



**DALHOUSIE**  
UNIVERSITY

**CSCI 5410**  
**SERVERLESS DATA PROCESSING**

**“Quick Data Processor”**

**Sprint 2 Report**

**SDP#Team8**

Submitted By:

- Rushil Borad [rs519505@dal.ca]
- Mansi Kalathiya [mn410013@dal.ca]
- Bhavya Dave [bh392017@dal.ca]
- Avadh Rakholiya [AV786964@DAL.ca]

**GitLab Repository:** [https://git.cs.dal.ca/rakholiya/serverless\\_project](https://git.cs.dal.ca/rakholiya/serverless_project)

## Contents

Project Planning Overview .....	4
System Architecture Design .....	5
Architecture Overview .....	5
Module Implementations .....	6
Module 1: User Management & Authentication .....	6
Overview: .....	6
Implementation Details: .....	6
Testing and Validation: .....	6
Pseudo-code .....	11
Flowchart/activity Diagram .....	14
Module 4: Notifications .....	15
Overview: .....	15
Implementation Details: (partially implemented) .....	15
Testing and Validation: .....	15
Pseudo-code .....	17
Flowchart/activity Diagram .....	19
Module 2: Virtual Assistant .....	20
Overview .....	20
Implementation Details (partially implemented) .....	20
Testing and Validation .....	20
Pseudo-code .....	21
Flowchart/activity Diagram .....	22
Module 7: Frontend Web Application .....	23
Overview .....	23
Implementation Details .....	23
Testing and Validation .....	23
Gantt Chart .....	24
Conclusion and Next Steps .....	25
References .....	26

## Table of Figures

Figure 1: Sign-up form .....	7
Figure 2 : Sign Up with user details .....	7
Figure 3 : OTP Verification .....	8
Figure 4 : OTP Received through email .....	8
Figure 5 : Choose Security question.....	8
Figure 6 : List of security questions .....	9
Figure 7 : Set security question and answer .....	9
Figure 8 : Login Page .....	10
Figure 9 : Login Verification .....	10
Figure 10 : Architecture Diagram for authentication .....	14
Figure 11 : Login Sucessful email.....	16
Figure 12 : Registration successful email .....	16
Figure 13 : Architecture Diagram for authentication .....	19
Figure 14 : The chatbot UI on the website greeting and navigating .....	20
Figure 15 : The chatbot listing the feature of the website .....	21
Figure 16 : Clients query processing in Dialogflow .....	22
Figure 17 : Frontend Dashboard of the application.....	23

# Project Planning Overview

Our primary goal in Sprint 2 was to complete and integrate at least two essential modules—User Management & Authentication and Notifications—with partial completion of the Virtual Assistant module. This approach was crucial to set up a solid, interconnected foundation for the rest of the application's modules, allowing for streamlined development and smoother integration in future sprints.

## 1. Modules Planned for Sprint 2:

- **Module 1:** User Management & Authentication
- **Module 2:** Virtual Assistant (partial completion)
- **Module 4:** Notifications
- **Module 7:** Initial setup for the frontend web application

## 2. Rationale for Module Selection in Sprint 2:

The chosen modules play pivotal roles in establishing core application functions such as user login/registration, authentication, and notification delivery.

Completing these modules early allowed us to:

- Define and validate the essential backend processes required for secure user onboarding and multi-factor authentication.
- Build initial user notification systems that are fundamental to user experience and engagement.
- Develop an early version of the frontend interface to help the integration and testing of other modules in later sprints.

## 3. Challenges Addressed:

In Sprint 2, we anticipated several dependencies, especially around data processing needs for the Virtual Assistant module. To address these, we focused on creating modular, scalable services that would allow for flexibility and future growth, reducing the risk of integration issues in later phases.

# System Architecture Design

## Architecture Overview

Our system architecture integrates AWS and GCP services to build a hybrid serverless application. This architecture is designed to maximize scalability, optimize costs, and enhance security while leveraging the specialized capabilities of each platform.

### 1. AWS Services:

- **Cognito** for multi-factor authentication in User Management.
- **Lambda** for handling third-factor math-based authentication and processing tasks.
- **SNS** for sending notifications (email) post user registration and login.
- **DynamoDB** as the primary storage solution for user information, chosen for its high availability and serverless model.

### 2. GCP Services:

- **Dialogflow** for the Virtual Assistant module, allowing automated user assistance and navigation.
- **Cloud Functions** for backend support in the Virtual Assistant and Message Passing functionalities.
- **Firestore** for real-time data storage and retrieval to support messaging and user queries.

### 3. Application Frontend:

- **React** was chosen for building the frontend due to its component-based structure, which provides an interactive and user-friendly interface. It also allows easy integration with backend APIs.
- **GCP Cloud Run** is planned for frontend deployment, allowing serverless hosting and seamless scalability.

### 4. Integration and Communication:

- AWS SNS is connected to the user registration workflow to trigger email notifications.
- GCP Pub/Sub will be used for messaging once the Message Passing module is ready in Sprint 3, ensuring asynchronous communication for customer concerns and query forwarding.

# Module Implementations

## Module 1: User Management & Authentication

### Overview:

This module handles the core user authentication flow, implementing secure registration, multi-factor authentication (MFA), and session management. The backend is primarily hosted on AWS using Cognito for two-factor authentication (user ID/password and security questions) with a third factor based on a math skill check through AWS Lambda.

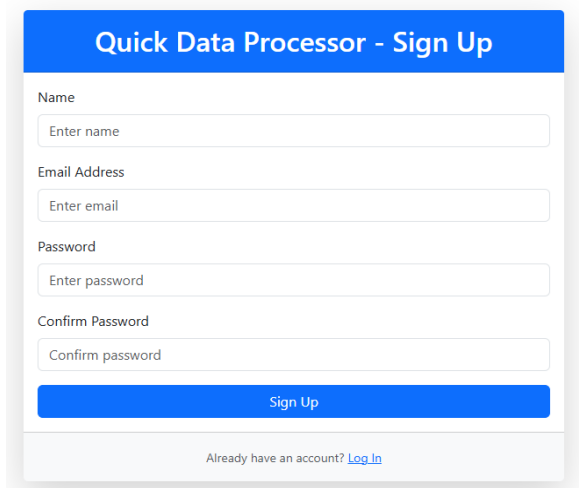
### Implementation Details:

- **User Registration:** Users register through the React frontend, triggering an API call to AWS Cognito. Upon successful registration, their information, including security questions, is stored in DynamoDB.
- **Multi-Factor Authentication (MFA):**
  - **First Factor:** Standard user ID and password verification.
  - **Second Factor:** Security questions, validated through DynamoDB to enhance user security.
  - **Third Factor:** A math problem generated via AWS Lambda, requiring correct validation before login is completed.
- **Session Management:** Cognito manages the session tokens, ensuring user sessions are secure and encrypted [1].

