

**CS673 Software Engineering**  
**Team 1 - BU Academic Navigator (BUAN)**  
**Software Design Document**



<u>Team Member</u>	<u>Role(s)</u>	<u>Signature</u>	<u>Date</u>
Natasya Liew	Team Leader	<u>Natasya Liew</u>	<u>September 2, 2024</u>
Natthaphon Foithong	Design and Implementation Lead	<u>Natthaphon Foithong</u>	<u>September 7, 2024</u>
Ananya Singh	Security Lead	<u>Ananya Singh</u>	<u>September 7, 2024</u>
Battal Cevik	QA Lead	<u>Battal Cevik</u>	<u>September 8, 2024</u>
Poom Chantarapornrat	Requirement Lead	<u>Chan P.</u>	<u>September 6, 2024</u>
Yu Jun Liu	Configuration Lead	<u>Yujun Liu</u>	<u>September 7, 2024</u>

**Revision history**

<u>Version</u>	<u>Author</u>	<u>Date</u>	<u>Change</u>
2.0.0	Ananya Singh	<u>September 26, 2024.</u> <u>October 6th, 2024.</u> <u>October 14th, 2024</u>	Adding the SB, Auth, and Security Frameworks, Add Class Diagram for Security and updated security notes, Automatic Security Vulnerability Detection
2.0.1	Natasya Liew	<u>September 28, 2024.</u> <u>October 5, 2024</u>	Updating changes in the Frontend Design, Update the Database design section, Tools added, Security Design
2.0.2	Yu Jun Liu	<u>September 29, 2024.</u> <u>October 5, 20224</u>	Update Diagrams

2.0.3	Natthaphon Foithong	<u>September 30, 2024,</u> <u>October 2, 2024,</u> <u>October 5.</u>	Update Mockup Images, Update UI Component features, Class diagram (figure3) and UI design session
2.0.4	Poom Chantarapornrat	<u>October 7, 2024</u>	Updated the frontend information and frameworks. Update the UI screenshots.
3.0.1	Poom Chantarapornrat	<u>October 14, 2024</u>	Update ai-service's vector storage technology, updated the frontend validation information, fixed some typos, formatted the document.
3.0.2	Battal Cevik	<u>October 14, 2024</u>	Updated Testing Design and CICD design steps
3.0.3	Natasya Liew	<u>October 14, 2024</u>	Final Proofread and edits. Adding the security layer on the requirement check and the future goals of the CourseTree Object.

[Introduction](#)

[Software Architecture](#)

[Class Diagram](#)

[UI Design](#)

[Database Design](#)

[Security Design](#)

[Business Logic and/or Key Algorithms](#)

[Design Patterns](#)

[Models and Tools](#)

[References](#)

[Glossary](#)

# Introduction

This document outlines the design and architecture of a **BUAN** chatbot-driven web application developed to assist students with course selection planning for the upcoming semester. The core functionality of the system is built around a conversational AI (OpenAI ChatGPT-4o mini via Langchain API) that provides personalized course recommendations and answers course-related queries for BU MET students in the Computer Science program. The BUAN chatbot system is embedded within a larger framework that supports real-time interaction, secure user authentication, and advanced features such as chat history caching, sharing, and email integration.

The primary goal of this project is to create a robust and scalable academic advising system that simplifies course selection for students, particularly within the MS Software Development (MSSD) program. The BUAN chatbot system is designed with flexibility, scalability, and user-centricity in mind; integrating cutting-edge technologies, including React for the frontend, Spring Boot (Java) for backend services, Postgresql for database management, and Python for the implementation of the BUAN AI Chatbot service leveraging the OpenAI's GPT-4o mini model. The system uses the Langchain tool for Retrieval-Augmented Generation (RAG), natural language processing, and logic-based querying. Additionally, real-time communication between the client and server is established using WebSockets, providing a smooth and responsive user experience.

## Design Goals

### 1. Seamless Chatbot Interaction:

Provide an intuitive and responsive chatbot interface that allows students to easily inquire about courses, prerequisites, and personalized recommendations. The chatbot should deliver accurate responses, leveraging the AI model through OpenAI's ChatGPT-4o mini model via Langchain API. The Langchain API was also used for easy Retrieval-Augmented Generation (RAG) to provide accurate and reliable results to users.

### 2. Modularity and Component Reusability:

Design the system with a clear separation of concerns, ensuring that the UI, backend services, and AI model remain modular and can be maintained independently. This approach will make the system more adaptable to changes and ensure that individual components can be upgraded or replaced without affecting the overall architecture, enabling easier debugging and future enhancements.

### 3. Performance Optimization:

Incorporate caching mechanisms to optimize system performance. Client-side caching will ensure faster response times during chat interactions, while server-side caching will reduce the load on the system by frequently requested data, such as course descriptions and prerequisite details. The system uses Langchain to store chat history in JSON format, ensuring rapid data retrieval and streamlined communication between services.

**4. Security and User Privacy:**

Implement secure user authentication using JWT tokens to ensure user data protection. All user-related information, including chat history, user's academic history, and course preference, will be encrypted and transmitted through secure communication channels. This ensures that personal data is safeguarded at all stages of the interaction with the system.

**5. Chat History and Enhanced Features:**

Allow users to store, retrieve, and share chat history. Chatlogs will be available for review so that users can revisit prior conversations with the BUAN chatbot. Enhanced features, such as the ability to email or print chat logs, will further assist students in managing their course information efficiently.

**6. Real-time Communication:**

Enable real-time chat functionality to facilitate seamless conversations between students and the BUAN chatbot. WebSockets will be employed to maintain immediate responses and dynamic updates during chat sessions, creating a smooth and interactive user experience.

**7. Personalized Course Recommendations:**

The course recommendation system is dynamically generated based on the student's academic history and their path of interest, using a custom-built course tree structure. The system takes into account courses that the student has already completed and their chosen specialization (e.g., web development, AI/ML, data science, secure software development, or app development), and recommends core and elective courses accordingly. This process ensures that students receive personalized and optimized course suggestions, tailored to their academic progress and goals. The recommendation system also incorporates logic to manage course prerequisites and elective requirements. For example, if certain courses have already been taken or certain prerequisites are met, the system will adjust its recommendations dynamically, ensuring that students can efficiently complete their program while meeting all the necessary requirements.

**8. User Experience Focus:**

Design a clean and intuitive user interface that enhances the BUAN chatbot experience. The UI is accessible and provides seamless interaction with the system. Emphasizing ease of use, the interface will allow students to engage with the system without unnecessary complexity or friction.

**9. LangChain integration for RAG and Chat History creation:**

Through Langchain, the BUAN Chatbot system will utilize Retrieval-Augmented Generation (RAG) to answer basic course information queries using the courses and programs CSV files generated based on the BU MET Computer Science webpage. The course builder functionality will allow students to plan their semester, while logic-based queries will handle questions beyond the scope of the CSV data. Langchain will manage the chat history in JSON format, pushing it from the Python-based AI service to the Spring Boot backend, enabling chat retrieval on the front-end (this JSON will eventually be used to populate our Postgresql user database for future retrieval. We also had the prompt library in the format of a JSON format for RAG retrieval in order to increase the scope of which our BUAN chatbot

can provide an accurate response for.

## **10. CI/CD Deployment and Scalability:**

The application is containerized using Docker to ensure consistent environments across development, testing, and production. For orchestration, Docker Compose is used to manage multi-container deployments efficiently. The entire system is deployed on an AWS EC2 instance, providing a flexible and scalable cloud-based infrastructure.

GitHub Actions is used to establish the continuous integration and continuous deployment (CI/CD) pipelines, automating the processes of building, testing, and deploying the application. The pipeline, defined in a GitHub Actions YAML file, triggers on every code change, running automated tests and ensuring that only successfully validated code is deployed to the AWS EC2 instance. This setup ensures that the system is continuously monitored, tested, and deployed, enabling rapid iteration and reliable scaling in a production environment.

## **11. Automation Testing Design**

The BUAN chatbot system's automation test suite focuses on ensuring robust functionality across API interactions and UI elements. Our automation tests cover a broad range of scenarios, including API validations, user authentication, chatbot interactions, and personalized course recommendations. Automated API tests validate the system's ability to handle valid and invalid inputs, ensuring that proper HTTP status codes (e.g., 200, 400, 401) are returned based on the request context. UI automation scripts verify that the user interface elements, such as login forms and navigation buttons, function seamlessly across different platforms and browsers. We employ tools Selenium Grid for UI testing and Postman for API testing to simulate real-world user interactions, ensuring that the chatbot and course recommendation engine perform accurately under various conditions. These tests are integrated into the CI/CD pipeline to run automatically during every deployment, ensuring early detection of bugs and issues.

## **12. Manual Testing Design**

While automation provides comprehensive coverage for repetitive and critical paths, manual testing is equally essential to ensure the quality of new features, UI enhancements, and user flows that are difficult to automate. Manual testing focuses on exploratory testing, UI/UX validation, and edge-case scenarios that require human judgment, such as assessing the chatbot's conversational accuracy and user experience. Manual testers interact with the system as students would, verifying the chatbot's responses for accuracy in course suggestions, prerequisites, and advice, ensuring a user-friendly and intuitive experience. These manual tests complement automated scripts, providing a deeper understanding of the real-world behavior of the system and identifying any inconsistencies in the chatbot's interaction patterns.

## **13. Regression Testing Design**

Given the ongoing development and frequent updates to the chatbot system, a robust regression testing strategy is in place. The regression suite ensures that existing functionalities, such as chat history retrieval, user authentication, and the course recommendation engine, are

not affected by new feature implementations or bug fixes. Automated regression tests are executed after every code update via the CI/CD pipeline to verify the system's stability. These tests cover critical modules, api, ui and ai services to query responses, personalized recommendations, and user data encryption mechanisms. Regression tests also evaluate the performance optimization features, such as client- and server-side caching, to ensure they continue functioning as expected. This comprehensive approach ensures that every deployment maintains the system's overall reliability without introducing new issues, allowing for seamless scaling and future enhancements.

#### **14. Multi-language Integration:**

This project demonstrates the power of integrating multiple programming languages and technologies, including React for the front end, Java (via Spring Boot) for backend services, SQL (Postgres) for data storage, and Python for AI-driven features. This system mirrors real-world software engineering practices and highlights how various technologies can work together effectively.

