

CSCI 5410 -Project Report

Team: **SApp_19**

(1) Hashik Donthineni - B00868314

(2) Fenil Nikeshkumar Shah - B00871894

(3) Pathik Kumar Patel – B00869765

(4) Dharaben Thakorbhai Gohil –B00884960

Table Of Contents

Problem Statement.....	4
Requirements.....	4
List of Features.....	4
Initial Feasibility Study	6
Sign-in, Sign-up and Multi-Factor Authentication (MFA)	7
Customer-Restaurant Instant Messaging	7
Data-Processing Module (WordCloud)	7
Menu Configuration.....	7
Chatbot for application navigation	8
Recipe Similarity score.....	8
Provide Feedback & Ratings	8
Visualization	8
Order Food.....	8
Final Feasibility Study	9
Sign-in, Sign-up and Multi-Factor Authentication (MFA)	9
Customer-Restaurant Instant Messaging	9
Data-Processing Module (WordCloud)	10
Menu Configuration.....	10
Chatbot for application navigation	11
Recipe Similarity score.....	11
Provide Feedback & Ratings	12
Visualization	12
Order Food.....	12
Initial Application Design	13
Final Application Design.....	14
Implementation Details	15
Sign-in, Sign-up and Multi-Factor Authentication (MFA)	15
Customer-Restaurant Instant Messaging	19
Data-Processing Module (WordCloud)	26
Menu Configuration.....	28
Chatbot for application navigation	31
Recipe Similarity score.....	34

Provide Feedback & Ratings	36
Restaurant Menu	37
Order Food	38
Data Visualization	40
Integration Details	41
Deployment Details	41
Testing Details.....	42
Sign-in, Sign-up and Multi-Factor Authentication (MFA)	42
Sign-up Testing.....	42
Sign-in Testing.....	42
MFA Testing	43
Chatbot for application navigation	43
Recipe Similarity score.....	45
Feedback and Ratings	45
Customer-Restaurant Instant Messaging	46
Restaurant Menu	47
Order Food	48
Visualization	49
Meeting Details.....	49
Team Information	51
Team Strengths:	51
Team Weakness:	51
Challenges faced	51
References	53

Problem Statement

The primary objective of this project is to build cloud plumbing system, where an application will be designed using serverless technologies to process data (more specifically large-scale data). This is like building a water plumbing system, which is built once with plenty of interconnected pieces, and used many times. It does not require any specialist for the operation. In this project, you will be building a “cloud data plumbing system”, which could be used by many clients to process their data. Different backend services, and simple front-end application would be used to build the system.

Requirements

We will be creating a Food Delivery System using serverless components by using the concept of a “Cloud Data plumbing system” to process the data. The application will have three user roles: Admin, Customer, Restaurant management. The customers can place the order, track the orders, and provide ratings & feedbacks to the restaurants, and can use the discount coupons offered by the restaurants. Also, they can interact with the chatbot to navigate through the application and, they can chat in real-time with the restaurant representatives and even the managers in case of escalation.

For the restaurants’ management team, they can add/modify the menu and add discount coupons, check the similarity score for their recipes, recommend food items to the customers and keep track of their revenue using the reporting and visualization module.

List of Features

- User sign-up and sign-in
 - o Users can sign up as a restaurant or as a customer.
 - o Sign-up, sign-in validation is done using Google SSO.
 - o **AWS Cognito** is used for sign-up, sign-in, and user management.
 - o **Technologies used:** React(front-end), AWS Cognito, Google SSO.
- Multifactor authentication
 - o Users can use app based TOTP for multifactor authentication.
 - o MFA of a user is done with TOTP from **AWS Cognito**
 - o **Technologies used:** React(front-end), AWS Cognito
- Browse the restaurants and place the food order.
 - o The customer will browse the restaurants and food items from the menu and after that, they can place the food order simply by selecting the items of their choice and the quantity of the items.
 - o **Technologies used:** The **React** framework will render the restaurant and food details. The order details of the customer will be stored in the **AWS RDS database**.
- Chatbot for application navigation
 - o The User can ask any queries about application navigation and tracking the other features. Users can communicate with chatbots.
 - o **Technologies used:** **AWS Lex** is useful to create and build conversational chatbots quickly. For that one must specify the conversational flow to create a bot.

- Customer Support using instant messaging.
 - o The customers would be able to contact the restaurant customer service representative in case of any query related to the service or inquiry.
 - o **Technologies used:** A Containerized application using **Google Cloud PubSub** deployed on **CloudRun** will be used in the backend to provide the instant messaging service.
- Recipe Similarity score.
 - o Each restaurant can upload its famous recipe on the app. The app scans the recipe, gives a tag, and shows the similarity score with other recipes in the app.
 - o **Technology Used:** An NLP-based machine learning model trained using **GCP AutoML** service will be used to check the similarity score of the recipe uploaded.
- Tracking the order delivery:
 - o Customers will be provided to track their order delivery status right from placing the order till it is delivered.
 - o **Technologies used:** For this, the delivery status will be updated through API and the status of the delivery will be maintained into the **AWS RDS MySQL** database.
- Provide Feedback & Ratings
 - o Customers will be allowed to provide ratings and feedback on the food items.
 - o **Technologies used:** The ratings and customer feedback would be stored and managed into the AWS RDS database and will be fed to the data processing module which uses **Amazon QuickSight** service to create a word cloud based on the data.
- Menu
 - o Users can go through the menu of the restaurants. In the menu, one can see the dish's name and its price.
 - o **Technologies used:** For Menu, the AWS RDS SQL instance will be used.
- Menu Configuration
 - o Restaurant team can go add, edit and delete food-items from their menu, which can be seen by the customers. In the menu-item, restaurant team can add name, ingredients of the dish and price.
 - o **Technologies used:** For Menu, the AWS RDS SQL instance will be used.
- Visualizations
 - o Customers can see visualizations of their spending based on the food item categories and restaurants.
 - o **Technologies used:** For the visualizations, we'll be using **Google Data Studio** and the data will be fed from the **AWS RDS SQL** instance.
- Food-Cloud
 - o Based on the ratings provided by the customers on food items, the data will be fed to a service that will create a word cloud based on the ratings, and from it, the restaurant owners can get the cumulative ratings provided by the customers on various food items.
 - o **Technologies used:** **AWS Comprehend** will be used for this, and the service will be deployed and managed on Amazon ECS.

- Report Generation Module
 - o System Admin will be provided with the functionality to create the reports related to the usage of the services, user statics, and billing information.
 - o For the above factors, the visualizations will be provided to the admin.
 - o **Technologies used:** For this feature, services such as **Google data studio** will be used.

Initial Feasibility Study

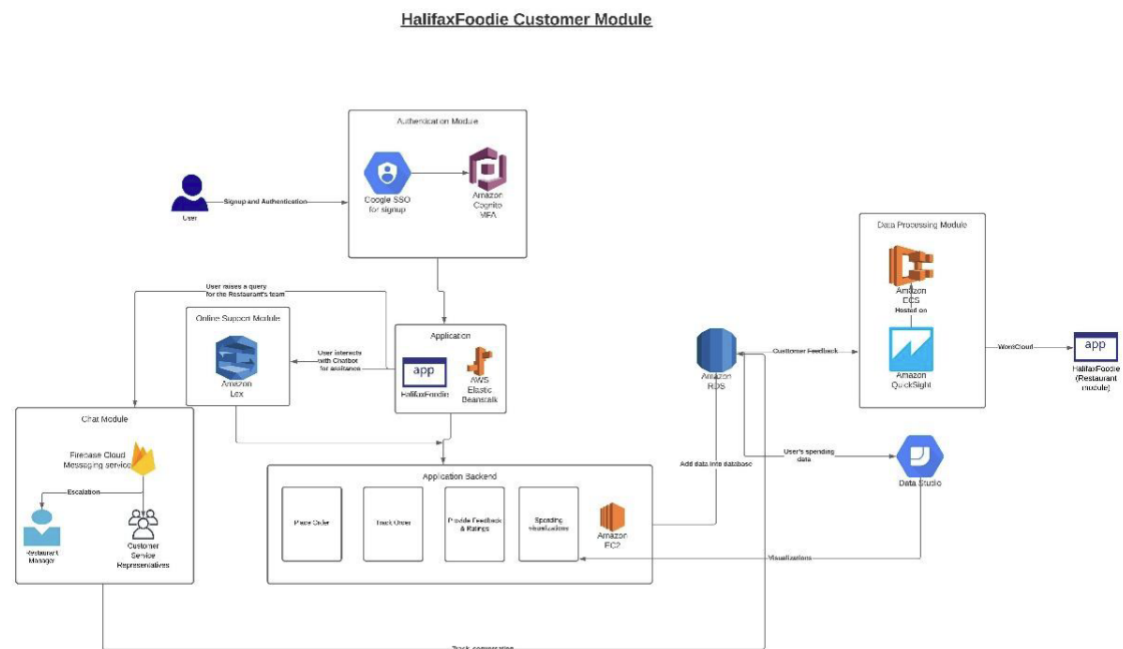


Figure1: Initial Feasibility Study – Customer Application Design [1]

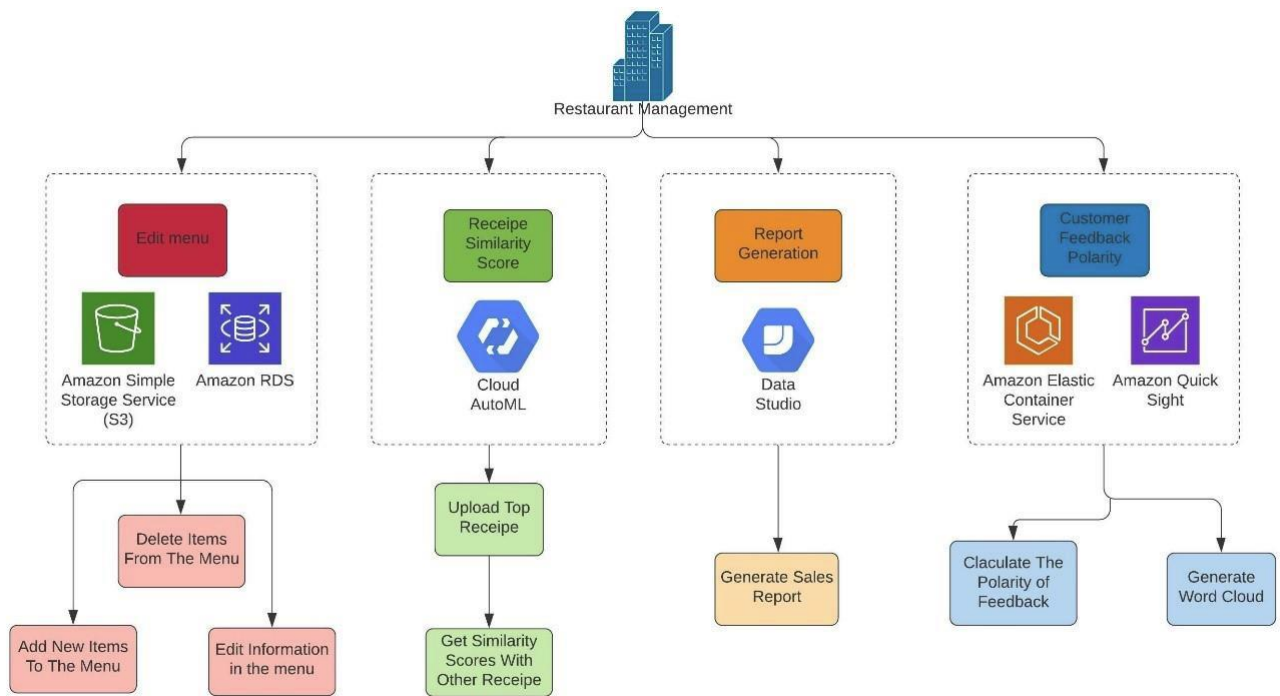


Figure 2: Initial Feasibility Study – Restaurant Application Design [1]

Sign-in, Sign-up and Multi-Factor Authentication (MFA)

For the initial feasibility for our report, we planned on using the Google single sign-on as authentication and as a sign-up method, after which it will be redirected to Cognito for the MFA question and answer or TOTP. This will all be done with the help of AWS Cognito-hosted UI.

