DALHOUSIE UNIVERSITY

Project Report

CSCI 5410 – SERVERLESS DATA PROCESSING

GROUP – 25

Harshil Shah	hs@dal.ca
Khushi Shah	kh482021@dal.ca
Mohit Sojitra	mh546676@dal.ca
Mansi Vaghasiya	mansi.vaghasiya@dal.ca
Edwin Adams	ed743899@dal.ca

Contents

LIST OF FIGURES

LIST OF TABLES

Feature Specification

1. User Authentication

How I planned it:

In this implementation, I planned to use AWS Cognito for user authentication and authorization [1].

User details, such as email, name, phone number, user type, and key (plain text), will be stored in AWS DynamoDB, which is a NoSQL database service. AWS Lambda, an event-driven serverless computing service, will handle storing user details to the database and generating and verifying cipher keys [2].

For the question-and-answer details, I will use GCP Firestore, a NoSQL document database provided by Google Cloud Platform (GCP). GCP Cloud Function, a serverless computing service, will be used to set and get question and answer details from the database.

How I built it:

Instead of AWS Cognito for user authentication and authorization, I've decided to use Firebase Authentication, which is a service provided by Google for user authentication. It is easier to implement and can work well with multiple sign-in services [2].

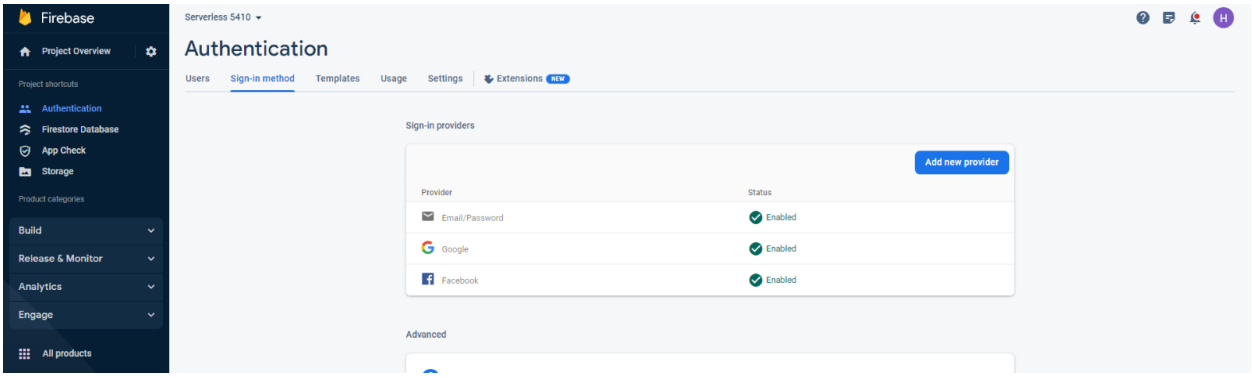


Figure 1: Firebase Authentication

User details, such as email, name, question answer and other user details, will be stored in AWS DynamoDB, which is a NoSQL database service. AWS Lambda, an event-driven serverless computing service, will handle storing user details to the database and generating and verifying cipher keys [3].

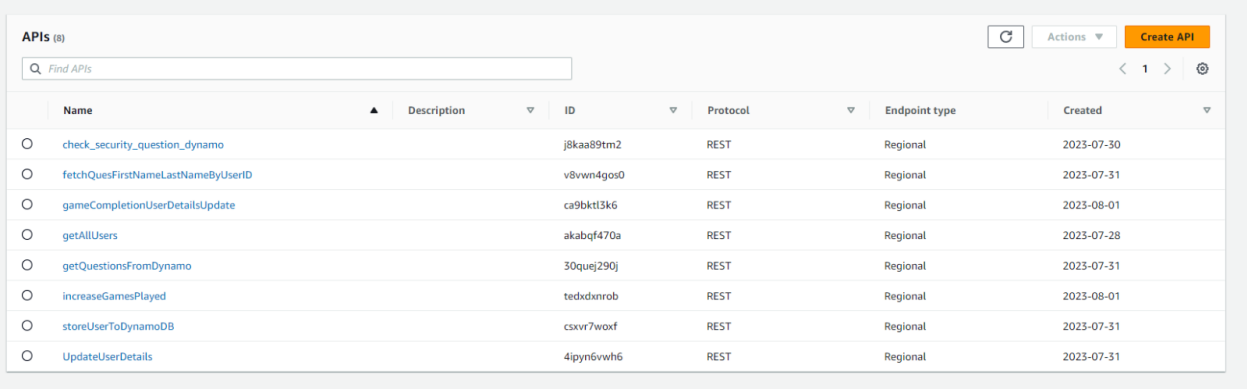


Figure 2: Lambdas

For the question-and-answer details, I wanted to use GCP Firestore, a NoSQL document database provided by Google Cloud Platform (GCP). However, I've decided to use AWS DynamoDB for storing the question-and-answer details as well, to maintain consistency within the AWS environment and reduce complexity [2].

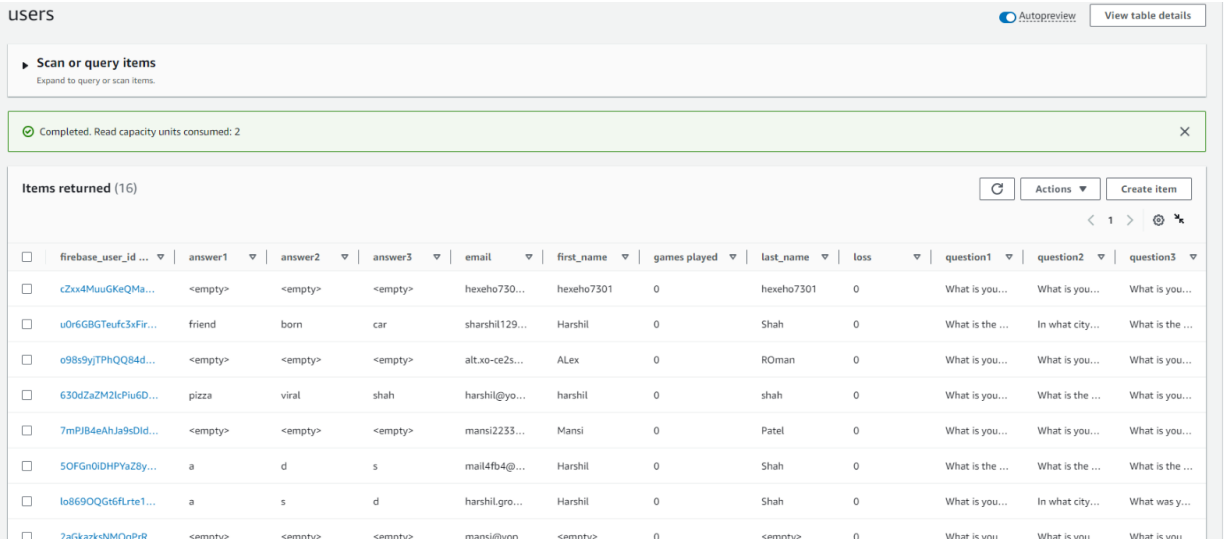


Figure 3: Users DynamoDB table

Finally, I've used GCP Buckets to store profile picture images and stored its URL to DynamoDB [4].

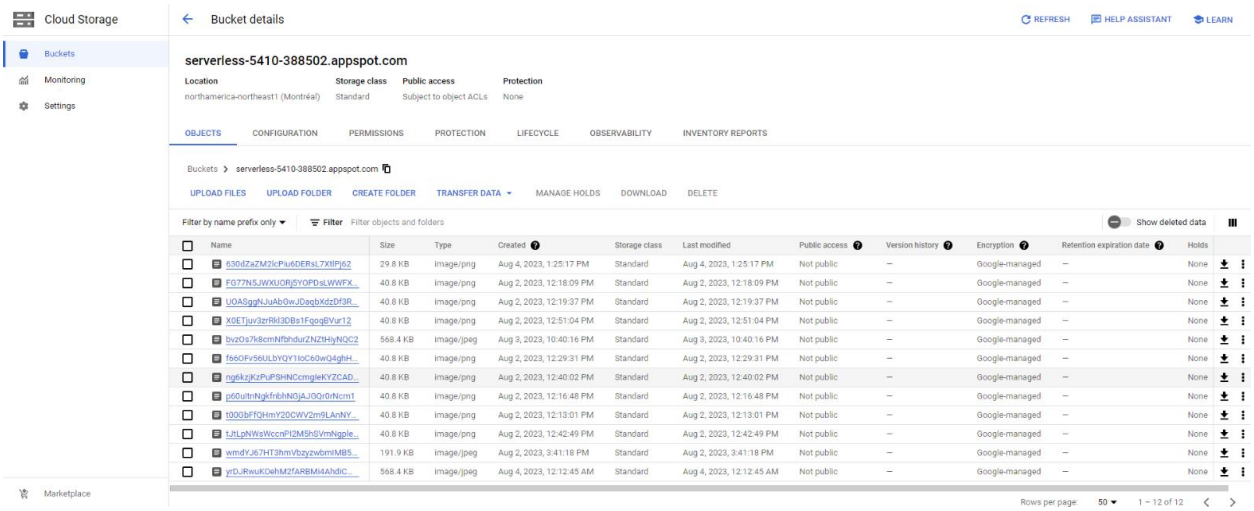


Figure 4: GCP bucket for user profile image storage

API Testing -

All API endpoints were tested using Postman with appropriate inputs of all possible

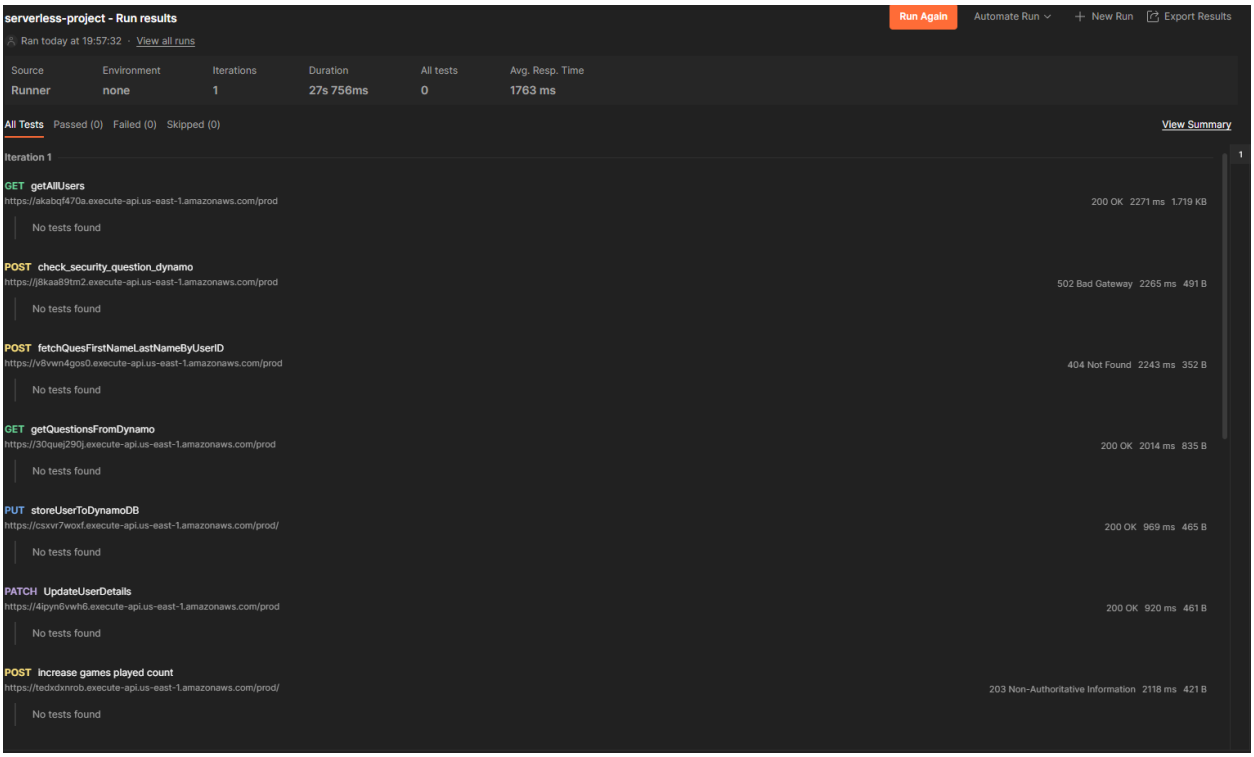


Figure 5: API testing via Postman

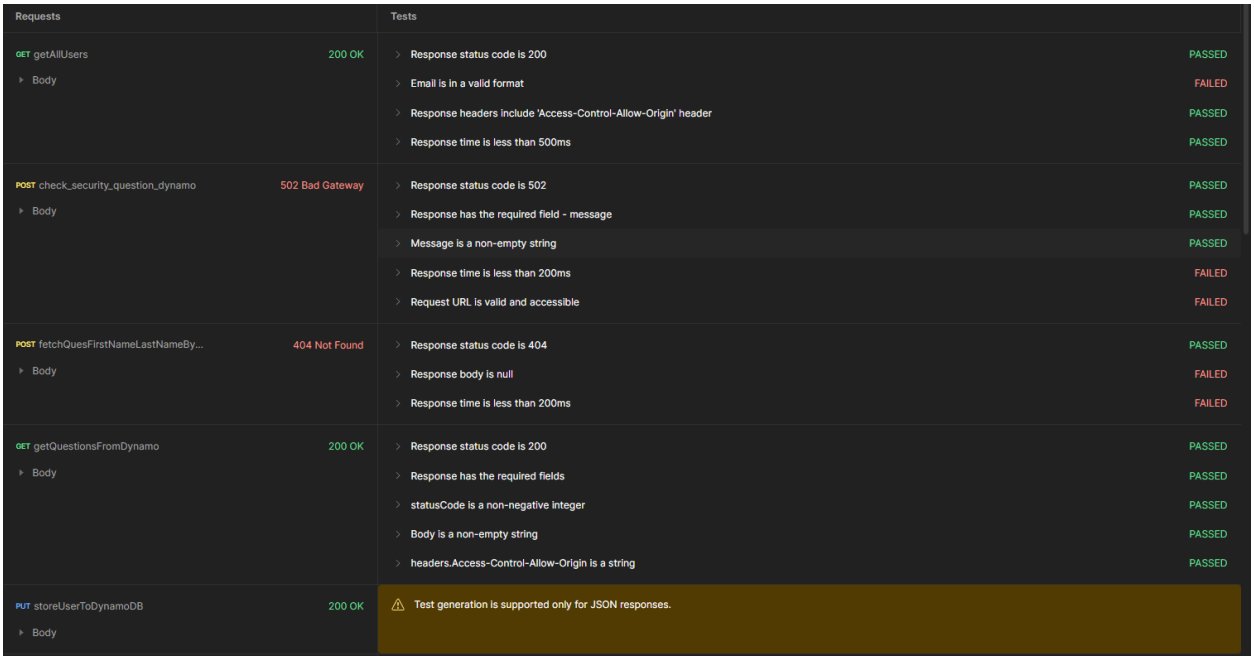


Figure 6: API testing via Postman 2

Table 1: Functional Testing Result

Request	**Type**	**Result**
getAllUsers	GET	PASS
check_security_question_dynamo	POST	PASS
fetchQuesFirstNameLastNameByUserID	POST	PASS
getQuestionsFromDynamo	GET	PASS
storeUserToDynamoDB	PUT	PASS
UpdateUserDetails	PATCH	PASS
increase games played count	POST	PASS
games Completion status update	POST	PASS
get games details by userid	GET	PASS
KHUSHI_sendmailonregister	POST	PASS

mohit_get_user_details	GET	PASS
khushi_mohit_req	GET	PASS
pubsub	POST	PASS
getteamsByUserId	GET	PASS
removefromteam	DEL	PASS

Table 2: Planned vs Used services

	Planned Service	Used Service
Auth	Cognito	Firebase Authentication
Database	Firestore	DynamoDB
Compute	Cloud Function	Lambda

Functional Test

Table 3: Functional Testing

Test Case	**Description**	**Expected**	**Results**
1	User doesn’t provide any credentials	System doesn't let users register	PASS

2	User provides an already registered email	System doesn't let users proceed	PASS
3	User enters correct registration details	User is taken to setting question form	PASS
4	User provides incorrect credentials during login	User is given invalid credentials error from backend	PASS
5	User provides correct email and password	System takes user to question verify pop up	PASS
6	User provides incorrect answer	System shows error to the end user	PASS
7	User provides correct answer	System takes user to cipher key verification	PASS
8	User provides incorrect cipher key	System shows error to the user	PASS

2. User Profile Management

How I planned it:

For our user interaction system, I have chosen to utilize GCP Firestore and Cloud Functions to build a scalable and efficient solution. To enable users to edit their personal information, I will create a Firestore collection to store user profiles. This collection will include fields for profile picture, name, contact information, and other relevant details. Additionally, Cloud Functions will be implemented to handle user profile updates securely, ensuring data integrity and authentication. For user statistics, such as games played, win/loss ratio, and total points earned, I plan to set up another Firestore collection dedicated to storing this data. Cloud Functions will be leveraged to calculate and update these statistics in real-time, ensuring users have access to up-to-date information. Furthermore, I will create a Firestore collection to

manage team affiliations, allowing users to join or leave teams seamlessly. Cloud Functions will play a vital role in handling these team-related interactions while enforcing necessary authorization checks. Lastly, to provide users with a way to view and compare achievements, I will design a Firestore collection to store achievement data. Cloud Functions will be responsible for retrieving and comparing achievements between users efficiently, offering an interactive and engaging experience.

How I built it:

During the implementation phase, I had to deviate from my original plan to use Firestore and cloud functions, as I have used DynamoDB in the User Authentication module, utilizing AWS DynamoDB and Lambda Functions to build our user interaction system. For editing personal information, I updated the DynamoDB table that I had set up in the first module that stores user profiles, which includes fields for profile picture, name, and security questions.

AWS Lambda Functions were developed to handle user profile updates securely, ensuring that only authorized users can modify their data via API gateway. To address user statistics, I created a DynamoDB table to store relevant metrics such as games played, win/loss ratio, and total points earned for each user. All predefined security questions are loaded from the following DynamoDB “questions” table at run time.

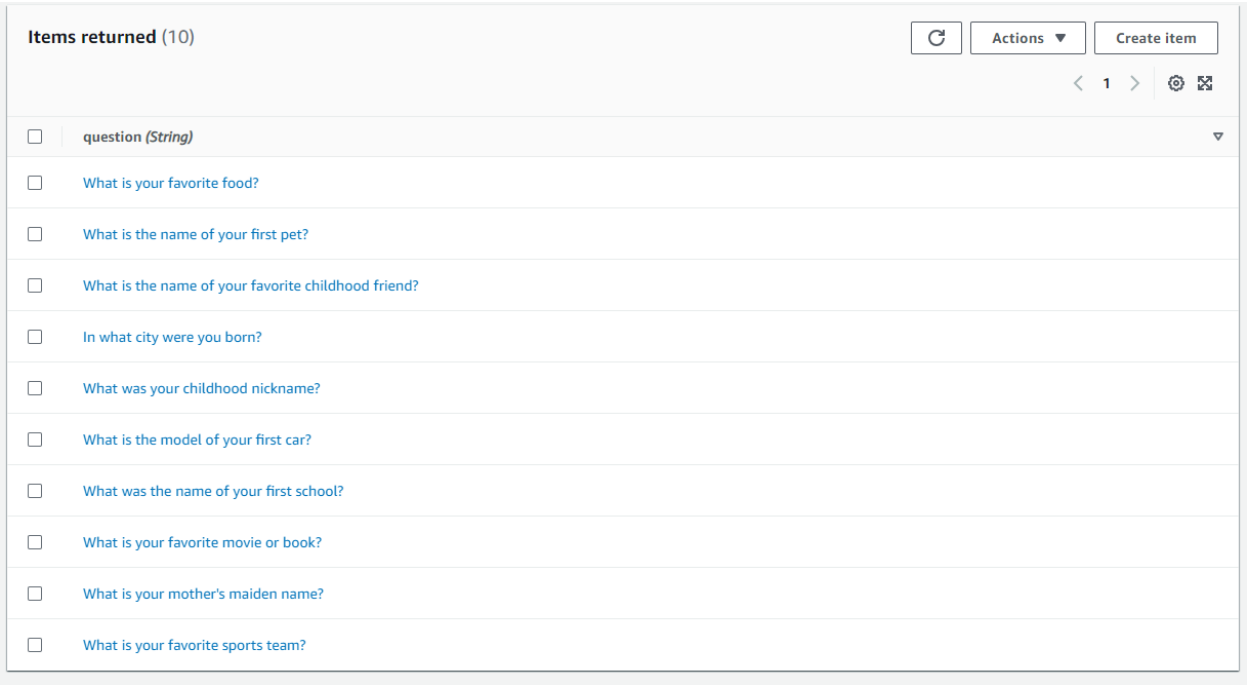


Figure 7: Questions DynamoDB table

AWS Lambda Functions were leveraged to perform calculations and update the statistics when games are completed or points are earned. For managing team affiliations, I designed a DynamoDB table for teams and another one to manage team affiliations for each user.

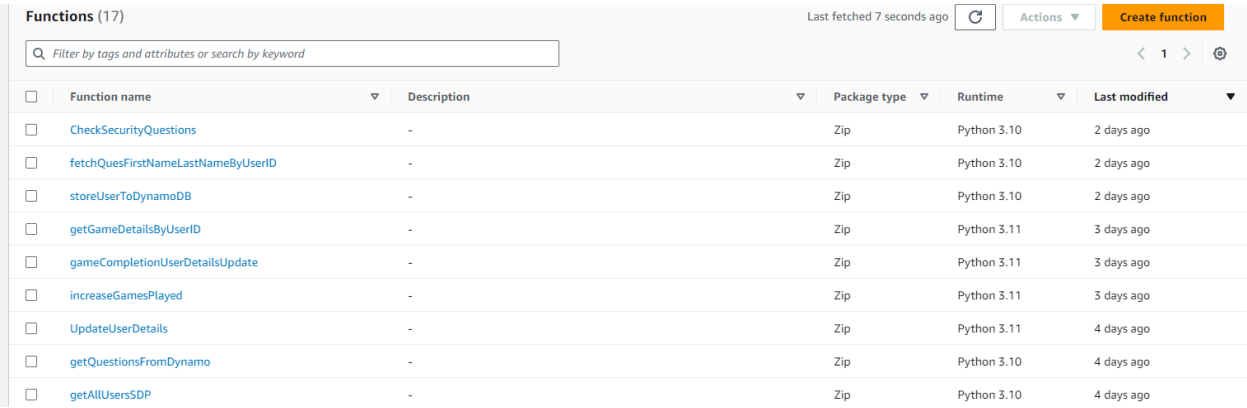


Figure 8: Lambdas

I called feature 3's API gateway to handle team leaving operations, ensuring data consistency and security. Lastly, to enable users to view and compare achievements, I stored the achievement data in DynamoDB. AWS Lambda Functions were responsible for fetching and comparing achievements between users efficiently, offering a seamless and rewarding experience to our users. With the AWS stack, our user interaction system is now scalable, robust, and well-integrated, providing an optimal solution to meet our requirements [4].

API Testing:

Used Postman to test all APIs whenever something is updated to ensure everything is working properly at all times.

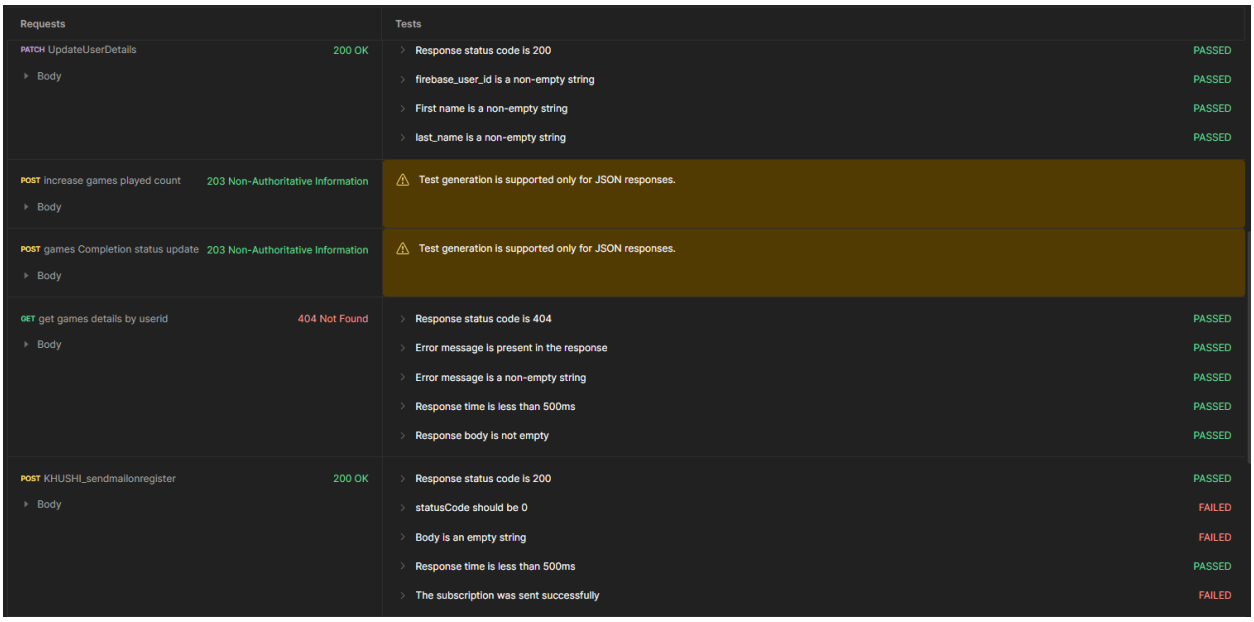


Figure 9: API Testing 1

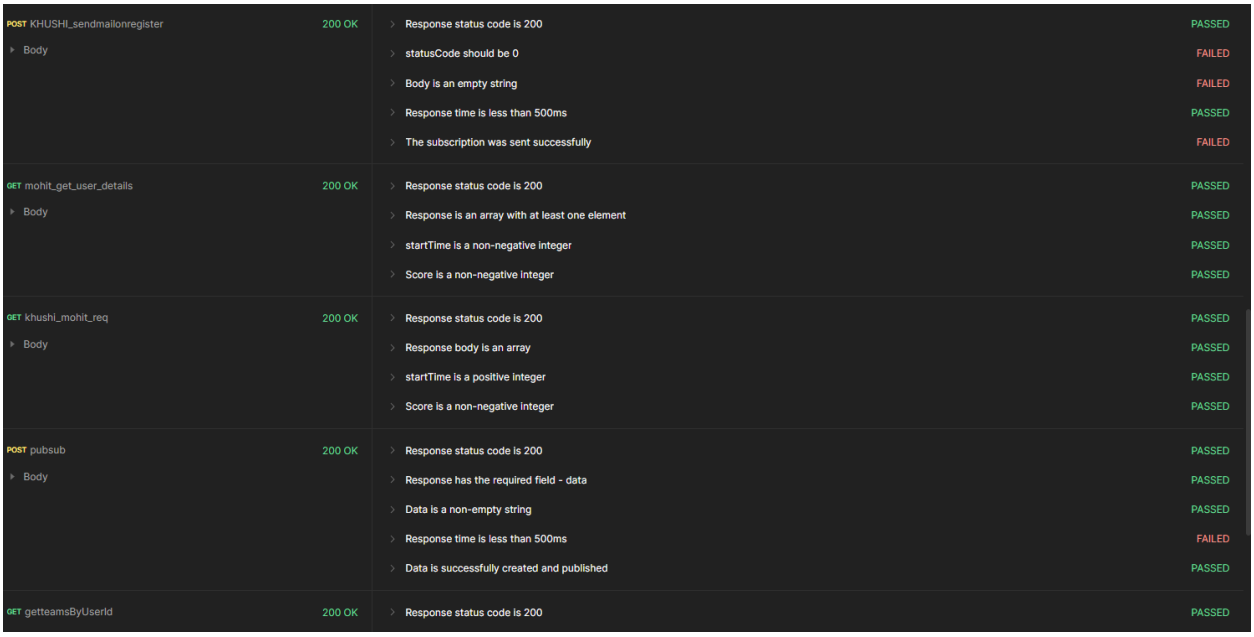


Figure 10: API Testing 2

Table 4: Functional Test Case Results

Test Case	Description	Expected	Results
1	User doesn’t provide proper first name	System doesn't let users update profile	PASS
2	User doesn’t provide proper last name	System doesn't let users proceed	PASS

Table 5: Used Service VS Planned Services

	Planned Service	Used Service
Database	Firestore	DynamoDB
Compute	Cloud Function	Lambda

3. Team Management

The central functionality of this module is to be able to create team and select users to add the users to that team and on clicking invite, all the selected users will receive an email notification to accept the invitation to the team. Upon selecting accept, that user will be added to the team and if declined that user will not be added to the team.

After the user is added to the team, there is a table where the Your Teams and Your participation teams are displayed. Here multiple teams will be displayed for the logged in user. Now selecting a particular team, the user will be navigated to the dashboard of that team. Here the team leader and team members are displayed. Along with that if the logged in user is a normal participant, they will get an option to leave the team. Similarly, if the logged in user is a team creator, they can remove anyone from the team and promote someone to admin but can't leave the team. And if the logged in user admin, they can remove anyone and can leave the team. Moreover, there is an option to view different team statistics. The team statistics displays data regarding the total games played by the team and a user-specific bifurcation regarding the games played by each user. Moreover, the win -loss ratio for the user is displayed and the total score by the team is displayed.

Furthermore, there is an option to navigate to different pages like explore games and view leaderboards.

The frontend of this feature is built in ReactJS, and the backend is serverless. All the logic is written in lambda with python programming language.

Planning of the feature:

The planning of this feature can be broken down in several tasks.

Requirements gathering:

Understanding this feature was quite complex and tricky. First, I developed a proper understanding of the module.

Then I looked for the services provided by the cloud provider which will help me to fulfill the expectations of this module.

Database design schema:

For this module, I've used DynamoDB to store data.

Building of the feature:

The building of the feature can be broken in several tasks.

Create Team:

This create team feature is built with Lambda, API Gateway and DynamoDB.

The requirement states that the team's name should be created dynamically every time.

For this, I have used the OpenAI API, which will be called by the Lambda associated with the API invoked at the create team button click.

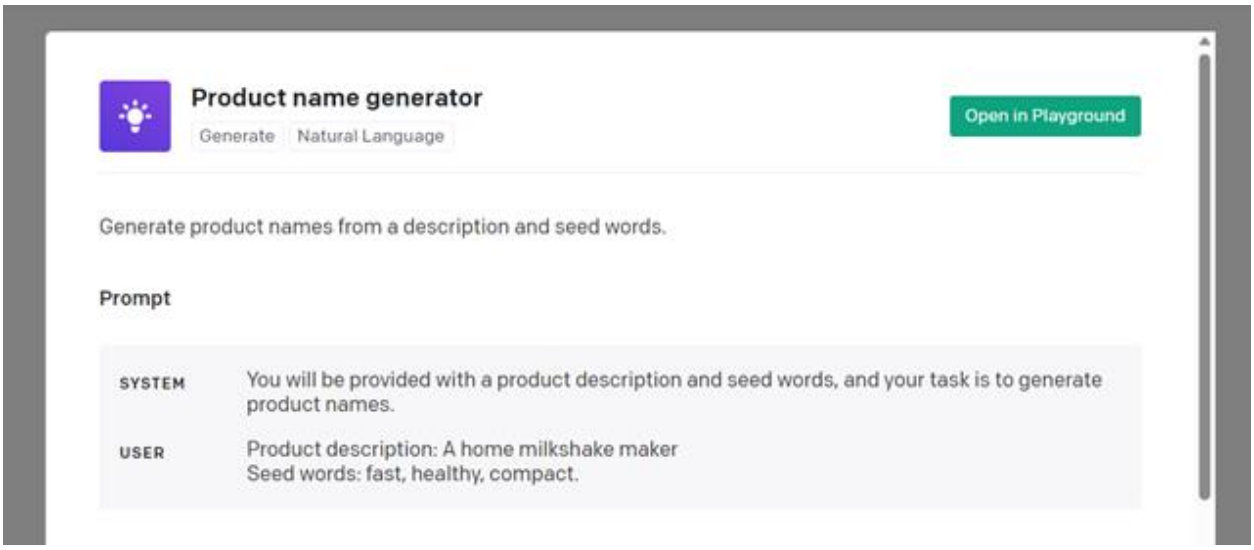


Figure 11: Tag Generation API

This API will always generate a unique one-word team name, and return the name, which will be shown in the dialog [5].

Later, the user will get the option to add the users to the team. Only the registered SNS users will be displayed in the dropdown and will be asked an email invite to accept/decline the team invite.

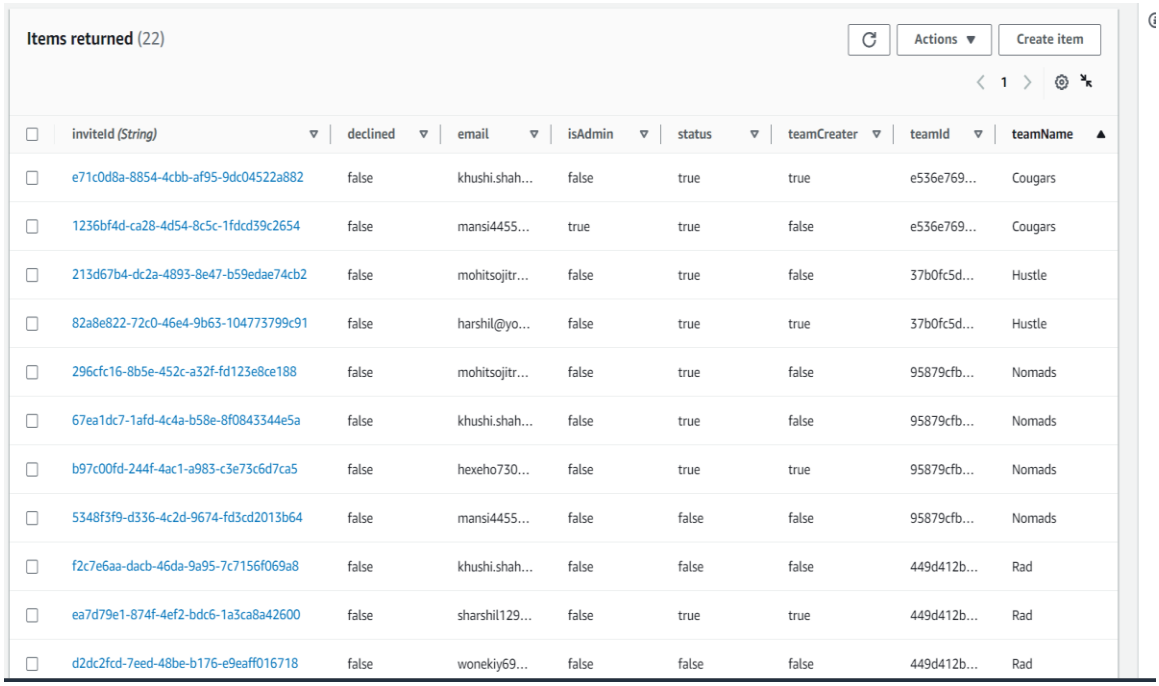


Figure 12: DynamoDB table – User Roles

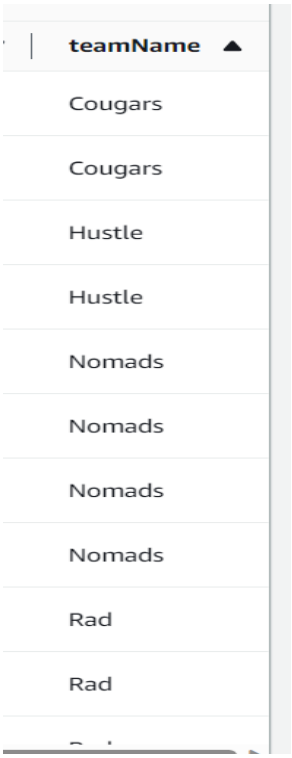


Figure 13: Store Generated Team Names using AI API - DynamoDB table

In the below screenshot you will demonstrate the functional testing of this feature.