#### **15. Attention to Security, Testing, and Best Practices:**

Security, authentication, and best practices will be fundamental to the system's design. Unit testing and automated CI/CD pipelines will be implemented to ensure that potential threats and vulnerabilities are addressed. By following industry standards for security and testing, the system will be both reliable and secure for student and advisor use.

The BUAN chatbot system is designed to improve student engagement with the academic advising process by making course selection easier, faster, and more personalized. Through the integration of modern AI, secure authentication mechanisms, and well-structured architecture, this application aims to provide a cutting-edge, user-friendly experience for students at Boston University.

## Software Architecture

The BUAN chatbot application is designed with a client-server model architecture, following a three-tier structure: the frontend, backend (Springboot and Python AI service), and database layers. This decomposition aims to ensure scalability, maintainability, and efficiency. In this section, we will break down each component, describe the relationship between them, and illustrate the communication interfaces. We will also describe the various frameworks and technologies used across each layer.

The system consists of several key components: a **Frontend Layer** (React, Fetch API) that handles the user interface, provides course information, and personalized course recommendations, and integrates a BUAN chatbot with WebSocket for real-time chat updates. The **Backend Layer** is split into two parts: a Spring Boot service for authentication, security, and interaction with a PostgreSQL database, and an **AI Application Layer** using Langchain powered by OpenAI's GPT-4o for Retrieval-Augmented Generation (RAG) to answer queries and recommend courses-based on student data. The **Database Layer** (PostgreSQL) stores user data, course history, and chat logs. Leveraged JWT for authentication, and the system is deployed using Docker and AWS for CI/CD.

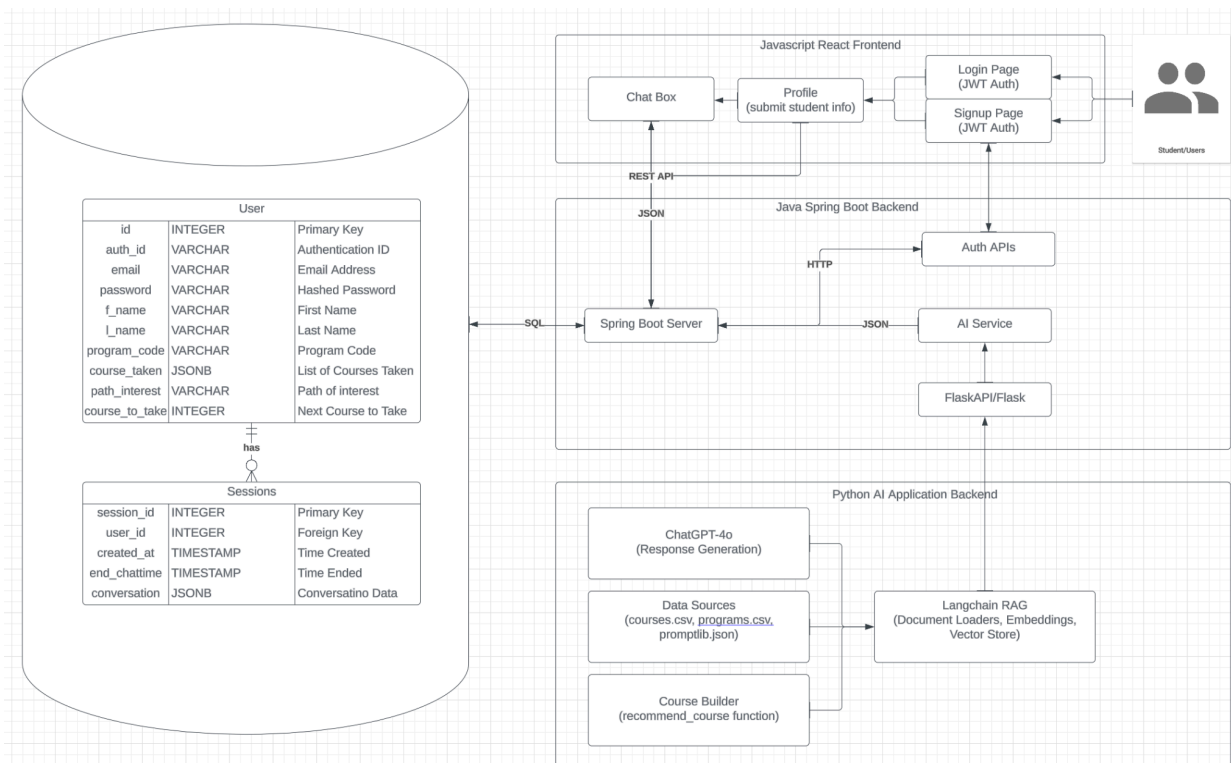


Figure 1. System Architecture Diagram



## Front-end Layer – (React.js)

The front end handles the user interface, capturing inputs and delivering real-time responses to the user. It also manages the display of course information, personalized course recommendations for the upcoming semester, and interaction through the BUAN chatbot whilst also allowing users to have access to their previous chat histories with our program's chatbot. The React application communicates with the backend via RESTful API calls using the Fetch API. WebSocket integration allows for real-time updates during chat sessions.

### Components

- *Chat Interface* – The main user interface space where users can ask questions and get responses from the chatbot.
- *Navigation Bar* – Allows navigation between the homepage and chat history.
- *Fetch Services* – Handle communication between the frontend and Spring Boot backend via RESTful API calls.
- *Chat History Saving and Downloading* – Allows users interact with the chatbot, all messages (from both the user and the bot) are continuously logged in. And users can export their chat history in a structured format (e.g., PDF or JSON file). Users can download the entire conversation from the current session or select specific parts of the chat history they wish to export.
- *Caching* – Cookies or session storage is used to cache chat data and user preferences for faster loading.
- *User Authentication* – JWT is used for authentication ensures that the user is logged in securely. Later on, we might switch to okta to boost for security

### Frameworks and Libraries

- *React.js* – For building responsive and interactive UIs on the web.
- *Vite.js* - For faster front end server deployment and development with hot reload.
- *Fetch* – For HTTP requests to communicate with the backend.
- *Material UI* – For pre-built UI components like buttons, icons, etc.

## Back-end Layer – (Spring Boot/Java and AI API integration)

The backend is responsible for managing user authentication, processing chat queries, and coordinating data exchanges between the front end and the database.

**Spring Boot** Handles the authentication, security, and interaction with the Postgres database. It also receives chat history data (in JSON format) from the Python-based AI service and makes it available for future retrieval by the front end.

Communication between the front-end and back-end occurs through REST API endpoints and WebSockets for real-time chat functionality. Internally, the Python service interacts with the Spring Boot service via HTTP requests, exchanging data such as chat logs and course recommendations.

## Components

- *API Layer* – Provides REST endpoints for the front-end to interact with. It includes endpoints for chat processing, course retrieval, and chat history management.
- *Authentication Service* – Manages secure user logins using JWT, including token verification, will leverage Okta for enhanced authentication later iteration.
- *Data Layer* – communicates with PostgreSQL database

## Frameworks and Libraries

- *Java (Spring Boot)* – Manages API endpoints, security, and business logic.
- *Spring Data JPA* – For database communication (PostgreSQL).
- *JWT Tokens* – For managing user authentication.

## AI Application Layer - (Python, Langchain, FastAPI)

The AI application layer is responsible for handling AI/chat responses to user inputs, leveraging several key components to create an efficient conversational experience. It integrates **ChatGPT-4o mini** using **Langchain** to manage complex queries and answer course-related questions by interacting with external data sources. Course and program data from Boston University are collected via Python scripts, stored in **CSV** format ([courses.csv](#), [programs.csv](#)) and **JSON** format ([promptlib.json](#)), and embedded for the model to generate accurate responses. These data are used to create embeddings, stored in a **FAISS vector database** accessible via **Langchain**. Langchain then acts as the document retriever for **Retrieval-Augmented Generation (RAG)** within the workflow. For personalized course recommendations, the [recommend\\_course](#) function from the **course builder** class generates suggestions based on program details and past coursework. **FastAPI** is used to expose the application as an API, offering high performance, automatic documentation, and easy integration for both backend and frontend teams, which **FlaskAPI** as a backup in case the FastAPI does not work. The application stack also utilizes essential libraries and frameworks like **FAISS** for vector storage and **Pandas** for data manipulation.

## Components

### LLM Application

This component integrates the large language model **ChatGPT-4o mini** to handle all responses in natural language. We use **Langchain** to manage more complex queries, such as answering course-related questions by retrieving relevant information from the data sources. Langchain provides a framework that simplifies the process of building applications with **Large Language Models (LLMs)**, facilitating integration with external data sources and creating RAG workflows for contextually accurate responses. LangChain also provides tools for prompt engineering, memory management, document handling, and custom component creation, enabling efficient interaction and decision-making capabilities within applications.

## Data Collection and Integration

In this project, we opted to avoid web scraping techniques for collecting course and program data from the Boston University MET website. This approach was driven by the need to maintain the integrity of the website, comply with data usage policies, and avoid burdening the university's IT team during their integration of the new Registration Management System (MS). Therefore, we relied on internally compiled data stored in pre-existing CSV files created by team members, which contain comprehensive information about courses and programs. This dataset is used in conjunction with Langchain to embed the data into a vector store, enabling accurate responses to user queries regarding academic courses and programs. This approach ensures **data consistency**, minimizes potential disruptions, and adheres to ethical data collection standards.

## Vector Storage

To enable efficient data retrieval, course and program data are stored in a vector database. **FAISS** was chosen as the vector storage solution, which integrates seamlessly with Langchain. It is a lightweight package compared to the Chroma vector store. **Embeddings** are generated from the CSV data and stored in FAISS. During chatbot interactions, Langchain uses FAISS as the document retriever to efficiently find and present the most relevant information. This setup makes the retrieval process accurate and performant while supporting similarity search for improved context matching.

## Course Builder Function

Generating a recommended course schedule is one of the complex requirements that LLMs may struggle with due to the risk of hallucinations. To address this, we use the `recommend_course` function from a **custom course builder class**. This function generates a list of recommended courses based on the student's program, area of interest, and the number of previously completed courses. The structured output from `recommend_course` is then used by **ChatGPT-4o mini** to generate natural language responses, ensuring accurate and useful recommendations for students.

