Software Engineering Project

Milestone 06

30 March 2025

Team 29

Ajay Thiagarajan (21f1003242)

A J R Vasu (21f3002975)

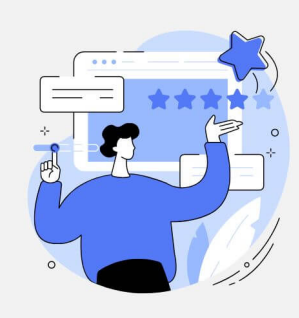
Anand K Iyer (21f1001185)

Anuj Gupta (21f3001598)

Ghanashyam R (21f1003387)

Jalaj Trivedi (21f2000730)

Niraj Kumar (21f1006589)



Milestone 6 - Project Submission

Acknowledgements

We, Team 29 of the Software Engineering Project (January Term), extend our gratitude to everyone who contributed to our journey throughout this project.

We sincerely appreciate our IIT Madras and the Software Engineering course team, professors, mentors and clients their support and guidance.

And finally thank you to team 29 for coming this far in the journey collectively !

Meet The Team!

About the Project

Integrating Generative AI into the SEEK Learning Portal

With the rapid advancement of generative AI (GenAI) technologies, there are exciting opportunities to enhance learning experiences by integrating GenAI into self-paced platforms like the SEEK portal. The SEEK portal serves as a self-learning environment, offering lecture videos, resource materials (transcripts), programming quizzes, and links to discussion forums where students can post subject-specific queries.

The main idea is to find out how GenAI can be added to SEEK to make learning better and more interactive.

What SEEK Offers

i) Learning
- Video lectures that students can watch at their own pace.
- Resource materials and transcripts to help understand the topics better.
- Programming quizzes and graded assignments to test knowledge.

ii) Coding
- A built-in code editor where students can write code and instantly see the output.
- Students can also submit both practice and graded assignments to get their work checked.

How GenAI Can Help

Chatbot Support
A GenAI chatbot can answer students' questions about course content, videos, or any doubts they have while learning.

Code Help
While working on programming problems, GenAI can guide students with tips on syntax, good coding practices, and fixing errors.

Feedback
After submitting code, GenAI can suggest better or more efficient versions of the code and explain how they work.

Milestone 1

Objective

The objective of Milestone 1 was to systematically identify and define the key user groups of the SEEK portal—Primary (Students), Secondary (Faculty), and Tertiary (Admins, Developers, ML Research/Ops Teams)—and outline their specific requirements. This milestone aimed to enhance the existing system by integrating Gen AI-powered features that improve user experience, academic engagement, and administrative efficiency.

A crucial focus was placed on writing user stories following the SMART (Specific, Measurable, Achievable, Relevant, and Time-bound) guidelines. These user stories highlighted the AI-driven enhancements that will be incorporated into the SEEK portal, ensuring that students receive personalized learning support, faculty gain data-driven insights, and administrators have better engagement tracking tools.

1) USER TYPES

We have 3 user types for this application and a brief description is given below.

i) Primary Users

Students are the primary user for this application. They use the SEEK portal for accessing academic material and guidance as part of their academic program.

The Gen AI based enhancements will primarily focus on the requirements of the students as they are the key stakeholders and beneficiaries of the academic program.

ii) Secondary Users

Faculty which comprises teaching assistants and instructors.

They will be responsible for managing the study material, assignments, organizing and taking the live session and interacting with students to clarify their doubts.

iii) Tertiary Users

Institution's Administrators (admin), the team responsible for maintaining data assets on SEEK, managing study material, assignments, monitoring portal's usage and managing users' access to SEEK.

Developers, the team which will maintain/ develop the code based and maintain the SEEK infrastructure.

Assumptions

Before diving into the user requirements and stories, below are the list of assumptions we have made. Based on these assumptions we will omit some of the obvious requirements, which we assume are already part of the portal. Consequently, the user stories focused more the AI agentic enhancements.
Further, we are focusing more on the three users' groups, namely, students, faculty and the admin team.

1. Students can browse through their,
   a. Weekly content such as slides and lecture videos
   b. They can view and respond to their assignments including programming assignments
   c. They can access their profile page and view details

2. The faculty team can,
   a. Upload/ view weekly content

3. The admin team,
   a. Can upload/delete content
   b. They have privileges to add/ remove access to students and faculty members
   c. They can view activities by students and faculty

2. User Requirements

2.1 Students

- Students should be able to log in using their institute-provided credentials to simplify the sign-in process and ensure secure authentication.
- Over and above accessing study material, students should have access to a searchable knowledge graph to retrieve relevant course materials and additional resources efficiently for their topics of interest.

- Students should be able to extract concise summaries from multiple sources related to their query topics.
- Enable students to save summarization results as formatted PDF files for offline access and sharing.
- Students should get intelligent code suggestions, including error corrections and debugging tips, to help them improve their programming skills.
- Students should receive reminders for upcoming assignment deadlines, quizzes, and other academic events.
- Students should be able to view and mark tasks and milestones as completed, including deadlines and calendar views.

2.2 Faculty

- Over and above uploading content, provide faculty with analytics to identify the most queried topics by students over the last week, helping them address knowledge gaps in live sessions.
- Provide faculty with data on the most error-prone programming questions over the last week, enabling them to focus on these issues in live programming sessions.
- Provide a dashboard that gives an overview of student progress, including queried topics, engagement metrics, and milestones.
- Suggest live session content to the faculty, based on student queries and programming errors, making sessions more relevant to the students' needs.

2.3 Admin

- Implement clear role-based access for students, faculty, and admins, ensuring secure and appropriate access to system components.
- Provide admins with a dashboard to track student engagement, including metrics like resource usage, query trends, and attendance in sessions.
- Enable admins to send automated reminders and nudges to students based on engagement metrics to improve their academic involvement.

3. User stories

STUDENTS

1. Feature: Single Sign On

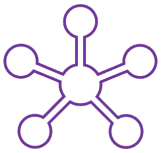
As a student,

I want a simplified sign in process which is integrated with my institute ID so that accessibility is easier.

2. Feature: Knowledge Graph

As a student,

I should be directed to relevant course materials and additional learning resources when searching for a topic of interest, so that I can find information quickly and efficiently.



3. Feature: Summarization

As a student,

I want to extract a summary response from across sources for my query topic, so that I have an overview of the topic.

4. Feature: Notes

As a student, I want to save a formatted PDF copy of the summarization response, so that I can access this information anytime later and share with fellow students.

5. Feature: Programming and Debugging

As a student,

I want suggestions on how to correct my code for my programming assignments and practice, so that I can improve my coding skills.

6. Feature: Smart Reminders

As a student,

I want reminders about upcoming assignments and deadlines, so that I do not miss any deadline.

7. Feature: Quiz Master

As a student,

I want to get a summary of the questions and the answers I queried in that term for a particular course so that I can revise the hot topics for the quiz.

FACULTY

8. Feature: Knowledge Gaps

As faculty,

I should be able to access the most queried topics (top 10, over the last one week) so that I can address these issues in live sessions.

9. Feature: Coding Challenges

As faculty,

I should be able to identify the most error-prone programming questions (top 5, over the last one week) so that I can address these issues in live programming sessions.

ADMIN

10. Feature: Engagement Metrics

As the admin,

I should see students' engagement metrics so that reminders can be sent to them to nudge them towards better involvement.

Conclusion

Milestone 1 successfully identifies all user roles and clearly defines their functionalities and requirements within the SEEK portal. The user stories ensure that the AI-driven features align with user needs while maintaining SMART principles. These stories emphasize student-centric enhancements like knowledge graphs, AI-driven summarization, intelligent coding assistance, and automated reminders, alongside faculty analytics and admin engagement tools.

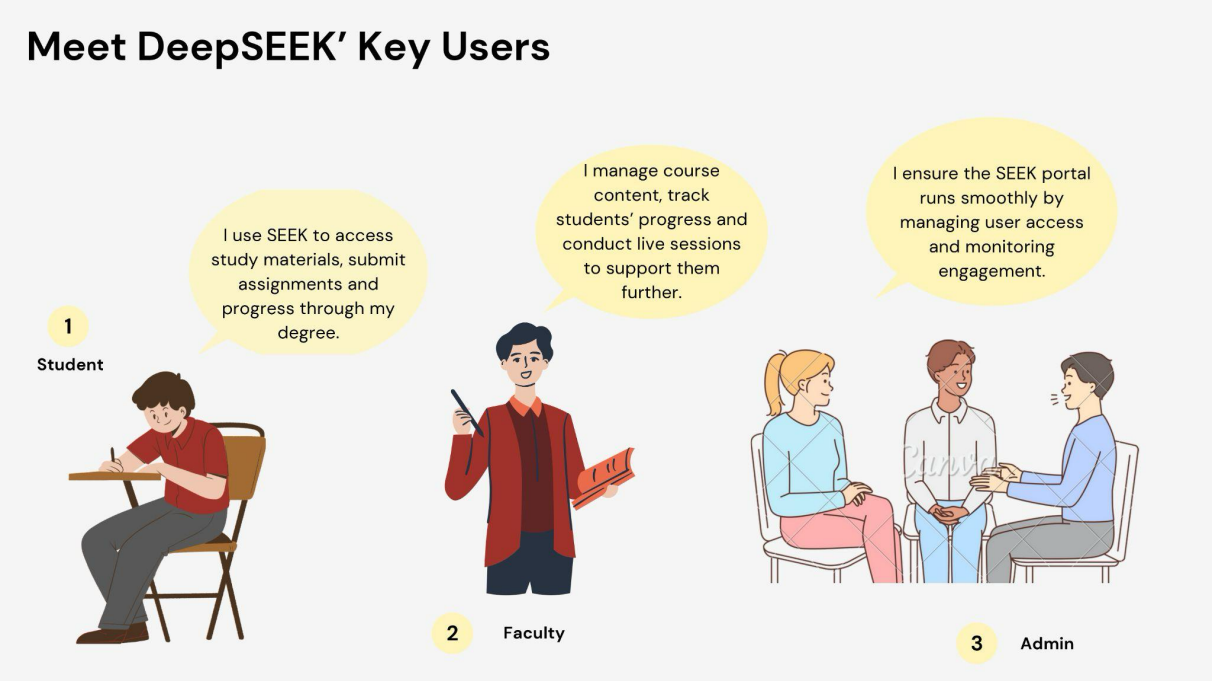
By following a structured approach, this milestone lays the groundwork for the implementation phase, where these features will be integrated into the existing system. The next steps involve refining these user stories based on peer evaluations and ensuring alignment with development priorities before moving toward implementation.

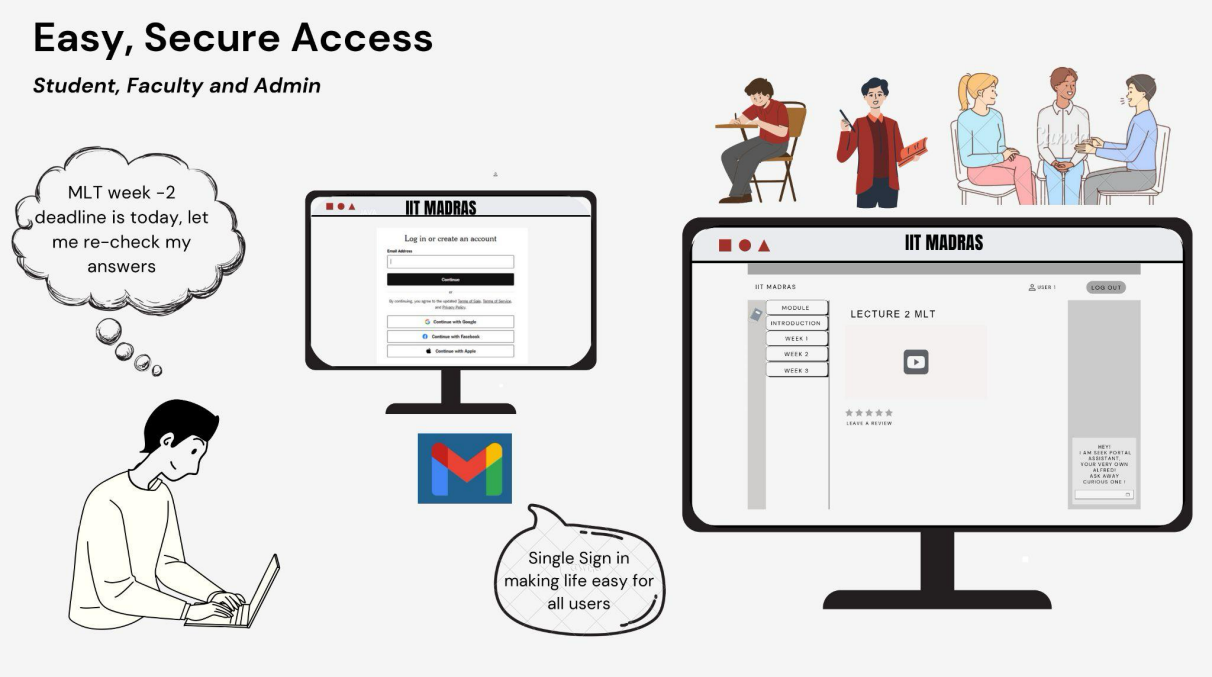
Milestone 2

Objective

The goal of Milestone 2 was to design the user interface (UI) of the AI Agent for Academic Guidance. This involved creating a storyboard to visually represent user interactions and developing low-fidelity wireframes for each identified user story. The UI design will follow usability heuristics and best practices to ensure an intuitive, accessible, and user-friendly experience. These wireframes serve as the foundation for the final UI, ensuring that user requirements are effectively translated into a structured interface.

