CS673 Software Engineering
Team 1 - BU Academic Navigator (BUAN)
Project Proposal and Planning

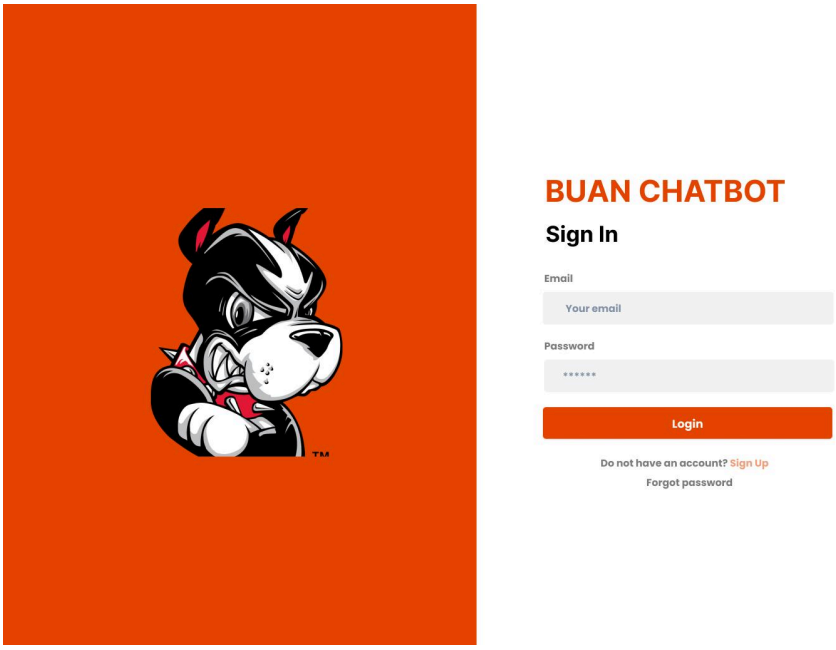
Team Member	Role(s)	Signature	Date
Natasya Liew	Team Leader	*Natasya Liew*	September 7, 2024
Natthaphon Foithong	Design and Implementation Lead	*Natthaphon Foithong*	September 9, 2024
Ananya Singh	Security Lead	*Ananya Singh*	September 8, 2024
Battal Cevik	QA Lead	*Battal Cevik*	September 8, 2024
Poom Chantarapornrat	Requirement Lead	*Chan P.*	September 7, 2024
Yu Jun Liu	Configuration Lead	*Yujun Liu*	September 8, 2024

Revision history

Version	**Author**	**Date**	**Change**
3.0.0	Poom Chantarapornrat	Oct 8, 2024	Added more detail in Requirements, Python AI service, and Frontend detail
3.0.1	Poom Chantarapornrat	Oct 14, 2024	Updated the requirement analysis section, formatted the document.
3.0.2	Ananya Singh	Oct 14, 2024	GitHub Security Features
3.0.2	Battal Cevik	Oct 14, 2024	Updated Testing Metrics
3.0.3	Natasya Liew	October 14, 2024	Final Proofreading and editing. Adding

			the Security measure against XSS

Chatbot Web Application



Team: BU MET CS Course Building Chatbot (Group 1)

Overview

This document presents the proposal and planning for the BUAN chatbot-driven web application designed to streamline the course selection process for students in the BU MET Computer Science program. Utilizing cutting-edge technologies, this web application integrates OpenAI's ChatGPT-4o mini via the Langchain API to offer personalized course recommendations tailored to individual student needs and academic pathways.

The application leverages a robust backend developed in Java using the Spring Boot framework, coupled with a dynamic frontend built with React. Python is utilized for integrating the AI capabilities, ensuring that students receive relevant advice and guidance based on their unique academic histories. Data management is handled via a PostgreSQL database, created to compensate for the unavailability of the BU Registration MS database, which stores comprehensive program and course information.

The core functionality of the BUAN chatbot includes features such as chat history sharing, email integration, caching, and secure user authentication through JWT Authentication, ensuring a seamless and secure experience for users. By implementing a decision tree algorithm, the system effectively navigates course prerequisites and offers insights into elective options, empowering students to make informed choices as they plan their academic journey.

This project aims not only to enhance the user experience for students but also to provide a scalable and flexible platform that can adapt to the evolving needs of academic advising.

Related Work

1. **Coursera**: Coursera leverages collaborative filtering for course recommendations based on user history. In contrast, our system utilizes the GPT model integrated with real-time, AI-driven conversations, offering personalized recommendations powered by PostgreSQL-stored data and a decision tree algorithm.

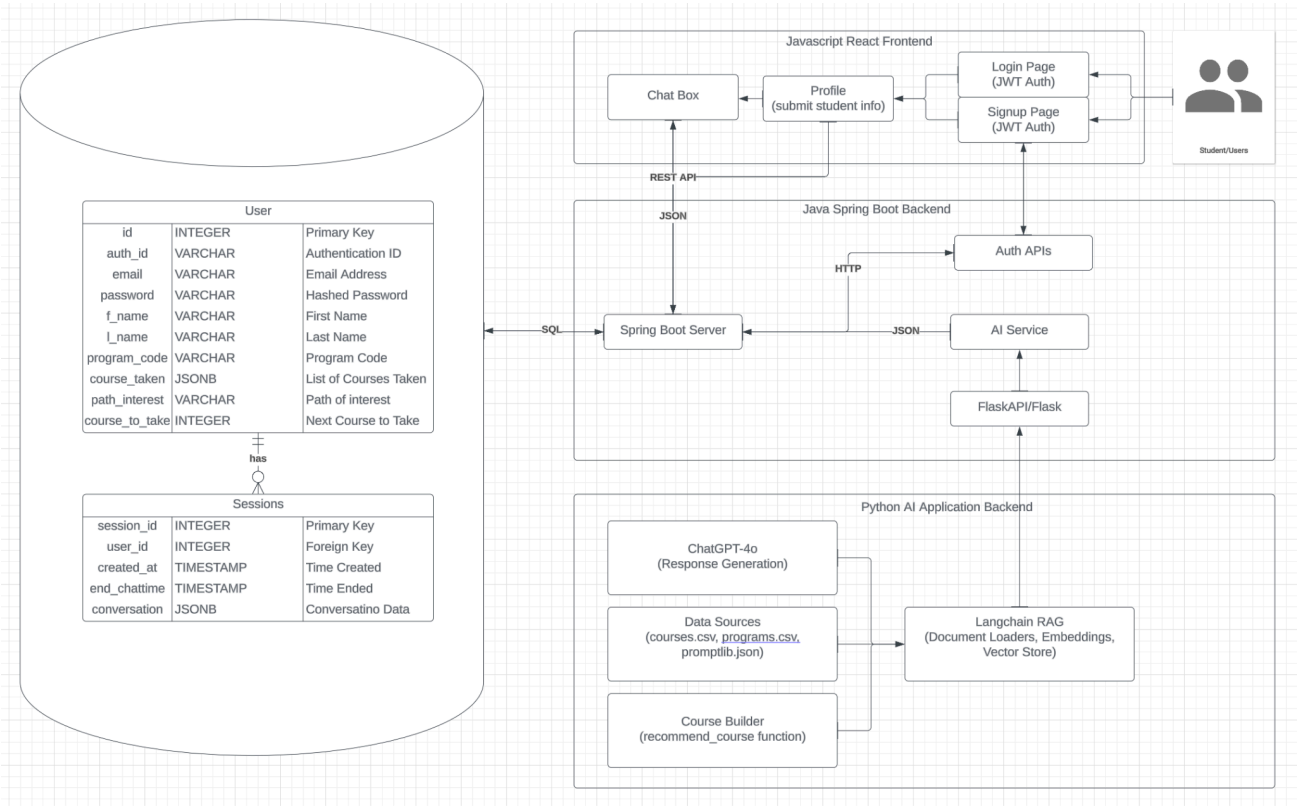
2. **EdX Chatbot**: While EdX employs a rule-based chatbot for basic queries, our advanced chatbot, using the GPT model, delivers highly personalized course recommendations through interactive, natural language conversations, enhancing user engagement and decision-making.

