = emailAttribute.Value;
173
174       currentUser.globalSignOut({
175         onSuccess: async () => {
176           localStorage.removeItem('security-question-answer-status');
177           localStorage.removeItem('cipher-key-verification-status');
178           document.dispatchEvent(userLogOutEvent);
179           await axios.post('/set-user-status', {
180             email,
181             status: false
182           });
183           await axios.post('/update_session_timestamp/', {
184             email: email,
185           })      You, 2 days ago • Add user login statistics …
186           window.location.assign('/login');
187         },
188         onFailure: (error) => {
189           throw error;
190         }
191       });
192     });
193   });
```

Figure 93: API Triggered when user logs out.

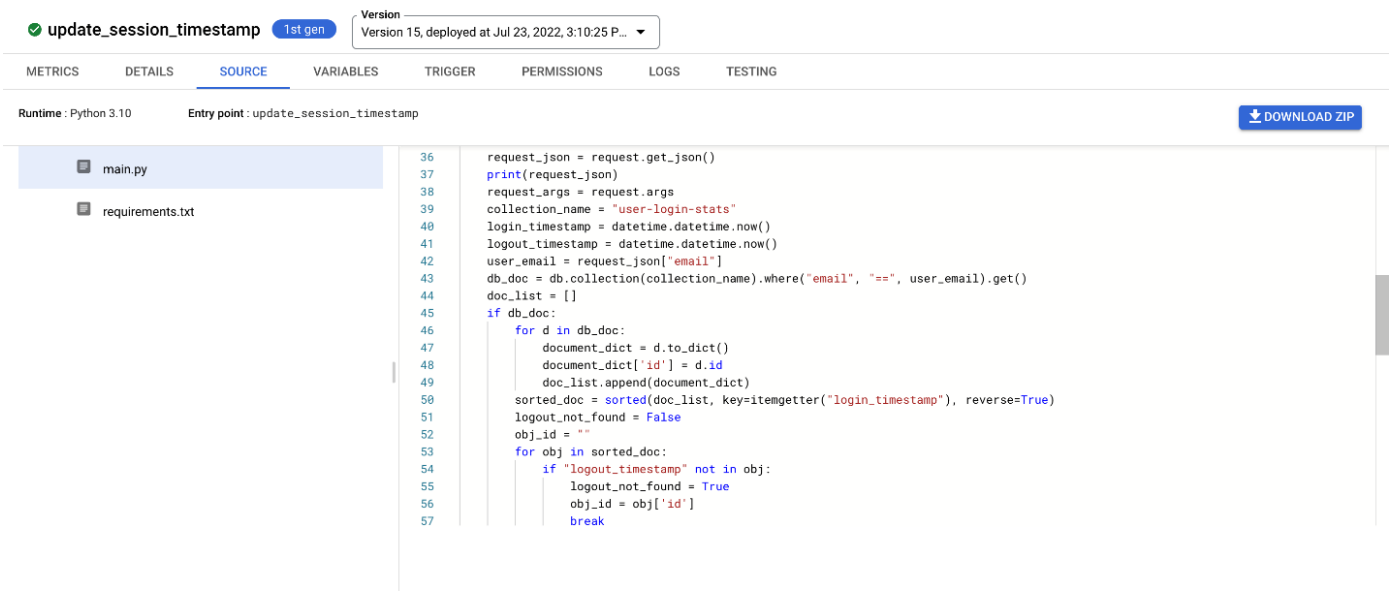

Figure 94: update_session_timestamp cloud function.

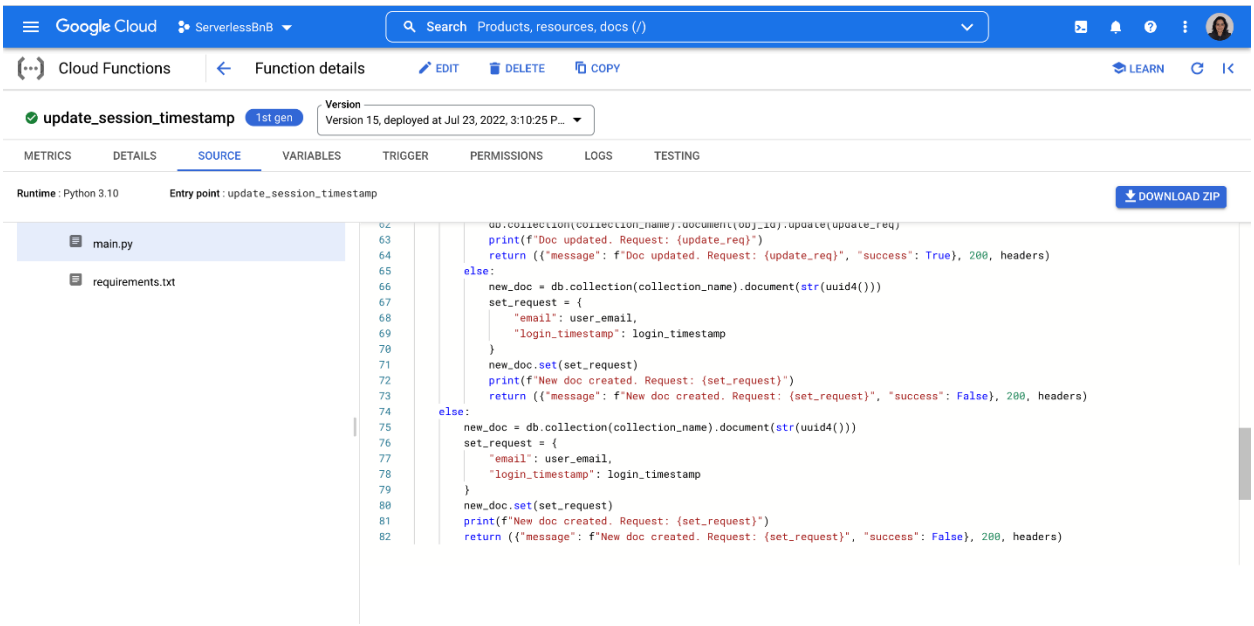

Figure 95: update_session_timestamp cloud function (continued).

If a logged in user wants to see their previous logged in statistics, they will be able to see the 'Login Statistics' link on the header of the webpage. On clicking the link, they will be navigated to the login statistics page. When the component loads for the first time, the *get_user_session_stats* API is called, and the email id of the user is passed in the query params.

```
VisualizationsPage.jsx | Header.jsx | main.py .../get-user-session-stats | LoginStatisticsPage.jsx 1
frontend > src > Pages > LoginStatisticsPage.jsx > LoginStatisticsPage > url
You, 2 days ago | 1 author (You)
1  import React from 'react';
2  import axios from '../Config/AxiosConfig';
3  import { useState, useEffect } from 'react';
4  import { useLocation } from 'react-router-dom';
5
6  const LoginStatisticsPage = (props) => {
7    const [sessionData, setSessionData] = useState([]);
8    const location = useLocation();
9    const userEmail = location.state.userId;
10   const url = "/get_user_session_stats?email=" + userEmail;      You, 2 days ago • Add user login stat
11
12   useEffect(() => {
13     axios.get(url)
14       .then(res => {
15         const data = res.data;
16         setSessionData(data.message);
17       })
18   }, []);
19
20
```

Figure 96: get_user_session_stats called to get the login information for the logged in user.

The *get_user_session_stats* cloud function will sort the documents in the *user_session_stats* collection so that the user will be able to check their sessions in the descending order.

```
Cloud Functions | Function details | EDIT | DELETE | COPY | LEARN
get_user_session_stats  1st gen  Version: Version 14, deployed at Jul 23, 2022, 7:25:35 P...
METRICS | DETAILS | SOURCE | VARIABLES | TRIGGER | PERMISSIONS | LOGS | TESTING
Runtime : Python 3.10   Entry point : get_session_for_user   DOWNLOAD ZIP
main.py
requirements.txt

    }
    user_email = request.args.get("email")
    print("email: ",user_email)
    collection_name = "user-login-stats"
    db_doc = db.collection(collection_name).where("email", "==", user_email).get()
    doc_list = []
    if db_doc:
        for d in db_doc:
            document_dict = d.to_dict()
            document_dict['id'] = d.id
            doc_list.append(document_dict)
    print(doc_list)
    sorted_doc = sorted(doc_list, key=itemgetter("login_timestamp"), reverse=True)
    return ({"message": sorted_doc, "success": True}, 200, headers)
    # else:
    #     return('', 204, headers)
except Exception as e:
    return ({"message": e, "success": False}, 500, headers)
```

Figure 97: get_user_session stats cloud function.

Following is the example of the logged in user '*agharkar.mugdha@gmail.com*'. Their previous 9 login sessions are visible with the login and logout timestamp.