### Testing and Validation:

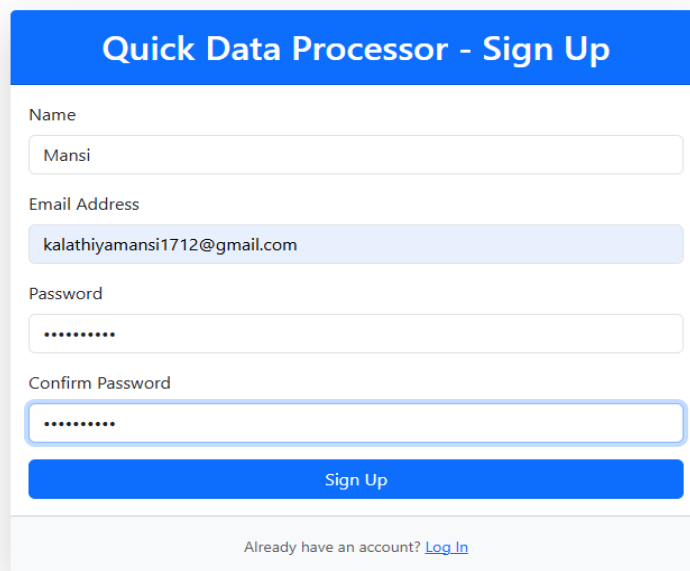
- **Functional Testing:** We tested each authentication factor independently to ensure they functioned correctly and reliably.
- **Screenshots:**
- **Validation Testing:** Multiple test cases were executed, including incorrect passwords, security answers, and invalid math solutions, to confirm the system correctly prevents unauthorized access.

## Screenshots:



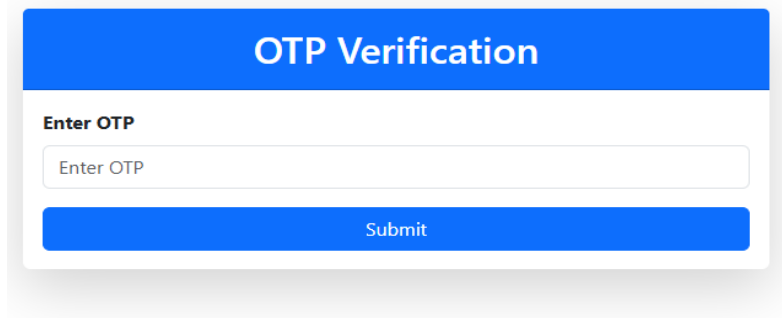
The screenshot shows a sign-up form titled "Quick Data Processor - Sign Up". It contains five input fields: "Name" (placeholder: "Enter name"), "Email Address" (placeholder: "Enter email"), "Password" (placeholder: "Enter password"), and "Confirm Password" (placeholder: "Confirm password"). A blue "Sign Up" button is at the bottom. Below the button, there is a link: "Already have an account? [Log In](#)".

Figure 1: Sign-up form



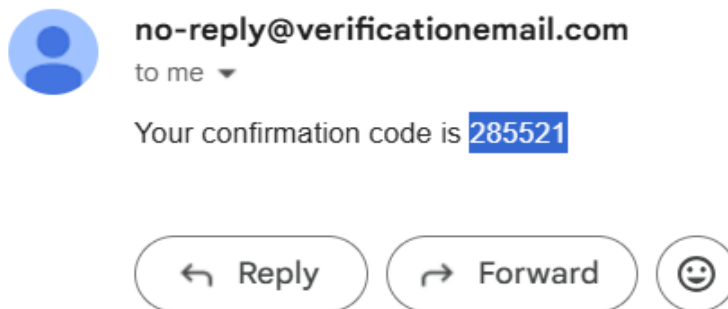
The screenshot shows the same sign-up form, but with user details filled in. The "Name" field contains "Mansi". The "Email Address" field contains "kalathiyamansi1712@gmail.com". The "Password" and "Confirm Password" fields contain masked text (dots). The blue "Sign Up" button is still present. Below the button, there is a link: "Already have an account? [Log In](#)".

Figure 2 : Sign Up with user details



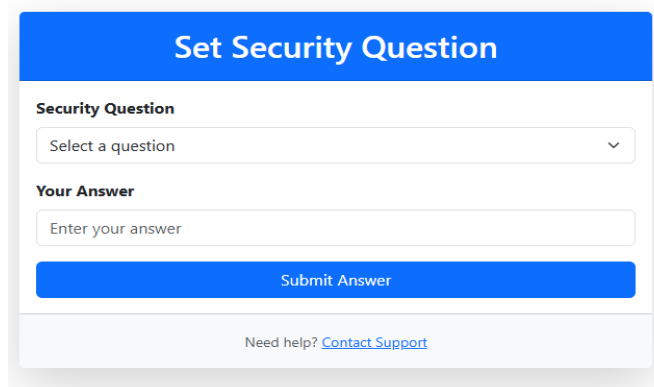
The image shows a web form titled "OTP Verification" with a blue header. Below the header, there is a section labeled "Enter OTP" containing a text input field with the placeholder text "Enter OTP". Below the input field is a blue button labeled "Submit".

Figure 3 : OTP Verification



The image shows an email interface. At the top, there is a blue circular profile icon next to the email address "no-reply@verificationemail.com". Below the address, it says "to me" with a dropdown arrow. The main body of the email states "Your confirmation code is 285521", where the code is highlighted in blue. At the bottom, there are three buttons: "Reply" (with a left arrow icon), "Forward" (with a right arrow icon), and a smiley face icon.