## API Creation

After building the application using **Python**, **Langchain**, and vector storage, it is exposed as an API for backend and frontend team integration. **FastAPI** was chosen as the API framework due to its speed and efficiency in handling asynchronous requests. FastAPI simplifies the process of building RESTful APIs by providing automatic **OpenAPI documentation** and an interactive **Swagger UI**, making it easy for developers to explore and test endpoints. Its data validation through **Pydantic** ensures robust input handling, while its performance is on par with frameworks like **Node.js** and **Go**, making it ideal for high-performance applications. FastAPI's compatibility with Python type hints allows for cleaner code and easier debugging, ensuring smooth integration with the rest of our application stack.

## Frameworks and Libraries

- **Langchain:** A powerful framework designed for building applications with large language models (LLMs), enabling seamless integration with data sources and enhanced natural language processing capabilities. Langchain supports **RAG**, document handling, and complex query processing, which is crucial for generating contextually accurate responses.
- **FastAPI:** A modern web framework for building high-performance APIs with Python. It provides automatic documentation, supports asynchronous requests, and simplifies input validation through Python type hints, making it suitable for integrating the application with the frontend.
- **FAISS:** A vector storage service chosen to store and retrieve high-dimensional data. It integrates with **LangChain** to support similarity search, acting as a document retriever during user interactions for real-time and accurate data retrieval.
- **Pandas:** A versatile Python library for **data manipulation and analysis**, offering powerful data structures like DataFrames to efficiently process and clean data for use in the chatbot.

This updated AI application layer focuses on building a modular, scalable, and responsive chatbot that meets user requirements efficiently while being easy to maintain and expand in the future.

## Database Layer - (PostgreSQL)

The **PostgreSQL database** is responsible for storing **user data**, **course history**, and **chat logs**, providing a centralized and efficient mechanism for persisting and managing key data related to chatbot interactions. This relational database ensures that all user interactions and queries are **persisted** for future retrieval, **personalized recommendations**, and analysis. The **backend (Spring Boot)** interacts with the PostgreSQL database via **JDBC (Java Database Connectivity)** to facilitate efficient data storage and retrieval, enhancing the chatbot's ability to provide contextual and personalized responses.

## Components

### PostgreSQL Database

- **User Data:**
  - The **Users** table stores key information about each user, including their **auth\_id**, **email**, **hashed password**, **first and last name**, **program code**, courses taken (**course\_taken** as a JSONB list), **path interests**, and **next planned course** (**course\_to\_take**).
  - User data is indexed on several fields, including **program code** and **path interests**, to **optimize query performance** when searching for user-specific data or generating personalized course recommendations.

- **Chat History and Sessions:**
  - The **Sessions** table stores each user's chat history (**conversation** stored as a JSONB object), including the **session start time** (**created\_at**) and **session end time** (**end\_chattime**). This structure allows for efficient tracking of user interactions across multiple sessions.
  - **Indexes** have been added to key columns, such as **user\_id** (in the Sessions table) and **conversation**, to improve performance when retrieving past user interactions.
- **Structured Relational Storage:**
  - The database is designed with a **structured relational model** to manage user profiles, session information, and course histories effectively. This structure ensures data consistency and supports efficient querying for features such as **recommendation generation**, **user profiling**, and **session tracking**.

#### Indexes for Performance Optimization

- To enhance **query performance**, indexes have been added to key attributes across both **Users** and **Sessions** tables:
  - **Users Table:** Indexed fields include **f\_name**, **program\_code**, **course\_taken**, **course\_to\_take**, and **path\_interest**, enabling fast lookups for personalized recommendations.
  - **Sessions Table:** Indexed fields include **user\_id**, **conversation**, and **created\_at**, allowing for efficient retrieval of session history and supporting a seamless conversational experience.

#### Redis (Optional for Caching)

- **Redis** could be used optionally for **caching frequently accessed data**, such as:
  - **Course Lists:** This would help reduce database load when multiple users request course-related data.
  - **User Chat History:** Caching recent user interactions allows for **faster responses** and minimizes database read times, especially when users return to ongoing sessions.
  - This approach would significantly **improve the performance** of frequently accessed data, ensuring a faster response time for users.

#### Schema Overview and Updates

- **Users Table:**
  - Stores core user details, including program and course details in a **JSONB format**, which allows for **flexible storage** and retrieval of course data (e.g., completed courses).
- **Sessions Table:**
  - Captures **user interactions** with detailed conversation history in **JSONB**, making it easy to parse and use data for user behavior analysis and personalized recommendations.

- **Indexes:**
  - Indexes on key columns ensure **fast querying** of data, optimizing performance during heavy loads or complex queries, such as when retrieving historical chat logs for a specific user.

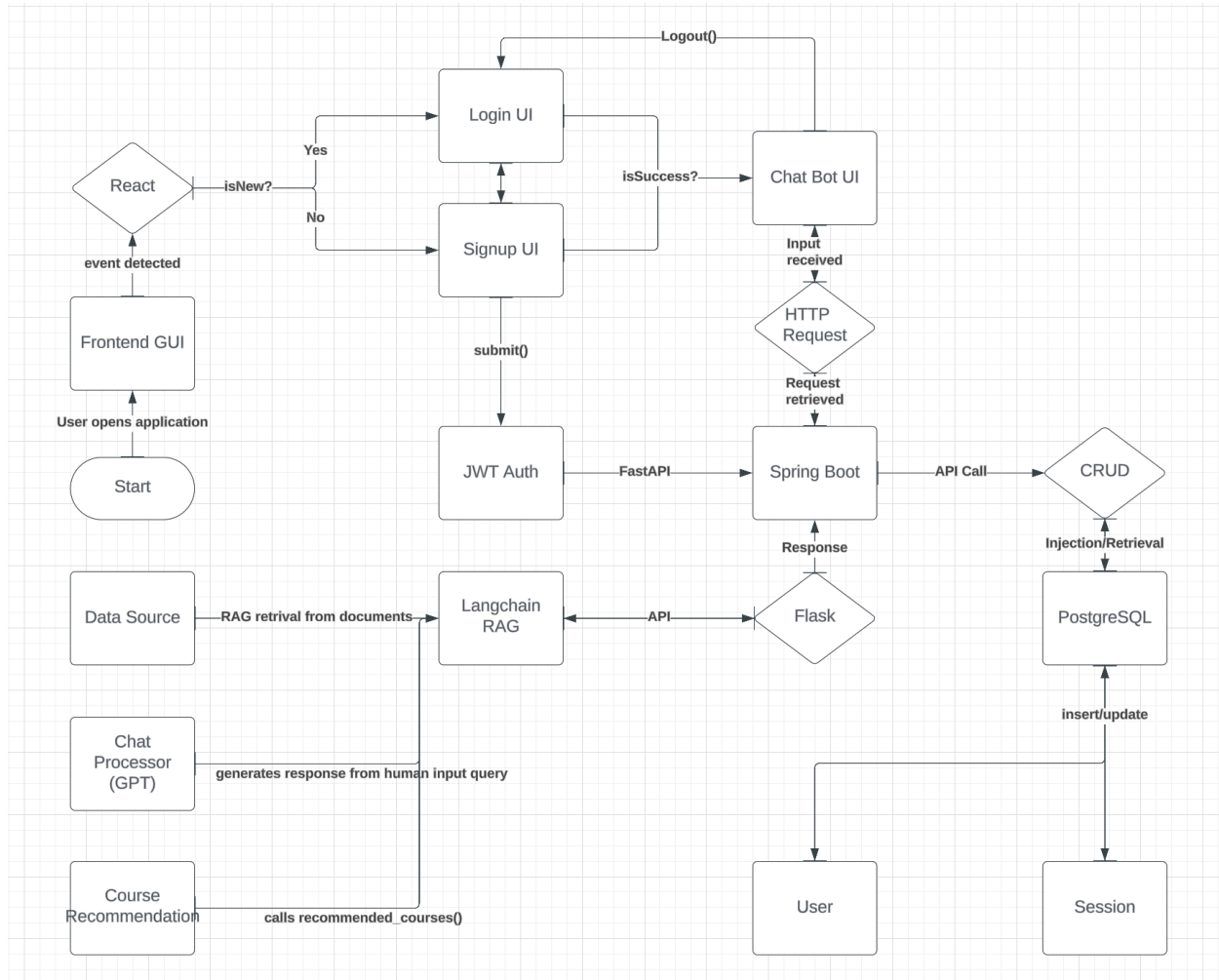
#### Benefits of This Database Design

- **User Personalization:** By storing user profiles, course history, and chat history, we can provide a more personalized experience. The **JSONB fields** (e.g., `course_taken` and `conversation`) allow for flexible and detailed storage, which is key to offering tailored responses and recommendations.
- **Scalable Interaction Management:** The **Sessions** table allows for tracking of user interactions over time, making it possible to continue conversations seamlessly or analyze user behavior for future improvements.
- **Performance Optimization:** Using **indexes** on frequently queried fields improves response times, especially during user interaction-heavy periods. The optional use of **Redis** for caching further enhances performance by reducing the number of database reads for commonly requested data.

#### Future Integration Consideration

- In future iterations, we plan to **integrate the PostgreSQL database** directly with our **Python AI service** and **Langchain RAG**. This would replace the existing CSV-based data retrieval mechanism, enabling **real-time updates**, **scalability**, and **direct querying** for generating responses. This would ensure that both course data and user information remain up-to-date and can be accessed efficiently, contributing to a more responsive and scalable chatbot system.

This updated **database layer** ensures that user interactions, course data, and session history are persistently stored and efficiently retrieved, providing the foundation for personalized recommendations and enhancing the overall experience for users interacting with the BUAN chatbot.



**Figure 2.** An Example of Data Communication Flow Across the Program

## Data Communication Flow Explained

The Data Flow diagram illustrates the architecture of BUAN Chatbot in perspective of users. The application starts with a React-based Frontend GUI where users are directed either to a Login or Signup UI, depending on whether they are new users. Successful authentication through JWT leads to access to a Chatbot UI.

The backend architecture involves a mix of FastAPI, Spring Boot, and Flask frameworks to handle different aspects of data processing and API management. Data retrieval and manipulation are handled through CRUD operations with a PostgreSQL database.