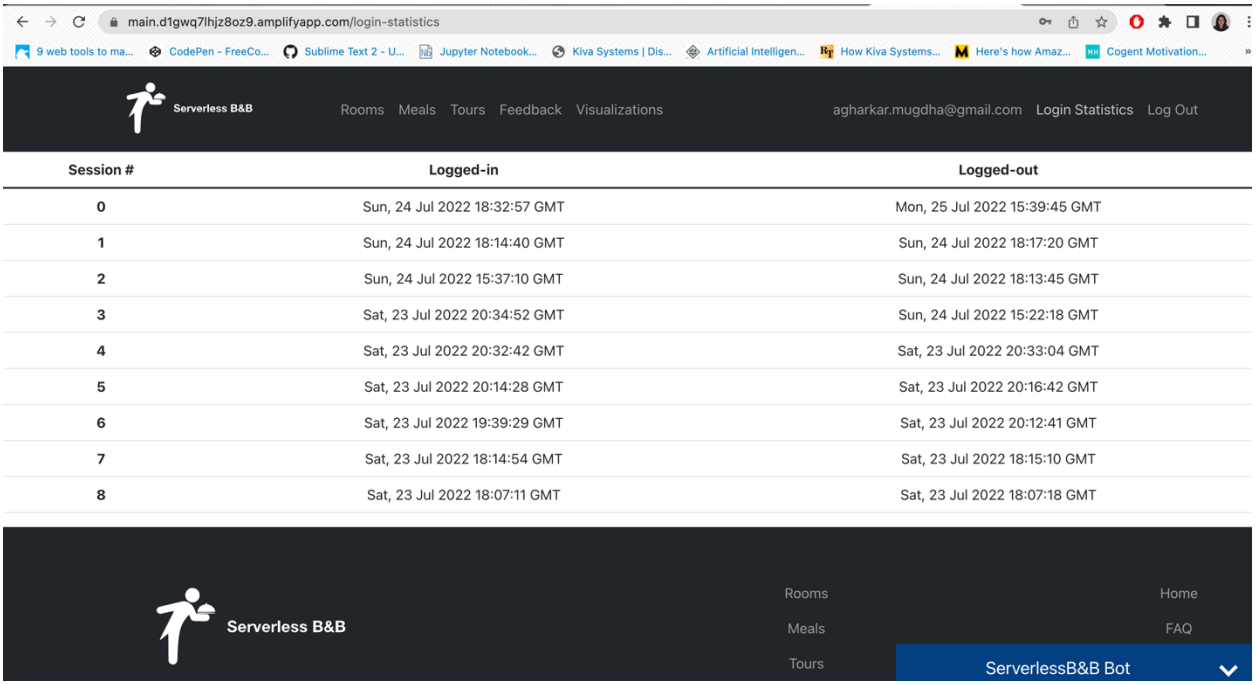

Figure 98: The logged in user can see their previous logged in sessions.

Visualizations:

For the visualization module, we have considered analysis for three as described below. We have used BigQuery for running the queries on our data and used Google Data Studio [10] to visualize the data.

1. Popular tours among the available tours – Adventure, Wildlife, Nature

   The *tours* collection in Firestore contains information about the different types of tours available. The *tour_booking* collection contains the booking information of all the tours. These two collections are exported from Firestore and imported into the *Serverlessbnb_visualizations* collection in BigQuery.

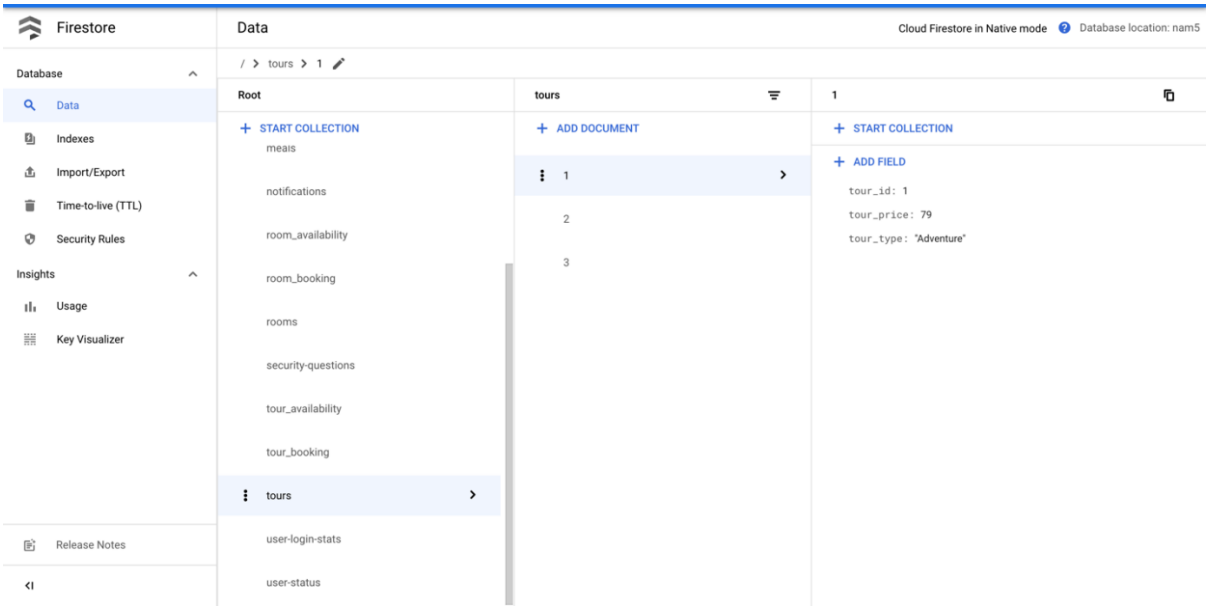

Figure 99: tours collection.

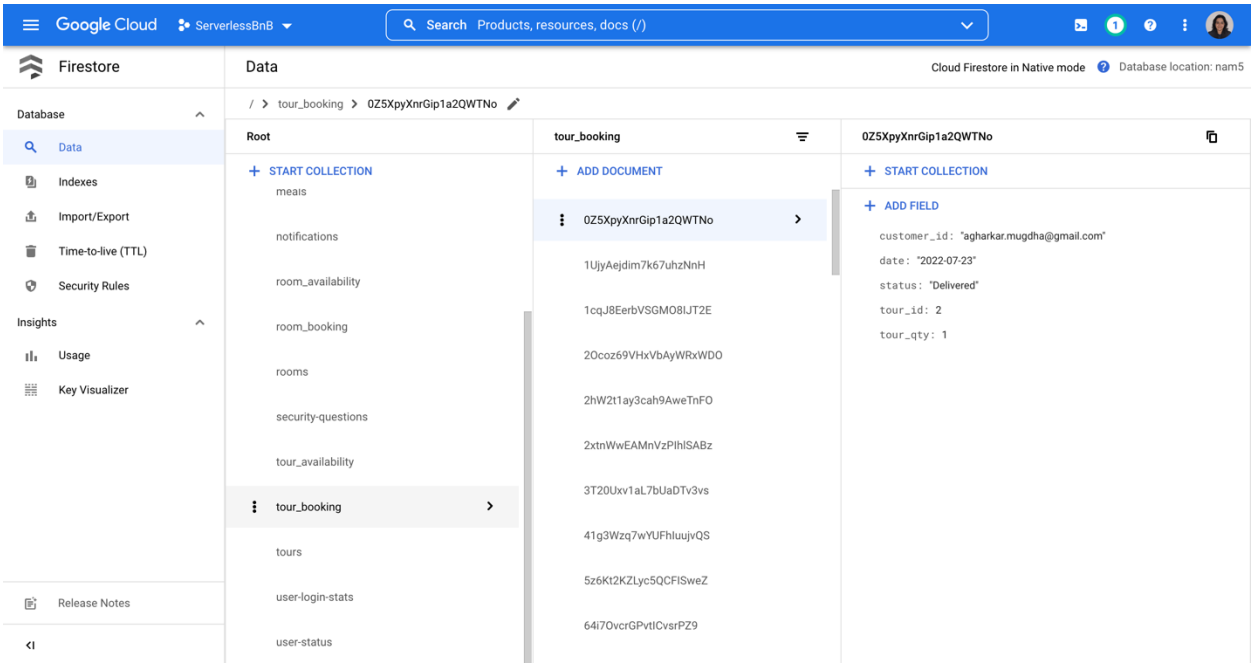

Figure 100: tour_booking collection.

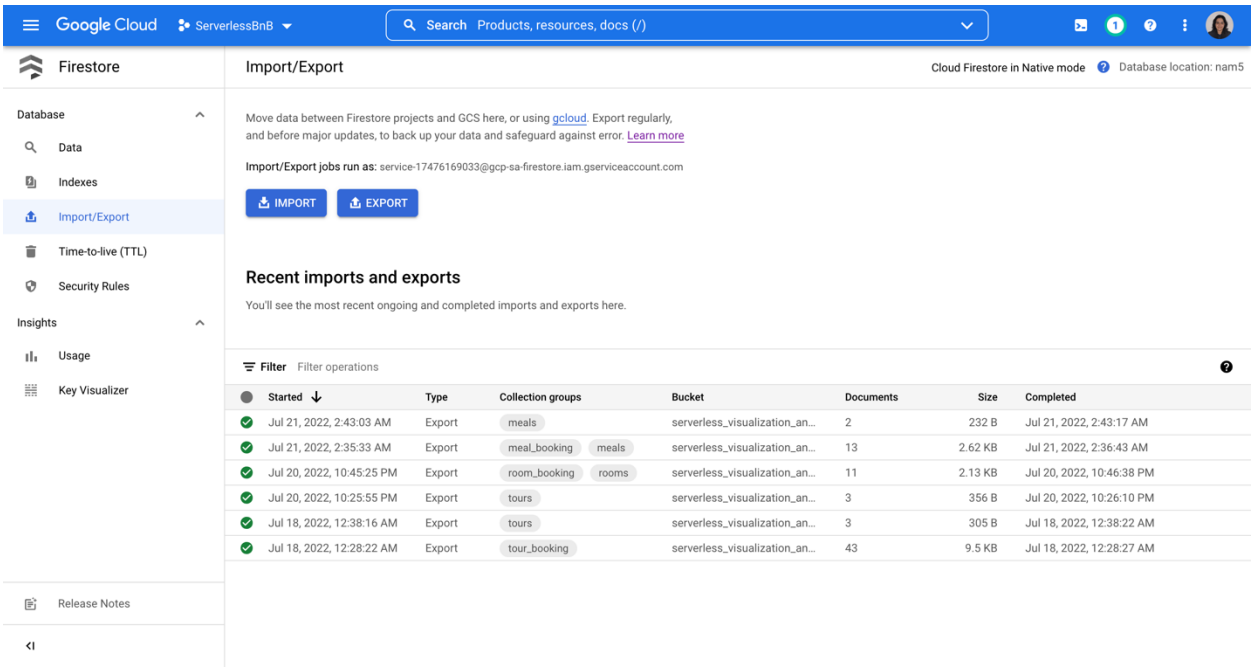

Figure 101: All exported collections from firestore

The *tours* and *tour_booking* collections are imported in a temporary folder in BigQuery, where a join query was performed to calculate the count of type of meals opted by the customers so far. There is an option in a dashboard to visualize the data with Google Data Studio as seen in the following figure.