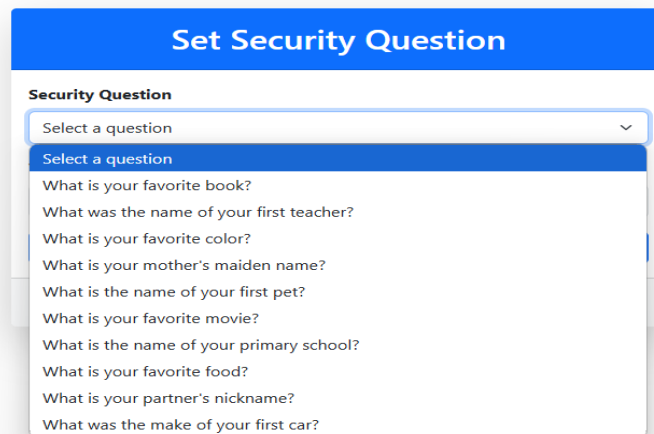
Figure 4 : OTP Received through email



The image shows a web form titled "Set Security Question" with a blue header. Below the header, there is a section labeled "Security Question" containing a dropdown menu with the text "Select a question" and a downward arrow. Below this is a section labeled "Your Answer" containing a text input field with the placeholder text "Enter your answer". Below the input field is a blue button labeled "Submit Answer". At the bottom of the form, there is a link that says "Need help? [Contact Support](#)".

Figure 5 : Choose Security question





**Set Security Question**

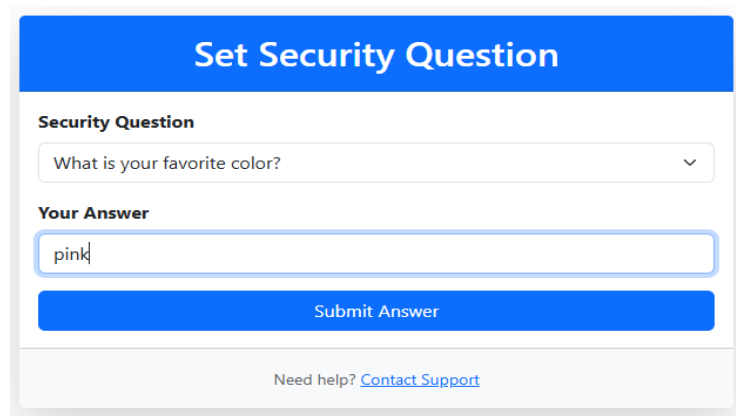
**Security Question**

Select a question ▼

Select a question

- What is your favorite book?
- What was the name of your first teacher?
- What is your favorite color?
- What is your mother's maiden name?
- What is the name of your first pet?
- What is your favorite movie?
- What is the name of your primary school?
- What is your favorite food?
- What is your partner's nickname?
- What was the make of your first car?

*Figure 6 : List of security questions*



**Set Security Question**

**Security Question**

What is your favorite color? ▼

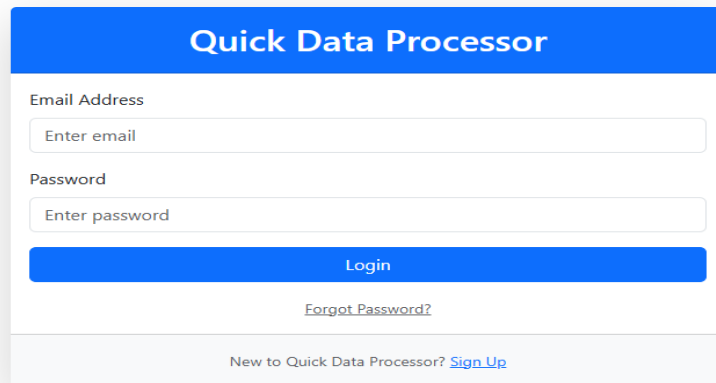
**Your Answer**

pink

**Submit Answer**

Need help? [Contact Support](#)

*Figure 7 : Set security question and answer*



The image shows a login page for 'Quick Data Processor'. It has a blue header with the title. Below it are two input fields for 'Email Address' and 'Password', each with a placeholder text. A blue 'Login' button is positioned below the password field. A link for 'Forgot Password?' is located below the login button. At the bottom, there is a link for 'Sign Up' for new users.

**Quick Data Processor**

Email Address  
Enter email

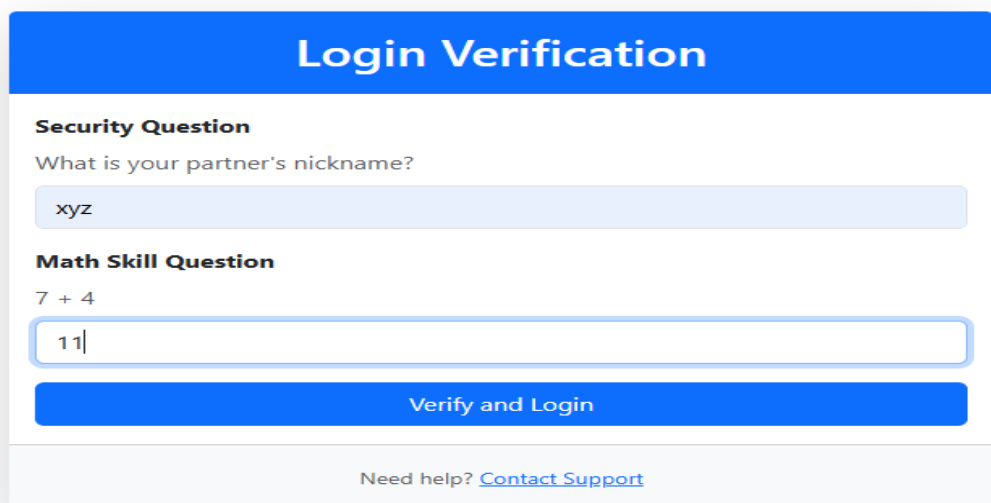
Password  
Enter password

Login

[Forgot Password?](#)

New to Quick Data Processor? [Sign Up](#)

Figure 8 : Login Page



The image shows a login verification page titled 'Login Verification'. It contains two sections: 'Security Question' with the question 'What is your partner's nickname?' and an input field containing 'xyz'; and 'Math Skill Question' with the question '7 + 4' and an input field containing '11'. A blue 'Verify and Login' button is at the bottom of the form. A link for 'Contact Support' is at the very bottom.