Customer-Restaurant Instant Messaging

For the initial feasibility study, we planned to use Firebase Cloud Messaging service in the backend to provide the instant messaging service to the customers in order to contact the restaurant.

Data-Processing Module (WordCloud)

In the initial feasibility study, Amazon QuickSight became our choice in order to create a WordCloud visualization based on customer feedbacks for a particular food item to be shown while the item is added to the cart for ordering.

Menu Configuration

In the initial feasibility study, our group decided to use Lambda functions as the backend to be accessed from the frontend via API gateway. We decided to create different API methods and Lambda functions to perform the CRUD operations on the menu-item.

Chatbot for application navigation

In the initial feasibility study, our group decided to use Amazon Lex for chatbot. The customer will interact with the chatbot for general queries. In case of escalation the customer will be redirected to the restaurant staff and can chat with them.

Recipe Similarity score

In the initial feasibility study, our group has decided to use Google AutoML for prediction of document similarity. The customer will post its recipe and the frontend will send the data to Google AutoML for prediction. After prediction the AutoML will send data back to frontend and the restaurant staff can see the score.

Provide Feedback & Ratings

In the initial feasibility study, we decided that we will take customer input for the orders and food items of the restaurant and will store that data into an RDS instance on Amazon. That data will be used to for data processing and creating word cloud.

Visualization

In the initial feasibility study, we decided that the customers can see their spending on restaurants and foods in a graphical format. So, user just must enter select the option of visualization and the graphs will be appeared.

Order Food

In the initial feasibility study, after selecting the food, customer can place the order. Before confirming the order user has to enter the delivery details.

Final Feasibility Study

Sign-in, Sign-up and Multi-Factor Authentication (MFA)

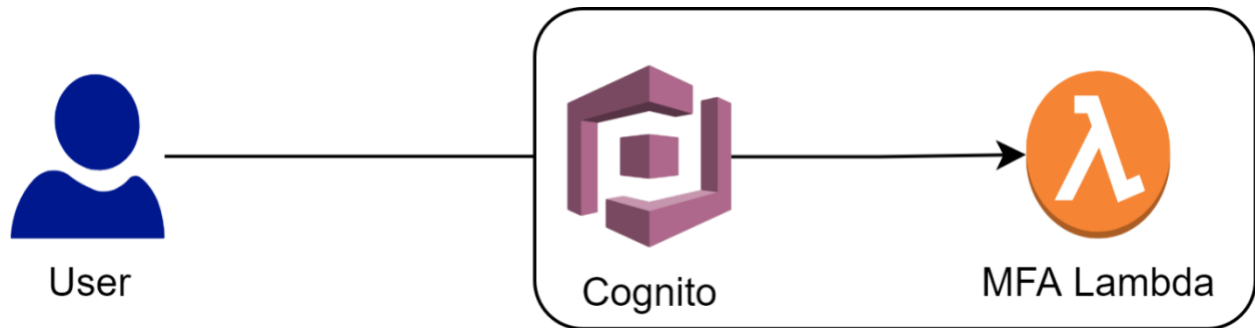


Figure 3: Authentication and MFA module [1]

For the final feasibility, we figured that it is not an optimal idea to use Cognito just for MFA as it gives a full set of tools and using it just for the sake of MFA destroys its purpose. Keeping that in mind, we changed the architecture to use Cognito as the authentication engine, this also allows us to secure the APIs using Cognito as the authentication engine in the API Gateway.

After the authentication with Cognito is successful the MFA will be triggered.

Customer-Restaurant Instant Messaging

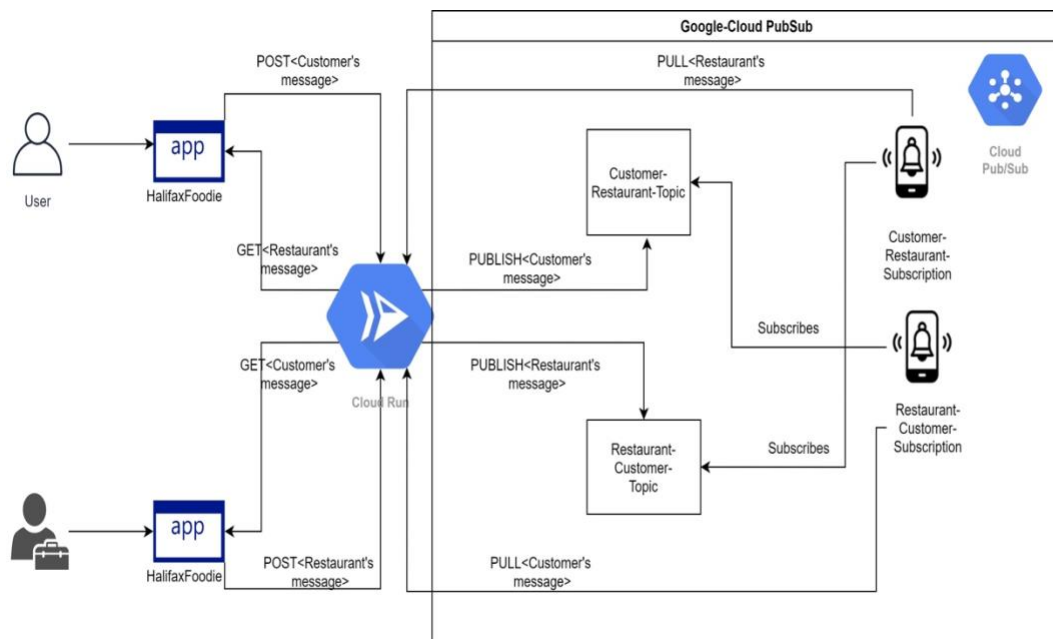


Figure 4: Final Feasibility Study For Instant Messaging Feature Design [1]

After the mid-term review, the main goal of the chatting service was to use a Publisher-Subscriber module and hence, as the Firebase Instant messaging handles the backend on its own, we figured that Firebase was not the good choice and hence, decided to use Google Cloud PubSub module as the backend and deploy it on the CloudRun.

So, now in the final implementation, the front-end application calls the APIs of the backend application which implements the Google Cloud PubSub APIs, deployed on the CloudRun for providing the instant messaging functionality to the customer in order to contact the Restaurant team.

Data-Processing Module (WordCloud)

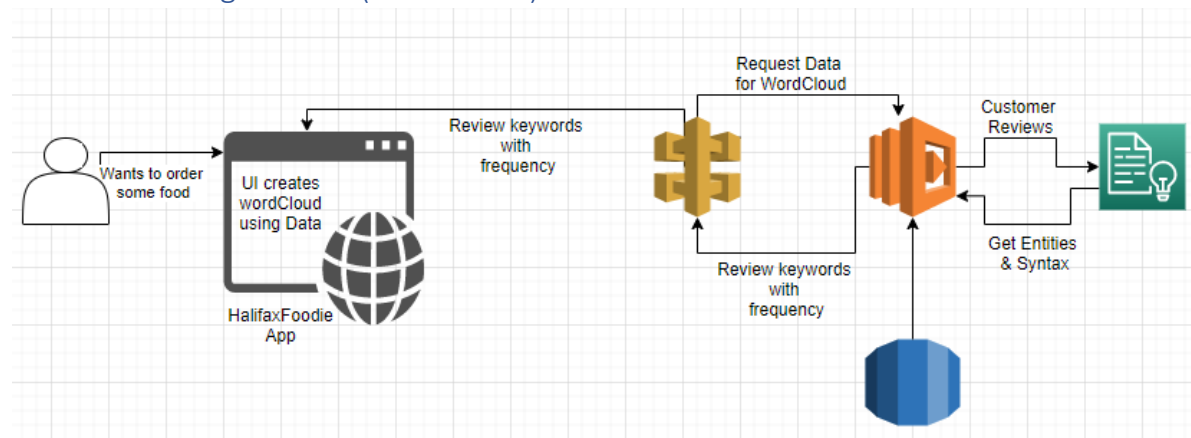


Figure 5: Final Feasibility Study For Data-Processing (WordCloud)[1]

During the final designing of this feature, we figured that access to Amazon QuickSight in the Educate Accounts is not provided and hence, we couldn't implement this feature using QuickSight service.

Hence, an alternative approach to this was followed. We decided to use Lambda Function to fetch and process the customer feedbacks from the mySQL database and feed it to the Amazon Comprehend service to detect the entities (uppercase named entities) and syntax (adjectives that describes the food-item) from customers' feedback statements and then calculate the frequencies of the words to create an array of a key-value pairs with entities as the key and their frequency as values and passed to the front-end to create a WordCloud.

We have used "Random Food Adjective Generator" [2] to create food-item reviews in order to have more data for processing.

Menu Configuration

Final feasibility study for this feature remained the same as the initial study as to our group, using Lambda functions via API Gateway seemed the optimal solution.

We have used TheMealDB.com [3] to create a food item dataset with ingredients.