3. **AI Chatbots for Higher Education (Ada, AdmitHub)**: These systems focus on general student services like admissions and support. Our chatbot specifically addresses course selection, storing chat history in PostgreSQL, and offering features such as history-sharing, enhancing both personalization and user experience.

4. **Degree Compass**: Degree Compass uses predictive analytics for course recommendations. Our system combines dynamic, real-time AI responses with program-specific data and user inputs, providing a more flexible and interactive course recommendation experience.

Proposed High-level Requirements

High-level design:



Functional Requirements

Essential Features

1. Natural Language Interaction

- **User Story:** As a user, I want to interact with the platform by using chat messages rather than clicking buttons, so that I can conveniently take a consultation from the agent.
- **Process Breakdown:**
    1. User types a message in the chat interface.
    2. The system processes the natural language input using NLP algorithms.
    3. The system determines the user's intent and retrieves appropriate responses or actions.
    4. The system sends a reply to the user in the chat interface.
- **Acceptance Criteria:**
    1. **Given** a natural language input, **when** the user submits it, **then** the system interprets the request with at least 90% accuracy.
    2. **Given** various input formats, **when** the user asks for assistance, **then** the system responds appropriately.
    3. **Given** common phrases, **when** the user submits them, **then** the system recognizes and processes them.
    4. **Given** an interruption in conversation, **when** the user continues, **then** the system maintains context.
    5. **Given** an unrecognized query, **when** the user submits it, **then** the system provides a fallback response.

2. Provide Course Description

- **User Story:** As a student, I want to read the course description given a course number before I enroll, so that I can decide whether the course is right for me.
- **Process Breakdown:**
    1. User inputs the course number in the chat.
    2. The system retrieves the course description from the vector storage.
    3. The system formats the description for readability.
    4. The system sends the formatted description back to the user.
- **Acceptance Criteria:**
    1. **Given** a valid course number, **when** the user requests the course description, **then** the system displays the description within less than 5 seconds.
    2. **Given** a valid course number, **when** the course description is retrieved, **then** it includes prerequisites and relevant notes.
    3. **Given** multiple course requests, **when** the user submits them, **then** the system processes and responds to all requests in one interaction.
    4. **Given** an invalid course number, **when** the user requests a description, **then** the system provides a user-friendly error message.

5. **Given** course descriptions are updated in the database, **when** the user requests a description, **then** the system reflects the latest information.

3. Generate Class Schedule

- **User Story:** As a student, I want to have a personalized course schedule based on my completed courses, so that I can fulfill prerequisites and graduate on time.
- **Process Breakdown:**
    1. User logs in or signs up and provides information about completed courses, path of interest, and the number of courses wanted to take.
    2. The system checks for unmet prerequisites based on the user's program.
    3. The system generates a list of recommended classes for the upcoming semester.
    4. The system presents the schedule in a natural language format.
- **Acceptance Criteria:**
    1. **Given** completed courses, path of interest, and the number of courses to take, **when** the user requests a class schedule via text message, **then** the system responds and displays it within 3 seconds.
    2. **Given** a generated schedule, **when** the user reviews it, **then** it must include the class information and all necessary prerequisites for each course.
    3. **Given** the chat interface, **when** the user requests the recommended class schedule, **then** the system generates the recommendation at any time in the conversation.
    4. **Given** a user-requested course that is unavailable, **when** the schedule is generated, **then** the system notifies the user.

4. **Answering Questions about Courses within the Program**

- **User Story**: As a student, I want to ask questions about the courses offered within my program, so that I can make informed decisions about my studies.
- **Process Breakdown**:
    - User initiates a question about program courses.
    - The system retrieves relevant course data and prerequisites.
- **Acceptance Criteria**:
    - **Given** the user asks about specific courses, **when** they submit their question, **then** the system should return accurate information.
    - **Given** the user inquires about prerequisites, **when** they request details, **then** the system should provide clear guidance on necessary courses.
    - **Given** the user is considering elective options, **when** they ask for recommendations, **then** the system should suggest electives based on their program.

5. **Courses Provided in the CS Department**

- **User Story**: As a student, I want to know the courses offered in the Computer Science department, along with their prerequisites that align with my personal interests.
- **Process Breakdown**:

    - User types in a question about classes related to their personal interest, for example, “what are the classes that are related to artificial intelligence?”
    - The system retrieves and displays the list of courses that relate to the topic of interest along with their prerequisites.
- **Acceptance Criteria**:
    - **Given** a chat box interface, **when** the user requests for some course information related to a topic of interest, **then** the system should return a comprehensive list of related courses.
    - **Given** the courses are displayed, **when** the user asks about more course information, **then** the system should provide detailed information for each course.

Desirable Features

6. Chat History and Caching Consolidation

- **User Story:** As a student, I want to access my previous interactions with the BUAN chatbot so that I can conveniently review the information provided during my prior sessions.
- **Process Breakdown:**
    1. User requests access to their chat history.
    2. The system retrieves saved conversations from the database.
    3. The system caches retrieved chat history to ensure faster future access.
    4. The system displays the chat history in an easy-to-read format.
- **Acceptance Criteria:**
    1. **Given** the user requests chat history**, when** the system retrieves it**, then** all previous interactions should be displayed accurately and in chronological order based on timestamp**.**
    2. **Given** chat history is requested**, when** the system retrieves it**, then** it takes no longer than 2 seconds to load**.**
    3. **Given** the chat history is deleted**, when** the user tries to find it again**, then** the system returns a `no result found`**.**

7. Data Caching and Chat History Integration

- **User Story:** As a student, I want frequently accessed data (e.g., course descriptions, chat history) to load quickly so that I can easily obtain the information I need without waiting.
- **Process Breakdown:**
    1. The system caches frequently accessed data such as chat sessions, user profile, and chat history.
    2. Upon subsequent requests, the system retrieves data from the cache.
    3. The cache is updated whenever there are changes to the underlying data in the database.
    4. Cached data is provided to the user in a readable format for convenience.
- **Acceptance Criteria:**

1. **Given** cached data is available**, when** the user revisits the site**, then** the data loads within 1 second.
2. **Given** new data is available**, when** the user accesses it**, then** the system updates the cache accordingly to reflect the most current information.
3. **Given** the cache limit is reached**, when** new data is added**, then** the oldest cached data is removed.
4. **Given** cached chat history is requested**, when** the system retrieves it**, then** all previous interactions are displayed in chronological order based on timestamp.

8. Student Status Inquiry

- **User Story**: As an international student, I want to inquire about my visa status and related academic requirements so that I can ensure compliance with regulations.
- **Process Breakdown**:
    - User selects the option to inquire about student status.
    - The system retrieves information related to F-1 and J-1 visa requirements.
    - Relevant academic policies are presented to the user.
- **Acceptance Criteria**:
    - **Given** the user selects student status inquiry, **when** they submit their visa type, **then** the system should display the relevant academic requirements for that visa.
    - **Given** the user is an F-1 student, **when** they inquire about work limitations, **then** the system should provide accurate information on work eligibility.
    - **Given** the user is a J-1 student, **when** they ask about academic requirements, **then** the system should detail specific course or credit hour requirements.
    - **Given** the inquiry is made, **when** the user finishes reading, **then** they should be able to ask follow-up questions seamlessly.
    - **Given** the inquiry is related to visa requirements, **when** the user submits the inquiry, **then** the system should save the chat history for future reference.