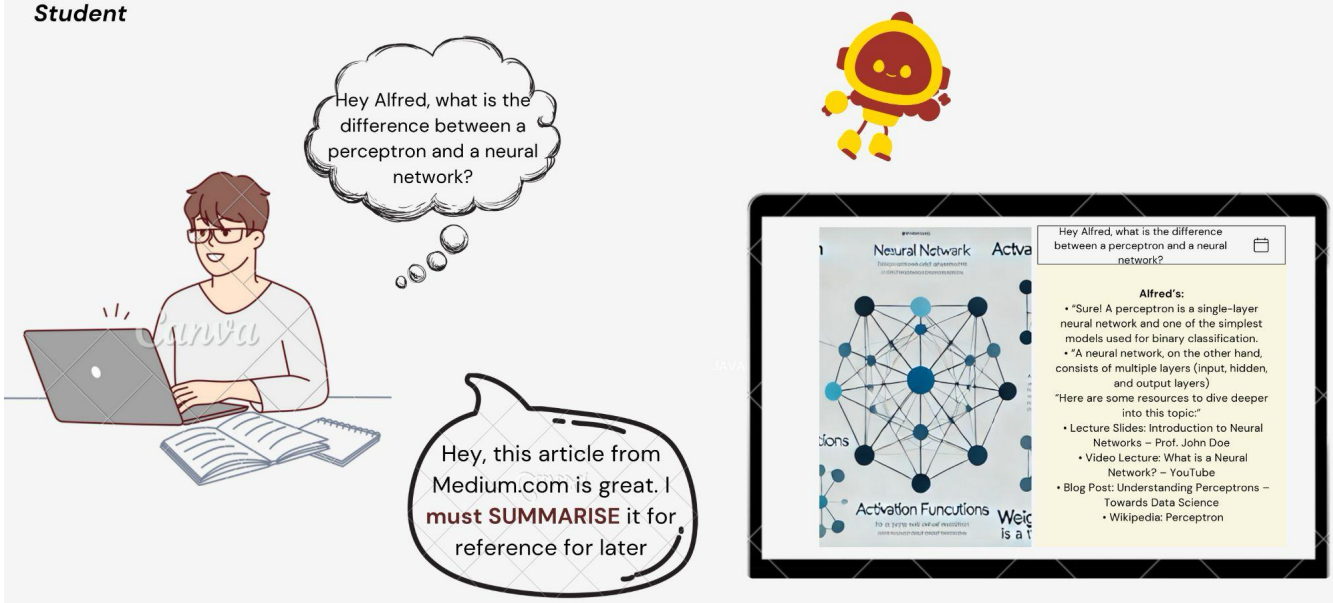
Storyboards





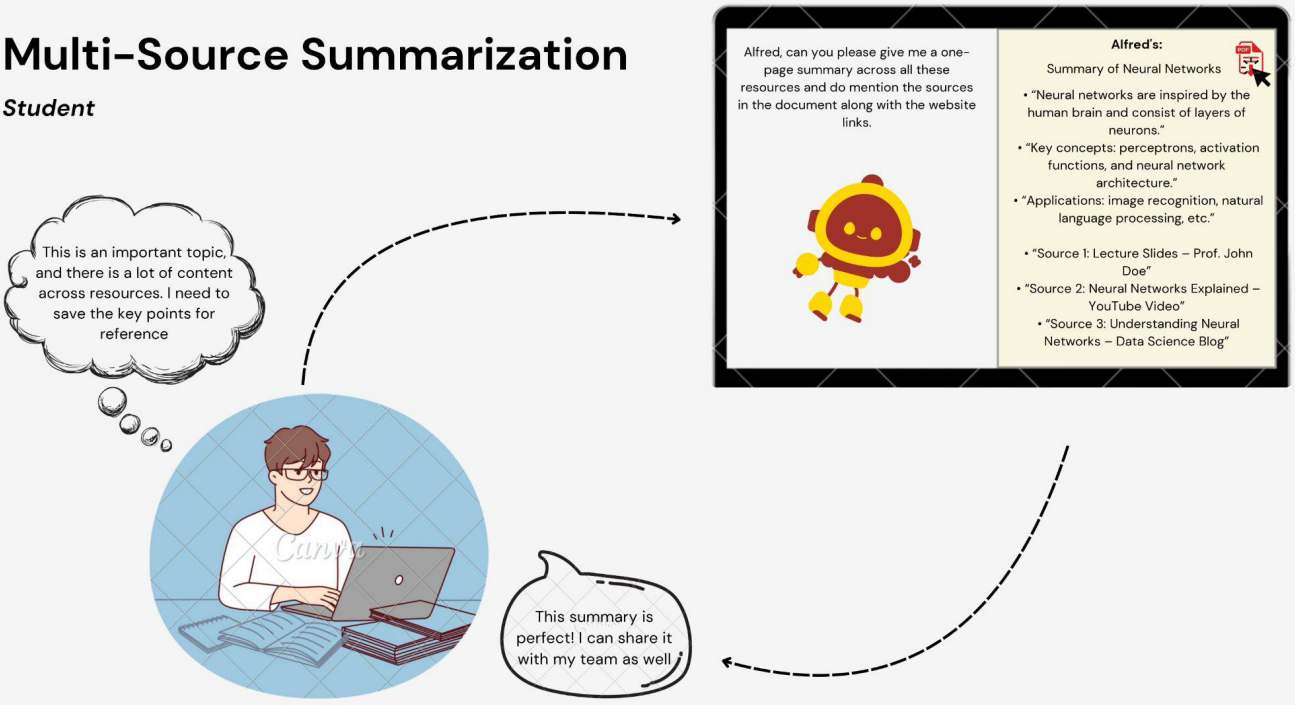
Knowledge Graph for a consolidate view

Student

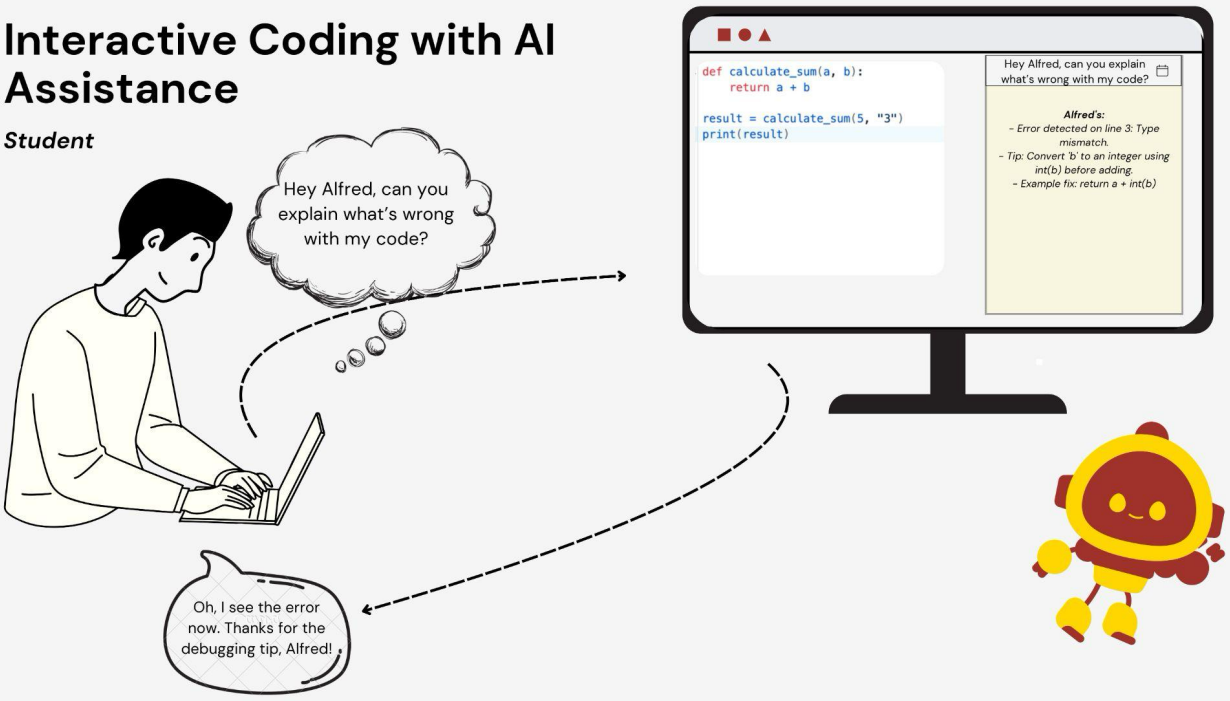


Multi-Source Summarization

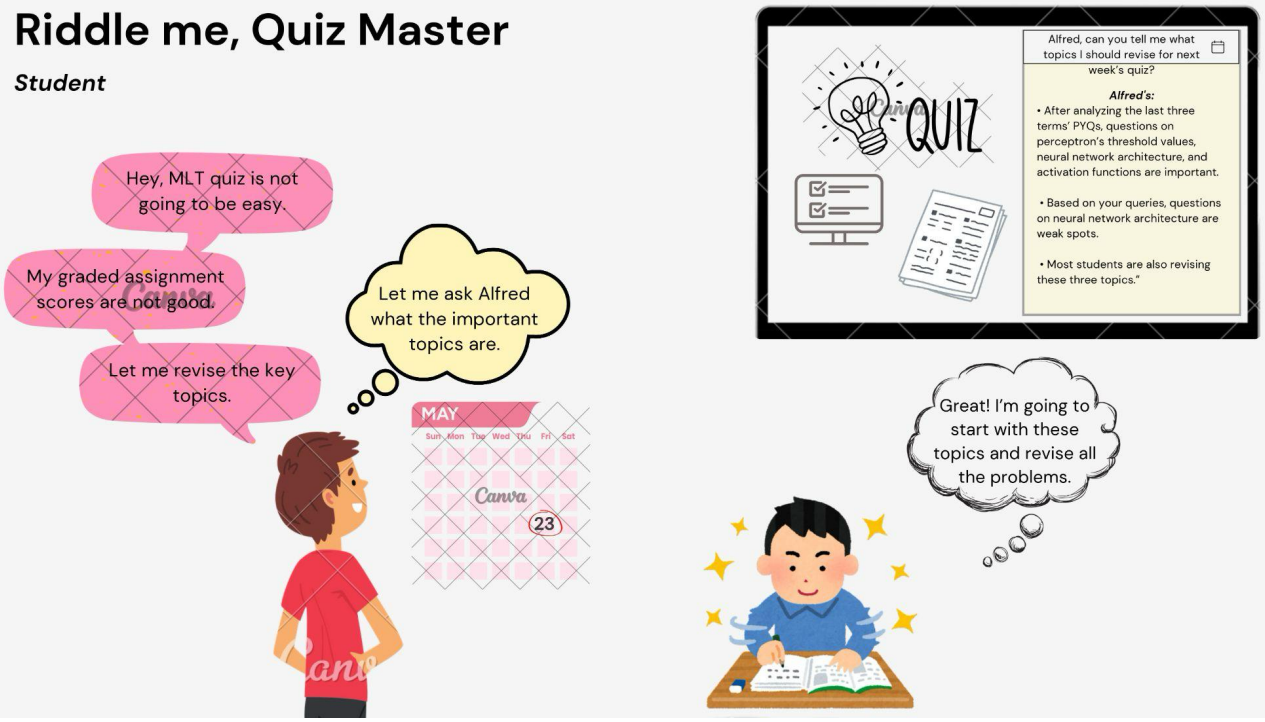
Student



Interactive Coding with AI Assistance

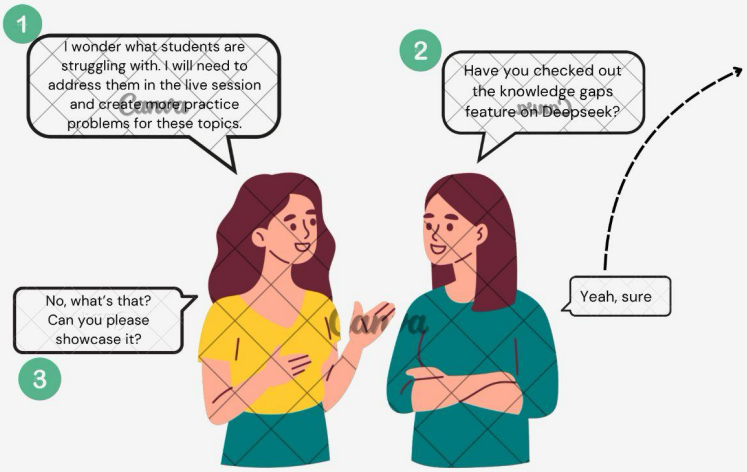


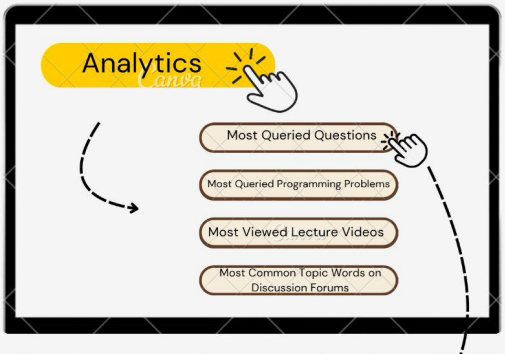
Riddle me, Quiz Master

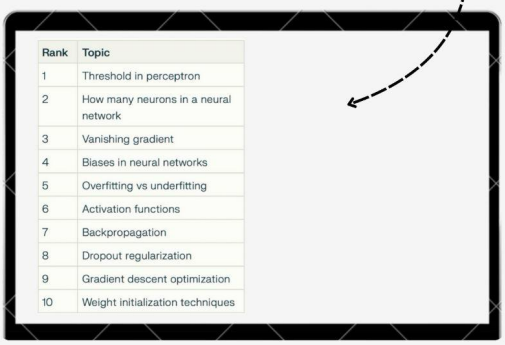


Which Knowledge Gaps to plug?

Faculty



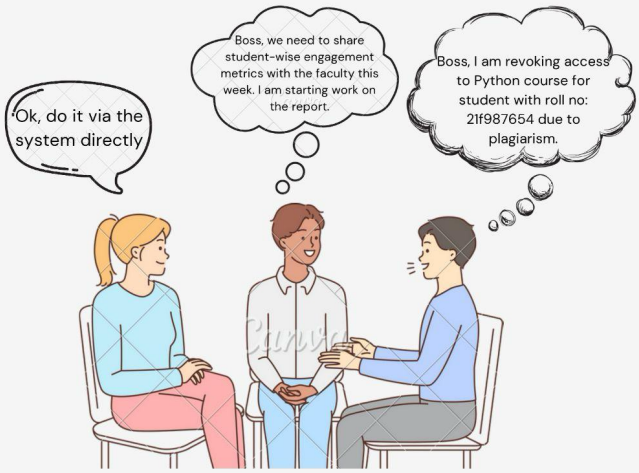




Rank	Topic
1	Threshold in perceptron
2	How many neurons in a neural network
3	Vanishing gradient
4	Biases in neural networks
5	Overfitting vs underfitting
6	Activation functions
7	Backpropagation
8	Dropout regularization
9	Gradient descent optimization
10	Weight initialization techniques