Chatbot for application navigation

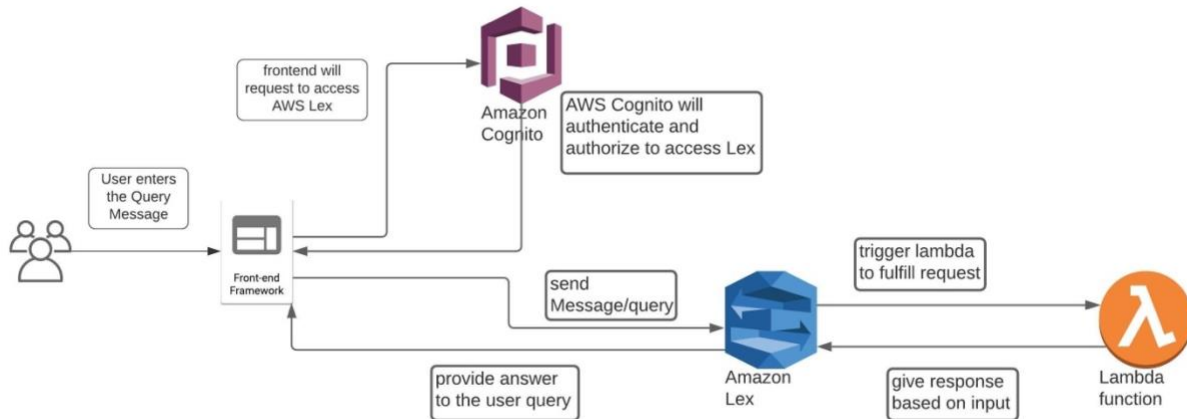


Figure 6: Final Design For Chatbot module [1]

In the Final Architecture, frontend will directly connect with the AWS Lex chatbot using the react chatbot library. AWS Lex will call relevant Lambda function to fulfill the customer request.

Recipe Similarity score

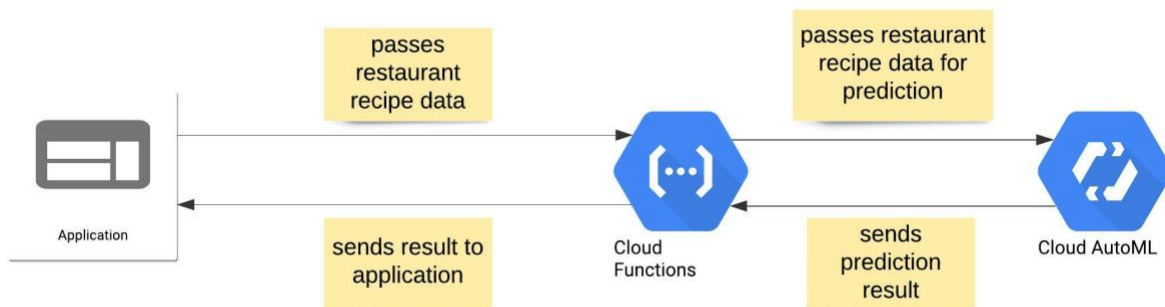


Figure 7: Final Design For Recipe Similarity Score [1]

In the final feasibility we are not accessing the Google AutoML directly. Instead, first the frontend send the recipe data to the Google Cloud Functions and Cloud Functions will make an api call to the AutoML model. After prediction the AutoML will pass prediction result to the cloud functions and cloud functions will sent it back to the frontend. So, the user can have the prediction result. The reason for using the cloud function is we cannot pass data directly to AutoML for prediction. we need to have authorization to the model. So, the cloud function is authorized to access the trained model in the AutoML.

Provide Feedback & Ratings

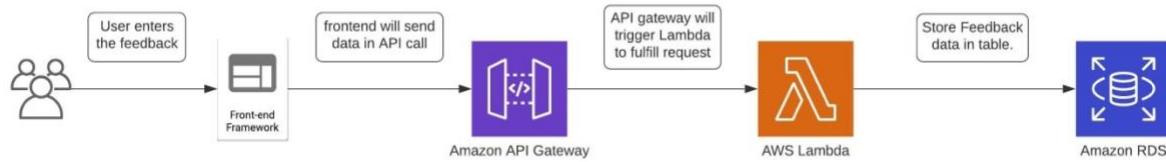


Figure 8: Final Design Feedback and Ratings module [1]

In the final feasibility. The frontend will make an API call to AWS API gateway. API gateway will trigger the backend Lambda function and store the feedback data in the RDS instance.

Visualization

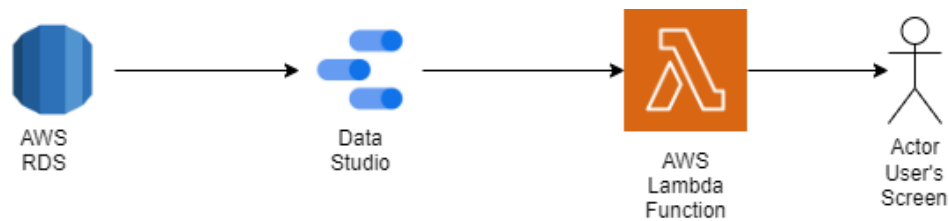


Figure 9: Final Design Visualization module [1]

Final feasibility study for this feature remained the same as the initial study. Data studio is better option for representing the data in the graphical format.

Order Food

In the final feasibility study, the process of ordering the food is remains same as the initial study.

Initial Application Design

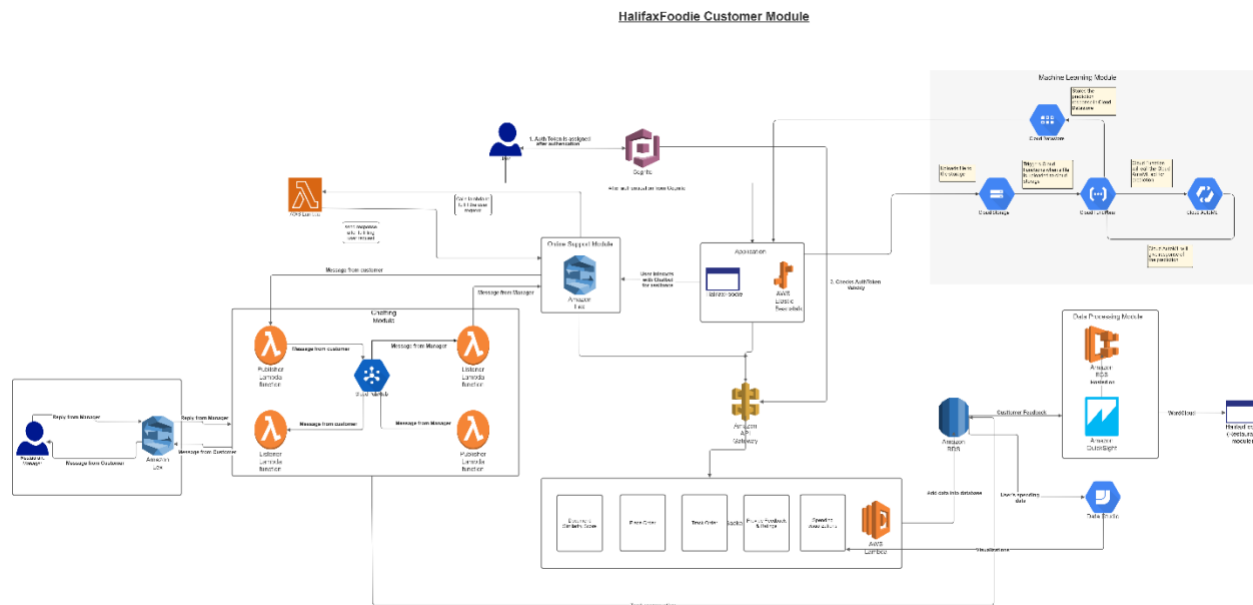


Figure 10: Initial Application Architecture [1]

Initially, we designed our application with the frontend consisting of a React application calling different APIs hosted via API Gateway with Lambda functions acting as our backend for CRUD operations along with other different AWS services. We planned to host our frontend application on AWS Beanstalk and the event-driven bridge was constructed using Lambda functions triggered via API Gateway for most of the features. For instant messaging functionality, we initially thought to use combination of Lambda functions and Firebase Instant Messaging service and Amazon QuickSight for data processing module. For Recipe similarity score feature, we planned to use a combination of Google Cloud services such as Cloud Functions, Cloud AutoML, Cloud Datastore and Cloud Storage.

Final Application Design

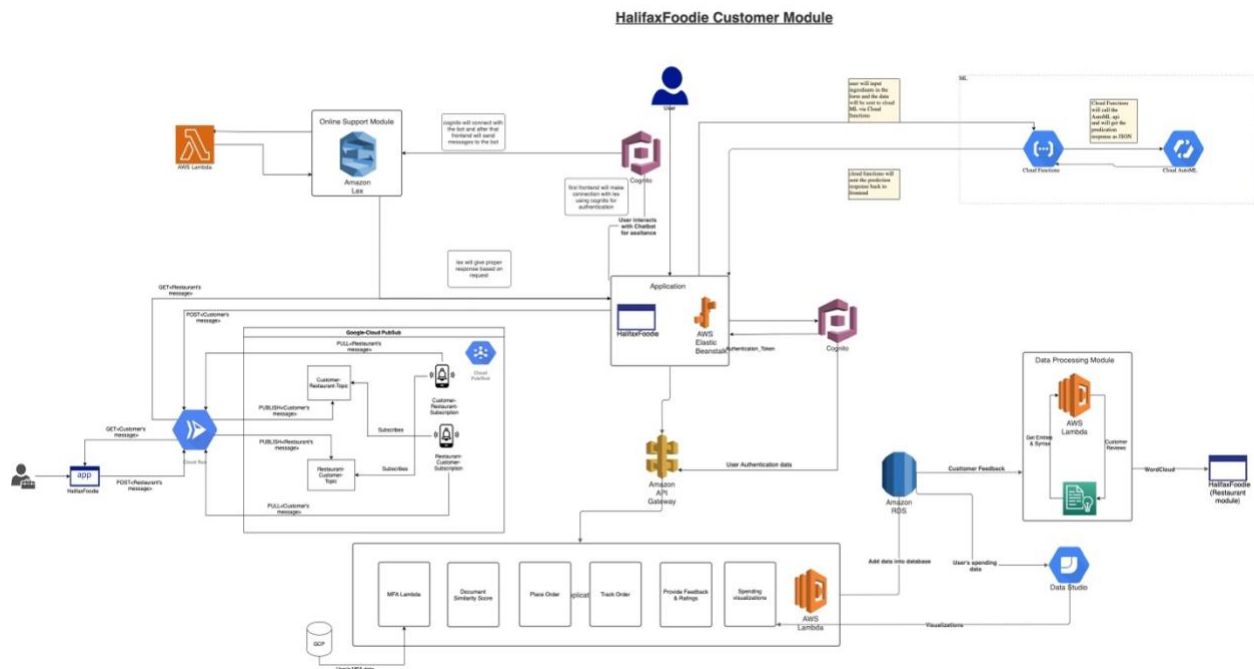


Figure 11: Final Application Architecture [1]

After some more brainstorming and review on the initial design, we explored the selected services and found that some services we choose for implementation either have some accessibility issues or are not serverless such as access rights for Amazon QuickSight are not included in the AWS Educate accounts and Firebase Instant Messaging is not a Publisher-Subscriber model. So, we changed our design for some of the features and came up with the final design as shown above.

