Project Specification – Version 2.0.1

Release Date: Sep 28, 2022 [Version 1.0 released on Sep 22, 2022]
Project Lead: Bharat Shankaranarayanan
Client: Dr. Saurabh Dey

Title:

HalifaxFoodie *- A Multi-Cloud based Serverless Food Delivery System*

Objective:

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. You will use different backend services, and simple front-end application to build the system.

Explanation:

This project is introduced in the Serverless Data Processing Course (CSCI 5410) to fulfill the course requirement. This is a group project (weightage 40%), and each group is required to perform specific tasks within a given time frame. There are project constraints, and scope, which must be followed by each team. The project will follow an agile model, where each team should welcome changes in the requirements. However, considering the time and resource restriction, requirement changes will be limited.

Hypothetical Scenario:

DALSoft5410 is building a serverless Food Delivery system using multi-cloud deployment model, and backend-as-a-service (BaaS). The Food Delivery System - "*HalifaxFoodie*", should provide customization feature, and additional services for restaurant owners, and limited services to customers. The application should provide an online virtual assistance, which can quickly answer the queries of registered restaurant owners, and customers or guests.

The virtual assistant functionality can be extended to support customer escalation. The application should initiate a chat functionality between a registered customer, and customer service representative if there are any delivery or service quality issues.

The application will provide data security, user {restaurant, and customers} management, customer feedback polarity analysis, food item delivery tracker, restaurant ratings, customer recommendation, and discount coupon etc. As an additional feature, each restaurant can upload their top recipe on the system for checking similarity score.

DALSoft5410 has selected serverless application to minimize the development and project running cost. The company has identified two cloud platforms - AWS, and GCP to build, test, and deploy their application. They have decided to follow the official documentations of AWS and GCP to build the different pieces.

If they select server-oriented architecture, then they need to manage and configure the backend service, which they cannot do due to their resource limitations. Therefore, serverless is the only solution they found at this point. They have obtained two types of accounts from AWS, and one account from GCP, which they can use for building, testing, and deploying their application.

Since they are going to follow agile method, they can build, test, and change each component of the project whenever there is a change in the requirements.

Specification Details:

In this application, you are required to build a simple front-end, which will interact with Backend-as-a-Service (BaaS). Some specific requirements are identified, which should be considered for design and implementation. **You must use Severless architecture (there could be some exceptions) **

Front-End (Non-essential module considering course objective)

Build a front-end application using suitable framework and use it to call backend services – Use of ***React*** as a front-end can be an option. You can use any other web technologies as well.

Hosting of entire front-end application or user facing interface should be done as a microservice on **GCP CloudRun**

Backend Services (Essential modules)

1. User Management Module:

- **Registration:** The system should accept registration request from multiple people and should validate the registration process.
  - Customer, and Restaurant signup and validation should be done using AWS Cognito. The userID, and password used in registration will be used during authentication as **1st factor** authentication.
  - Customers' and Restaurants' other essential details should be kept in DynamoDb using AWS Lambda
  - Customer and Restaurant Question-Answer for **2nd factor** authentication must be kept in GCP Firestore using a function (CloudFunction). The questions can be fixed, and displayed on the screen, and user can enter answers for the questions. Once it is done, the CloudFunction will store both question and answer in the GCP Firestore.
  - For **3rd Factor** (Columnar Cipher) – accept a key K (a string from user of length 4), and a plain text P_{text}. Then using cloud function or Lambda compute the cipher text C_{text}. Return ciphertext C_{text} to user. Keep the key, and plain text in the DynamoDb or Firestore {$userID, K, P_{text}$}.

- **Authentication:** The system should authenticate the user using three factors, and then allow the user to access the services. Note: 3rd factor will be enabled only after 2nd factor is validated, and 2nd factor should be enabled only after 1st factor is validated.
  - The customer or restaurant enters the userID and password through the front-end, which interacts with the Cognito, and validates the user.
  - If 1st factor is "True", then control moves to GCP CloudFunction, which performs question-answer check.
  - If 2nd factor is "True", then Lambda or CloudFunction performs the Columnar Cipher validation. It will ask user to enter the cipher text C_{text}', which is given to the user during registration. Once it is entered by the user, the Lambda or CloudFunction will scan the database, and retrieve the key, and plain text. Using the registration logic, it will compute cipher text C_{text}'', and match the cipher text entered by the user {if $C_{text}' == C_{text}''$}. If "True", user gains access to the system.

2. Online Support Module: Bots should respond to queries (AWS Lex)

- Online virtual assistance for website navigation (static response), which should also work for guest users
- The AWS Lex chatbot should contain multiple intents to handle each scenario.