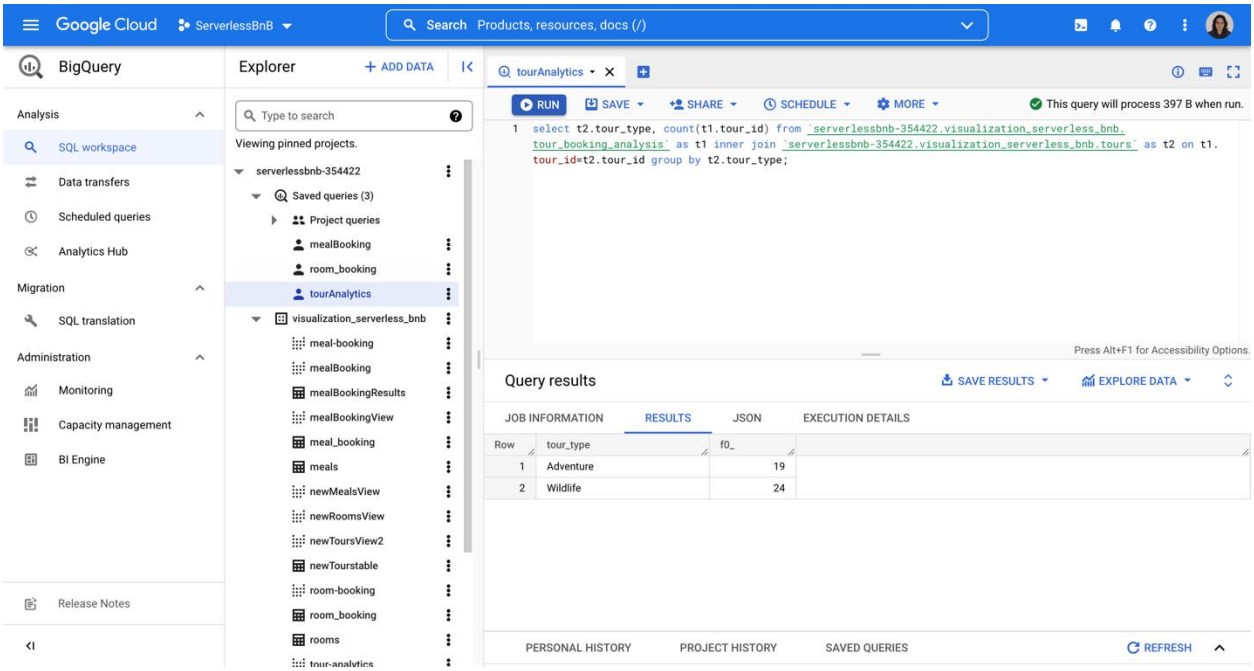

Figure 102: popular tours query in bigQuery.

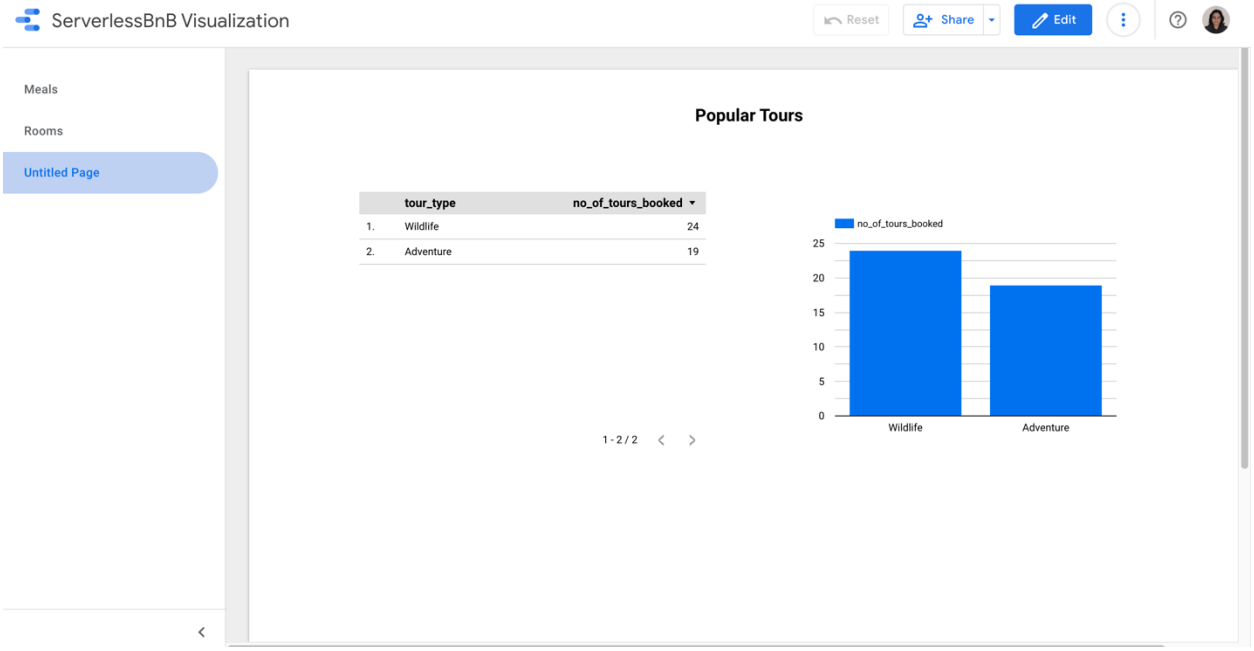

Figure 103: Popular Tours visualization.

2. Popular meals among the available meals – Veg and Non-Veg

   The *meals* collection in Firestore contains information about the different types of tours available. The *meal_booking* collection contains the booking information of all the tours. These two collections are exported from Firestore and imported into the *Serverlessbnb_visualizations* collection in BigQuery.

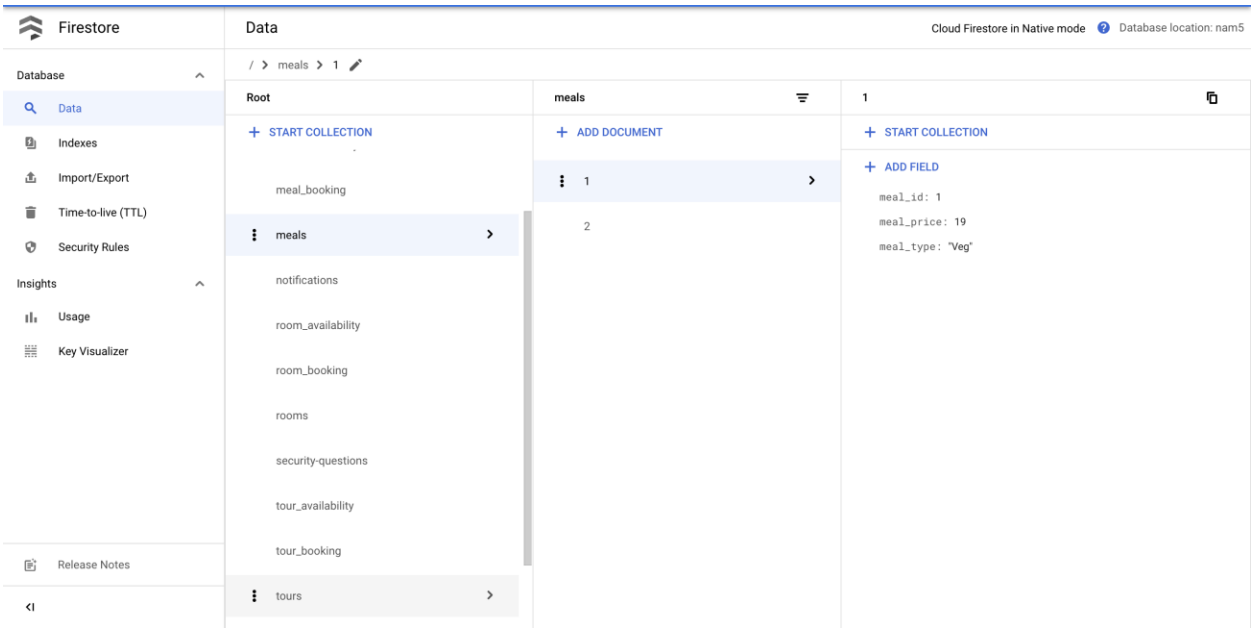

Figure 104: meals collection.

