Course Experience Exchange

CSCI 5253-003

DataCenter Scale Computing (Fall 2024)

Final Project - Group 117

1. **DEVELOPERS**

- **Sneha Nagaraju**

2. INTRODUCTION

The "Course Experience Exchange" is a comprehensive, web-based platform aimed at helping students make well-informed decisions regarding their academic courses. By enabling the sharing of detailed reviews, rating and feedback, the platform seeks to improve the educational experience for students and foster a culture of continuous improvement in academia.

3. PROJECT GOALS

3.1. Objectives

The primary objectives of the Course Experience Exchange platform are:

- Build a platform for learners to review and access feedback on academic courses.
- To enable students to share comprehensive feedback based on their academic experiences.
- To assist prospective students in making decisions that align with their academic aspirations and learning preferences.
- To promote transparency and continuous improvement in academic offerings.

3.2. Accomplishments

1. Functional Components:
   - Built a fully functional platform for anonymous course feedback collection, ensuring users can express honest opinions without fear of bias.
2. User Authentication
   - Implemented secure login and user session management.
   - Provided a seamless experience for users to access the application.
3. Course Management Services
   - Users can filter courses based on department and graduation level.
   - Courses comments can include images that will be stored in cloud storage.

4. Enhance Transparency:
    - Provided students with detailed insights into courses and instructors through peer reviews and feedback, aiding informed decision-making.

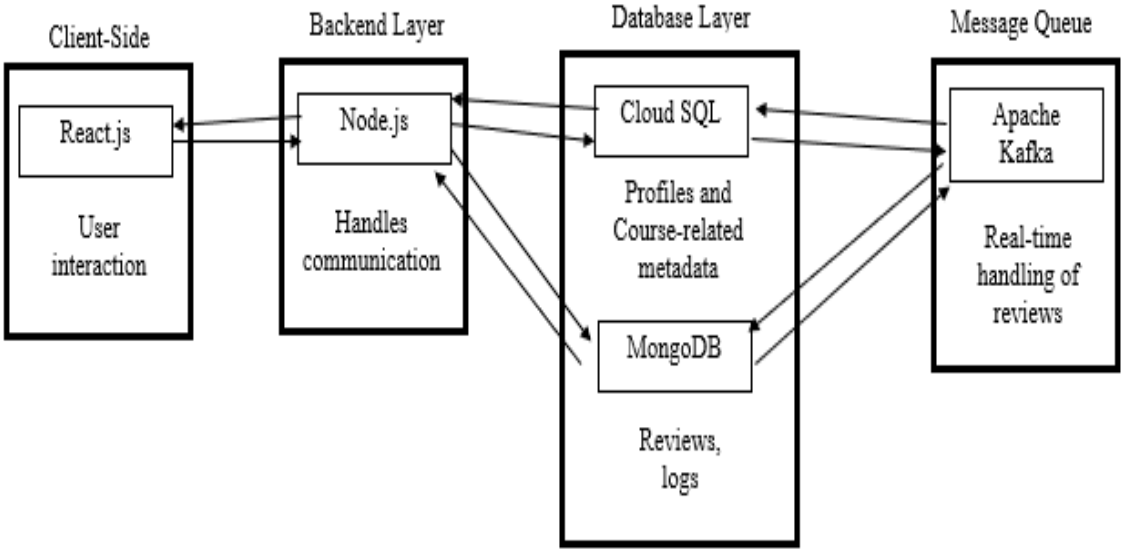
3.3. **Summary**

The platform enables anonymous course feedback collection, allowing users to share honest opinions. It features secure user authentication and easy access, along with course filtering by department and graduation level. Users can add comments with images stored in cloud storage. The system enhances transparency, helping students make informed decisions through peer reviews and feedback, fostering trust and informed choices.