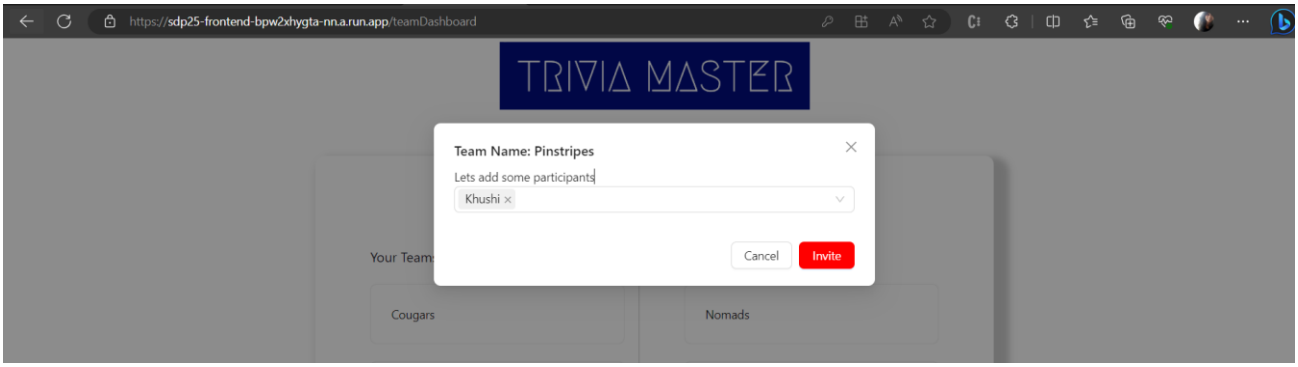


Figure 14: Functional Testing 1

In DynamoDB table teamNames and invite entry is added. And at the same time email will be given to the invitee [6].

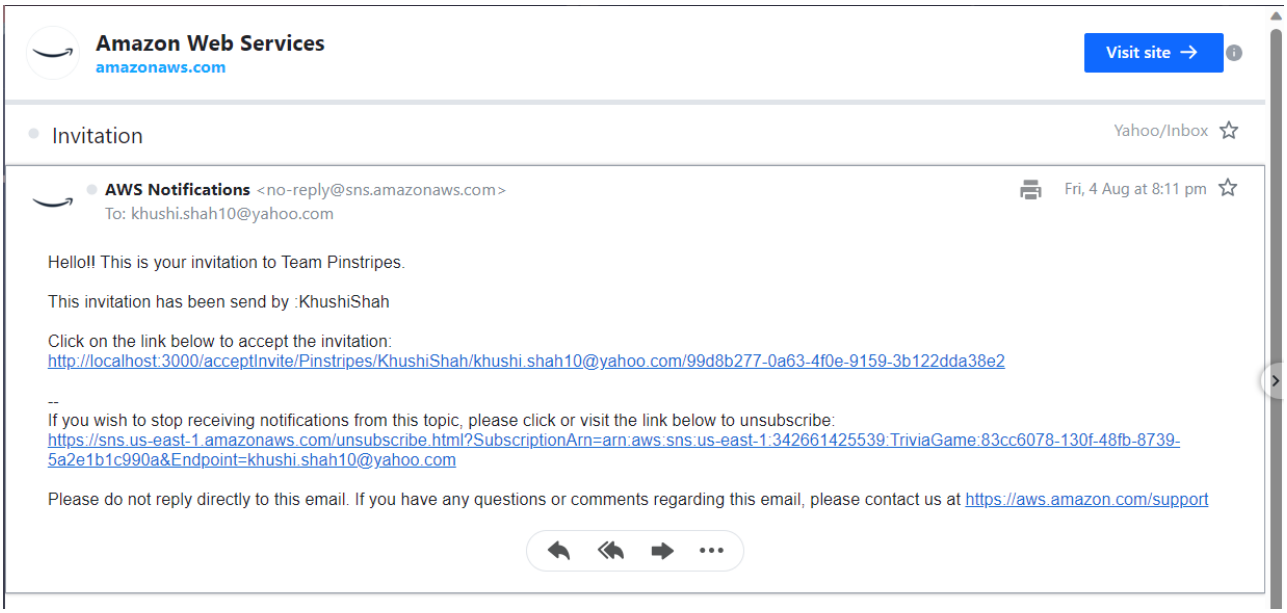


Figure 15: Invite entry mail on DynamoDB entry addition

Accepting/Declining Invitations:

To implement this feature, I've used Lambda, SNS, API gateway, DynamoDB.

Requested users will receive email with a link which will redirect to the accept/decline page.

There will be 2 buttons to accept or decline, and based on the users' choice the data will be updated in DynamoDB. If accepted then it will add them to the respective team.

Below are the steps where users will either accept or decline the team invite.

In the below screenshot you will demonstrate the functional testing of this feature [7].

Figure 16: Functional Testing 2 - Accept/Decline Invitation

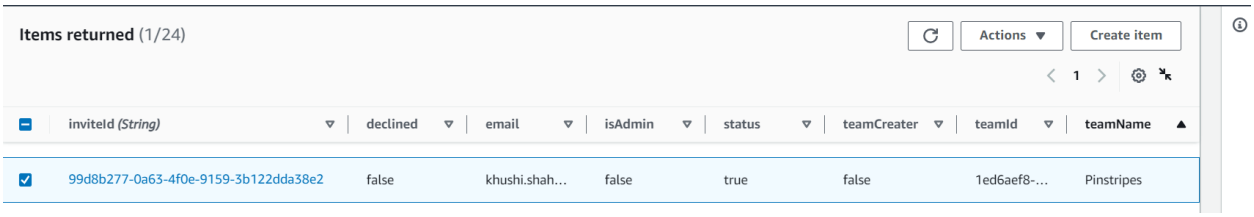


Figure 17: Functional Testing 3 - Invite Status

If the users decline then they won't be added to the team and here in the status it would be false and declined would be true.

Team Dashboard

To build the team dashboard, I used services like Lambda, API gateway and DynamoDB.

In this team dashboard, users will be able to see the teams they have created under "Your Teams" and the teams they have participated in "Your participations".

By clicking on the team names, they will be navigated to the team page [6].

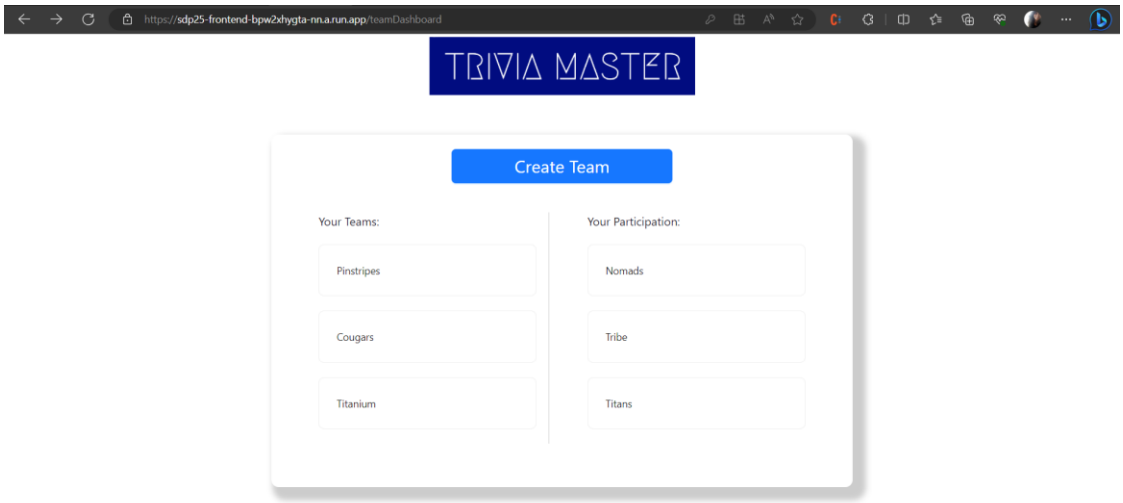


Figure 18: Team management

On the team dashboard, the team creator and other team participants details will be visible. And the team creator will get the option to promote the users to the admin or remove them. Also there is a view leaderboard option, which will store Team statistics like, games played, win-loss ratio and total points earned.

In the below screenshot you will demonstrate the functional testing of this feature.

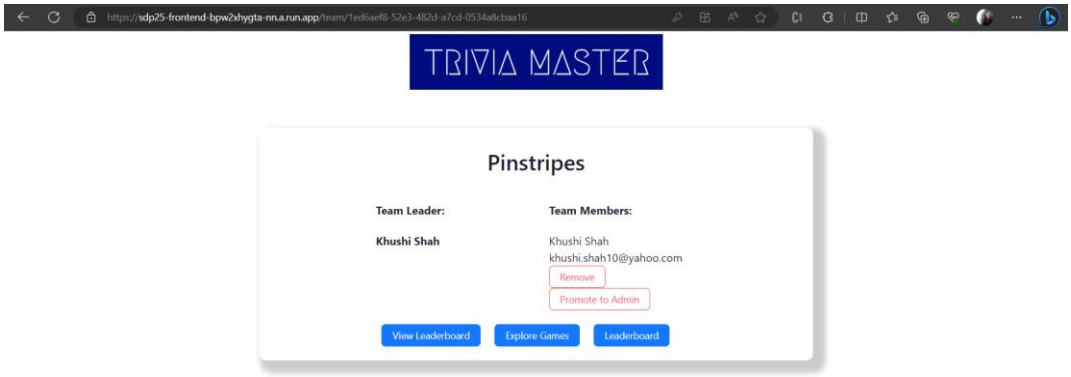


Figure 19: Functional Testing for Team management 1

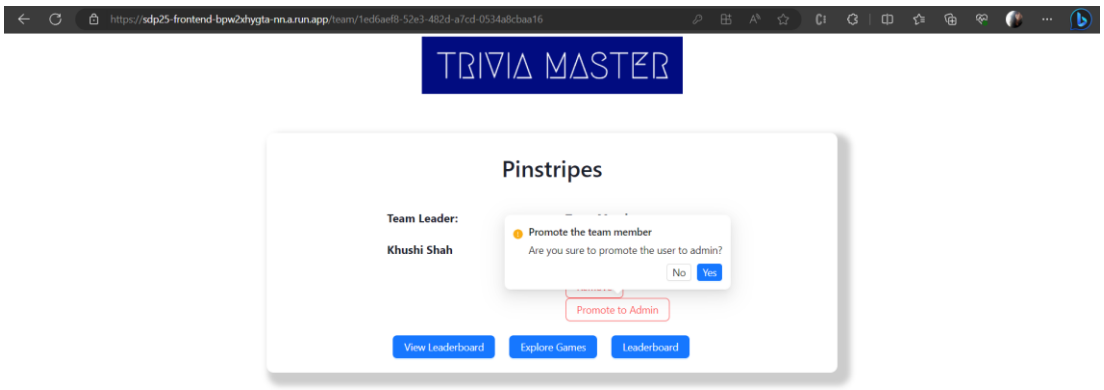


Figure 20:Functional Testing for Team management 2

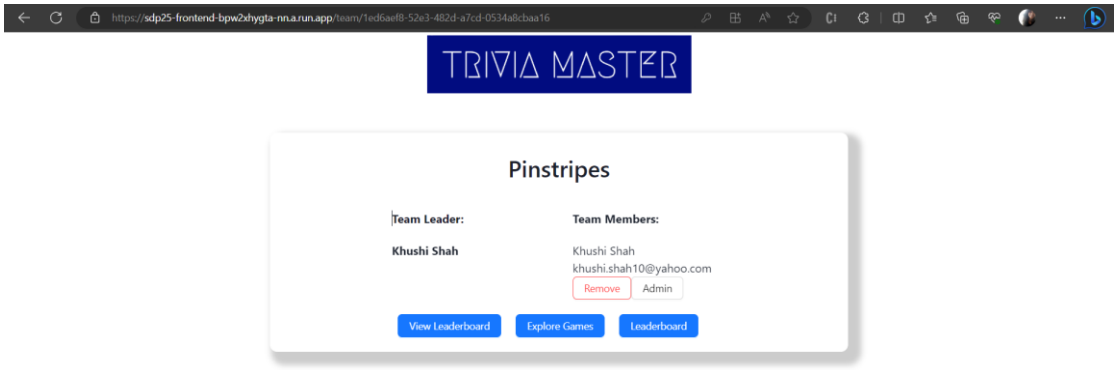


Figure 21: Functional Testing for Team management 3

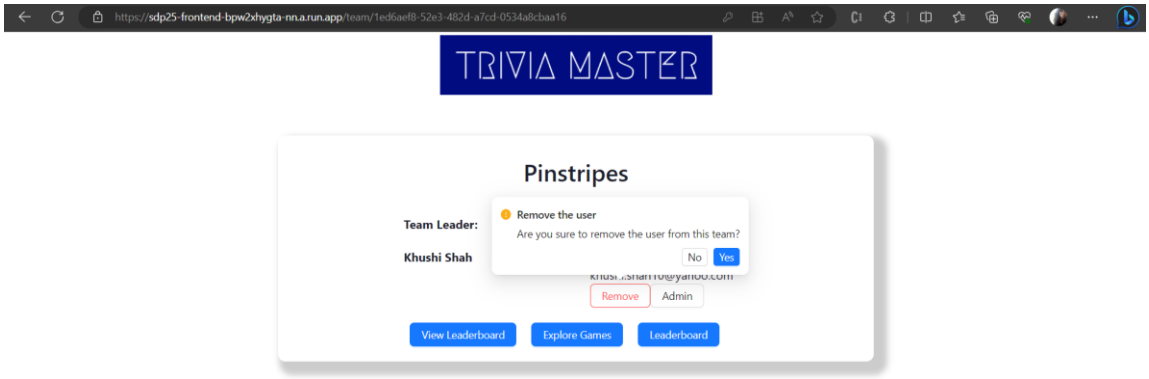


Figure 22:Functional Testing for Team management 4

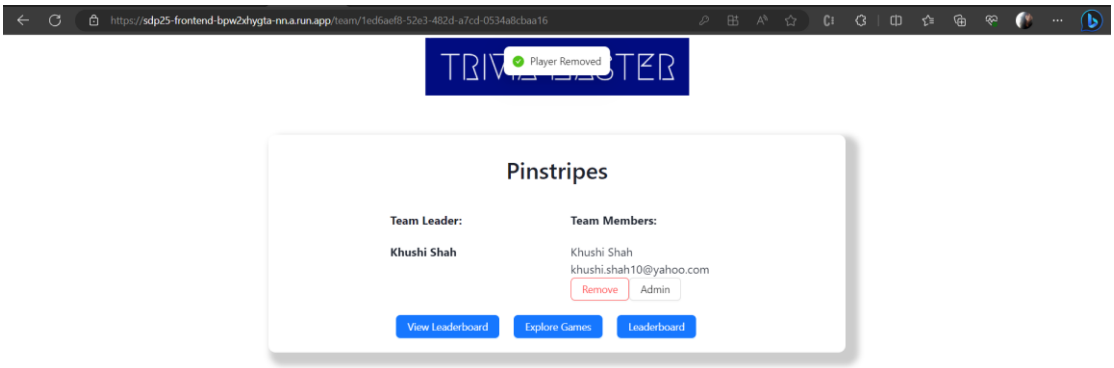


Figure 23: Functional Testing for Team management 5

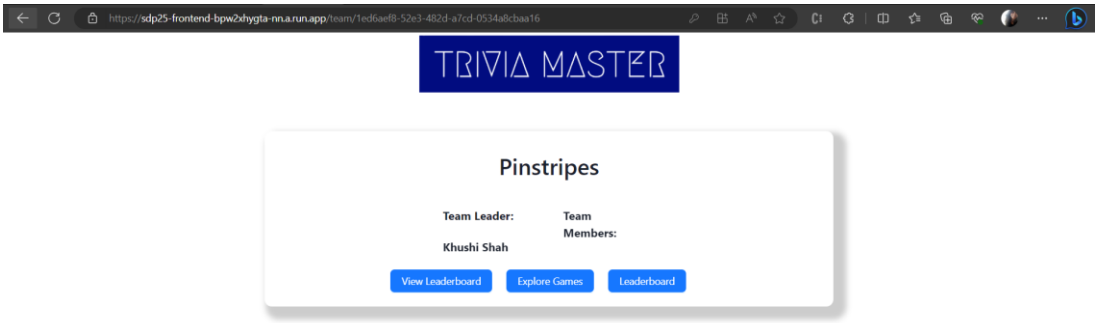


Figure 24: Functional Testing for Team management 6

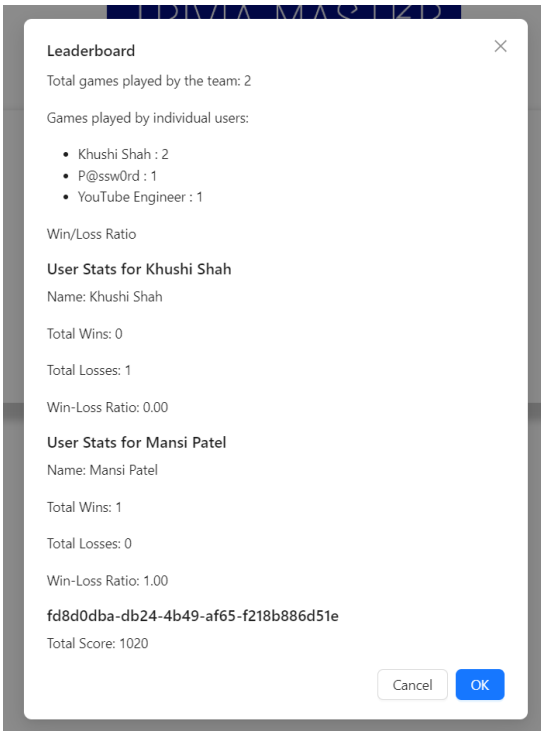


Figure 25: Functional Testing for Team management 7

API Testing (from postman):

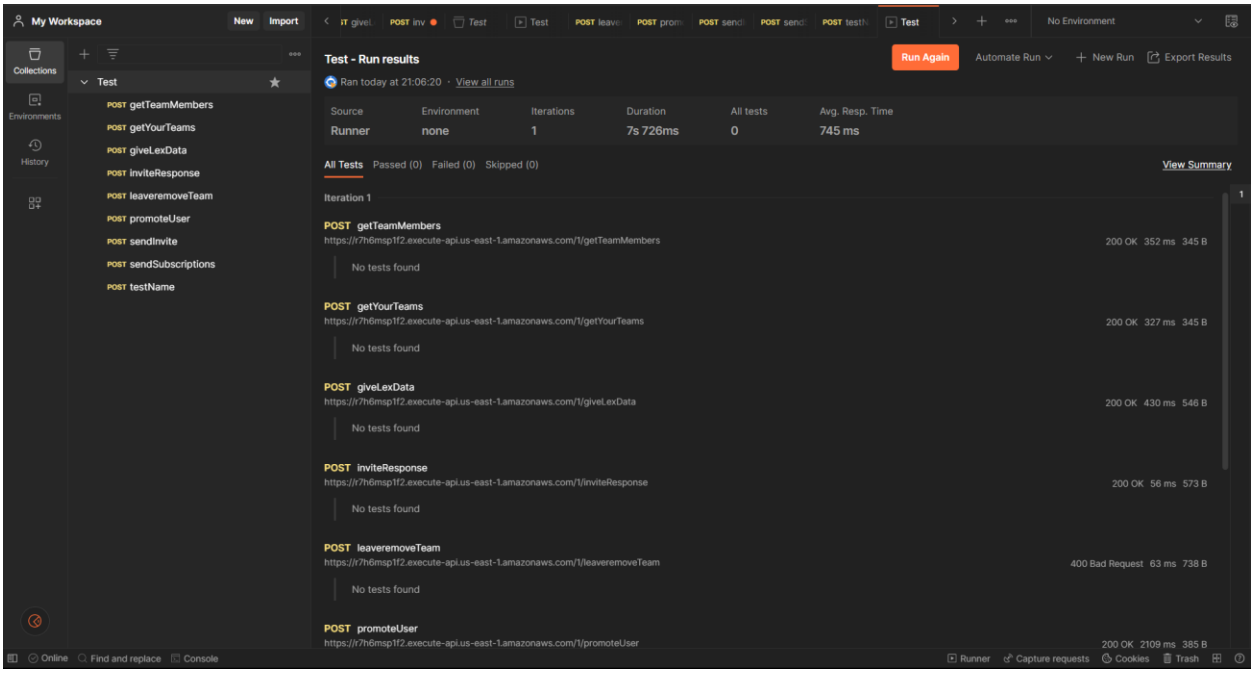


Figure 26: API testing 1

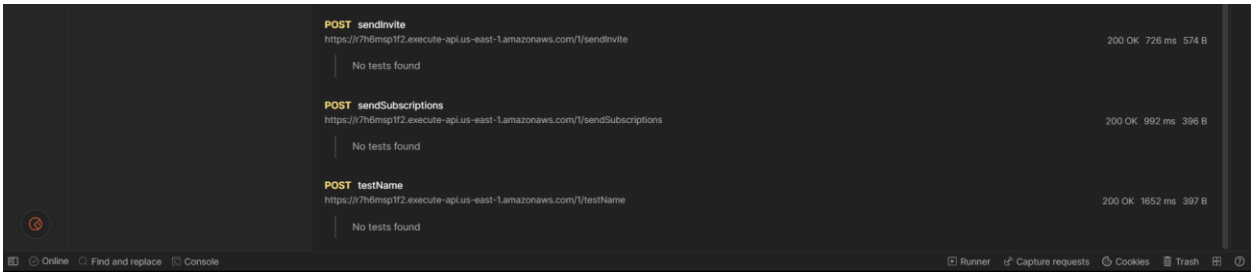


Figure 27: API testing 2

API used for this service:

Table 6: API endpoints

Method	Endpoint
post	/getTeamMembers
post	/getYourTeam
post	/inviteResponse
post	/leaveRemoveTeam
post	/promoteUser
post	/sendInvite
post	/sendSubscriptions
post	/testName

Table 7: Planned Vs Used Services

	Planned Service	Used Service
Database	DynamoDB	DynamoDB
Compute	Lambda, API Gateway	Lambda, API Gateway
Subscriptions	SNS	SNS

Functional Testing:

Table 8: Functional Testing Status

TestName	Expectations	Result	Status

Generate a new team name dynamically	The button click should generate new team names everytime	It creates new names	PASS
Display users to add on team	Should only display users that are confirmed	No other users are displayed except the one that are confirmed	PASS
Send emails	Send invitations to the added users	Email received	PASS
Accept/Reject Invite	User decides to accept/reject and accordingly DB must be updated and they must be added to the team	Users who accepted are added to the team, declined are not	PASS
View my teams and my participations	Only the logged in user's teams must be visible	Yes, only logged in users team are visible	PASS
Remove User	Admin and the team creator can remove the users	Users are removed from the team and DB	PASS
Promote to admin	Team creator can promote any user to the admin	Users are promoted to admin only by team creator	PASS
View Leaderboard	Display team stats for the opened team	Only statistics for the team are visible	PASS

4. Trivia Game Lobby

This feature contains Trivia Game Lobby with Category Filtering and Game Scheduling. The feature retrieves game data from a centralized datatable and presents it to players along with Name, Category, Difficulty level and time frame.

Fetching Game Data: The trivia game lobby will retrieve game information, including game title, category, time frame, and difficulty level, from the centralized datatable. This datatable will be handled by the admin portal to render information.
Category Filtering: Upon loading the trivia game lobby, players will have the option to filter the available games by category, difficulty level and time frame.
Email Notifications: Upon joining a game, players will receive an email notification that the game is scheduled to start in X minutes (X being a configurable value). The email will contain details such as the game title, scheduled start time,
Countdown Timer: Once a game's start time is scheduled, a countdown timer will be displayed in nextpage. The timer will indicate the time remaining until the game begins. Players can see the timer update in real-time.
Gather Questions: The questions which is stored with Category information in Question table will be called and gather data, With correct time stamp the start game will be triggered.

Choice of Services

During the initial stages of developing this feature, we carefully considered various options for building it. We performed a brief analysis to determine which cloud provider would be the most suitable. After evaluating Firestore, we realized that managing additional document-like structures for each game entry and question could be a challenge. Consequently, we opted to use DynamoDB as the database for this feature [8].
Once the database was selected, we found that combining DynamoDB with Cloud Functions might introduce portability and design issues. To avoid these potential complications, we made the decision to focus on implementing the game and question entry functionality within the AWS ecosystem instead of GCP. This choice allowed us to streamline development and ensure a smoother integration of the feature.

How I planned

I planned to do everything systematically, the steps include requirement analysis, database design, user interface design, technology stack selection [5], task breakdown, API design, game scheduling logic, email notification configuration.

How I did

After planning everything, I started working on following the steps. In between we had a meeting too, to validate the requirement of each module is being satisfied with another modules. I started with creating a lambda function to get the information from DynamoDB and process that data to make it useful for filtration. After completing his I created logical way to manage wait time and coordinated with team member working on in game experience module. Provided the needful data via lambda to start game [9].
To occupies this, started with designing the database structure and parameters

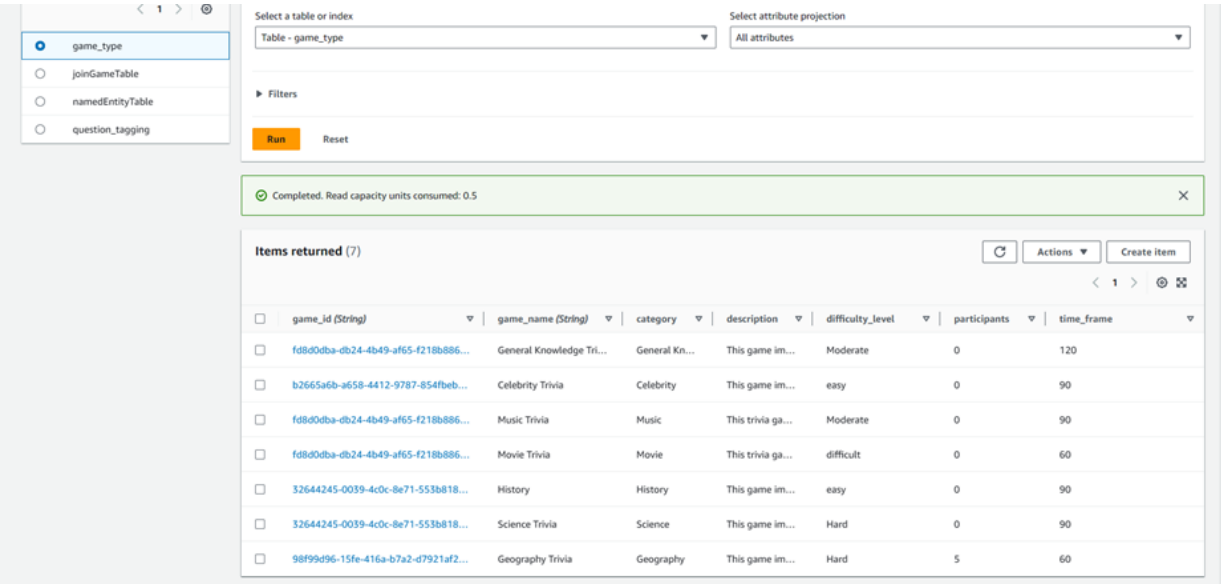


Figure 28: Game Type DynamoDB Table

This DynamoDB data are being used to render the information on the exploreGame page. These attribute values are responsible for filtering the game values [9].

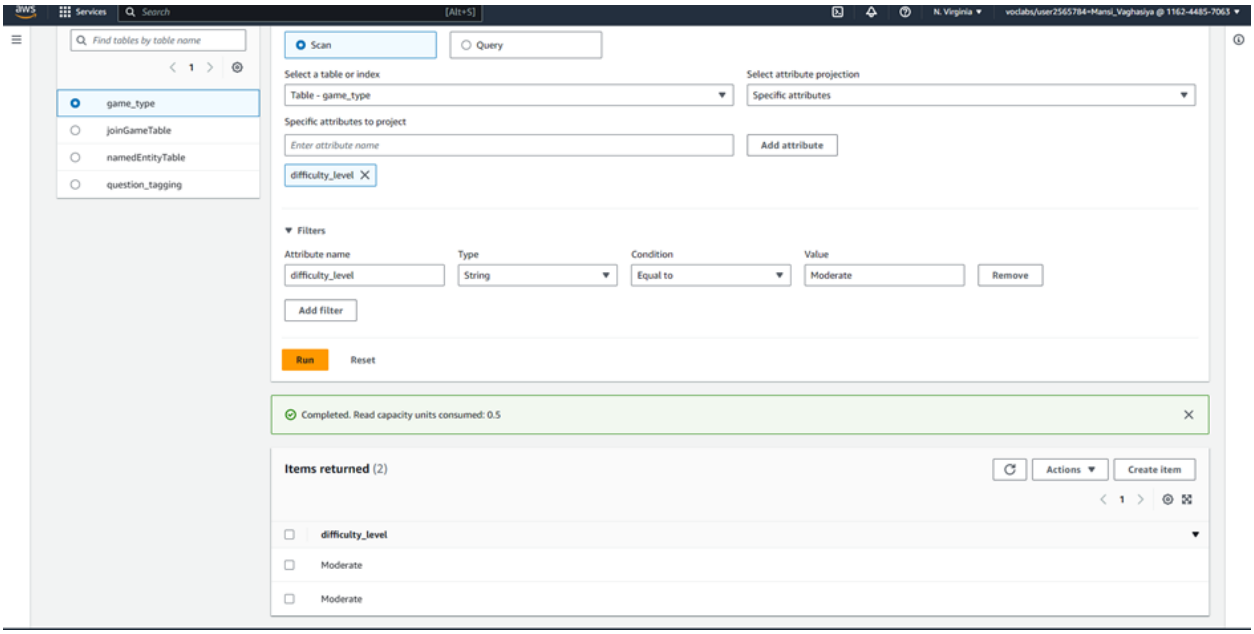


Figure 29: Filtered GameType DynamoDB table

I have created lambda functions to communicate between frontend and database [3].

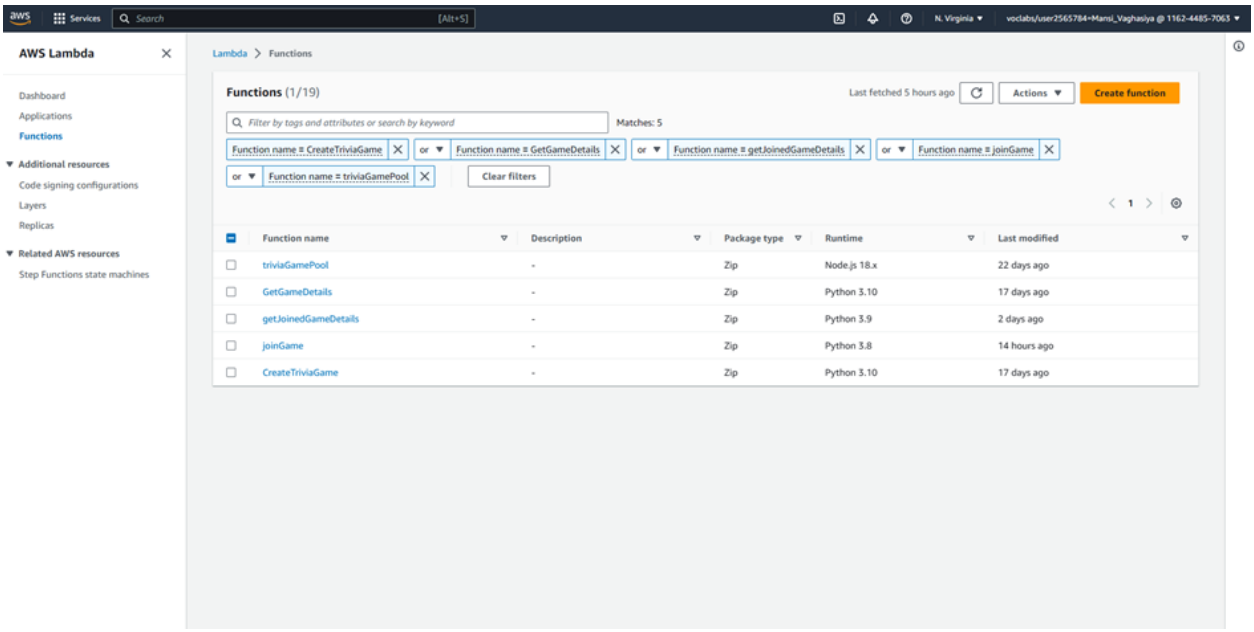


Figure 30: Lambda Functions

The module is designed in a way that It will access data from kinds of database mapping the game id, and render that according to database data [10].

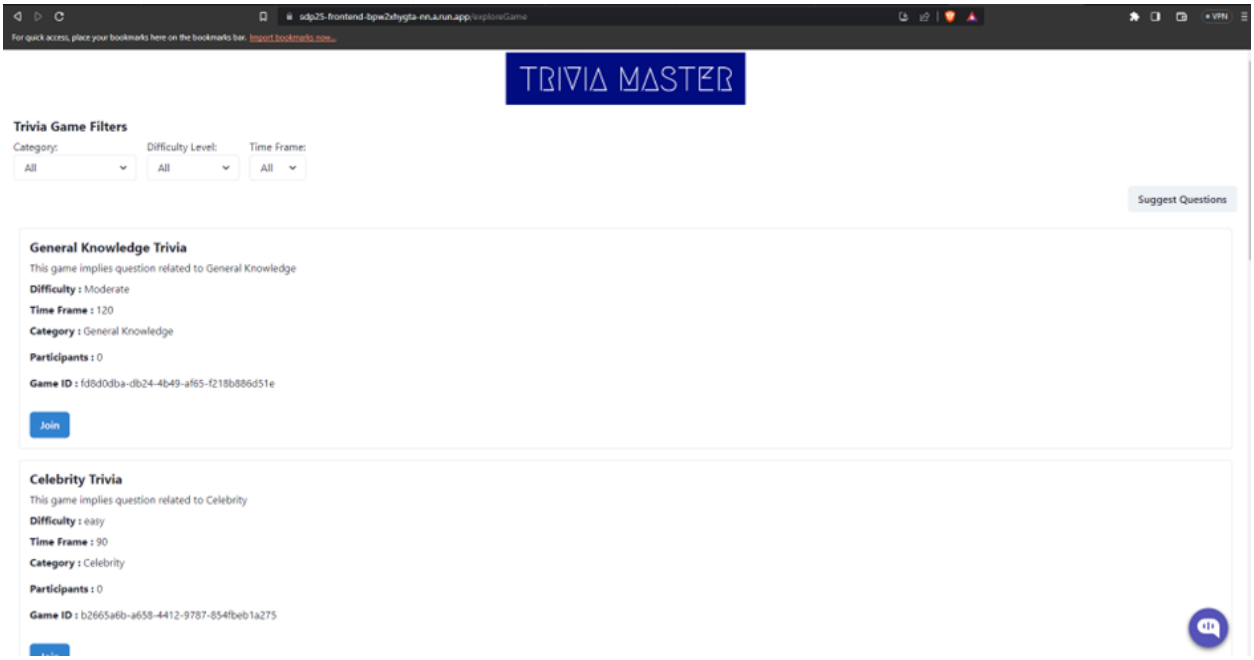


Figure 31: Games Dashboard

Here the details are shown like

Name

Difficulty Level

Time Frame

Category

Current Participants

Game Id

Here the games are being filtered by Category, Difficulty Level and Time Frame. This filter will be applied on AND policy, so if one Category is selected and one Difficulty level is selected than only entry will be shown which is satisfying both the conditions.