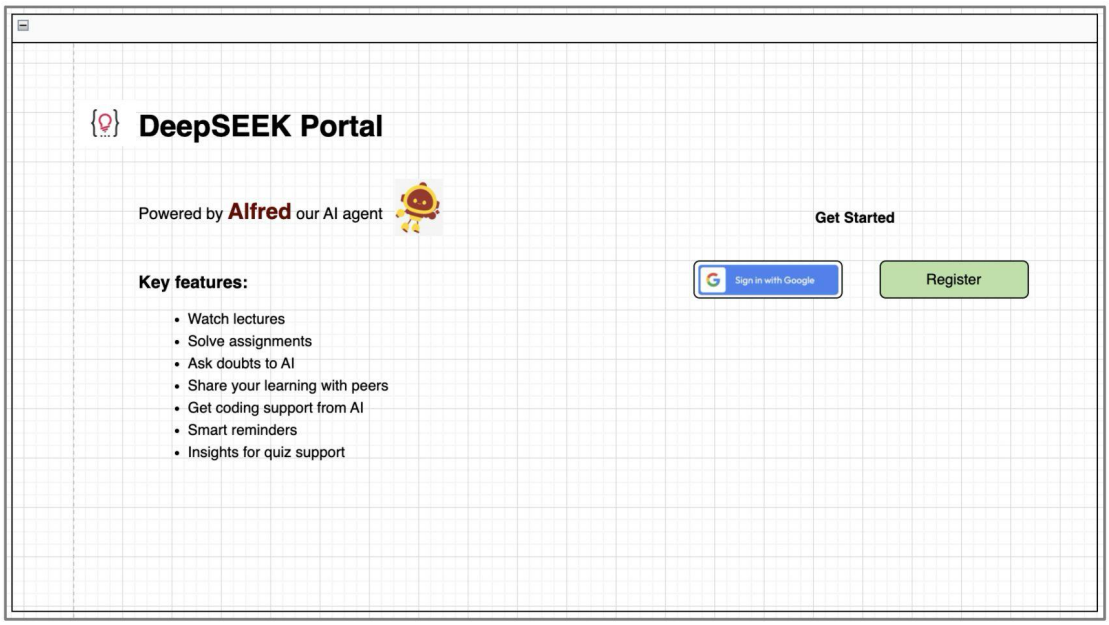
Engagement Metrics

Admin

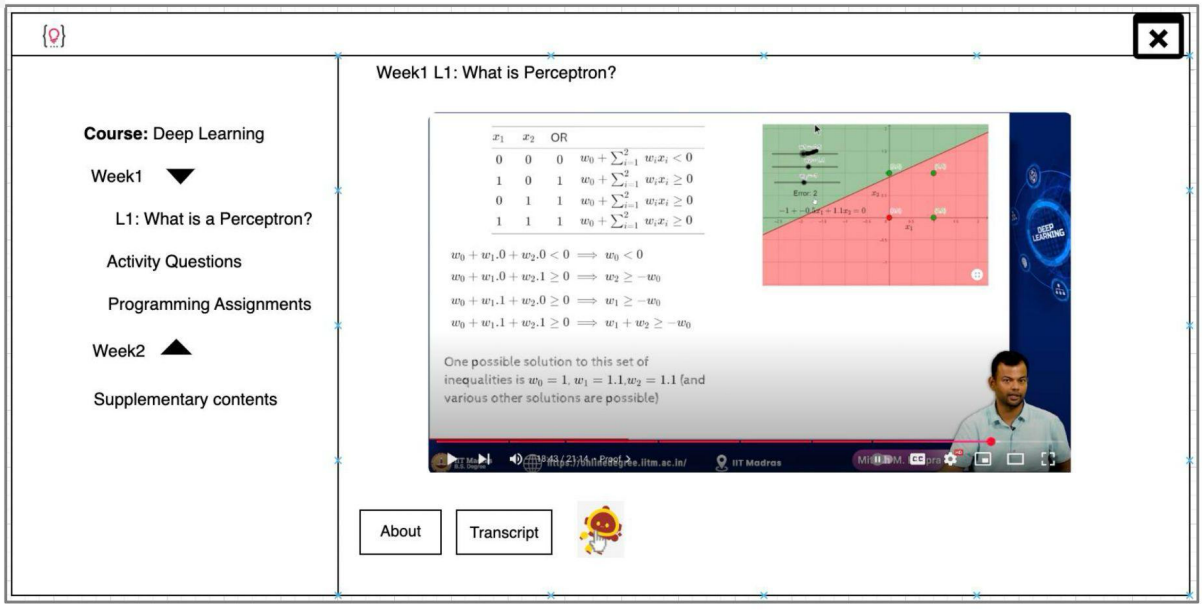


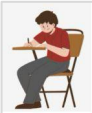
Wireframes

Single sign-on enabled registration and access

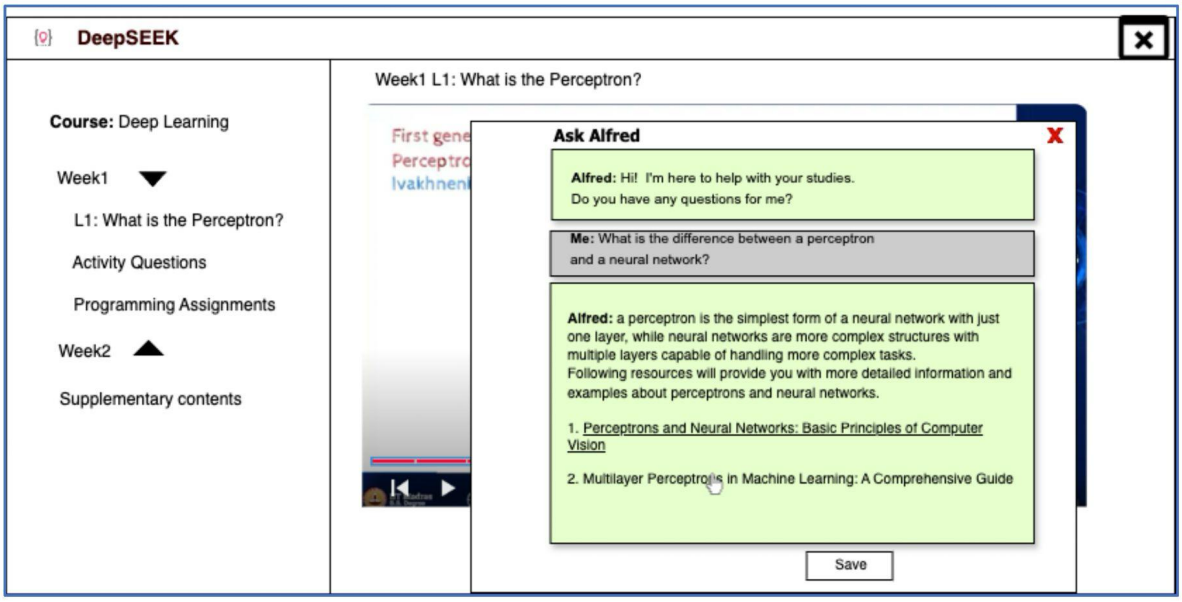


Resource page for accessing content by the student

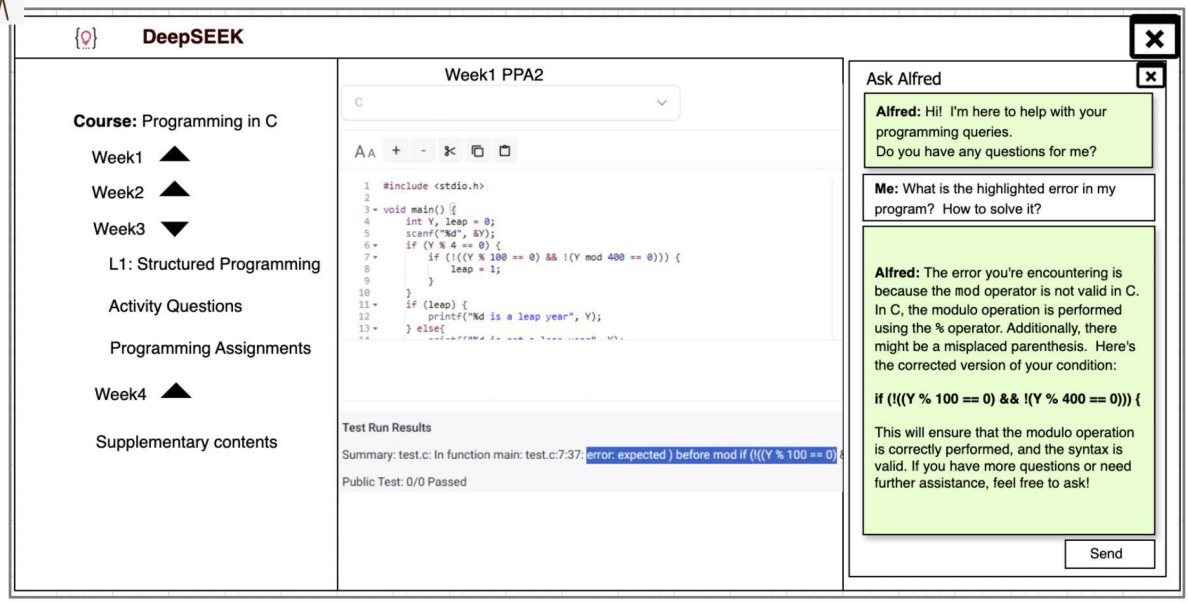




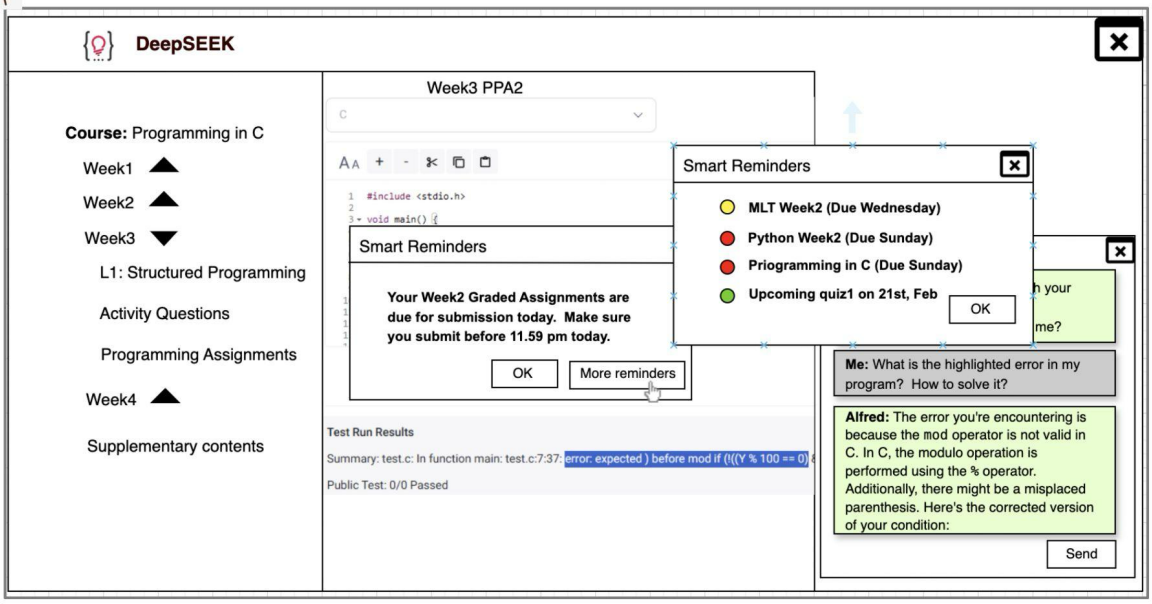
Question and answer, knowledge graph from AI chatbot - Alfred



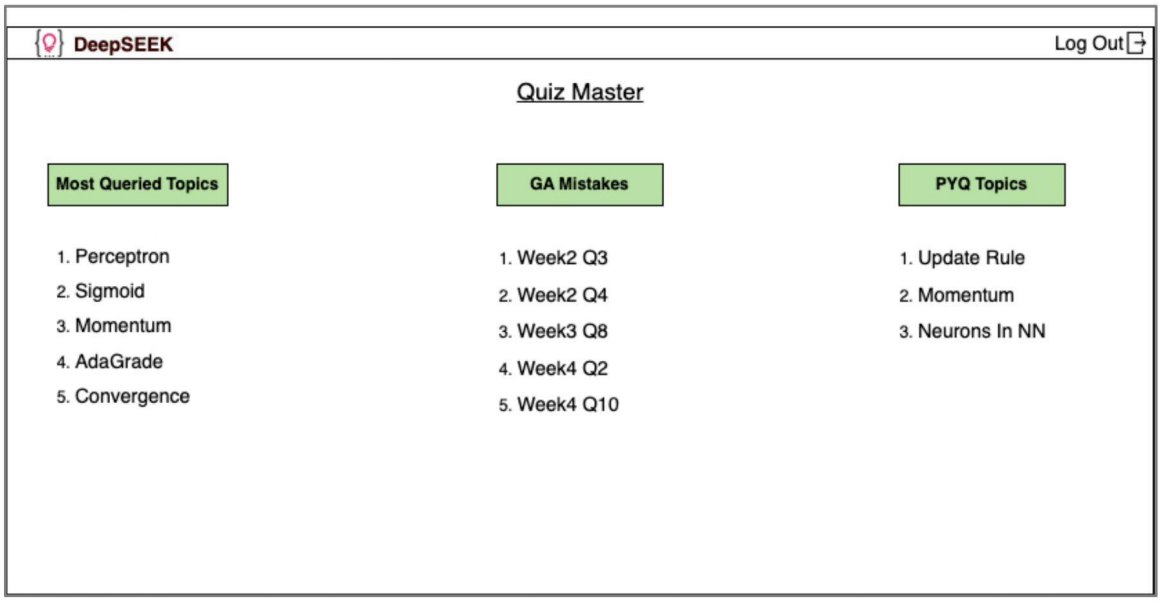
Interactive coding and support from AI chatbot - Alfred



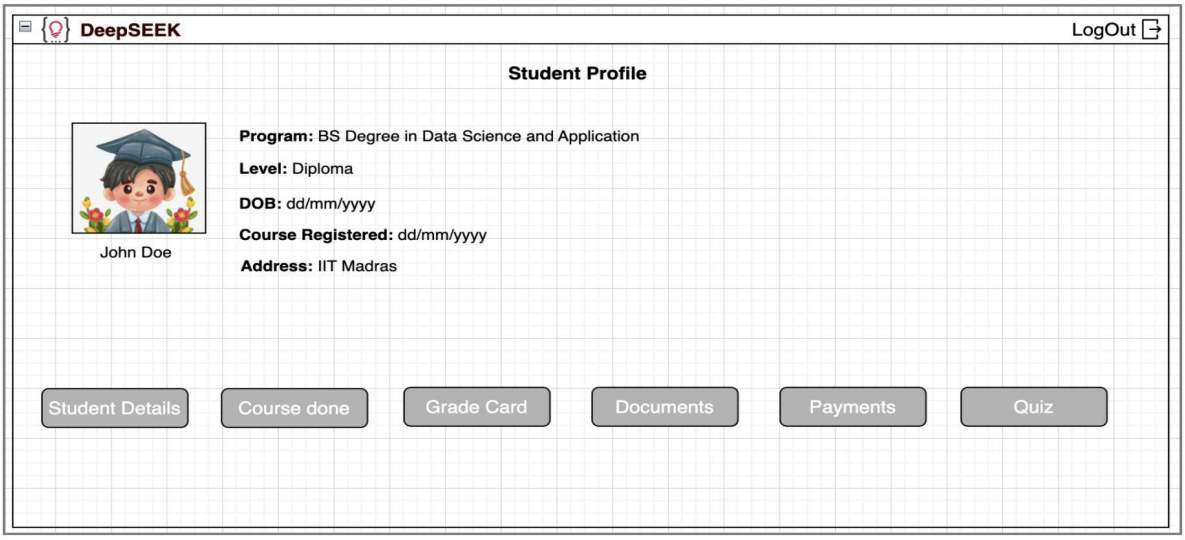
Smart reminders and notifications for students