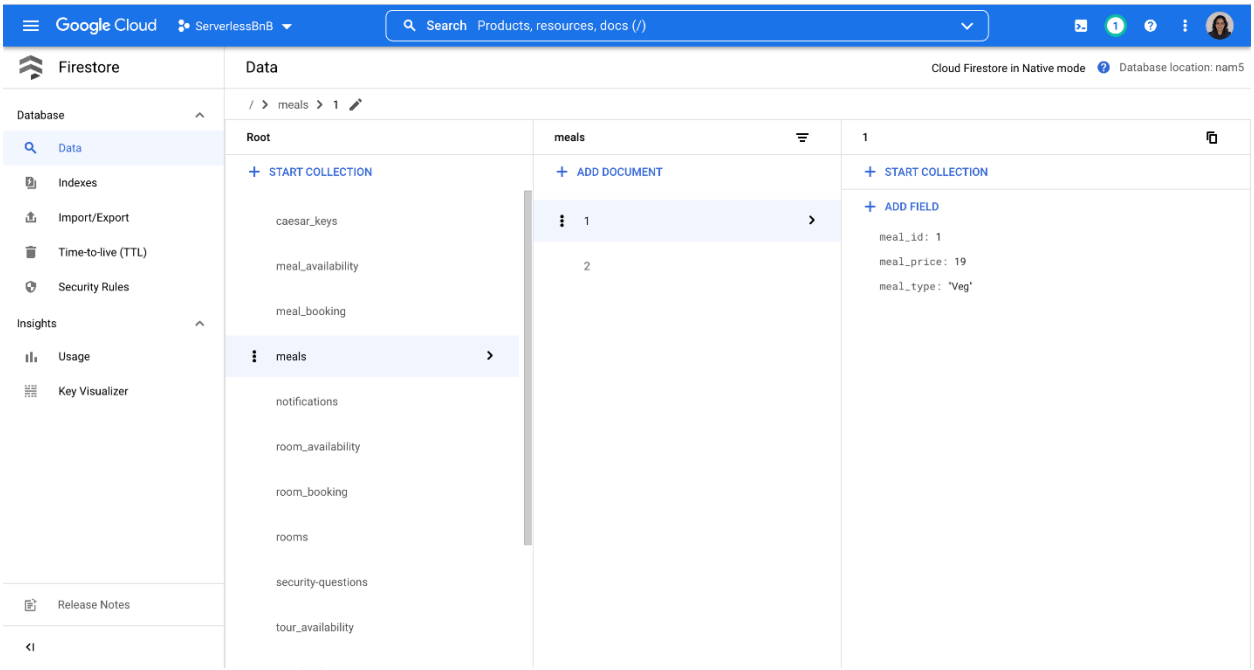

Figure 105: meal_booking collection.

The *meals* and *meal_booking* collections are imported in a temporary folder in BigQuery, where a join query was performed to calculate the count of type of meals opted by the customers so far. There is an option in a dashboard to visualize the data with Google Data Studio as seen in the following figure.

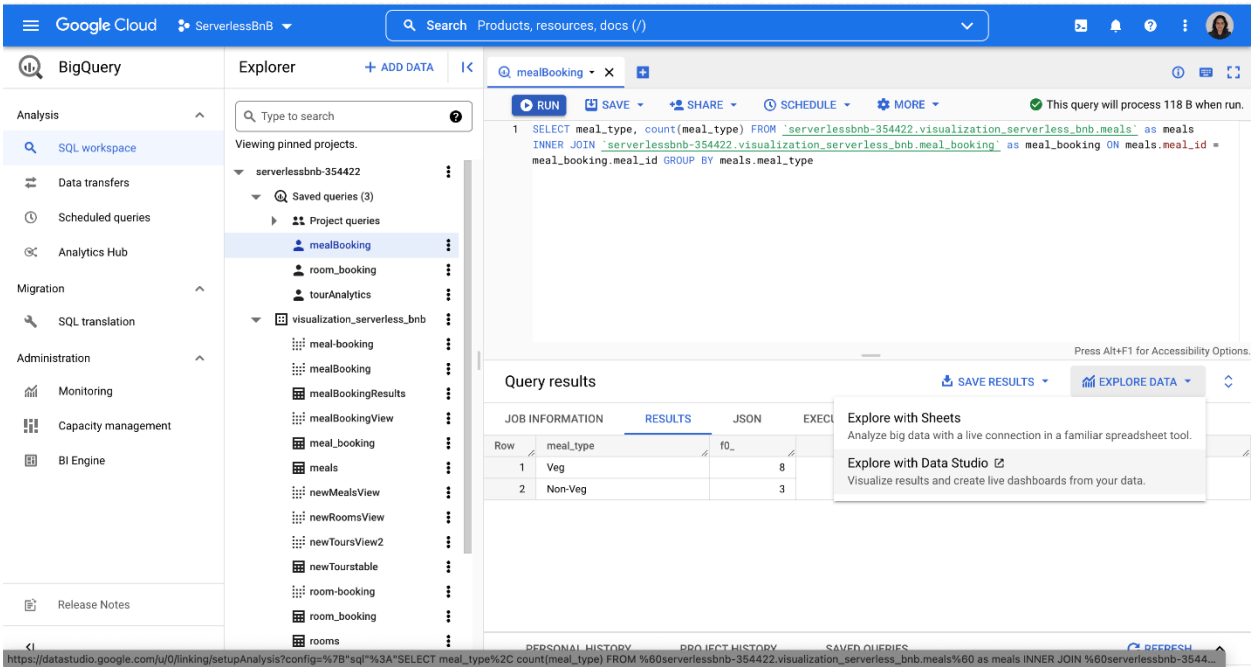

Figure 106: popular meals query in bigquery.

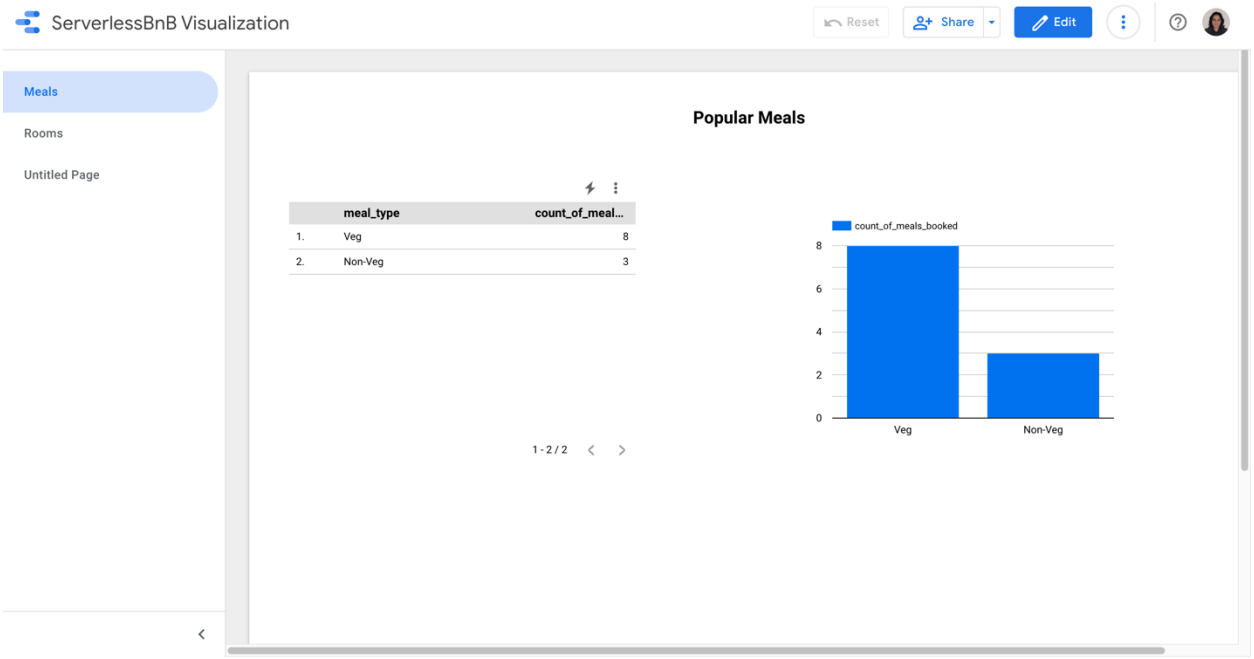

Figure 107: Popular meals visualization.

3. Popular rooms among the room types – AC, Non-AC, Deluxe, Suite

   The *rooms* collection in Firestore contains information about the different types of tours available. The *room_booking* collection contains the booking information of all the tours. These two collections are exported from Firestore and imported into the *Serverlessbnb_visualizations* collection in BigQuery.