With 1 filter only

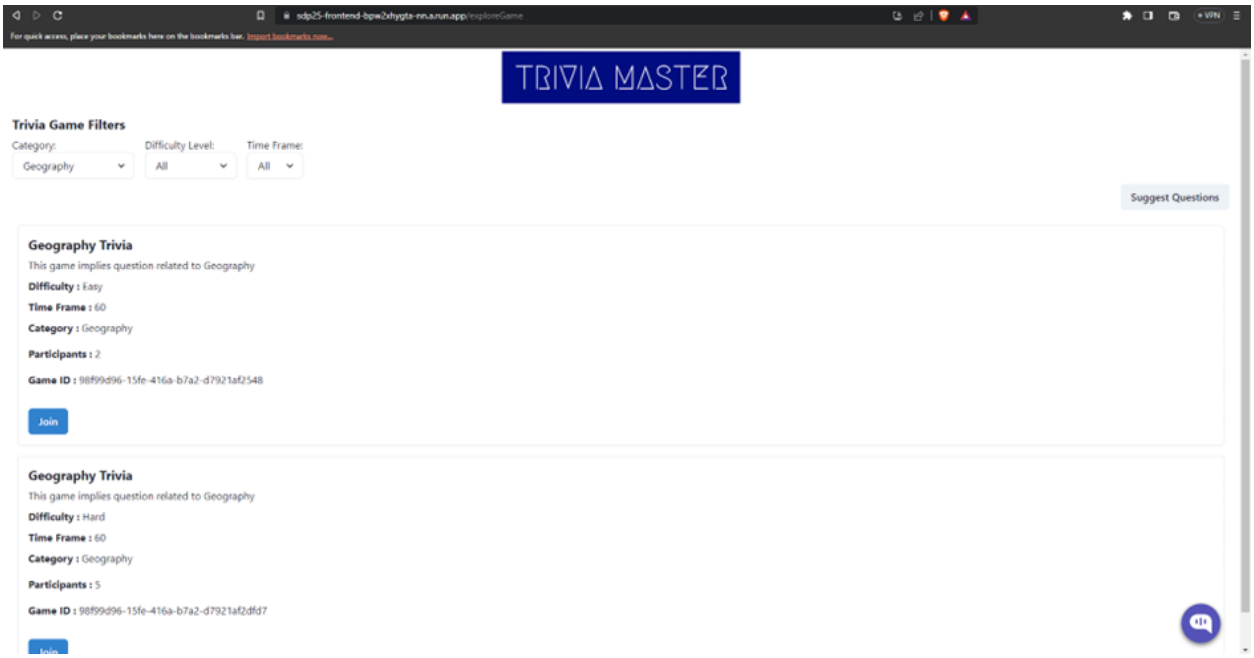


Figure 32: Filters - 1

With 2 filters

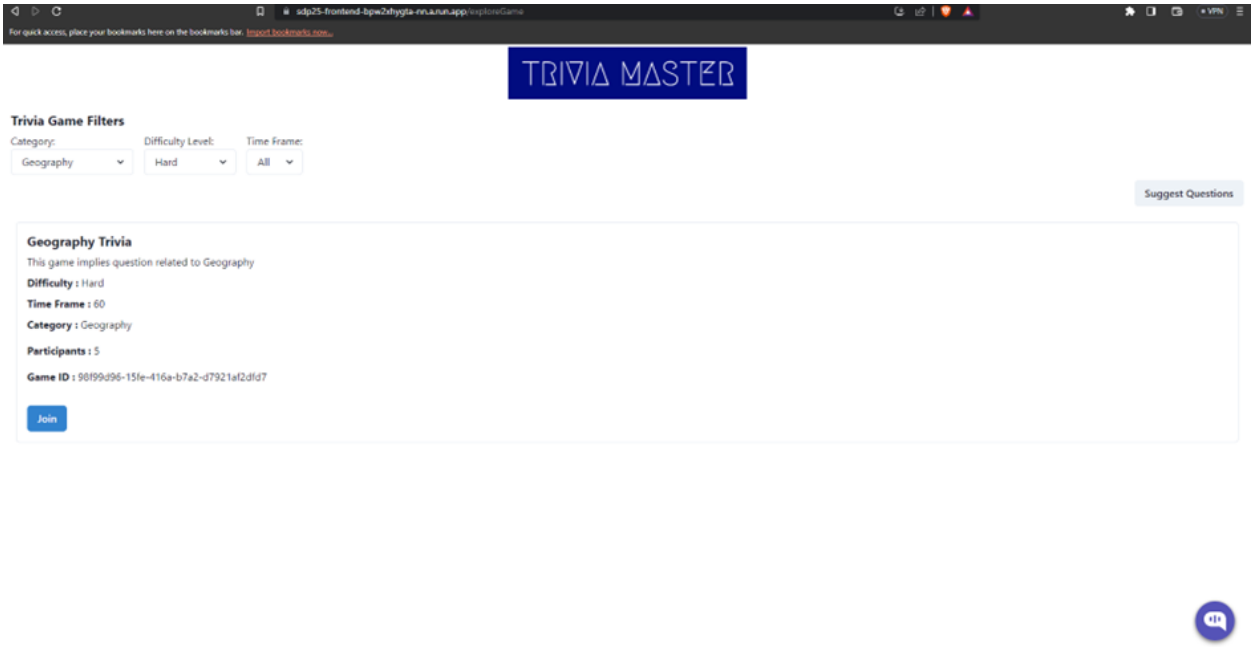


Figure 33: Filter - 2

Here the participants key is dynamic, which will be 0 at the time when game is just created. As the people will join the game, it will be updated by either number of participants waiting in the lobby or by number of participants playing the game.

Before anybody has joined the game

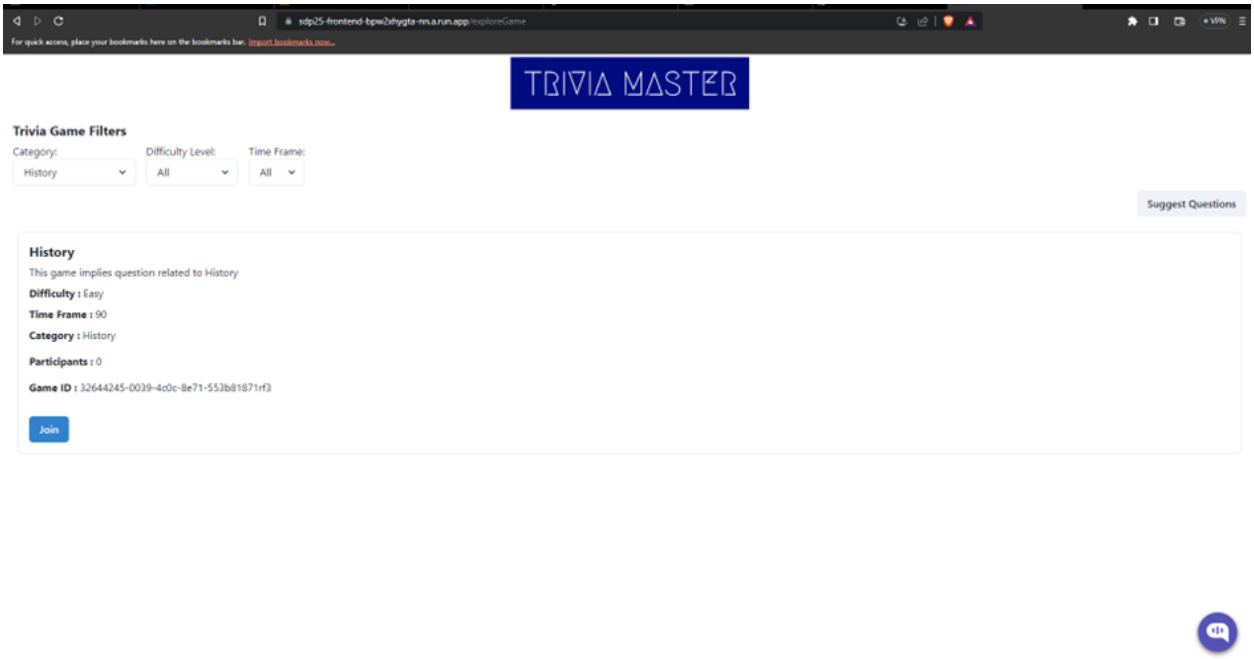


Figure 34: Dynamic Participants shown

After I join the game, the number of participants will be changed

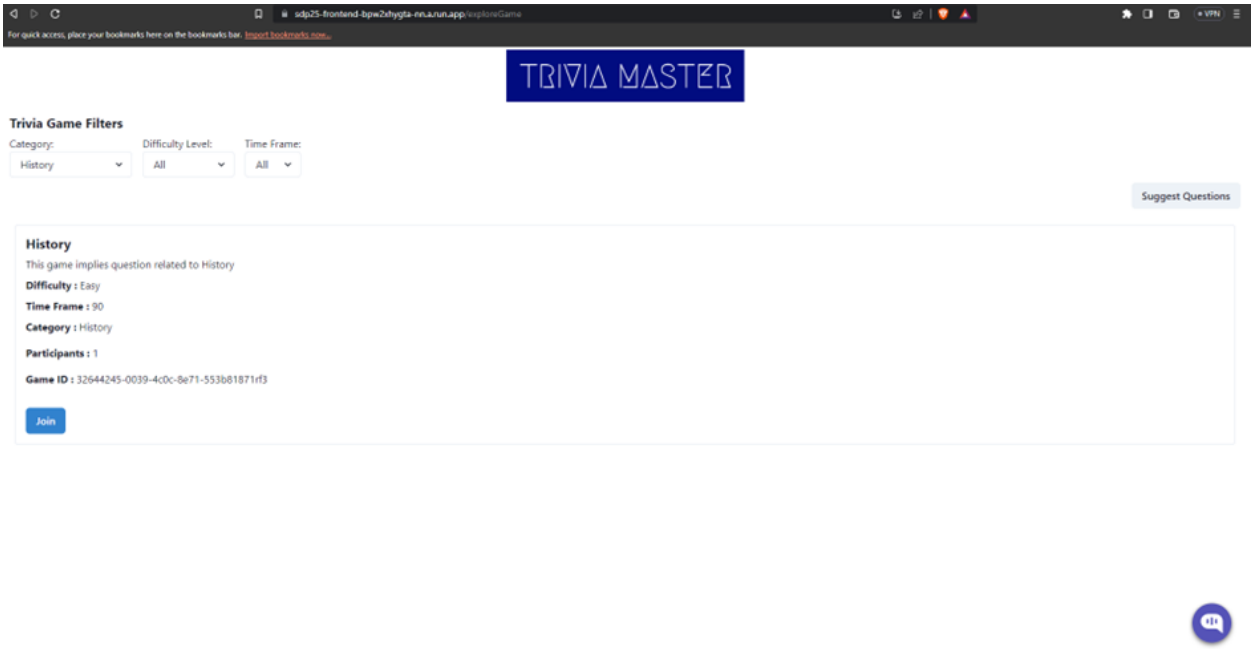


Figure 35: Waiting Lobby

After clicking on join button, the user will be redirected to waiting lobby and there will be reverse timer.

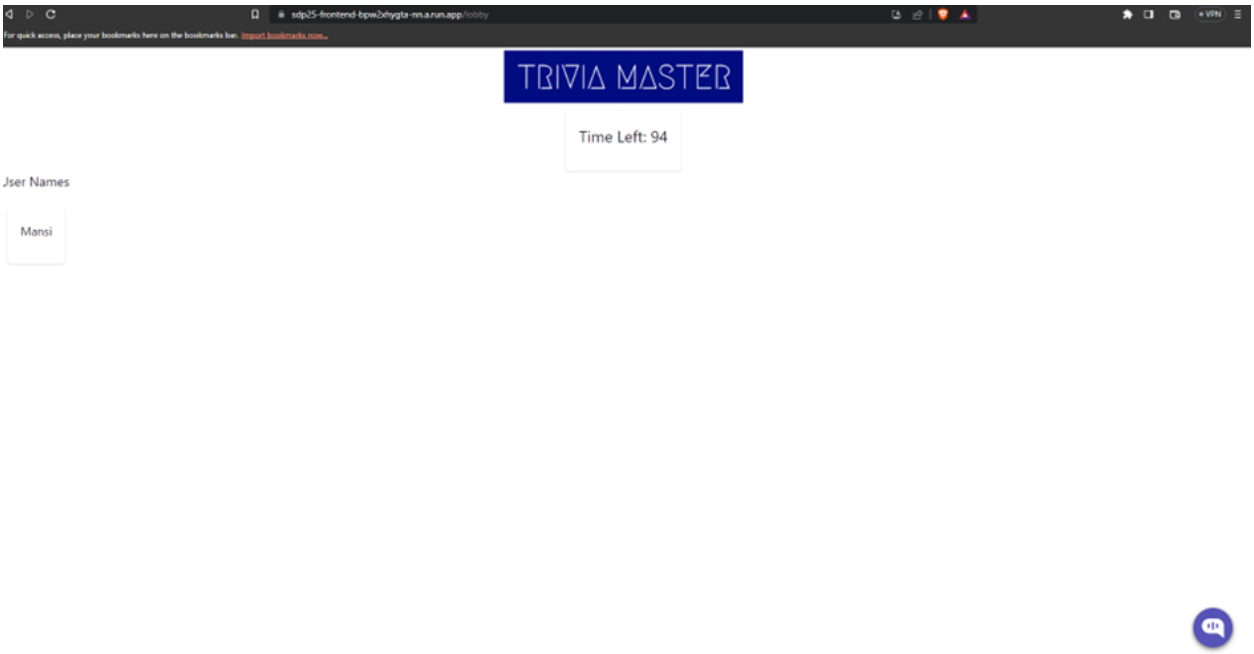


Figure 36: Waiting Lobby

When any other user will join the game, both the user and remaining time will be shown to user.

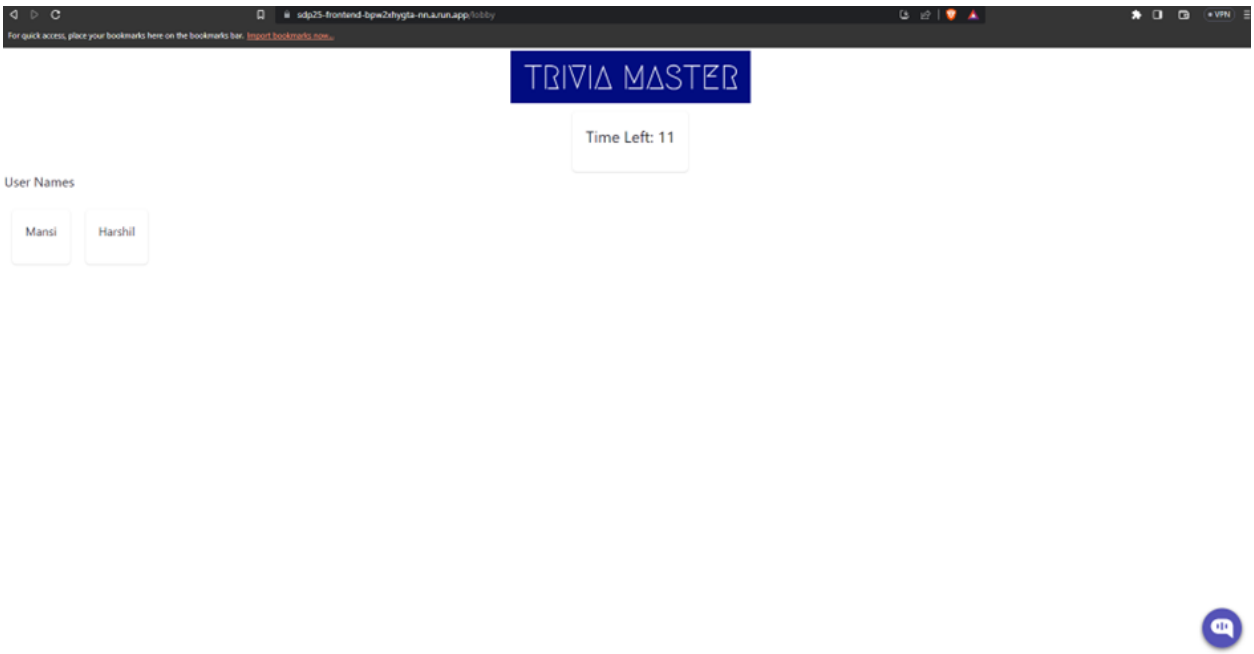


Figure 37: Time Left

By mean time I will gather data from API of Admin module, reformat that data and share it to In Game Experience module.

Planning

Table 9: Planned Service VS Used service

Planned	Used
AWS DynamoDb	AWS DynamoDB
AWS Lambda	AWS Lambda
AWS SNS	API Gateway

Tests performed

Table 10: Functional Testing

Test Name	**Expectation**	**Pass**
Filtering the data by Category.	It should filter the data properly	Pass
Testing two filters together	Work on AND condition	Pass
Testing the wait lobby functionality in a normal scenario.	It should show remaining time	Pass
Testing the wait lobby after refreshing the page.	It should start with continues time	Pass
Verifying the participant variable updates dynamically.	Another joined user should get same time	Pass

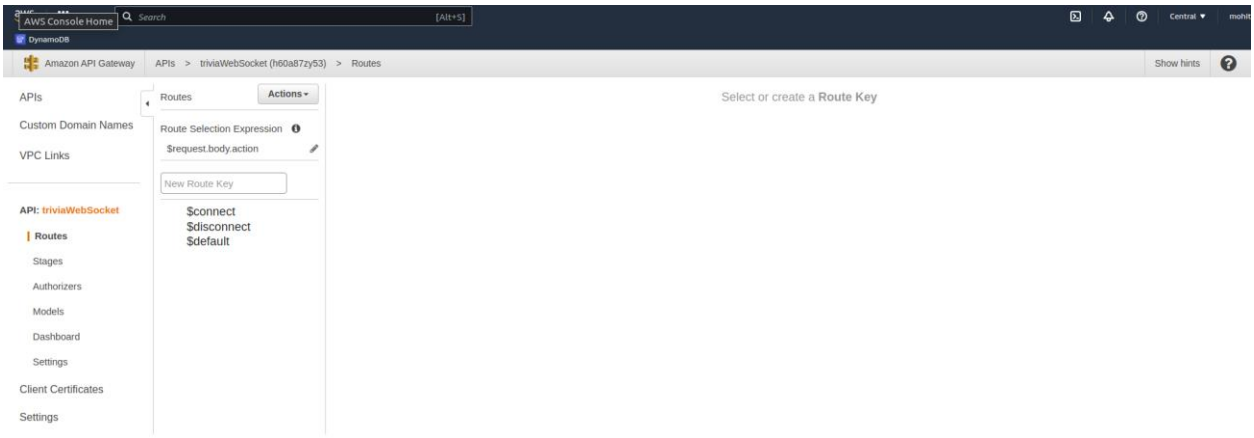
5. In-Game Experience

How I planned :

In the development of the game experience module, the requirement to answer multiple questions within specific time constraints, along with the need for real-time chat functionality, necessitated the adoption of a suitable two-way communication protocol. After careful consideration, the decision was made to employ the socket protocol. The evaluation process

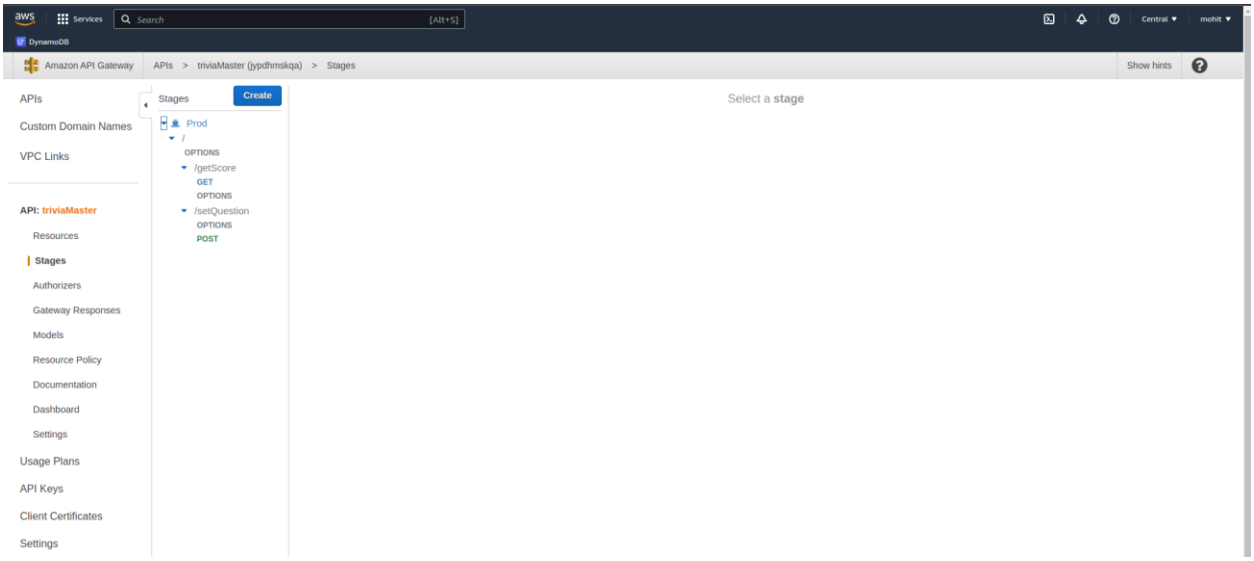
involved weighing two alternatives: Firebase Realtime Database and API Gateway with WebSocket. Ultimately, the selection was made in favor of API Gateway with WebSocket API, particularly due to the integration of AWS DynamoDB for storing the questions.
The chosen solution involved a combination of AWS services to achieve a cohesive and efficient system. AWS API Gateway with WebSocket functionality enabled real-time chat capabilities, allowing seamless communication between users. Moreover, the use of AWS DynamoDB ensured the effective storage and retrieval of questions. To address the handling of multiple questions with specific time periods, AWS SQS (Simple Queue Service) and Lambda functions were employed, facilitating a responsive and smooth user experience.
How I built it:
I used Serverless Applicaton Model to build the entire infrastructure from the scratch. I created two api gateway to one for the websocker api and other for the rest api.
APIGateway (Websocket API)

Table 11: API Gateway



APIGateway (RestAPI)

Table 12: Lambda Functions



Lambda functions to handle the requests:

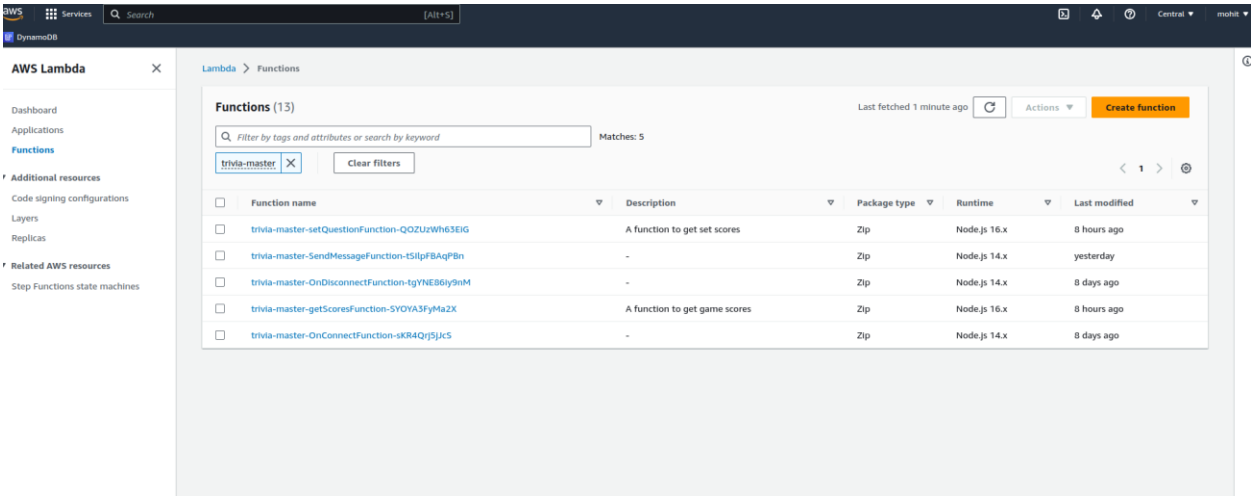


Figure 38: Lambda Functions

3.1 trivia-master-setQuestionFunction-QOZUzWh63EiG

The purpose of this Lambda function is to facilitate the creation and scheduling of games in a DynamoDB table while ensuring that each game is associated with a specific question. When invoked, the function takes relevant input data, such as the game details and the scheduled time, and then inserts this information into the DynamoDB table. Additionally, it associates a corresponding question with each game entry. By executing this function, games can be efficiently organized, and players will receive the appropriate question for the respective scheduled game at the designated time.

3.2 trivia-master-SendMessageFunction-tSIlpFBAqPBn

This Lambda function serves as the message handler for a Socket API, effectively managing various functionalities by breaking down the process into distinct components. It is responsible for handling incoming messages sent to the Socket API and is structured into separate segments to handle the functionality of sending messages, saving user answers, providing hints to users, and distributing questions among connected users within the game. This modular approach allows for efficient and organized management of real-time interactions and ensures smooth communication between users and the game system through the Socket API.

3.2 trivia-master-OnDisconnectFunction-tgYNE86iy9nM

This is a straightforward Lambda function designed to handle disconnections from the Socket API. Its primary objective is to remove the userdata associated with a connected user from the connected user table. When a disconnection event occurs, the function is triggered, and it efficiently removes the relevant user data from the table, ensuring that the user's presence is no longer tracked within the Socket API system.

3.3 trivia-master-OnConnectFunction-sKR4Qrj5jJcS

Upon a user's connection to the socket, this Lambda function executes to efficiently store the user's information, including user_id, connection_id, and team_id, into a dedicated database table. The function is triggered whenever a new user establishes a connection, and it promptly captures and records the relevant user details into the table. By doing so, it ensures that crucial user information is accurately preserved and readily accessible for seamless real-time interactions within the Socket API system.

SQS Queue:

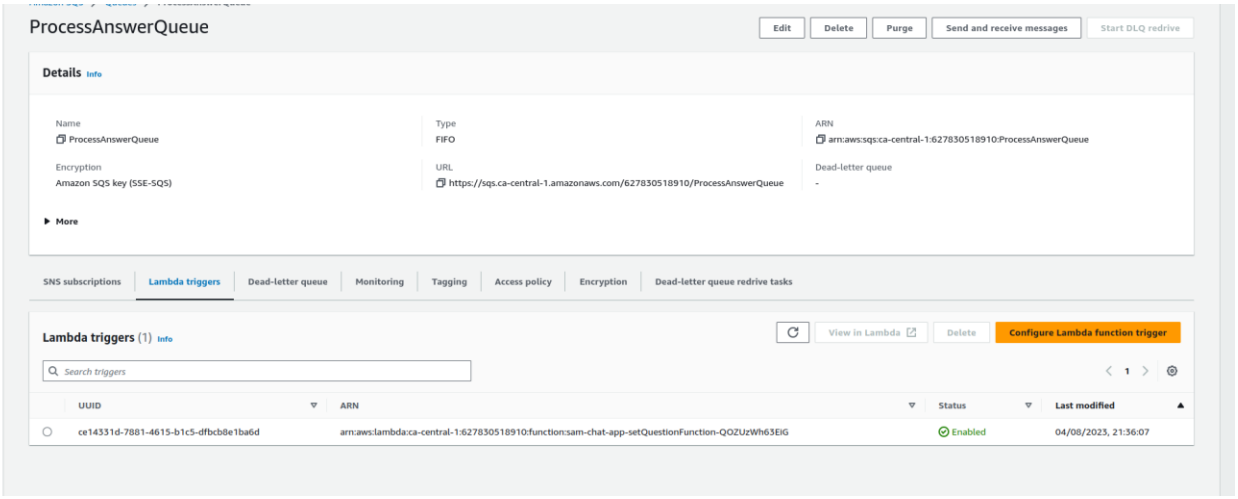


Figure 39: SQS Queue

This queue system serves as a mechanism to efficiently manage and prioritize answers submitted by users within a game or application. It adopts a First-In-First-Out (FIFO) approach, ensuring that the user who submits an answer first receives higher points for the correct response. As users submit their answers, the queue stores the question-answer ranks, preserving the order in which answers are received. This orderly arrangement allows for

accurate determination of the first responder, who is rewarded with higher points in accordance with their timely submission. By employing this queue system, the application can effectively handle and process user answers in a fair and equitable manner, promoting engagement and competition among users while maintaining a streamlined and organized flow of responses.

DynamoDB table:

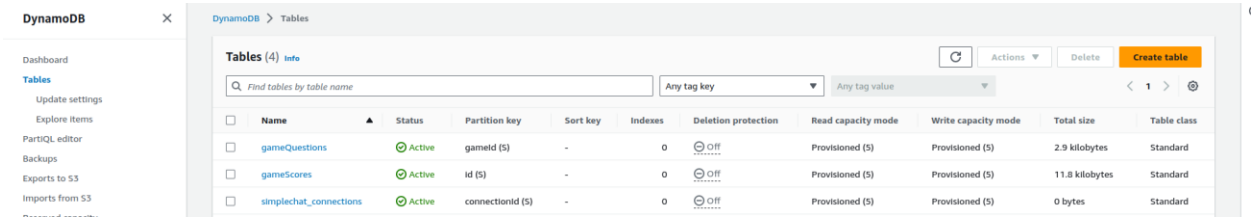


Figure 40: DynamoDB

The "gameQuestions," "gameScores," and "simplechat_application" tables serve as essential components of the application's data storage and management system. The "gameQuestions" table is designed to store the questions used in the game, enabling easy access and retrieval of questions during gameplay. Meanwhile, the "gameScores" table is responsible for storing user scores and related information, allowing for the tracking and updating of individual and team scores as the game progresses. Lastly, the "simplechat_application" table is dedicated to storing socket connection data, recording user details such as user_id, connection_id, and team_id upon their connection to the socket.

Testing:

Websocket: To monitor WebSocket connections and trace received messages, I utilized a Chrome extension called "WebSocket Testing." This extension facilitated the observation of message exchanges over the WebSocket protocol.

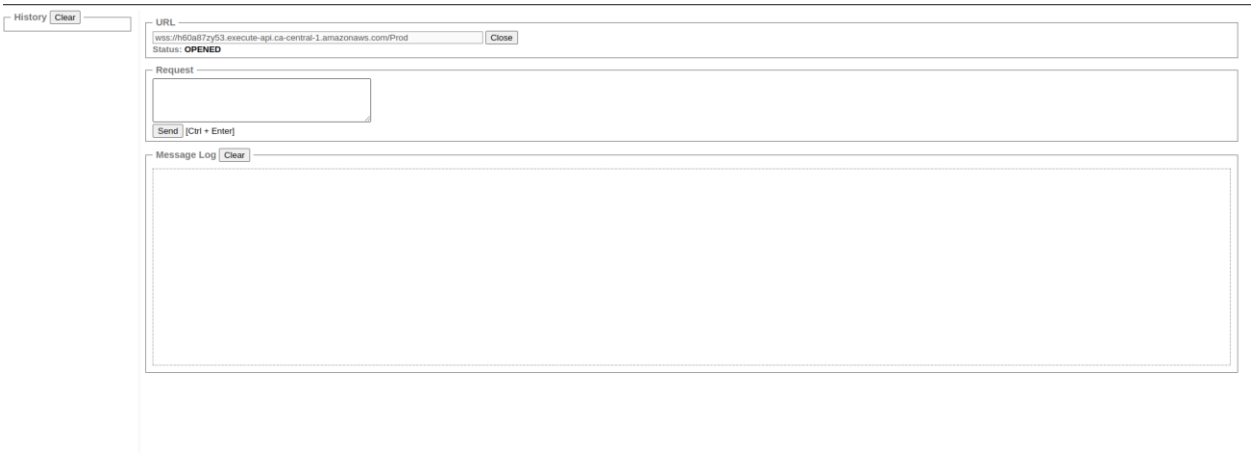


Figure 41: WebSocket Monitoring

Testcases:

Table 13: Test Case

Input	**Expected Behavior**
saveIdeneity with User data JSON	Save user details on table
sendMessage with message	Publish the message to all team member
getQuestion with current timeframe and gameid	Fetch and send the question to that user

RestAPIS: To inspect and interact with RESTful APIs, I employed the widely-used tool Postman. With Postman, I was able to send HTTP requests (such as GET, POST, PUT, DELETE, etc.) to the API endpoints, examine the responses, and trace the data flow during the testing process.

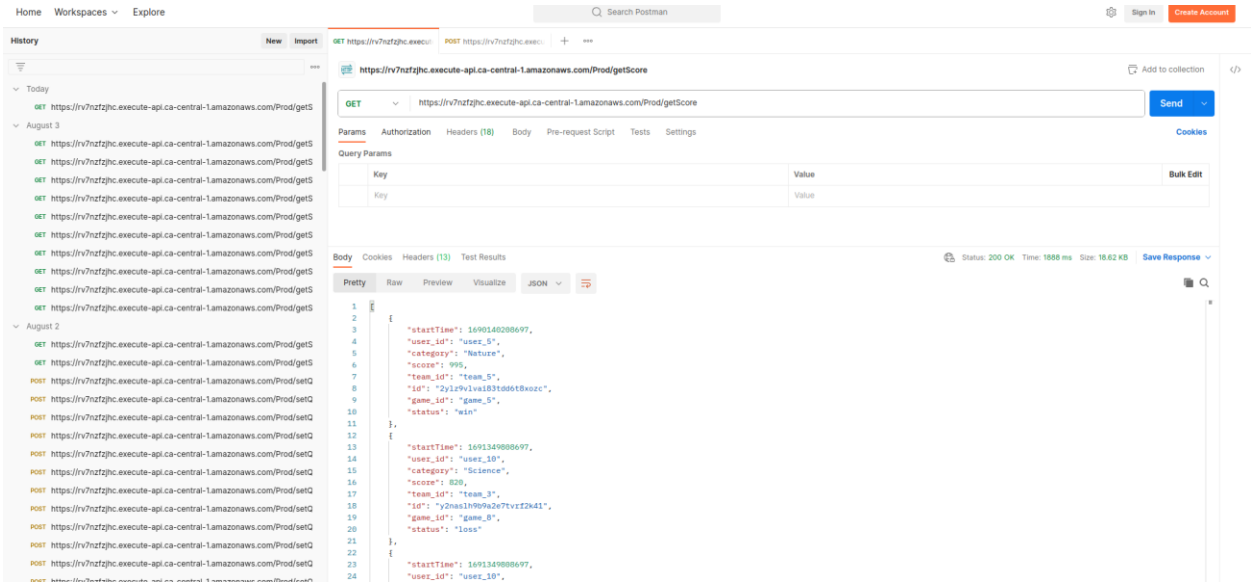


Figure 42: Postman Testing

Testcases:

Table 14: Test case

Endpoint	**Input**	**Expected Output**
/getScore (GET)	user_id or team_id as QueryParams	Return the score array of that user or team
/setScore (POST)	JSON object or submitted answer, time	Return the Score

/setQuestions	JSON array of question for scheduled game	Return the success resposne

All the test cases were run as per requirement

Cloud service planned vs Cloud Service Used:

Table 15: Planned Vs Used Services

Planned	**Used**
AWS DynamoDB	AWS DynamoDB
AWS Lambda	AWS Lambda
API Gateway	API Gateway
AWS SNS	AWS SQS

6. Leaderboards

How I planned :

In the context of implementing the leaderboard module, the requirement entails access to user details, team details, and individual user scores achieved per team. Additionally, the integration of this data into a Business Intelligence (BI) technology is essential. In this regard, two options were considered: Looker Studio and AWS QuickSight.

After careful evaluation, the decision was made to utilize AWS QuickSight for generating reports. This choice aligns well with the existing data infrastructure, as user details and scores are stored in an AWS DynamoDB table. To facilitate the data integration process, an AWS free tier account will be utilized, along with AWS Lambda for Extract, Transform, Load (ETL) operations from DynamoDB to AWS QuickSight. By leveraging AWS QuickSight's robust features, the goal is to create dynamic leadership graphs and insightful visualizations based on the accumulated data.

This approach ensures seamless integration of user and team data into AWS QuickSight, enabling the generation of comprehensive and meaningful reports. By effectively utilizing AWS Lambda for data ETL operations and leveraging the capabilities of AWS QuickSight, the leaderboard module will deliver valuable insights and actionable analytics to enhance the gaming experience.

How I built it:

aws-athena-query-results-ca-central-1-627830518910

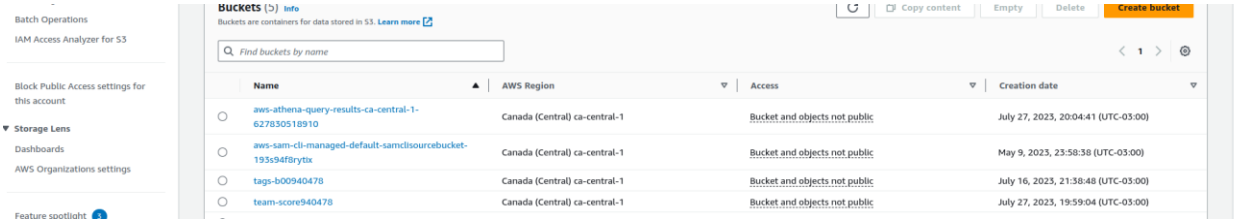


Figure 43: Buckets

You need the aws-Athena-query-results S3 bucket for Amazon Athena because it serves as the default location where Athena stores the query results and query execution metadata. When you run a query in Athena, the results are typically large and can be stored in multiple files. Athena data source.