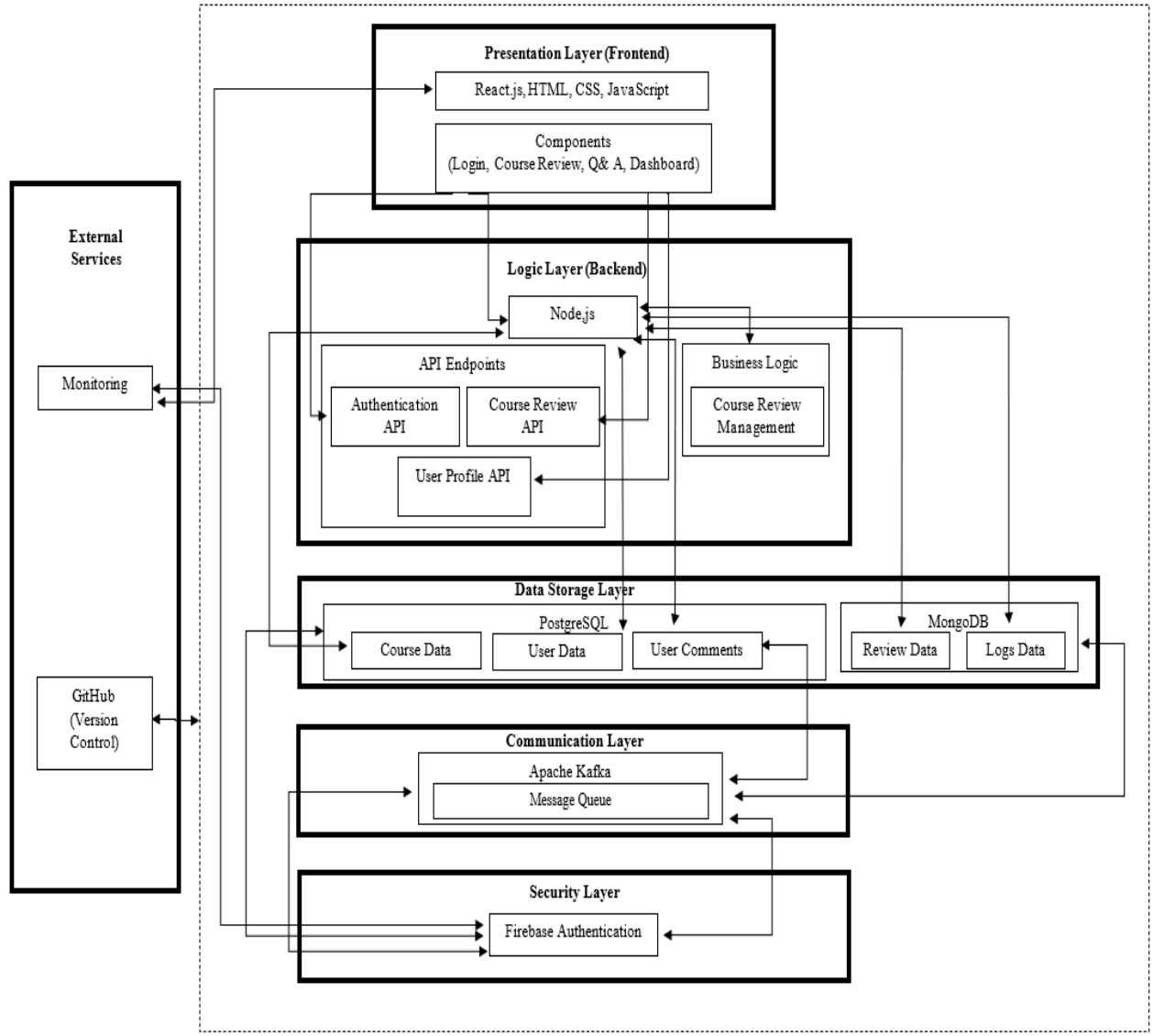
4. SOFTWARE COMPONENTS

- **Frontend**: React.js; HTML, CSS, JavaScript
- **Backend**: Node.js with Express.js
- **Database**: Cloud SQL (PostgreSQL)
- **Database Management**: MongoDB
- **Message Queue**: Apache Kafka
- **Authentication**: Firebase Authentication
- **Version Control**: Git and GitHub