In the final design, we changed the flow of authentication module where earlier we planned on using the Google single sign-on as authentication and as a sign-up method and AWS Cognito for multi-factor authentication. But, using Cognito just for authentication was not the optimal solution as it gives a full set of tools, and using it just for the sake of MFA destroys its purpose. So, we changed the architecture to use Cognito as the authentication engine.

In addition to that, we dropped the idea of using Firebase Instant Messaging service and instead planned to use Google Cloud PubSub as a publisher-subscriber service for the instant messaging service by creating a backend-application to be hosted on Google Cloud Run that implements PubSub API for publishing and fetching the messages from the queue. So, here the backend application hosted on Google Cloud Run acted as an event-driven bridge between our frontend and backend.

Furthermore, we changed the implementation of QuickSight service for data processing module due to accessibility issues and instead decided to use Lambda function to fetch and process data along with Amazon Comprehend for detecting entities and syntax from the customer feedbacks.

The last thing that we changed is the design of the machine learning service for recipe similarity score. Earlier we planned to use bunch of google services in order to implement this feature, where some

services were unnecessary. Hence, we decided to use only Cloud Functions and Cloud AutoML to implement this feature.

Apart from the above-mentioned changes, the design remained as we planned initially.

Implementation Details

The most important components of the code from each module will be discussed in the implementation details

Sign-in, Sign-up and Multi-Factor Authentication (MFA)

The user pool details:

```
const UserPool = new CognitoPool({
  UserPoolId: "us-east-1_X43BKHLC9",
  ClientId: "36hd2394u2lu7892qss27fb9r",
});

export default UserPool;
```

Code to sign-in the user, this set after the onClick or the submission of the form:

```
const user = new CognitoUser({
  Username: mailID,
  Pool: UserPool,
});

user.authenticateUser(
  new AuthenticationDetails({
    Username: mailID,
    Password: pass,
  }),
  {
    onSuccess: (d) => {
      localStorage.setItem(EMAIL_KEY, mailID);
      history.push("/mfa");
    },
    onFailure: (e) => {
      console.error("fail: ", e);
      setError(e.message);
    },
    newPasswordRequired: (d) => {
      console.log("needNewPass: ", d);
    },
  }
);
```

Sign-up of the user user the userpool:

```
var attributeRole = new AmazonCognitoIdentity.CognitoUserAttribute({
  Name: "profile",
  Value: accountType,
});

UserPool.signUp(email, pass, [attributeRole], null, (err, data) => {
  if (err) {
    setError(err.message);
  } else {
    setsubmitError(false);
    history.push("/login", { signup: true });
  }
});
console.log("Submit");
```

Fetching the MFA questions from the lambda:

```
axios
  .post(MFA_PATH, JSON.stringify({ email: email }), {
    headers: {
      "Content-Type": "application/json",
      AccessToken: idToken,
    },
  })
  .then((res) => {
    if (res.data.success) {
      setquestionData(res.data.questions);
    }
    setUserQuestionsStatus(res.data.success);
  })
  .catch((e) => {
    console.error(e.message);
  });
```

Writing the MFA question using the lambda which connects to GCP cloud SQL:

```
axios
  .post(MFA_PATH, body, {
    headers: {
      "Content-Type": "application/json",
      AccessToken: accessToken,
    },
  })
```



```

.then((res) => {
  if (res.data.success) {
    localStorage.setItem(MFA_KEY, true);
    history.push("/home");
  } else {
    setsubmitError(true);
    seterrMsg("MFA Updation failed.");
  }
})
.catch((e) => {
  console.error(e.message);
});

```

Fetching and updating the questions inside lambda:

```

const getQuestionsString = (email) => {
  const SELECT_QUERY = "SELECT questions FROM mfa.questions WHERE email=?;";

  return new Promise((resolve, reject) => {
    sqlCon.query(SELECT_QUERY, [email], (err, res, flds) => {
      if (err) {
        reject(err);
      } else {
        if (res.length > 0) {
          resolve(res[0]["questions"]);
        } else {
          reject();
        }
      }
    });
  });
};

const putQuestionsString = (email, questions) => {
  const INSERT_QUERY =
    "INSERT INTO mfa.questions (email, questions) VALUES (?, ?);";

  return new Promise((resolve, reject) => {
    sqlCon.query(INSERT_QUERY, [email, questions], (err, res, flds) => {
      if (err) {
        reject(err);
      } else {
        resolve();
      }
    });
  });
};

```

```
});  
};
```

The MFA lambda handler:

```
const handler = async (event) => {  
  console.log("Event Logged" + JSON.stringify(event));  
  let response = { success: false };  
  let qString;  
  
  if (event.questions) {  
    //Inserting questions  
    let strToInsert;  
    await jsonString(event.questions).then((data) => {  
      strToInsert = data;  
    });  
  
    await putQuestionsString(event.email, strToInsert)  
      .then(() => {  
        response = { success: true };  
      })  
      .catch((err) => {  
        console.error(err);  
        response = { success: false, message: err.message };  
      });  
  } else {  
    await getQuestionsString(event.email)  
      .then((data) => {  
        qString = data;  
        console.log("From Database:", qString);  
      })  
      .catch((e) => {});  
  }  
  if (qString) {  
    const jsonArray = stringToJson(qString);  
    response = { success: true, questions: jsonArray };  
  }  
  
  return response;  
};
```

Customer-Restaurant Instant Messaging

Backend Application: Initialization of variables and google-cloud pubsub api initialization with the projectId of the project created in the Google Cloud:

```
const {PubSub} = require('@google-cloud/pubsub');
require('dotenv').config();
const projectId = process.env.PROJECT_ID;
const express = require('express');
const bodyParser = require('body-parser');
const cors = require('cors');
const app = express();
const { response } = require('express');
let userTopicName="";
let resTopicName="";
let userSubscriptionName="";
let resSubscriptionName="";
let userTopic;
let resTopic;
let userSubscription;
let resSubscription;
let userMessages = [];
let resMessages = [];

app.enable('trust proxy');
app.use(cors());
app.use(express.json());
const pubsub = new PubSub({projectId});
...
const port = process.env.PORT || 3003;
app.listen(port,()=>{
  console.log("Running on port 3003");
})
```

POST API to create the PubSub topics and subscriptions on chat initiation from the Customer:

```
app.post('/initiateChat', async (req, res) => {
  try {
    const user = req.body.user;
    const restaurant = req.body.restaurant;
    let options = {
      "messageRetentionDuration": "11.0s",
    }
    //compute topic names for both
    userTopicName = user + "-" + restaurant + "-publisher";
    resTopicName = restaurant + "-" + user + "-publisher";
    //compute subscription names for both
    userSubscriptionName = user + "-" + restaurant + "-subscription";
    resSubscriptionName = restaurant + "-" + user + "-subscription";
    //create topics for both users to push
    [userTopic] = await pubsub.createTopic(userTopicName);
    console.log(`Topic ${userTopic.name} created.`);
    [resTopic] = await pubsub.createTopic(resTopicName);
    console.log(`Topic ${resTopic.name} created.`);
    //create subscription for both users to pull
    [userSubscription] = await
resTopic.createSubscription(userSubscriptionName, {
      retainAckedMessages: false,
      messageRetentionDuration: "10mins",
    });
    console.log(`Subscription ${userSubscriptionName} created.`);
    [resSubscription] = await
userTopic.createSubscription(resSubscriptionName, {
      retainAckedMessages: false,
      messageRetentionDuration: "10mins",
    });
    console.log(`Subscription ${resSubscriptionName} created.`);
    subscriptionHandler();
    console.log("subscriptionHandler Called");
    console.log("Chat Initiated");
    res.send({
      status: true,
      message: "Chat initiated"
    });

  } catch (error) {
    res.send({
      status: false,
      message: error
    });
  }
});
```

Subscription Handler method to fetch the message from the event and add it to the respective message array and acknowledge the message to prevent the duplication:

```
async function subscriptionHandler(){
  try {
    userSubscription.on('message',message=>{
      resMessages.push({
        Id:message.id,
        Time:message.publishTime,
        Message:message.data.toString()
      });
      message.ack();
    });

    userSubscription.on('error', error => {
      console.error('Received error:', error);
    });

    resSubscription.on('message',message=>{
      userMessages.push({
        Id:message.id,
        Time:message.publishTime,
        Message:message.data.toString()
      });
      message.ack();
    });

    resSubscription.on('error', error => {
      console.error('Received error:', error);
    });

  } catch (error) {
    console.log(error);
  }
}
```

GET Api to called by Restaurant team periodically to check if the chat is initiated by the customer:

```
app.get('/checkChatInitiation/:restaurant', async (req, res) => {
  try {
    let restaurant = req.params.restaurant;
    let [allTopics] = await pubsub.getTopics();
    let regExp = new RegExp("projects/" + projectId + "/topics/" + restaurant +
"-[a-zA-Z0-9]*-publisher");
    console.log(regExp);
    let status = false;
    let user = "";
    allTopics.every(topic => {
      if(topic.name.toString().match(regExp))
      {
        console.log(topic.name.toString());
        status = true;
        user = topic.name.toString().split("/").pop().split("-")[1];
        console.log(user);
        return false;
      }
      else
      {
        console.log("Not Matched");
        status = false;
        user = "";
        return true;
      }
    });
    res.send({
      status: status,
      message: (status ? "Topic Found" : "Topic Not Found"),
      user: user
    });
  } catch (error) {
    console.log(error);
    res.send({
      status: false,
      message: error
    });
  }
});
```

POST API to publish message to the respective Cloud PubSub topic by the user (customer/ restaurant):

```
app.post('/pushMessage', async(req, res)=>{
  let role = req.body.role;
  let message = req.body.message;
  try {
    if(role === "user")
    {
      userTopic.publish(Buffer.from(message));
    }
    else
    {
      resTopic.publish(Buffer.from(message));
    }
    console.log("Message Published");
    res.send("Message Sent");
  } catch (error) {
    console.log(error);
    res.send(error);
  }
});
```

GET API to fetch message from the array where the message was added by the backend app when the subscription gets the event on message publish:

```
app.get('/getMessage', async(req, res)=>{
  let role = req.query.role;
  try {
    if(role==="user")
    {
      res.send(resMessages);
    }
    else
    {
      res.send(userMessages);
    }
  } catch (error) {
    res.send(error);
  }
  finally{
    if(role === "user")
    {
      resMessages = [];
    }
    else
    {

```

```

        userMessages = [];
    }
}
});