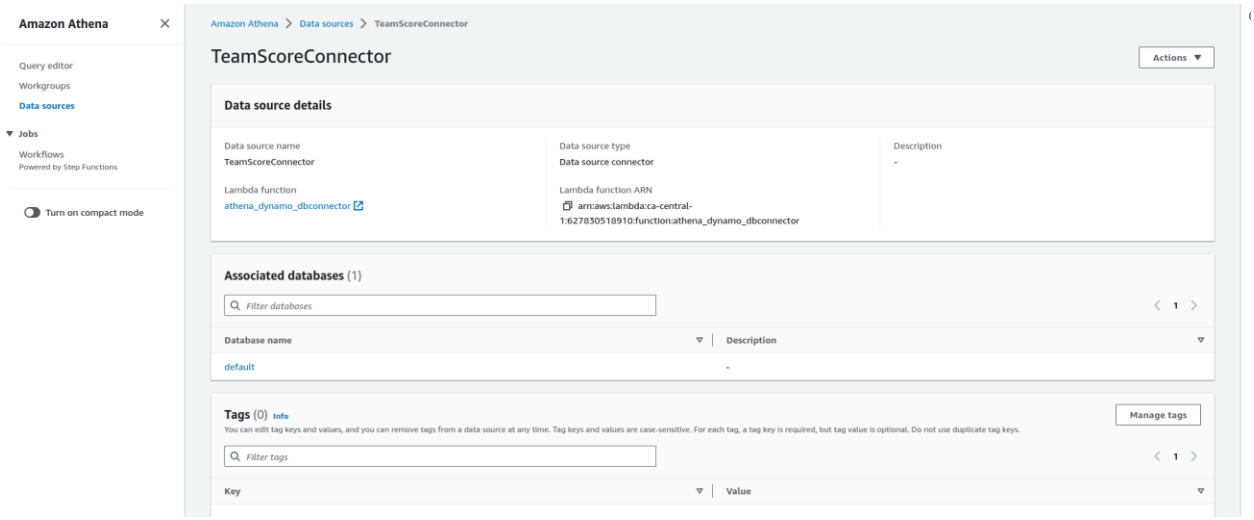


Figure 44: AWS Athena

Amazon Athena is an intermediary data source for Amazon DynamoDB in Amazon QuickSight, offering advantages for data analytics and reporting. By transforming NoSQL data into a tabular format, Athena's SQL querying capabilities enable complex queries, joins, and aggregations. Its serverless architecture eliminates infrastructure management and costs for queries run. This integration allows users to create interactive dashboards, gain deeper insights, and make data-driven decisions using QuickSight's robust visualization and reporting features.

Connect Amazon QuickSight and Amazon Athena with Federated Query

Amazon QuickSight supports Federated Query, enabling cross-data source analysis between Amazon QuickSight and Amazon Athena. To connect, set up an AWS Glue Data Catalog, create IAM Policies, configure data sources, connect to AWS Glue Data Catalog, define query options, import data, and create visualizations and dashboards. This enables sophisticated insights and decision-making by combining data from multiple sources. For up-to-date information, consult the official AWS documentation.

Create QuickSight dataset of DynamoDB via Athena

he Amazon QuickSight console, go to the "Manage Data" screen, and click on "New data set." Select "Athena" as the data source and connect it to your AWS Glue Data Catalog, which maps to your DynamoDB data. Additionally, you can connect other data sources, such as Amazon S3, Amazon Redshift, Amazon RDS, etc., if you want to combine data from multiple sources.

Graph's creation in Quicksight analysis

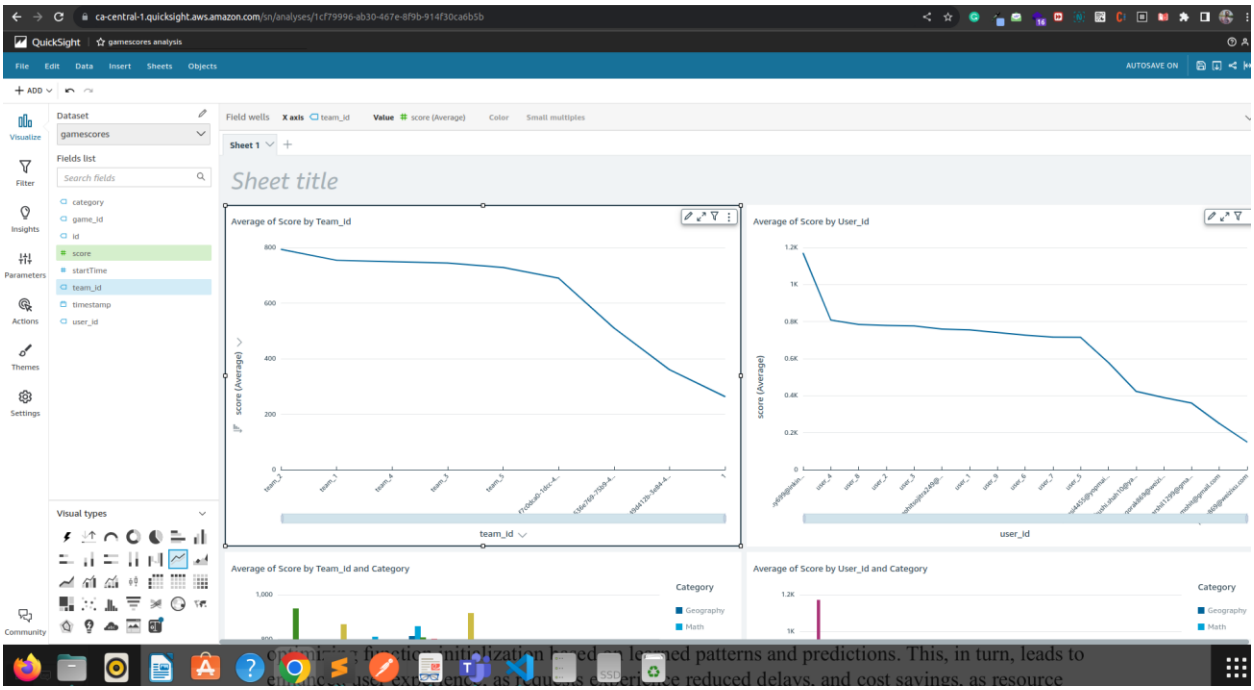


Figure 45: Quick Sight (Login Required)

By creating custom filters in Amazon QuickSight, We can efficiently filter leaderboards to display data based on different time frames such as weekly, monthly, and daily intervals. These filters enable dynamic data selection, as you can pass specific parameters through the dashboard URL to control the time range for the leaderboard. By configuring the custom filters, users can interactively choose our desired time frames, adjusting the displayed data accordingly without the need for manual query modifications.

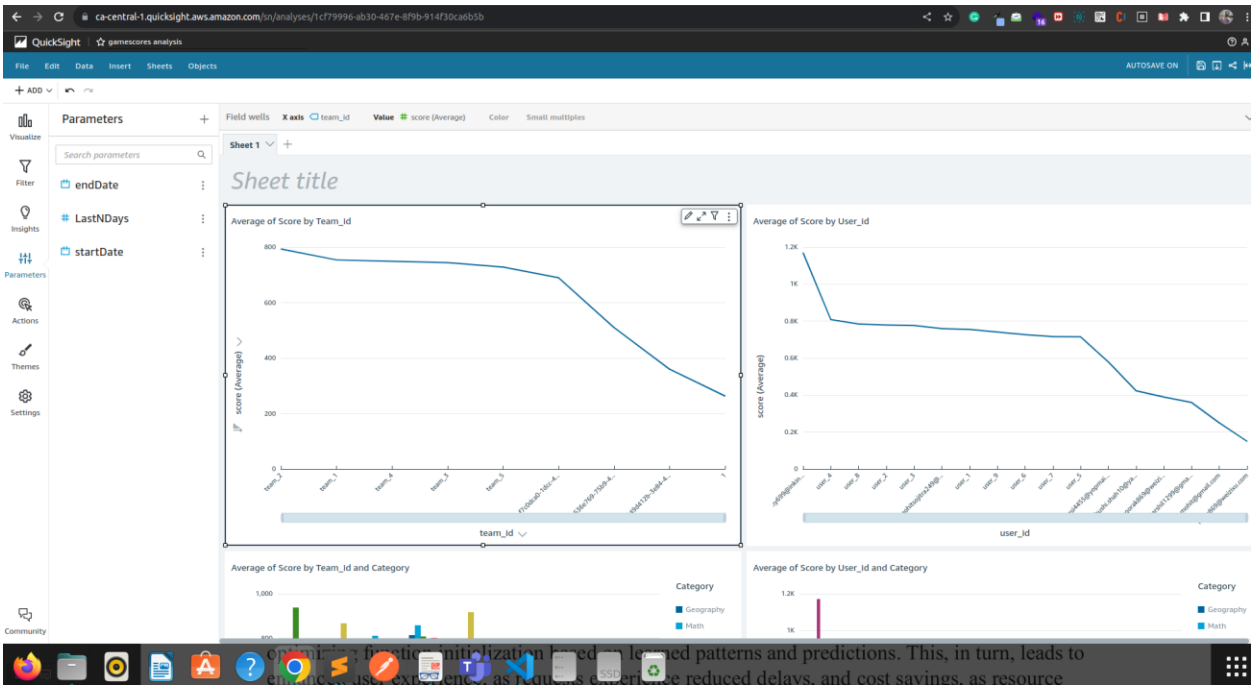


Figure 46: QuickSight 2

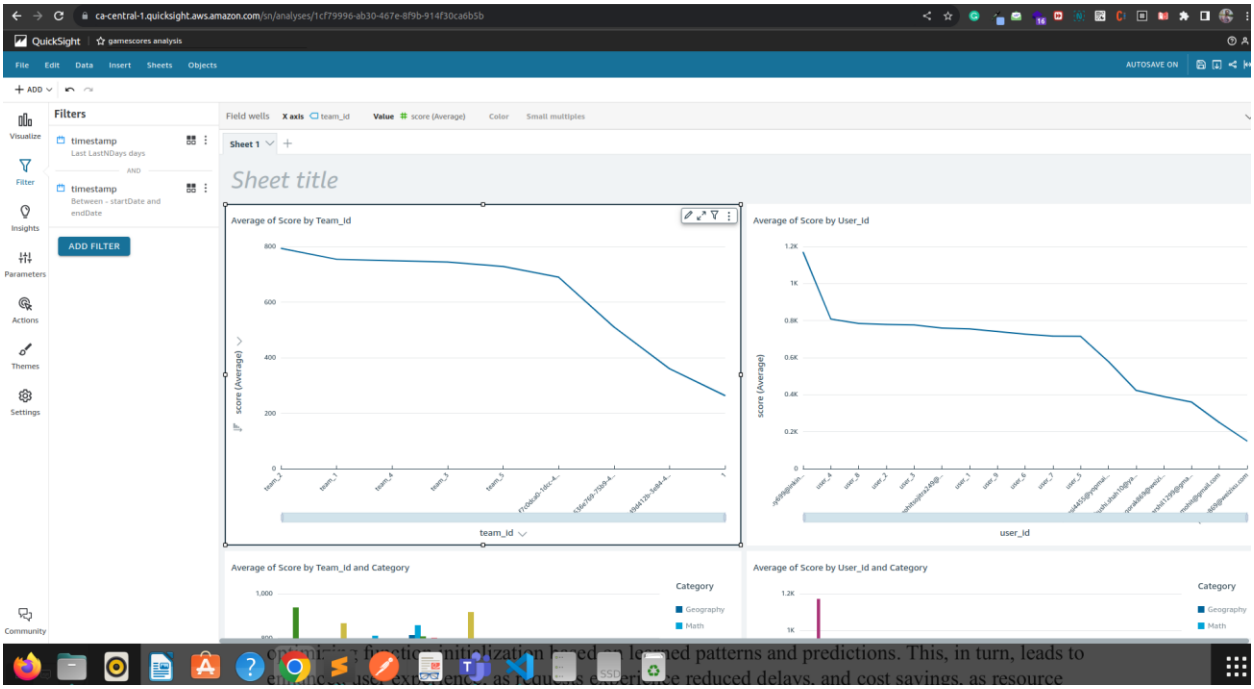


Figure 47: QuickSight 3

Additionally, Amazon QuickSight allows users to export the created dashboard and integrate it into their applications as an iframe source. This capability enables seamless embedding of QuickSight dashboards within external applications, offering a cohesive user experience. By exporting the dashboard as an iframe, developers can present data-driven visualizations

directly within their applications, enriching the user interface and providing valuable insights to end-users without requiring them to navigate to the QuickSight interface separately.

Testing:

To test the graphs and its authenticity I created a script which generate random score of 5 teams and 10 users for 10 games. So my script has generated 100 fake records which I store over dynanmoDB table and then mapped that table to amazon quicksight.

```
for (let i = 0; i < 100; i++) {
  const user_id = Math.floor(Math.random() * 10) + 1;
  const game_id = Math.floor(Math.random() * 10) + 1;
  const team_id = Math.floor(Math.random() * 5) + 1;
  const score = Math.floor(Math.random() * 501) + 500; // Generates a random number between 500 and 1000 (inclusive)
  const startTime = gameIdStartTimeMapping[user_id-1];
  const category = gameCategoryMapping[user_id-1];
  const params = {
    TableName: "gameScores",
    Item: {
      id: {
        S:
          Math.random().toString(36).substring(2, 15) +
          Math.random().toString(36).substring(2, 15),
      },
      user_id: {
        S: `user_${user_id}`,
      },
      game_id: {      You, 2 days ago • be code
        S: `game_${game_id}`,
      },
      team_id: {
        S: `team_${team_id}`,
      },
      score: {
        N: score.toString(),
      },
      startTime: {
        N: startTime.toString(),
      },
      category: {
        S: category,
      }
    },
  };

  randomData.push(params);
```

Figure 48: Graph Testing Script

This test worked as expected and now I was able to see the graphs, However I don't have that dashboard now So I cannot share a snapshot over here.

Cloud service planned vs Cloud Service Used:

Table 16: Planned Feature VS Used Features

Planned	Used
Looker Studio	AWS Quicksight
AWS Firebase	AWS DynamoDB
-	AWS Athena
-	AWS S3

7. Trivia Content Management

Initial Plan

The initial plan for implementing the trivia contact management feature was to use a combination of AWS DynamoDB, Lambda Functions, SQS and SNS plus AWS QuickSight. The initial thinking was that keeping as many of our services on the same platform and within similar services common among features would simplify the development process. While the this remained true for the core services, as development proceeded, the structure had to be adapted to circumvent the limitations of the LabRole account provided to us for access to AWS services. AWS QuickSight in particular caused a lot of problems for me because a provider-managed role as shown below:

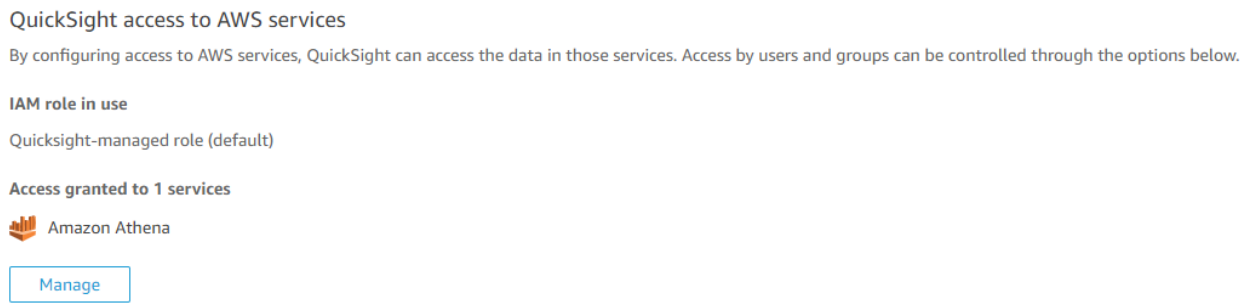


Figure 49: QuickSight provider-managed role

QuickSight does not offer any out-of-the-box integration with DynamoDB, which I used consistently to integrate with all the other services in my architecture, with a preference being for relation databases as shown below:

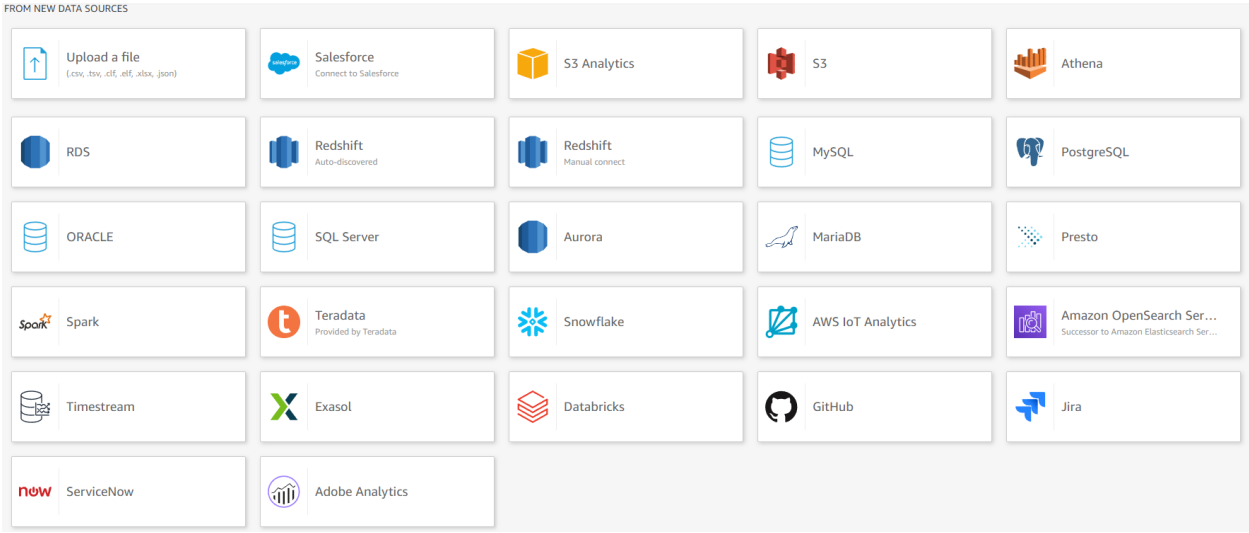


Figure 50: Services which QuickSight natively supports as data sources for analysis

However, although not as straight forward, it is possible to feed data from a DynamoDB Table into QuickSight for Analysis using an alternative method involving the ETL processes provided by AWS GLUE, Storage in S3 and Data preparation in AWS Athena as highlighted in this amazon article. I followed this method successfully and create a crawler for my DB and performed simple ETL with a spark script to extract my data and place it in Athena. Unfortunately, at the final step, Athena was not able to access schema I crawled with Glue had to be stored in s3, which by default cannot be accessed by QuickSight. Enabling this access is just a matter of toggling a selection option and hitting ok, but the lab role blocks the ability to modify permissions.

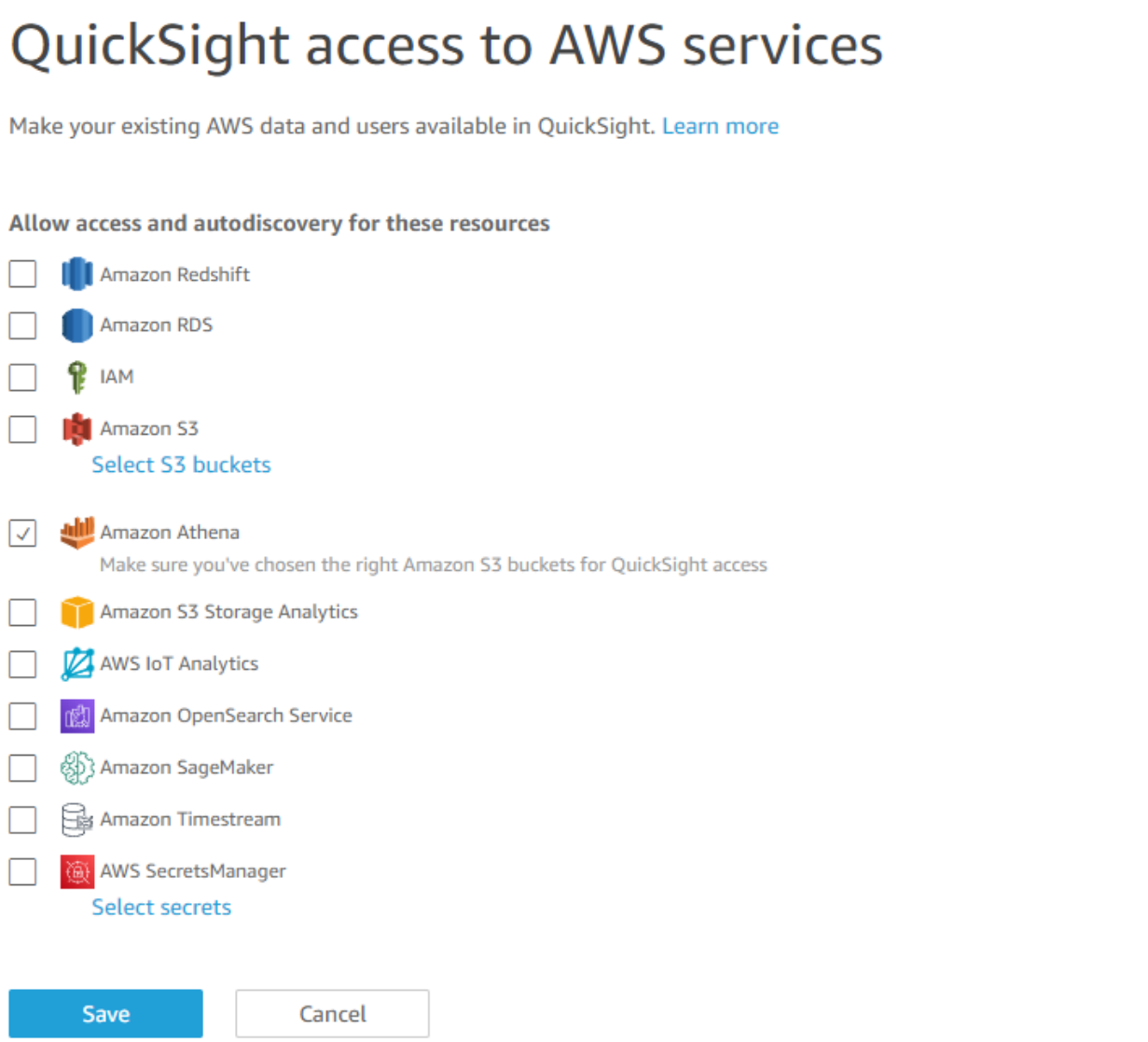


Figure 51: Default QuickSight Access and Management console

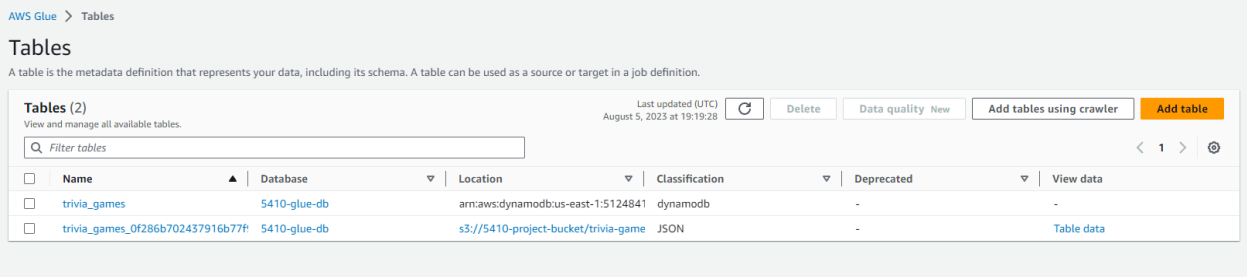


Figure 52: AWS Glue Data Catalogs

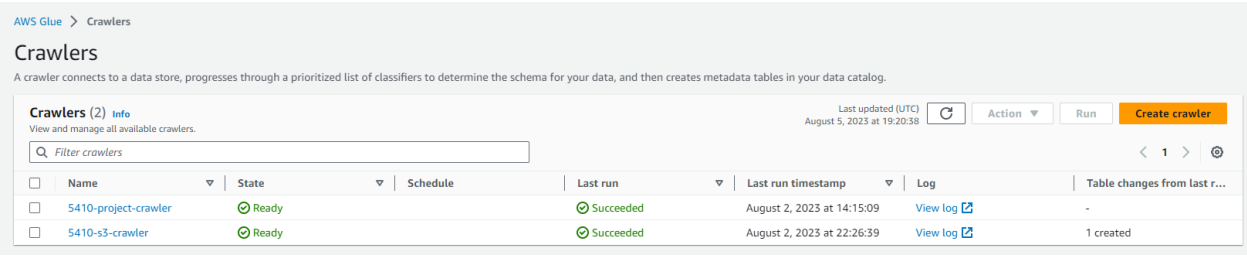


Figure 53: AWS Glue Crawlers (Dump to S3)

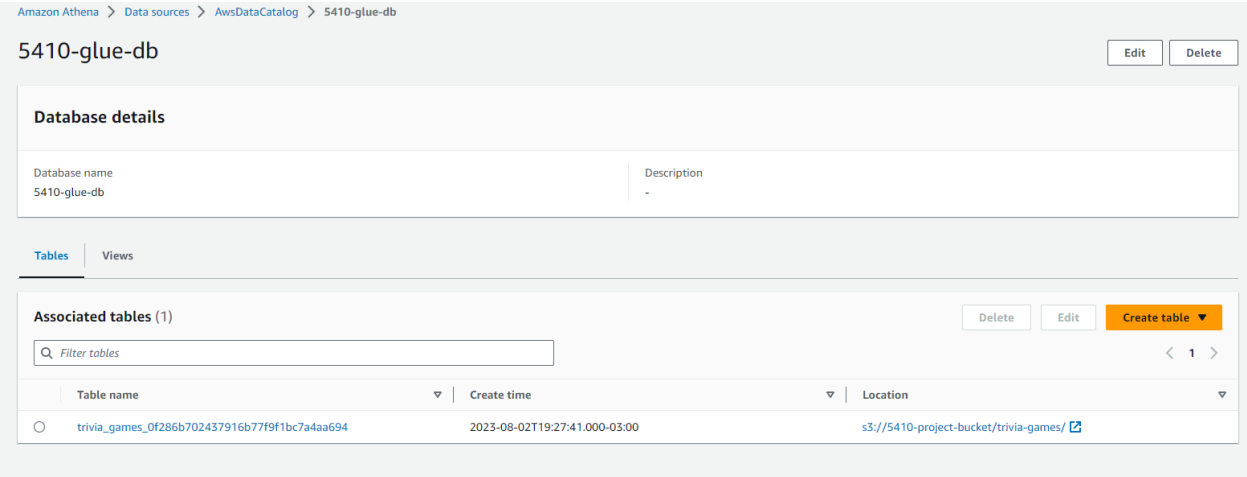


Figure 54: Athena Catalog integrations (From Glue to S3 to Athena)

What I actually built:

I was able to complete the feature using a combination of an API Gateway, four lambda functions, four DynamoDB tables, two SNS Topics for segregating notifications for administrators and users, EventBridge for automated cloud triggers and a combination of the GoogleSheet API and LookerStudio to satisfy part 3 of feature 7. Below are screenshots of the provisioned resources used in the project.

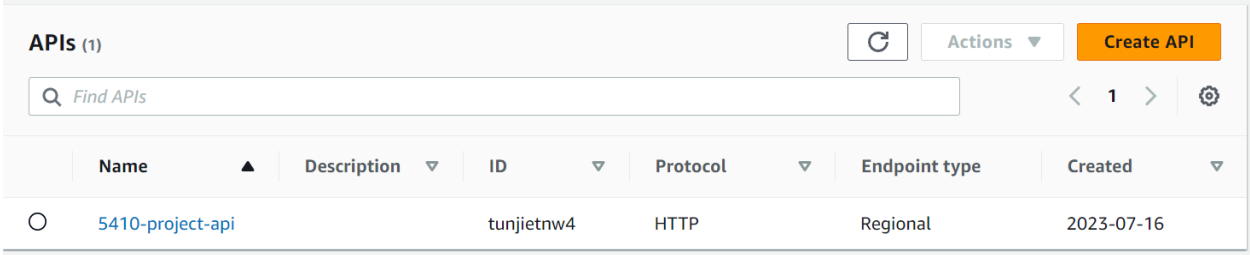


Figure 55: HTTP API

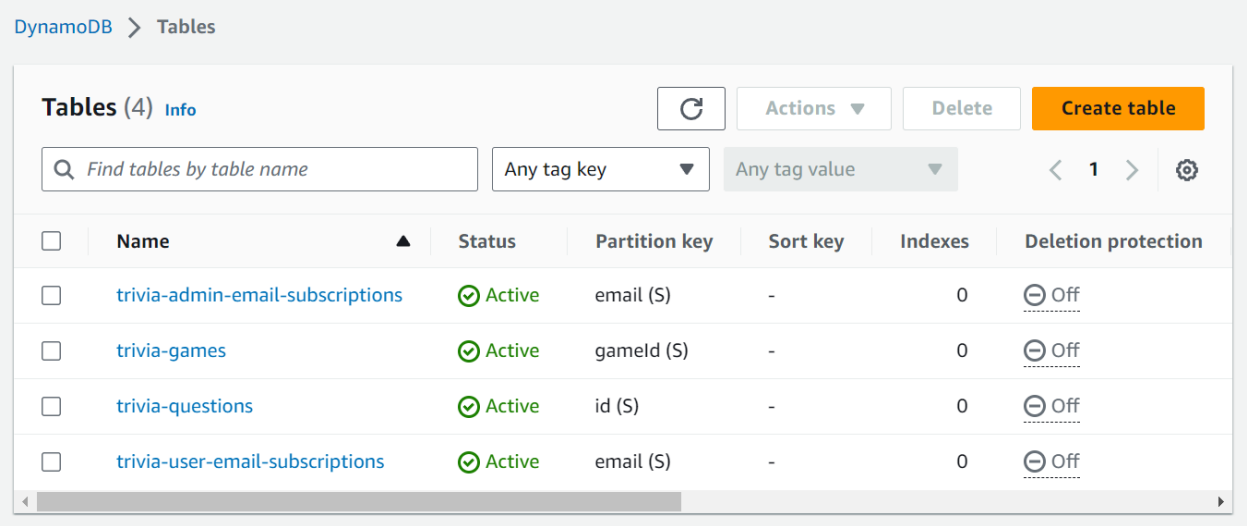


Figure 56: DynamoDB Tables

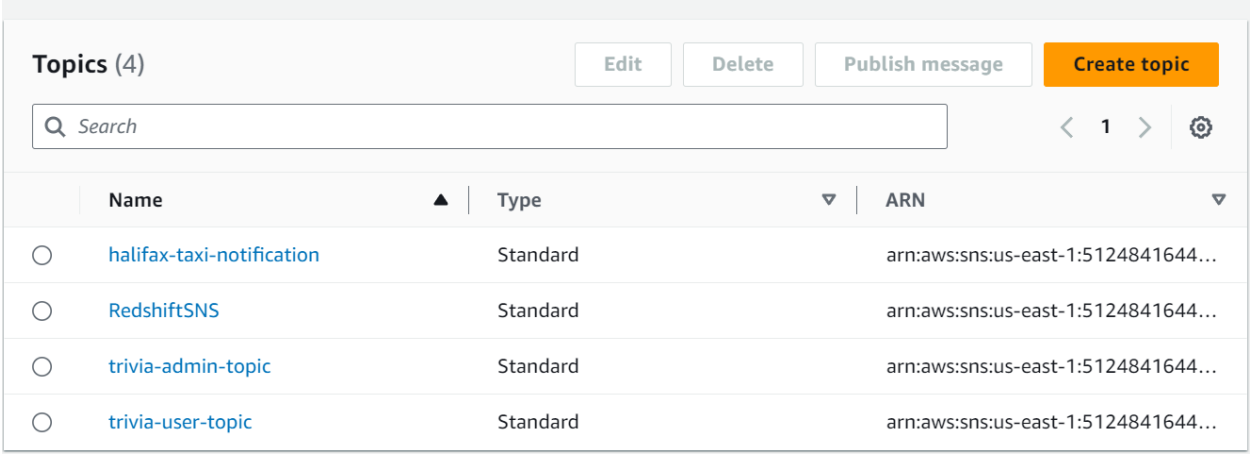


Figure 57: AWS SNS Topics

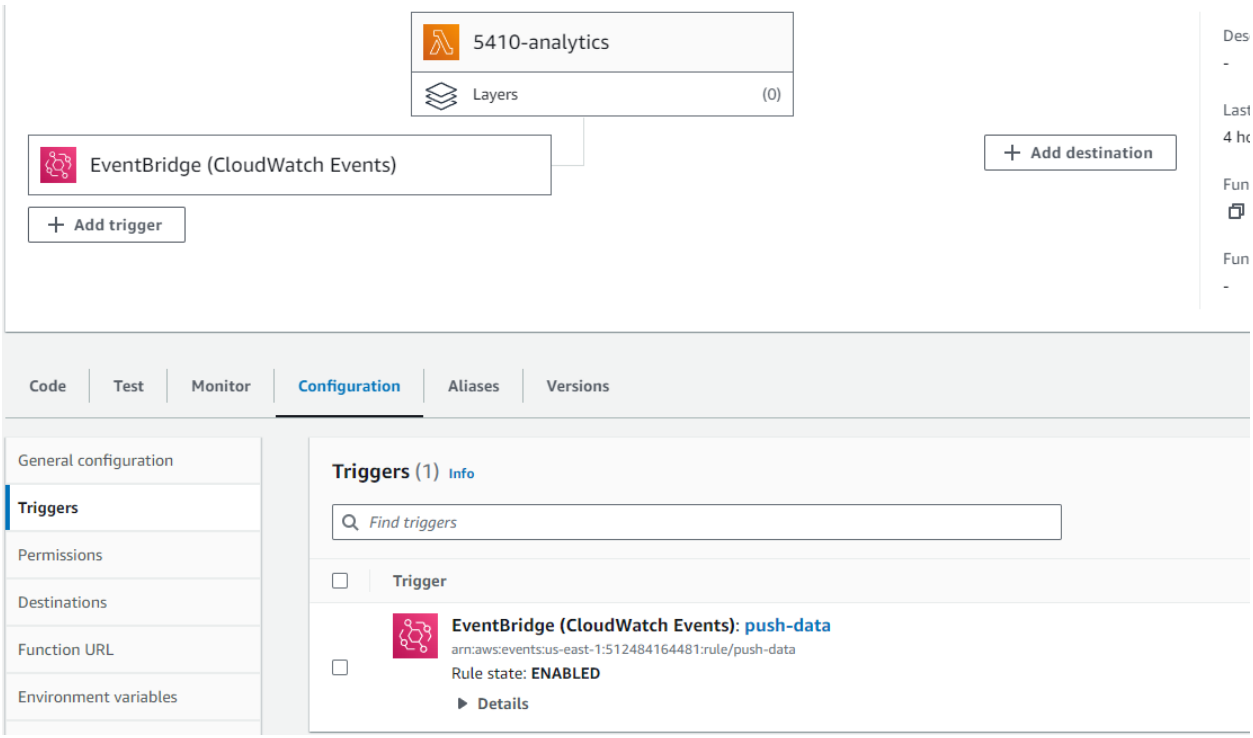
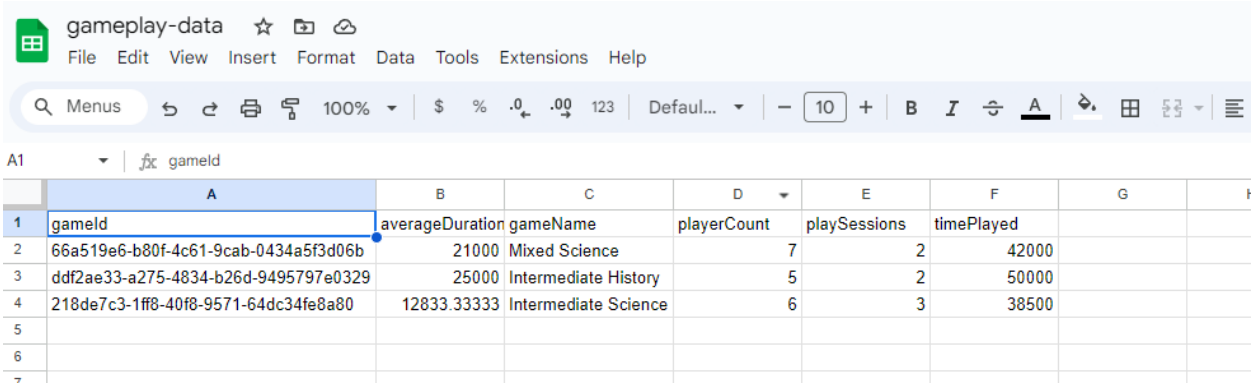


Figure 58: Events Bridge recurring (4 hours) Trigger for Data-Source Lambda



	A	B	C	D	E	F	G
1	gameId	averageDuration	gameName	playerCount	playSessions	timePlayed	
2	66a519e6-b80f-4c61-9cab-0434a5f3d06b	21000	Mixed Science	7	2	42000	
3	ddf2ae33-a275-4834-b26d-9495797e0329	25000	Intermediate History	5	2	50000	
4	218de7c3-1ff8-40f8-9571-64dc34fe8a80	12833.33333	Intermediate Science	6	3	38500	
5							
6							

Figure 59:GoogleSheets API Datasource

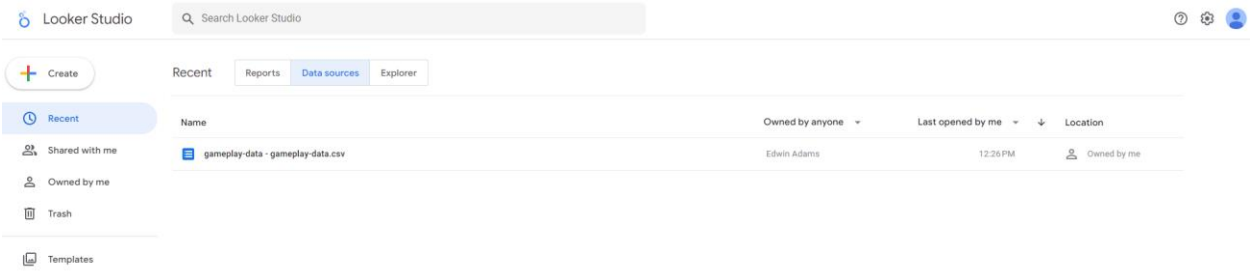


Figure 60: LookerStudio Data Source

Testing

Testing of the main functionality was doing using API Postman calls to the endpoints I exposed on API Gateway and integrated with my Lambda Functions. Below are a few screenshots from API Tests.