Basically, the data flow like this:

User Login/Signup → ChatBot UI → Chat → Spring Boot → AI-service & Database

# Class Diagram

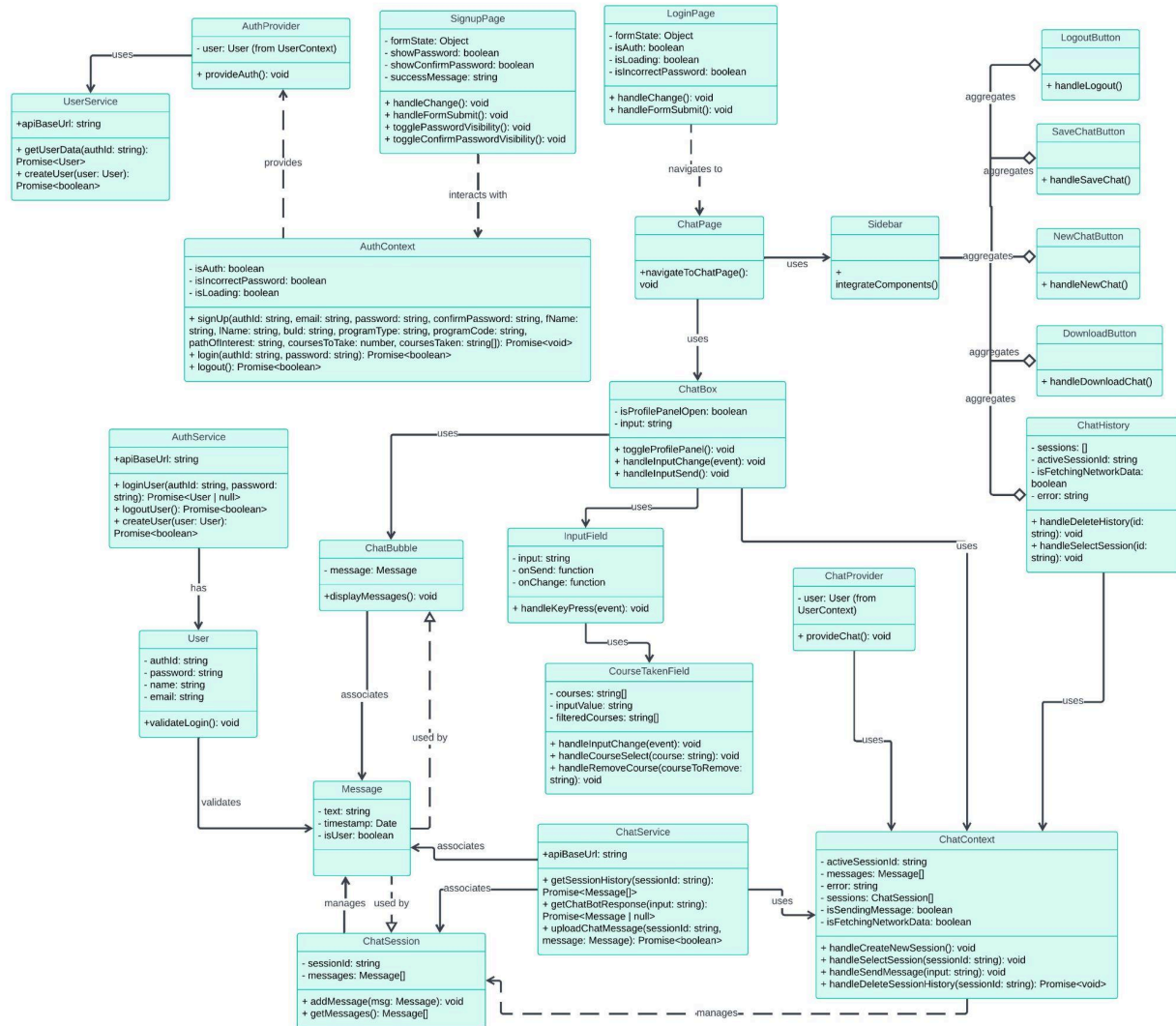
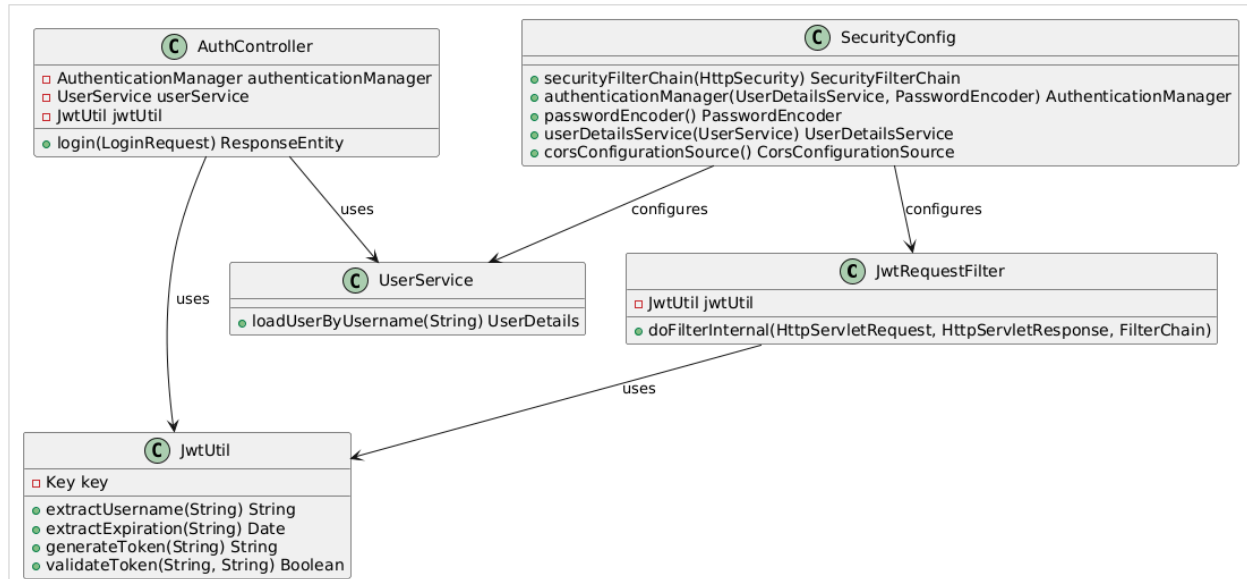


Figure 3. Application Class Diagram





Security Filtering Class Diagram

### ChatBox

- **Attributes:**
  - `isProfilePanelOpen`: boolean - Indicates if the profile panel is open.
  - `input`: string - Holds the current input from the user.
- **Methods:**
  - `toggleProfilePanel()`: Toggles the visibility of the profile panel.
  - `handleInputChange(event)`: Handles input changes from the user.
  - `handleInputSend()`: Sends the input when the user submits it.
- **Relationship:** Uses ChatBubble to display messages.

### ChatBubble

- **Attributes:**
  - `message`: Message - Holds the message object.
- **Methods:**
  - `ChatBubble(message: Message)`: Constructor to create a ChatBubble with a message.

### InputField

- **Attributes:**
  - `input`: string - Current input value.
  - `onSend`: function - Function to call when sending input.
  - `onChange`: function - Function to call when the input changes.
- **Methods:**
  - `handleKeyPress(event)`: Handles key press events, such as sending a message.

### CourseTakenField

- Attributes:
  - courses: string[] - List of courses taken by the user.
  - inputValue: string - Current input value for the course.
  - filteredCourses: string[] - List of filtered courses based on input.
- Methods:
  - handleInputChange(event): Handles changes in input.
  - handleCourseSelect(course: string): Handles the selection of a course.
  - handleRemoveCourse(courseToRemove: string): Removes a course from the list.

### ChatHistory

- Attributes:
  - sessions: [] - List of chat sessions.
  - activeSessionId: string - ID of the currently active session.
  - isFetchingNetworkData: boolean - Indicates if network data is being fetched.
  - error: string - Holds error messages, if any.
- Methods:
  - handleDeleteHistory(id: string): Deletes chat history by session ID.
  - handleSelectSession(id: string): Selects a chat session to view.

### Sidebar

- Attributes:
  - topSection: div - Component for the top section of the sidebar.
  - bottomSection: div - Component for the bottom section of the sidebar.
- Methods:
  - integrateComponents(): Integrates components into the sidebar.

### Buttons (SaveButton, DownloadButton, LogoutButton, NewChatButton)

- These classes represent buttons in the application.
- Methods:
  - Each button has a specific action to perform, such as save chat history, downloading chat, logging out, or starting a new chat.

### Pages (ChatPage, LoginPage, SignupPage)

- These classes represent different pages in the application.
- Attributes:
  - Each page has attributes for managing state (e.g., formState, isAuth, isLoading).
- Methods:
  - Each page contains methods for handling changes and form submissions.

### AuthContext, ChatContext, AuthProvider, ChatProvider

- Attributes:
  - These classes manage authentication and chat context, holding relevant state attributes.
- Methods:

- They provide methods for authentication actions (login, logout, sign-up) and chat actions (sending messages, handling sessions).

### **Services** (AuthService, ChatService, UserService)

- These classes handle API calls for authentication, chat, and user data.
- Attributes:
  - API\_BASE\_URL: string - Base URL for API requests.
- Methods:
  - Methods to login, logout, create users, and fetch session history or messages.

### **User**

- Attributes:
  - authId: string, password: string, name: string, email: string - Attributes related to the user.
- Methods:
  - validateUser(): Validates user credentials.

### **Message**

- Attributes:
  - text: string, timestamp: Date, isUser: boolean - Attributes for the message object.

### **ChatSession**

- Attributes:
  - sessionId: string, messages: Message[] - Attributes for managing chat sessions.
- Methods:
  - addMessage(msg: Message): Adds a message to the session.
  - getMessages(): Retrieves messages from the session.

### **Relationships:**

- ChatBox uses ChatBubble to display chat messages and InputField for user input.
- InputField uses CourseTakenField to allow users to select courses taken.
- ChatHistory uses ChatContext to manage chat sessions.
- Sidebar aggregates NewChatButton, ChatHistory, and LogoutButton.
- ChatPage uses ChatBox and Sidebar for navigation and chat management.
- LoginPage navigates to ChatPage after successful login.
- SignupPage interacts with AuthContext for user signup.
- AuthContext provides AuthProvider for managing authentication state.
- AuthProvider uses UserService for user-related operations.
- ChatContext manages ChatSession for active chat sessions.
- ChatService uses ChatContext for managing chat functionalities and associates with Message and ChatSession.