9. Course Drop Dates Information

- **User Story**: As a student, I want to know the important dates for dropping courses so that I can avoid academic penalties.
- **Process Breakdown**:
    - User accesses the course drop inquiry section.
    - The system retrieves current academic calendar information.
    - Drop dates with or without a 'W' grade are displayed.
- **Acceptance Criteria**:
    - **Given** the user selects course drop inquiry, **when** they submit their request, **then** the system should show upcoming drop dates.
    - **Given** the user specifies 'W' or 'without W', **when** they request drop dates, **then** the system should display accurate corresponding deadlines.
    - **Given** the user is considering dropping a course, **when** they ask about consequences, **then** the system should explain potential impacts on their transcript.

    - **Given** the inquiry is made, **when** the user finishes reading, **then** they should be able to access their chat history for future reference.
    - **Given** the user submits the request, **when** they return to the chat, **then** the system should present related resources or contacts for further assistance.

10. Graduation Timeline Inquiry

- **User Story**: As a student, I want to know how to graduate before a specific semester so that I can plan my course load effectively.
- **Process Breakdown**:
    - User selects the graduation timeline inquiry feature.
    - The system checks the user's completed courses and remaining requirements.
    - Recommendations for completing requirements by the desired semester are provided.
- **Acceptance Criteria**:
    - **Given** the user inputs their desired graduation semester, **when** they request a timeline, **then** the system should show a list of remaining requirements.
    - **Given** the user's course history, **when** they inquire about accelerated graduation, **then** the system should suggest a feasible course load.
    - **Given** the user has specific courses in mind, **when** they request alternatives, **then** the system should provide options to meet requirements within the desired timeline.
    - **Given** the graduation timeline inquiry, **when** the user receives their recommendation, **then** they should be able to save or print the details.
    - **Given** the inquiry is related to graduation, **when** the user submits the request, **then** the system should log the interaction in chat history.

11. Program Change and Course Inclusion Inquiry

- **User Story**: As a student considering a program change, I want to see which courses will transfer to my new degree so that I can make informed decisions.
- **Process Breakdown**:
    - User accesses the program change inquiry feature.
    - The system retrieves information on course transferability.
    - A list of applicable courses is generated.
- **Acceptance Criteria**:
    - **Given** the user selects a new program, **when** they inquire about course transfers, **then** the system should display applicable courses.
    - **Given** the user provides their current course history, **when** they request a comparison, **then** the system should identify which courses will be accepted in the new program.
    - **Given** the user is changing programs, **when** they submit the request, **then** the system should offer advice on additional requirements for the new program.
    - **Given** the course transfer inquiry, **when** the user completes their request, **then** they should receive confirmation of their inquiry being saved for later reference.

    - **Given** the user is unsure about the implications of changing programs, **when** they inquire, **then** the system should present information about potential impacts on graduation timelines.

12. Course Transfer from Other Institutions

- **User Story**: As a student transferring from another institution, I want to know which of my previous courses can be credited towards my current program so that I can maximize my course credits.
- **Process Breakdown**:
    - User inputs previous courses for evaluation.
    - The system checks against the current program's requirements.
    - Transferable courses are displayed to the user.
- **Acceptance Criteria**:
    - **Given** the user submits previous course details, **when** they request a transfer evaluation, **then** the system should display transferable courses.
    - **Given** the user's previous institution and course descriptions, **when** the evaluation is complete, **then** the system should specify which courses are accepted.
    - **Given** the user is inquiring about transfer credits, **when** they ask about the process, **then** the system should provide information on documentation needed for credit transfer.
    - **Given** the inquiry is related to transfer credits, **when** the user submits their request, **then** the system should log the inquiry for follow-up.
    - **Given** the user is unsure about transfer eligibility, **when** they submit questions, **then** the system should guide them through potential eligibility criteria.

Optional Features

13. Smooth Text Transition

- **User Story**: As a chat user, I want to see the text message smoothly appear letter by letter rather than a big chunk at once, so that I can read along.
- **Process Breakdown**:
    - The chat system implements a typing effect for message delivery.
    - Text appears gradually to enhance user engagement.
- **Acceptance Criteria**:
    - **Given** a new message is sent, **when** the system displays it, **then** the text should appear letter by letter.
    - **Given** the user is reading, **when** the typing effect is active, **then** they should be able to pause the text display at any time.

    - **Given** the user is engaged in conversation, **when** a message is sent, **then** the transition should maintain a natural flow.
    - **Given** the user prefers rapid messages, **when** they adjust settings, **then** they should be able to toggle the typing effect on or off.
    - **Given** the user finds the effect distracting, **when** they provide feedback, **then** the system should allow for customization of this feature.

Nonfunctional Requirements

1. Security Requirements

Chat Privacy

- **User Story:** As a student, I want to make sure no one can see my chat messages so that I can be more comfortable chatting with privacy.
- **Process Breakdown:**
    1. The system encrypts chat messages in transit and at rest.
    2. User authentication is required to access chat history.
    3. Unauthorized access attempts are logged and flagged for review.
- **Acceptance Criteria:**
    1. **Given** chat messages are sent, **when** they are stored, **then** they are encrypted using industry-standard protocols.
    2. **Given** a user logs in, **when** they try to access chat history, **then** they can only see their own messages.
    3. **Given** an unauthorized access attempt occurs, **when** it is detected, **then** the system triggers an alert.
    4. **Given** a successful login, **when** the user accesses the chat, **then** they receive a notification.
    5. **Given** inactivity, **when** the session times out, **then** the user is logged out automatically.

2. Protect Sensitive Information

- **User Story:** As a user, I do not want to expose my password to other people so that I can be safe from malicious hackers.
- **Process Breakdown:**
    1. The system uses hashing to store passwords securely.
    2. Multi-factor authentication is implemented for added security.
    3. Users receive alerts for suspicious login attempts.
- **Acceptance Criteria:**
    1. **Given** a password is created, **when** it is stored, **then** it is hashed using strong algorithms (e.g., bcrypt).
    2. **Given** multi-factor authentication is enabled, **when** the user logs in, **then** they must complete the second verification step.

3. **Given** a login attempt occurs, **when** it is successful, **then** the user receives a notification.
4. **Given** multiple failed login attempts, **when** the threshold is reached, **then** the account is locked temporarily.
5. **Given** a password reset is requested, **when** the user verifies their identity, **then** they can set a new password securely.

3. **Performance and Scalability**

- **User Story**: As a user, I want the web app to handle multiple concurrent users without performance degradation.
- **Acceptance Criteria**:
    - **Given** peak usage times, **when** multiple users access the platform, **then** response times should remain under 2 seconds.
    - **Given** increased data load, **when** the user queries information, **then** the system should utilize caching to maintain performance.
    - **Given** the growth of the user base, **when** system architecture is evaluated, **then** scalability options should be presented for future upgrades.

4. Usability

- **User Story**: As a user, I want the interface to be intuitive and easy to navigate so that I can find information quickly.
- **Acceptance Criteria**:
    - **Given** the user is new to the platform, **when** they access the app, **then** they should be able to complete a task within three clicks.
    - **Given** the user encounters a problem, **when** they seek help, **then** the system should provide contextual assistance or FAQs.
    - **Given** the design of the interface, **when** the user interacts with elements, **then** they should have clear visual feedback on their actions.

Management Plan

Objectives and Priorities

Objective 1: Front-End Development (Front-End Engineer) – 2 Members

Priority 1: Deliver a responsive user interface using React.js, HTML, and CSS.

- **Natt** will design the frontend to align with the **Essential Feature: Course Selection and Program Browsing**, ensuring users can easily interact with the system.
- **Poom** will assist **Natt** to complete the UI components and pages, as **Python AI service functionalities** are complete for now.
- Focus on building a **modular and reusable component-based structure** to support features such as **Natural Language Interaction** and **Chat History Retrieval**.
- **Ensure UI integrates seamlessly** with backend services, following modern design principles, to support features such as **Student Status Inquiry** and **Graduation Timeline Inquiry**.