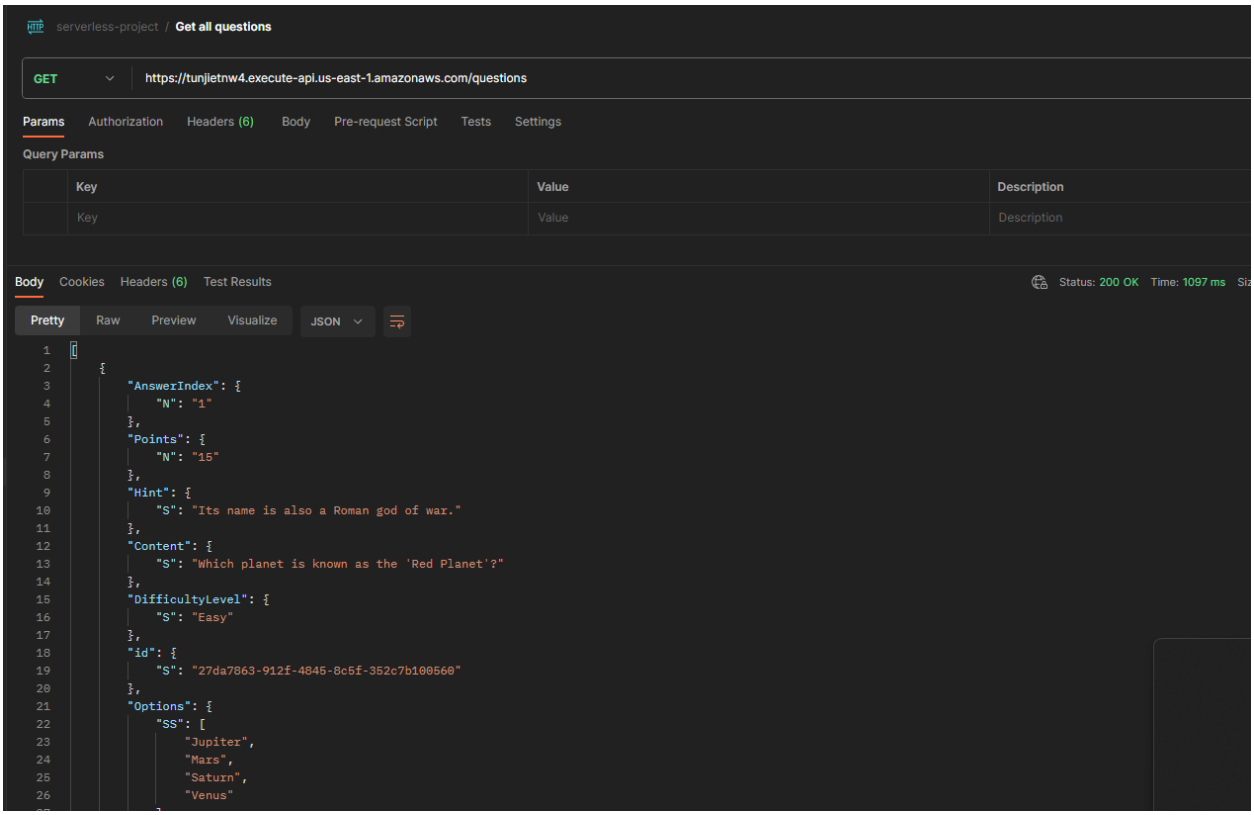


Figure 61: GET all questions

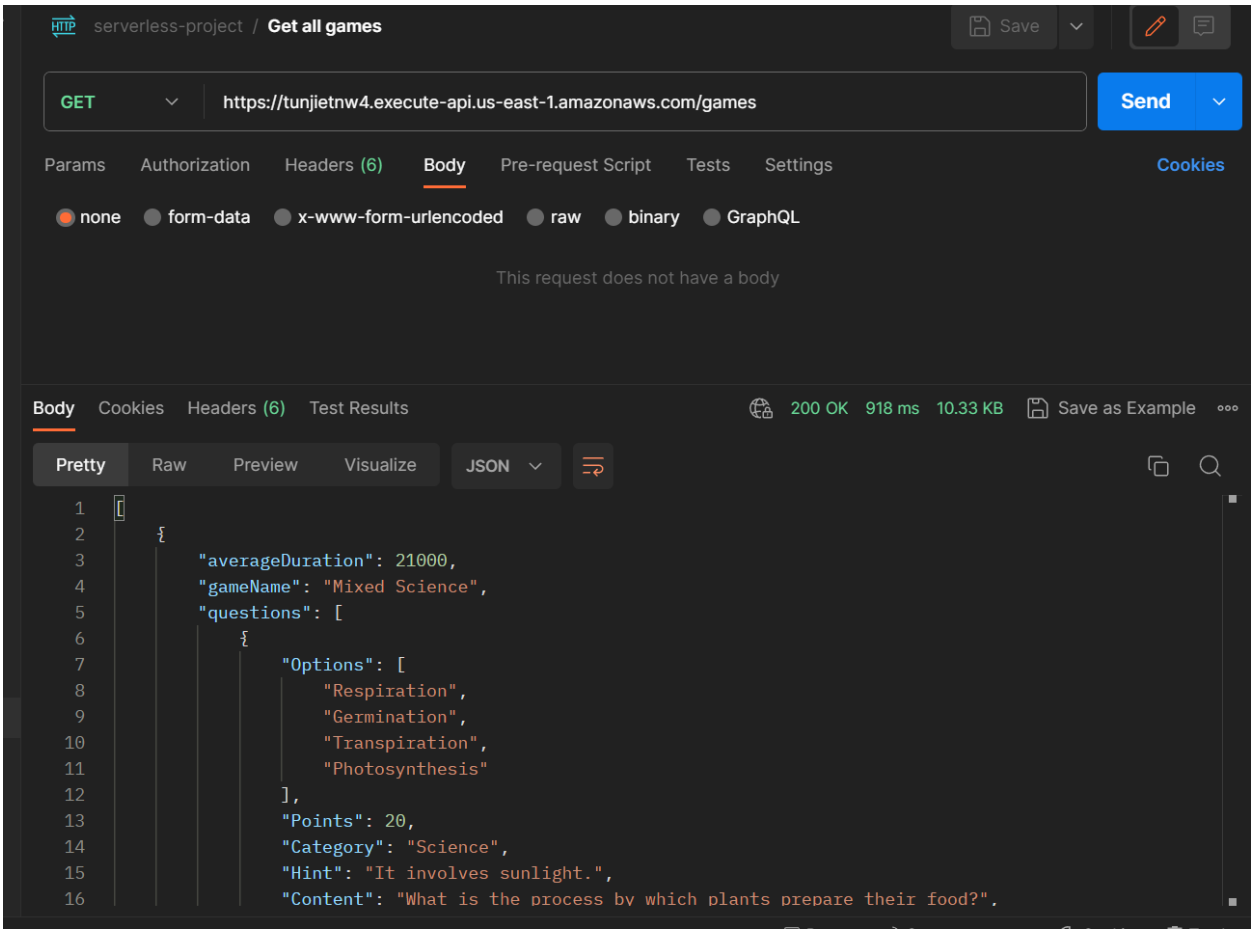


Figure 62:GET all Games

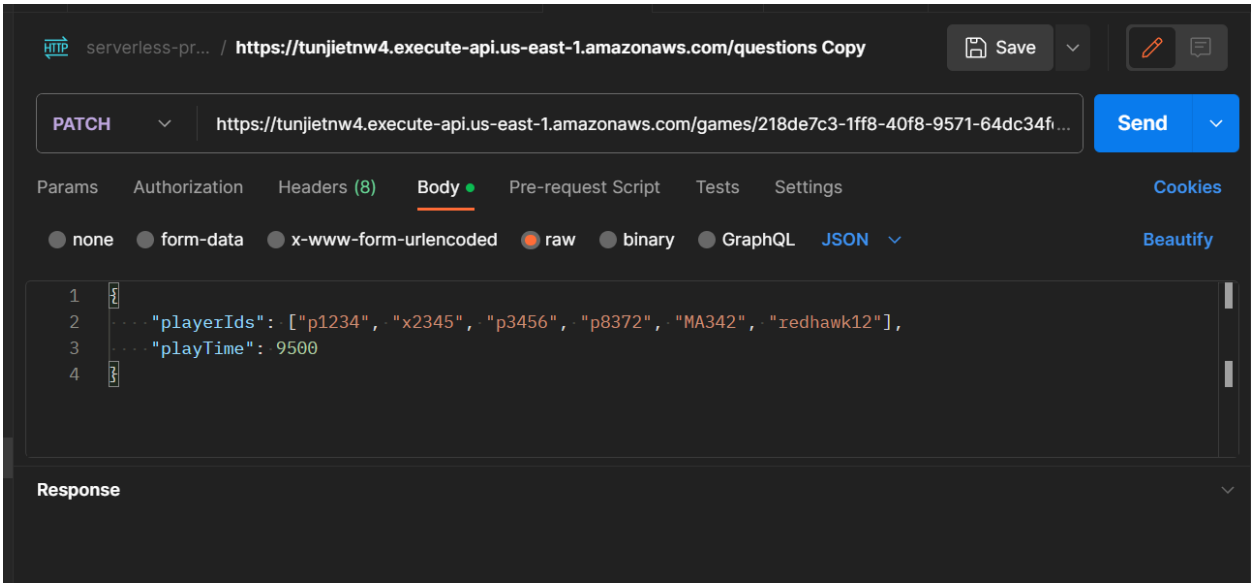


Figure 63: PATCH – Updates game data upon completion of a live game for data collection

8. Notifications and Alerts

Initial Plan

Initially I intended to complete this feature simple using SNS attached to a lambda function. However, during development, it became apparent that this would not be enough, because a user needs to subscribe to the notification list first, before they can receive notification messages. Another problem I faced was that even after a user already subscribed, my lambda function would send repeated emails requestion subscription rather than the actual email message.

Actual Implementation

To resolve the issues of recurring subscriptions, DynamoDB tables were implemented to store the lists of subscribed email addresses to the SNS Topic. What this allowed us to do then was to tie user subscriptions to the registration process, so that after that we could simply push notifications as needed. Similarly on the admin side, I set up monitoring alerts to be sent to my personal email whenever changes were made regarding trivia content management.

Testing

Below are some screenshots of email notifications I have been receiving as a result of API calls from the frontend application and postman to indicate that changes are being made:

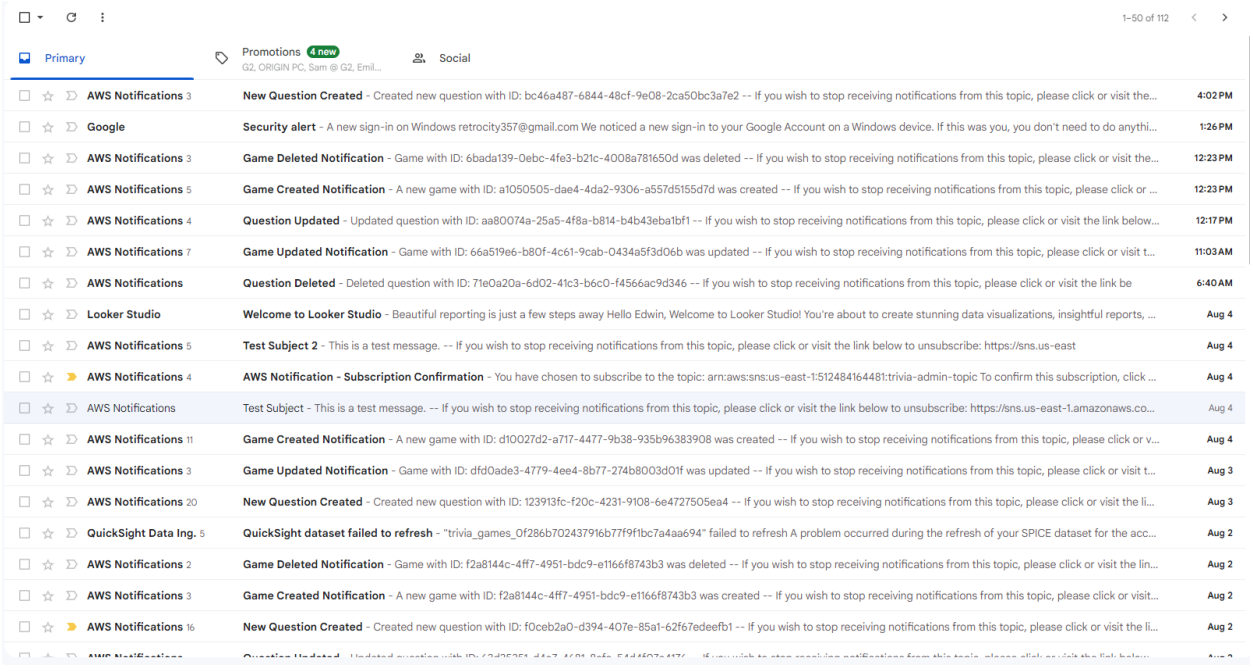


Figure 64: Admin Email Notifications

9. Automated Question Tagging

The Question Tagging Module is used to allow administrators to tag trivia questions with relevant categories. For now, the tagging option will be enabled for all the registered users as well as for admin. The user can suggest to us the question which can be added according to them in our system. The tagging mechanism will store the data given by the user in a database with its associated tags. The thought behind this is that, admin can choose and add questions to the question bank stored according to the tags associated with that.

Choice of Technology

My first choice for this task was Comprehend and Lambda. When I started browsing for the solution, I found that comprehend provides services like Sentiment Analysis, Entity Detection, Keyword Extraction and there was one service for tagging the document from S3 buckets. When I explored more about tagging of S3, it was more about classifying and not about providing actual suggestions. Also tried the approach of gathering the keywords from the comprehend API and then checking those words into a list of categories, but then this solution was not really good.

After this I moved to Natural Language API, and that is providing many services like Sentiment Analysis, Entity Detection, Syntax analysis, classifying content etc. After analyzing and testing I found Classifying Content API as best suited for our use case.

What I decided

For this task I decided to use the Comprehend API, so the plan was that there will be a lambda function which will be called by frontend with a question. That lambda will trigger AWS Comprehend service and find relevant words. I will maintain a list of words which can be associated with categories according to words suggested by comprehend. So we will store question along with tag in DynamoDB

What I did

After figured that this approach will not work, I transferred my focus from AWS to GCP for this task. NLP API of Google is working really well

I created a cloud function in python and tested with Postman API

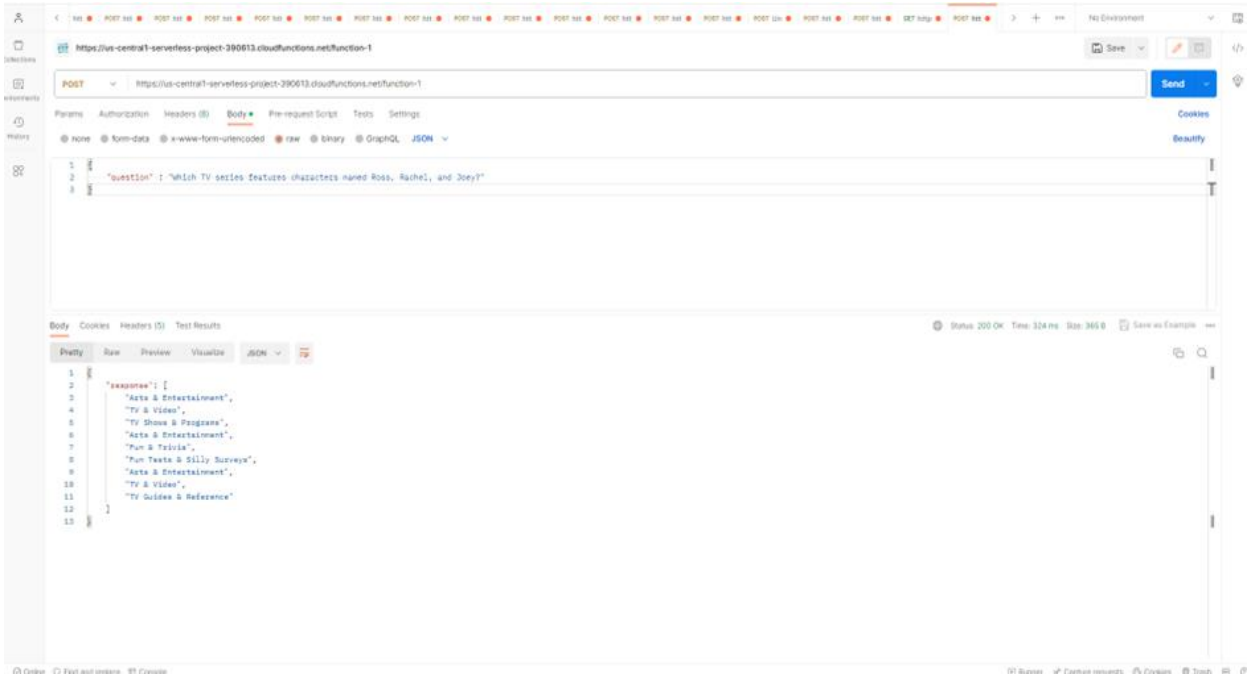


Figure 65: Postman Testing 1

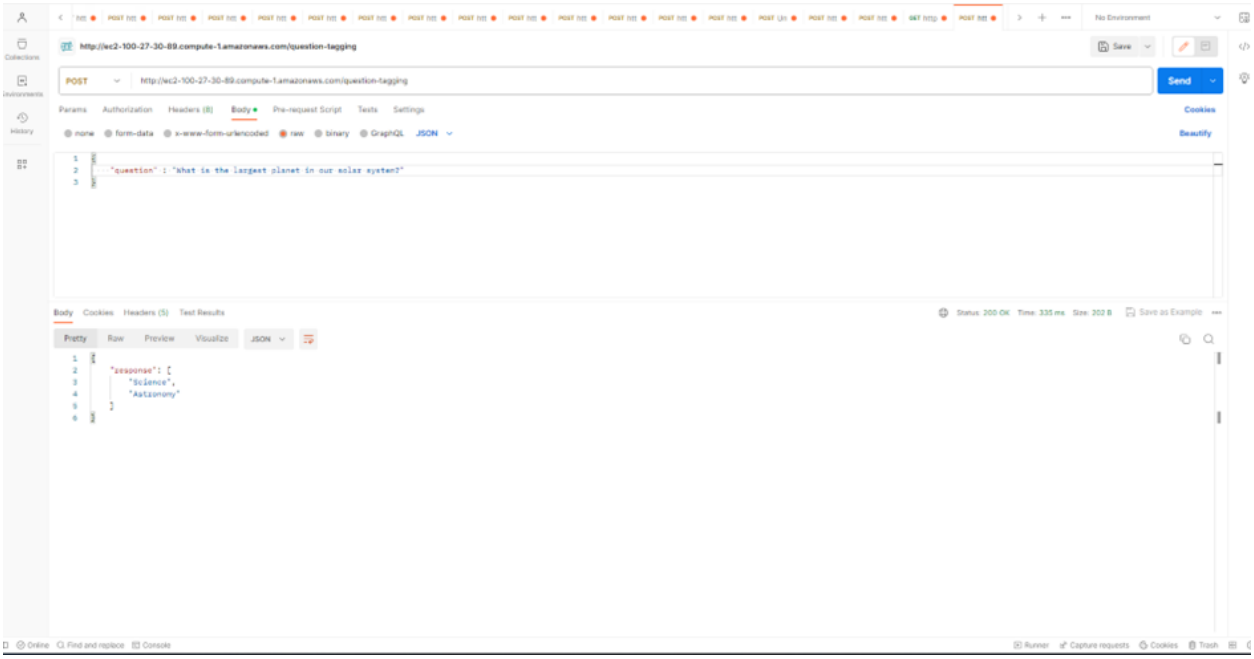


Figure 66:Postman Testing 2

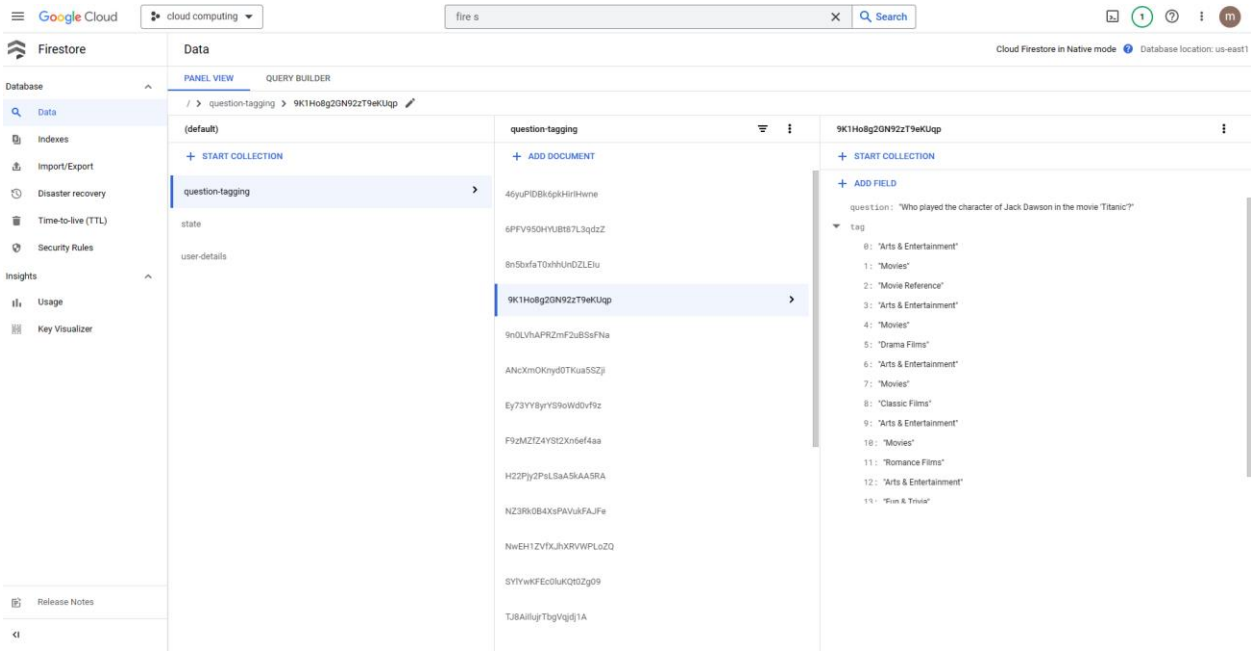


Figure 67: GCP Firestore table – questions-tagging

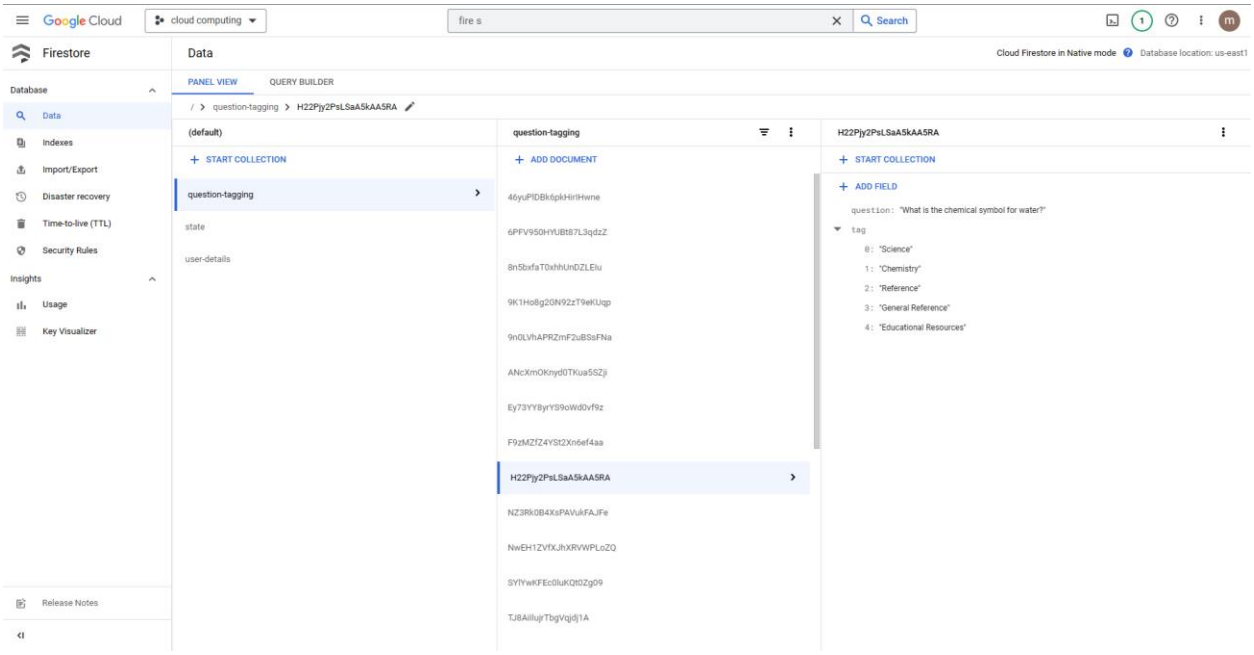


Figure 68: Questions tagging

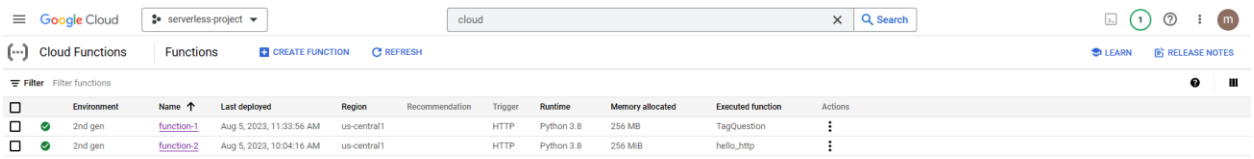


Figure 69: Cloud Functions

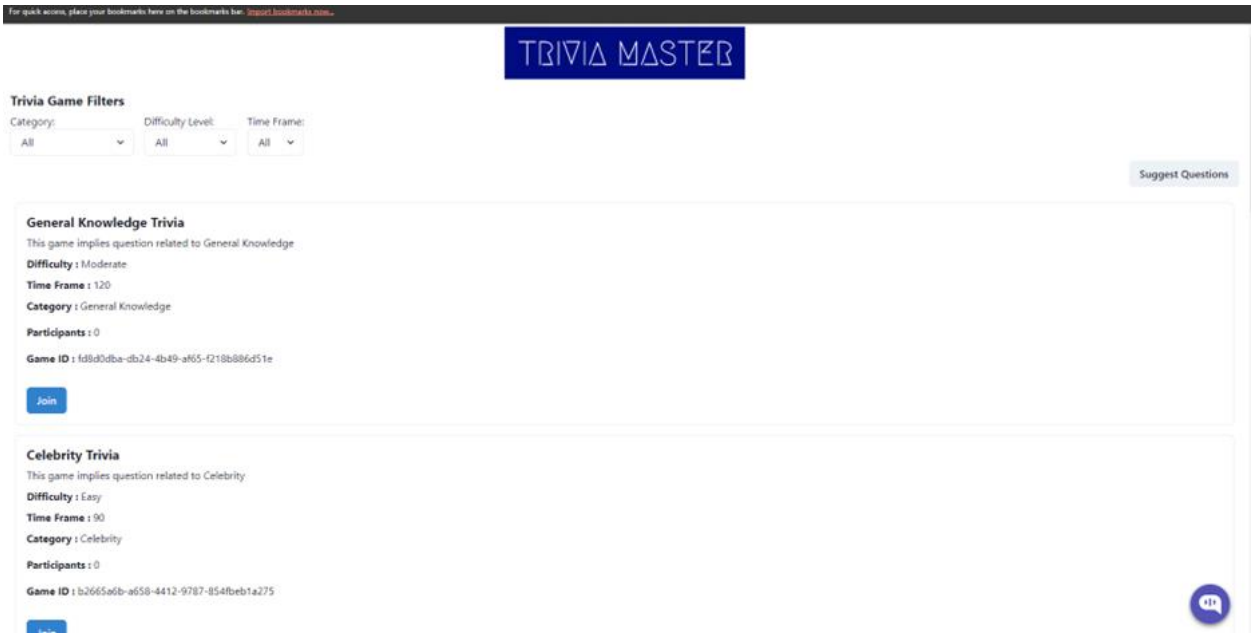


Figure 70: Game lobby

Figure 71: Question tags generation from frontend

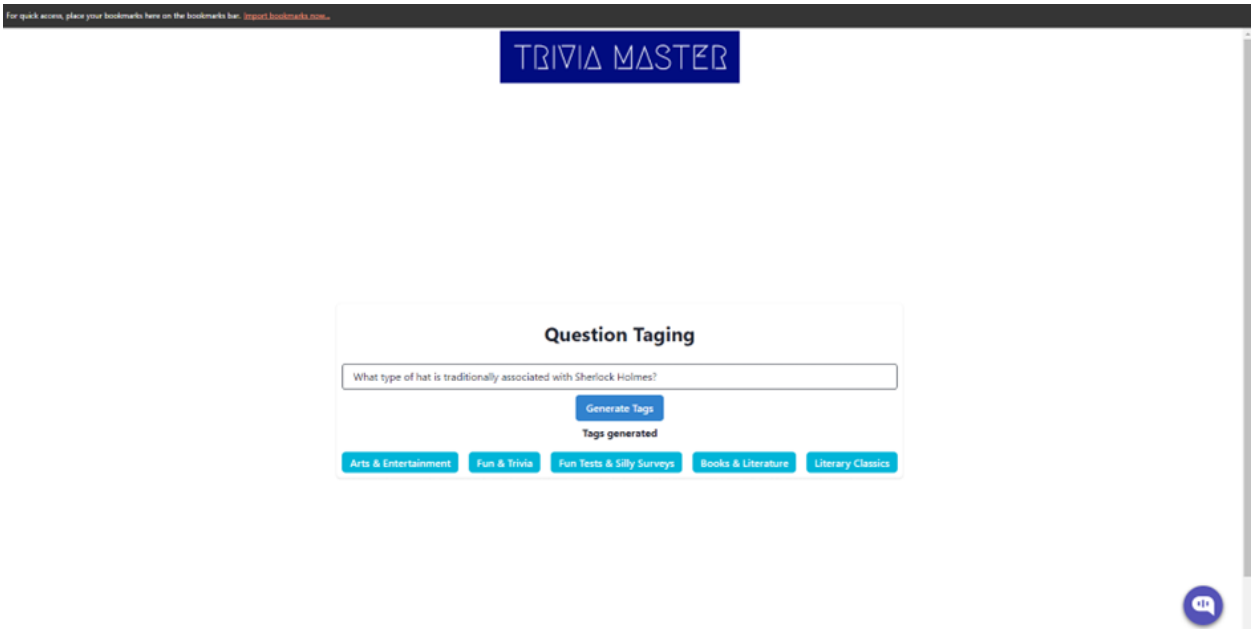


Figure 72: Question tagging 2

Planning

Table 17: Planned VS Used

Planned	**Used**
AWS DynamoDb	Firestore

AWS Lambda	CLoud Function
AWS Comprehend	Natural Language API

Tests performed

Table 18: Testing

Test Name	**Expectation**	**Pass**
Performance of every question	It should provide relevant tags	Pass
If no question tag returned	It should not store anything to DB	Pass
Many category questions	It should only give max 5 suggestion	Pass

10. Virtual Assistance

The virtual assistance is developed using AWS service Lex. This virtual assistance gives basic information on important navigations like, login, create team, explore game and view score. Here I have integrated a lambda function to get the score of the team dynamically from the database and give it to the user. This lambda will take the input from the lex bot and then perform a filter on the database and will calculate the total score and display the results to the user.

This lex bot is integrated with the frontend by using Kommunicate Library. This library directly integrates the bot with the UI.

Below are the screenshots for the functional testing of the VA.

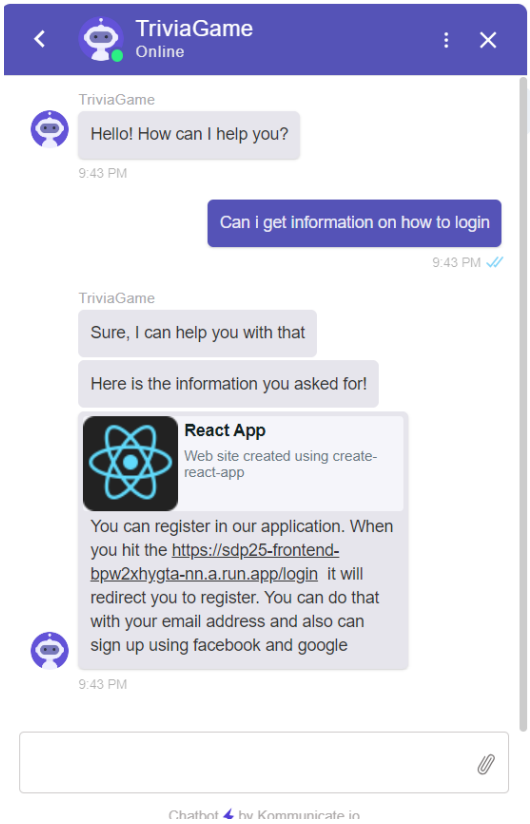


Figure 73: Chatbot

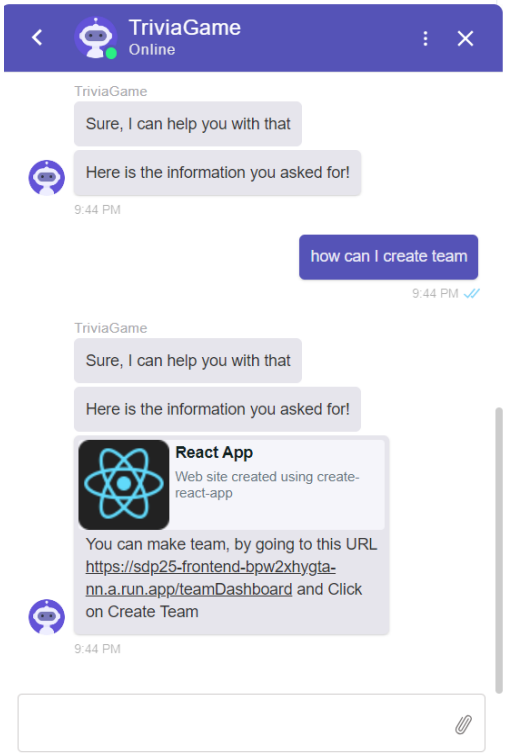


Figure 74: Chatbot response 2

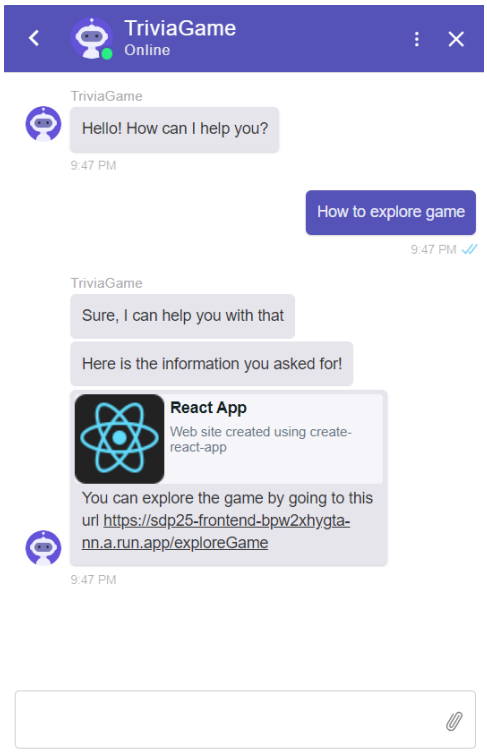


Figure 75: Chatbot response 3

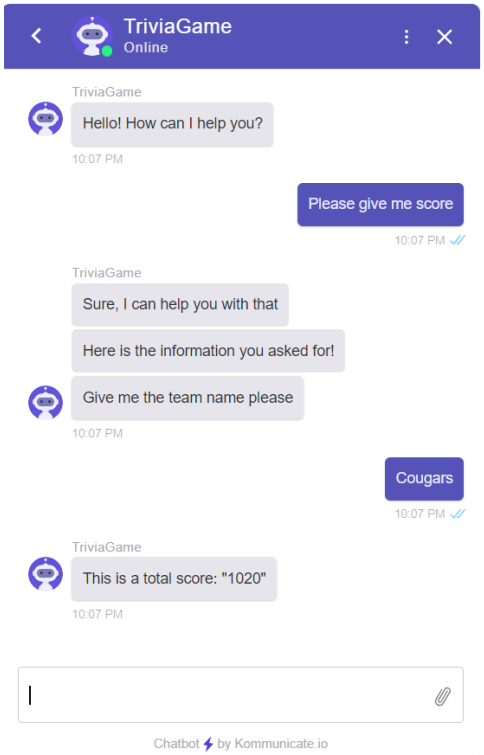


Figure 76: Chatbot response 3

The above chatbot answers 4 different questions, related to login, create team, explore games and score of teams.