**Login Verification**

**Security Question**  
What is your partner's nickname?  
xyz

**Math Skill Question**  
7 + 4  
11

Verify and Login

Need help? [Contact Support](#)

Figure 9 : Login Verification

## Pseudo-code

### 1) CheckAnswer Function

- **Define Function verifyUserAnswer:**
  - Input: event
- **Extract Required Fields from the Event:**
  - Assign answer = event['ans']
  - Assign email = event['email']
- **Retrieve User Data from DynamoDB:**
  - Call getItemFromDynamoDB(email) and store the result in item.
- **Check if Item Exists:**
  - **If item is not found:**
    - Return an error response indicating "Item retrieval error".
- **Extract Real Answers:**
  - Assign realAnswers = item['ans'].
- **Compare the User's Answer with Real Answers:**
  - **If answer equals realAnswers:**
    - Return a success response indicating "Correct Answer".
  - **Else:**
    - Return an error response indicating "Incorrect Answer".

### 2) postAnswer Function

- **Define Function saveUserAnswer:**
  - Input: event
- **Extract Required Fields from the Event:**
  - Assign email = event['email']
  - Assign ans = event['ans']
- **Create a Dictionary items:**
  - Set "email" to the extracted email.
  - Set "ans" to the extracted ans.
- **Insert Data into DynamoDB:**
  - Call insertItemIntoDynamoDB(items) and store the result in response.
- **Return Success Response:**
  - Return a response with status code 200 indicating success.

### 3) Add user Function

- **Define Function addUser:**
  - Input: event
- **Wrap Logic in Try-Except Block:**
  - **Try Block:**
    - **Extract Fields from Event:**
      - Assign name = event['name']
      - Assign email = event['email']
      - Assign role = event['role']
      - Assign ques = event['queld']
    - **Log Adding User Details:**
      - Print "Adding user:" along with name, email, role, and ques.
    - **Create an item Dictionary with User Details:**
      - 'userId': Randomly generated 16-18 digit integer.
      - 'name': Set to name.
      - 'email': Set to email.
      - 'queld': Set to ques.
      - 'createdAt': Current UTC timestamp in ISO format.
      - 'role': Set to role.
    - **Log User Item:**
      - Print "User item:" along with the item dictionary.
    - **Save Item to DynamoDB Table:**
      - Use the put\_item method to save item.
    - **Create a Success Response:**
      - Dictionary with:
        - 'statusCode': 200.
        - 'body': Message "User added successfully".
    - **Return Success Response.**
  - **Except Block:**
    - **Log Error Details:**
      - Print "Error occurred:" along with error details.
    - **Create an Error Response:**
      - Dictionary with:
        - 'statusCode': 500.
        - 'body': JSON string containing:
          - 'message': "Failed to add user".
          - 'error': Error details.
    - **Return Error Response.**

#### 4) Security question Function

- **Define Function getAllSecurityQuestions:**
  - Input: event
- **Wrap Logic in Try-Except Block:**
  - **Try Block:**
    - **Perform Scan Operation:**
      - Scan the DynamoDB question table to retrieve all items.
    - **Extract Questions List:**
      - Extract the list of questions from the scan response and assign to questions.
    - **Create a Success Response:**
      - Return a dictionary with:
        - 'statusCode': 200.
        - 'headers':
          - 'Access-Control-Allow-Origin': '\*'.
          - 'Content-Type': 'application/json'.
        - 'body': The extracted questions list.
  - **Except Block:**
    - **Log Error Details:**
      - Print "Error retrieving questions" along with error details.
    - **Create an Error Response:**
      - Return a dictionary with:
        - 'statusCode': 500.
        - 'headers':
          - 'Access-Control-Allow-Origin': '\*'.
          - 'Content-Type': 'application/json'.
        - 'body': JSON string containing:
          - 'message': "Failed to retrieve questions".
          - 'error': Error details.

## Flowchart/activity Diagram

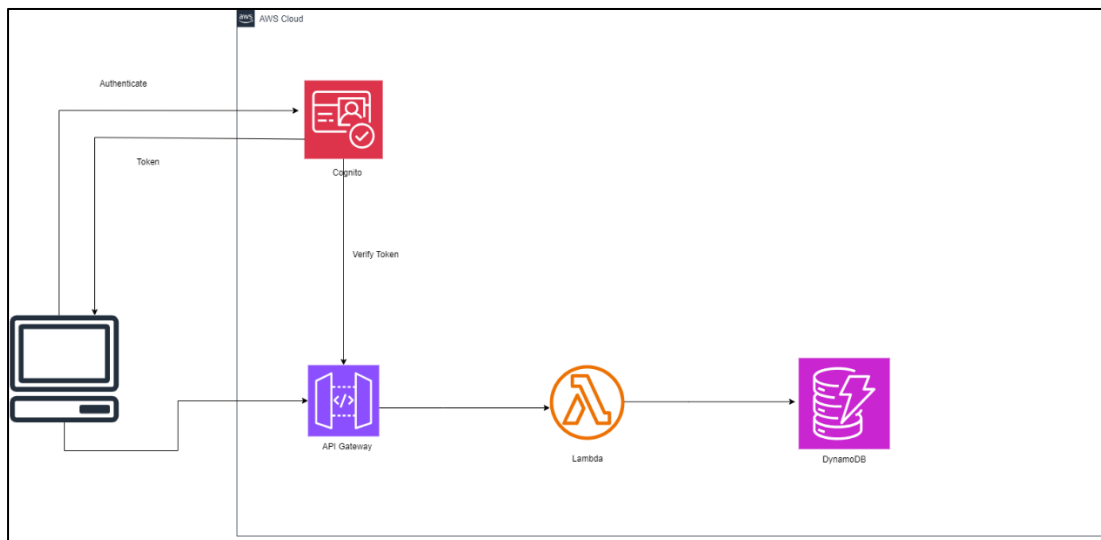


Figure 10 : Architecture Diagram for authentication

## Module 4: Notifications

### Overview:

This module provides email notifications for key user actions (registration and login) using AWS SNS. It allows users to receive immediate feedback on their actions, improving engagement and security awareness.

### Implementation Details: (partially implemented)

- **Registration Notification:** When a new user registers, SNS triggers an email notification confirming successful account creation [2].
- **Login Notification:** Each successful login also triggers an SNS notification to notify the user of the login event, adding an extra security layer.
- **Subscription Filters:** We configured filters to ensure each SNS topic only sends relevant messages, reducing unnecessary notifications.