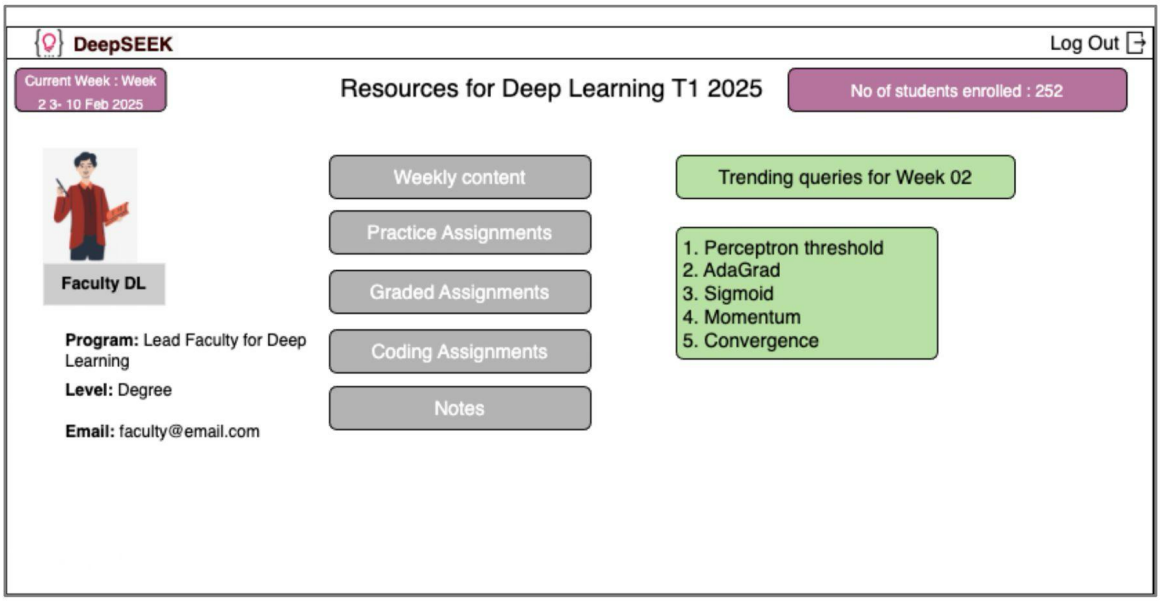
Support for Quiz leveraging RAG content



Profile and complete access of resources



Accessing and managing content for the Faculty



Conclusion

By the end of this milestone, a comprehensive storyboard and low-fidelity wireframes has been completed, aligning with the defined user stories. The designs adhere to usability principles, providing a clear and structured layout for the AI agent's interface. This milestone ensures that the UI is user-centric, easy to navigate, and functionally efficient, setting the stage for further development in subsequent milestones.

Milestone 3

Objective

Milestone 3 focused on project management and front-end development to ensure the seamless execution of tasks and alignment with project goals. This involved:

- Defining components, epics, user stories, and tasks for effective sprint planning.
- Setting up the backlog and organizing development priorities.
- Finalizing the system architecture, including the class diagram, database schema, ER diagram, and RAG framework.
- Developing the front-end components for user management, course management, and learning modules.

1. Project Schedule

1.1. Task Distribution

1. The entire project was divided into SDLC segments of Planning, Design, Development, Testing and Implementation as prescribed by the requirements.
2. The sprints were aligned to meet these deadlines.
3. The project segments and consequently the milestones were aligned to team members.
4. Finally, the sprints were aligned to meet the milestone deadlines while keeping the overall project progress in mind.

Milestone #	Milestone	SPOC	Key Roles
01.	User requirements/ stories	Vasu	• Vasu, Ghanashyam
02.	Storyboarding	Ghanashyam	• Ghanashyam
03.	Project Management, Front End	Vasu	• Vasu for Jira • Anuj for Front End • Jalaj for RAG Modeling
04.	APIs	Ajay	• Ajay for APIs • Niraj for RAG Backend
05.	Testing	Anand	• Anand
06.	Implementation	Jalaj	• Jalaj, Ajay, Niraj, Anuj

1.2 Roles by Team Members

Key roles and development responsibility of the team members is aligned based on their strengths and interests.

S.No	Team Member	Key Roles
01.	Anuj Gupta	Front End
02.	Ghanashyam R	Planning, Research, Design
03.	A J R Vasu	Scrum, Product Development
04.	Jalaj Trivedi	RAG R&D, development and implementation
05.	Niraj Kumar	RAG R&D, Backend development and implementation
06.	Anand Iyer	Testing and Correctness
07.	Ajay T	Application Backend

2. Sprints' Schedule

The Sprints have been aligned to both meet the milestone deadlines and keep the overall project progress on track.

2.1. Sprints' details

1. There are overall 8 sprints covering user requirements, storyboarding, UI development, implementing backend models/ logic, API development, RAG development, integration of all components and integrated testing.
2. The schedule for the sprints is given below.

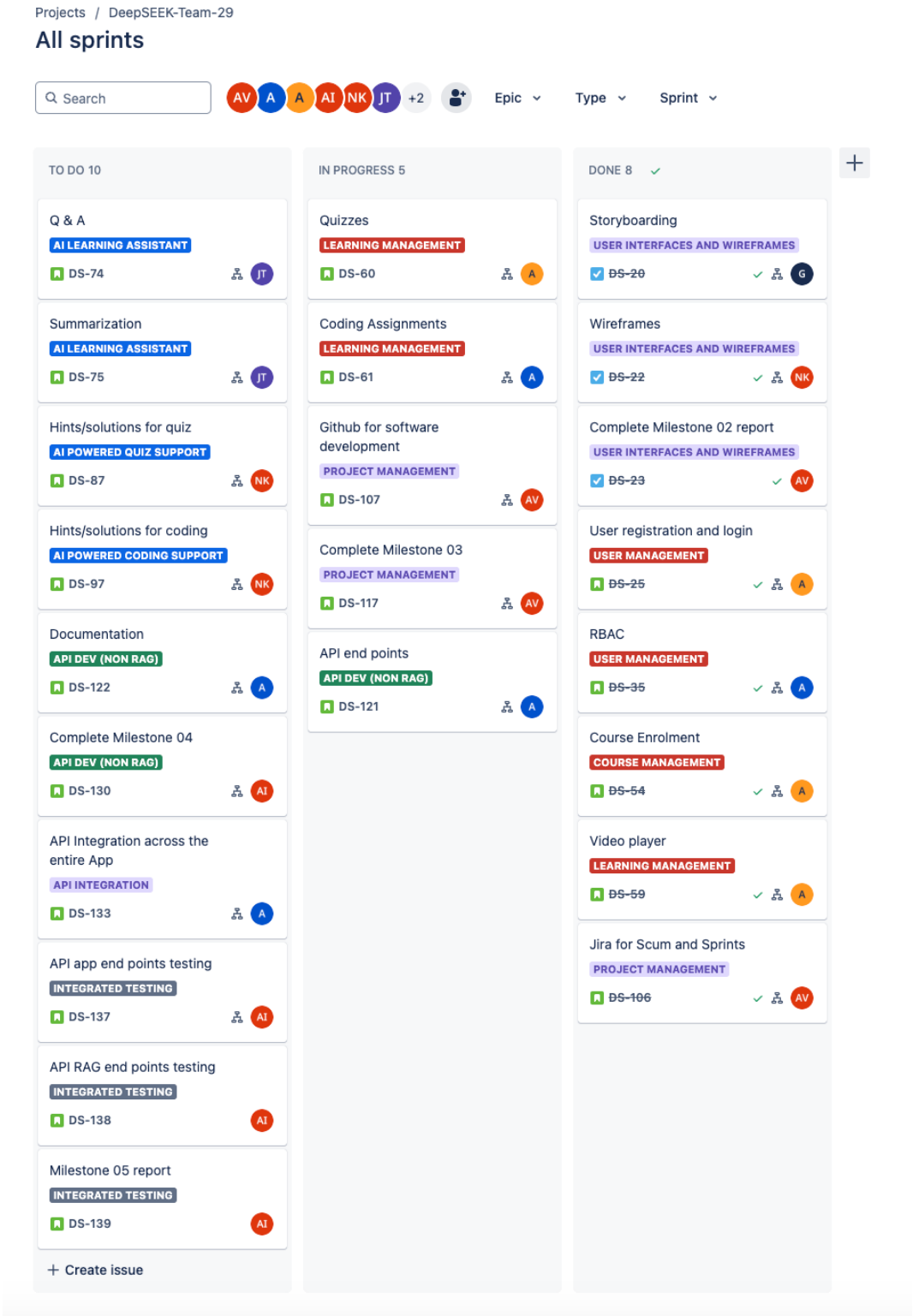
Sprint #	Topic/ EPIC	Start Date	End Date
Sprint-01	Identifying User Requirements	20 January 2025	26 January 2025
Sprint-02	User Interfaces and Wireframes	28 January 2025	02 February 2025
Sprint-03	Project Management	03 February 2025	09 February 2025
Sprint-04	Front End Development	10 February 2025	18 February 2025
Sprint-05	API Dev and Implementation	19 February 2025	28 February 2025
Sprint-06	AI Learning Assistant Development	03 March 2025	12 March 2025
Sprint-07	APIs, RAG and UI Integration	17 March 2025	23 March 2025
Sprint-08	Integrated Testing	24 March 2025	28 March 2025

2.2 Gantt Chart

1. We used Jira for project management.
2. Below is the Sprints timelines

3. Sprints Scrum Board

The snapshot of the Kanban scrum board is given below,



4. Scrum Meeting Details

Sprint 1 Scrum meetings minutes/details
Dates: 15 January, 2025 - 25 January, 2025

- The team started by finalizing roles within the project. Each member was assigned responsibilities based on their expertise and project requirements.
- Following this, users types and their requirements for the problem statement (SEEP portal) were identified.
- Subsequently user stories were developed to represent different users' interactions with the system.

Sprint 2 Scrum meetings minutes/details:
Dates: 26 January, 2025 - 2 February, 2025

- The team began work on milestone-2, starting with storyboard creation using Canva.
- Sample storyboards and wireframes were presented and discussed, allowing team members to provide feedback and refine the designs.

Sprint 3 Scrum meetings minutes/details:
Dates: 3 February, 2025 - 9 February, 2025

- The team will shift focus to milestone-3, which involves defining components, epics, stories, tasks, and sprint planning.
- Subsequently, the team discussed all the backlogs and set up the sprints basis mutually agreed timelines.
- Following this, a session was conducted to finalize the system architecture, including the class diagram, database schema, ER diagram and RAG framework and workflow which are also required for the milestone-3 report.

Sprint 4 Scrum meetings minutes/details:
Dates: 10 February, 2025 - 18 February, 2025

- Following sprint 3, discussions and demo were held to review the front-end implementation for user management, course management and learning management modules, ensuring it aligns with project requirements.
- Based on the feedback, the UI was further refined to accommodate the client suggestions and our own internal feedback.

5. Application Design Components

5.1 Front End

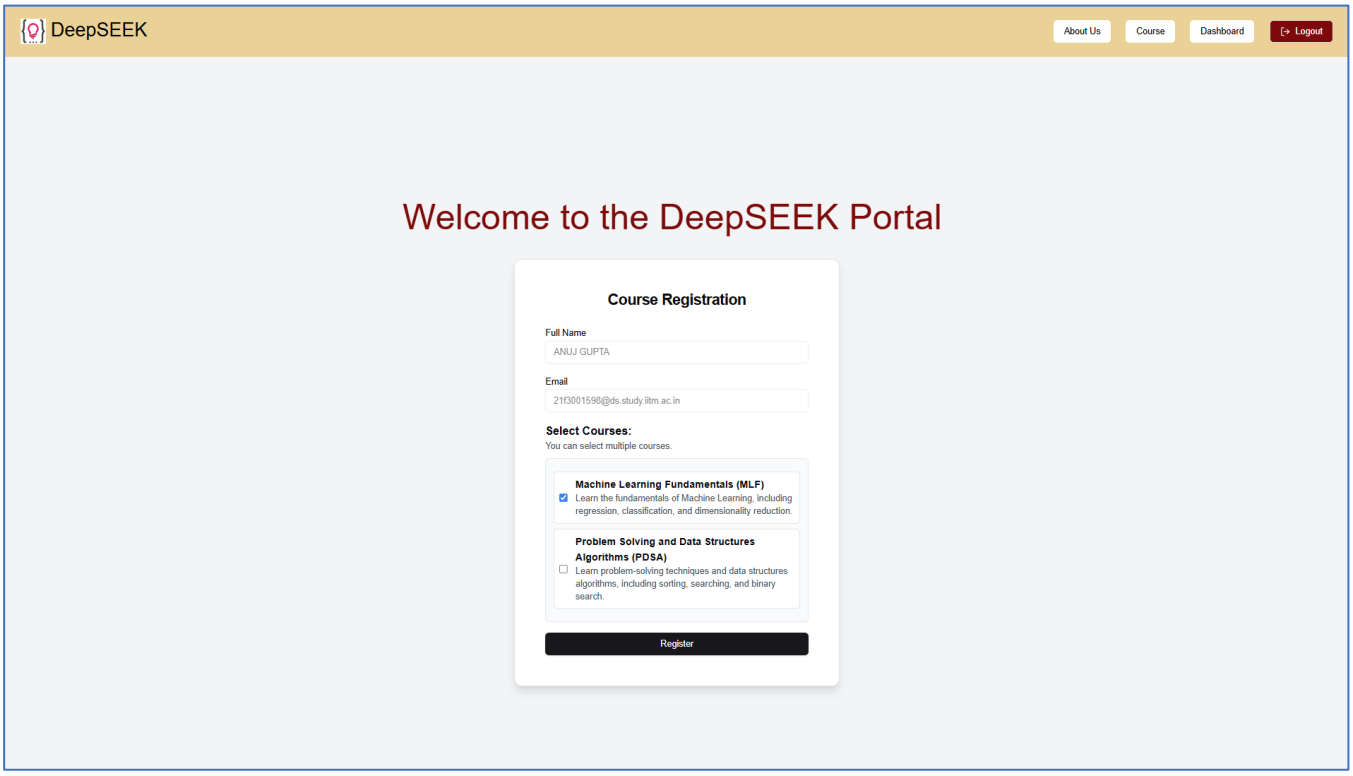
- The front end of DeepSEEK portal, the learning application, is built using React.js.
- It designed as a self-paced learning portal with AI assistance for more effective learning.
- The main components are,
  1. **Lecture Videos:** Allows students to watch lectures and access an AI Chatbot for queries.
  2. **Practice Assignments:** Offers interactive questions with AI assistance for hints, detailed explanations, and additional practice.
  3. **Programming Challenges:** Enables code submission with test case evaluation, plus AI support for debugging and optimizing code efficiency.
  4. **Assessments & Performance:** Supports graded assignments and provides AI-generated performance reports to highlight strengths and areas for improvement.
  5. **Dashboard:** Supports understanding of frequently asked questions to help revise.