Objective 2: Back-End Development (Java Team) – 2 Members

Priority 1: Set up the foundational server architecture using Spring Boot.

- **Ananya** to lead backend development and integration with PostgreSQL, focusing on **Course Data Storage** and **User Authentication**, supporting features such as **Course Transfer from Other Institutions**, which is accessed through the **Python AI Service**.
- Establish **skeleton code for the server structure**, including **routing, controller, and service layers**, allowing for efficient interaction between front-end and AI services.

Priority 2: Collaboratively implement PostgreSQL with the AI team.

- **Ananya** will take the lead in integrating **PostgreSQL**, working closely with **Natasya** to update database schemas based on updated design.
- **Ensure smooth database integration** for user authentication, chat history, course data storage, and AI interaction, covering essential requirements such as **Chat History and Caching**.

Objective 3: CI/CD Integration and Testing (QA Lead and Mentorship) – 2 Members

Priority 1: Implement CI/CD pipelines using GitHub Actions.

- **Battal** will handle CI setup, using **GitHub Actions** to implement a seamless deployment pipeline.
- **Battal** will create all test cases based on the requirements and map with them.
- **Battal** will also mentor **AJ** on setting up **testing, linting, and formatting** to support best practices.

- **Create and configure a YML file** for automated builds, testing, and deployment, ensuring **QA processes** cover functional, regression, and integration testing to maintain stability for features like **Course Drop Dates Information** and **Graduation Timeline Inquiry**.

  Priority 2: Integrate Docker for consistent development and production environments.

- **Battal** will handle **Docker maintenance** to ensure the system's environment consistency, improving stability for features such as **Natural Language Interaction** and backend services.
- **Battal** will create an automation test framework using BDD cucumber and selenium for UI and rest assured for API.

  Priority 3: Deployment to the production environments and testing

- **Battal** will deploy application to the AWS EC2 instance and test it
- **Battal** will create a branching strategy to implement 3 branches, develop, stage and production.
- **Battal** will create cucumber feature files in order to test respective requirements
- Collaborate between QA and Backend Teams to ensure Docker environments meet testing and production needs.

Objective 4: AI Service Development (AI Team) – 2 Members

Priority 1: Build AI services using FastAPI integrated with GPT-4.

- **Natasya** will mentor **Poom** to guide the development of AI services, providing **static course builder code** and **drafting integration** with Langchain.
- The AI service will support essential features like **Answering Questions about Courses**, **Course Builder**, and **Course Transfer from Other Institutions**.
- Develop basic **question-answering capabilities** and **course recommendations** based on user inputs.

Priority 2: Develop a fallback Flask-based back-end.

- Ensure a **failover mechanism** is in place, maintaining chatbot functionality if the primary API encounters issues.
- If time allows during **Iteration 3**, the **Course Builder** code will be made **more modular** to facilitate future scalability and ease of updates.
- **Natasya** will oversee documentation updates and act as the **Technical Project Manager**, coordinating across teams to ensure smooth development and integration of AI capabilities.

Objective 5: Technical Project Management and Integration Supervision (TPM) – 1 Member

Priority 1: Coordinate Across Teams

- **Natasya** will take on the **Technical Project Manager** role, supervising the integration between services and covering any critical gaps to ensure timely delivery of a functional product.
- **Ananya** will oversee **integration between services**, focusing on **Spring Boot**, **database interaction**, and **security aspects** like **3rd-party authentication (Okta)** and **CAPTCHA** for enhanced security.
- **Natasya** will supervise **documentation**, ensuring that it aligns with the **functional requirements**, and providing updates based on progress and feature implementation to support clear and accurate development communication.

Summary of Changes and Focus Areas

- **Mentorship and Team Roles**: Emphasized mentor-mentee relationships to leverage experienced members like **Natasya**, **Battal**, and **Ananya** to guide less experienced team members.
- **Clear Feature Links**: Made explicit references to functional and nonfunctional features throughout the development process to ensure **Objectives and Priorities** are directly supporting each feature in the SPPP.
- **Integrated Supervision and Communication**: Assigned **Natasya** as **TPM** to enhance coordination across teams, supervise documentation, and ensure the focus remains on delivering a cohesive product with a clear user journey.

These updates ensure all features are properly aligned with their corresponding objectives, with added clarity on responsibilities, mentorship, and integration, resulting in a more cohesive plan for completing the project effectively.

Risk Management

Identified Risks

1. Team Engagement & Output

- **Risk:** Lack of motivation, procrastination, inconsistent work output, or team members dropping out could hinder progress and lead to missed deadlines.
- **Mitigation Strategy:** Promote frequent communication via Discord and task reassignment to maintain balanced workloads. Implement a buddy system and mentorship by experienced members like **Natasya**, **Ananya**, and **Battal** to boost accountability and motivation. Daily standups or weekly check-ins will be held to track progress, identify challenges, and prevent

burnout. If a member experiences an unexpected emergency or drops the course, the project scope will be redefined, and tasks reassigned.

2. Scope Creep & Requirement Ambiguity

- **Risk:** Unclear requirements, frequent changes, or uncontrolled expansion of project scope could lead to delays and inefficiencies.
- **Mitigation Strategy:** Create and maintain a **requirements checklist** that will be reviewed and signed off before development starts. Team members should feel confident in pushing back on unclear requirements, and changes to requirements will be restricted to **once per iteration**. During bi-weekly meetings, **Natasya** will promote open communication to address and prevent requirement ambiguity.

3. Workload Management Using Jira

- **Mitigation Strategy:** Use **Jira Epics, Child Issues, and Labels** (e.g., Icebox, Urgent) to track and prioritize work. Assign tasks based on individual strengths, taking into account team members' varying experience levels. Experienced members like **Ananya** and **Natasya** will supervise less experienced team members, and **Battal** will mentor **AJ** on best practices in QA and CI, ensuring all tasks are clearly defined as **user stories** and **tasks** to facilitate effective workload distribution.

4. Integration & CI/CD Workflow Failures

- **Risk:** Integration challenges, failed CI pipeline runs, or inconsistent development and production environments could cause delays and regression.
- **Mitigation Strategy:** Start integration early, using **smaller, testable increments**. **Battal** will handle **Docker maintenance, AWS EC2 deployment**, for consistent development and production environments (with **Ananya** will support if necessary) and will oversee the **CI/CD pipeline** (while mentoring **YuJun (AJ)** as executor). Allocate extra time during each iteration to resolve integration issues and optimize the pipeline for efficiency.

5. Testing & Quality Control

- **Risk:** Insufficient testing could lead to undetected bugs or issues in production, reducing system reliability.
- **Mitigation Strategy:** Enforce **mandatory unit, integration, and end-to-end testing** before merging any code. **Battal** will ensure that all team members follow a structured **test automation framework** as part of the CI/CD pipeline. **AJ** will assist with QA efforts under **Battal's** guidance, covering functional, regression, and integration testing to maintain code quality.

6. Technology Learning Curve

- **Risk:** Team members may face challenges using new technologies like **Docker**, **FastAPI**, **PostgreSQL**, **GPT-4**, or **Spring Boot**, leading to delays in task completion.
- **Mitigation Strategy:** Provide **resources, tutorials, and mentorship** to upskill team members. **Natasya** will guide **Poom** on features to integrate using **LangChain** for the AI services, and **Ananya** will focus on **Spring Boot** and database integration. Team members with experience will lead **knowledge-sharing sessions** for unfamiliar tools. Assign tasks based on individual strengths to balance the learning curve and maintain productivity.

7. Communication Risks

- **Risk:** Miscommunication, avoidance of issues, or poor collaboration can lead to duplicate work, missed deadlines, or incomplete tasks.
- **Mitigation Strategy:** Promote **open communication** during bi-weekly meetings, encourage team members to voice concerns, and make use of Discord for updates on progress. **Natasya**, as the **Technical Project Manager**, will ensure effective coordination across teams, minimize miscommunication, and ensure proper documentation is maintained.