### Testing and Validation:

- **Email Delivery Testing:** Verified that emails were sent and received successfully upon registration and login actions.
- **Error Handling:** Tested scenarios such as invalid email addresses or network interruptions to ensure notifications fail gracefully.

## Screenshots:

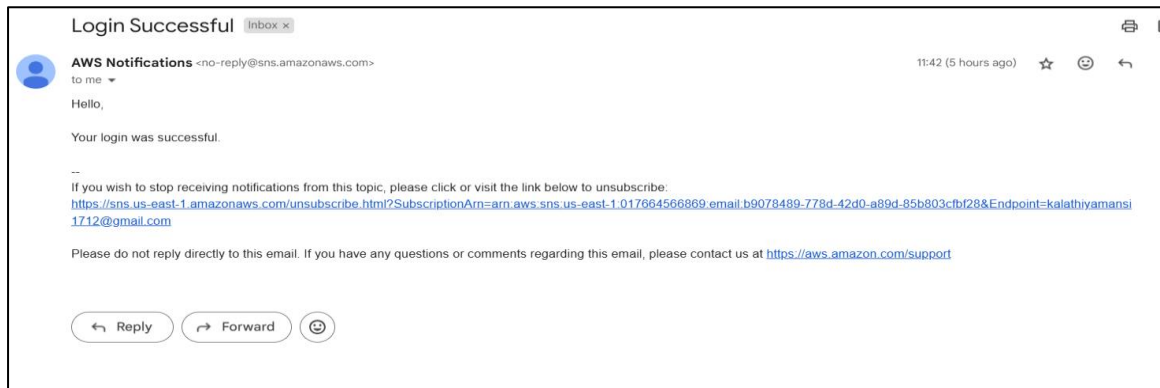


Figure 11 : Login Successful email

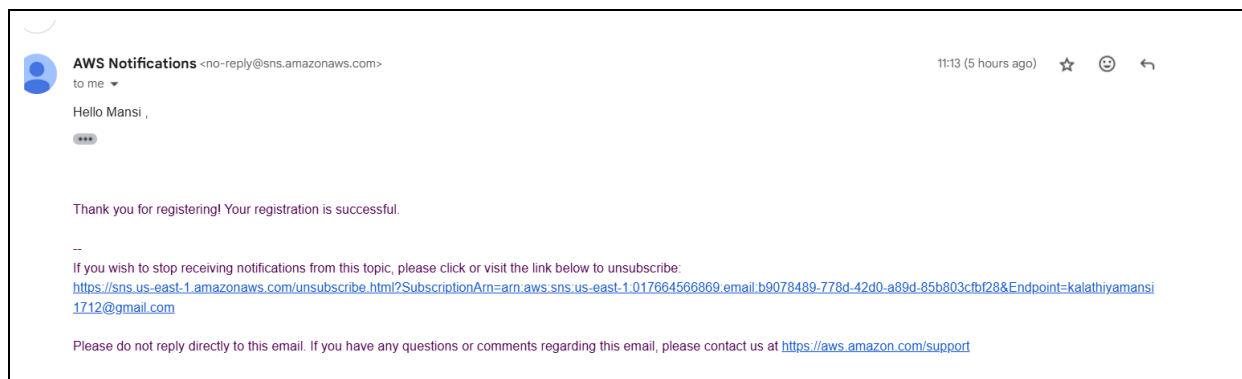


Figure 12 : Registration successful email



## Pseudo-code

### PostUser Lambda

- **Initialize Resources and Environment Variables:**
  - Connect to DynamoDB and get a reference to the user table.
  - Connect to SNS using the boto3 client.
  - Retrieve the SNS\_TOPIC\_ARN environment variable.
- **Lambda Handler Function:**
  - **Extract Inputs:**
    - Retrieve name, email, role, and queId from the incoming event.
  - **Log User Information:**
    - Print the user details to the logs for debugging.
  - **Prepare User Item:**
    - Generate a random userId.
    - Create a dictionary for the user with:
      - userId
      - name
      - email
      - queId
      - Current timestamp (createdAt)
      - role
    - Print the user item to logs.
  - **Save User in DynamoDB:**
    - Add the user item to the DynamoDB table.
  - **Subscribe User to SNS Topic:**
    - Use the SNS subscribe method to subscribe the user's email to the SNS topic.
    - Log the subscription response.
  - **Send Confirmation Email:**
    - Construct a confirmation message with the user's name.
    - Publish the message to the SNS topic with a subject like "Registration Successful."
  - **Return Success Response:**
    - Return a statusCode of 200 with a success message.
- **Error Handling:**
  - If any error occurs:
    - Log the error details.
    - Return a statusCode of 500 with an error message.

## GetUser Lambda

- **Log the Incoming Event:**
  - Print the event object for debugging.
- **Extract the Email Parameter:**
  - Attempt to retrieve email from queryStringParameters.
  - If not found, use the email field from the event body.
  - Log the extracted email.
- **Query DynamoDB Table:**
  - Query the user table using the email attribute with the email-index.
  - Retrieve the query result and extract the Items.
- **Check Query Results:**
  - **If user data is found:**
    - Convert the retrieved items to JSON-compatible format using decimal\_to\_int.
    - **Subscribe User to SNS:**
      - Use SNS to subscribe the user's email to the specified topic.
    - **Send Confirmation Email:**
      - Construct a "Login Successful" message.
      - Publish the message to the SNS topic.
    - **Return Success Response:**
      - Return a statusCode of 200 with user data in the response body.
  - **If no user data is found:**
    - Return a statusCode of 404 with a "User not found" message.
- **Handle Errors:**
  - If an exception occurs during any of the steps:
    - Log the error details.
    - Return a statusCode of 500 with an error message and the exception details.