5.2 Frontend UI (updated screens)

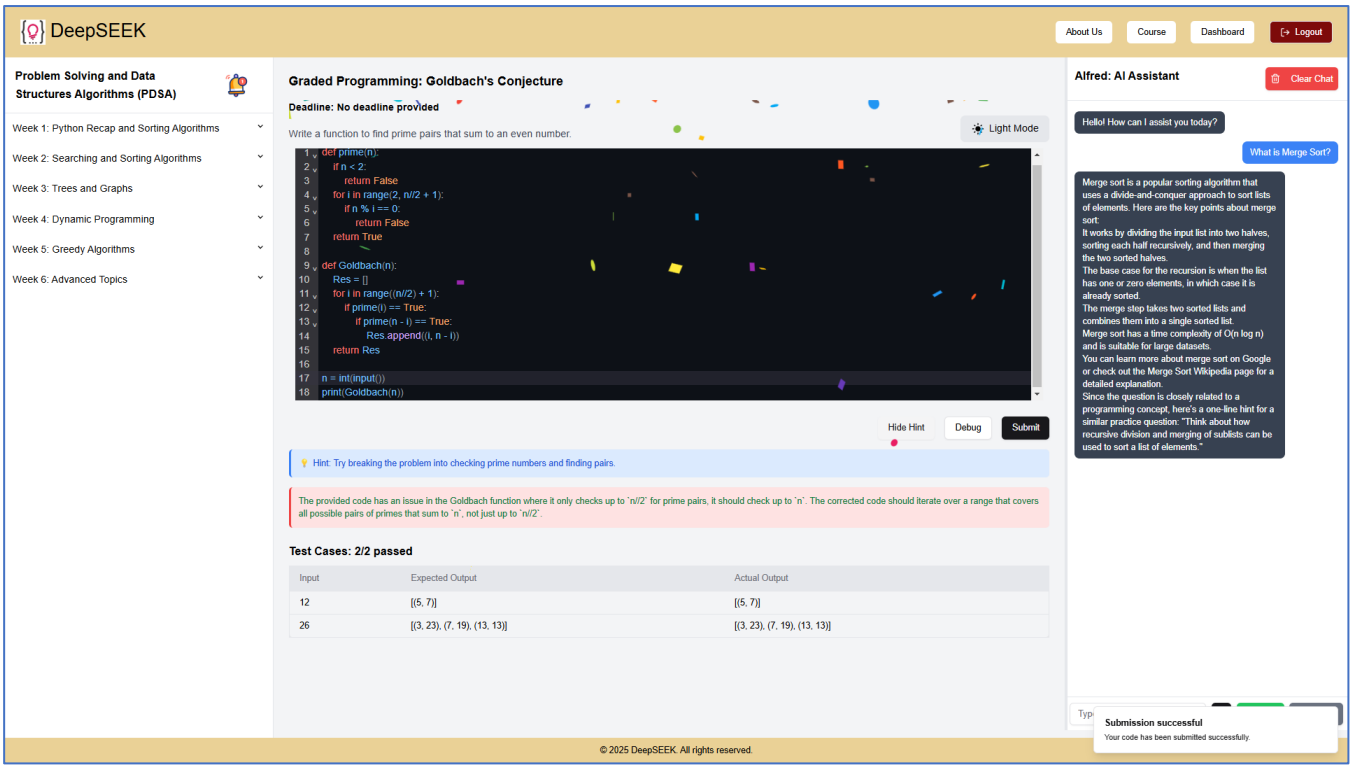
- The AI assistant chatbot is available all through the UI for asking concept led questions.
- In case of quiz questions and coding support the AI assistant in embedded and provides hints and debugging support.

Viewing videos, interacting with chatbot

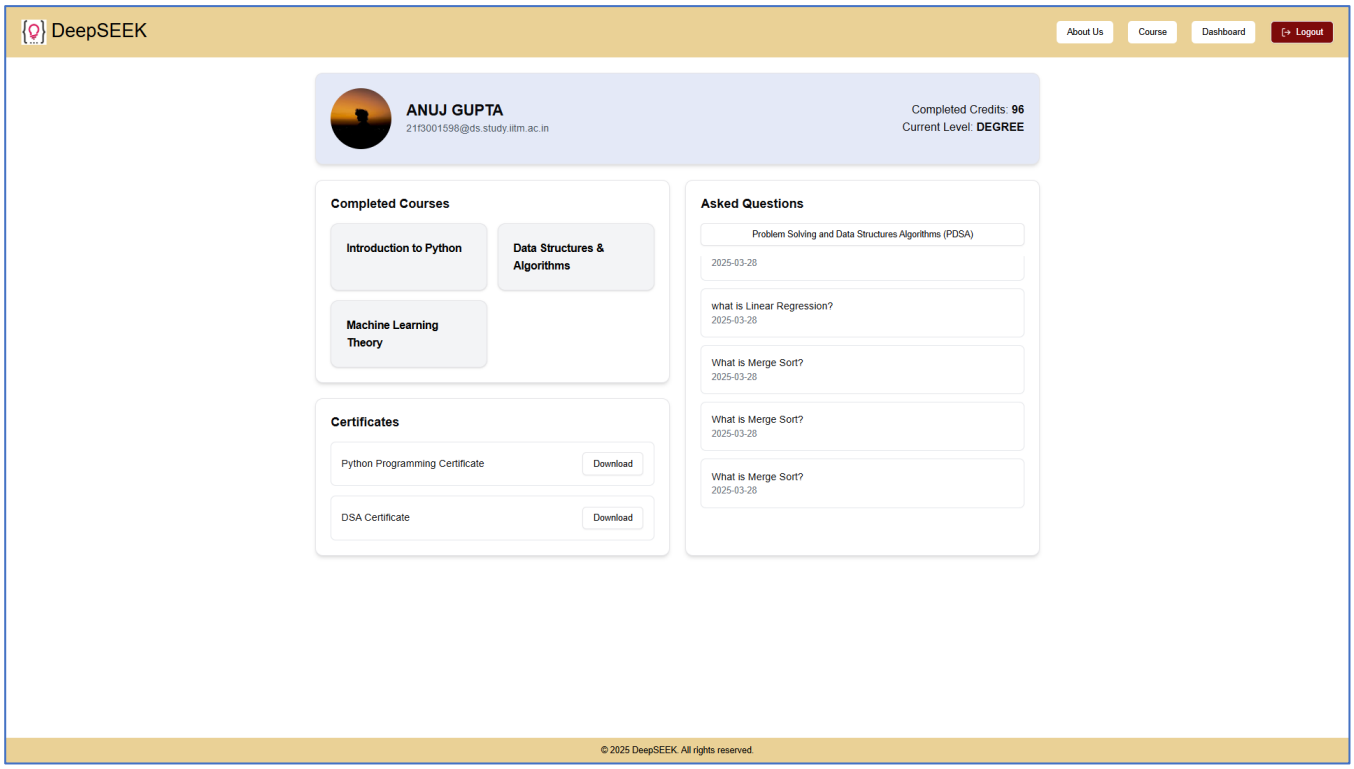
Login and registering for courses



Coding and debugging support via AI



User Dashboard

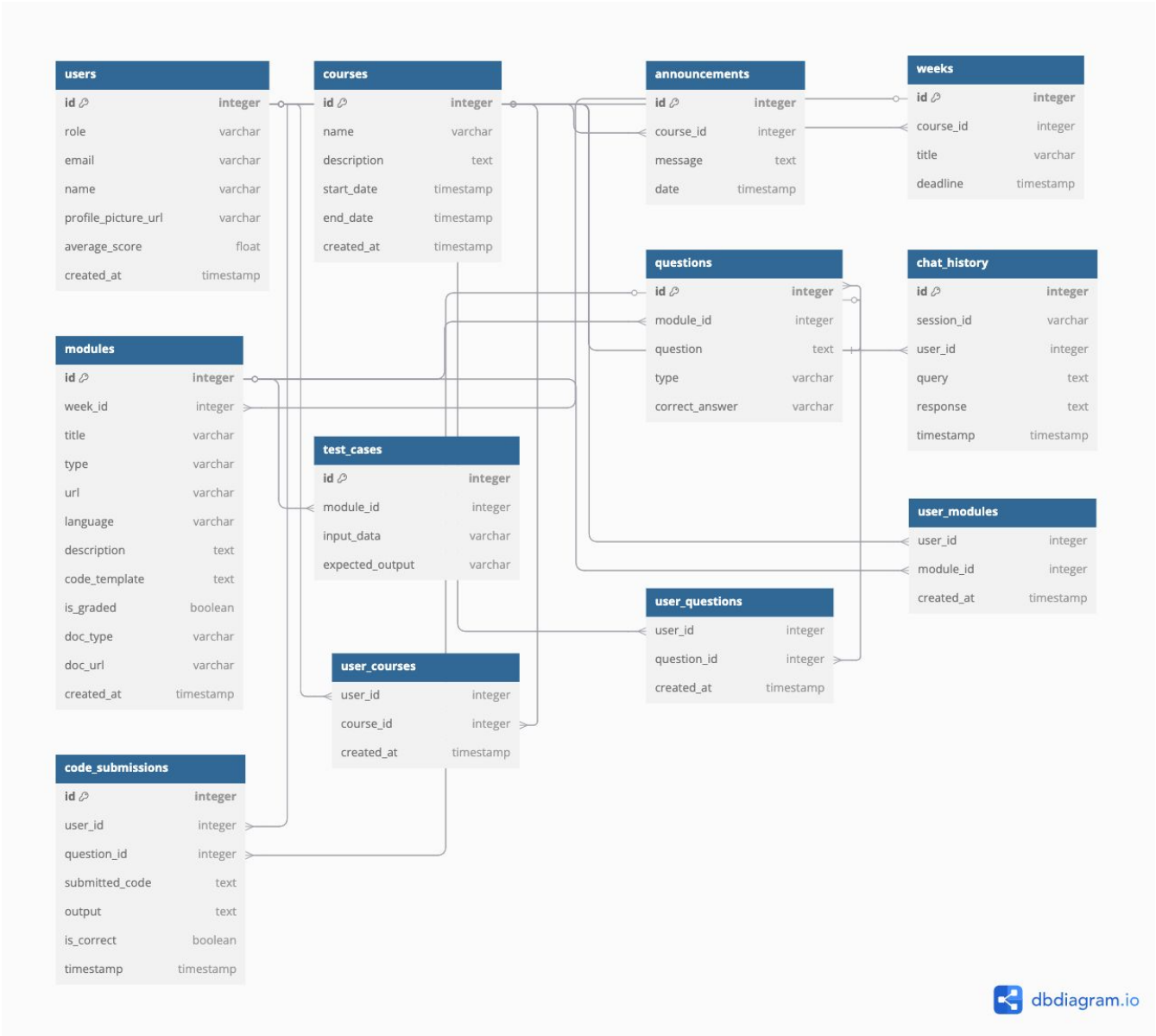


5.3 Back End

- The application utilizes the Flask micro webservices framework and Flask RESTful to connect the backend and frontend components.
- With MongoDB as the database the backend utilizes PyMongo and MongoEngine (ODM) to connect with the dB.
- RAG implementation utilizes FAISS (Vector database) and FastAPI for connectively.
- The main components are,
    1. **Request Management:** Flask and Flask RESTful handles user actions such as watching videos, submitting responses and completing assignments.
    2. **Business Logic & Data Processing**: Flask based APIs and custom functions process user requests, validates data, and manages database operations with integrated logging for error tracking.
    3. **Security Enforcement:** Implements user authentication leveraging Google's single sign-on and Flask based RBAC to ensure secure access and reliable error handling.
    4. **Gen AI Integration:** Leverages LangChain framework and connects with open-source LLMs via APIs to support chatbot interactions.

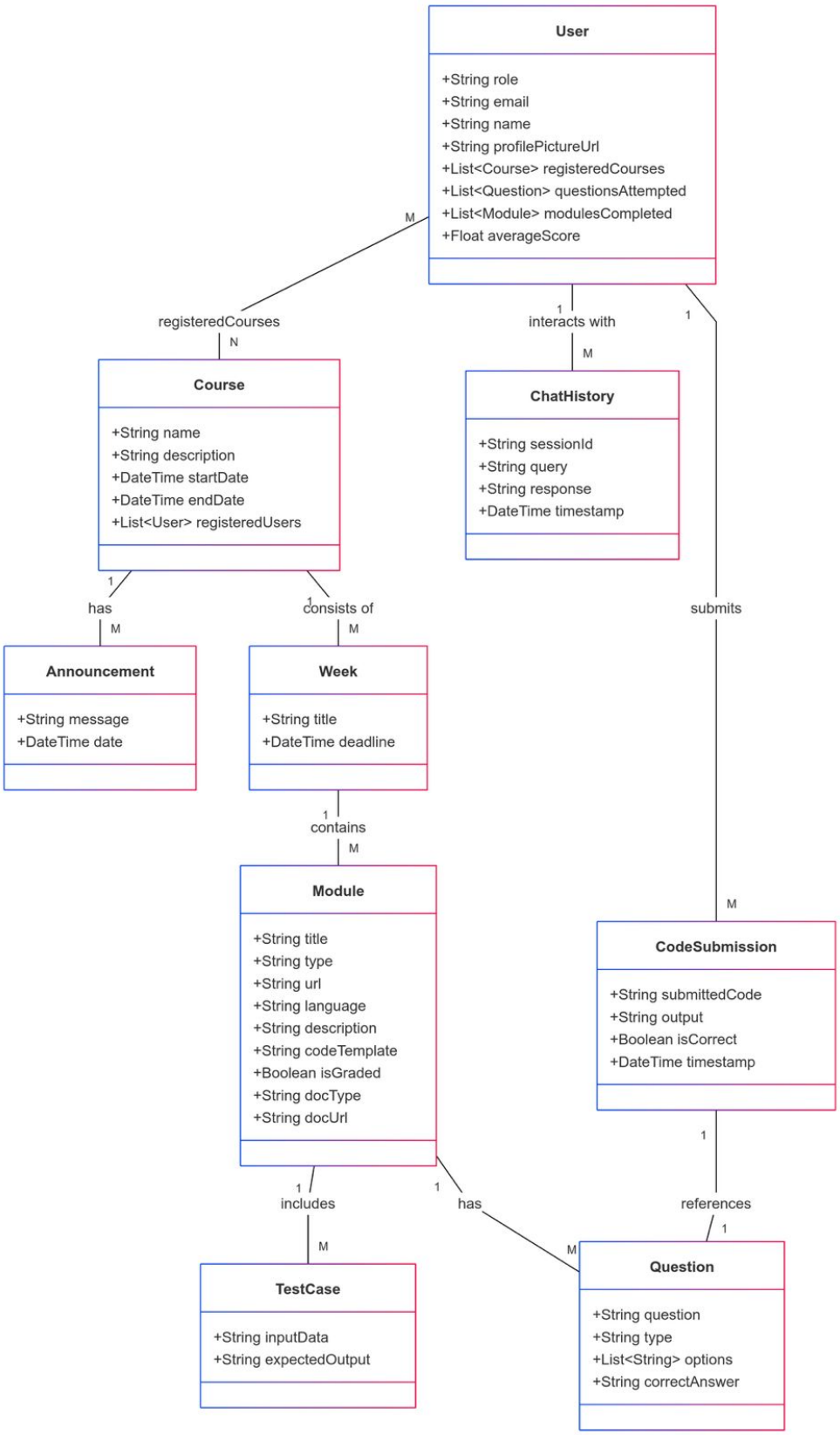
5.4 ER diagram

The below ER diagram shows entities and their relationships in the system. The database used for storing the tables and data for the application is MongoDB.