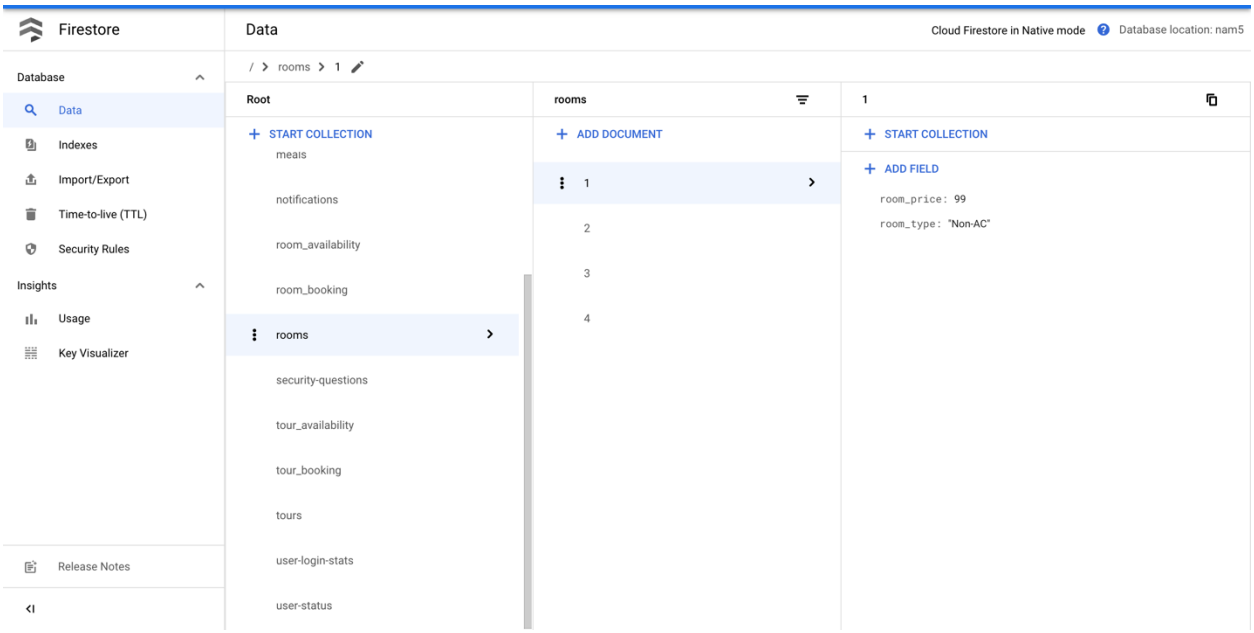

Figure 108: rooms collection

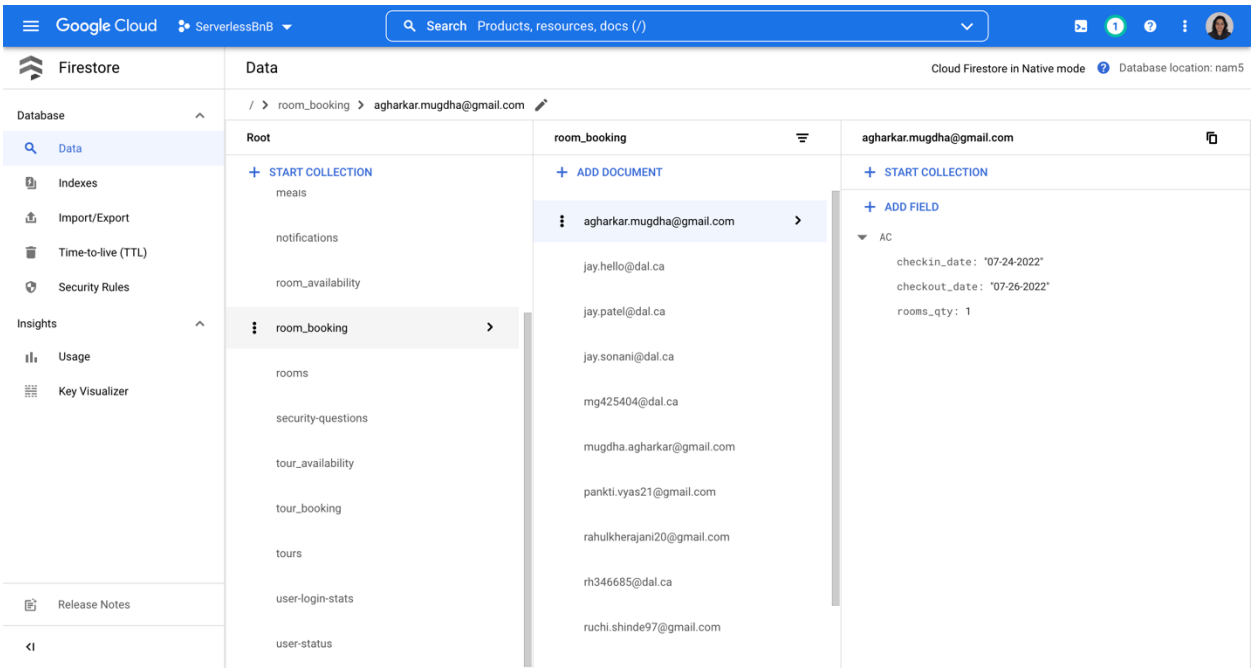

Figure 109: room_booking collection

The *rooms* and *room_booking* collections are imported in a temporary folder in BigQuery, where a join query was performed to calculate the count of type of meals opted by the customers so far. There is an option in a dashboard to visualize the data with Google Data Studio as seen in the following figure.

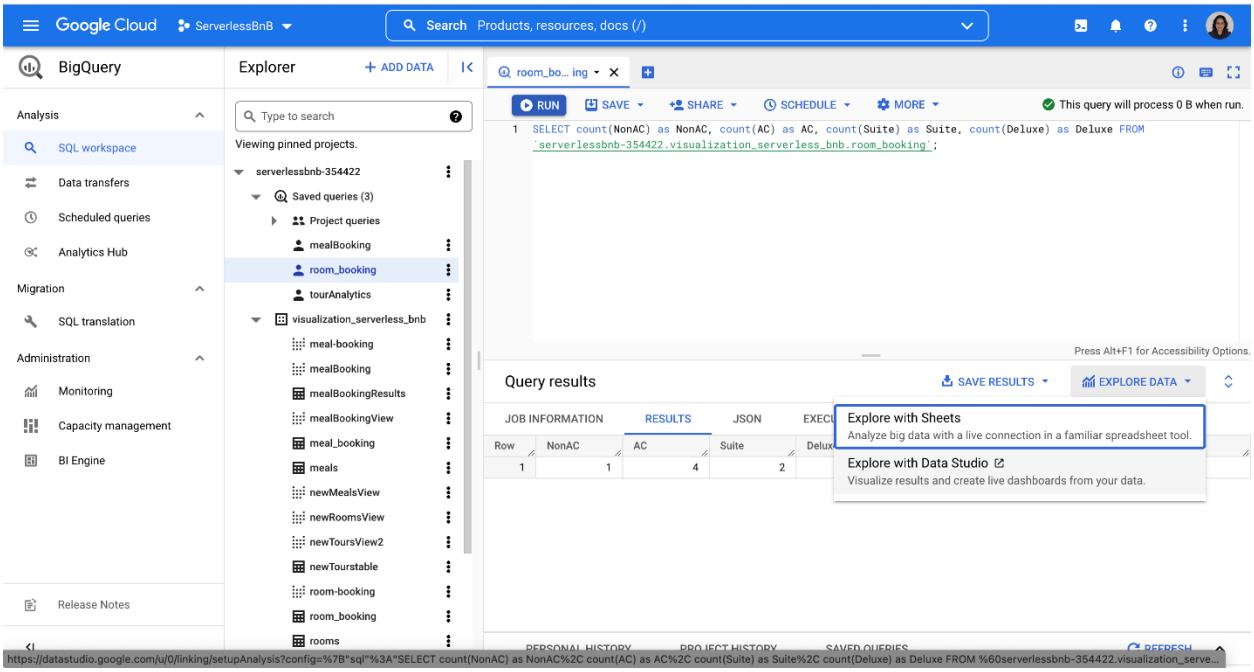

Figure 110: popular room types query in BigQuery

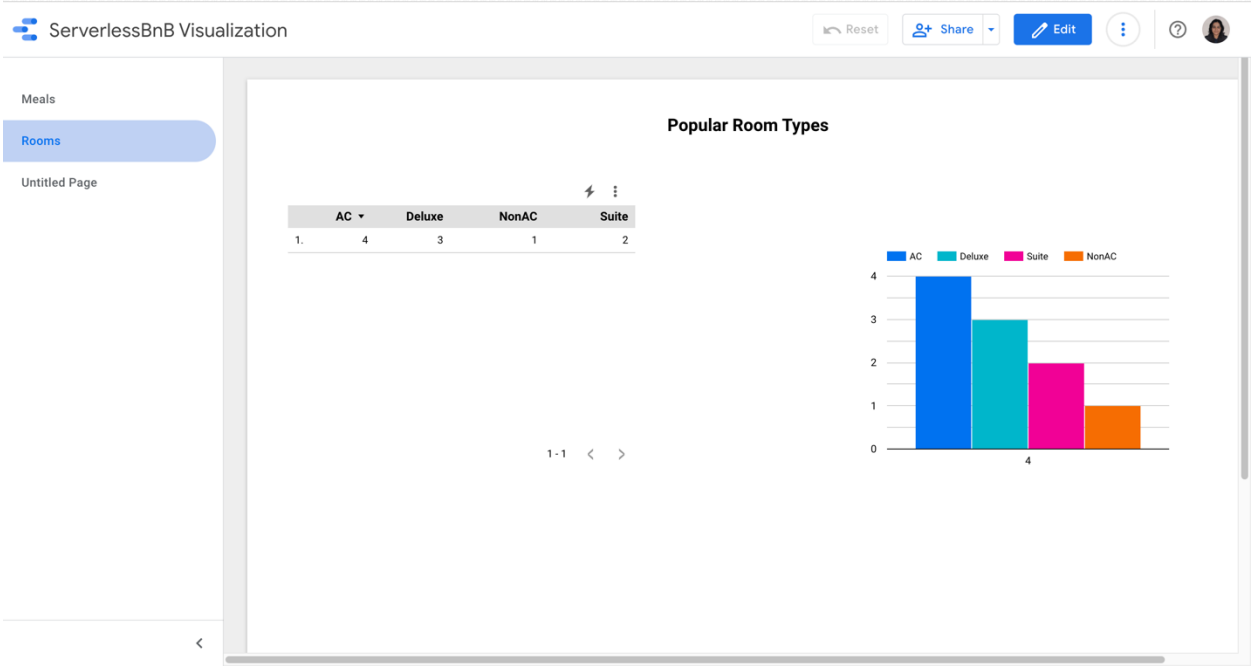

Figure 111: Popular room types visualization.

All these visualizations were stored in one report, where we get an option to embed the report in a webpage. This link was copied and added in an iframe on the front-end, so that it could be rendered directly in a component on the Visualizations page.