```

Delete API to delete the create cloud topics and subscriptions created and stop the chat functionality:

```

app.delete('/disconnectChat', async(req, res)=>{
    try {
        //Detaching the subscriptions
        await pubsub.detachSubscription(userSubscriptionName);
        console.log(`Subscription ${userSubscriptionName} detach request was sent.`);
        await pubsub.detachSubscription(resSubscriptionName);
        console.log(`Subscription ${resSubscriptionName} detach request was sent.`);

        //Delete the Topics
        await pubsub.topic(userTopicName).delete();
        console.log(`Topic ${userTopicName} deleted.`);
        await pubsub.topic(resTopicName).delete();
        console.log(`Topic ${resTopicName} deleted.`);

        //Delete the subscriptions
        await pubsub.subscription(userSubscriptionName).delete();
        console.log(`Subscription ${userSubscriptionName} deleted.`);
        await pubsub.subscription(resSubscriptionName).delete();
        console.log(`Subscription ${resSubscriptionName} deleted.`);

        userTopicName="";
        resTopicName="";
        userSubscriptionName="";
        resSubscriptionName="";
        userMessages = [];
        resMessages = [];

        console.log("Cloud assets deleted.");
        // res.send("Chat Disconnected");
        res.send({
            status:true,
            message:"Chat Disconnected"
        });

    } catch (error) {
        console.log(error);
        // res.send(error);
        res.send({
            status:false,

```



```
        message:error
    });
}
});
```

UI function to call pullMessage API of the backend periodically at every 4 seconds if the chat is initiated:

```
function ToggleAvailability(){
    // let timerId;
    console.log("Inside ToggleAvailability",isOnline);
    if(isOnline)
    {
        setTimeout(setInterval(() => {
            pullMessage();
        }, 4000));
    }
    else
    {
        clearInterval(timerId);
        if(role === "user")
        {
            history.push("/home");
        }
        else
        {
            history.push("/restaurantHome");
        }
    }
}
```

Front-end Function to publish the message by calling the backend pushMessage API and adding message to the UI:

```
function addToConversation(fromOtherUser)
{
    let prefix = fromOtherUser ? "[" + (role === USER_PROFILE ? restaurantName : user)
+ "]" : "[You]:";
    let prevMessages = conversation;
    console.log("Conversation",conversation);
    if(!fromOtherUser)
    {
        pushMessage();
        prevMessages += prefix + userMessage + "\n";
        setConversation(prevMessages);
        setUserMessage('');
    }
    else
    {
        if(pulledMessage.length > 0)
        {
            prevMessages += prefix + pulledMessage + "\n";
            console.log("prevMessage before clearing",prevMessages);
            setConversation(prevMessages);
            setPulledMessage('');
        }
    }
    console.log("PreviousMessage",prevMessages);
}
```

Data-Processing Module (WordCloud)

Lambda handler that fetches the customer feedbacks and processes the data:

```
import pymysql.cursors
import boto3
import json
import os

RDS_HOST = os.environ['RDS_HOST']
RDS_DB = os.environ['RDS_DB']
RDS_USER = os.environ['RDS_USER']
RDS_PASSWORD = os.environ['RDS_PASSWORD']
port=3306
Region = os.environ['Region']

try:
    conn = pymysql.connect(host=RDS_HOST, user=RDS_USER,password=RDS_PASSWORD, db=RDS_DB)
    comprehend = boto3.client(service_name='comprehend',region_name=Region)
except Exception as e:
    print(e)

def lambda_handler(event,context):
```

```

statusCode=0
data={}
message=""
itemId = event.get('itemId', 0)
feedbacks = []
entities = {}
try:
    if(itemId != 0):
        sql = "SELECT * FROM fooditemFeedbacks WHERE menuItemId= {}".format(itemId);
        print(sql)
        with conn.cursor() as cur:
            cur.execute(sql)
            body = cur.fetchall()
            for row in body:
                # print(row)
                feedbacks.append(row[2])
            # print(feedbacks)
        for feedback in feedbacks:
            # print(feedback)
            result =
json.dumps(comprehend.detect_entities(Text=feedback,LanguageCode='en')['Entities'])
            # print(result)
            if(len(result) > 0):
                for entity in result:
                    text = entity['Text']
                    if(text in entities.keys()):
                        entities[text] += 1
                    else:
                        entities[text] = 1
                # print(entities)
                statusCode=200
                menuItemId="Entities extracted"
            else:
                statusCode=400
                message="Bad Request"
        except Exception as e:
            print(e)
            statusCode=500
            message="Internal Server Error"

    return {
        'statusCode': statusCode,
        'message':message,
        'data':json.dumps(entities)
    }

```

Front-end code that calls the Lambda handler via API Gateway:

```

async function getFeedbacks(menuItemId)
{
    await axios({
        method:"get",
        url:"https://fgy5izgkjh.execute-api.us-east-1.amazonaws.com/PROD",
        params:{'menuItemId':menuItemId}
    })
}

```

```

    }).then((response)=>{
      console.log(response.data.data);
      setWords(response.data.data);
      setIsLoading(false);
    }).catch((error)=>{
      alert(error);
      // history.push("/home");
      setIsLoading(false);
    });
  }
}

```

Menu Configuration

Lambda function to add a food-item to menu:

```

const mysql = require('mysql2');
const con = mysql.createConnection({
  host      : process.env.RDS_HOSTNAME,
  user      : process.env.RDS_USERNAME,
  password  : process.env.RDS_PASSWORD,
  port      : process.env.RDS_PORT,
  database  : process.env.RDS_DATABASE
});
con.connect();

const editData = (menuItemId) => {
  const sql = "DELETE FROM menuItems where Id = " + menuItemId;

  return new Promise((resolve, reject) => {
    con.query(sql, (err, res, fields) => {
      if (err) {
        reject(err);
      }
      else {
        {
          resolve(res);
        }
      }
    });
  });
}

exports.handler = async (event, context, callback) => {
  // allows for using callbacks as finish/error-handlers
  // context.callbackWaitsForEmptyEventLoop = false;
  let message = "";
  let menuItemId = 0;
  let responseCode = 500;
  let sql = "";
  let response = {};

```

```

console.log(event);
if (event.Id) {
    menuItemId = event.Id;
    responseCode=200;
    await editData(menuItemId).then((res)=>{
        response = {
            statusCode:responseCode,
            message:"Item Updated successfully",
        }
    })
    .catch((e)=>{
        response={
            statusCode:500,
            message : "Internal Server Error Occurred",
        }
    })
}
else
{
    response={
        statusCode:400,
        message : "Bad request",
    }
}
return response;
};

```

Lambda function to edit a food-item to menu:

```

const editData = (menuItemId,name,price,ingredients) => {
    const sql = "UPDATE menuItems SET name = '" + name + "', price=" + price +
    ",ingredients='" + ingredients + "' where Id = " + menuItemId;

    return new Promise((resolve,reject)=>{
        con.query(sql,(err,res,fields) =>{
            if(err){
                reject(err);
            }
            else
            {
                resolve(res);
            }
        });
    });
}

exports.handler = async (event,context,callback) => {
    let message="";
    let menuItemId=0;
    let price=0;
    let ingredients='';
    let name='';
    let responseCode = 500;

```

```

let sql = "";
let response = {};
console.log(event);
if (event.Id && event.name && event.price && event.ingredients) {
    menuItemId = event.Id;
    name=event.name;
    price=event.price;
    ingredients=event.ingredients;
    responseCode=200;
    await editData(menuItemId,name,price,ingredients).then((res)=>{
        response = {
            statusCode:responseCode,
            message:"Item Updated successfully",
        }).catch((e)=>{
            response={
                statusCode:500,
                message : "Internal Server Error Occurred",
            });
        });
    }
    else{
        response={
            statusCode:400,
            message : "Bad request",
        }
    }
    return response;
};

```

Lambda function to delete a food-item to menu:

```

const deleteData = (menuItemId) => {
    const sql = "DELETE FROM menuItems where Id = " + menuItemId;
    return new Promise((resolve,reject)=>{
        con.query(sql,(err,res,fields) =>{
            if(err){
                reject(err);
            }
            else{
                resolve(res);
            }
        });
    });
}

exports.handler = async (event,context,callback) => {
    let menuItemId=0;
    let responseCode = 500;
    let response = {};
    console.log(event);
    if (event.Id) {
        menuItemId = event.Id;
        responseCode=200;
        await deleteData(menuItemId).then((res)=>{
            response = {

```

```

        statusCode:responseCode,
        message:"Item Updated successfully",
    }
    }).catch((e)=>{
        response={
            statusCode:500,
            message : "Internal Server Error Occurred",
        }
    });
}
else{
    response={
        statusCode:400,
        message : "Bad request",
    }
}
return response;
};

```

Chatbot for application navigation

JavaScript Function to authorize the use of AWS Lex via Cognito:

```

componentDidMount() {
    document.getElementById("inputField").focus();
    AWS.config.region = 'us-east-1';
    AWS.config.credentials = new AWS.CognitoIdentityCredentials({
        IdentityPoolId: 'us-east-1:bb0d107f-cacc-450c-8531-13697ad44c2d',
    });
    var lexruntime = new AWS.LexRuntime();
    this.lexruntime = lexruntime;
}

```

Pushing messages from frontend to AWS Lex:

```

pushChat(event) {
    event.preventDefault();

    var inputFieldText = document.getElementById('inputField');

    if (inputFieldText && inputFieldText.value && inputFieldText.value.trim().length > 0) {

        // disable input to show we're sending it
    }
}

```

```

var inputField = inputFieldText.value.trim();
inputFieldText.value = '...';
inputFieldText.locked = true;

// send it to the Lex runtime
var params = {
  botAlias: '$LATEST',
  botName: 'Hfoodie',
  inputText: inputField,
  userId: this.state.lexUserId,
  sessionAttributes: this.state.sessionAttributes
};
this.showRequest(inputField);
var a = function(err, data) {
  if (err) {
    console.log(err, err.stack);
    this.showError('Error: ' + err.message + ' (see console for details)')
  }
  if (data) {
    // capture the sessionAttributes for the next cycle
    this.setState({sessionAttributes: data.sessionAttributes})
    //sessionAttributes = data.sessionAttributes;
    // show response and/or error/dialog status
    this.showResponse(data);
  }
  // re-enable input
  inputFieldText.value = '';
  inputFieldText.locked = false;
};

this.lexruntime.postText(params, a.bind(this));
}
// we always cancel form submission
return false;
}

```

Showing the response messages from Lex:


```

showResponse(lexResponse) {

    var conversationDiv = document.getElementById('conversation');
    var responsePara = document.createElement("P");
    responsePara.className = 'lexResponse';
    if (lexResponse.message) {
        responsePara.appendChild(document.createTextNode(lexResponse.message));
        responsePara.appendChild(document.createElement('br'));
    }
    if (lexResponse.dialogState === 'ReadyForFulfillment') {
        responsePara.appendChild(document.createTextNode(
            'Ready for fulfillment'));
        // TODO: show slot values
    } else {
        responsePara.appendChild(document.createTextNode(
            ""));
    }
    conversationDiv.appendChild(responsePara);
    conversationDiv.scrollTop = conversationDiv.scrollHeight;
}