## UI Design

**Layout Design:** The overall design follows a clean, two-column, two-row structure:

- Left column (Side panel): New Chat, Download Chat History, Save Chat, Logout button, and Chat history timestamp
- Right column (Chat Interface): For the conversation area and sending messages. Profile icon and logo and Branding.

## Key UI Components

### Chat Interface

- Chat Window: The main chat window occupies the majority of the screen
- Message Bubbles: User and Bot messages appear in distinct bubbles, with user messages aligned to the right and bot responses aligned to the left
- Chat Scroll: Vertical scrolling allows users to review chat history
- Time Stamps (Optional): Each message has a timestamp underneath
- Input Field: A text input field at the bottom for typing messages
  - Text Box: A wide field for typing messages
  - Send Button: To send messages
  - Grammarly Check (Optional): An optional real-time grammar-checking feature

### Action Buttons

- New Chat Button:
  - Function: Initiates a new chat session, clearing the current chat history and resetting the user input field. It allows users to start fresh without logging out.
  - Design: “New Chat” icon button with the label
- Download Chat History Button:
  - Function: Generate and download the chat history as a PDF file when the user clicks the button
  - Design: “Download” icon button with the label
- Logout Button:
  - Function: Ends the user session and returns to the login page
  - Design: “Logout” icon button with the label
- Send Message Button:
  - Function: To send the text input from the user to the chatbot. It triggers the backend function that processes the message and generates a response from the chatbot.
  - Design: “Send” icon button
- Save Chat Button:
  - Function: Saves the current chat history to the database or downloads it locally for the user. This ensures the user can revisit previous chat sessions.
  - Design: “Save” icon button with the label

## User Input Validation

- **Message Input**
  - *Input* – Cannot be empty, the length must be less than 200 characters per message.
- **Signup Form**
  - *Username* – it cannot be empty, the length must be in between 6-50 characters.
  - *Email* – must be in the correct format (name@email.com)
  - *Password* – must be strong, at least 8 characters, contains at least 1 lowercase, uppercase, and a special character.
  - *FirstName* - cannot be empty, the length must be in between 2-50 characters.
  - *LastName* - cannot be empty, the length must be in between 2-50 characters.
  - *BUID* – cannot be empty, the length must be in between 1-10 characters.
  - *Number of Courses to Take* – a number between 1-4
  - *Courses Taken* – a list that can be empty.

## ChatBot Modes and User Profile Panel

- **Profile Icon:** Displays the user's profile picture (or placeholder) and name after login.
- **Session Timer:** Tracks the session duration.
- **Course Builder:** Asks how many courses a user wants to take, checks what courses they've completed, and provides a recommendation based on their major.
- **Course Information Request:** Asks for specific course details and provides relevant information.
- *These options will appear in an options dropdown or as clickable buttons at the start of the conversation.*

## Colors and Typography

- **#E54500** (Orange-red)
- **#675DFF** (Blue)
- **#FFD0BC** (light orange)
- **#FFFFFF** (white)

## Typography:

- Outfit, San Serif

## Responsive Design (Optional)

- **Mobile View:**
  - The chat interface collapses into a single-column view.
  - Action buttons are represented as icons, a toolbar, or a dropdown.
  - The input field remains sticky at the bottom for easy access.

## Navigation Flow

- **Signup Page:** New users can create an account by entering their username, email, password, BU ID, Program Type, Program Name, Path of Interest, Courses to Take, and Courses Taken. Then the user can click the "Signup" button. Upon successful registration, users are redirected to the Login Page.
- **Login Page:** The user logs in with their email or BU login credentials via JWT authentication.
- **Welcome Page:** Once logged in, users are greeted with a welcome message from the chatbot, asking if they need help with course selection or course information.
- **Chat Interface:** Users engage with the chatbot, and actions such as emailing, sharing, and printing can be taken during or after the conversation.
- **Log Out:** After completing the session, users can click "Logout" to clear chat and logout.

## Examples of User Interfaces

- The Sign-up page (new user)



## BUAN CHATBOT

### Sign up

Username

Your full name

Email

Your email

Password

\*\*\*\*\*

Confirm Password

\*\*\*\*\*

First Name

Insert your first name

Last Name

Insert your lastname

BU ID

Insert your BU ID

Program Type

MS degree

Program Name

MS in Computer Science

Path of Interest

App Development

v

Courses to Take

3

v

Courses Taken

C5521 - Information Structure with Java


x

Type to search courses...

Sign Up

Already have an account? [Login](#)

- Example of the signup form validation



## Sign Up

Username

Username must be at least 6 characters

Email

Email is not valid

Password

Password must be at least 8 characters long and include at least one uppercase letter, one lowercase letter, one number, and one special character.

Confirm Password

Passwords do not match

First Name

First name must be at least 2 characters


Last Name

Last name must be at less than 50 characters

BU ID

BU ID must be at less than 10 characters

- The sign-in page



## BUAN CHATBOT

### Sign In

Email

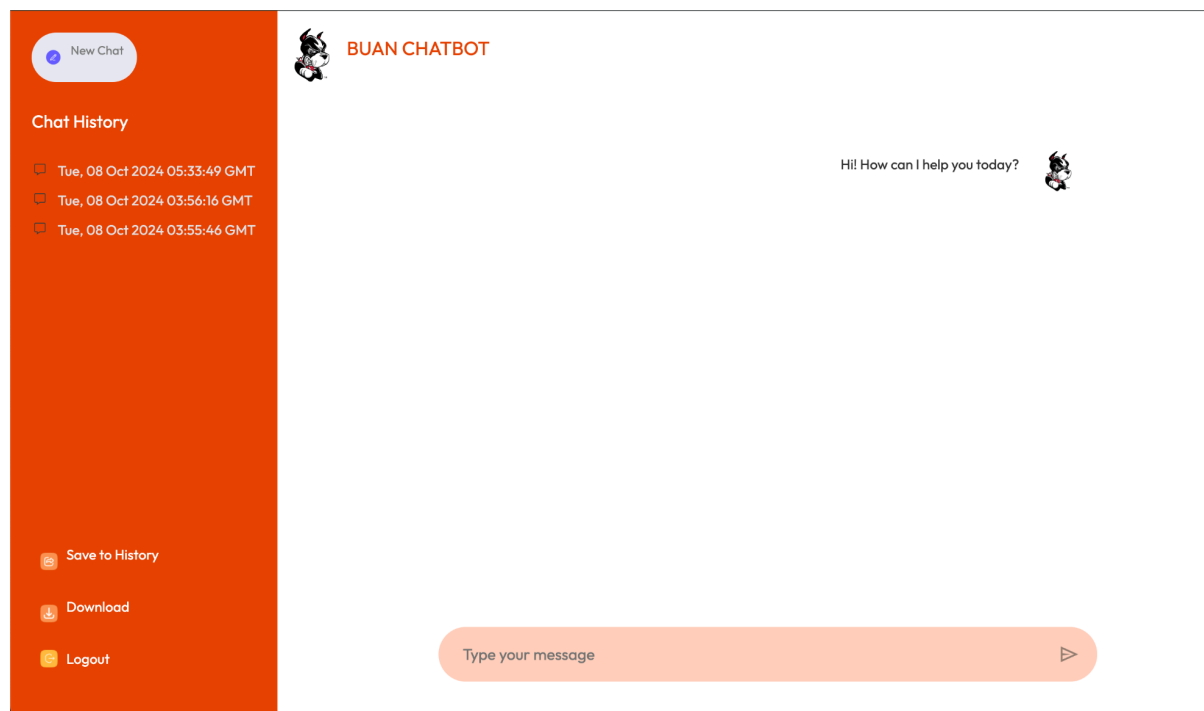
Password

Login

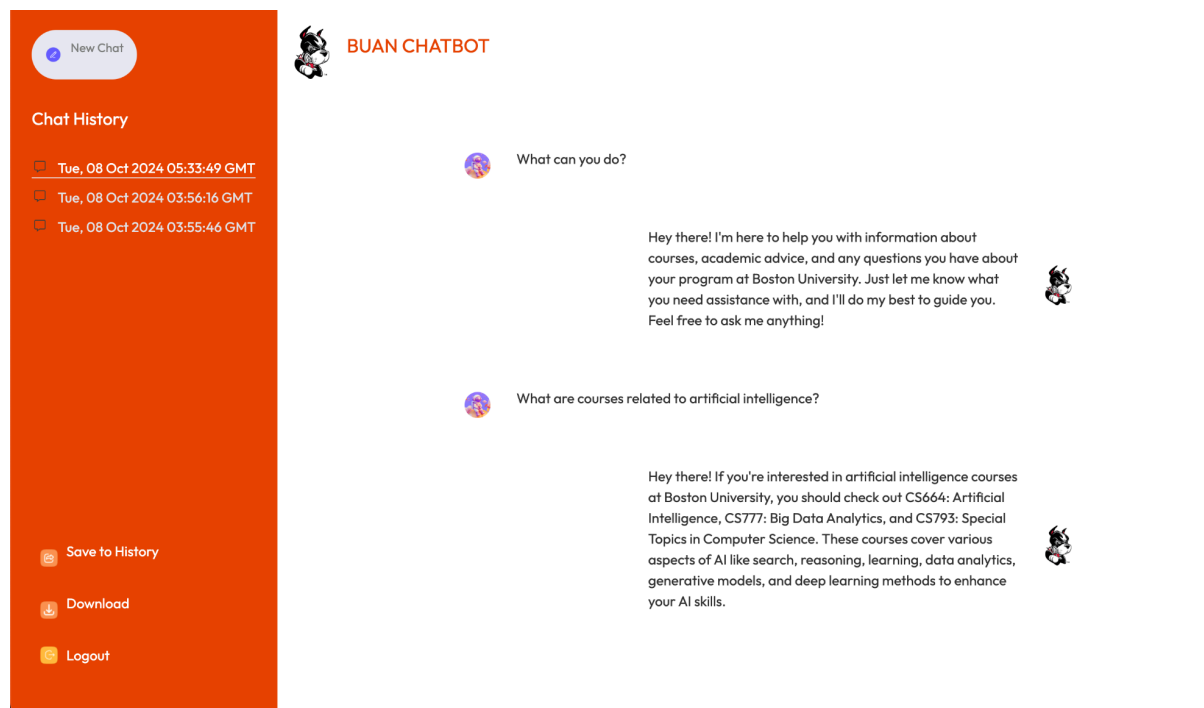
Do not have an account? [Sign Up](#)