8. Unbalanced Work Distribution

- **Risk:** Some team members may take on more work than others, leading to burnout or reduced quality of work output.
- **Mitigation Strategy:** Encourage **equal task distribution** and have experienced members like **Natasya** and **Ananya** supervise workload balance. If certain members volunteer for extra tasks, efforts will be made to **motivate and support** them while ensuring manageable workloads for all team members.

9. Integration Challenges for OpenAI API Access

- **Risk:** Single access to the **OpenAI API key** owned by **Poom** could create a bottleneck, limiting testing opportunities for AI-related services.
- **Mitigation Strategy: Poom** will share access and coordinate usage schedules for the **API key** to minimize downtime. Developers will code with minimal dependencies, reducing attempts to repeatedly test functionality. Poom's active participation ensures a minimized risk of blocking progress.

10. Personnel Availability and Emergencies

- **Risk: Unexpected emergencies** or **busy work schedules** might limit team members' ability to meet project deadlines.
- **Mitigation Strategy:** Prepare contingency plans for task reassignment if a member becomes unavailable due to emergencies. The **team leader** will redefine project scope if necessary and assign pending tasks to available members to maintain project progress.

By proactively addressing these risks and assigning clear roles, responsibilities, and mitigation measures, the team aims to ensure a smooth and successful development process focused on collaboration, effective learning, and efficient problem-solving.

a. Timeline

Iteration	Functional Requirements(Essential/ Disable/Option)	Tasks (Cross requirements tasks)	Estimated/real person-hours
1	Essential functions should achieved	Working Web App with UI, running server, responsive AI model.	20-30 hours for each team
2	Add on features	Add security(auth), UI, DB, complete functionality and features for the Python AI service, and services for auth	30-40 hours for each team
3	Fully tested and production-ready	Bug-free completed Web App production. Added security layer.	20-30 hours for each team

Progress Report Sheet CS673_ProgressReport_team1_iteration2.xlsx
Detailed Risk Management Sheet **CS673_SPPP_RiskManagement_team1_iteration2.xlsx**

Configuration Management Plan

Tools

- **Git**: Version control for source code management. The team will use GitHub as the primary platform for collaborative development, issue tracking, and pull requests.
- **CI/CD Tools**: Continuous Integration and Continuous Deployment (CI/CD) will be managed through GitHub Actions. This automates the testing, building, and deployment processes, ensuring that code changes are validated before reaching production.
- **Jira**: Jira will be used for agile project management. The team will create epics, tasks, and user stories for each deliverable, providing transparency on progress, priorities, and timelines.
- **Docker**: Docker will be used to containerize the application, ensuring consistency between the development and production environments. Docker images will be deployed across all stages of development, reducing dependency-related issues.
- **Discord**: Discord will serve as the main communication platform for daily stand-ups, informal discussions, and real-time collaboration within the team. It will also be used to maintain logs of conversations and decisions.

Code Commit Guideline and Git Branching Strategy

For this project, we will adopt the **Git Flow** branching strategy to promote efficient and collaborative development, ensuring stability in the production code.

- **Main Branch**: This branch will contain stable, production-ready code at all times. Any code merged into this branch should be fully tested, reviewed, and approved. This branch will always represent the latest functional version of the chatbot.
- **Stage Branch**: The stage branch will serve as the pre-production environment. It will be used to validate features in a production-like environment, ensuring that the application behaves correctly before merging into the main branch.
- **Development Branch**: The development branch will be the integration point for all new features. It will contain the latest in-progress features that are under active development by the team.
- **Feature Branches**: Each team member will work on their own feature branch, based on the development branch. Feature branches will focus on specific functionalities or improvements.
    - **Naming Convention**: feature/<member-name>/<feature-description> (e.g., feature/john-doe/course-recommendations).

Pull Requests (PRs)

- **PRs to Development Branch**: All feature branches will be merged into the development branch via pull requests (PRs).
- **Mandatory Code Reviews**: Each PR must be reviewed and approved by at least one team member before being merged.
- **Testing and Documentation**: PRs must include unit tests and proper documentation. Additionally, all tests in the CI/CD pipeline must pass before merging.

Deployment Plan

The chatbot application will follow a multi-phase deployment plan to handle front-end, back-end (API), and AI components, ensuring smooth integration and operation.

1. **Front-end Deployment**:
    - **Design and Implementation**: The front-end will be built using **React.js**, and will be responsible for handling user interactions and rendering UI components.
    - **Related Tools**:
        - React.js
        - Axios or Fetch API for handling HTTP requests
        - React Router for navigation
        - Context API for state management
        - Material-UI or Bootstrap for UI design
    - **Deployment Strategy**: The front-end will be deployed using **Docker** and integrated with the backend services.
2. **Back-end (API) Deployment**:
    - **Design and Implementation**: The back-end API will be developed using **Java (Spring Boot)**. It will handle business logic, data retrieval, user authentication, and API endpoints.
    - **Related Tools**:
        - Spring Boot for the core framework
        - Spring Security for authentication and authorization
        - Hibernate for ORM and database interaction
        - PostgreSQL for data storage
        - Flyway or Liquibase for database migration
        - Swagger/OpenAPI for API documentation
        - REST Assured for API testing
        - Docker for containerization
    - **Deployment Strategy**: The back-end services will be containerized using Docker and deployed on a cloud environment, ensuring scalability and fault tolerance.
3. **Back-end (AI Service) Deployment**:
    - **Design and Implementation**: The AI service, responsible for course recommendations and chatbot responses, will be implemented using **Python**. It will interact with the back-end API to provide intelligent responses based on user input.

- **Related Tools**:
  - TensorFlow or PyTorch for the AI model
  - Flask or FastAPI for serving the AI model through a web API
  - Gunicorn for WSGI server deployment
  - Redis for caching and queue management
  - Celery for asynchronous task processing
  - Docker for containerization
- **Deployment Strategy**: The AI service will be deployed as a separate containerized service, allowing it to scale independently based on user demand.

Best Practices for Deployment

1. **Continuous Integration**: All changes to the codebase will be automatically tested through GitHub Actions. This ensures that any new feature or bug fix does not introduce regressions.
2. **Continuous Deployment**: After passing all automated tests, the application will be deployed to a staging environment for final validation before being deployed to production.
3. **Multi-Phase Rollout**: The deployment process will follow a phased rollout plan, ensuring that each component (front-end, API, AI) is fully functional before moving to the next.
4. **Monitoring and Logs**: Post-deployment, we will implement monitoring and logging tools such as **ELK Stack (Elasticsearch, Logstash, Kibana)** to track application performance and capture error logs.

By following this structured approach, the chatbot application will be effectively containerized, tested, and deployed across multiple environments, ensuring a smooth user experience and high-quality software delivery.

Quality Assurance Plan

The Quality Assurance (QA) plan outlines the comprehensive testing strategy, objectives, tools, and processes to ensure that the BUAN chatbot web application meets the highest quality standards. The QA approach integrates both **manual**, **automation**, and **regression** testing, focusing on functional, integration, regression, and performance testing to validate the application's functionality, reliability, and performance.

To support a continuous delivery model, the QA process is tightly coupled with our deployment pipeline. The application is deployed to an **AWS EC2** instance using **Docker Compose**, which allows for easy orchestration of multiple containers and ensures that the environment remains consistent across different stages (development, testing, and production).

GitHub Actions automates the entire CI/CD pipeline, managing the building, testing, and deployment processes. Defined in a YAML file, the pipeline triggers on every code change, running both manual and automated tests and ensuring the application is ready for deployment. Upon successful test execution, the application is deployed to the AWS EC2 instance via Docker Compose, allowing for scalable and reliable deployment in the production environment.