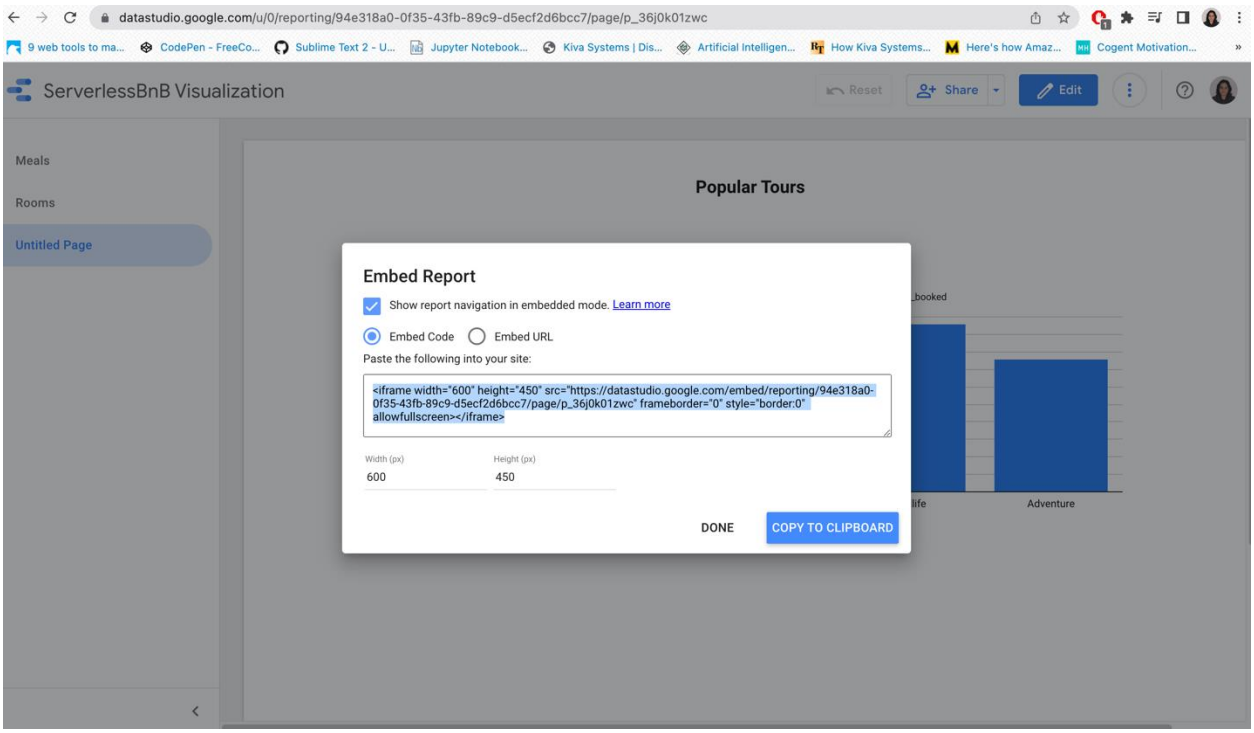

Figure 112: Embedding the report.

VisualizationsPage.jsx | Header.jsx | main.py .../get-user-session-stats | LoginStatisticsPage.jsx 1

frontend > src > Pages > VisualizationsPage.jsx > ...

Rahul Kherajani, 4 days ago | 2 authors (Rahul Kherajani and others)

```jsx
import React from 'react';      You, 4 days ago • Integrate visualization on the front-end …
import Header from '../Components/Header';
import Footer from '../Components/Footer';

const VisualizationsPage = () => {
  return (
    <>
      <Header />
      <iframe
        title='visualizations'
        width='1200'
        height='400'
        src='https://datastudio.google.com/embed/reporting/94e318a0-0f35-43fb-89c9-d5ecf2d6bcc7/page/p
        frameBorder='0'
        style={{ border: 0 }}
        allowFullScreen
      ></iframe>
      <Footer />
    </>
  );
};

export default VisualizationsPage;
```

PROBLEMS 1 OUTPUT TERMINAL GITLENS DEBUG CONSOLE zsh

(base) mugdhaagharkar@Mugdhas-MacBook-Pro csci5410_group10 %

Figure 113: Front-end code for displaying visualizations.

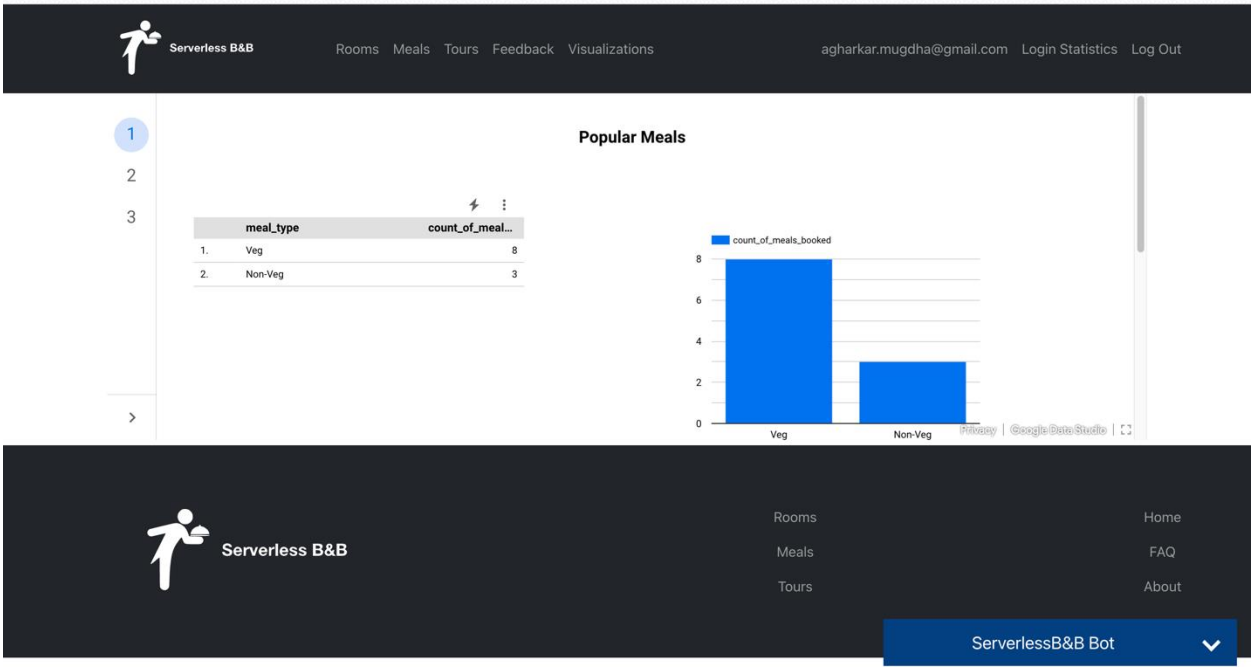

Figure 114: Visualization output page 1.

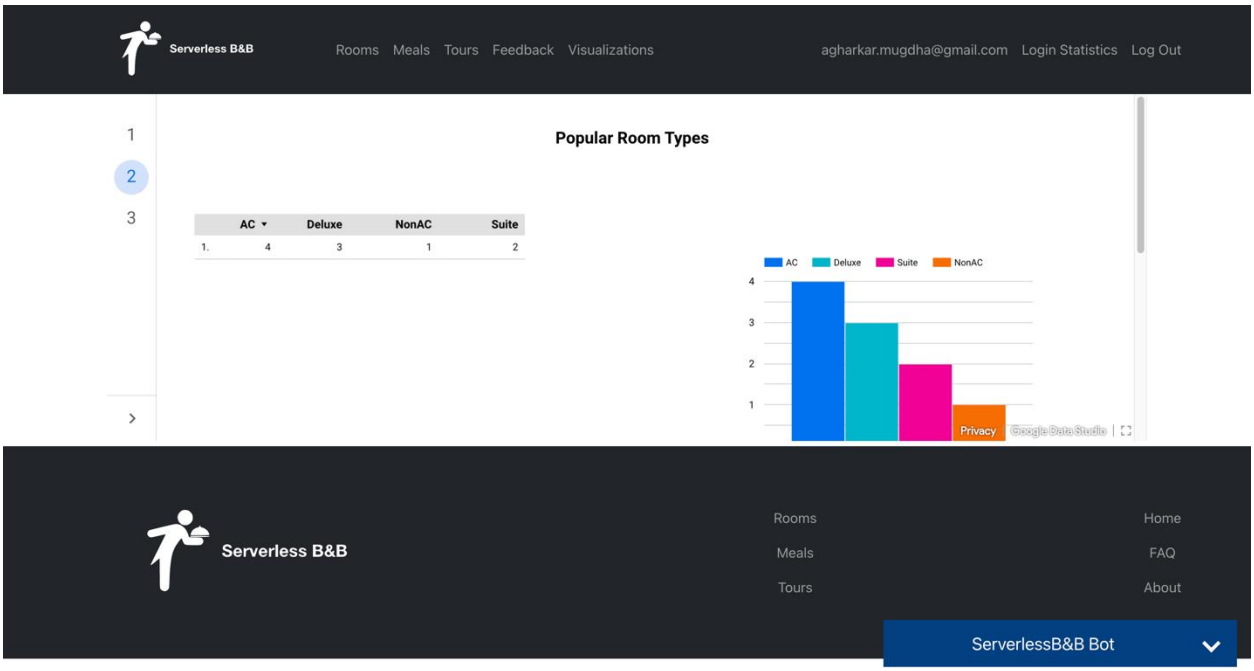

Figure 115: Visualization output page 2.

