A) System Architecture Overview

For our final project, we designed and developed a comprehensive mobile health application named StressEase, aimed at monitoring and managing user stress levels. The application leverages the capabilities of the Internet of Medical Things (IoMT) by integrating physiological data from wearable devices such as Fitbit with user self-reported stress input.

Our architecture follows a modular, client-server model utilizing Flutter for the frontend, AWS services for the backend, and Fitbit's Web API for collecting wearable health data. The system ensures secure communication between components, scalable performance, and ease of integration with cloud-native services.

Key Components and Functionalities

1. Frontend: Flutter Mobile Application

The frontend of our system is developed using Flutter, a cross-platform mobile development framework. It provides the user interface and interaction capabilities. The mobile app allows users to:

- Sign up or log in securely
- Log daily stress check-ins using emoji-based mood ratings, notes, and symptoms
- Connect and sync Fitbit devices to fetch health metrics like heart rate, sleep duration, and step count
- View insights and reports, including a comparison between self-reported and smartwatch-measured stress levels
- Navigate across multiple screens via named route navigation and a bottom navigation bar for ease of use

We ensured that the user interface is intuitive, consistent across screens, and mobile-responsive to accommodate a variety of devices.

2. Backend: AWS Cloud Services

We utilized AWS to design a fully serverless backend architecture to handle logic processing, data storage, and third-party API interactions. Key services include:

- **Amazon API Gateway**: Acts as the entry point for the mobile app to communicate with the backend. It securely routes HTTP requests to Lambda functions based on the endpoint.

- **AWS Lambda**: Serves as the execution environment for backend logic. We created several Lambda functions to:

    - Fetch heart rate and related data from the Fitbit API

    - Save daily stress check-in and wearable data to DynamoDB

    - Refresh expired Fitbit access tokens using the stored refresh tokens

    - Manage API responses and ensure secure data handling

- **Amazon DynamoDB**: A fast and scalable NoSQL database used to store:

    - **fitbitToken table** – Stores user IDs along with Fitbit access and refresh tokens, timestamps, and token types

    - **userStressData table** – Stores self-reported stress levels, symptoms, mood ratings, and associated wearable data

This backend architecture is highly scalable, cost-effective, and aligned with cloud-native best practices.

3. Third-Party Integration: Fitbit API

To incorporate real-time health data, we integrated with the **Fitbit Web API** using **OAuth 2.0 authorization**. This involved:

- Directing the user to authorize access to their Fitbit data

- Receiving an **authorization code** via redirect URI

- Exchanging the code for an **access token and refresh token**

- Using the access token to query Fitbit endpoints (e.g., heart rate logs)

- Storing and managing token lifecycle using refresh tokens saved in DynamoDB

The API integration enables our app to deliver accurate physiological data directly from a verified wearable device, enhancing the reliability of stress monitoring.

Component Interaction Workflow

The interaction between components can be summarized as follows:

1. **User Interaction**: The user logs in and chooses to enter a stress check-in or sync data.
2. **API Requests**: The app sends an API call through API Gateway.
3. **Lambda Processing**:
    - If fetching data, the Lambda function retrieves the user's token from DynamoDB, makes a secure request to the Fitbit API, and returns the response.
    - If saving data, it stores the user's inputs and fetched metrics into DynamoDB.
4. **Data Delivery**: The response is sent back to the frontend, where it's displayed via Flutter widgets.

B)Using a table to show the workload and responsibilities of your team member.

Name	Role	Responsibility	Task Descriptions
Saketha Kusu	Frontend Developer	Designed Flutter screens: login, create account, home page, and daily check-in UI	Worked on routing, emoji-based stress input UI, and navigation logic
Karunakar Uppalapati	Backend Developer	Implemented Lambda functions and API Gateway for heart rate fetching/saving	Developed getHeartRate and saveHeartRate logic and linked Fitbit API
Varshitha Reddy Davarapalli	Integration & Testing	Connected frontend to backend and handled data syncing from Fitbit	Worked on OAuth flow, token refresh logic, and tested API connections

C) Development Challenges and Solutions

What was the most challenging part of this application development for you?

One of the most challenging aspects of this project was implementing the OAuth 2.0 authentication flow for Fitbit API integration. Obtaining the authorization code, exchanging it for an access token and refresh token, and securely managing these tokens required a deep understanding of token-based authentication protocols, which was new to most of us.

Another major challenge was ensuring seamless communication between the frontend (Flutter app) and backend AWS services. Integrating the Lambda functions with API Gateway and linking them correctly with Flutter through HTTP requests required significant troubleshooting.

In addition, testing and debugging across multiple components—such as verifying data fetch from Fitbit, validating data storage in DynamoDB, and displaying correct results on the app UI—was complex, especially when handling expired tokens or missing data.

How did you solve these problems in the development and test stages?

To address the OAuth challenges, we thoroughly studied Fitbit's developer documentation and used tools like Postman to simulate the authorization and token exchange process before implementing it in code. We also learned how to encode client credentials in Base64, manage `redirect_uri`, and handle access token expiry by storing and refreshing tokens using DynamoDB.

For the backend integration, we broke the problem into smaller Lambda functions (e.g., `getHeartRate`, `saveHeartRate`) and tested each endpoint independently through API Gateway test tools before connecting them to the frontend. This modular approach helped us isolate and fix issues faster.

On the frontend, we used Flutter's FutureBuilder and Stateful widgets to display dynamic responses from the backend and show proper loading/error states. We also added navigation routes carefully to maintain a consistent flow across multiple pages.

Overall, we relied heavily on collaborative debugging, testing features step by step, and validating them using console logs, sample inputs, and emulator testing. Through consistent teamwork and resource-sharing, we overcame these challenges and successfully built a working IoMT-based stress monitoring system.