```

Lambda Function to fulfill the Lex request:

```

app.get("/orderHistory", (req, res) => {

    con.query(
        `SELECT * FROM orderHistory WHERE userID = 'fenilshah14@yahoo.com'`,
        function (err, result, fields) {
            if (err) {
                console.log(err);
                return res.status(500).json({
                    success: false,
                    message: "server error",
                });
            }

            if (result.length > 0) {

```

```

    return res.status(200).json({
      success: true,
      message: "orders retrieved",
      data: result
    });
  } else {
    return res.status(204).send({
      success: false,
      message: "No previous order found",
    });
  }
}
);
});

```

Recipe Similarity score

Javascript Function to make API call to the Google Cloud Function:

```

const submitHandler = (event) => {
  event.preventDefault();
  console.log("form submitted");
  const url =
    "https://us-central1-utility-league-321023.cloudfunctions.net/function-1";

  axios
    .post(url, data)
    .then(function (response) {
      console.log("the response is: " + response);
      if (response.status === 200) {
        let value = response.data.value / 1000;
        var predictionScore = value + 50;
        setScore(predictionScore);

        console.log(response.data.value);
        console.log(score);
      }
    })
}

```

```

        .catch(function (error) {
            console.log(error);
        });
    };

```

Fetching data from frontend in Google Cloud Functions and sending back the prediction result:

```

def hello_world(request):
    """Responds to any HTTP request.
    Args:
        request (flask.Request): HTTP request object.
    Returns:
        The response text or any set of values that can be turned into a
        Response object using
        `make_response <http://flask.pocoo.org/docs/1.0/api/#flask.Flask.make_response>`.
    """

    request_json = request.get_json()

    list_data = []
    list_data.append(request_json)
    instances = list_data
    x = predict_tabular_regression_sample(instances)
    if request.args and 'message' in request.args:
        return request.args.get('message')
    elif request_json:
        return x
    else:
        return f'something gone wrong'

```

Making Prediction call to Google AutoML from Cloud Functions:

```

def predict_tabular_regression_sample(
    instances
):

    project = "412171526637"
    location = "us-central1"
    endpoint = "8189145011489603584"

```

```

aiplatform.init(project=project, location=location)

endpoint = aiplatform.Endpoint(endpoint)

response = endpoint.predict(instances=instances)

return response.predictions[0]

```

Provide Feedback & Ratings

Javascript code for sending user feedback to API gateway:

```

const feedbackSubmitHandler = (event) => {
  event.preventDefault();
  console.log("ff");

  const url = "https://jg3re3ezm8.execute-api.us-east-1.amazonaws.com/api/feedback";

  axios
    .post(url, feedbackData)
    .then(function (response) {
      console.log("the response is: " + response);
      if (response.status === 200) {
        if (response.data.success) {
          props.history.push({
            pathname: "/restaurant/1",
          });
        } else {
          console.log("");
        }
      }
    })
    .catch(function (error) {
      console.log(error);
    });
};

```

Lambda code to store feedback in RDS database:

```

app.post('/feedback', (req, res) => {

```

```

var records = [[req.body.restaurantId, req.body.menuItemId, req.body.feedback, req.body.rating]];
if(records[0][0]!==null)
{
    con.query("INSERT into feedback (restaurantId,menuItemId,comments,ratings) VALUES
?",[records],function(err,result,fields){
        if(err)
        {
            console.log(err)
            return res.status(500).json({
                success: false,
                message: "server error"
            })
        }else
        {
            return res.status(200).json({
                success: true,
                message: "Feedback submitted",
            })
        }
    });
}

```

Restaurant Menu

Lambda function to fetch menu:

```

var SQL = require("mysql2");
const sqlCon = SQL.createConnection({
    host: "fooddelivery5410.cy8c8vgerfgo.us-east-1.rds.amazonaws.com",
    user: "admin",
    password: "password1234",
    database: "mfa",
});
sqlCon.connect();
const fetchAllMenu = () => {
    const FETCH_QUERY = "SELECT * FROM hfoodie.menuItems";
    return new Promise((resolve, reject) => {
        sqlCon.query(FETCH_QUERY, (err, res, flds) => {
            if (err) {
                reject(err);
            } else {
                resolve(res);
            }
        })
    })
}

```

```

    });
  });
};
exports.handler = async (event) => {
  let response = { success: false };
  await fetchAllMenu()
    .then((d) => {
      response = { success: true, data: d };
    })
    .catch((e) => {
      response = { success: false, message: e.message };
    });
  return response;
};

```

Order Food

Lambda function to order the food:

```

var SQL = require("mysql2");

const sqlCon = SQL.createConnection({
  host: "fooddelivery5410.cy8c8vgerfgo.us-east-1.rds.amazonaws.com",
  user: "admin",
  password: "password1234",
  database: "mfa",
});

sqlCon.connect();
const getRestaurantName = (resid) => {
  const SELECT_QUERY = "SELECT resname FROM hfoodie.restaurants WHERE resid=?";

  return new Promise((resolve, reject) => {
    sqlCon.query(SELECT_QUERY, [resid], (err, res, flds) => {
      if (err) {
        reject(err);
      } else {
        if (res.length > 0) {
          console.log(res)
          resolve(res[0]["resname"]);
        } else {
          reject();
        }
      }
    });
  });
};

const getMenuName = (menuItemID) => {
  const SELECT_QUERY = "SELECT * FROM hfoodie.menuItems WHERE id=?";

```

```

    return new Promise((resolve, reject) => {
      sqlCon.query(SELECT_QUERY, [menuItemID], (err, res, flds) => {
        if (err) {
          reject(err);
        } else {
          if (res.length > 0) {
            console.log(res)
            resolve(res[0]["name"]);
          }
        }
      })

      const SELECT_REST_QUERY =
        "SELECT resname FROM restaurants WHERE resid =
        "+res[0]["restaurantId"];

      console.log(SELECT_REST_QUERY);
      sqlCon.query(SELECT_REST_QUERY, (err, res1, flds) => {

        const INSERT_QUERY =
          "INSERT INTO hfoodie.orderHistory (restId, menuId,
          restaurantName, food, total_price) VALUES (?, ?, ?, ?, ?)";

        console.log(res1);
        sqlCon.query(INSERT_QUERY, [res[0]["restaurantId"],
        res[0]["Id"], res1, res[0]["name"],
        res[0]["price"]], (err, res, flds) => {});

      });
    });
  });
});

const putOrderFood = (resid, menuItemID, resname, name, price) => {
  const INSERT_QUERY =
    "INSERT INTO hfoodie.orderHistory (restId, menuId, restaurantName, food,
    total_price) VALUES (?, ?, ?, ?, ?)";

  return new Promise((resolve, reject) => {
    sqlCon.query(INSERT_QUERY, [resid, menuItemID, resname, name, price],
    (err, res, flds) => {
      if (err) {
        reject(err);
      } else {
        resolve();
      }
    });
  });
};

const handler = async (event) => {
  let response = { success: false };
  let qString;

```

```

console.log("event", event)

await getRestaurantName(1)
  .then((d) => {
    qString = d;
    console.log("Rest name:", qString);
    response = { success: true, data: d };
  })
  .catch((e) => {
    console.error(e);
    response = { success: false, message: e.message };
  });

await getMenuName(167)
  .then((d) => {
    qString = d;
    console.log("Menu name:", qString);
    response = { success: true, data: d };
  })
  .catch((e) => {
    console.error(e);
    //response = { success: false, message: e.message };
  });
return response;
};
exports.handler = handler;

```

Data Visualization

```

import React, { useState, useEffect } from "react";
import { Redirect, useParams } from "react-router-dom";
import axios from "axios";
import { EMAIL_KEY } from "../Utils/AccountUtils";

const Visualization = () => {
  const { id } = useParams();
  useEffect(() => {
    //fetchVisualization();
  }, []);

  async function fetchVisualization() {

    window.location.href =
    "https://datastudio.google.com/reporting/6e280305-62e7-4fd3-81da-0d9470e882aa";
  };

  return (
    <div>
      <h1> Visualization: </h1>
      <div className="form-group">

```



```
        <iframe width="1000" height="1000"
src="https://datastudio.google.com/embed/reporting/f5508dab-8392-4c06-9ce6-
b7d3745932c9/page/uNQWC" frameborder="0" allowfullscreen></iframe>
      </div>
    </div>
  );
};
export default Visualization;
```

Integration Details

We have used GitHub for version control and code integration. We started working on the same project template and developed our individual features on different branch. After completion of each feature, we merged our code and tested it. At last, all the features were merged into main branch.

For the AWS part, we created AWS services for our individual feature in our individual account. After the completion of the feature, we moved our services to a single account which was hosting Cognito. Thus, all of the services were provided from a single account with Cognito authorization.

Deployment Details

For the frontend part of our application. Initially we decided to deploy our frontend on AWS EC2 but after the design discussion with the professor we came to know that it is not fully serverless. So, we decided to use AWS Elastic Beanstalk (EBS) which is a service to deploy and scale web application and services. EBS automatically deploys and takes care of capacity provisioning, auto scaling, load balancing and other things. We hosted our frontend developed in node.js on EBS.

For our backend code, we have deployed our backend code on Lambda Function for each individual function of the backend like feedback, word cloud, order etc.

Testing Details

Sign-in, Sign-up and Multi-Factor Authentication (MFA)

The error handling or the testing details are shown in the images below,

Sign-up Testing

Halifax Foodie

SIGN UP

Email

Password

Re-enter Password

Account Type

User

Password didn't match

SIGN UP

Halifax Foodie

SIGN UP

Email

Password

Re-enter Password

Account Type

User

An account with the given email already exists.

SIGN UP

Sign-in Testing

Halifax Foodie

LOGIN

Email

Password

Incorrect username or password.

Note: Confirm/Verify the user with given E-mail before login.

LOGIN

SIGN UP

MFA Testing

Halifax Foodie

MFA

Am I an idiot?

Yes

Am I an idiot?

Yes

Am I an idiot?

No

MFA, Failed! Try again.

SUBMIT

LOG OUT

Chatbot for application navigation

Greeting intent on starting the chatbot.

Chat with our awesome bot^

hi

Hello, What is your name ?

pathik

Hello pathik, How can i help you ?

Chat with us

Moving to different intent based on user input.