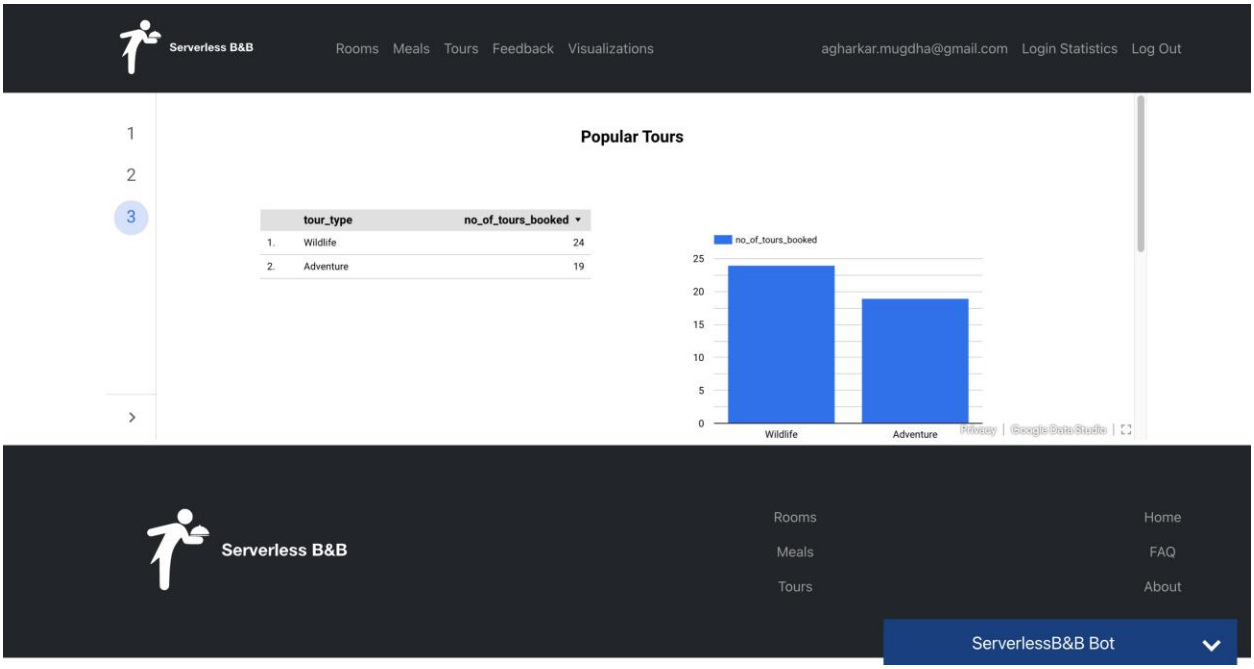

Figure 116: Visualization output page 3.

4. Project Architecture

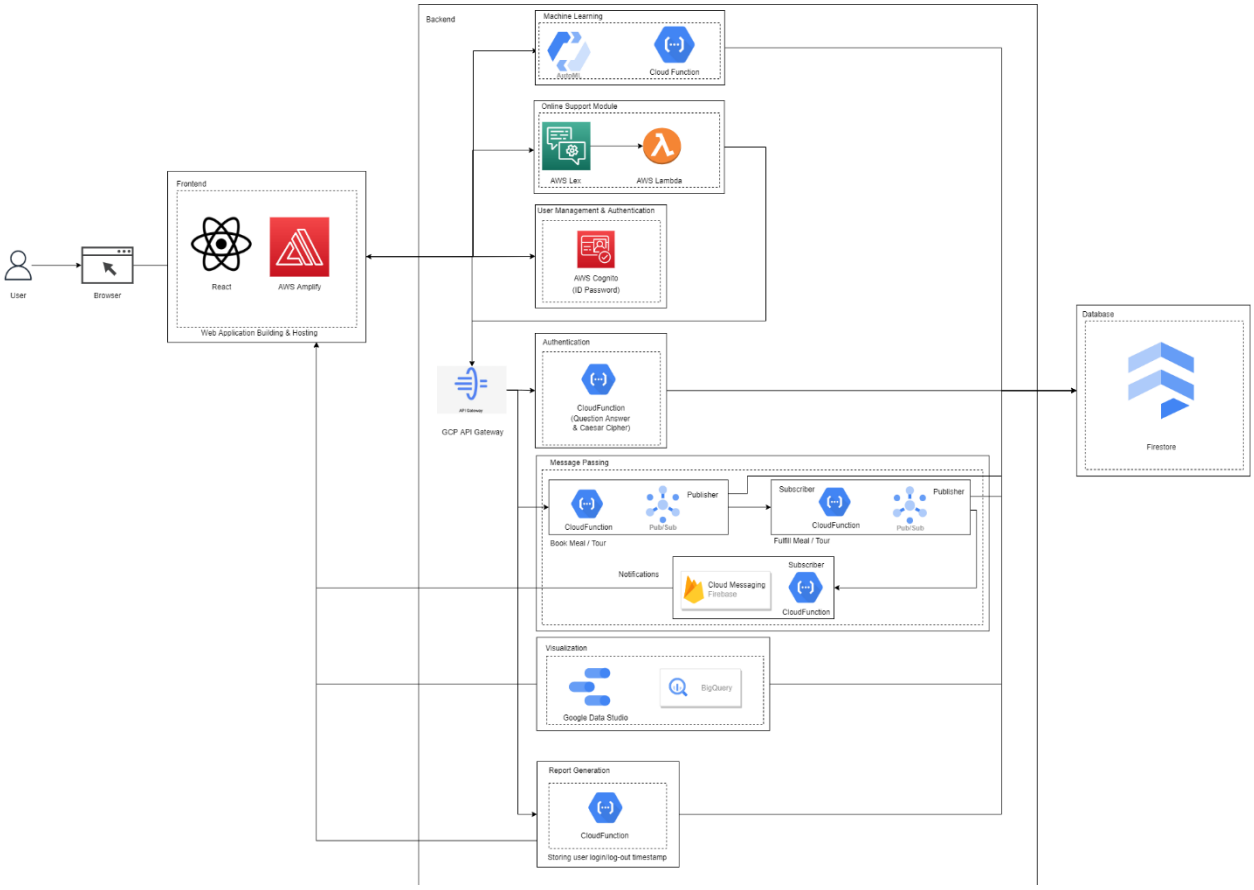

Figure 117: Architecture Diagram.

5. Flowchart diagram

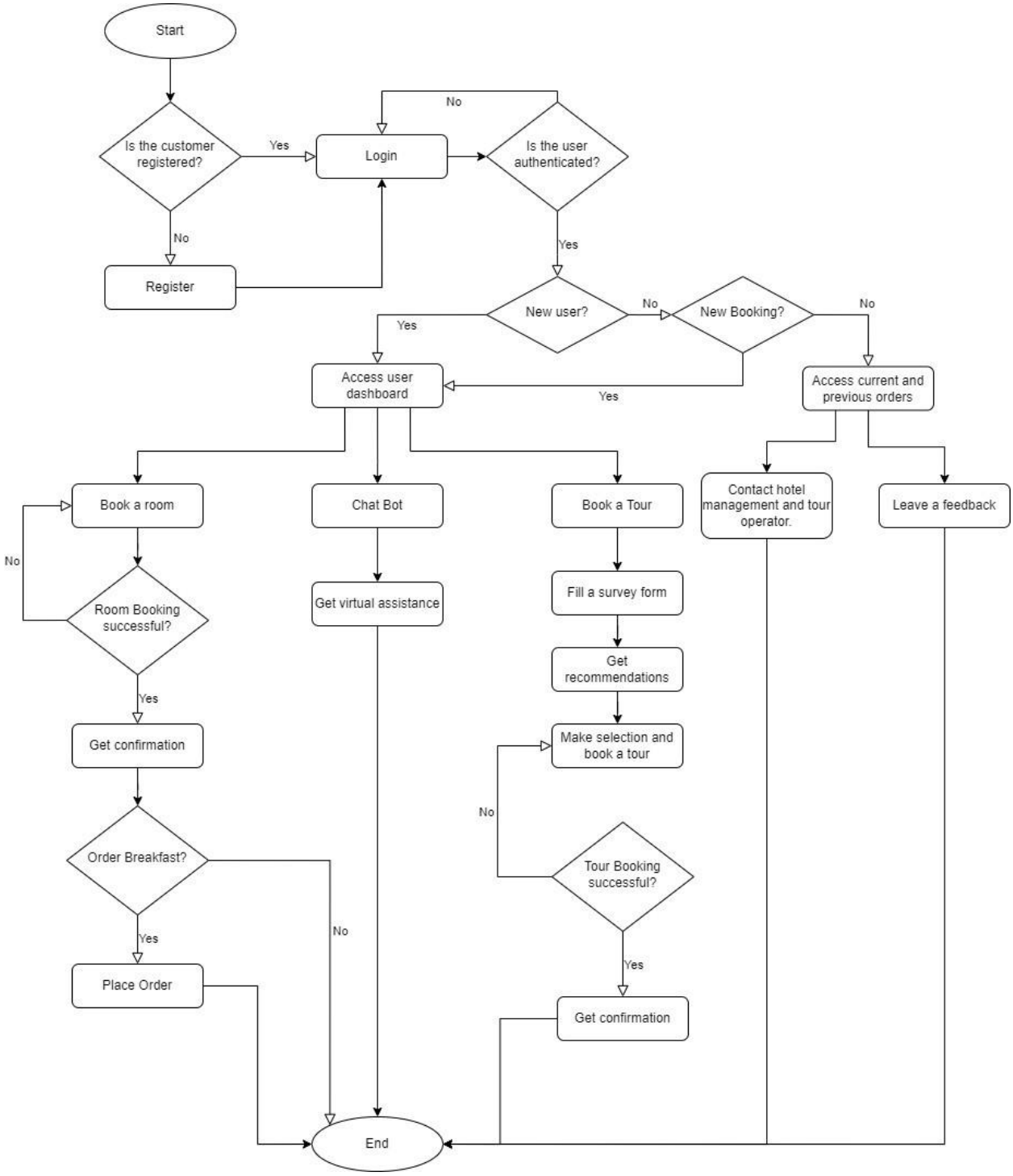

Figure 118: Flowchart Diagram.