- For some intents, AWS Lambda will be required to update DynamoDb or select data from DynamoDb
- For registered users, the bot should 1st validate the userID, and only then provide support for order tracking etc (dynamic response). AWS Lex + AWS Lambda + DynamoDb
- A registered user should be able to rate an order based on the order number. AWS Lex + AWS Lambda + DynamoDb
- A registered restaurant owner should be able to add recipe names, and prices etc. using the chatbot (dynamic response). AWS Lex + AWS Lambda + DynamoDb
- For order related issues, or complaints – the chatbot should end the current virtual support session and provide option for chatting with human, i.e., it should initiate chat module

3. **Chat Module:** Restaurant Customer Service Representative and Customer should be able to communicate (GCP Firebase to create chatroom) – This will be an instant messaging engine
    - Customer and Restaurant Manager should be able to chat in case of issues related to order or service.
    - With each problem, e.g., "late food delivery", a chatroom can be created dynamically, and user can interact with restaurant agent who is responsible for that specific problem
    - This module requires multiple logged in sessions. At least one session to represent a user, and one log-in session for restaurant agent.

4. **Data Processing:** Create a microservice or lambda function to extract named entities (uppercase entities) from food recipes. Use AWS Lambda + Comprehend.
    - This service will be used by restaurant owners to extract key ingredients from a recipe.
    - Restaurant owners should be able to upload recipes in the application, which will be stored as text files in S3 or in Cloud Store.
    - After uploading the recipes, the restaurant owner can click a button to extract title or key ingredients from the recipe (named entities), which will be stored as metadata in DynamoDb or Firestore along with the name of the recipe for easy search. This information could be used by machine learning module.
    - Using Lambda and Comprehend this module can be designed.

5. **Machine Learning:**
    - **Similarity Score:** To identify the similarity of the recipes uploaded use GCP AutoML or similar service. Similarity check of recipe files can be done based on various measures, such as Euclidean distance.
        - If recipe files on Veg Pizza, Veg Curry, Green Salad etc. are uploaded, then it should classify the files based on similarity score.
        - You can select any threshold as similarity measure. E.g., if two recipes have 60% similarity score based on some feature selection, then we can consider those as similar (since score > 50%)

    - **Polarity:** The polarity of the feedback that the users have entered should be measured using AWS Comprehend and visualized using AWS QuickSight. The restaurant owner should be able to click on option to view visualize customer feedback polarity. This will trigger Lambda function to invoke Comprehend, and analyze the feedback related to customer of that specific restaurant. In this scenario, feedback for analysis should be pulled from database based on Restaurant ID.

6. **Visualization:**
    - **Login Statistics:** In this module, the login statistics or traffic of Restaurant, and users should be visualized using some appropriate graphs or charts.
    - **Recipe uploaded:** Recipe uploaded by restaurant should be visualized by appropriate graphs.
    - Both visualizations can be embedded to Restaurant login page or restaurant admin page.
    - To create the visualization, you will need to use GCP Data Studio, and embed data studio dashboard in your webpage.

7. **Message Passing:** This module sends messages between services using GCP Pub/Sub. E.g., You have configured and written a service ***app1***, which needs to establish communication for sending data to another service that you created ***app2***. To achieve this ***app1*** should publish messages to a topic (predefined or dynamic topic depending on context), which will be received by ***app2*** in an asynchronous manner.
    - This module will work in the background and will not be directly visible from front-end
    - E.g., **scenario 1**: when AWS Lex initiates chatroom service using Firebase - AWS Lex can publish a message to a topic called "customer_complaints", this message will be pulled by a cloud function that will be responsible to initiate the chatroom operation. In this scenario, Lex will be the publisher, and the cloud function will be the subscriber, which will subscribe to the topic "customer_complaints".
    - E.g., **scenario 2**: Once the chatroom communication is over between restaurant agent, and a customer – The chatroom service will publish a message to a topic "communication_history", which will be pulled by the Customer service at the time of customer login. In this scenario, chatroom service, which is a cloud function will be the publisher, and the microservice responsible to handle user profile page will be the subscriber, which will subscribe to the topic "communication_history".

- This project requires writing of Test cases, and performing the following testing
    - Module testing to validate the functionality of each module (provide test cases, and screenshots of results, e.g., authentication failed, success etc.)
    - System testing to ensure the integration of the modules and completeness of the project. (provide test cases, and screenshots as evidence of testing)
- This project requires **extensive** and **systematic** documentation.
- Every team meeting must be logged with dates, and added as part of design document, and final report.