For getting the team scored dynamically a lambda function is invoked. For that I had to enable the fulfillment of the intent that wants to calculate the score.

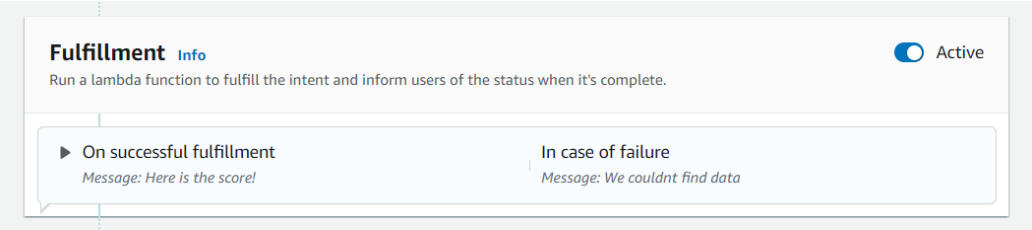


Figure 77: Chatbot fulfillment

This is the option where I can give the lambda i want to invoke.

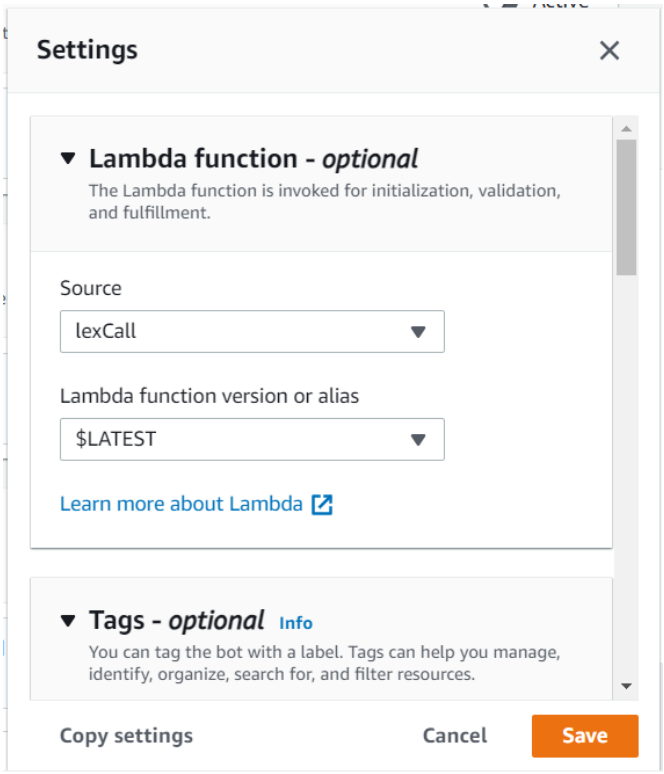


Figure 78: Call Lambda from chatbot

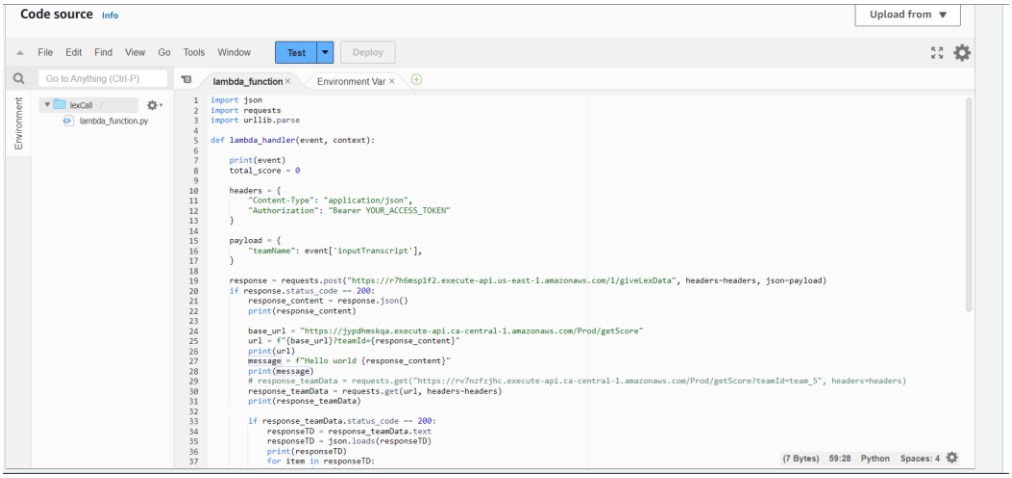


Figure 79: Called Lambda

The above snapshot has the code that calculates the total_score and returns it to the lex bot. The below screenshot displays the intentions used to make this bot.

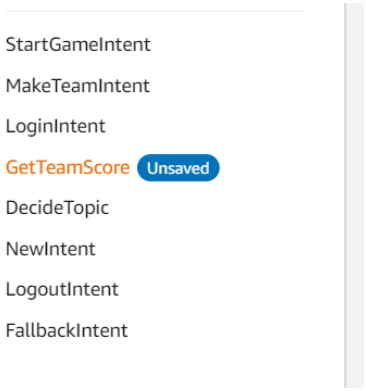


Figure 80: Intent calling lambda

The below screenshot demonstrates the conditional branching feature of the bot, it shows how the intent changes on the user's response.

Note: The AWS Lex only allowed 4 conditional branches so, I just implemented 4 conditions for the bot.

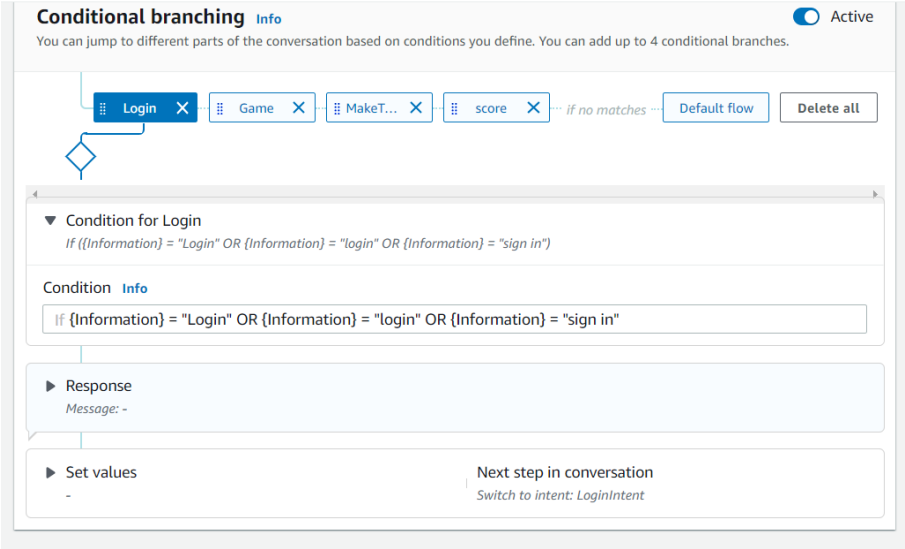


Figure 81: Condition in chatbot execution

API Testing (from postman):

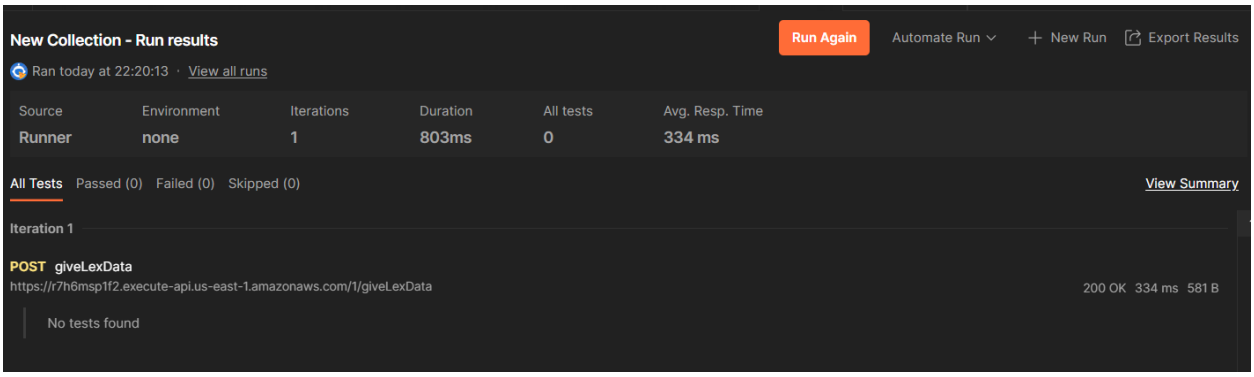


Figure 82; API testing

Method: Post Endpoint: /giveLexData.

Table 19: Planned VS Used Services

	Planned	**Implemented**
Virtual Assistant	AWS Lex	AWS Lex
Compute	Lambda	Lambda

Functional Testing:

Table 20: Chatbot Functional Testing

Test Name	**Expectations**	**Results**	**Status**

Start conversation with bot	Every time new conversation must be started	With the option to start the conversation, every time the new conversation is started	PASS
Previous conversation is preserved	Past conversations must be saved	Users can see their past conversations	PASS
Assistance	The virtual assistant should be able to answer the questions correctly and promptly	Yes, In no time the bot answers and resolves the users query	PASS

Application Roadmap

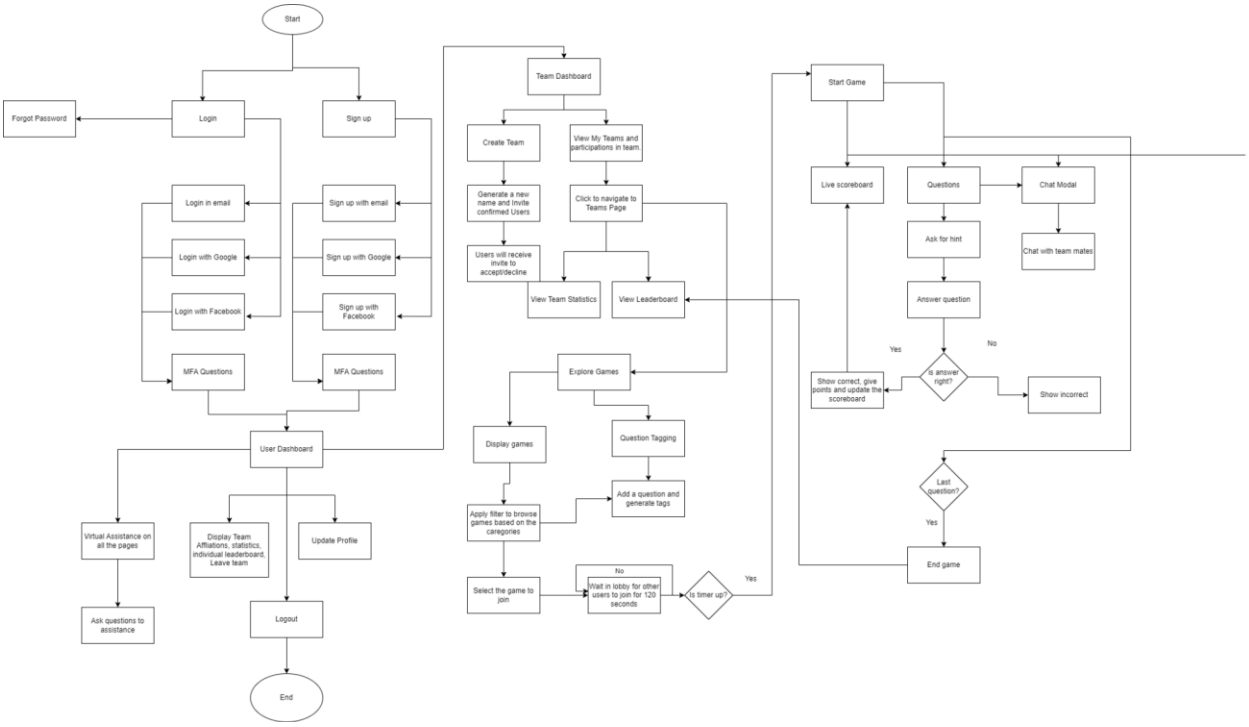


Figure 83: Application Roadmap [10]

We planned it exactly as we created it. However, we changed a few cloud services we wanted to use earlier, nothing was changed from the functionalist perspective. We implemented separate components following microservices architecture such that each component can work separately and statelessly. Then we integrated each other's components by giving proper routing and providing the required data.

User Authentication

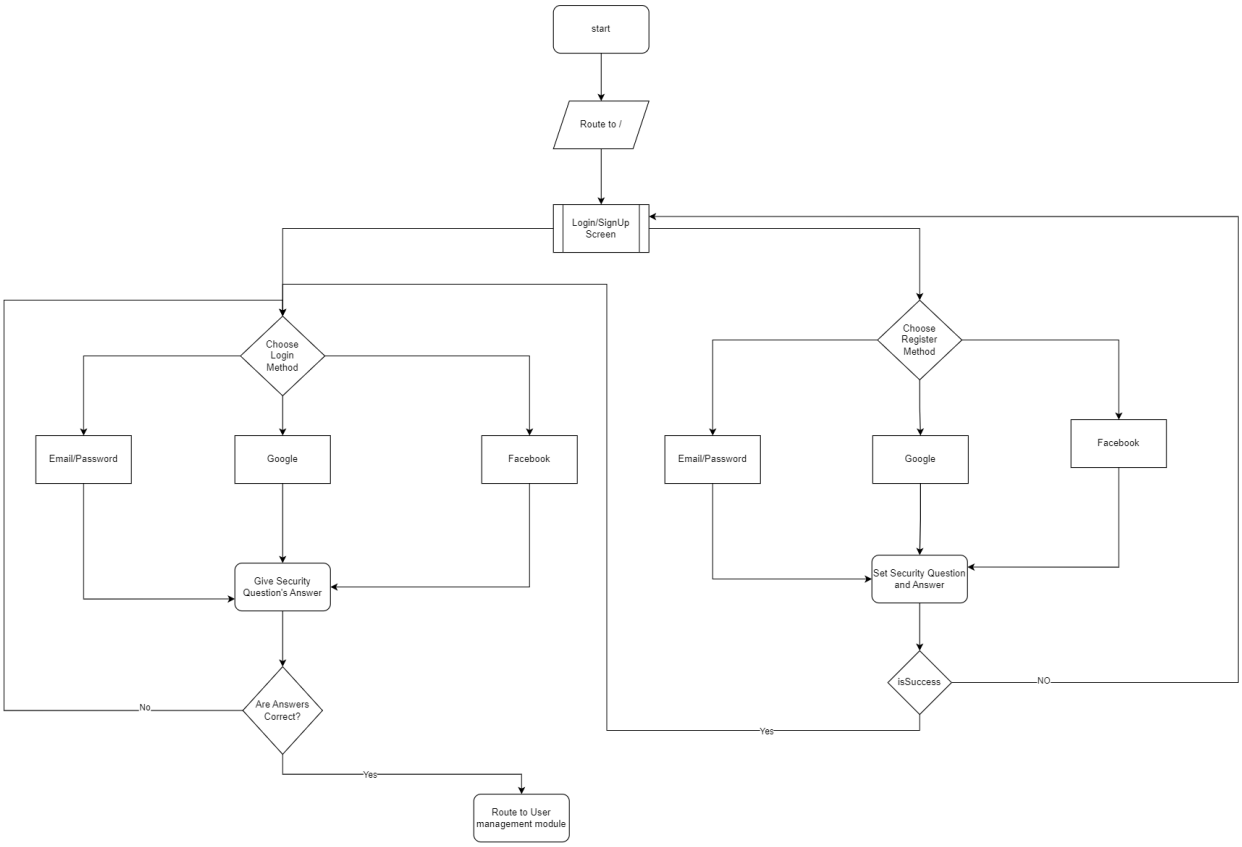


Figure 84: Feature 1 Flow diagram [10]

User Scenarios:

The User Authentication Module is designed with 2-factor authentications, and during the registration process, users are required to set up details for all three levels of verification. When a user attempts to log in, each level of verification is sequentially applied. The verification process follows these steps:

User provides email and password during login.

If the email and password are correct, the system proceeds to the next level.

If the email or password is incorrect, the user is prompted to provide the correct details and try again.

User is asked to answer the pre-set security question.

If the answer is correct, the system proceeds to the dashboard page.

If the answer is incorrect, the user is prompted to provide the correct answer and try again.

The planning and implementation of the 2-factor authentication process have been straightforward and consistent, ensuring that the user must provide the correct input at each level to proceed further. If the user gives a wrong input at any level, they are required to retry until the correct input is received by the system.

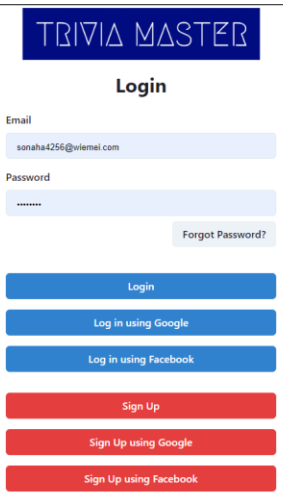


Figure 85:Functional Testing Feature 1

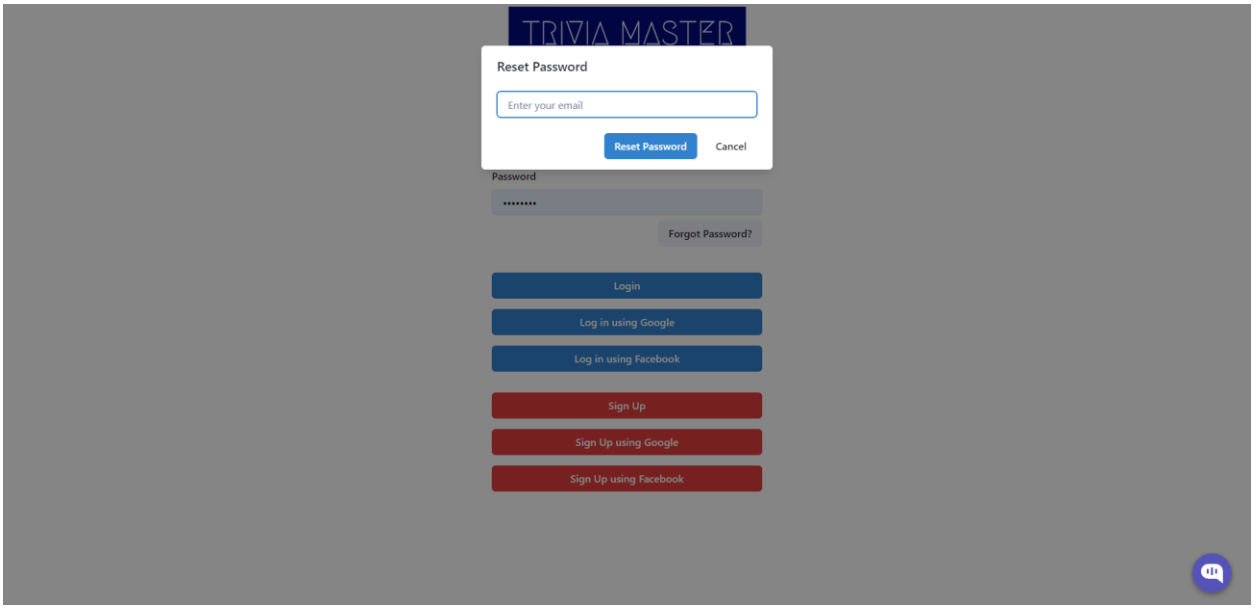


Figure 86: Functional Testing 2 - Feature 1

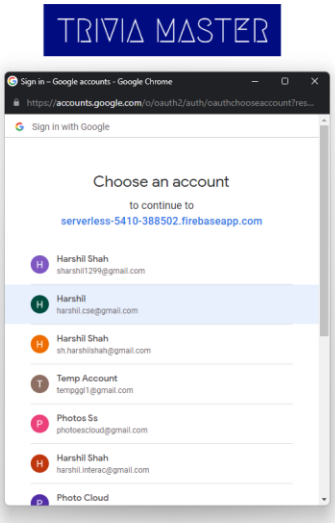


Figure 87: Functional Testing 3 - Feature 1

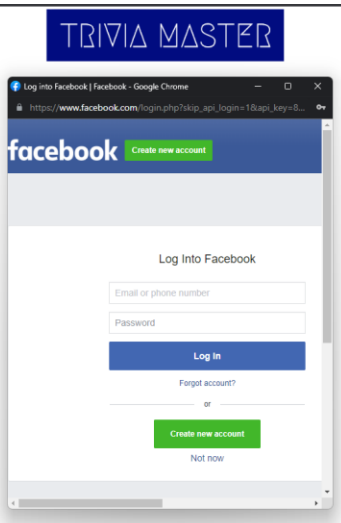


Figure 88: Functional Testing 4 - Feature 1

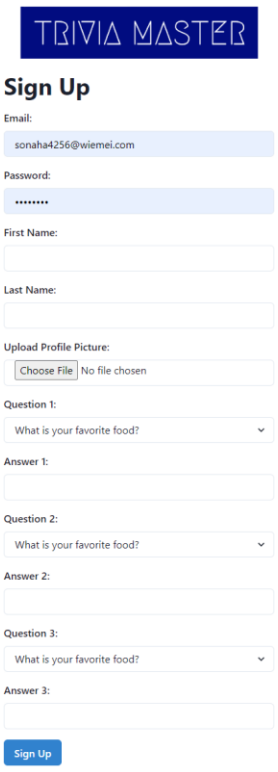


Figure 89: Functional Testing 5 - Feature 1

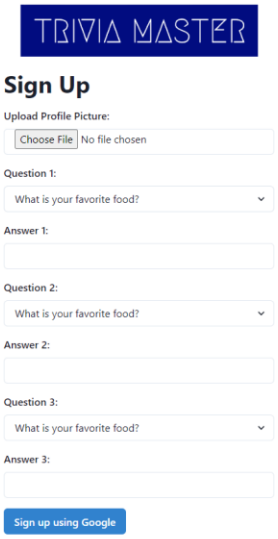


Figure 90: Functional Testing 6 - Feature 1

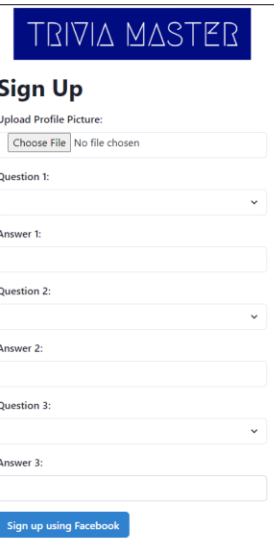


Figure 91: Functional Testing 7 - Feature 1

User Management

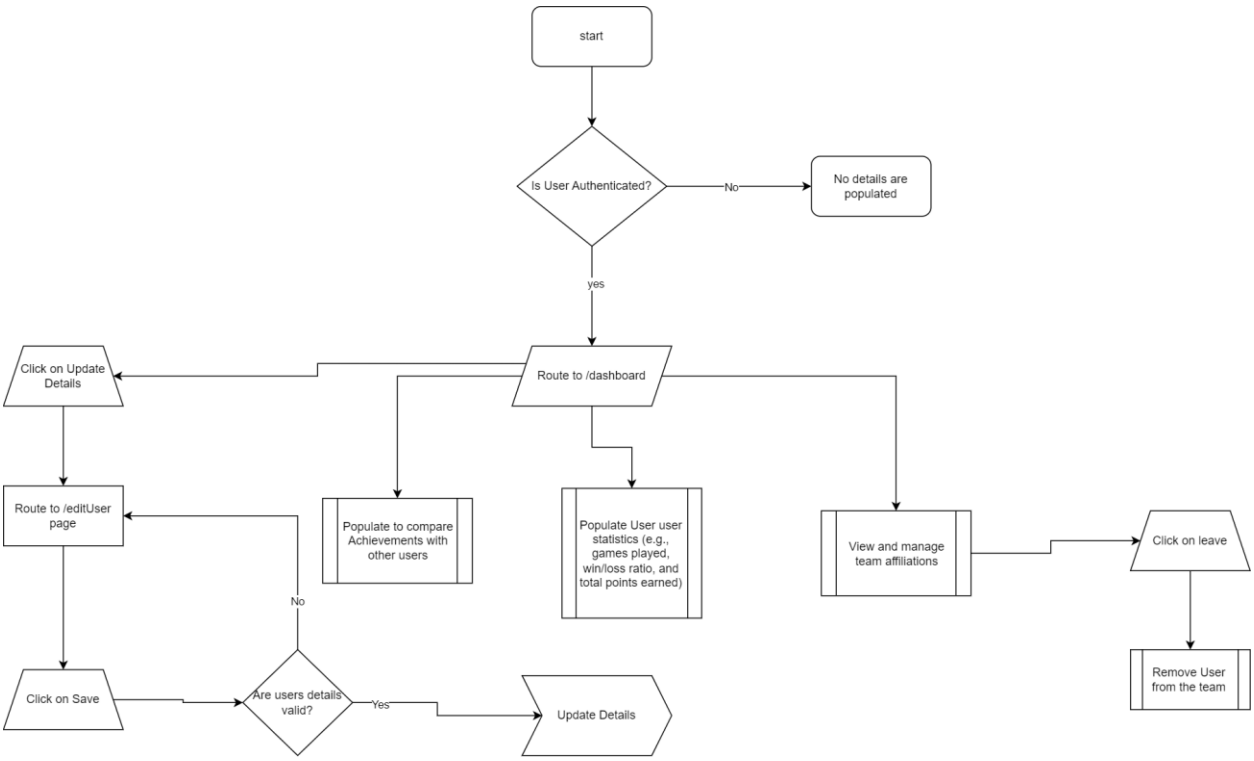


Figure 92: User Management - Feature 2 Workflow [10]

User Scenarios:

Obtain User information as well as user statistics and comparable other user statistics if the user is authenticated.

Otherwise, do not show user specific details.

To update user details, click on the Update Details button.

Reroute to /editUser

Validate all fields when use press on save details

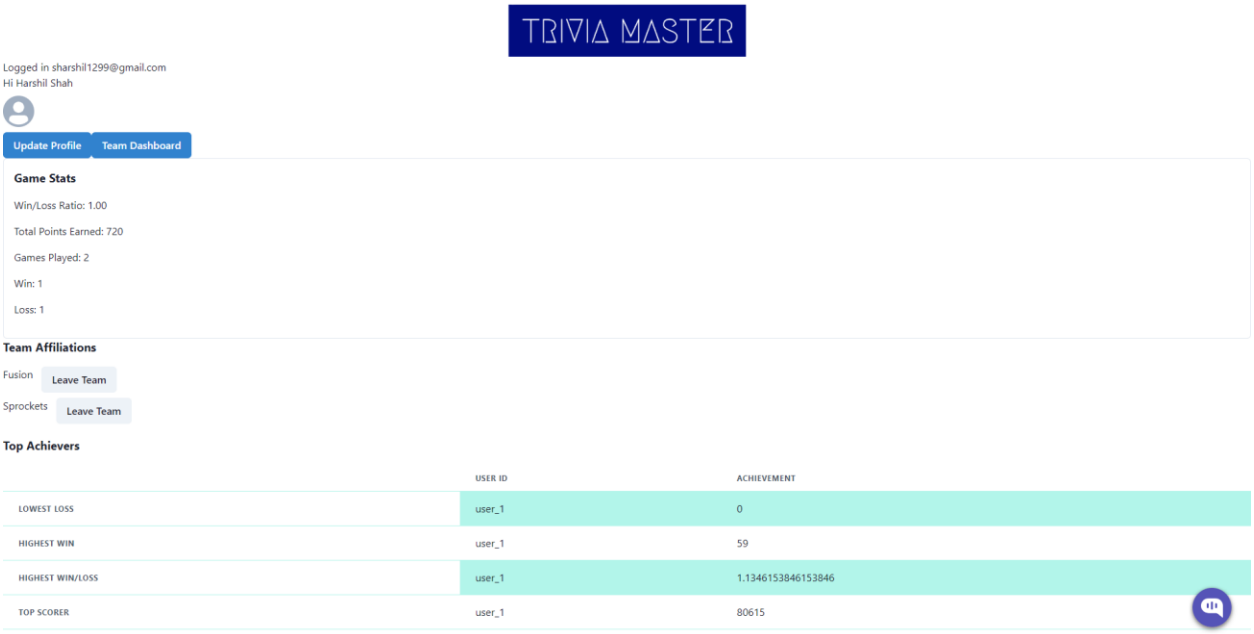


Figure 93: User Dashboard

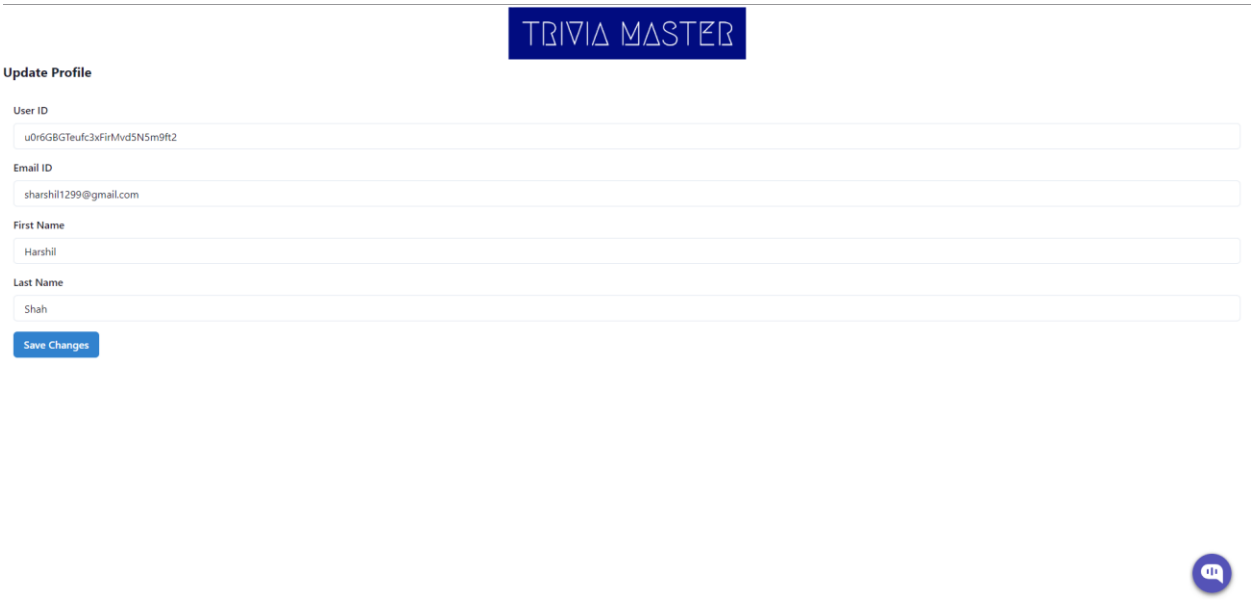


Figure 94: Update User Details

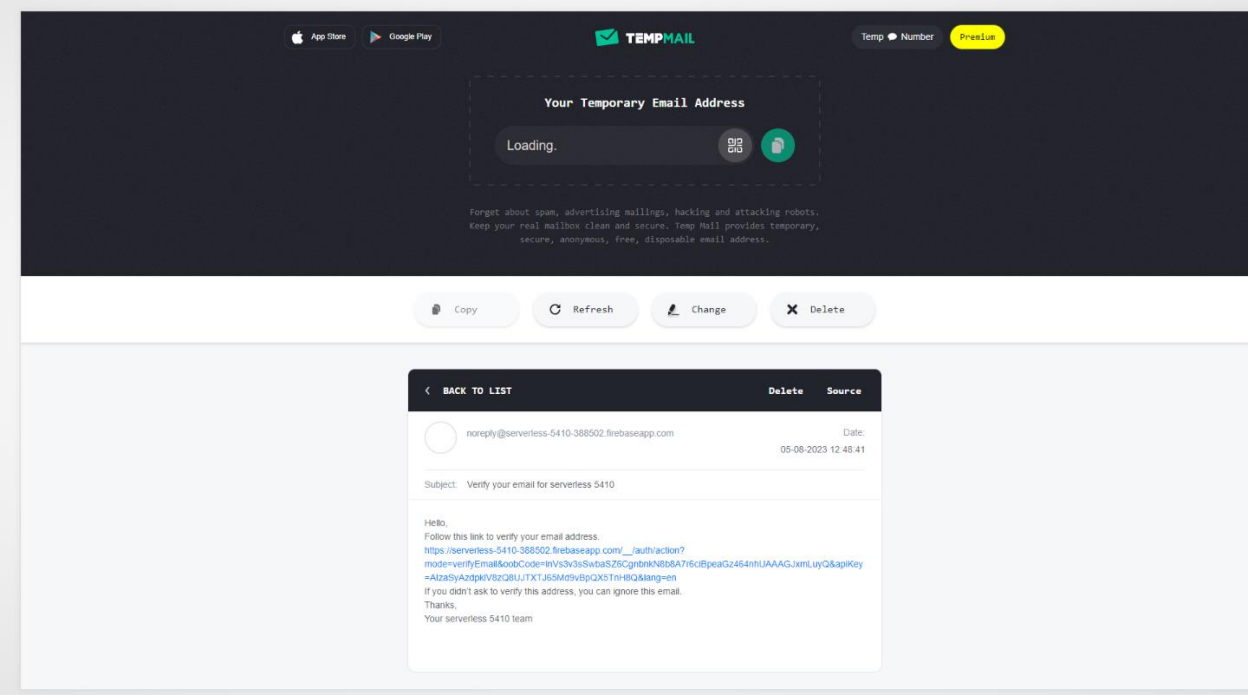


Figure 95: Verification mail on User register using Email ID

Team Management

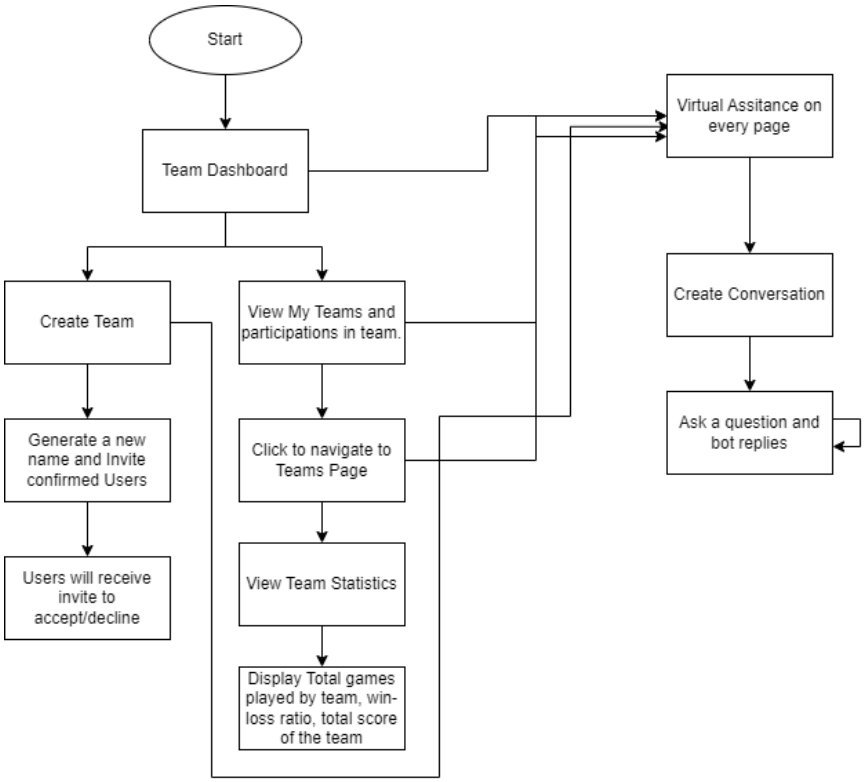


Figure 96: Flow Diagram Feature 3 [10]

The logged in user wants to play a game but does not have teammates nor is a part of any team. In this scenario, the user will create a team and invite people to join the team, just by clicking on the create team button and selecting the users, the selected users will get email notification and on accepting they will be added to the team. Now something happens and the logged in user does not want to play with a particular user, being the team owner, they can remove any user. And can promote anyone to admin so the admin will also have access to remove people from the team.
As a normal participant, they have the option to leave the game. And if any logged in user wants to see team statistics, they can view them by clicking on the View Leaderboard option.