Automation Testing

Automation testing is the cornerstone of the BUAN chatbot's QA process, covering repetitive, complex, and high-priority functionalities. Automated scripts are developed for both **API** and **UI** testing to ensure that critical areas such as chatbot interactions, user authentication, course recommendations, and personalized suggestions work as expected.

- **API Automation**: Focuses on ensuring the chatbot's API can handle various types of requests (valid, invalid, unauthorized), verifying status codes, and response times.
- **UI Automation**: Verifies that the user interface behaves correctly across different browsers and devices, ensuring features such as login, chatbot interaction, and navigation work seamlessly.
  Automation tests are integrated into the CI/CD pipeline via **GitHub Actions**, allowing for continuous monitoring of system integrity with each code update.

Manual Testing

Manual testing complements the automation strategy by focusing on areas that require human judgment and exploration. This includes testing **UI/UX elements**, chatbot conversations, and exploratory testing for newly developed features. Manual testing is essential for ensuring the chatbot's responses are accurate, natural, and intuitive from a user's perspective, and for validating edge cases or interactions that may not be covered by automated tests. Testers engage with the chatbot like a real student would, ensuring a smooth and engaging experience.

Regression Testing

Given the dynamic nature of chatbot development, **regression testing** is a key component to ensuring that new features or changes do not break existing functionality. Automated regression tests run after each deployment to validate that core functionalities—such as chat history retrieval, course recommendations, and user authentication—continue to work as expected.

- **Regression Test Automation**: Automated tests are executed as part of the CI/CD pipeline, ensuring that any issues introduced during new development or changes are detected early.
- **Manual Regression Testing**: For more complex scenarios or interactions that require human validation, manual regression tests are conducted to ensure seamless operation after changes.

The following metrics are used to ensure that quality standards are consistently met and tracked:

Metrics

Metric Name	Description
Number of Test Cases	The total number of test cases written for both manual and automated testing. Ensures proper test coverage.
Test Case Pass Rate	The percentage of test cases that pass successfully. Formula: (Passed Test Cases / Total Test Cases) * 100
Defect Density Rate	Measures the number of defects per KLOC (thousand lines of code). Formula: (Total Defects / KLOC).
Code Coverage	Percentage of code covered by unit and integration tests. Helps ensure that the most critical parts of the code are tested.
# of User Stories Completed	Tracks how many user stories are completed in each sprint/iteration. Helps measure progress and project velocity.
Automated Test Pass Rate	The percentage of automated tests that pass successfully during CI/CD pipeline execution. Formula: (Passed Automated Tests / Total Automated Tests) * 100

Mean Time to Resolution (MTTR)	The average time taken to resolve identified defects. A lower MTTR indicates efficient defect resolution processes.
User Satisfaction Score (future)	Feedback collected from users regarding their experience with the application, measured through surveys or feedback forms.

Coding Standard

The project will follow industry best practices and specific coding standards to maintain consistency, readability, and quality across the codebase.

- **Naming Conventions:** Variables, methods, and classes should use meaningful names that convey the purpose of the code.
    - **Classes:** Use PascalCase (e.g., CreateBackendService).
    - **Methods and Variables:** Use camelCase (e.g., getCourseDetails, studentName).
    - **Constants:** Use ALL_CAPS for constants (e.g., MAX_RETRY_COUNT).
- **Indentation:** Use 4 spaces for each level of indentation (avoid tabs).
- **Commenting:** Use Javadoc-style comments for public methods and classes. Inline comments should only be used to explain complex logic.
- **Code Structure:** Keep methods short and focused on a single task (preferably less than 50 lines). Classes should follow the Single Responsibility Principle (SRP).
- **Error Handling:** Use proper exception handling (try-catch blocks) and avoid suppressing exceptions silently.
- **Version Control:** Each commit should address one logical change, with descriptive commit messages.
- **Code Formatting Tools:** Use Prettier for frontend (JavaScript), Checkstyle for backend (Java), and Autopep8 for Python AI service codes.

Code Review Process

The code review process ensures high-quality code and consistency across the team, involving reviews for each pull request before merging into the main codebase.

- **Who Reviews the Code:**
    - The design leader will review all architectural changes.
    - The implementation lead will review all backend and API-related code.
    - The security lead will review code related to user authentication and chat history security.
    - Team members are encouraged to cross-review each other's code to foster

collaboration and knowledge sharing.

- **Review Checklist:**
  - **Functionality:** Does the code meet the requirements of the task? Are all use cases handled?
  - **Readability:** Is the code easy to understand, and are the comments clear where necessary?
  - **Coding Standards:** Does the code follow the established coding standards (naming, formatting, etc.)?
  - **Error Handling:** Are exceptions properly handled? Is the code resilient to failure?
  - **Security:** Does the code meet security best practices? Are sensitive data and authentication mechanisms secured?
  - **Testing:** Are there sufficient unit tests, and do they cover edge cases? Are all tests passing?
  - **Performance:** Is the code efficient and optimized for performance (especially for APIs and database calls)?
- **Pull Requests:**
  - Every pull request (PR) must be reviewed by at least one other team member. PRs should include relevant unit tests and be merged only after all CI/CD tests pass. Use GitHub PRs to manage code reviews. Reviewers will leave comments and suggest improvements before approving the merge.
- **Feedback:** Reviewers should provide constructive feedback, highlighting areas of improvement as well as what was done well. If necessary, a follow-up review may be requested after changes are implemented.

Project Management

Effective project management is crucial for the successful delivery of the BUAN chatbot-driven web application. The following practices and tools will be employed to ensure smooth collaboration and progress tracking:

Tools

- **JIRA:**
  - Use JIRA for task management, issue tracking, and sprint planning. All team members are expected to update their tasks regularly to maintain transparency.
  - Create user stories, tasks, and bugs in JIRA with detailed descriptions and acceptance criteria.
- **Google Drive:**
  - Utilize Google Drive for document storage and sharing. Maintain version control for key documents, including project plans, technical specifications, and testing reports.
  - Use shared folders for easy access to project-related materials among team members.
- **Discord/WhatsApp:**
  - Use Discord for real-time communication, team collaboration, and informal discussions. Create channels for different topics, such as development, design, and general announcements.

    - Utilize WhatsApp for quick updates and urgent communications that require immediate attention.
- **Meeting Minutes:**
    - Document all meeting minutes and share them in a dedicated Google Drive folder. Include action items, assigned responsibilities, and deadlines to ensure accountability and follow-up.
- **GitHub:**
    - Use GitHub for version control and collaboration on the codebase. Each team member should create branches for their work and follow best practices for commit messages and pull requests.
    - Regularly review and merge pull requests to maintain code quality and integrate new features promptly.

Best Practices

- **Regular Stand-ups:**
    - Conduct daily or weekly stand-up meetings to discuss progress, challenges, and upcoming tasks. This keeps everyone aligned and fosters open communication.
- **Sprint Planning and Retrospectives:**
    - Hold sprint planning sessions to outline tasks for each sprint and retrospectives to review what worked well and what can be improved in future sprints.
- **Documentation:**
    - Maintain comprehensive documentation throughout the project lifecycle. This includes technical documentation, user manuals, and testing reports, ensuring all information is accessible and up-to-date.
- **Task Ownership:**
    - Assign clear ownership for tasks in JIRA, ensuring each team member knows their responsibilities and deadlines.
- **Feedback Loops:**
    - Encourage feedback from team members throughout the development process. This can be achieved through code reviews, discussions in meetings, and informal channels like Discord.
- **Monitoring Progress:**
    - Use JIRA dashboards to monitor project progress and identify potential roadblocks early. Adjust plans as needed based on team capacity and project requirements.
- **Conflict Resolution:**
    - Foster a culture of open communication to address conflicts or misunderstandings promptly. Utilize team meetings or one-on-one discussions as necessary.

Communication Etiquette

- Use clear and concise language in all communications.
- Be respectful of others' time; avoid unnecessary meetings and strive for efficiency.
- Document important discussions and decisions to ensure everyone is on the same page.