- The home page (chat screen)



- The Chat History (chat screen)

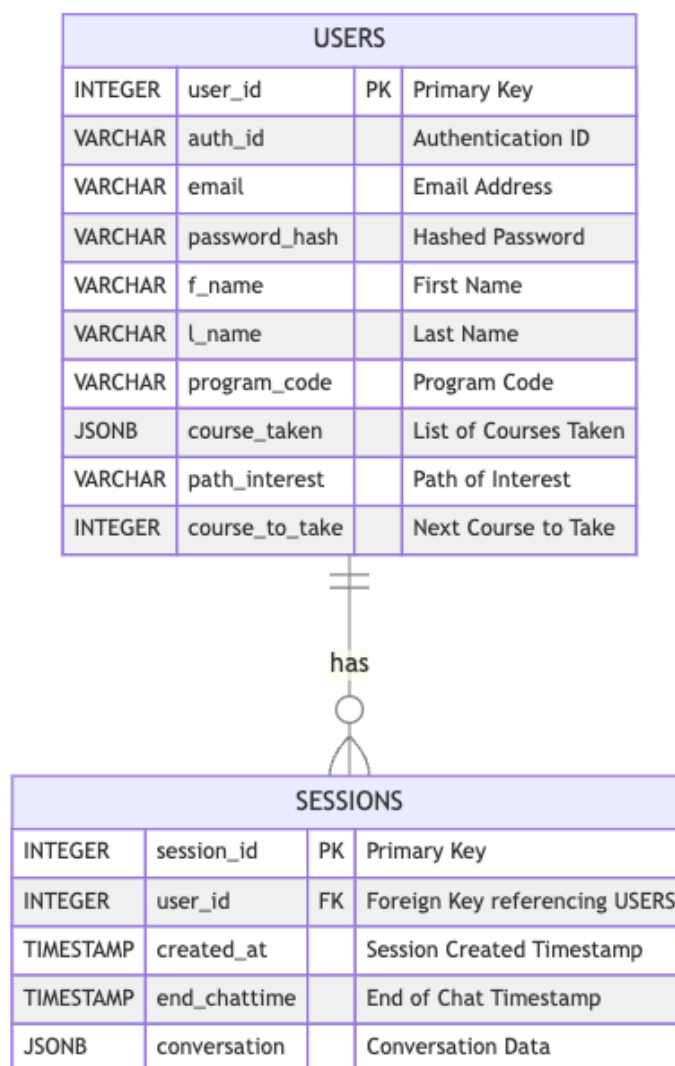


Please see all UI designs and prototype links below.

<https://www.figma.com/proto/gjNG1bADwnFvgDqclMwQVQ/Chat-AI-Bot---BUAN?node-id=119-525&node-type=canvas&t=nnJOFd7LkvOkXggt-1&scaling=min-zoom&content-scaling=fixed&page-id=0%3A1&starting-point-node-id=119%3A525>

## Database Design

### Current Design



## Chatbot Implementation

Using a PostgreSQL database, we have created two main tables: Users and Sessions, which form the backbone of our chatbot implementation. These tables allow us to manage user information, interactions, and chat history in a structured and efficient way.

### Users Table

Attributes:

- `user_id`: Primary key (serial), uniquely identifying each user.
- `auth_id`: Unique identifier for authentication (string, 50 characters), representing the integration with our external authentication provider.
- `email`: User email used for authentication (string, 50 characters), ensuring easy user identification.
- `password_hash`: Stores the hashed version of the user password for security purposes (string, 50 characters).
- `f_name` and `l_name`: First and last name of the user (strings, 50 characters each), used for personalized interactions.
- `program_code`: Represents the academic program code (string, 10 characters), allowing the chatbot to tailor responses to the user's program.
- `course_taken`: A JSONB field containing a list of courses completed by the user, stored as an array of integers. This enables easy tracking of user progress and tailoring course recommendations.
- `path_interest`: Represents the user's academic or career interests (string, 25 characters), used to provide personalized advice.
- `course_to_take`: Stores the next course the user is planning to take (integer), helping the chatbot offer relevant advice or information about prerequisites.

The Users table is crucial for ensuring the chatbot can provide a personalized experience. By storing essential user details such as academic programs, completed courses, and interests, we enable the chatbot to cater to its recommendations and answers effectively.

### Sessions Table

Attributes:

- `session_id`: Primary key (serial), uniquely identifying each user session.
- `user_id`: Foreign key referencing the `user_id` in the Users table, linking session data to the corresponding user.
- `created_at`: Timestamp with timezone, representing the session start time, defaulting to the current timestamp for efficient session tracking.
- `end_chattime`: Timestamp with timezone, representing when a user ends a session, allowing us to measure session duration.
- `conversation`: JSONB field that stores the entire chat conversation of the session, making it easy to retrieve chat history in a structured format.

The Sessions table is essential for tracking user interactions, storing chat history, and linking it back to the user. This allows the chatbot to retrieve and continue previous conversations, providing a more seamless user experience.

## Design Considerations for a Functional Prototype

In order to ensure a functional working prototype, we focused on simplicity and ease of data management. The primary goals were to make it straightforward to insert and retrieve user data and chat history, which is why the Users and Sessions tables were designed with this principle in mind.

### Data Insertion:

User records are created when they first interact with the chatbot, using simple attributes to identify the user and track their program details. Chat sessions are recorded every time a user starts a new interaction, with minimal fields to reduce complexity and maximize efficiency.

### Data Retrieval:

To provide quick and relevant responses, data retrieval was optimized with indexes on key columns like `user_id`, `program_code`, `course_taken`, and `conversation`. This makes it easy to look up user progress, generate course recommendations, and display chat history without significant performance bottlenecks.

### Simplified Structure:

To avoid overcomplicating the system during the early stages, we consolidated chat history and session details into the Sessions table. This simplified structure allows us to avoid maintaining multiple collections or tables, streamlining both data insertion and retrieval.

## Chat History Caching Implementation

The Sessions table plays a dual role by acting as both a session tracker and a chat history repository. The conversation attribute uses the JSONB format to store chat logs, making it possible to cache chat interactions in an efficient, searchable format. The decision to store chat history within the Sessions table rather than maintaining a separate collection, such as the previous ChatHistoryLog, was made to simplify our data model and ensure that each session could be easily traced back to a specific user. This approach allows:

### Easy Insertion:

Each session is recorded with a new row, and the corresponding conversation is added as a JSONB object, keeping the data insertion process simple and efficient.

### Seamless Retrieval:

By indexing `user_id` and `conversation`, retrieving past interactions during user sessions becomes efficient. Users can easily pick up where they left off, contributing to a seamless user experience.

## User-Centric Data Model

The Users and Sessions tables are designed to support a user-centric model where personalization and responsiveness are key. By keeping track of user-specific data such as courses taken, programs of interest, and chat history, we ensure that each user's experience is unique and relevant to their needs. The focus was on reducing the complexity of the data structure while maintaining the flexibility needed to personalize the chatbot's responses.

The diagram illustrates a database schema for a university system. The tables and their attributes are as follows:

- USERS** (PK: user\_id, Primary Key):
  - user\_id (PK)
  - username
  - user\_name
  - first\_name
  - last\_name
  - email
  - password\_hash
  - account\_created
  - last\_login
  - user\_preferences
- PROGRAM\_OWNER** (PK: program\_owner\_id, Primary Key):
  - program\_owner\_id (PK)
  - first\_name
  - last\_name
  - ssn
  - email
  - contact\_number
  - title
- PROGRAMS** (PK: program\_id, Primary Key):
  - program\_id (PK)
  - is\_distributer
  - is\_engineer
  - program\_name
  - prerequisite\_program
  - core\_programs
  - elective\_programs
  - concentration\_programs
  - program\_level
  - program\_code
  - session\_programs
  - concentration\_programs
  - program\_owner\_id
  - program\_status
- SESSION\_LOG** (PK: session\_id, Primary Key):
  - session\_id (PK)
  - user\_id
  - session\_start
  - session\_end
  - session\_duration
  - session\_status
- COURSE\_LIST** (PK: course\_id, Primary Key):
  - course\_id (PK)
  - course\_name
- STUDENT** (PK: student\_id, Primary Key):
  - student\_id (PK)
  - user\_id
  - program\_id
  - course\_status
  - public\_interest
  - interest\_id
  - ssn
  - graduation\_year
  - is\_in\_state
  - is\_out\_of\_state
  - is\_f1
  - is\_f2
  - is\_f3
  - is\_other
  - other\_address
  - is\_permanent
  - is\_full\_time
- COURSE\_REGISTRATION** (PK: registration\_id, Primary Key):
  - registration\_id (PK)
  - course\_id
  - course\_id
- NONINTEREST\_OPTION** (PK: noninterest\_option\_id, Primary Key):
  - noninterest\_option\_id (PK)
  - option\_name
- COURSE\_PREREQUISITE** (PK: prerequisite\_id, Primary Key):
  - prerequisite\_id (PK)
  - course\_id
  - course\_id
- NONINTEREST\_REGISTRATION** (PK: noninterest\_registration\_id, Primary Key):
  - noninterest\_registration\_id (PK)
  - public\_interest\_id
  - public\_interest\_id

Relationships are indicated by lines with crow's foot notation:

- USERS** to **SESSION\_LOG**: 1:M (1 user can have many sessions).
- PROGRAM\_OWNER** to **PROGRAMS**: 1:M (1 owner can have many programs).
- PROGRAMS** to **SESSION\_LOG**: 1:M (1 program can have many sessions).
- PROGRAMS** to **COURSE\_LIST**: 1:M (1 program can have many course lists).
- STUDENT** to **COURSE\_LIST**: 1:M (1 student can have many course lists).
- STUDENT** to **COURSE\_REGISTRATION**: 1:M (1 student can have many registrations).
- STUDENT** to **NONINTEREST\_OPTION**: 1:M (1 student can have many non-interest options).
- STUDENT** to **NONINTEREST\_REGISTRATION**: 1:M (1 student can have many non-interest registrations).
- COURSE\_LIST** to **COURSE\_REGISTRATION**: 1:M (1 course list can have many registrations).
- COURSE\_LIST** to **NONINTEREST\_OPTION**: 1:M (1 course list can have many non-interest options).
- COURSE\_LIST** to **COURSE\_PREREQUISITE**: 1:M (1 course list can have many prerequisites).
- COURSE\_LIST** to **NONINTEREST\_REGISTRATION**: 1:M (1 course list can have many non-interest registrations).