6. Individual and Team contribution

Table 3: Team Contribution per module.

Module name	Team member
User Management Module	Pankti Vyas
User Authentication Module	Pankti Vyas (Login, Security Question Authentication, Logout, Session Management) Jay Sonani (Caesar-Cipher Authentication)
Online Virtual Assistance Module	Vivekkumar Patel
Message Passing Module	Rahul Kherajani
Machine Learning Module	Ridham Kathiriya
Web Application building and hosting	Frontend Development (Everyone) Frontend Deployment (Rahul Kherajani)
Visualization and Report Generation	Mugdha Agharkar
Services	Jay Sonani, Mugdha Agharkar (Room Booking) Rahul Kherajani (Tour booking, meal booking)
Reports and Documentations	Everyone
Demo video	Module-wise demo: Everyone Overall demo: Mugdha Agharkar Architectrue : Rahul Kherajani

7. Limitations

As of now our ML models cannot be trained continuously because it is too costly. Sentiment analysis model requires 21$ per train and package predictor requires 23$ credits per train.

Also, API gateway configured for our application is open which means, the requests received by gateway are not authenticated for the users signed up on AWS Cognito. This was particularly challenging since GCP API gateway can be secured using GCP IAM users but configuring it with AWS Cognito lies in future scope for our project.

8. Meeting logs

8.1 Introductory meeting

Date: 19th May 2022

Total Time: 25 Minutes

Mode: Online (Teams)

Agenda: Get to know each other's strengths and have an overview of project requirements.

Outcome: We learned about everyone's strengths and familiarized ourselves with all the required features.

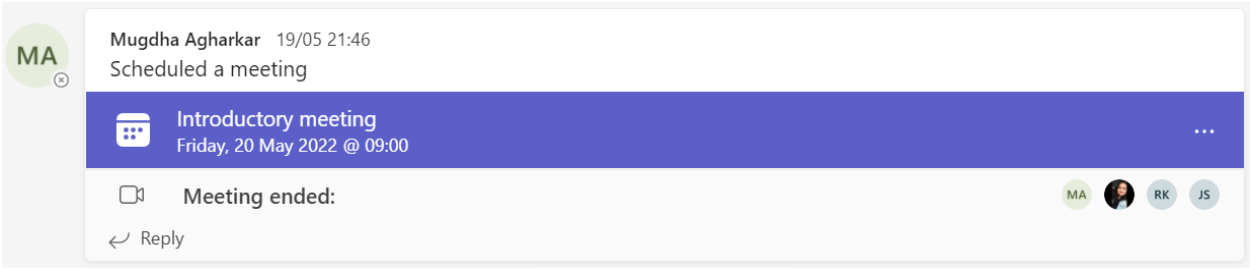

Figure 119: Meeting 1.

8.2 Finalize Tech stack

Date: 23rd May 2022

Total Time: 35 Minutes

Mode: Online (Teams)

Agenda: To decide project Tech stack

Outcome: We have decided to go with MERN stack.

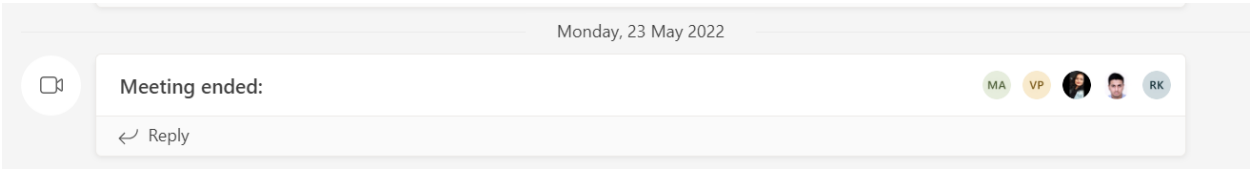

Figure 120: Meeting 2.

8.3 Feature distribution

Date: 30th May 2022

Total Time: 2 hours 30 Minutes

Mode: In person

Agenda: To divide design and development modules among group members

Outcome: Modules are equally distributed (Pankti: Module-1, Jay: Module-2, Vivek: Module-3, Rahul: Module-4, Ridham: Module-5, Mugdha: Module-7). Everyone will work on Module-6 and Module-8.

8.4 Conceptual report design

Date: 2nd Jun 2022

Total Time: 4 hours

Mode: In person (W.K Kellog Health Sciences Library)

Agenda: To prepare the first deliverable of our application

Outcome: Prepared and delivered the best project conception report.

8.5 Scrum Cycle 1

Date: 10th Jun 2022

Total Time: 2 hours

Mode: In person (W.K Kellog Health Sciences Library)

Agenda: To verify the progress and doubt solving

Outcome: Each member received more clarification on individual modules.

8.6 Scrum Cycle 2

Date: 17th Jun 2022

Total Time: 2 hours

Mode: In person (W.K Kellog Health Sciences Library)

Agenda: To decide the actual flow of application (How modules will interact with each other)

Outcome: We came up with the basic application flow chart.

8.7 Scrum Cycle 3

Date: 23rd Jun 2022

Total Time: 1 hours 22 minutes

Mode: Online

Agenda: To verify the progress and doubt solving

Outcome: Each member received more clarification on module interactions.

Figure 121: Scrum Cycle 3.

8.8 Project Design Document

Date: 4th July 2022

Total Time: 18 minutes

Mode: Online

Agenda: To prepare our second deliverable "The project design document"

Outcome: We divide the work among team members to complete the document.

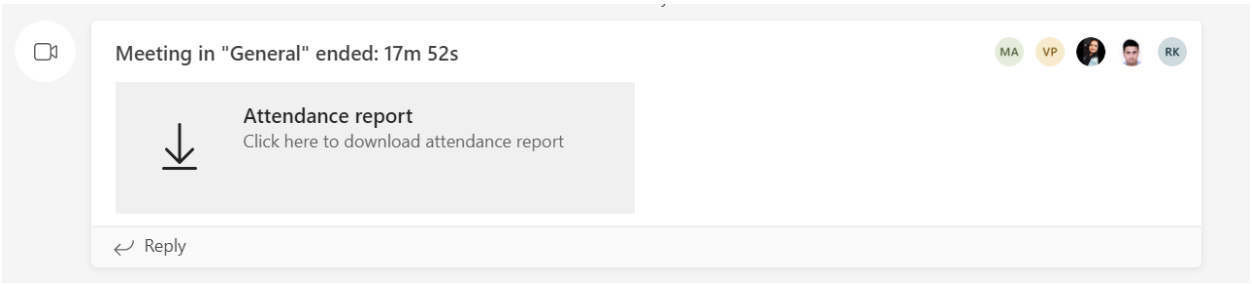

Figure 122: Project design meeting.

8.9 Scrum Cycle 4

Date: 10th July 2022

Total Time: 2 hours 22 minutes

Mode: In person (Kellog Library)

Agenda: To verify the progress and doubt solving

Outcome: Each member received more clarification on module interactions and integration.

8.10 Scrum Cycle 4

Date: 16th July 2022

Total Time: 2 hours

Mode: In person (Kellog Library)

Agenda: To complete the project

Outcome: Completed modules integrated successfully.

8.11 Project Demo

Date: 21st July 2022

Total Time: 30 minutes

Mode: Online (with professor)

Agenda: To answer the project specific questions

Outcome: provided satisfactory answers to the professor.

Figure 123: On a meeting with the professor.

9. References

[1] "HTTP CORS | Cloud Functions Documentation | Google Cloud", *Google Cloud*, 2022. [Online]. Available: https://cloud.google.com/functions/docs/samples/functions-http-cors#functions_http_cors-python [Accessed: 25- Jul- 2022]

[2] c. python, S. T. and D. Hampton, "check if a Firebase App is already initialized in python", *Stack Overflow*, 2022. [Online]. Available: https://stackoverflow.com/questions/44293241/check-if-a-firebase-app-is-already-initialized-in-python [Accessed: 25- Jul- 2022]

[3] "Lab 4-2: Caesar Cipher - Encrypting and Decrypting — CSP Python", *Teachen.info*, 2022. [Online]. Available: https://teachen.info/cspp/unit4/lab04-02.html [Accessed: 25- Jul- 2022]

[4] "Simple Caesar Cipher Python decryption function.", *Gist*, 2022. [Online]. Available: https://gist.github.com/AO8/3a89ba7c8f032c7a1ff505baa3ce970e [Accessed: 25- Jul- 2022]

[5] "Retrieving Data | Firebase Documentation", Firebase, 2022. [Online]. Available: https://firebase.google.com/docs/database/admin/retrieve-data#python [Accessed: 25- Jul- 2022]

[6] "Add data to Cloud Firestore | Firebase", Firebase, 2022. [Online]. Available: https://firebase.google.com/docs/firestore/manage-data/add-data#update_elements_in_an_array [Accessed: 25- Jul- 2022]

[7]"python list of dates between two dates Code Example", Codegrepper.com, 2022. [Online]. Available: https://www.codegrepper.com/code-examples/python/python+list+of+dates+between+two+dates [Accessed: 25- Jul- 2022]

[8] "Vertex AI documentation," Google Cloud. [Online]. Available: https://cloud.google.com/vertex-ai/docs [Accessed: 25-Jul-2022].

[9] "AWS Documentation," AWS Docs, 2022. [Online]. Available: https://docs.aws.amazon.com/lex/latest/dg/how-it-works.html [Accessed: 26- Jul- 2022].

[10] "Google Data Studio, " Google Data Studio. [Online]. Available: https://datastudio.google.com/u/0/ [Accessed: 25-Jul-2022].