5. BLOCK DIAGRAM:



6. ARCHITECTURAL DIAGRAM



7. INTERACTION OF COMPONENTS

7.1. Presentation Layer (Frontend)

- **Software**: Built using React.js, HTML, CSS, and JavaScript, this layer provides an interactive and responsive user interface. Users can log in, provide feedback, and rate courses.
- **Interaction**: The Presentation Layer communicates with the Logic Layer via secure APIs. When a user performs an action, such as submitting feedback, the frontend sends a request to the backend through these APIs. React ensures real-time updates and responsiveness.

- **Hardware**: The frontend runs on user devices (e.g., laptops, smartphones) that connect to the cloud-hosted backend services. These devices rely on the network infrastructure for smooth data transmission.

7.2. Database Layer

- **Software**: Cloud SQL (PostgreSQL) stores structured data like course details and user preferences, while MongoDB manages unstructured data like feedback and ratings.
- **Interaction**: The Logic Layer communicates with both databases to fetch, update, or store data. PostgreSQL handles relational data, ensuring consistency and integrity, while MongoDB provides flexibility to store dynamic content.

7.3. Logic Layer

- **Software**: The Node.js runtime environment, combined with the Express.js framework, powers the backend logic. This layer processes requests, applies business logic, validates data, and facilitates communication between the frontend and the database.
- **Interaction**: It acts as a mediator between the Presentation and Database Layers, ensuring smooth data flow. When a user submits feedback, the Logic Layer validates and stores it in MongoDB or Cloud SQL.

7.4. Security Layer

- **Software**: Firebase Authentication ensures that all users are authenticated securely. It supports various authentication methods, such as email/password or social login.
- **Interaction**: Firebase Authentication interacts with the Logic Layer to validate user credentials and manage secure sessions. It prevents unauthorized access by employing encrypted protocols.
- **Hardware**: The security protocols and session data are managed by Firebase's cloud infrastructure, which relies on secure data centers.

7.5. Communication Layer

- **Software**: Apache Kafka handles asynchronous communication between services, ensuring smooth queuing and processing of feedback data.
- **Interaction**: Kafka is integrated with the Logic Layer to handle tasks such as sending real-time notifications and managing feedback storage asynchronously. It ensures that feedback submission does not affect the performance or responsiveness of the platform.

- **Hardware**: Kafka operates on cloud-based servers that manage distributed message queues, ensuring high throughput and fault tolerance.

7.6. Overall Interaction

Each layer and component work together to ensure smooth user experience. The front end (Presentation Layer) sends user inputs to the backend (Logic Layer), which processes and stores data in the Database Layer. The Security Layer ensures safe access and authentication, while the Communication Layer (Kafka) guarantees smooth asynchronous processing. All components are hosted and run on cloud infrastructure, leveraging the scalability and reliability to manage hardware resources like servers, databases, and network devices. This architecture ensures modularity, with each layer focusing on specific tasks, while also providing flexibility, scalability, and security to handle growing user interactions and data volume.

8. DEBUGGING AND TESTING MECHANISM

To ensure the effective operation and reliability of the platform, we employed a comprehensive debugging strategy and rigorous testing mechanisms across different layers of the system.

8.1. Debugging Approach

- Log Analysis and Debugging- For debugging both the front end and backend, we leveraged Visual Studio Code (VS Code). The integrated terminal and debugging tools provided powerful features for tracking application performance, identifying runtime errors, and resolving issues quickly. The logging capabilities in VS Code allowed us to monitor the flow of requests, track error messages, and analyze exceptions efficiently, which helped us pinpoint performance bottlenecks and correct code-level issues in real-time.
- Frontend Debugging- To ensure the user interface (UI) was functioning as expected, we utilized browser developer tools. These tools allowed us to debug CSS rendering issues, inspect network requests, and resolve layout problems. They were crucial for detecting UI bugs, performance issues, and network failures (e.g., failed API calls), ensuring smooth user experience across different browsers and devices.
- Backend Error Handling- In the backend, we implemented try-catch blocks within the Node.js application to handle errors gracefully, preventing crashes and ensuring that the application could recover or provide meaningful error messages. For API endpoint

testing, we used Postman, which allowed us to manually test various API routes, checking the responses for correctness, status codes, and data integrity. Postman enabled us to simulate user requests and verify that the system met expectations under various scenarios.