Testing

1. **Types of Testing:**
    - **Unit Testing:** Developers will test individual modules (React components, Java services, Python AI model) using JUnit, PyTest, and Jest.
    - **Integration Testing:** Ensure that the backend (Spring Boot APIs, AI model) interacts seamlessly with the frontend and the databases (PostgreSQL) using Langchain for RAG.
    - **Regression Testing:** Simulate the user's journey across the web app, ensuring the integration between the UI and backend services is smooth.
    - **Performance Testing:** Measure the response time of the chatbot, database queries, and API calls.
    - **Security Testing:** Ensure user authentication (Okta) and data protection (chat history) are robust.
2. **Manual Test Plan:** The Manual Functional Test Plan focuses on verifying the core functionality of the web application by manually executing test cases. Testers will validate features like course recommendations, user authentication, and AI interactions to ensure they work as expected. Each test will be performed from the user's perspective, with any issues documented and reported for resolution. This plan ensures the application meets its functional requirements before moving to further testing phases.
3. **Automation Test Plan and Framework:** A modular, scalable automation framework is essential for handling UI, API, and backend services.
    - **Language:** Java (for both Selenium & REST Assured).
    - **Framework:** TestNG (for test case management), Selenium (UI), REST Assured (API), Maven (build tool). Automation will reduce manual efforts and increase the speed of regression testing.
    - **Frontend Automation:** Use Selenium with TestNG for automating UI tests. Test cases include form submissions (login, course search), verifying course recommendations, and chat history functionality.
    - **Test Cases UI Testing (Selenium):** Login and authentication using Okta, course selection and recommendations, chat history retrieval and display.
    - **Tools:** Selenium WebDriver, TestNG, Maven.
    - **Backend/API Automation:** Use REST Assured for API testing (Java Spring Boot endpoints). Test cases include verifying API responses (course retrieval, chat history save, etc.), response times, and security headers.
    - **API Testing (REST Assured):** Verify response of course list API (GET /courses), test chat history save/retrieve (POST /chat/save, GET /chat/history), check error responses for unauthorized access (403 Forbidden).
    - **Tools:** REST Assured, Postman for API testing, Spring Boot testing libraries.
    - **Database Automation:** Verify PostgreSQL (chat history) through API queries. Ensure data consistency, retrieval accuracy, and security for user data.
    - **Performance Testing:** Use JMeter for load testing API endpoints to simulate multiple users accessing course recommendations simultaneously.

4. **Code Review Process for Automation Tests:**
    - PRs for all code changes, reviewed by at least one peer. All test cases must be validated, and automated tests should pass in CI/CD pipelines before merging.

Defect Management:

- **Tracking:** GitHub Issues for tracking bugs and issues.
- **Types of Defects:** UI bugs (e.g., misaligned elements, broken navigation), API issues (e.g., incorrect data returned, slow response), security vulnerabilities (e.g., unprotected data).
- **Defect Workflow:** Issues will be logged in GitHub with details, assigned to relevant developers, and tracked until resolved. Bug fixes will be re-tested.

Security Plan

JWT-based API Protection

- **Token Generation**
    - Generate JWT upon successful user authentication
    - Include user ID, roles, and expiration time in token payload
- **API Request Authentication**
    - Require valid JWT in Authorization header for protected endpoints
    - Implement JwtRequestFilter to intercept and validate tokens
- **Token Validation**
    - Verify token signature using secret key
    - Check token expiration time
    - Validate user details against database
- **Protected Endpoints**
    - Secure all API endpoints except for login, registration, and public information
    - Return 401 Unauthorized for invalid or missing tokens
- **Role-based Access Control**
    - Implement role checks for sensitive operations (e.g., admin functions)

Input Validation for Login and Sign-Up Pages

Username Validation

- Allow only alphanumeric characters, underscores, and hyphens
- Prevent special characters and HTML tags to avoid XSS attacks

Password Complexity

- Require a minimum of 8 characters
- Enforce the use of uppercase letters, lowercase letters, numbers, and special characters

- Reject weak passwords to mitigate brute-force attacks

Password Matching

- Ensure "password" and "confirm password" fields match during sign-up
- Block form submission until both fields are identical

Client-Side Validation

- Implement regular expressions to sanitize and validate inputs before submission
- Provide real-time feedback on input requirements and errors

Server-Side Validation

- Revalidate all inputs to ensure security beyond client-side checks
- Sanitize inputs to strip harmful characters, preventing XSS vulnerabilities

Cross-Site Scripting (XSS) Defense

- Reject input containing scripts or HTML tags
- Ensure proper escaping of user inputs in dynamic content to prevent script execution

CAPTCHA Integration (did not end up implementing)

- **Implementation**
  - Integrate reCAPTCHA v2 or v3 for user registration and login attempts
- **Verification**
  - Validate CAPTCHA response on the server-side before processing requests
- **Adaptive Challenge**
  - Increase CAPTCHA difficulty after multiple failed login attempts

Session Management

- **Inactivity Timeout**
  - Implement 30-minute inactivity timeout
  - Clear user session and require re-authentication after timeout
- **Chat History Retention**
  - Store chat history in database with creation timestamp
  - Implement scheduled task to delete chat history older than 30 days
- **Secure Logout**
  - Invalidate JWT on user logout
  - Clear client-side stored tokens and session data

Token Refresh Mechanism

- **Short-lived Access Tokens**
  - Set access token expiration to 15 minutes
- **Refresh Tokens**
  - Issue long-lived refresh token (expiration: 7 days) alongside access token
  - Store refresh tokens securely in database with user association
- **Token Refresh Process**
  - Client requests new access token using refresh token
  - Validate refresh token and issue new access token if valid
  - Invalidate and rotate refresh token after use
- **Refresh Token Security**
  - Implement refresh token rotation to mitigate token theft
  - Allow only one valid refresh token per user at a time

Additional Security Measures

- **HTTPS Enforcement**
  - Require HTTPS for all API communications
  - Implement HSTS (HTTP Strict Transport Security)
- **Password Security**
  - Enforce strong password policy (minimum length, complexity)
  - Use bcrypt for password hashing
- **Rate Limiting**
  - Implement rate limiting on login and token refresh endpoints
  - Use IP-based and user-based rate limiting
- **Audit Logging**
  - Log all authentication attempts, token generations, and sensitive operations
  - Implement secure log storage and rotation
- **Regular Security Reviews**
  - Conduct periodic security audits of the codebase
  - Stay updated with dependencies and apply security patches promptly
  -
- **GitHub Security Features**
  - **Automated Vulnerability Detection**
    - Utilize GitHub's dependency scanning to identify vulnerabilities in project dependencies
    - Implement code scanning using CodeQL to analyze code for potential security issues
    - Enable secret scanning to detect accidentally committed secrets (e.g., API keys, tokens)
  - **Automated Pull Requests**

    - Use GitHub's Dependabot to automatically create pull requests for:
        - Updating dependencies to the latest secure versions
        - Suggesting fixes for identified security issues in the codebase
- **Security Advisories**
    - Monitor GitHub's security advisories for detailed information about detected vulnerabilities
    - Review severity levels, affected components, and recommended remediation steps
- **Automated Alerts**
    - Configure immediate notifications for the development team about critical security issues
    - Establish a protocol for prompt action on received alerts

References

BU MET CS Team 1. *(2024). Project documentation (SPPP, SPPP risk management, Progress Report, SDD, Readme.md). N. Liew, N. Foithong, A. Singh, B. Cevik, Y. Liu, P. Chantarapornrat (Authors).*

Okta. *(2023). Authentication API documentation. Retrieved from https://developer.okta.com/docs/reference/api-overview/*

Langchain. *(2023). API documentation for OpenAI ChatGPT 4.0, chat history generation, RAG, and session management. Retrieved from https://docs.langchain.com/docs/*