## Future Goal: Optimized Relational Database Design

The current database schema represents an **optimal structured relational database** for managing student, program, course, and session data within our chatbot application. Each table serves a distinct purpose, with **bridge tables** effectively capturing many-to-many relationships, and **indexes** in place to ensure efficient data retrieval. The key features of this schema are:

- **Separation of Concerns:** Different aspects of the user journey are separated into tables such as **Users**, **Programs**, **Courses**, and **Sessions**, ensuring that each component of the database is logically organized.
- **Bridge Tables for Relationships:** By using **bridge tables** like **course\_registration**, **course\_prerequisite**, and **pathinterest\_registration**, we can represent complex relationships between entities, such as which courses are prerequisites for others, and which courses are part of different paths of interest.
- **Indexes on Foreign Keys:** Foreign key fields in critical tables such as **course\_registration**, **course\_prerequisite**, and **session\_log** are indexed to optimize query performance, ensuring the database can handle complex queries efficiently.
- **Detailed User and Session Tracking:** With tables like **Users**, **Student**, **Sessions**, and **session\_log**, we can track the complete lifecycle of user interactions, including user preferences, program enrollment, courses taken, and chat history, providing a detailed picture of user behavior.

## Future Work: BU Course Registration Integration

**BU Course Registration Integration** is an envisioned feature that could significantly enhance our chatbot's capabilities. If granted access by BU MET to their **Student Management System (SMS)** database, we would integrate official student data into our system, creating a more cohesive and personalized experience for the user. This would involve the following steps:

- **Data Lake Integration:** We could establish a **data lake** to store and process data from the BU MET SMS. The data lake would serve as an intermediary, allowing us to efficiently collect, clean, and integrate student data into our **PostgreSQL database**.
- **Expanded Student Collection:** We would add a **Student collection** to our current PostgreSQL setup, including attributes like:
  - **\_id (Primary Key):** A unique identifier for each student.
  - **name:** The student's full name (string).
  - **completed\_courses:** An array of course IDs representing courses completed by the student.
  - **registered\_program:** The program ID representing the program the student is registered in.
- **Retraining the AI Model:** Once student data is integrated, we would **retrain the AI model** using this expanded dataset, allowing the chatbot to deliver more accurate and tailored advice based on actual student records from BU MET.

### Adapting the Original Architecture

Integrating data from BU MET would require modifications to our original architecture, including:

- **Natural Keys Usage:** We plan to use **natural keys** from BU MET's systems to align our data structure with the official university codes. For example, a course like **Software Engineering** might have a course ID of '673', and the **MS in Software Development** program might be represented by 'MSSD'.
- **New Relationships:** The introduction of data from BU MET would involve adding new relationships in our relational database, necessitating additional **foreign keys** and **bridge tables** to manage associations between students, programs, and courses more effectively.

### Integration with Python AI Service and Langchain RAG

By integrating the expanded **PostgreSQL database** with the **Python AI service**, we can make significant improvements to how our core features, like **course building and recommendation**, function:

- **Direct Database Access for Course Builder.py:**
  - Currently, the **course builder.py** relies on **separate CSV files** to fetch program, course, and prerequisite information. This approach, while functional, is not scalable for larger datasets or when data requires frequent updates.
  - With the proposed integration, **course builder.py** would directly pull data from the **PostgreSQL database**. This would:
    - **Improve Data Accuracy:** Direct integration means that any update to courses, programs, or prerequisites in the database is instantly available to the course builder. This reduces redundancy and the possibility of discrepancies.
    - **Increase Scalability:** Instead of updating and managing multiple CSV files, all course and program data would be managed centrally, making the course builder's logic easier to maintain and scale as new programs or changes are introduced.
    - **Modular Course Builder:** The real-time data access would enable us to make **course builder.py** more modular, allowing us to adjust course recommendations dynamically based on program updates or student progress without manual interventions.
- **Improved Langchain RAG Integration:**
  - Currently, the **Langchain Retrieval-Augmented Generation (RAG)** system relies on preprocessed data to generate responses. By integrating **PostgreSQL**, Langchain would be able to query the database in real time, allowing it to work with the most up-to-date information.
  - This database integration would allow Langchain to:



- **Provide Real-Time Responses:** For queries like course details, program requirements, or prerequisite information, Langchain could directly pull data from the **Courses**, **Programs**, and **Course Prerequisite** tables, ensuring the chatbot always provides accurate, up-to-date responses.
- **Dynamic Querying:** By leveraging the relational structure of the database, Langchain could perform more complex queries—such as identifying suitable elective courses based on a student’s completed courses or program path interests—without having to rely on static files or preprocess everything in advance.
- **Scalability and Efficiency:** Moving to a database-driven approach instead of using static CSV files or preprocessed data allows the **chatbot system** to scale seamlessly as the number of courses, programs, and users grows. With efficient indexing and the relational structure, querying large volumes of data will remain performant.

#### Benefits of This Future Integration

- **Enhanced Personalization:** Access to official BU MET data would allow us to provide more personalized course recommendations, identify missing prerequisites, and offer tailored graduation timelines for students based on their actual progress.
- **Seamless AI Integration:** By connecting the **Python AI service** and **Langchain RAG** to the centralized PostgreSQL database, we streamline data flow across components. The AI services can provide more **context-aware recommendations** by leveraging detailed user data in real time.
- **Better User Experience:** By having access to **real-time course enrollment and completion data**, the chatbot could provide up-to-date advice regarding class availability, prerequisites, and program changes, making it a one-stop solution for academic planning.
- **System Efficiency:** Using **natural keys** and integrating data with a standardized identifier system ensures **consistency and accuracy** across all tables, reducing data redundancy and improving the overall efficiency of our queries. This would reduce the load on developers to maintain static files and instead focus on expanding and improving features.

The proposed expansion of our chatbot database to integrate BU MET's Student Management/Registration System data represents a strategic enhancement that would greatly benefit user experience, allowing us to offer more personalized and precise guidance to students. By integrating the Python AI service and Langchain RAG directly with the PostgreSQL database, we enhance the scalability, efficiency, and responsiveness of the entire system. This future work would involve adapting the architecture and implementing additional relationships, but the potential for providing a superior level of service makes it a worthwhile goal for subsequent project phases.

## Security Design

### User Authentication

- **JWT Authentication:** The chatbot system utilizes JSON Web Tokens (JWT) for secure user authentication. Due to time constraints, we switched from integrating Okta Authentication to JWT. This decision was made to ensure timely project completion while maintaining a robust authentication mechanism. JWT allows for stateless authentication and can be easily integrated into our existing architecture. It provides a lightweight solution that securely transmits user information between parties as a JSON object. The token is digitally signed using a secret key, ensuring its integrity and authenticity. Our implementation includes token generation upon successful login, validation for each protected API request, and a refresh mechanism to maintain user sessions securely. We've also implemented token expiration and secure storage practices to mitigate potential security risks. In the future, if time permits, we plan to switch back to Okta for its enhanced security features, including advanced identity management and single sign-on (SSO) capabilities. This transition would provide additional benefits such as multi-factor authentication, user provisioning, and more sophisticated access control policies, further strengthening our application's security posture.
- **Requirement Checks for input:** For both the login and sign-up pages, we implemented several input validation checks aimed at ensuring security and preventing common web vulnerabilities, such as Cross-Site Scripting (XSS). Usernames are restricted to a specific pattern, allowing only alphanumeric characters and a few safe special symbols (like underscores and hyphens). This eliminates the risk of attackers injecting harmful scripts or HTML elements that could execute malicious code in the user's browser. Passwords must meet strict complexity requirements, including at least eight characters with a combination of uppercase and lowercase letters, numbers, and special characters. This not only ensures strong protection against brute-force attacks but also prevents attackers from using simplistic, guessable passwords. Additionally, for sign-up forms, we enforce that the "password" and "confirm password" fields must match to ensure consistency in user input. These input validation checks, combined with both client-side and server-side sanitization, form an essential defense against XSS attacks, as they prevent attackers from embedding malicious scripts in input fields. By rejecting invalid or potentially harmful inputs before they are processed, we reduce the risk of the application executing malicious code, making this an effective layer of protection for our authentication process.

### Code Quality Assurance

**SonarQube:** The system employs SonarQube, an open-source platform for continuous code quality inspection. It plays a critical role in our development process by automatically detecting bugs, vulnerabilities, and code smells in our software projects. Regular integration of SonarQube scans helps maintain high standards of code quality and security, allowing developers to identify and resolve potential issues early in the development lifecycle.

## Automatic Security Vulnerability Detection

- We leverage GitHub's built-in security features to enhance our code quality and security posture. GitHub automatically scans our repositories for known vulnerabilities in dependencies and the codebase. This includes dependency scanning to identify vulnerabilities in project dependencies, code scanning using CodeQL to analyze code for potential security issues, and secret scanning to detect accidentally committed secrets like API keys or tokens. GitHub's Dependabot feature automatically creates pull requests to update dependencies to the latest secure versions when vulnerabilities are detected and suggests fixes for identified security issues in the codebase. The platform also provides detailed security advisories and sends automated alerts to the development team about critical security issues, allowing for prompt action. By integrating these GitHub features into our workflow, we ensure continuous monitoring and rapid response to potential security threats.

## Sensitive Data Handling

- **Data Minimization:** The system adheres to the principle of data minimization, only collecting and storing the minimum amount of personal information necessary for its functionality. This practice not only enhances user privacy but also reduces the risk of exposure in case of a data breach.
- **Chat Logs Management:** Chat history is retained in the system for a predefined period (e.g., 30 days) and is automatically deleted thereafter. Users have the option to save or export their chat history via email or other methods before deletion. This policy ensures compliance with data retention best practices while allowing users to maintain access to important interactions.
- **Personal Identifiable Information (PII) Protection:** Any PII stored in the system, such as names or email addresses, is encrypted to prevent unauthorized access. This encryption applies both at rest and in transit, ensuring that sensitive data remains secure against potential breaches.