## Flowchart/activity Diagram

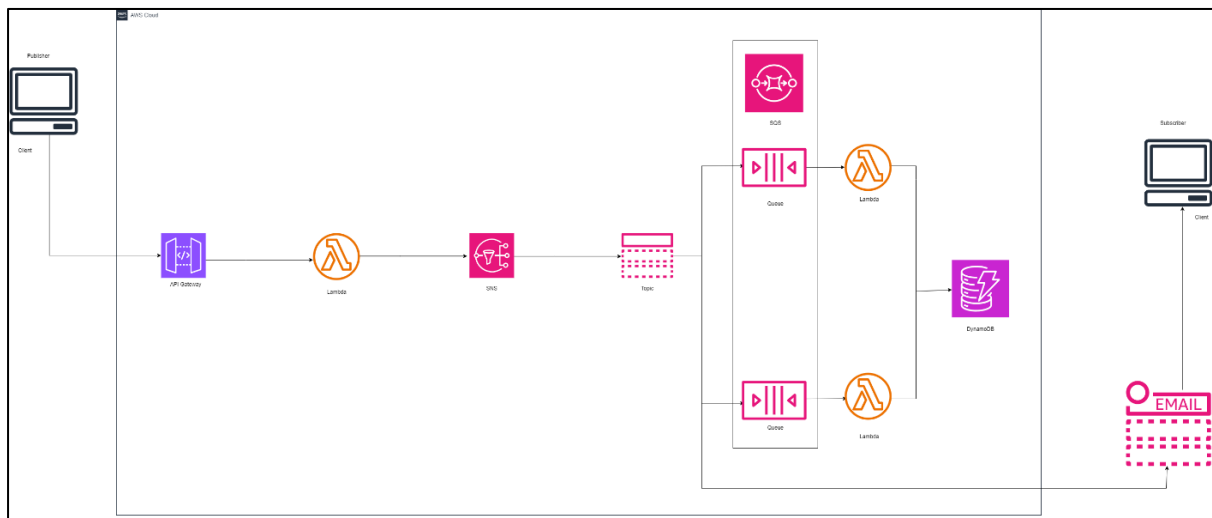


Figure 13 : Architecture Diagram for authentication

## Module 2: Virtual Assistant

### Overview

The Virtual Assistant module provides in-app assistance using **GCP Dialogflow** for natural language processing [3]. This chatbot offers users quick guidance, such as registration steps and navigation help, and handles queries related to processed data by retrieving files based on a reference code. Additionally, it collects customer concerns, which are forwarded to a QDP agent for further support.

### Implementation Details (partially implemented)

- Setup of Dialogflow and Firestore integration for basic query handling.
- Built initial intents for common queries such as "how to register" and assisting user for any other website related queries.
- Implementation will continue into Sprint 3 once the Data Processing module (Module 5) is ready to support advanced queries.

### Testing and Validation

- **Functional Testing:** Queries were tested to verify the accuracy of responses. Dialogflow intents were adjusted based on user interactions to improve response quality.