- Kafka Message Monitoring- To monitor Apache Kafka and ensure the seamless flow of real-time data across services, we utilized Confluent Control Center. This monitoring tool provided us with insights into Kafka topics, message processing speeds, and potential backlogs. It helped us troubleshoot communication delays and data inconsistencies, ensuring that messages were processed correctly and promptly across all components of the system.

8.2. Testing Mechanisms

- Database Validation- For maintaining data integrity across the platform, we used Cloud SQL Query Editor (for PostgreSQL) to validate relational data, such as course details and user profiles. We wrote and executed queries to verify that all data met the expected schema and maintained consistency. In addition, we used MongoDB Compass for verifying the integrity of unstructured data, such as feedback, reviews, and comments, ensuring consistency and proper formatting of documents across MongoDB collections.
- Unit and Integration Testing- We conducted unit tests to verify the functionality of individual components, ensuring each piece of the system behaved as expected in isolation. For integration testing, we checked how components interacted within the larger system, simulating real user interactions with the APIs and backend services. Testing included ensuring the correct flow of data between the frontend, backend, and databases, and verifying that the authentication system worked seamlessly across all user scenarios.
- Load Testing and Performance Monitoring- To evaluate the scalability and reliability of the platform under heavy user load, we employed load testing tools, simulating multiple concurrent users interacting with the platform. We measured response times, server performance, and system reliability to identify potential bottlenecks. Performance monitoring tools were integrated into the system to continually track resource usage and system health.

8.3. Training Mechanisms

- Team Training on Tools and Best Practices- We ensured that the development team was well-versed in using debugging tools like Visual Studio Code, Postman, and browser developer tools through internal training sessions. This allowed team members to efficiently troubleshoot issues across both frontend and backend. We also provided training on best practices for error handling, testing strategies, and database validation to maintain code quality and prevent regressions.
- Continuous Learning and Code Reviews- Regular code reviews were conducted to ensure that best practices were being followed in error handling, data validation, and testing. This process facilitated knowledge sharing, allowed for identifying potential issues early in the development cycle, and ensured code quality remained high.

8.4. Summary

We utilized a combination of manual and automated debugging techniques, along with rigorous testing mechanisms across all layers of the system, to ensure the platform operated smoothly and reliably. These strategies included effective logging, frontend and backend error handling, real-time Kafka message monitoring, and thorough database validation. Additionally, the team participated in training to stay current on best practices and effectively leverage the tools and technologies used in the project.

9. WORKING SYSTEM, EXPECTED WORKLOAD AND CAPACITY, POTENTIAL BOTTLENECKS AND MINIMIZATION

9.1. Working System Overview

The platform is designed to facilitate anonymous course feedback collection and enhance transparency in the academic environment. The system is built on multi-layered architecture, with components working seamlessly to handle user requests, process feedback, and provide real-time interactions.

The home screen consists of a list of courses available and an option to login. We have a feature to filter the courses based on department and graduation level. Without logging, the comments provided are submitted as Anonymous. On logging, you can post the comment, and the name of the user gets recorded. The user has an option to check the course details, see its description, grad level and ratings. Apart from giving their feedback/comment, the user can see the already posted comments as well.