CS673 Course Team. *(2024). Notes from CS673 slides. Blackboard Course MS.*

Spring.io. *(2023). Spring Boot documentation. Retrieved from https://spring.io/projects/spring-boot*

Docker, Inc. *(2023). Docker documentation. Retrieved from https://docs.docker.com/*

Axios. *(2023). Axios documentation. Retrieved from https://axios-http.com/docs/intro*

Atlassian. *(2023). JIRA documentation. Retrieved from https://support.atlassian.com/jira-software-cloud/docs/*

GitHub. *(2023). GitHub documentation. Retrieved from https://docs.github.com/en*

PostgreSQL Global Development Group. *(2023). PostgreSQL documentation. Retrieved from https://www.postgresql.org/docs/*

Braude, E., & Bernstein, M. E. *(2016). Software engineering: Modern approaches (2nd ed.). Waveland Press, Inc.*

Martin, R. C. *(2003). Agile software development: Principles, patterns, and practices.*

Bruegge, B., & Dutoit, A. H. *(2010). Object-oriented software engineering: Using UML, patterns, and Java.*

Pfleeger, S. L., & Atlee, J. M. *(2010). Software engineering: Theory and practice.*

Pressman, R. S. *(2014). Software engineering: A practitioner's approach (9th ed.). McGraw-Hill.*

Van Vliet, H. *(2008). Software engineering: Principles and practice.*

Sommerville, I. *(2016). Software engineering (10th ed.).*

Sommerville, I. *(2011). Engineering software products: An introduction to modern software engineering.*

Farley, D. *(2022). Modern software engineering: Doing what works to build better software faster.*

Brooks, F. P., Jr. *(1995). The mythical man month: Essays on software engineering (2nd ed.). Addison-Wesley.*

Freeman, E., Freeman, E., Bates, B., & Sierra, K. *(2004). Head first design patterns. O'Reilly Media.*

Fowler, M., Beck, K., & Roberts, D. *(2019). Refactoring: Improving the design of existing code (2nd ed.). Addison-Wesley.*

McConnell, S. *(2004). Code complete: A practical handbook of software construction (2nd ed.). Microsoft Press.*

Martin, R. C. *(2008). Clean code: A handbook of agile software craftsmanship. Prentice Hall.*

Thomas, D., & Hunt, A. *(2019). The pragmatic programmer: Your journey to mastery (20th Anniversary ed.). Addison-Wesley.*

Winters, T., Manshreck, T., & Wright, H. *(2020). Software engineering at Google: Lessons learned from programming over time. O'Reilly Media.*

Humble, J., & Farley, D. *(2010). Continuous delivery: Reliable software releases through build, test, and deployment automation. Addison-Wesley.*

Kim, G., Behr, K., Spafford, G., & Ruen, C. *(2018). The phoenix project: A novel about IT, DevOps, and helping your business win (3rd ed.). IT Revolution Press.*

Forsgren, N., Humble, J., & Kim, G. *(2018). Accelerate: The science of lean software and DevOps: Building and scaling high-performance organizations. IT Revolution Press.*

Kim, G., Humble, J., Debois, P., Willis, J., & Forsgren, N. *(2016). The DevOps handbook: How to create world-class agility, reliability, & security in technology organizations. IT Revolution Press.*

Farley, D. *(2021). Continuous delivery pipelines: How to build better software faster. O'Reilly Media.*

Glossary of Terms

Academic Advisor: A faculty or staff member who guides students on academic courses, degree requirements, and career planning.

AI (Artificial Intelligence): The simulation of human intelligence by machines, specifically algorithms that allow computers to perform tasks like understanding natural language and making decisions. This project integrates OpenAI's ChatGPT 3.5 and Llama 2 for real-time course recommendations.

API (Application Programming Interface): A set of protocols and tools that allow different software components to communicate and share data. In this project, APIs connect the front-end application to back-end services, enabling data exchange between the chatbot, databases, and AI models.

Application: A software program designed to perform a specific function for the user.

Authentication: The process of verifying the identity of a user to ensure secure access. Okta is used in the project for handling user authentication, ensuring only authorized users can log in.

Axios: A JavaScript library for making HTTP requests from Node.js or the browser.

AWS (Amazon Web Services): A cloud computing platform offering services like storage, databases, and AI tools.

Backend: The part of the system responsible for connecting databases, tools, APIs, and services to frontend components.

Branches: Versions of a codebase used for managing features or changes before merging into the main code.

Bugs: Errors in code causing malfunction or failure in the system.

Caching: A mechanism to temporarily store data for quick access, reducing load times. This project considers using caching for faster page loads by retaining frequently accessed data.

Chat: A platform for real-time communication via text, voice, or video.

Chatbot: An AI-driven system designed to engage in conversations with users, providing information and assistance based on user queries.

ChatGPT 3.5: The AI model used in the web application.

Components: Reusable parts of an application, such as UI elements or modular code.

Database: An organized collection of data stored electronically.

Docker: A platform that uses containers to package and deploy applications, ensuring that software runs the same way regardless of the environment. Docker is an optional tool in this project for maintaining consistent development and production environments.

Fetch API: A JavaScript API for making network requests to servers.

Figma: A cloud-based design tool for interface design and prototyping.
Framework: A collection of tools and libraries to streamline software development.
Frontend: The user-facing side of the web application, built using React.js. It provides the interface through which users interact with the chatbot and access course recommendations.
CI/CD (Continuous Integration/Continuous Deployment): A development practice where code changes are automatically tested and deployed, ensuring that new features or bug fixes are regularly integrated into the project. GitHub Actions is used for this purpose.
Configuration/Config: A set of parameters to customize a program's behavior.
GitHub: A file management system for collaborative software development.
HTTPs: Hypertext Transfer Protocol Secure, a secure version of HTTP.
Issues: Problems or tasks tracked in project management tools.
Java: A programming language used for building applications.
JIRA: A project management tool for tracking issues, bugs, and tasks.
Langchain: A company offering AI tooling for Retrieval-Augmented Generation (RAG) and session management.
LLM (Large Language Model): A type of AI model trained on vast amounts of text data to understand and generate human-like text responses.
Management: Coordinating resources and tasks to achieve objectives.
Microservices: A software architecture where applications are structured as independent, deployable services.
Okta: A cloud-based identity and access management service.
OpenAI: A company that provides the ChatGPT AI model.
PostgreSQL: An open-source relational database management system used for storing structured data efficiently and chat history and other user data in this project.
Python3: A version of the Python programming language.
RAG (Retrieval-Augmented Generation): A technique that combines retrieval of relevant information and generative responses to improve the accuracy of chatbot replies.
React.js: A JavaScript library used to build the front end of the application. It allows for the creation of interactive user interfaces and handles the chat interaction between the user and the AI.
RESTful API: A set of guidelines for building APIs using standard HTTP methods.
Regression Testing: A software testing practice that ensures new code changes don’t adversely affect existing functionality. The project implements regression testing to ensure that updates to the AI chatbot don’t disrupt other features.
REST Assured: A Java library used for testing RESTful APIs. The project uses REST Assured to

validate that the backend APIs respond correctly and efficiently.

Selenium: A testing framework used to automate web browsers, validating the chatbot's user interface. Selenium ensures the front end functions properly after each code change.

Spring Boot: A Java-based framework used for building and deploying back-end services, handling API requests, and managing interactions with the databases.

TensorFlow: A machine learning framework used to develop and train AI models. TensorFlow may be used to enhance the chatbot's learning capabilities.

Unit Testing: A type of software testing where individual components of the application are tested in isolation. This project uses unit testing to ensure each module (React components, Java services, and AI models) works correctly before integrating them.

WebSockets: A protocol enabling real-time, two-way communication between a client and server, allowing for instant updates during chat sessions.

Workflow: A sequence of steps to complete a task or achieve an objective.