Trivia Game Lobby

In the Trivia Game Lobby, users are presented with a list of available trivia games they can join. The lobby fetches essential game data, including the game title, category, difficulty level, and time frame, from a centralized data table. Users have the option to filter the games based on their preferred categories, difficulty levels, and time frames, allowing them to find games that suit their interests. The filtered games are displayed with relevant details, such as the game

name, category, difficulty level, and the number of participants already in each game. Users can then join a game they find appealing and wait for it to start. For scheduled games, a countdown timer is visible, indicating the time remaining until the game begins. On this screen if any new user joins the game, already joined person can see the list of user who have joined the game. At the scheduled start time, the game server fetches questions from the database based on the selected category, and the game starts, allowing all participants to answer trivia questions within a specified time frame.

In-game experience

Upon completion of the lobby registration and timer expiration, all users will be directed to a central screen where they will encounter questions presented one by one. Users have the opportunity to submit their answers within their respective teams. The win-loss ratio for each team will be computed based on the collective performance of its members. Each question will be displayed for approximately 20 seconds, after which the correct answer, along with an explanation, will be revealed for 5 seconds.

On the left side of the screen, a column will showcase the scores of all participating users in real-time, updating after each answer submission. Simultaneously, a chat window on the right side allows teams to communicate and discuss their answers. Additionally, a "hint" button is available to provide assistance with the questions. Once all the questions have been answered, participants will be redirected to the leaderboard screen to view the final rankings.

The scoring is calculated based on the promptness of the participants' responses, the submission status of their teammates, and the use of hints during the quiz. Faster answer times, synchronized team submissions, and non-reliance on hints contribute to higher marks. The scoring system takes into account individual and collaborative performances, emphasizing the significance of timely and well-coordinated contributions from team members.

Leaderboards

In the leaderboard screen, users are presented with graphical representations showcasing the scores of individual players and teams throughout the entire duration of the game. The interface offers filtering options, enabling users to view data based on daily, weekly, monthly, or all-time statistics. Two graphs are dedicated to displaying the static leaderboards, one for teams and another for individual players. Additionally, two more graphs exhibit the top-performing teams and players, highlighting their achievements within the game.

Automated Question Tagging

The user enters a question into the application that incorporates the question tagging model, which uses the Natural Language API. The model processes the input question, analyzing its context and semantics, and predicts relevant tags or categories. The user then sees these predicted tags displayed on the frontend of the application.

Virtual Assistant

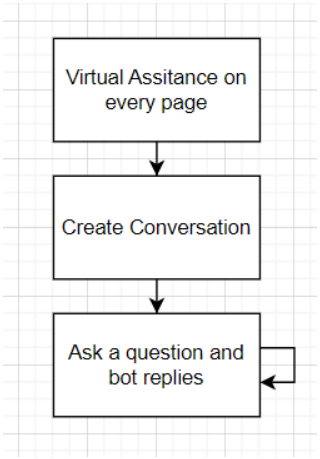


Figure 97: Flow Diagram Feature 10 [10]

The logged in user faces difficulty in navigation, they can take help from the VA and ask them proper questions and the VA will provide assistance to navigate through the web application. If they want to know the current total score of a team, the user can ask VA for the score of a particular team and the assistant will reply with the total score of that team.
These are user flow diagrams as to how the user will be using every feature can be explained using flow charts feature by feature and possible user scenarios can be explored in the flow diagram.

Architecture

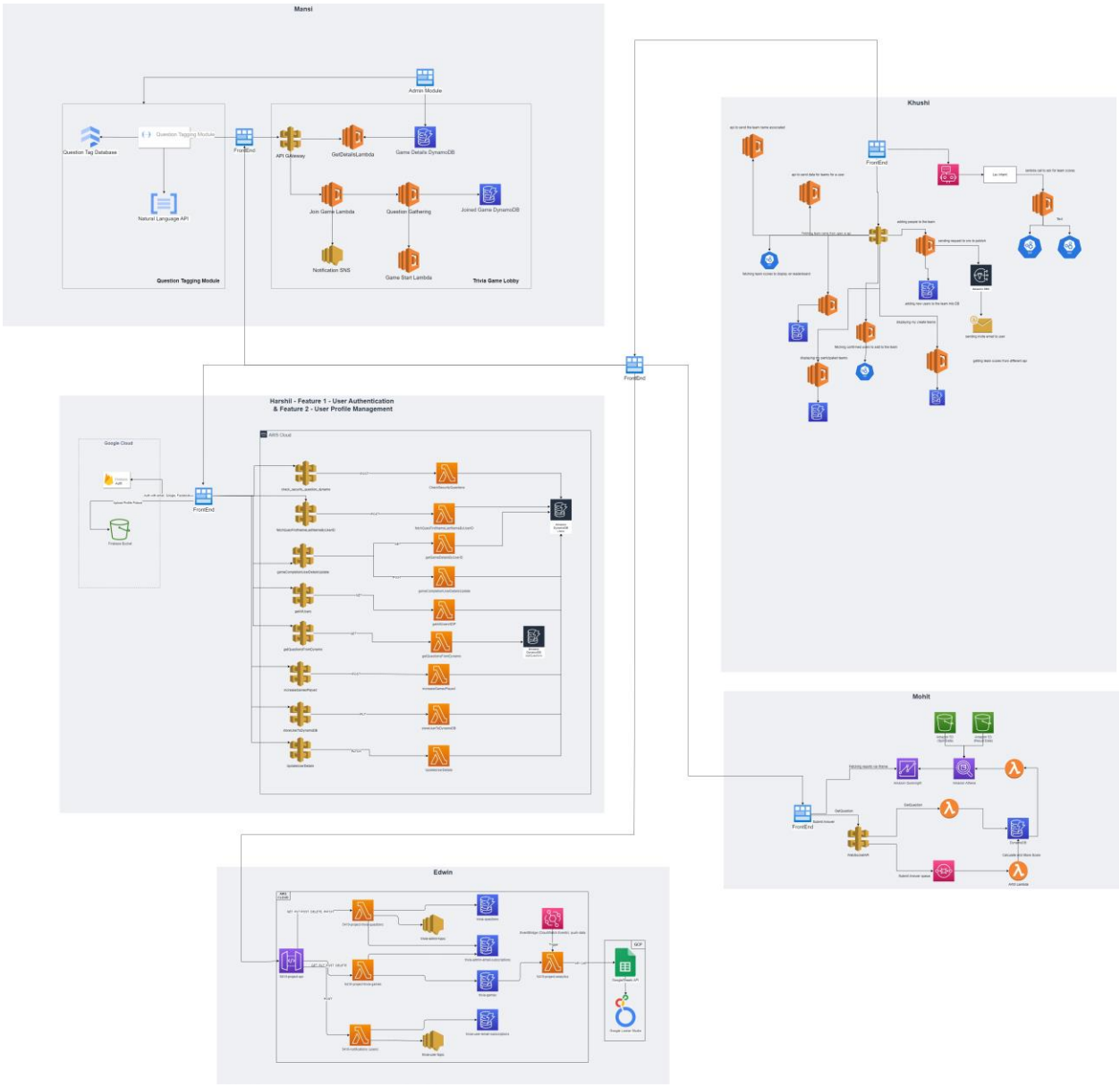


Figure 98: Architecture Diagram [10]

Worksheet

Module Owners

Table 21: Module Owners

Module	**Module Owners**
1. User Authentication	Harshil Shah
2. User Profile Management	Harshil Shah
3. Team Management	Khushi Shah
4. Trivia Game Lobby	Mansi Vaghasiya
5. In-game Experience	Mohit Sojitra
6. Leaderboards	Mohit Sojitra
7. Trivia Content Management (Administrators)	Edwin Adams
8. Notifications and Alerts	Edwin Adams
9. Automated Question Tagging	Mansi Vaghasiya
10. Virtual Assistance	Khushi Shah

There were no changes in the allocation of work within the team members.

Paragraph on Individual Component:

1. User Authentication (Harshil Shah)

In this implementation, the initial plan was to utilize AWS Cognito for user authentication and authorization, with user details stored in AWS DynamoDB and question-and-answer details in GCP Firestore. AWS Lambda would handle data storage and cipher key generation. However, during the build process, the decision was made to switch to Firebase Authentication for user authentication due to its ease of implementation and compatibility with multiple sign-in services. User details, including email, name, question-answer pairs, and other information, continue to be stored in AWS DynamoDB using AWS Lambda for data management. The initial idea of using GCP Firestore for question-and-answer details was dropped in favor of

maintaining consistency within the AWS environment and reducing complexity. Additionally, GCP Buckets were employed to store profile picture images, with the image URLs stored in DynamoDB.

2. User Profile Management (Harshil Shah)

In the initial plan, the user interaction system was intended to be built using GCP Firestore and Cloud Functions for scalability and efficiency. Firestore collections were proposed for user profiles, user statistics, team affiliations, and achievements, with Cloud Functions handling interactions and data updates. However, during the implementation, the decision was made to switch to AWS DynamoDB and Lambda Functions. User profiles, statistics, and achievements were stored in DynamoDB, and Lambda Functions were developed to handle profile updates, calculate statistics, and manage team affiliations securely. Despite the deviation, the AWS stack successfully delivered a scalable, robust, and well-integrated user interaction system, meeting the project's requirements effectively.

3. Team Management (Khushi Shah)

Team management is developed using AWS Lambda, API Gateway, DynamoDB and SNS. It has the requirement for dynamically generating unique team names by integrating open ai API which generates new names. Once the name is generated, users who have SNS already subscribed will be displayed in the dropdown. When the user selects the invite option, the system automatically sends personalized emails with SNS. When a user accepts an invitation, the corresponding data is promptly updated in DynamoDB. The email will have an accept or decline option. And based on the user selection, the DB will be updated accordingly. Finally, the team dashboard is the central page for team management. It displays various team names and by clicking on any team, the system navigates to the team page. The Team Dashboard also grants team creators more rights, such as the ability to make users admins or kick them off the team. The "View Leaderboard" feature enhances the gameplay by giving players access to crucial team information including games played, win-loss percentages, and overall point totals.

4. Trivia Game Lobby (Mansi Vaghasiya)

Trivia game lobby is developed using AWS Lambda, DynamoDB and SNS. Using this module the admin generated game and In Game Experience will be coordinate. Lambda function will be called for fetching the information from DynamoDB. This data will be filtered by criteria and visible to the end-user. When the user clicks on the join button the game will be started and users will get an email in the general channel saying that the game is getting started in our lobby. Also we will make an entry in DynamoDB for the first user and count down will be

started from the first join click. When other users will be joined they will see the name of users already joined in the game and when the counter goes to 0 all users will be redirected to In game experience page. Meanwhile the data will be fetched for category questions to make requests in In Game Experience to start the game.

5. In-game experience (Mohit Sojitra)

During the in-game experience, pertinent data is extracted from the trivia game lobby component, where game selection and scheduling occur. After the game schedule is finalized, the required information, such as game questions, correct answers, start time, number of teams, and hints, is retrieved from Amazon DynamoDB.
Once the game commences, backend communication with the frontend is established through WebSocket connections. The backend sends questions to the frontend every 25 seconds, allowing the frontend to display each question for 20 seconds, followed by a 5-second display of the correct answer and its explanation. Upon the frontend's submission of answers and corresponding timings, the data is forwarded to AWS SQS from Amazon DynamoDB.
The backend then proceeds to calculate the scores based on the responses, storing the results back into DynamoDB. Simultaneously, the updated scores are transmitted to the frontend in real-time for display. Upon game completion, a game-over message is dispatched to the frontend via Lambda, prompting the frontend to redirect to the leaderboard screen for final rankings and results.

6. Leaderboard (Mohit Sojitra)

The leaderboard feature retrieves real-time data directly from the Amazon DynamoDB table. To accomplish this, it leverages AWS Athena serverless, which is responsible for querying the data stored in DynamoDB. AWS Lambda and AWS S3 play essential roles in this process, aiding in the execution and storage of the query results obtained from Athena.
Once the data is successfully queried from DynamoDB and processed through AWS Athena, it is seamlessly integrated into an Amazon QuickSight dashboard. This dashboard provides a comprehensive and visual representation of the leaderboard, allowing users to access up-to-date scores and statistics. The combination of AWS Athena, AWS Lambda, and AWS S3 empowers the system to efficiently manage and display dynamic leaderboard data, providing a smooth and effective user experience.

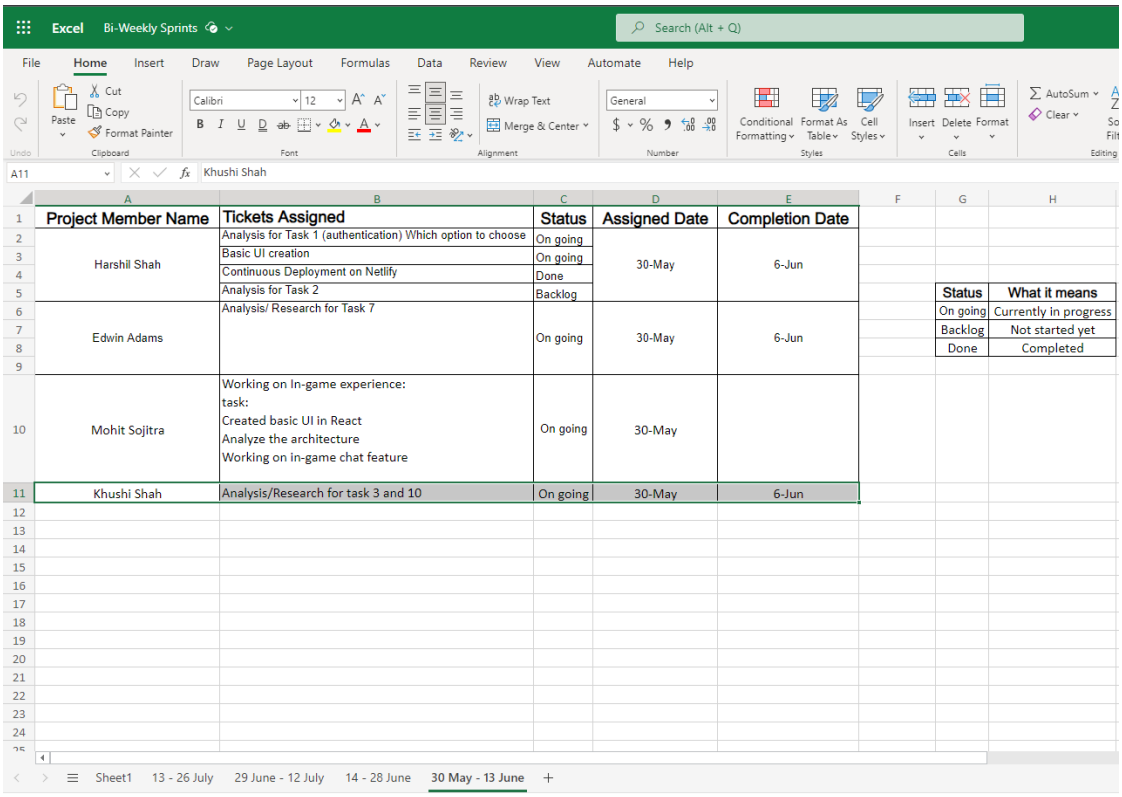
9. Automated Question Tagging (Mansi Vaghasiya)

In the Question Tagging module, the Natural Language API is used to find associative tags for questions. Additionally, Cloud Function and Firestore are employed to store question data. This

module has an individual page for user experience. When a user adds any question from the page, the Cloud Function will be triggered, which in turn calls the NLP API for classification. The Language module of the NLP API classifies the text content and suggests some category tags that may be relevant to the user's given data. There are various tags that can be associated with a question, but for simplicity, only 5 will be considered.

10. Virtual Assistance (Khushi Shah)

Virtual Assistance, developed using AWS Lexi, is a tool designed to provide users with basic information and facilitate navigations such as login, team creation, game exploration and score viewing. One of the information and standout features of this bot is that it provides a dynamic scoring system. This feature is made possible by integrating it with Lambda function and the lambda function is connected with the database. When users ask for the score, the bot asks the team's name and when the users give the team name it will take that value and invoke the lambda function and will fetch the total score of the team and display it in the response of the bot. This lex bot is integrated with the front end with the help of Kommunicate library which makes the integration easy and user experience seamless. With functional testing, the virtual assistant has proven its efficacy, providing accurate and timely responses to user queries.



Project Member Name	Tickets Assigned	Status	Assigned Date	Completion Date
Harshil Shah	Analysis for Task 1 (authentication) Which option to choose	On going	30-May	6-Jun
	Basic UI creation	On going		
	Continuous Deployment on Netlify	Done		
	Analysis for Task 2	Backlog		
Edwin Adams	Analysis/ Research for Task 7	On going	30-May	6-Jun
Mohit Sojitra	Working on In-game experience: task: Created basic UI in React Analyze the architecture Working on in-game chat feature	On going	30-May	
Khushi Shah	Analysis/Research for task 3 and 10	On going	30-May	6-Jun

Status	What it means
On going	Currently in progress
Backlog	Not started yet
Done	Completed

Figure 99: Sprint Planning Excel

Code commit and Merge Request details with the screenshot is under Spring Plan Section.

Sprint Plan

During the planning phase to launch our task, we had a series of meetings to get organized. We carefully reviewed the project requirements in one meeting and then assigned tasks to team members in the next one. Each person got two specific tasks, and each team member was responsible for choosing the right cloud service (AWS or GCP) and documenting it in a sheet for their assigned task.

Once we knew what we needed to do, we split the work into a two-week sprint cycle. We had regular meetings on weekends to see what tasks were done and what was left. Sometimes, we had extra meetings to handle new requirements or things from other parts of the project. We planned to finish the first part of the project after two sprints, which was about a month. The second part would also take two sprints. The rest of the time was dedicated to putting everything together and making sure we had all the reports we needed for the launch.

Table 22: Sprint Planning

Planned Sprint	**Team Member**	**About to finish**
29th May - 12 June	Khushi	Team Management - Architecture planning and requirement gathering
	Harshil	User Authentication - Architecture planning and requirement gathering
	Mansi	Trivia Game Lobby - Architecture planning and requirement gathering
	Mohit	In-game Experience - Architecture planning and requirement gathering
	Adwin	Content MAnagement - Architecture planning and requirement gathering
12 June - 3 July	Khushi	Team Management - Completion
Extended due to reading week	Harshil	User Authentication - Completion
	Mansi	Trivia Game Lobby - Completion

	Mohit	In-game Experience - Nearly Completed
	Adwin	Content Management - Completed
3 July - 17 July	Khushi	Virtual Assistance
	Harshil	User Profile Management
	Mansi	Question tagging
	Mohit	Ingame Lobby complete & Leaderboard
	Adwin	Notification module
17 July - 31st July	Khushi	Virtual Assistance - complete
	Harshil	User Profile Management - complete
	Mansi	Question tagging - complete
	Mohit	Leaderboard - completed
	Adwin	Notification - completed
31st July - 4th August	Team	Integration and deployment

At the beginning, we made a timeline considering mid-term exams and the features' complexity. The first two parts went well and were completed on time. However, after mid-term, we faced deadline issues of other subjects, and the second feature took longer than expected, finishing on 2nd August instead of the planned 31st July. Luckily, because we communicated well, the integration part was faster, taking only one day instead of the 2-3 days we thought it would take.

Individual experience:

Harshil Shah:

Since the beginning of the project, we divided the features based on individual comfort and expertise. I started my journey by conducting thorough research to determine the best approach

for implementation, considering the right services for each specific problem. Initially, I opted for Amazon Cognito due to its extensive features and control. However, upon further exploration, I found that Firebase Authentication offered easy integration with the application and provided all the necessary details like profile picture URLs, names, and email IDs. Thus, I switched to Firebase Authentication, which required dedicated effort and about a month to successfully integrate Google, Facebook, and email-password logins, overcoming several challenges along the way.

Simultaneously, I worked on setting up a CI/CD pipeline for deploying the application on Netlify. However, later in the process, I discovered the cloud run requirement, which led me to revisit the deployment strategy. With midterms and assignments piling up, my progress temporarily slowed down, but I managed to find time during the end term to focus on my modules.

After completing my module, the team gathered to integrate our individual modules and create the required APIs for seamless communication between the modules. To streamline the deployment process, I developed a run.bat script to build and deploy the frontend to cloud run. Finally, with rigorous testing and deployment, we successfully completed a fully functional project just before the deadline. It was a rewarding experience to witness the project come together cohesively, and we celebrated the successful implementation of the features we had envisioned.

Khushi Shah:

In the beginning, everything was on track and research and analysis of the feature was going as expected, but the logic I wrote previously only allowed me to add a user to one team only. But that was not the requirement so I lagged behind for that module. So, I missed that sprint. Later on when I was trying to figure out that problem and how to solve it, mid-term exams and a lot of assignment submissions came so it slowed me down a bit. But once I was done with the prior submissions, I took responsibility and completed the module which was pending. And started working on the second module feature. I faced some challenges in this module but eventually finished that document.

Mansi:

In the beginning, everything went well for the first two sprints. But when I was creating the question tagging module, it took me more time than I thought. Testing with Comprehend took a lot of time. So, I decided to switch to GCP to finish the work. Because of these delays, I missed the deadline for sprint 3. However, I didn't give up and managed to complete the module within the original timeframe we had planned. Even though there were some challenges, I believe the extra effort I put in will make the question tagging module better in the end.

Mohit:

Initially, I anticipated that the in-game experience module would be straightforward and efficient. However, as the project progressed, I encountered the complexity of maintaining user sessions and synchronizing data across multiple users, leading to unexpected challenges.

Consequently, I had to invest additional time in learning how to establish two-way communication in the cloud, which caused a delay in my progress, and I missed the first sprint milestone.
Despite the initial setbacks, I remained dedicated to the task and managed to complete it within the defined sprint timeline. The integration of user sessions and real-time data synchronization was successfully implemented. Additionally, I faced a challenge when I discovered that LookerStudio lacked support for my credit account, prompting me to switch to AWS QuickSight for creating the leaderboard.
Thankfully, my experience with LookerStudio proved beneficial, as it prepared me well for working with ETL tools and real-time BI tools. This background knowledge enabled a smooth transition to AWS QuickSight, and I was able to complete the leaderboard integration on time. Throughout the process, I learned valuable lessons about cloud-based two-way communication and the versatility of cloud services in data analysis and visualization.

Merge Requests Report

Table 23: Merge Request Details

Title	State	Source Branch	Target Branch	Target Project ID	Author	Author Username	Created At (UTC)
Login Signup	closed	login-signup	master	76362	Harshil Kalpeshkumar Shah	harshils	2023-06-02 20:41:29 UTC
Automated deployment to netlify	merged	ci-cd	master	76362	Harshil Kalpeshkumar Shah	harshils	2023-06-02 20:49:18 UTC
Cd	merged	cd	master	76362	Harshil Kalpeshkumar Shah	harshils	2023-06-02 21:35:11 UTC
Login Signup	merged	login-signup	master	76362	Harshil Kalpeshkumar Shah	harshils	2023-06-02 22:43:26 UTC

converted to typescript	merged	login-signup	master	76362	Harshil Kalpeshkumar Shah	harshils	2023-06-04 15:27:50 UTC
In game window	merged	in-game-window	master	76362	Harshil Kalpeshkumar Shah	harshils	2023-08-02 18:52:09 UTC
Khushi	merged	khushi	master	76362	Khushi Hirenbhai Shah	khshah	2023-08-02 19:42:26 UTC
forget password	merged	login-signup	master	76362	Harshil Kalpeshkumar Shah	harshils	2023-08-03 14:07:52 UTC
Khushi URL working fine	merged	f8_notification	master	76362	Harshil Kalpeshkumar Shah	harshils	2023-08-03 22:04:44 UTC
Mv gamelobby dev	merged	mv-gamelobby-dev	master	76362	Mansi Vaghasiya	mvaghasiya	2023-08-03 23:26:29 UTC
Khushi	merged	khushi	master	76362	Khushi Hirenbhai Shah	khshah	2023-08-04 01:34:22 UTC
Khushi	merged	khushi	master	76362	Khushi Hirenbhai Shah	khshah	2023-08-04 02:05:57 UTC
Minor change	merged	khushi	master	76362	Khushi Hirenbhai Shah	khshah	2023-08-04 02:10:12 UTC
In game window	merged	in-game-window	master	76362	Mohit Rajeshkumar Sojitra	msojitra	2023-08-04 02:19:50 UTC

CHanging routes and adding buttons	merged	khushi	master	76362	Khushi Hirenbhai Shah	khshah	2023-08-04 02:55:22 UTC
Mv gamelobby dev	merged	mv-gamelobby-dev	master	76362	Mansi Vaghasiya	mvaghasiya	2023-08-04 02:58:37 UTC
Khushi	merged	khushi	master	76362	Khushi Hirenbhai Shah	khshah	2023-08-04 13:39:30 UTC