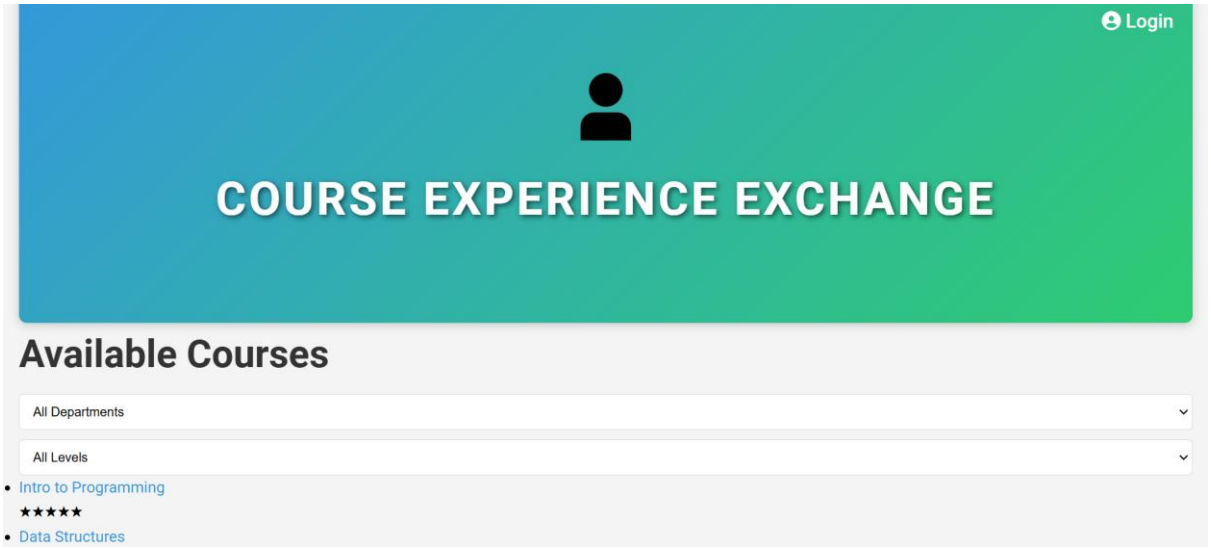


Fig. Home Page

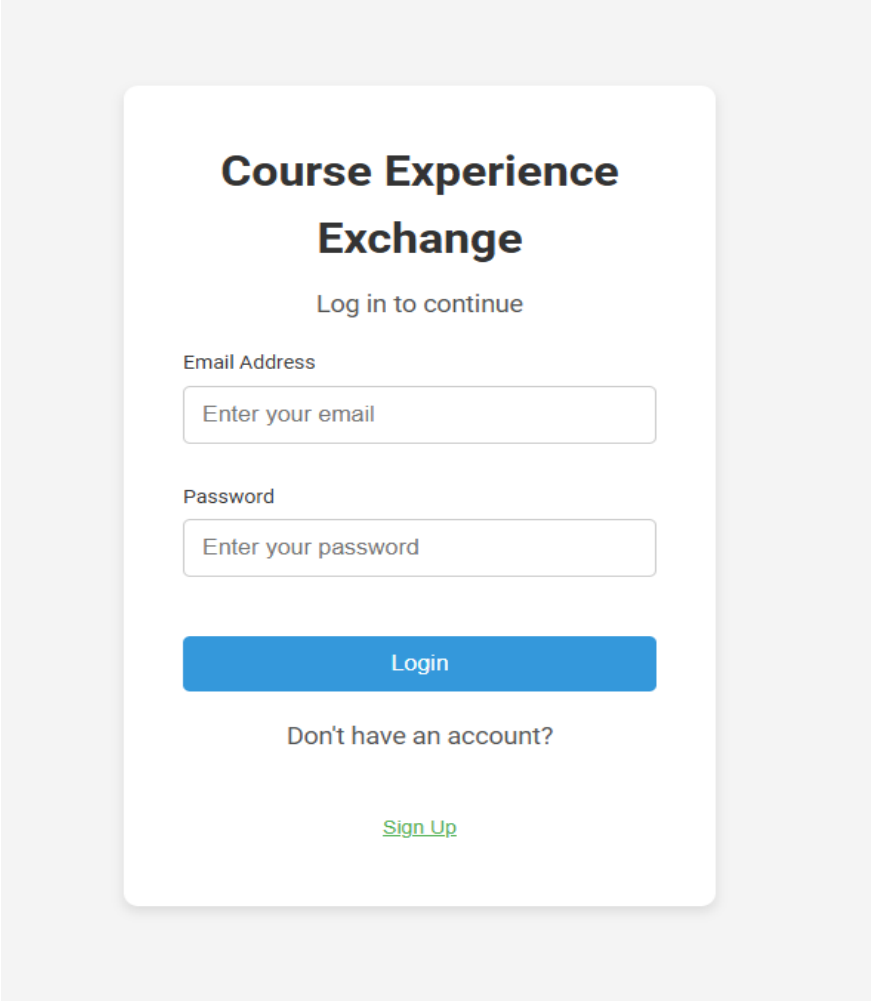


Fig: Login Page

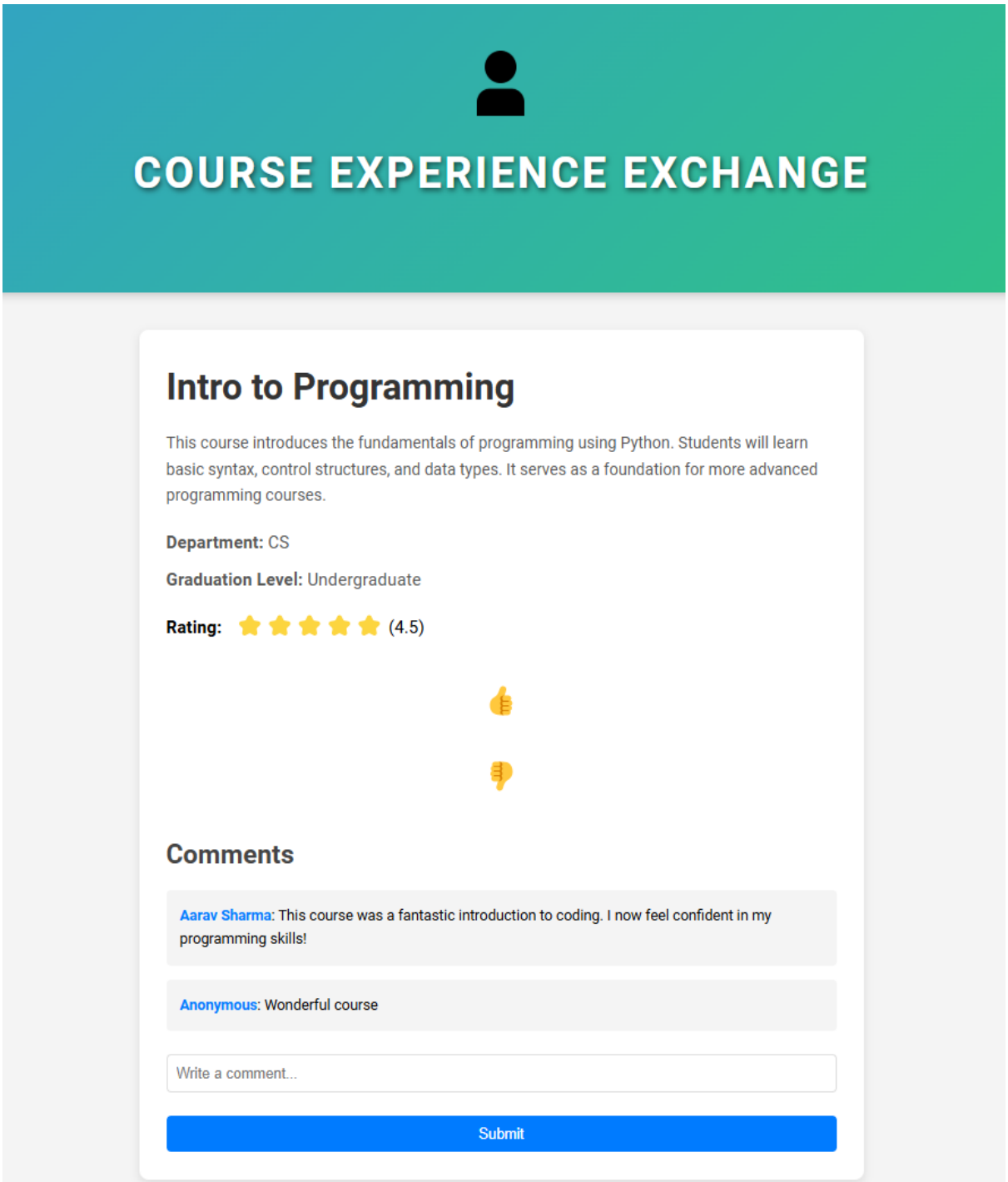


Fig: Course Details

9.2. Workload the System Can Handle

Scalability

- The use of MongoDB allows the system to efficiently handle unstructured data at scale, ensuring that new feedback and reviews can be added without performance degradation.
- Apache Kafka ensures asynchronous task handling and avoids blocking processes, allowing the system to handle high volumes of requests without degrading performance.

Real-Time Interactions

- The integration of Kafka for real-time messaging ensures that notifications and updates are processed and delivered quickly, even as traffic spikes.
- The system can handle multiple users submitting feedback simultaneously without blocking other actions, ensuring responsiveness.

User Growth

- The system is designed to scale with growing numbers of users, leveraging the flexibility of cloud-based services. This includes handling more user registrations, reviews, and feedback submissions without compromising performance.

9.3. Potential Bottlenecks

Despite its scalable design, the system may face bottlenecks in certain areas under specific conditions:

Database Performance (Cloud SQL and MongoDB)