Chat with our awesome bot^	Chat with our awesome bot^
<p>hi</p> <p>Hello, What is your name ?</p>	<p>hi</p> <p>Hello, What is your name ?</p>
<p>pathik</p>	<p>pathik</p>
<p>Hello pathik, How can i help you ?</p>	<p>Hello pathik, How can i help you ?</p>
<p>i want to order</p>	<p>i want help with my order</p>
<p>please go to orders and select the restaurants of your choice. Then, order your favourite items.</p>	<p>At the bottom of the page you can find our contact info. You can also try our chat functionality to talk with restaurant.</p>
<input type="text" value="Chat with us"/>	<input type="text" value="Chat with us"/>

Calling Lambda function in backend to get the order status

Chat with our awesome bot^	Chat with our awesome bot^
<p>hi</p> <p>Hello, What is your name ?</p>	<p>hi</p> <p>Hello, What is your name ?</p>
<p>pathik</p>	<p>pathik</p>
<p>Hello pathik, How can i help you ?</p>	<p>Hello pathik, How can i help you ?</p>
<p>what is my order status</p>	<p>what is my order status</p>
<p>1</p> <p>what is your orderId ?</p> <p>your order is: delivered</p>	<p>2</p> <p>what is your orderId ?</p> <p>your order is: cancelled</p>
<input type="text" value="Chat with us"/>	<input type="text" value="Chat with us"/>

Recipe Similarity score

Input the ingredients information in the form.

Enter your recipe

your recipe similarity score is :

Cuisine

italian

Ingredient 1

egg

Ingredient 2

chilli

Ingredients 3

beef

Ingredients 4

salt

Ingredients 5

pepper

Ingredients 6

turmeric

Ingredients 7

pork

Ingredients 8

wheat

Ingredients 9

rice

Submit

Getting the prediction Response based on the input.

Enter your recipe

your recipe similarity score is
:74.766669921875

Cuisine

italian

Ingredient 1

egg

Ingredient 2

chilli

Ingredients 3

beef

Ingredients 4

salt

Ingredients 5

Feedback and Ratings

Feedback Form: After successful submission the page will be redirected back to the Feedback page where all of the items are there.

Feedback

food Item: Beef Dumpling Stew

Ratings:

☐ 1 ☐ 2 ☐ 3 ☒ 4 ☐ 5

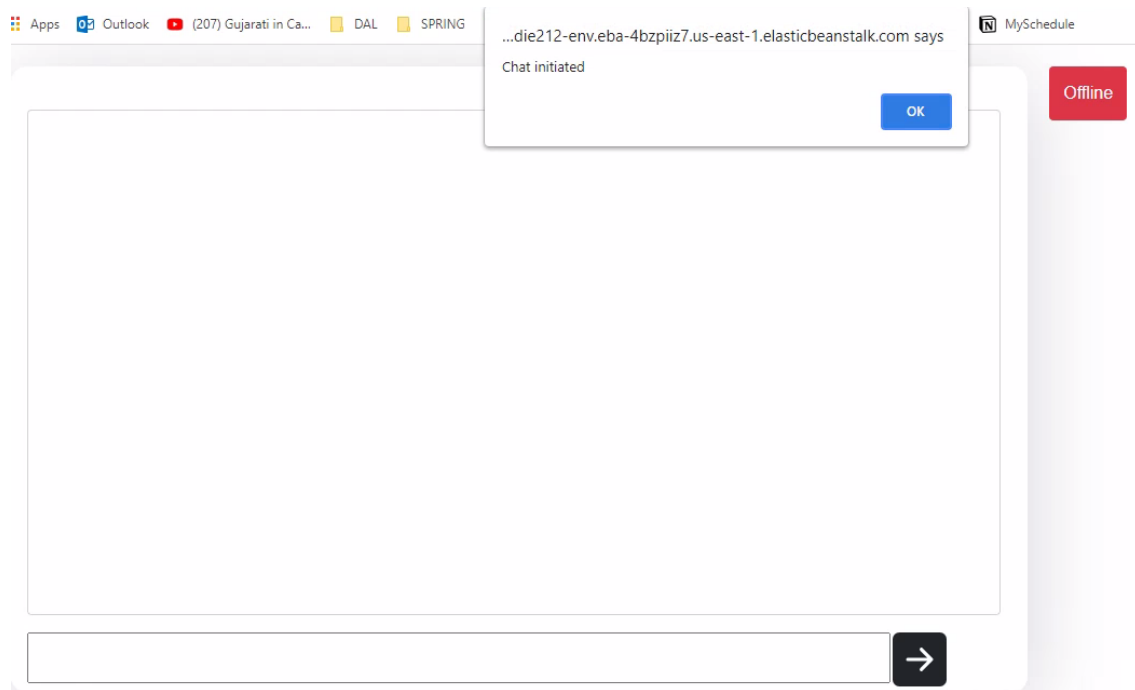
Comments

this food is very delicious

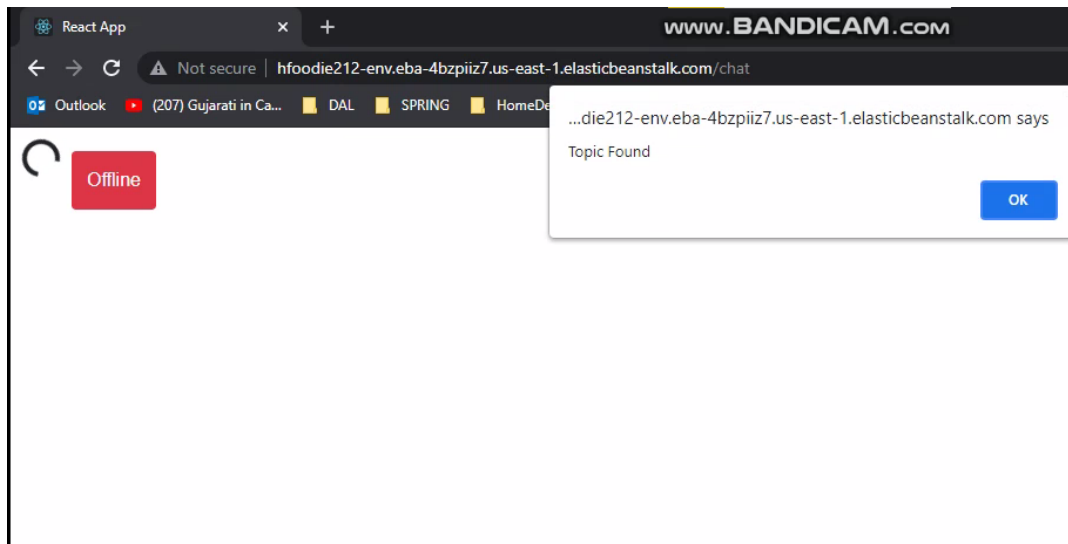
Submit

Customer-Restaurant Instant Messaging

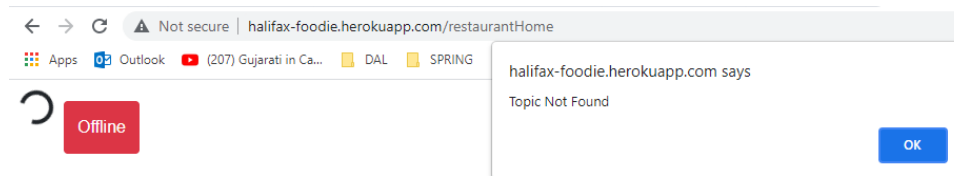
Success Alert message on Chat initiation on Customer-end



"Topic-Found" alert message on Restaurant-end if any customer has initiated the chat

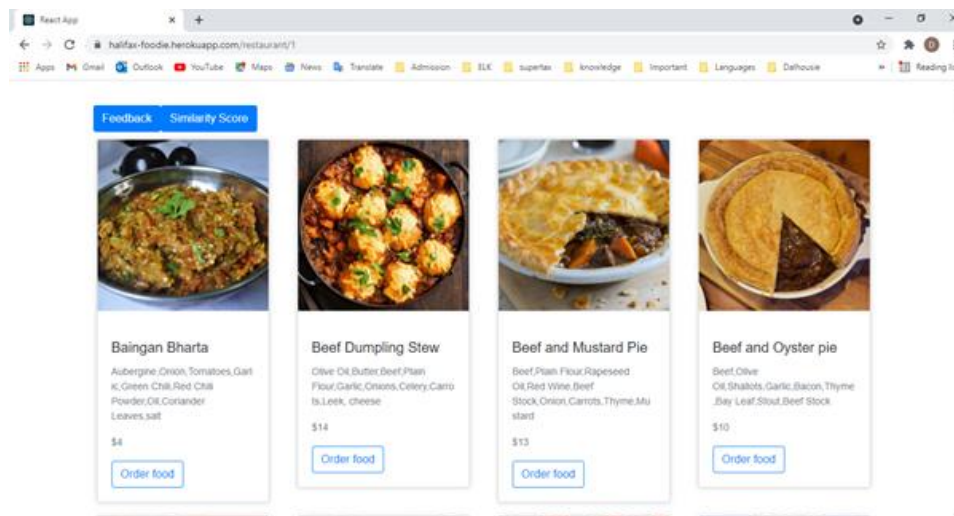


"Topic Not Found" alert message on Restaurant-end if any customer has not initiated the chat



Restaurant Menu

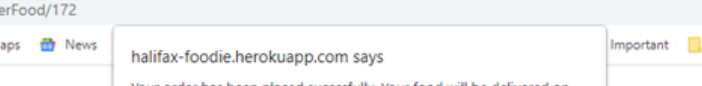
When The user select any particular restaurant, the following page will open.



When the user clicks on the 'Order Food' button, the user information page will open which will take the information of delivery.

After that, user enters the details of delivery like below picture.

After submitting the details, the user gets the confirmation of the order.



app.com/orderFood/172

YouTube Maps News Important Languages

Details:

halifax-foodie.herokuapp.com says

Your order has been placed successfully. Your food will be delivered on your address soon.

OK

Visualization

For the visualization page, Click the button 'Visualization'.

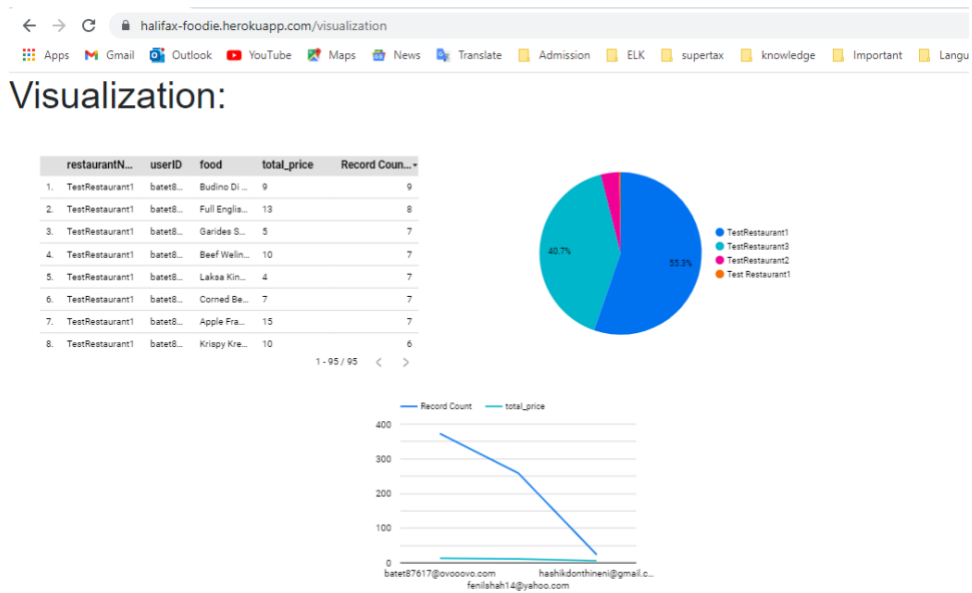
Home

Log Out

Visualization

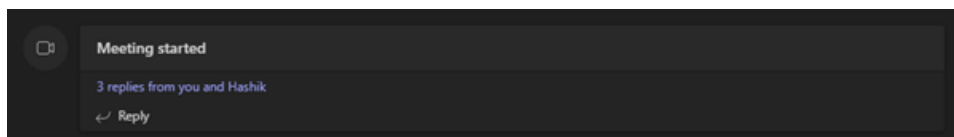


After clicking 'Visualization' button, the following page will open.