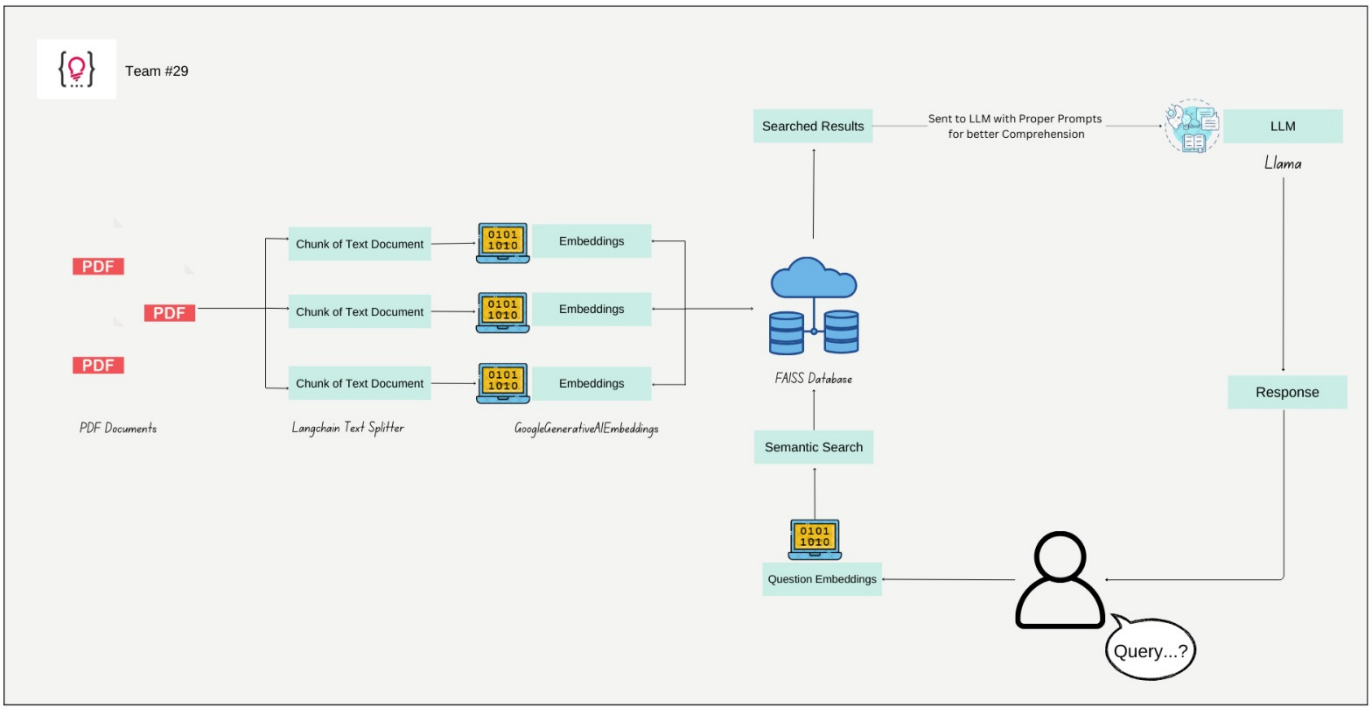
5.5 UML - Class diagram

Class diagram of the learning application depicts the various classes like Users, Courses, Lessons, Assignments, their attributes and methods and interactions.



5.6 RAG (Retrieval Augmented Generation) framework visualization

The below visual demonstrates how we plan to implement RAG workflow into the SEEK application using LangChain and how the various sub-systems interact to enable the AI chatbot.



Conclusion

By completing Milestone 3, the project team successfully established a well-structured project management framework and developed essential front-end components. The Jira-based sprint setup ensured effective tracking of tasks and backlog management. Additionally, the system architecture, including ER diagrams, class diagrams, and RAG workflows, was finalized, providing a robust foundation for subsequent API and AI development. The front-end UI was refined based on feedback, ensuring a user-friendly and intuitive design. With this milestone completed, the project is now ready for backend and API integration in the next phase.

Milestone 4

Objective

The objective of this milestone was to design and implement API endpoints for authentication, user management, course handling, AI-powered chatbot interactions, transcript retrieval, and code execution. The APIs were developed to facilitate seamless user onboarding, efficient course enrollment, AI-assisted learning support, and real-time analytics for administrators. Additionally, Swagger-compatible YAML documentation was created for the newly developed APIs, ensuring standardization and ease of integration.

API Endpoints

1.Authentication

POST /login
Handles user authentication by email and automatically creates accounts for new users. Returns user details including ID, role, name, email, profile picture, and last login timestamp. Validates required email field and handles server errors appropriately. Designed for seamless first-time user onboarding with minimal required fields.

2.Users

GET /users
Retrieves complete list of all registered users (admin/faculty only). Returns array with each user's ID, role, email, name, profile picture, and registered courses. Does not support pagination or filtering yet. Maintains consistent error response structure.

GET /user/{userId}
Fetches detailed profile for specific user after validating MongoDB ObjectId format. Returns 404 if user not found. Includes registered courses list and all basic user information in response.

DELETE /user/{userId}
Admin-only endpoint to permanently delete user accounts after ID validation. Returns a simple success message without confirmation step. Does not implement soft-delete functionality for recovery.

GET /user-statistics/{userId}
Provides learning analytics (questions attempted, modules completed, average scores) for individual users. Restricted to admin/faculty roles. Returns structured data for analytics dashboards.

GET /dashboard/user/questions
Organizes user's chatbot questions by course with timestamps. Sorts questions newest-first. Requires email parameters to identify users and fetch their question history.

3.Courses

GET /courses
Lists all available courses with basic info (ID, name, description, dates). Response optimized for course selection interfaces.

GET /course/{courseId}
Returns complete course structure with nested weekly content, announcements, and materials. Coding modules show test cases; assignments include questions with hints. Complex response supports rich navigation.

GET /registered-courses
Shows courses enrolled by specific user (email parameter). Returns simplified info (ID, name, description). Handles new users by returning empty array.

POST /registered-courses
Enrolls users in courses or creates new accounts. Accepts email + course ID array. Creates bidirectional user-course relationships automatically.

4.Transcripts

GET /video-transcript
Fetches YouTube captions using video URL parameter. Extracts video ID automatically and concatenates transcript chunks. Caches transcripts in the database to avoid repeated API calls.

5.AI Chatbot

POST /chatbot
AI course assistant processing natural language queries with conversation history. Logs questions by course/week for analytics. Integrates with external RAG API for responses.

POST /ask
Processes user queries by retrieving relevant document chunks from the FAISS database and combining them with chat history to generate a contextual response using an LLM. If no relevant documents are found, the LLM responds based on chat history alone.

POST /top-questions
Instructor-only AI analysis of frequent question topics. Processes all course questions to detect common themes. Returns top five formatted topics.

POST /debug/code
Provides AI-powered debugging via Groq API with strict prompt engineering. Returns concise two-line error explanations in consistent format.

POST /pdf
Retrieves a list of available PDFs from the document store. When a user clicks on the PDF icon, it generates and downloads the chat history as a PDF file for reference.

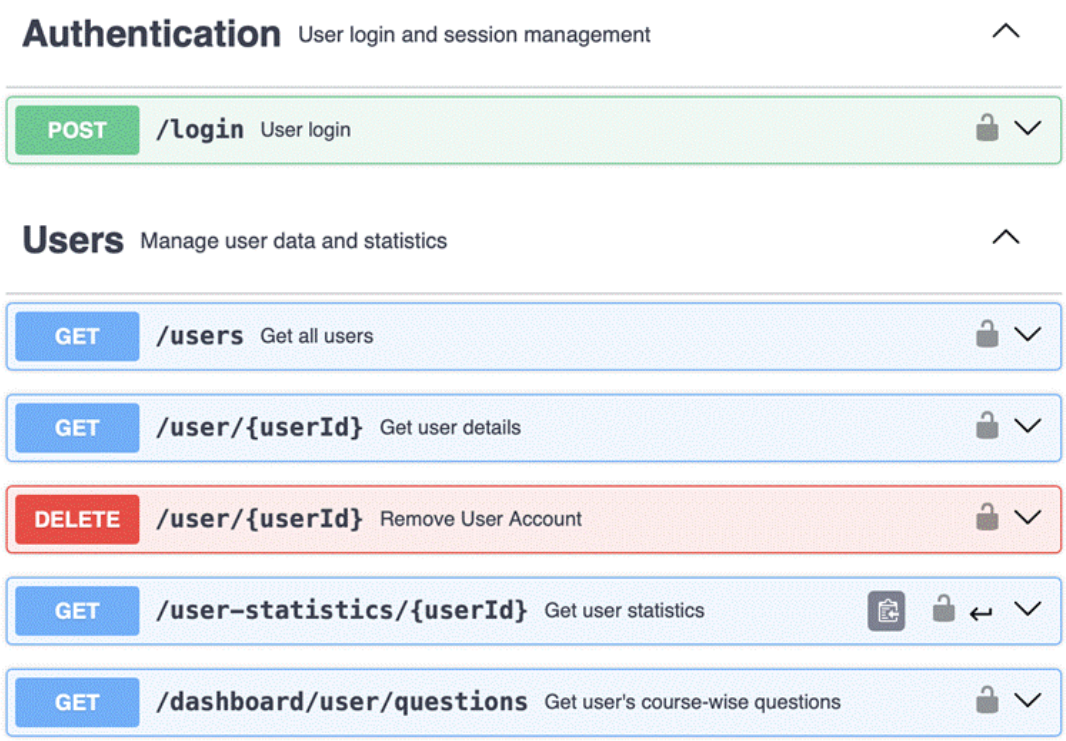
6.Code Execution

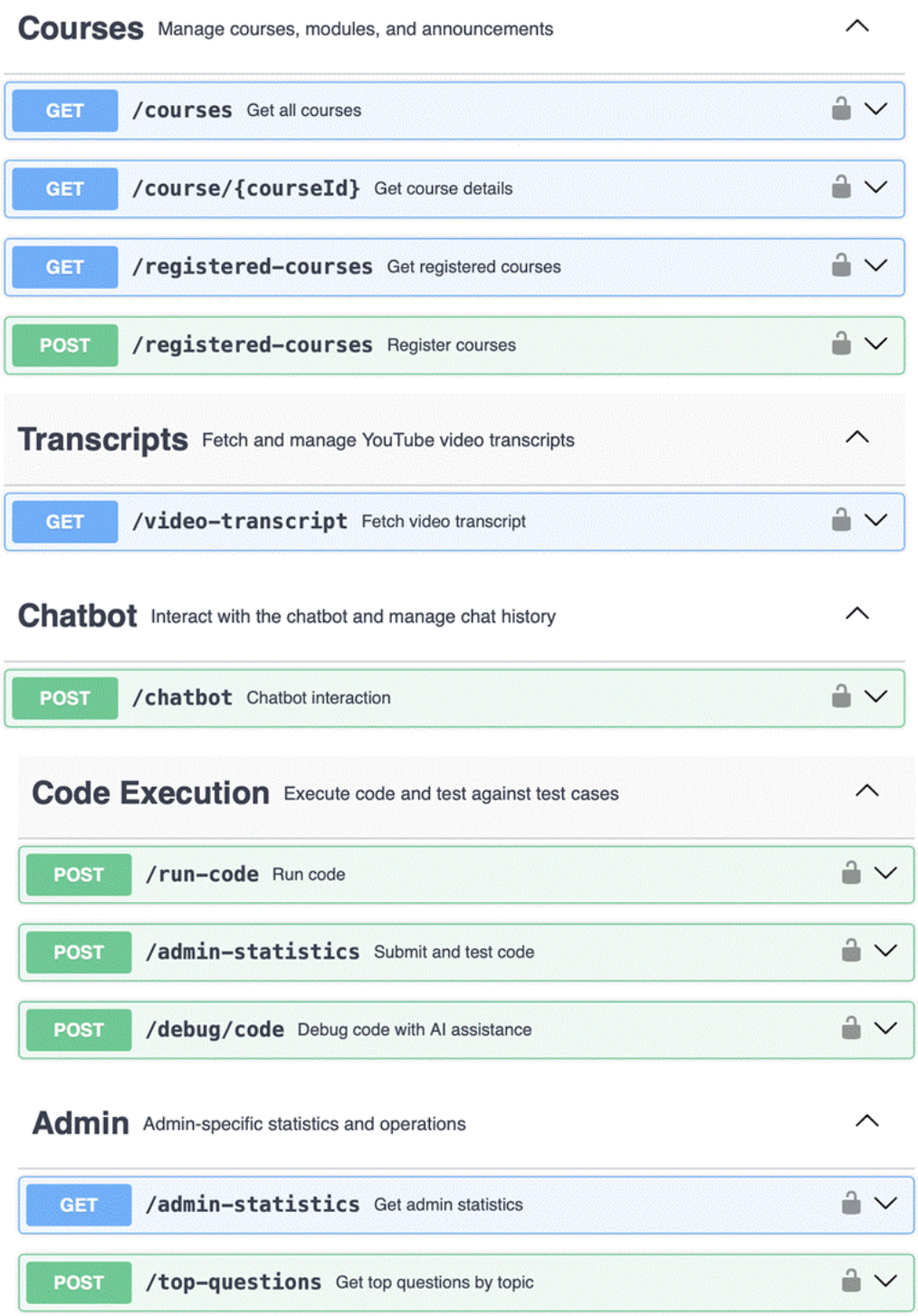
POST /run-code
Executes and validates code against module's test cases. Returns detailed input/output comparisons. Marks module complete if all tests pass.

POST /submit/code
Tests programming assignments with comprehensive feedback. Validates syntax before execution. Updates user progress upon successful completion.

7.Admin

GET /admin-statistics
System metrics for admins: user counts, content totals, aggregate activity. Shows sum of all questions attempted across platforms.





Conclusion

This milestone successfully delivered the required API endpoints for user authentication, course management, AI chatbot interactions, and code execution. Key functionalities, such as user authentication via google email, dynamic course enrollment, AI-driven question analysis, and automated transcript retrieval, have been implemented. Swagger documentation for the created APIs has been completed, ensuring API usability and integration readiness.

Milestone 5

Objective

The objective of this milestone was to design and execute a comprehensive test suite for all API endpoints to ensure robustness, reliability, and correctness of the system. Each API was tested using well-structured test cases, covering expected inputs, outputs, and edge cases. Automated tests were developed to validate API responses against expected results, ensuring that the authentication, user statistics, chatbot interactions, and other critical functionalities operate as intended.

1.Test Suite

This test suite is designed to verify the core functionalities of the system, including user authentication, course registration, user and admin statistics, and AI chatbot interactions. Each test module covers specific aspects of the application, ensuring reliability and consistency across various features.

Test Modules and Cases

Database Status

- **db_status**: Get DB connection status.

Login Tests (test_login.py)

- **1_student_login**: Verify login for student users.
- **2_admin_login**: Verify login for admin users.
- **3_faculty_login**: Verify login for faculty users.
- **4_email_required**: Ensure email is required for login.

Course Listing (test_courses.py)

- **list_courses**: Retrieve a list of all available courses.