- Cloud SQL (PostgreSQL): As the volume of structured data (course details, user profiles) grows, queries that involve complex joins or aggregate functions could slow down, especially if the database schema is not optimized for large-scale data. High read/write loads during peak usage times could also lead to slower response times.
- MongoDB: While MongoDB is designed to handle unstructured data efficiently, large-scale documents with heavy reads/writes could lead to performance degradation, especially if indexes are not properly configured or if the data model isn’t optimized for query patterns.

Real-Time Message Processing (Apache Kafka)

- While Kafka helps ensure asynchronous processing, heavy traffic and message queues could lead to delays if the Kafka cluster is not properly scaled. If message processing is delayed, it could impact on the timeliness of notifications, feedback processing, and other real-time interactions.

Backend (Node.js with Express.js)

- API Rate Limiting: The Node.js backend is designed to handle high traffic, but if there is a surge in requests, the server could be overwhelmed, especially if requests are not load-balanced or rate-limited. This can lead to delays in processing user requests, particularly when multiple users are interacting with the system simultaneously.

User Authentication (Firebase)

- High Login Load: During peak usage periods, many simultaneous logins might overwhelm the Firebase Authentication system, causing delays in user login or session creation. This can affect user experience, particularly if the platform sees a sudden surge in user registrations or logins.

9.4. Mitigation Strategies

To address these potential bottlenecks, the following strategies can be implemented:

1. Database Optimization: Indexing and query optimization for both Cloud SQL and MongoDB to ensure efficient data retrieval. Using caching mechanisms (e.g., Redis) to reduce the load on databases for frequently accessed data.
2. Kafka Scaling: Regularly monitoring Kafka clusters and scaling them based on traffic patterns to ensure that message queues are processed efficiently. Implementing consumer group architectures to balance the load and ensure quick processing of messages.
3. API Rate Limiting and Load Balancing: Implementing rate-limiting and load balancing mechanisms to distribute incoming requests across multiple backend instances and ensure consistent performance during high traffic periods.

4. Optimized Caching: Introducing caching layers at various points (e.g., API responses, frequent database queries) to reduce load on the backend and databases, ensuring faster response times.

By addressing these potential bottlenecks and implementing proper scalability and optimization techniques, the platform can handle high traffic and large workloads effectively, providing a seamless user experience even during peak usage times.