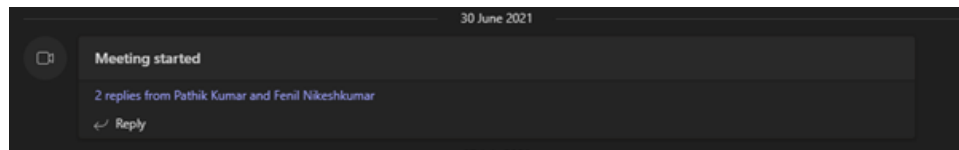


Meeting Details

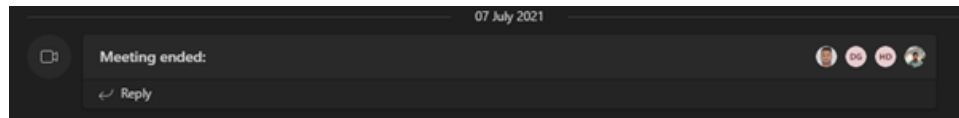
25th June:



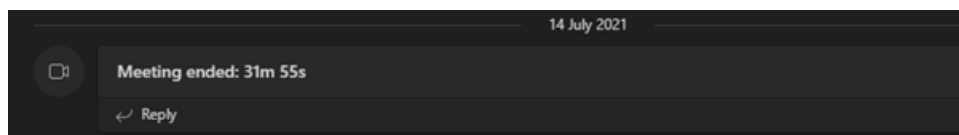
30th June:



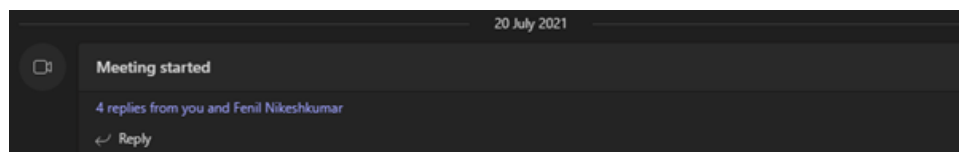
7th July:



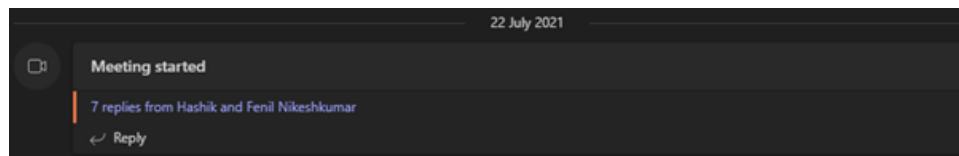
14th July:



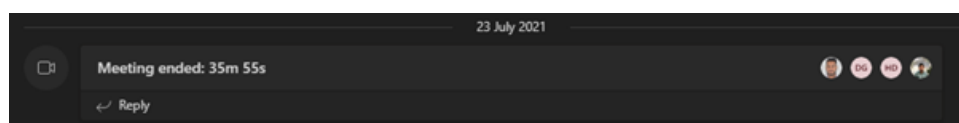
20th July:



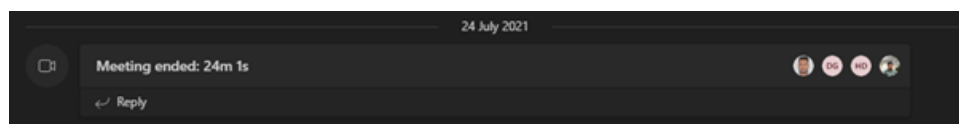
22nd July:



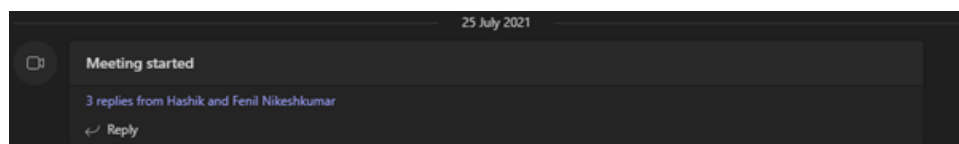
23rd July:



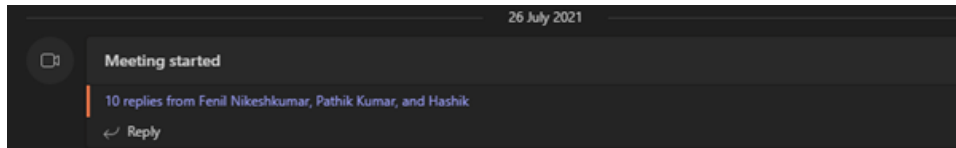
24th July:



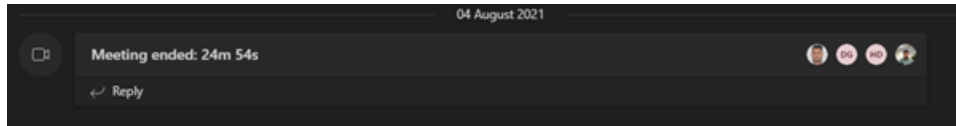
25th July:



26th July:



4th August:



Team Information

Team Strengths:

- The communication was fair and open among the team which helped in conveying ideas more clearly and with openness to accept best ideas.
- Everyone contributed their fair share in the project and made sure other team members are not overloaded with tasks.
- During meetings, every member shared their progress and blockers and helped each other to achieve the goal.
- Most importantly, every member was eager to learn and highlight their efforts and talents through their work, which helped us as a team to present a good project work.

Team Weakness:

- Every team member had the same level of experience with the serverless technologies. So, designing the most optimum solution was hard to achieve.
- Two of our group members were in India. So, there were some time constraints for meeting and collaboration.

Challenges faced

1. Implementation of Cognito, API Gateway
 - a. Cognito is not well documented, and few know companies use it.
 - b. Editing the hosted Cognito UI requires learning and getting used to. Better to write our own UI with SDK.
 - c. SDK for Cognito as well is not documented well.
 - d. API Gateway takes time (delay) for deployment of Options method which is frustrating and sometime misleading us to believe the service is not working and re-deploying multiple times.
2. Implementation of Instant Messaging module
 - a. Since, Cloud PubSub is not ideal to be used as a chatting service, implementing the instant messaging module using Cloud PubSub was a bit difficult.
 - b. Also, it seemed impossible to implement the feature in a fully serverless way as Google does not provide any API using which the service APIs can be accessed directly from the frontend. Google provides PubSub API which requires it to be running in the backend in

order to listen to the subscription events whenever a message is published to a PubSub topic.

- c. Hence, we figured out to create an event-driven bridge between the frontend application and the PubSub service through a backend application running on Cloud Run.
3. Data Storage and Retrieval using AWS RDS, API Gateway, and Lambda functions
 - a. First, we had to know how lambda works, lambda is for single function. so, its code structure is somewhat different from developing the whole backend.
 - b. After that we need had to find how to connect RDS instance with Lambda.
 - c. Also, we had to find how to trigger lambda function using API gateway and creating the endpoint for our API.
4. Document similarity using Machine Learning.
 - a. There are many models available for Machine Learning in Google like classification, regression etc. and using that model on different database. We had to find out which model is most suitable for our problem.
 - b. After finding the suitable model the next step was to learn how to train that model and what type of features to consider for prediction.
 - c. After training, the main difficulty was to feed data to the model from our frontend and getting the prediction result. We tried a lot of stuff and find out that authorization is needed to access model. So we decided to use Google cloud function.
 - d. At last we had to figure out how cloud functions works and how the data can be received from the frontend and making call to AutoML model.
5. Chatbot Using AWS Lex.
 - a. The bot creation is straightforward in AWS but the main difficulty was to find how we can move from one intent to another based on the user input i.e. after greeting intent how to move to particular query intent based on user input.
 - b. The next step was to incorporating the lambda function with Lex. We had to find out how to pass the data Lex to lambda and also how to fulfill the request and give back proper response to the Lex bot.
 - c. After completing the Lex bot in AWS. The main difficulty was to find how to access lex bot from frontend. We had to search a lot of AWS docs and also went to many tutorials to find out that. At last, we find solution of how to incorporate Lex in frontend using the Cognito authorization.
6. Visualization:
 - a. The implementation of visualization in Data Studio was easy but the difficulty we faced is the connection between the visualization and AWS lambda function. We had to find a way to represent the graphs, which were made in Data Studio, to the customers in a dynamic way.

References

- [1] "Flowchart Maker & Online Diagram Software", *App.diagrams.net*, 2021. [Online]. Available: <https://app.diagrams.net/>. [Accessed: 20- Jul- 2021]
- [2] "Random Food Adjective Generator", *Perchance.org*, 2021. [Online]. Available: <https://perchance.org/food-adjective>. [Accessed: 15- Jul- 2021].
- [3] "TheMealDB.com", *Themealdb.com*, 2021. [Online]. Available: <https://www.themealdb.com/>. [Accessed: 14- Jul- 2021].
- [4] "@google-cloud/pubsub", *npm*, 2021. [Online]. Available: <https://www.npmjs.com/package/@google-cloud/pubsub>. [Accessed: 16- Jul- 2021].
- [5] "react-wordcloud", *npm*, 2021. [Online]. Available: <https://www.npmjs.com/package/react-wordcloud>. [Accessed: 07- Jul- 2021].
- [6] "npm: react-lex-plus," Npmjs.com. [Online]. Available: <https://www.npmjs.com/package/react-lex-plus>. [Accessed: 07-Aug-2021].
- [7] Amazon.com. [Online]. Available: <https://docs.aws.amazon.com/>. [Accessed: 07-Aug-2021].
- [8] "Vertex AI documentation," Google.com. [Online]. Available: <https://cloud.google.com/vertex-ai/docs>. [Accessed: 07-Aug-2021].
- [9] Dashboarding & data visualization tools - Google Data Studio. (n.d.). Retrieved August 7, 2021, from Google.com