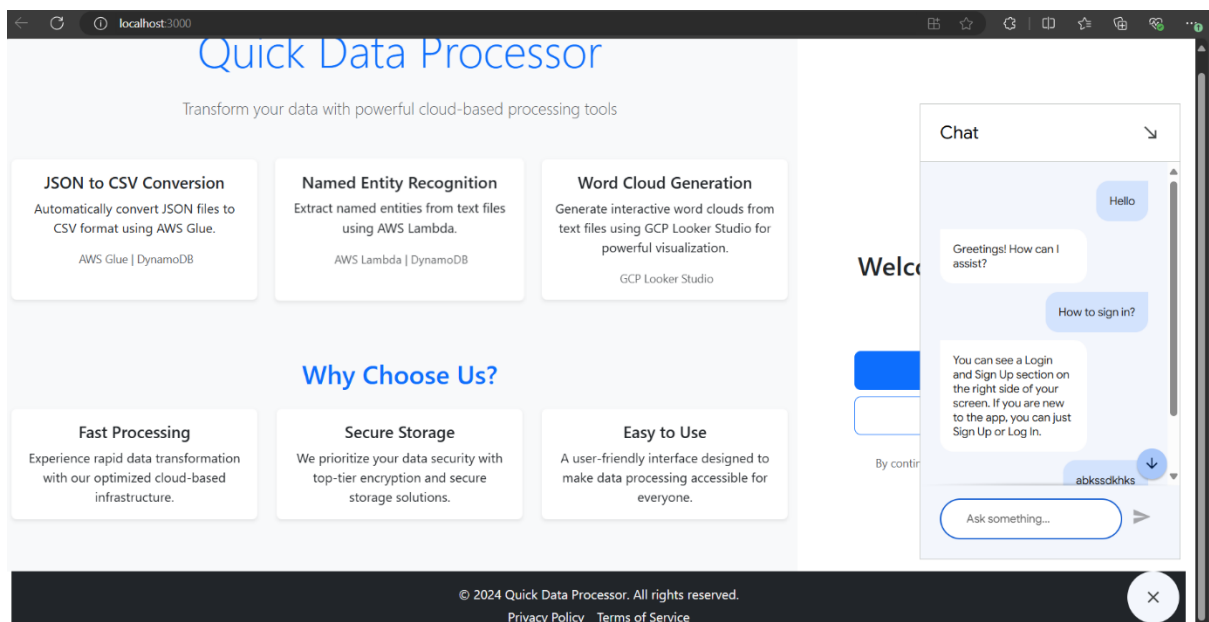


Figure 14 : The chatbot UI on the website greeting and navigating

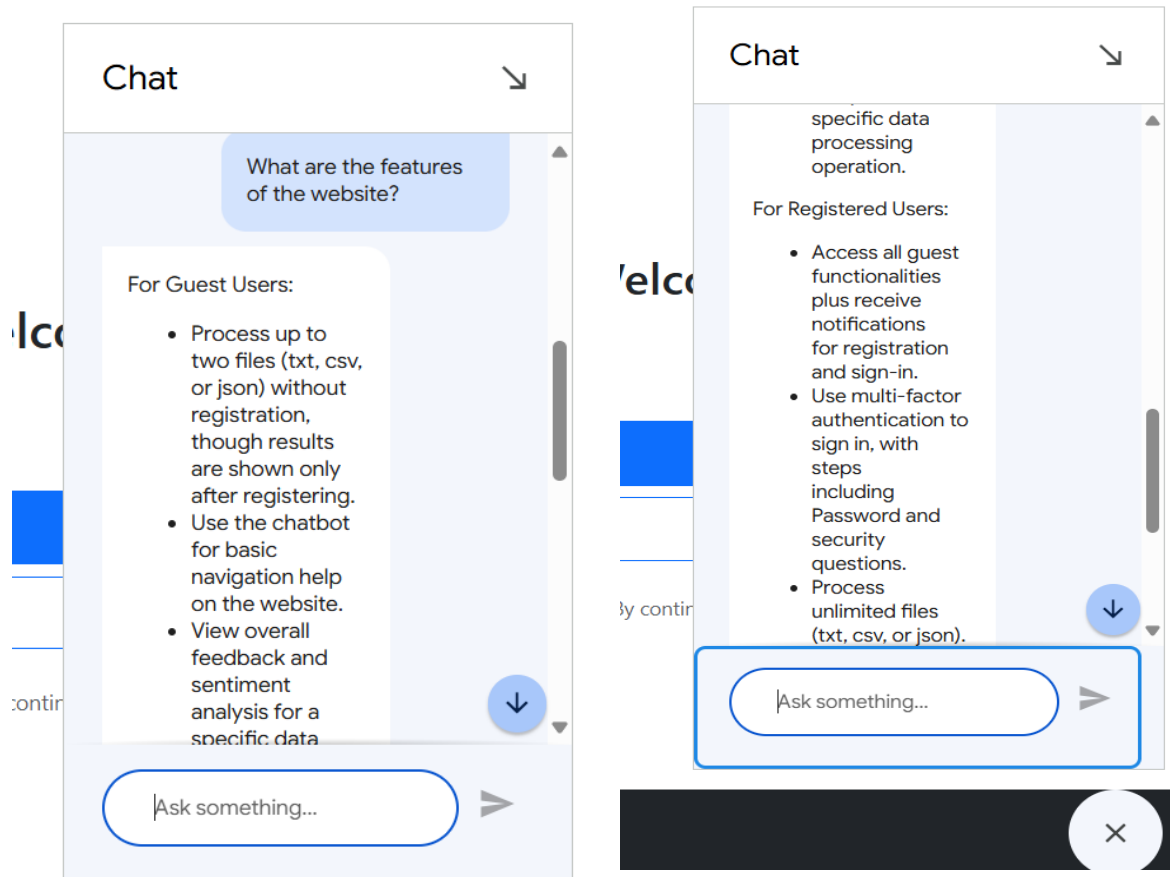


Figure 15 : The chatbot listing the feature of the website

## Pseudo-code

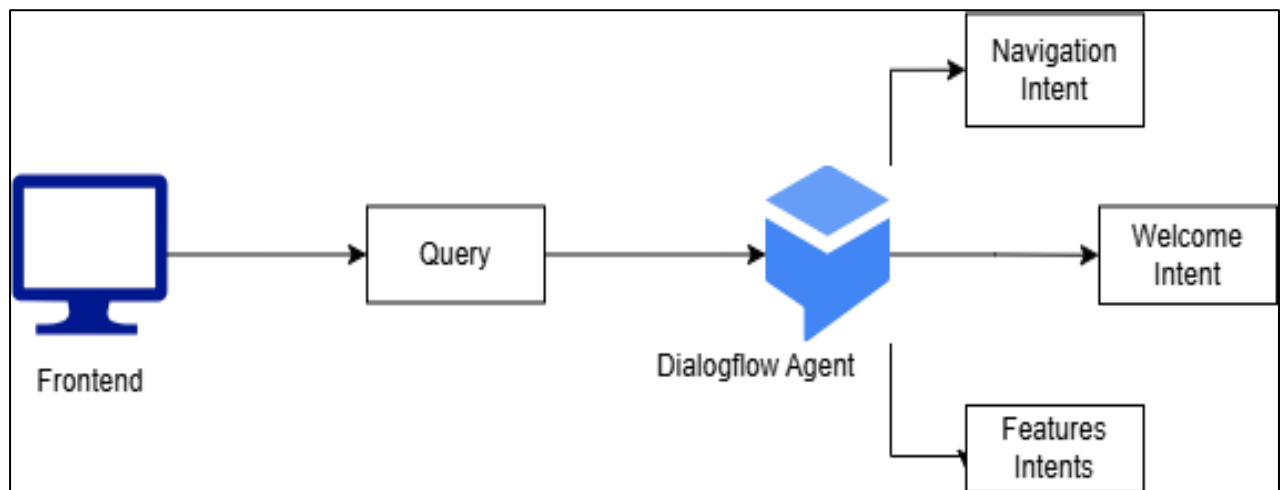
Initialize useEffect to run once on component load:

- **Check if Dialogflow Messenger has already been loaded:**
  - If not:
    - **Mark that Dialogflow Messenger has been loaded** to prevent multiple loads.
    - **Create a <link> element for Dialogflow Messenger styles:**
      - Set the rel attribute to "stylesheet".
      - Set the href attribute to Dialogflow Messenger stylesheet URL.
      - Append the <link> element to the document head.
    - **Create a <script> element for Dialogflow Messenger script:**
      - Set the src attribute to Dialogflow Messenger script URL.
      - Set async to true for non-blocking load.
      - Set crossOrigin to "anonymous" for cross-origin support.
    - **On script load:**
      - Create the <df-messenger> component:
        - Set location attribute (e.g., "us-central1").

- Set project-id attribute to "quickdataprocessor" (or actual project ID).
- Set agent-id attribute to Dialogflow agent ID.
- Set language-code attribute to "en".
- Append the <df-messenger> component to the document body.
- Append the <script> element to the document body to load the Dialogflow Messenger script.

**End useEffect.**

### Flowchart/activity Diagram



*Figure 16 : Clients query processing in Dialogflow*

## Module 7: Frontend Web Application

### Overview

The frontend, built using **React** and deployed via **GCP Cloud Run**, is the main user interface for QDP [4]. The application provides a seamless and secure environment for user registration, login, and other interactions. It also integrates with backend services, allowing dynamic content updates and real-time feedback.

### Implementation Details

- **Initial Setup & Authentication:** Complete authentication is set up using AWS Cognito, enabling secure login and registration within the React app [5].
  - The frontend communicates with AWS services to retrieve user information and manage sessions.
- **Modular Structure:** The frontend is designed with reusable components, separate pages and react hooks, making it easier to integrate upcoming modules, such as data processing and notifications.
- **User Experience:** The application has a responsive design, allowing users to access the platform on various devices while maintaining consistency in look and feel.

### Testing and Validation

- Tested the frontend thoroughly and validated all the buttons, tabs and features. Tested all the components separately with different interactions.