Course Details (test_course.py)

- **course_details**: Fetch detailed information about a specific course using its ID.

Course Registration (test_register_courses.py)

- **1_register_course**: Allow a user to register for a course.
- **2_list_registered_courses**: Retrieve a list of registered courses for a user.
- **3_identify_user**: Identify a user during the registration process.
- **4_user_delete**: Delete a user record.
- **5_create_user**: Create a new user.
- **6_register_two_courses**: Register two courses for a user.
- **7_list_registered_courses**: Retrieve registered courses after multiple registrations.

- **8_register_invalid_course**: Attempt to register an invalid course.
- **9_list_registered_courses_without_email**: Try to list registered courses without providing an email.
- **10_list_registered_courses_invalid_user**: Attempt to list registered courses for an invalid user.
- **11_register_courses_empty_payload**: Handle registration with an empty request payload.

User Listing (test_users.py)

- **list_users**: Retrieve a list of all registered users in the system.

User Information (test_user.py)

- **1_login**: Authenticate a user.
- **2_user_details**: Fetch details of a specific user by their ID.
- **3_user_delete**: Delete a user record.
- **4_user_delete_user_not_found**: Handle cases where a non-existent user is deleted.
- **5_user_details_user_not_found**: Handle cases where details are requested for a non-existent user.

User Statistics (test_user_statistics.py)

- **1_login**: Authenticate a user before retrieving statistics.
- **2_user_statistics**: Retrieve statistics for a specific user.
- **3_user_delete**: Delete a user account.
- **4_user_statistics_user_not_found**: Handle cases where user statistics are requested for a non-existent user.

Video Transcript Retrieval (test_video_transcript.py)

- **1_transcript_video**: Retrieve a transcript for a lecture video using its URL.
- **2_transcript_invalid_video**: Handle cases where an invalid video URL is provided.
- **3_no_parameters**: Handle cases where no parameters are passed.

AI Chatbot Interaction (test_chatbot.py)

- **1_chatbot**: Verify chatbot query and response functionality.
- **2_incomplete_payload**: Handle cases where an incomplete payload is sent.
- **3_invalid_user**: Handle cases where an invalid user attempts to interact with the chatbot.

Admin Statistics (test_admin_statistics.py)

- **admin_statistics**: Retrieve administrative statistics, including user and system data management.

2. API and Individual Tests

Note: Only a few key test cases are shown here. Full report can be viewed here.

Login

Test 1: Student Login

API URL: https://api-deepseek.vercel.app/login
API Method: POST

Input Data:

```
{
    "email": "student_mail",
    "name": "student_name",
    "picture": "profile_picture"
}
```

Expected Status Code:

200

Actual Status Code:

200

Expected Keys and Data Types in Response:

- userId : str
- name : str
- email : str
- role : str
- message : str
- picture : str

Actual Keys and Data Types in Response:

- userId : str
- name : str
- email : str
- role : str
- message : str
- picture : str

Python Code:

```
def test_1_student_login(student_mail,
                         student_name,
                         profile_picture,
                         login_success_msg,
```

```
                        helpers):
    input_data = {
        "email": student_mail,
        "name": student_name,
        "picture": profile_picture
    }
    payload = json.dumps(input_data)

    response = requests.post(API_LOGIN, data=payload, headers=headers)

    assertEquals(response.status_code, 200)

    assertEquals(response.headers["Content-Type"], "application/json")

    data = response.json()

    assertInstance(data, dict)

    required_keys = {"userId":str,
                     "name":str,
                     "email":str,
                     "role":str,
                     "message":str,
                     "picture":str}
    verify_keys(required_keys, data)

    assertEquals(data['email'], student_mail)

    assertEquals(data['message'], login_success_msg)

    assertEquals(data['role'], "student")
```

Test 2: Admin Login

API URL: https://api-deepseek.vercel.app/login
API Method: POST

Input Data:

```
{
    "email": "admin_mail",
    "name": "admin_name",
    "picture": "profile_picture"
}
```

Expected Status Code:

200

Actual Status Code:

200

Expected Keys and Data Types in Response:

- userId : str
- name : str
- email : str
- role : str
- message : str
- picture : str

Actual Keys and Data Types in Response:

- userId : str
- name : str
- email : str
- role : str
- message : str
- picture : str

Python Code:

```
def test_2_admin_login(admin_mail,
                       admin_name,
                       profile_picture,
                       login_success_msg):
    input_data = {
    "email": admin_mail,
    "name": admin_name,
    "picture": profile_picture
    }
    payload = json.dumps(input_data)

    response = requests.post(API_LOGIN, data=payload, headers=headers)

    assertEquals(response.status_code, 200)

    assertEquals(response.headers["Content-Type"], "application/json")

    data = response.json()

    assertInstance(data, dict)

    required_keys = {"userId":str,
                     "name":str,
                     "email":str,
                     "role":str,
                     "message":str,
                     "picture":str}
    verify_keys(required_keys, data)

    assertEquals(data['email'], admin_mail)

    assertEquals(data['message'], login_success_msg)
```

```
    assertEquals(data['role'], "admin")
```

Test 3: Faculty Login

API URL: `https://api-deepseek.vercel.app/login`
API Method: `POST`

Input Data:

```
{
    "email": "faculty_mail",
    "name": "faculty_name",
    "picture": "profile_picture"
}
```

Expected Status Code:

`200`

Actual Status Code:

`200`

Expected Keys and Data Types in Response:

- `userId : str`
- `name : str`
- `email : str`
- `role : str`
- `message : str`
- `picture : str`

Actual Keys and Data Types in Response:

- `userId : str`
- `name : str`
- `email : str`
- `role : str`
- `message : str`
- `picture : str`

Python Code:

```
def test_3_faculty_login(faculty_mail,
                         faculty_name,
                         profile_picture,
                         login_success_msg):
    input_data = {
    "email": faculty_mail,
    "name": faculty_name,
```

```
    "picture": profile_picture
    }
    payload = json.dumps(input_data)

    response = requests.post(API_LOGIN, data=payload, headers=headers)

    assertEquals(response.status_code, 200)

    assertEquals(response.headers["Content-Type"], "application/json")

    data = response.json()

    assertInstance(data, dict)

    required_keys = {"userId":str,
                     "name":str,
                     "email":str,
                     "role":str,
                     "message":str,
                     "picture":str}
    verify_keys(required_keys, data)

    assertEquals(data['email'], faculty_mail)

    assertEquals(data['message'], login_success_msg)

    assertEquals(data['role'], "faculty")
```

Test 4: Email Required

API URL: https://api-deepseek.vercel.app/login
API Method: POST

Input Data:

```
{
    "email": "faculty_mail",
    "name": "faculty_name",
    "picture": "profile_picture"
}
```

Expected Status Code:

400

Actual Status Code:

400

Expected Error Message:

Email is required

Actual Error Message:

```
Email is required
```

Expected Keys and Data Types in Response:

- `error : str`

Actual Keys and Data Types in Response:

- `error : str`

Python Code:

```
def test_4_email_required(student_name,
                          profile_picture,
                          email_required_msg):
    input_data = {
    "email": "",
    "name": student_name,
    "picture": profile_picture
    }
    payload = json.dumps(input_data)

    response = requests.post(API_LOGIN, data=payload, headers=headers)

    assertEquals(response.status_code, 400)

    data = response.json()

    required_keys = {"error":str}
    verify_keys(required_keys, data)

    assertEquals(data['error'], email_required_msg)
```

AI Chatbot API Test Documentation

Test 1: Chatbot Interaction

API URL: `https://api-deepseek.vercel.app/chatbot`
API Method: `POST`

Input Data:

```
{
    "query": "What is ML?",
    "option": "option",
    "sessionId": "S1234",
    "userEmail": "student_mail"
}
```

Expected Status Code:

`200`

Actual Status Code:

`200`

Expected Keys and Data Types in Response:

- `sessionId` : `str`
- `question` : `str`
- `answer` : `str`
- `chatHistory` : `list`

Actual Keys and Data Types in Response:

- `sessionId` : `str`
- `question` : `str`
- `answer` : `str`
- `chatHistory` : `list`

Additional Data Verification

Chat History Structure:

Each item in the `chatHistory` list must contain:

- `query` : `str`
- `answer` : `str`
- `timestamp` : `str`
- `user` : `dict`

User Structure:

Each `user` object in the `chatHistory` must contain:

- id : str
- name : str
- email : str
- role : str
- profilePictureUrl : str

Python Code:

```
def test_1_chatbot(student_mail,
                   student_id,
                   student_name,
                   student_pp):
    session_id = "S1234"
    query = "What is ML?"
    input_data = {
        "query": query,
        "option": "option",
        "sessionId": session_id,
        "userEmail": student_mail
    }
    payload = json.dumps(input_data)

    response = requests.post(API_CHATBOT, data=payload, headers=headers)

    assertEquals(response.status_code, 200)

    assertEquals(response.headers["Content-Type"], "application/json")

    data = response.json()

    assertInstance(data, dict)

    required_keys = {"sessionId":str,
                         "question":str,
                         "answer":str,
                         "chatHistory": list
                     }
    verify_keys(required_keys, data)

    assertEquals(data['sessionId'], session_id)

    assertEquals(data['question'], query)

    chathistories = data["chatHistory"]
    for chathistory in chathistories:
        assertInstance(chathistory, dict)

        required_keys = {"query":str,
                             "answer":str,
                             "timestamp":str,
                             "user": dict
                         }
        verify_keys(required_keys, chathistory)
```

```
        user = chathistory['user']
        required_keys = {"id":str,
                            "name":str,
                            "email":str,
                            "role": str,
                            "profilePictureUrl": str
                        }
        verify_keys(required_keys, user)

        assertEquals(user['name'], student_name)

        assertEquals(user['email'], student_mail)

        assertEquals(user['role'], "student")

        assertEquals(user['profilePictureUrl'], student_pp)
```

Test 2: Incomplete Payload

API URL: `https://api-deepseek.vercel.app/chatbot`
API Method: `POST`

Input Data:

```
{
    "option": "option",
    "sessionId": "S1234",
    "userEmail": "student_mail"
}
```

Expected Status Code:

`400`

Actual Status Code:

`400`

Expected Error Message:

`Query and option are required`

Actual Error Message:

`Query and option are required`

Expected Keys and Data Types in Response:

- `error : str`

Actual Keys and Data Types in Response:

- `error : str`

Python Code:

```
def test_2_incomplete_payload(student_mail,
                                    query_option_required_msg):
    input_data = {
        "option": "option",
        "sessionId": "S1234",
        "userEmail": student_mail
    }
    payload = json.dumps(input_data)

    response = requests.post(API_CHATBOT, data=payload, headers=headers)

    assertEquals(response.status_code, 400)

    assertEquals(response.headers["Content-Type"], "application/json")

    data = response.json()

    assertInstance(data, dict)

    required_keys = {"error":str}
    verify_keys(required_keys, data)

    assertEquals(data['error'], query_option_required_msg)
```

Test 3: Invalid User

API URL: https://api-deepseek.vercel.app/chatbot
API Method: POST

Input Data:

```
{
    "query": "What is ML?",
    "option": "option",
    "sessionId": "S1234",
    "userEmail": "invalid_student_mail"
}
```

Expected Status Code:

404

Actual Status Code:

404

Expected Error Message:

User not found

Actual Error Message:

```
User not found
```

Expected Keys and Data Types in Response:

- `error : str`

Actual Keys and Data Types in Response:

- `error : str`

Python Code:

```
def test_3_invalid_user(invalid_student_mail,
                            user_not_found_msg):
    input_data = {
        "query": "What is ML?",
        "option": "option",
        "sessionId": "S1234",
        "userEmail": invalid_student_mail
    }
    payload = json.dumps(input_data)

    response = requests.post(API_CHATBOT, data=payload, headers=headers)

    assertEquals(response.status_code, 404)

    assertEquals(response.headers["Content-Type"], "application/json")

    data = response.json()

    assertInstance(data, dict)

    required_keys = {"error":str}
    verify_keys(required_keys, data)

    assertEquals(data['error'], user_not_found_msg)
```

Conclusion

This milestone successfully developed and executed a test suite to validate the key API functionalities, ensuring the robustness of user authentication, chatbot interactions, and user statistics retrieval. The structured test cases and automation scripts provide a strong foundation for ongoing validation and maintenance. With all tests passing successfully, the milestone objectives have been fully accomplished.

Milestone 6

Objective

The objective of this milestone is to bring together all the individual components of the project into a cohesive and fully functional application, ready for deployment. This includes the seamless integration of the Frontend interface, Backend services, Retrieval-Augmented Generation (RAG) system, and the Large Language Model (LLM), which is connected through APIs. The focus is on ensuring that these components communicate effectively and function as a unified system.

In addition to the technical integration, the milestone report also provides an overview of the various software technologies, libraries, frameworks, and tools that were utilized throughout the project lifecycle. This includes tools for version control, code reviews, issue tracking, and collaborative development, all of which played a critical role in ensuring smooth functioning.

1. Technologies and tools used

The below table summarizes the libraries, frameworks and tools used for executing the project and the application area for each of the tools.