Code Commits Report

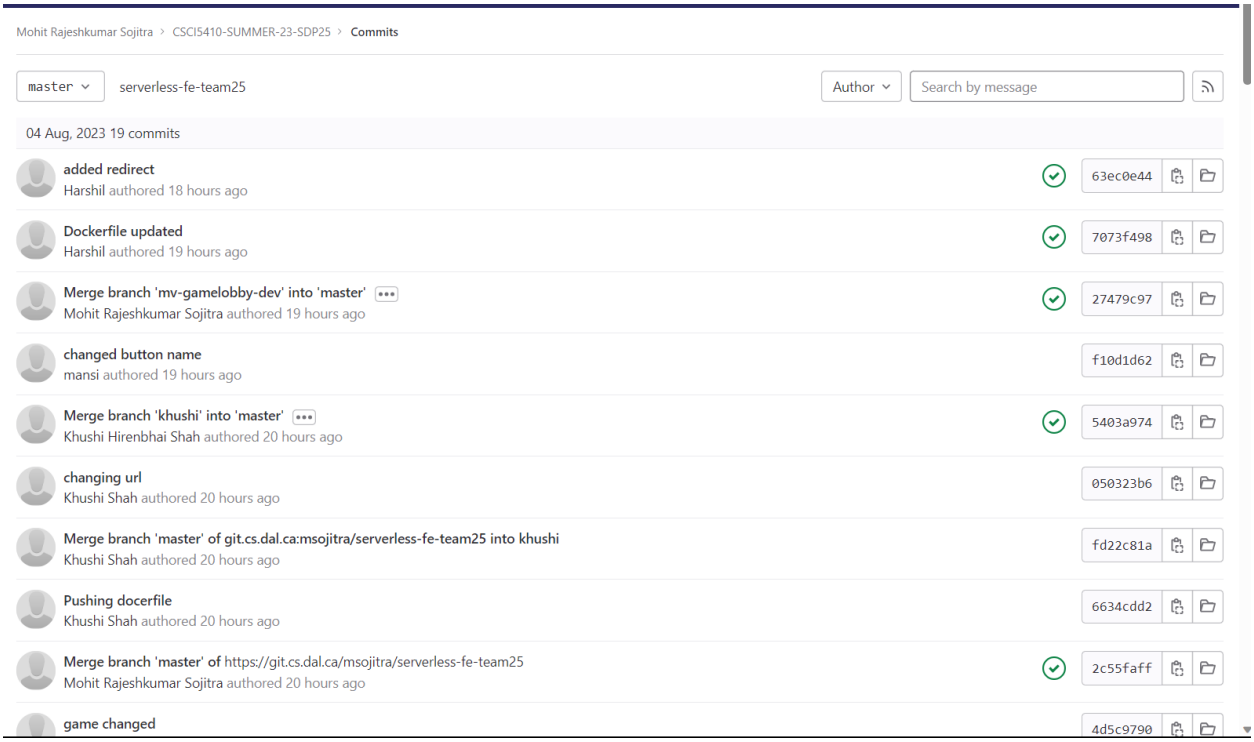


Figure 100: Sprint Commit 1

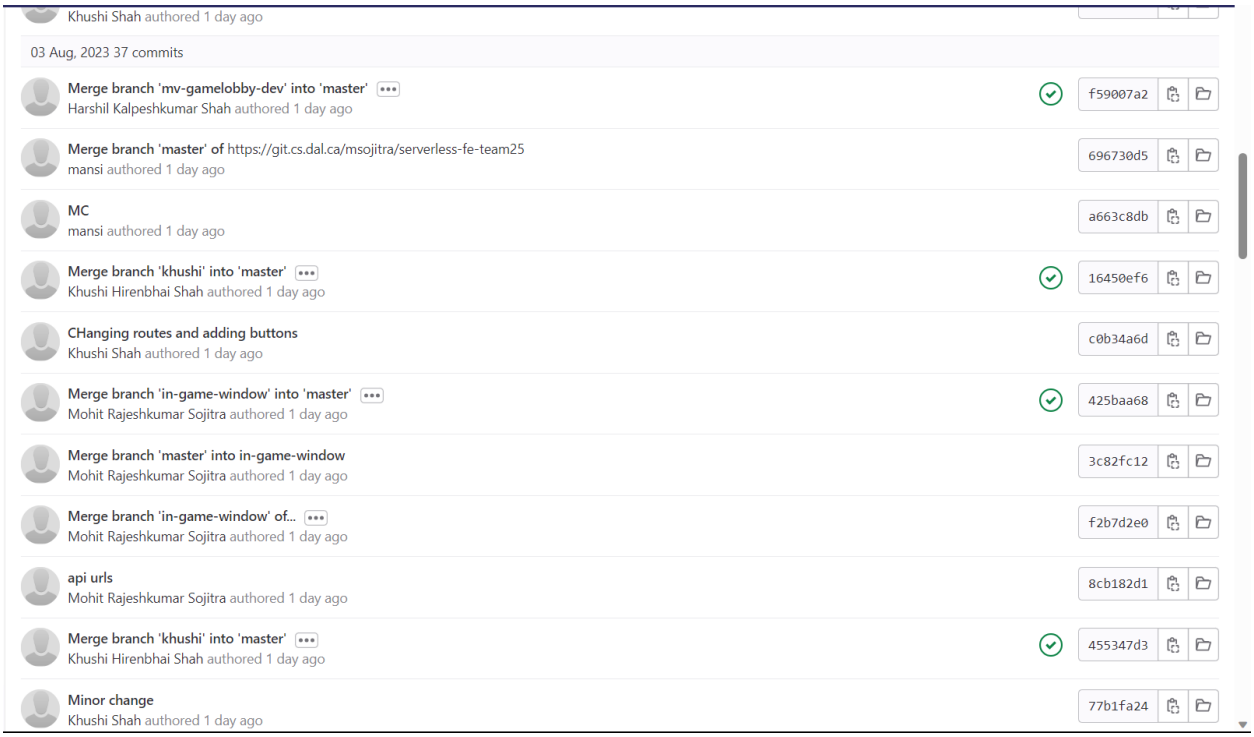


Figure 101: Sprint Commit 2

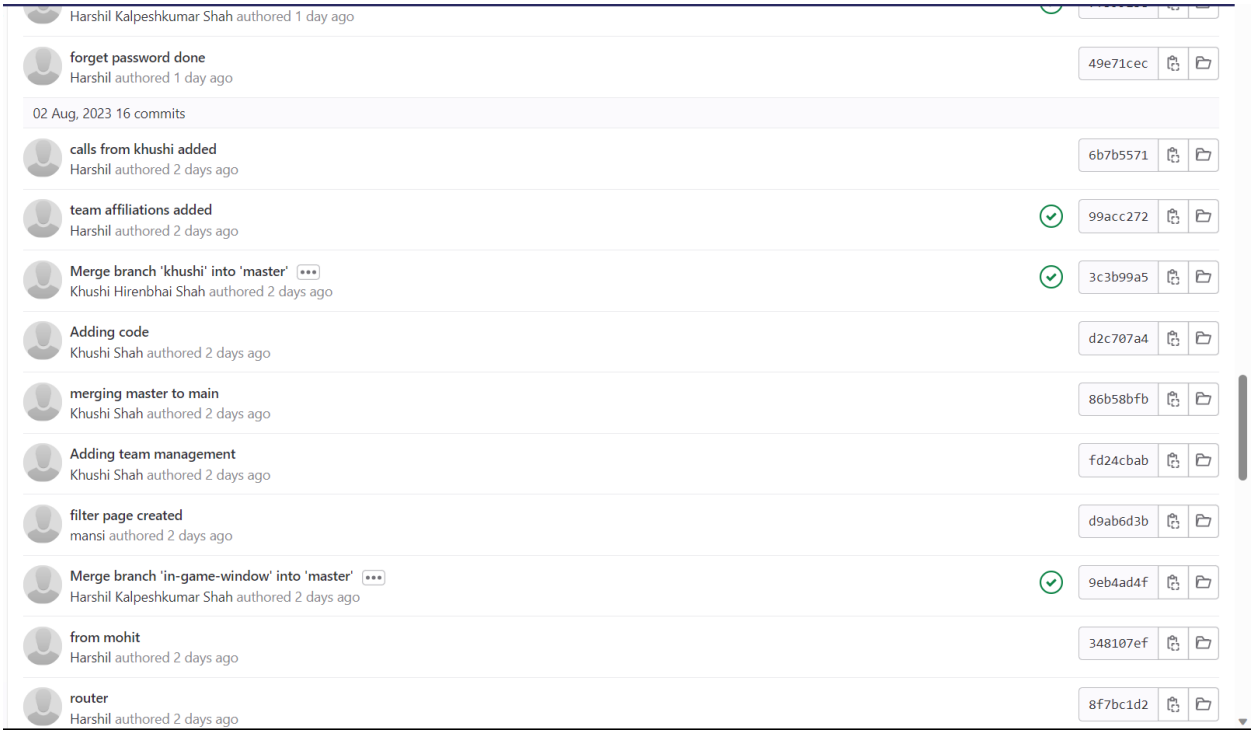


Figure 102: Sprint Commit 3

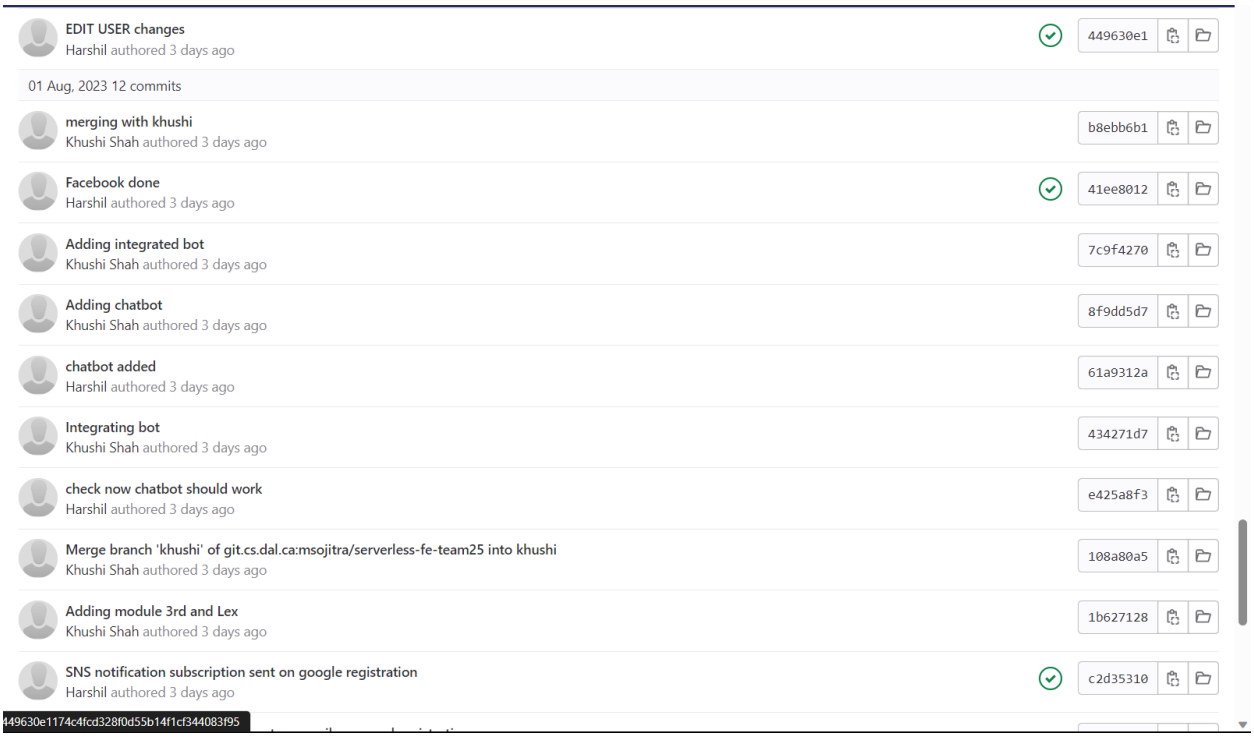


Figure 103: Sprint Commit 4

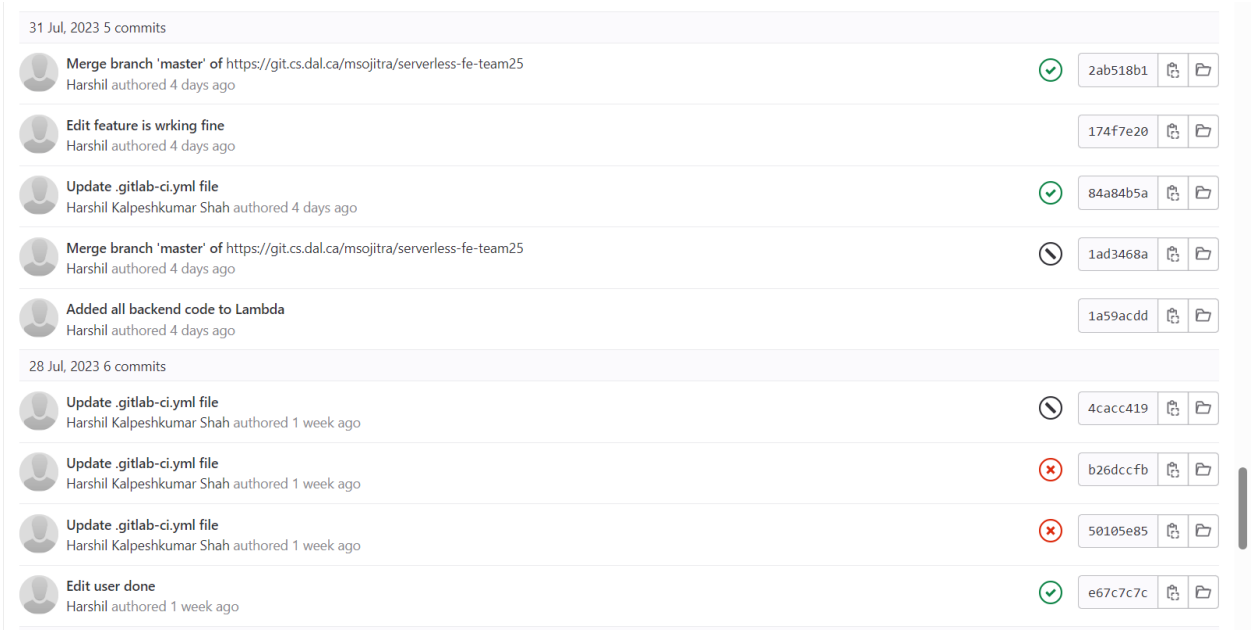


Figure 104: Sprint Commit 5

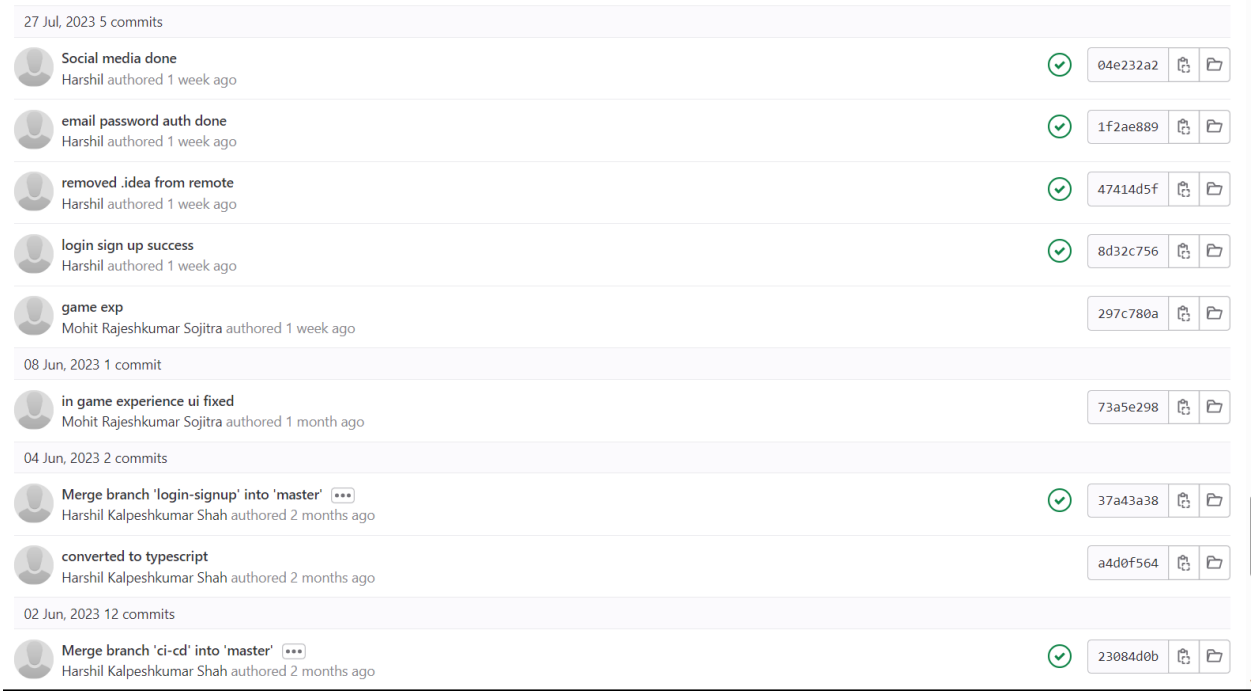


Figure 105: Sprint Commit 6

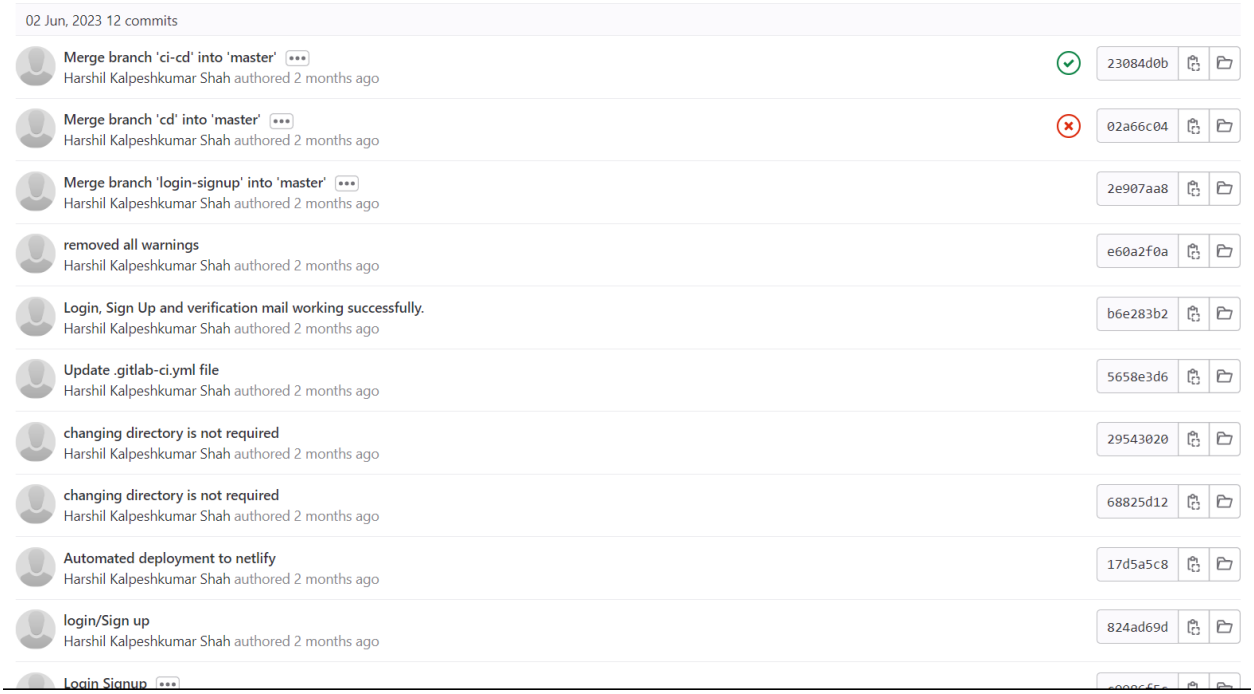


Figure 106: Sprint Commit 7

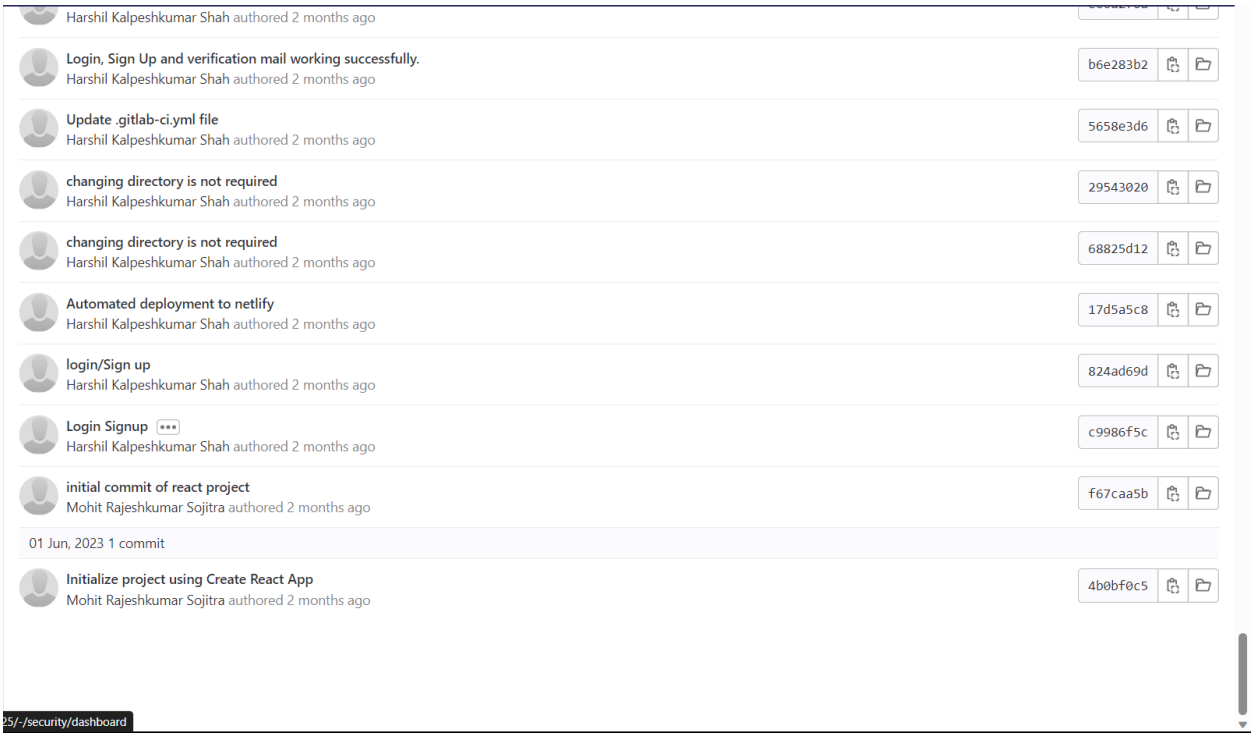


Figure 107: Sprint Commit 8

Meeting Logs

Figure 108: Meeting 1

Figure 109: Meeting 2

Figure 110: Meeting Minutes 1

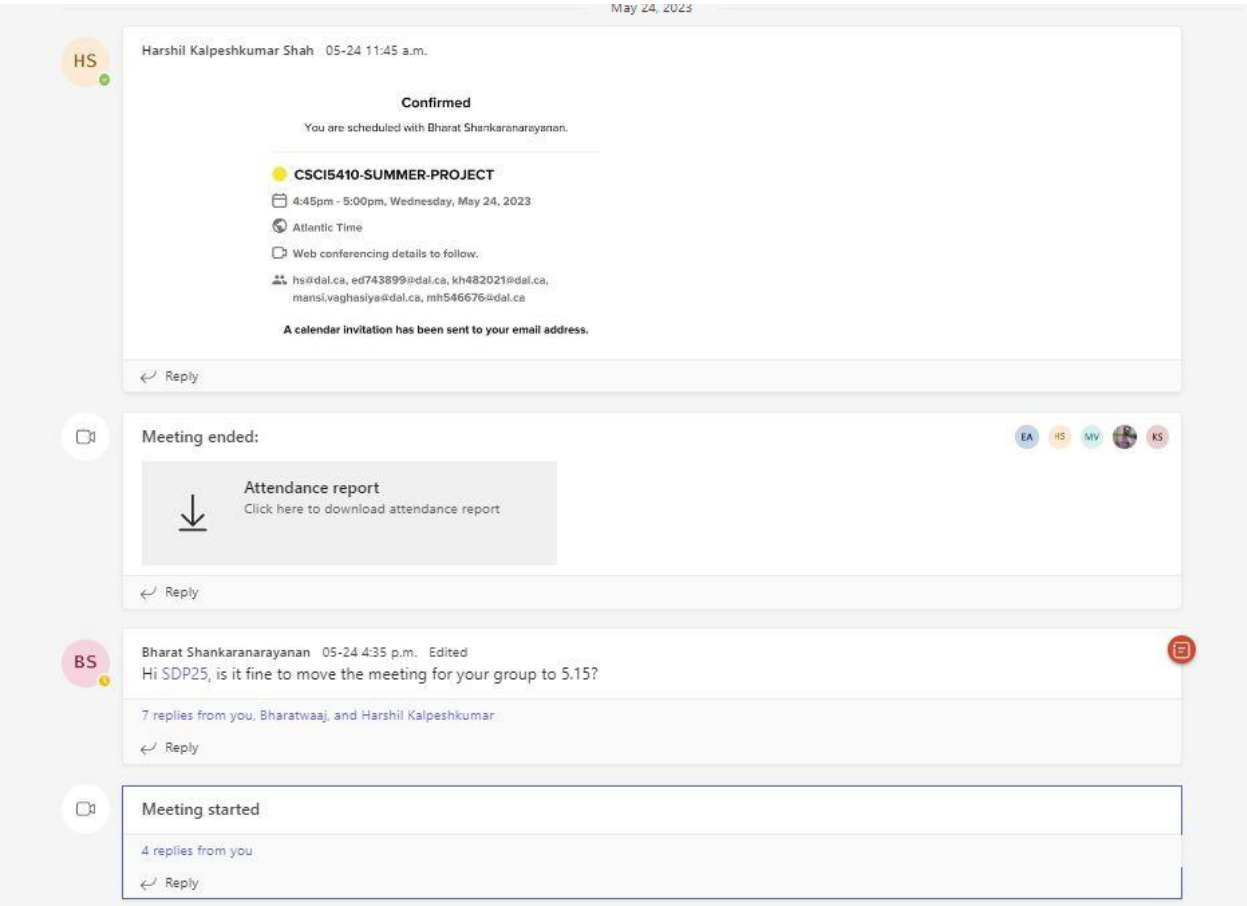


Figure 111: Meeting 3 with meeting minutes

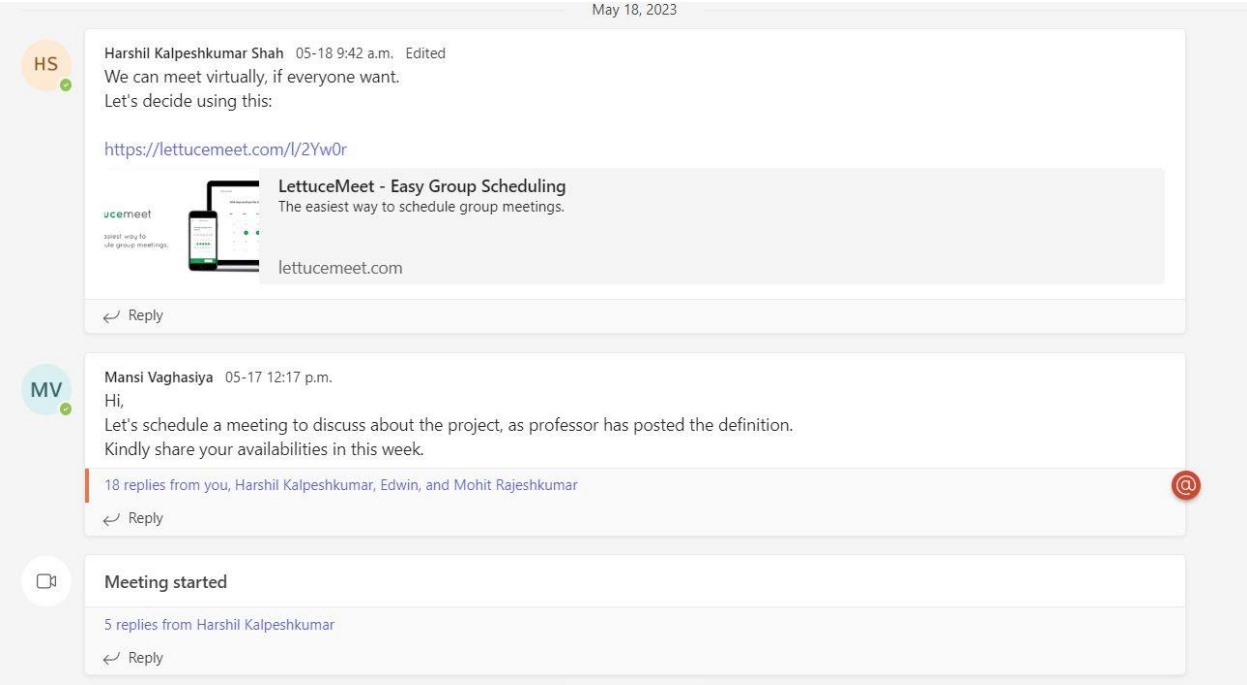


Figure 112: Meeting 4 with meeting minutes

Minutes of Meeting:

Figure 113: MOM 3

May 26, 2023

KS **Khushi Hirenbhai Shah** 05-24 6:40 p.m. Edited

Hi Team,
I have a question.

We all have different GCP and AWS accounts
Now while we create features individually in our own accounts, how will we share code with each other?

See less

Collapse all

HS **Harshil Kalpeshkumar Shah** 05-24 9:54 p.m. Edited

We don't have to share the code, I guess
For GCP we can use something like IAM to create users to collaborate
For AWS we have to rely on the person who is implementing the feature and have to use the public url/API gateway
I am not 100% sure

Mohit Sojitra 05-26 10:43 p.m.

If we use IAM users anyway, the credit of only root account will get deducted.

Reply

Figure 114: MOM 4

Screenshots of the meeting

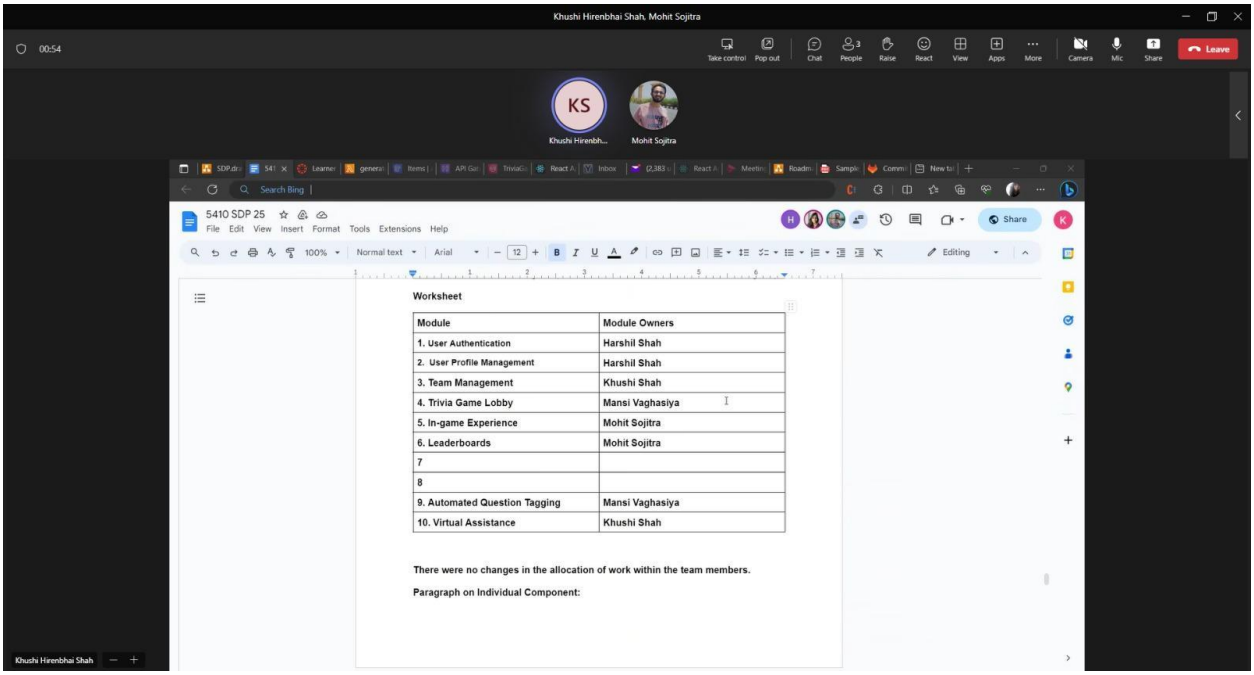


Figure 115: Meeting 1

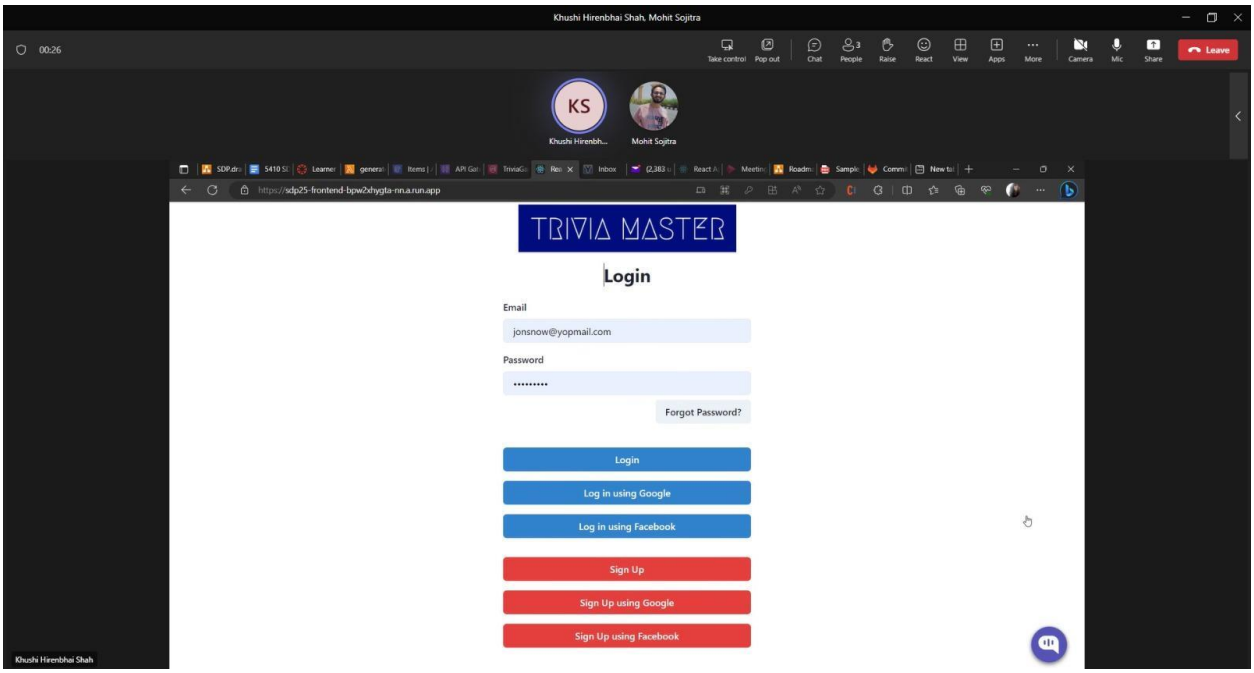


Figure 116: Meeting 2

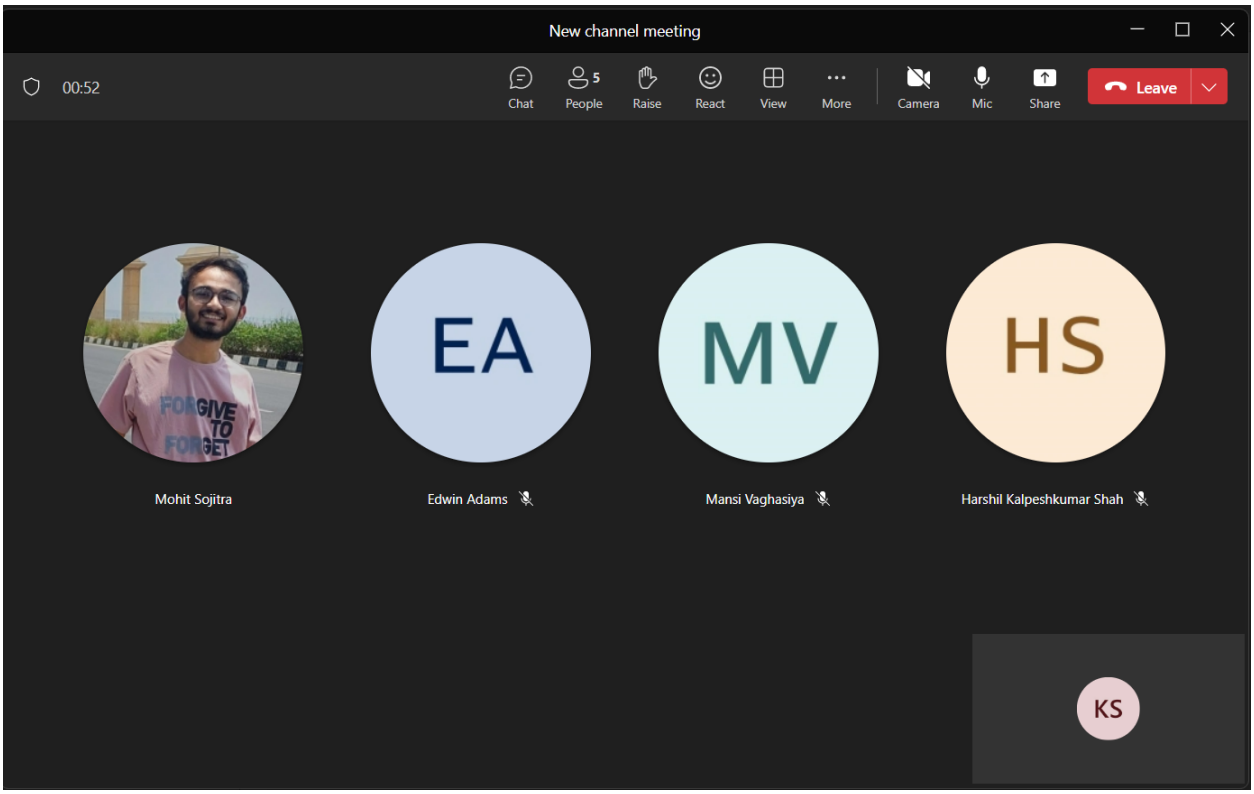


Figure 117: Meeting 3

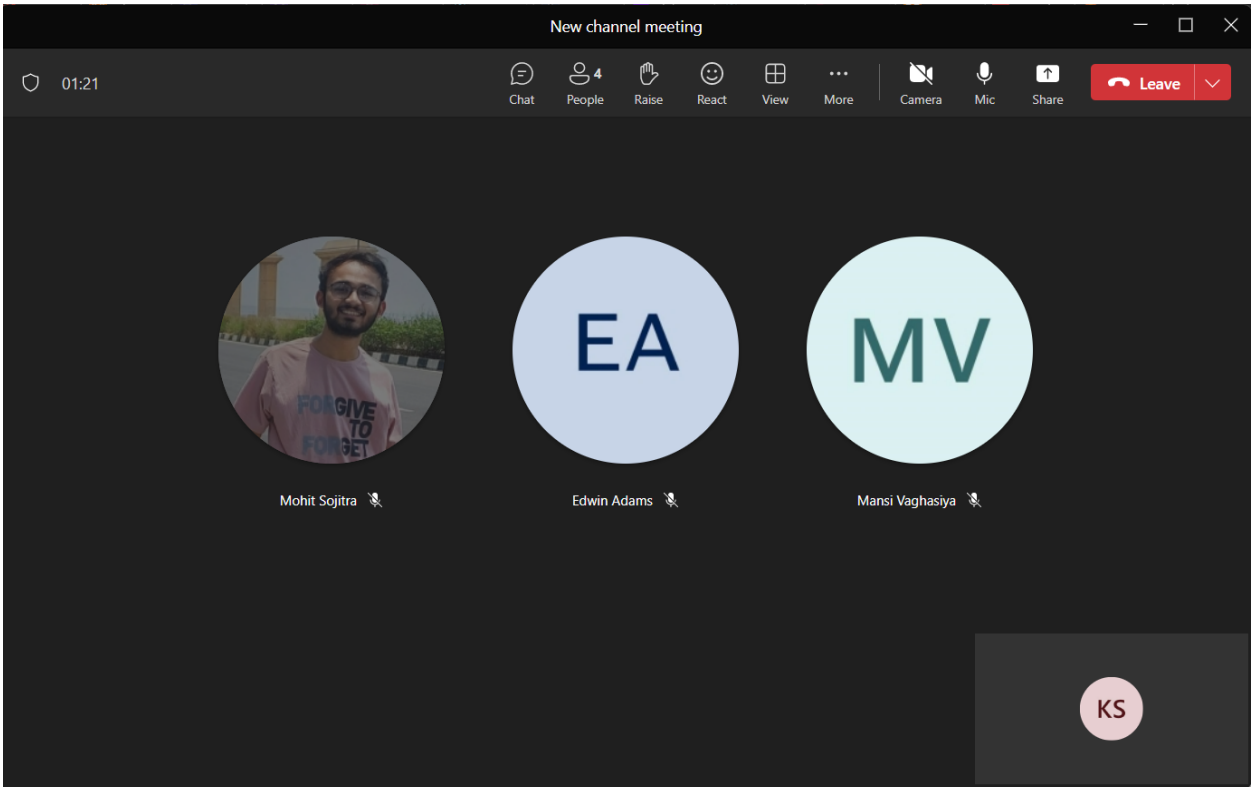


Figure 118: Meeting 4

Individual Contribution

Harshil Shah

Features built: User Authentication, User Management
For both the features, there are no deviations. The final output is the same as planned for requirements.

Khushi Shah

Features built: Team management, Virtual Assistant
For both the features, there are no deviations. The final output is the same as requirements and expectations.

Mohit Sojitra

Features built: Ingame experience, leaderboard
For both the features, there are no deviations. The final output is the same as requirements and expectations.

Mansi Vaghasiya

Features built: Game Lobby, Question tagging
For both the features, there are no deviations. The final output is the same as requirements and expectations.
Edwin Adams

Novelty

Harshil Shah

The novel aspect of my features lies in its independence and microservices architecture, which enables each feature to function autonomously.
Utilizing stateless communications with APIs, the modules are seamlessly integrated, allowing smooth interactions between them.
What sets this architecture apart is its multi-cloud multi-account approach, where each feature is deployed individually on distinct AWS accounts, yet collaborates harmoniously as a unified system.
This collaboration is facilitated through robust authentication methods, implemented via API Gateways, ensuring secure and efficient data exchange.

Khushi Shah

Code comments are present for both the modules, Team management and Virtual Assistance.
For virtual assistance, the users can create multiple chats and can save them and can refer to them later on.
For a smoother experience of the virtual assistance, I have created the new AWS free tier account which gives me more permissions and makes the assistance experience seamless. But my database was present in the AWS Academy provided by the course. So to integrate both the things together I created an api in a new account which calls another account and fetches the data from the DB and returns the result.

Mohit Sojitra

Used serverless application development framework through the development to protect the architecture from accidental loss.
For a smoother experience of the virtual assistance, I have created the new AWS free tier account which gives me more permissions and makes the assistance experience seamless.

Mansi Vaghasiya

Make is commented thoroughly in the development.
For exploring the possibilities with comprehend, I created a free tier account.
In game lobby, all the user will be redirected to their In game experience after start time of the game approaches
Edmin Adams

Cloud Run Hosted URL

[React App (sdp25-frontend-bpw2xhygta-nn.a.run.app)]

References

[1] "Amazon Cognito," Amazon, 2023. [Online]. Available: https://aws.amazon.com/cognito/.

[2] "Amazon Lambda," Amazon, 2023. [Online]. Available: https://aws.amazon.com/lambda/.

[3] "Chatbot run lambda function remotely tutorial," Amazon, 2023. [Online]. Available: https://docs.aws.amazon.com/chatbot/latest/adminguide/chatbot-run-lambda-function-remotely-tutorial.html.

[4] "Google Sheets: Online Spreadsheet Editor," Google.ca, 2023. [Online]. Available: https://www.google.ca/sheets/about/.

[5] "Create buckets," Google Cloud, 2023. [Online]. Available: https://cloud.google.com/storage/docs/creating-buckets.

[6] "Looker Studio Overview," Google.com, 2023. [Online]. Available: https://lookerstudio.google.com/u/0/navigation/reporting.

[7] "How to connect Google Cloud Functions to Google Sheets," Actiondesk.io, 2023. [Online]. Available: https://www.actiondesk.io/google-sheets-integration/google-cloud-functions.

[8] "Firebase authentication," Firebase, 2023. [Online]. Available: https://firebase.google.com/docs/auth.

[9] "Develop Chatbot using Amazon Lex and integration with React JS web application," Medium, 2021. [Online]. Available: https://medium.com/@ideepaksharma/develop-and-integrate-amazon-lex-chatbot-into-website-47f50fe4cfed.

[10] "Flowchart maker and online diagram software," Diagrams.net, 2023. [Online]. Available: https://app.diagrams.net/.

[11] "Amazon API Gateway," Amazon, 2023. [Online]. Available: https://aws.amazon.com/api-gateway/.

[12] "AWS QuickSight," Amazon, 2023. [Online]. Available: https://aws.amazon.com/quicksight/.

NOTE: We have added the access date in the word citation but it is not showing up here