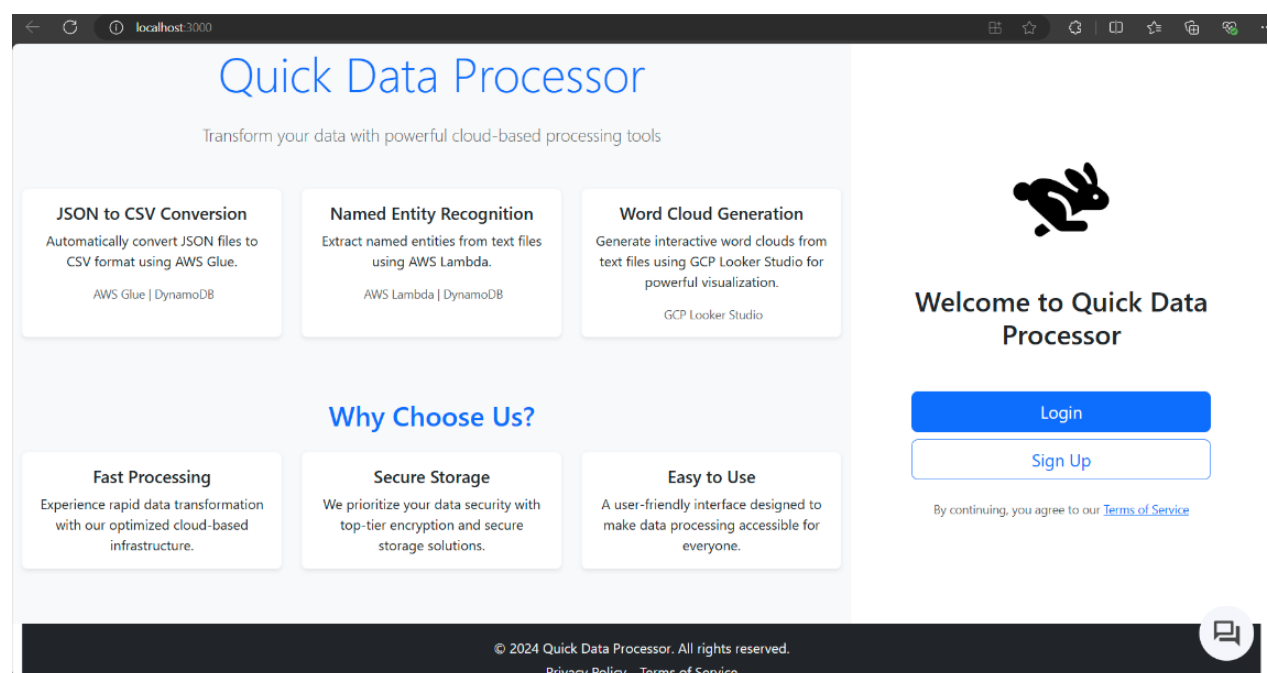


Figure 17 : Frontend Dashboard of the application

# Gantt Chart

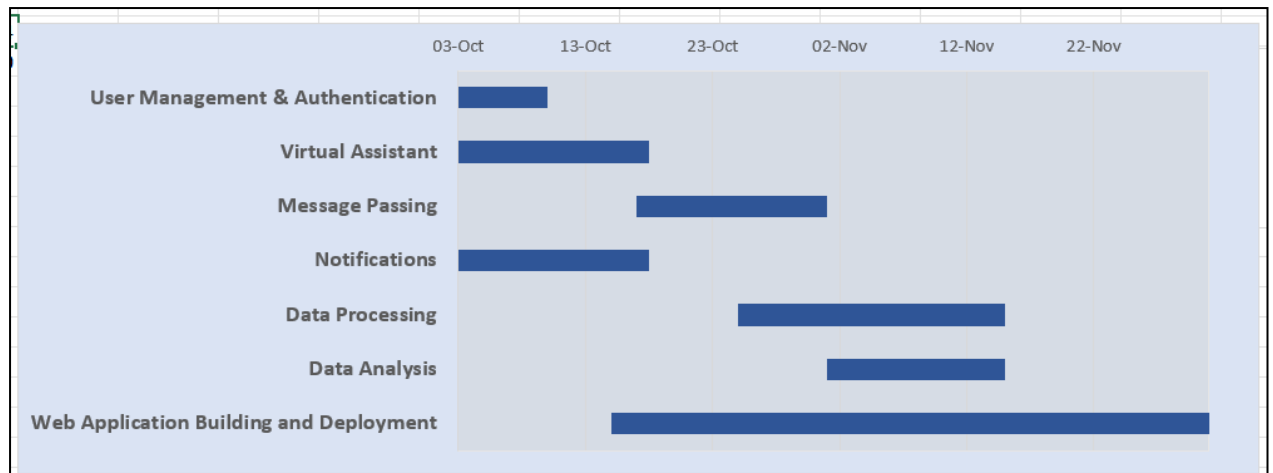


Figure 17: Gantt Chart



## Conclusion and Next Steps

In Sprint 2, we completed core foundational modules that enable secure authentication, user notifications, and basic UI functionality. These components are integral to future development, as they support user-facing features and secure backend processes that other modules will build upon. Moving into Sprint 3, our focus will be on enhancing inter-module communication and expanding backend processing, which will unlock the full capabilities of the QDP application.

### **Next Steps:**

- Complete Modules 3, 5, 6 and remaining parts of other Modules.
- Conduct integration testing to validate that Modules 1, 4, and 7 function cohesively with newly implemented features.
- Refine UI/UX for enhanced usability as we integrate more modules.

## References

- [1] DharmilVakani, "User sign-up and session management using AWS Cognito," *Simform Engineering*, 18-Apr-2023. [Online]. Available: <https://medium.com/simform-engineering/user-sign-up-and-session-management-using-aws-cognito-6311062c095>. [Accessed: 04-Nov-2024].
- [2] *Amazon.com*. [Online]. Available: <https://docs.aws.amazon.com/sns/latest/dg/sns-email-notifications.html>. [Accessed: 02-Nov-2024].
- [3] "Conversational agents and dialogflow," *Google Cloud*. [Online]. Available: <https://cloud.google.com/products/conversational-agents>. [Accessed: 01-Nov-2024].
- [4] "Cloud run," *Google Cloud*. [Online]. Available: <https://cloud.google.com/run>. [Accessed: 02-Nov-2024].
- [5] *Amazon.com*. [Online]. Available: <https://docs.aws.amazon.com/cognito/latest/developerguide/authentication.html>. [Accessed: 01-Nov-2024].