Library / Framework / Tool/ Model	Application area
Vercel, Render	Hosting, CI
React JS, Router, TypeScript, TailwindCSS, ShadCN, Vite	Application Frontend
Python	Core programming
Flask	Application Backend
MongoDB	Databasing
PyMongo and MongoEngine (ODM)	Database Connectivity
FAISS Vector Database plus FastAPIs	RAG Database and Connectivity
Flask RESTful	Application Interface
Google SSO (Google OAuth)	Authentication, Security, Access
Swagger API Editor	API Documentation
Langchain	AI Chatbot
Llama (llama-3.3-70b-versatile)	Language Model
Postman	Manual Testing
Pytest	Automated Testing
GitHub for Code review , Issue reporting and tracking	Versioning and Development
Jira	Productivity
Canva, draw.io	UML

2. Instructions to run the application

Getting Started

Clone the Repository

```
git clone https://github.com/21f3002975/seek-portal-ai-agent.git
cd seek-portal-ai-agent
```

Backend Setup (Flask)

Set Up a Virtual Environment

```
cd backend
python3 -m venv venv
source venv/bin/activate  # MacOS/Linux
venv\Scripts\activate  # Windows
pip install -r requirements.txt
```

Configure Environment Variables

Create a `.env` file and add:

```
MONGO_URI=
RAG_API=
GROQ_API_KEY=
```

Run the Backend Server

```
python -m api.app
```

API is live at: `http://localhost:5000/`

Frontend Setup (React + Vite)

Install Dependencies

```
cd frontend
npm install
```

Configure Environment Variables

Create a `.env` file in the root directory and add:

```
VITE_API_URL=
VITE_GOOGLE_CLIENT_ID=
```

Start the Frontend Server

```
npm run dev
```

App is live at: `http://localhost:3000/`

RAG & AI Setup (FastAPI + LangChain)

Set Up Virtual Environment

```
cd rag
python -m venv venv
source venv/bin/activate  # MacOS/Linux
venv\Scripts\activate  # Windows
```

Install Dependencies

```
pip install -r requirements.txt
```

Configure Environment Variables

Create a `.env` file and add:

```
GROQ_API_KEY=your_groq_api_key
GOOGLE_API_KEY=your_google_api_key
```

Run the RAG & AI Server

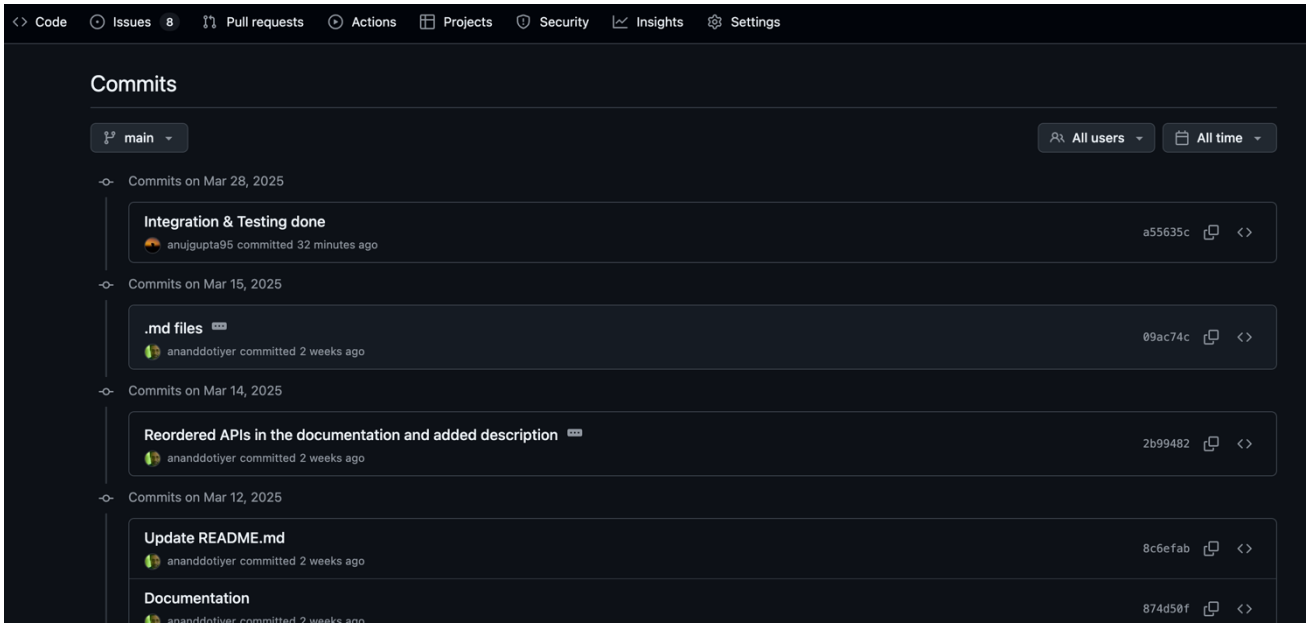
```
uvicorn app:app --reload --host 0.0.0.0 --port 8000
```

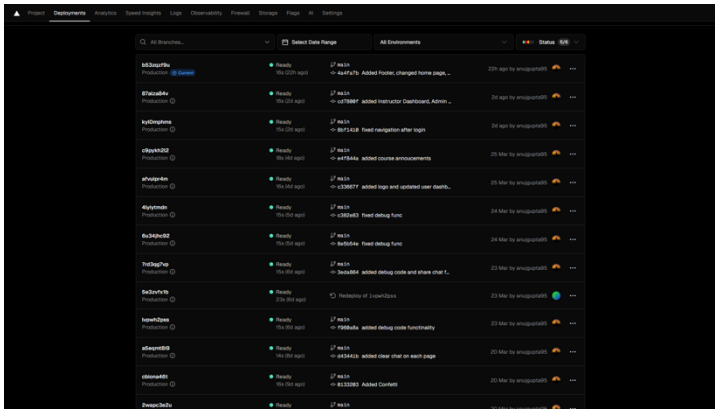
API is live at: `http://localhost:8000/`

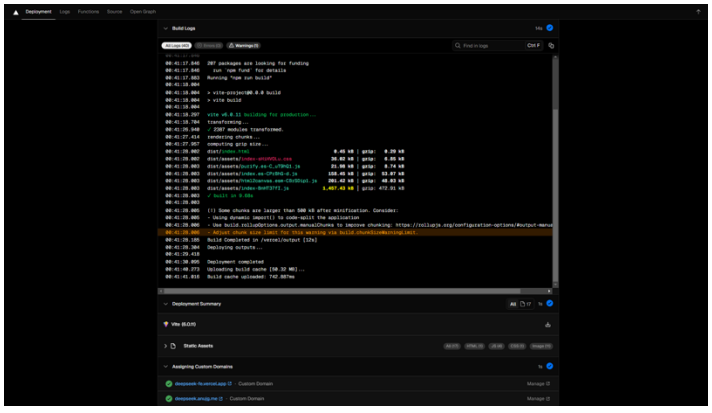
3. Code development, review, issues tracking and deployment

- We have leveraged GitHub for versioning, code management and issue tracking.
- Additionally, we have implemented continuous integration using Git and Vercel. The updated repo from Git is fetched by Vercel and the application is deployed if it is a successful build.
- WhatsApp group chat has also been leveraged for updating each other and to conduct reviews.

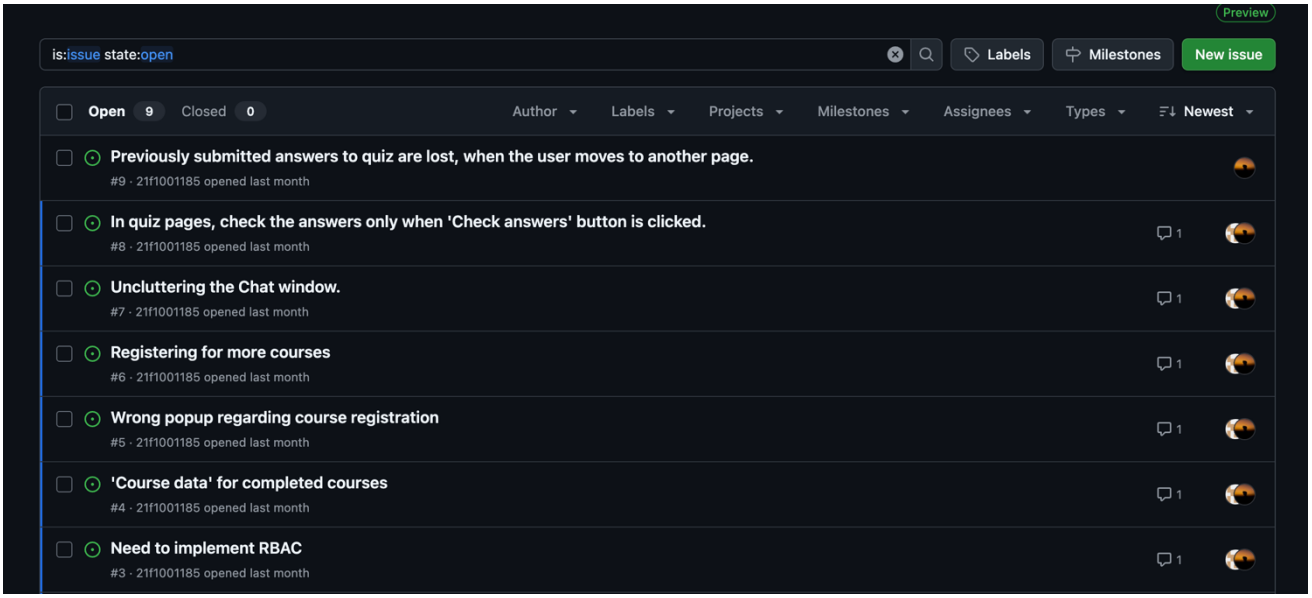
3.1 Commits, builds and deployment

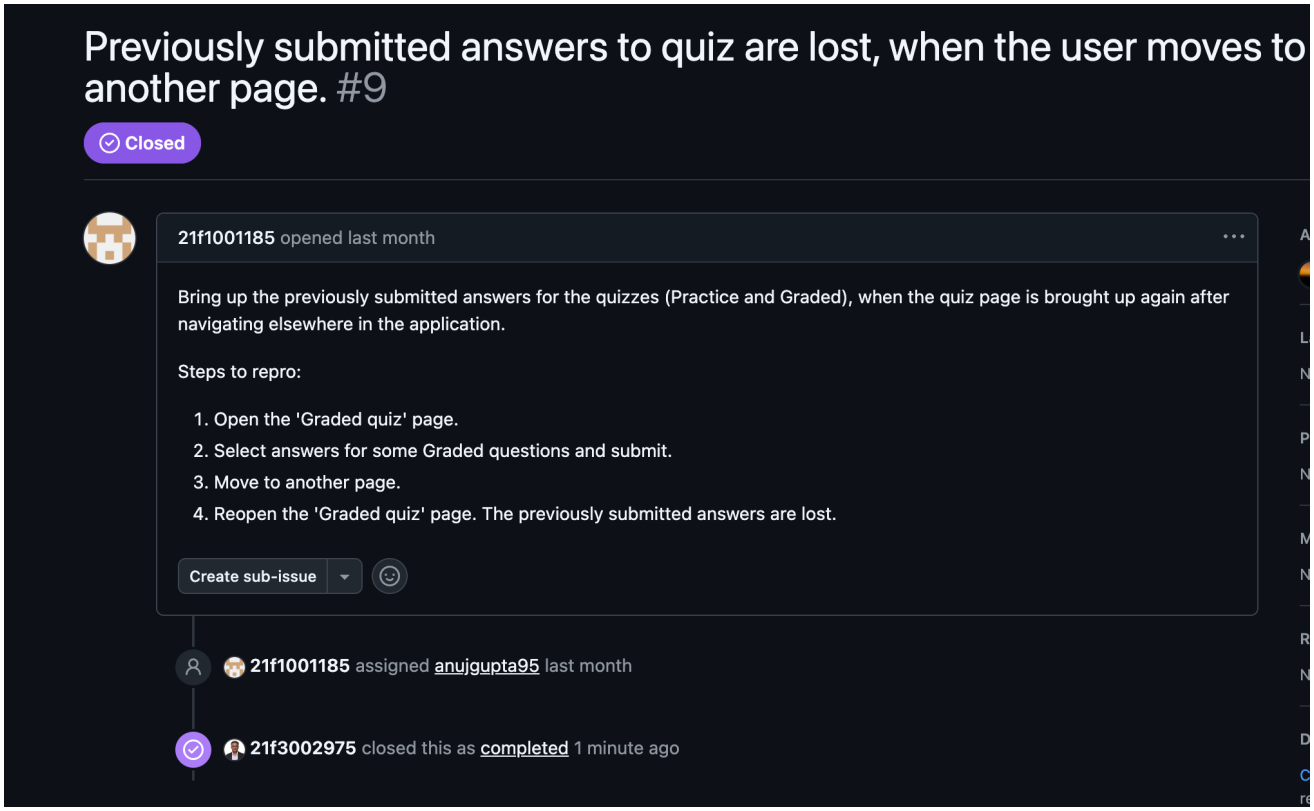




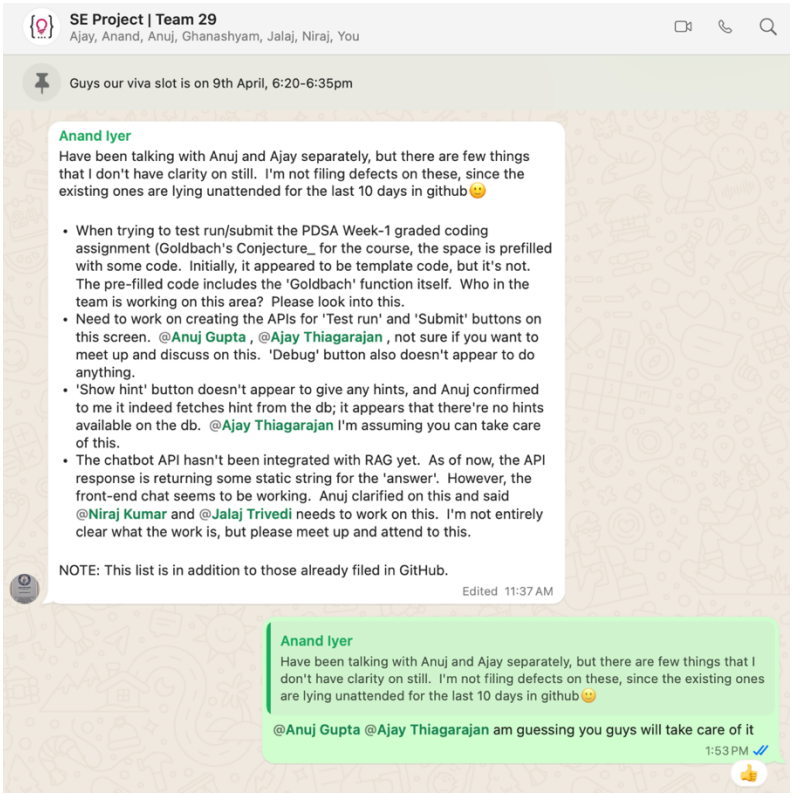


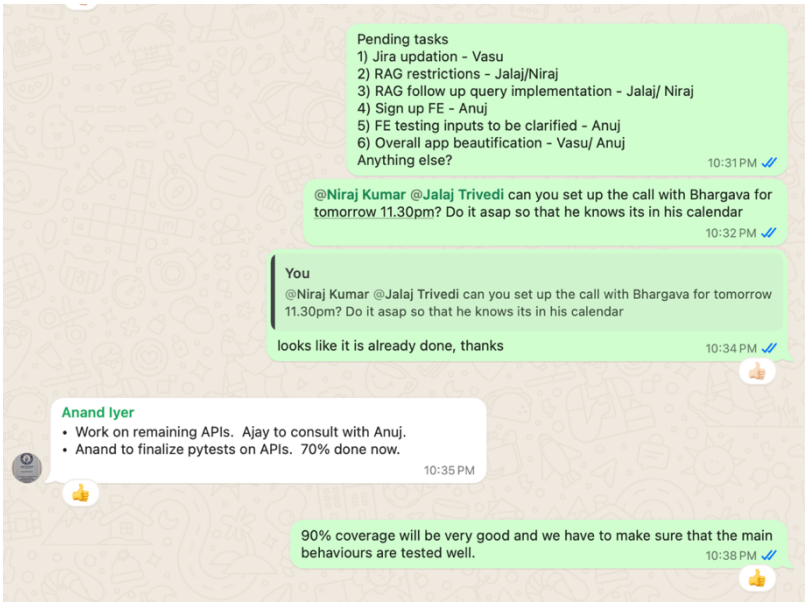
3.2 Issues tracking





3.3 Reviews and Updates





Conclusion

This milestone successfully integrated and deployed the final application, DeepSEEK. The seamless combination of the frontend, backend, RAG pipeline, and LLM through API integration marks a significant achievement in the development cycle. This milestone lays a strong foundation for future enhancements and scaling of DeepSEEK.

Thank you for reading and reviewing !

For questions and suggestions :
21f3002975@ds.study.iitm.ac.in