## Secure Session Management

- **Session Cookies:** Session cookies are marked with the Secure and HttpOnly flags, ensuring they are transmitted only over HTTPS. This prevents interception during transmission and makes them inaccessible via JavaScript, mitigating the risk of cross-site scripting (XSS) attacks. The Secure flag guarantees that the cookie is sent only over encrypted HTTPS connections, protecting it from potential eavesdropping on network traffic. The HttpOnly flag prevents client-side scripts from accessing the cookie, which is a critical defense against XSS attacks that attempt to steal session identifiers. Additionally, we've implemented SameSite cookie attributes set to 'Strict' to prevent cross-site request forgery (CSRF) attacks. These measures collectively create a robust defense against common web application vulnerabilities related to session management.

- **Session Timeout:** The system includes an automatic session timeout feature, logging users out after a specified period of inactivity. This mechanism is crucial in preventing unauthorized access to user accounts, especially in shared or public environments, thereby enhancing overall system security. We've set the inactivity threshold to 30 minutes, balancing security needs with user convenience. After this period, the user's session is invalidated on the server-side, and any subsequent requests require re-authentication. This feature is implemented using a combination of server-side session tracking and client-side JavaScript to provide a seamless user experience. Users receive a warning notification shortly before the timeout occurs, allowing them to extend their session if they're still active. In addition to inactivity timeouts, we've also implemented absolute session timeouts of 30 minutes, requiring users to re-authenticate periodically regardless of activity, further reducing the window of opportunity for session hijacking attacks.

## Business Logic and/or Key Algorithms

### Business Logic for the BUAN Chatbot Academic Advisor Support Web App

#### 1. Program and Course Information Retrieval:

- Objective: Allow students to query program and course information by interacting with the chatbot.
- Logic: The chatbot will use data from `programs.csv` and `courses.csv` to provide students with detailed information about their selected program, including program type, core courses, elective options, credit requirements, and prerequisite details.
- Flow:
  - The user asks about a specific program or course.
  - The chatbot extracts the relevant program/course ID and retrieves details from the CSV files.
  - The chatbot presents the information in a readable format to the student.

#### 2. Course Selection for the Upcoming Semester:

- Objective: Help students build their course selection for the next semester based on courses they've already taken, their chosen program, and their path of interest.
- Logic:
  - The student provides their path interest, e.g., "AI/ML," "Web Development," or "Data Science."
  - The chatbot checks the list of courses the student has taken (input provided by the user or from chat history).
  - The course recommendation system dynamically builds a list of recommended courses using decision logic to ensure prerequisites are met, elective requirements are filled, and the student's preferences are accounted for.
- Flow:
  - The user provides a list of courses taken and their academic interests.
  - The system builds a course tree and recommends a selection of core and elective courses for the upcoming semester.
  - The chatbot presents the course list, including details about credit hours and prerequisites.

#### 3. Personalized Chat History Caching and Retrieval:

- Objective: Cache and retrieve previous interactions to maintain context during the conversation and allow students to reference past advice.
- Logic:
  - When a student interacts with the chatbot, the session's questions, answers, and recommendations are cached.
  - This cache is stored in a database, linked to the student's unique identifier.

- The chatbot can retrieve past interactions based on this identifier to resume conversations or reference prior recommendations.
- Flow:
  - Upon logging in, the chatbot retrieves the student's chat history.
  - Students can ask to revisit past recommendations or questions.
  - The chatbot can reuse cached data or update it based on any new input from the student.

#### 4. (Iteration 2) Logic-Based Query Handling:

- Objective: Extend the chatbot's ability to handle more complex, logic-based queries that cannot be answered solely using the program and courses CSV files.
- Logic:
  - Use custom logic rules to answer questions such as "What is the best path for AI/ML in this program?" or "What courses should I take to specialize in Web Development?"
  - Logic-based queries may involve integrating external knowledge bases or algorithms to handle more subjective or multifaceted questions.
- Flow:
  - The chatbot identifies the type of query and, if it cannot be answered by the CSV data, applies additional logic rules or performs a more advanced query on available program/course data.
  - The chatbot formulates a response based on those rules or paths.

By following this business logic, the BUAN Chatbot provides comprehensive academic support, offering both general information and personalized course recommendations.

### Algorithm Overview (course builder)

The course recommendation system in the provided code uses a **tree-based structure** to guide students through course sequences based on their **path of interest** and **courses already taken**. It implements an adaptive course-building algorithm that balances core courses and electives, while considering prerequisites and course sequences. Here's a breakdown of how it works:

#### Input

1. **Programs Data (read\_programs\_csv):**
  - Takes a CSV file containing information on various programs, their required core courses, elective courses, prerequisites, and other metadata.
  - Returns a dictionary where each program ID maps to its corresponding data (core/elective courses, credits, prerequisites, etc.).
2. **Courses Data (read\_courses\_csv):**
  - Takes a CSV file containing course details such as name, credits, difficulty level, prerequisites, and department.
  - Returns a dictionary where each course ID maps to its specific details.

### 3. Student's Profile:

- **course\_taken:** A list of courses the student has already completed.
- **path\_interest:** The student's declared area of interest (e.g., AI/ML, web development).
- **course\_to\_take:** The number of courses the student wishes to enroll in for the upcoming semester.

## Core Functions

### CourseTree Class

This class manages the course recommendations by constructing a tree-based structure:

1. **add\_branch(self, branch, course\_taken, skip\_courses=None):**
  - Adds a sequence of courses (called a branch) to the tree, ensuring that courses already completed or skipped are not re-added.
  - **Input:** A branch of course IDs to add, a list of completed courses, and optionally a list of courses to skip.
  - **Output:** A tree structure with nodes representing courses.
2. **build\_mssd\_tree(self, course\_taken, path\_interest):**
  - Dynamically builds a tree of courses based on a student's **path of interest** and **courses already taken**.
  - Handles course dependencies (e.g., skipping certain courses if prerequisites are fulfilled) and adds elective courses up to a specified number.
  - **Input:** Completed courses and a chosen path of interest.
  - **Output:** A complete course tree guiding the student through the recommended courses.
3. **recommend\_courses(self, course\_taken, path\_interest, course\_to\_take):**
  - Traverses the tree to recommend a specified number of courses for the next semester.
  - Fills elective courses based on the remaining courses the student needs.
  - **Input:** Completed courses, path of interest, and desired number of courses to take.
  - **Output:** A list of recommended courses.
4. **display\_tree(self, node=None, level=0):**
  - A helper function for visualizing the constructed course tree.

In the future, we hope that for each user, we can build a tree representing the courses they have taken, their program, and path interest. This tree will contain branches of all possible course pathway they can have, selecting the top 'x' courses based on their selected 'courses\_to\_take' value. Every time the user calls the recommended\_courses function through the chatbot interface, the algorithm will randomly select a branch and take their top 'x' courses. Doing this will make the application more efficient, caching their possible course pathway with the userId and limited the amount of times the system has to create a tree for each user.

## Algorithm Operation

### 1. Course Tree Construction:

- The program initializes a course tree with core course sequences based on the student's path of interest (e.g., AI/ML, data science).
- Branches are dynamically built to account for courses already taken (e.g., skipping core courses or adding prerequisites as needed).
- The tree is constructed by traversing through predefined sequences of core and elective courses, adding them as nodes in the tree.

### 2. Recommendation Process:

- After the tree is built, the recommendation process begins by **traversing** the tree.
- The algorithm identifies the courses that the student hasn't taken yet and selects courses in the right sequence, up to the number specified by the student.
- Prerequisite rules are enforced (e.g., skipping electives that depend on yet-to-be-taken core courses).

## Time Complexity

### 1. Tree Construction (`build_mssd_tree`):

- The tree is built by iterating over the course sequences and adding them as branches to the tree.
- Since each course is added only once and there are multiple paths depending on the core/elective configuration, the **time complexity** of this operation is roughly  **$O(n)$**  where  $n$  is the total number of courses being considered. This complexity might vary depending on the branching factor and number of electives.

### 2. Course Recommendation (`recommend_courses`):

- The course recommendation process involves traversing the tree to select new courses. Since each node is visited once, this operation also takes  **$O(n)$**  where  $n$  is the number of courses in the tree.
- **Traversing** the tree to recommend courses involves depth-first traversal and ensures that up to `course_to_take` number of courses are recommended.

### 3. Overall Complexity:

- Given that both tree-building and recommendation processes involve a linear pass through the course data, the overall time complexity is approximately  **$O(n)$** , where  $n$  is the number of courses considered in both the core and elective sequences.

## Outputs

- **Program and Course Data (from CSV files):** Two dictionaries with program and course information.
- **Course Recommendations:** A list of courses recommended for the upcoming semester based on the student's past coursework, path of interest, and elective needs.
- **Course Tree Visualization:** If needed, the course tree can be displayed, showing the hierarchical course progression.



This tree-based approach ensures that students receive **personalized recommendations** that are consistent with program requirements, electives, and course dependencies. The algorithm effectively handles core prerequisites and elective selections while allowing dynamic tree-building based on the student's progress.

## Design Patterns

### Factory Method Pattern

The **Factory Method Pattern** is employed to handle the dynamic creation of various chatbot services based on the user's specific requests, such as course recommendations, course information queries, or actions related to academic history like printing or emailing transcripts.

#### Use Case

When a user interacts with the BUAN chatbot, they may request different types of services, including:

- **Course Recommendations:** Personalized course suggestions based on their academic path, past courses, and program prerequisites.
- **Course Information:** Detailed queries about a specific course's description, prerequisites, or other details.
- **Transcript-Related Actions:** Such as printing or emailing the chat history and academic details for future reference.

Each of these services involves distinct logic and must instantiate the appropriate class to process the user's request. The **Factory Method Pattern** simplifies this by centralizing the object creation logic. For example, the system dynamically generates service classes like `CourseRecommendationService`, `CourseInfoService`, or `TranscriptService` based on the user's input.

#### Implementation

The `CourseTree` class is responsible for creating instances of these possible course pathway branches depending on the user's profile (e.g., `CourseTaken`, `ProgramCode`, and more). This factory ensures that new services can be introduced in the future without altering the application's core structure.

For example:

- If a new user signs up within the application and calls `recommended_courses()` for the first time, `CourseTree` will instantiate `CourseRecommendationService` to handle the request.
- If a user seeks course information, the factory will instantiate `CourseInfoService`.

This pattern provides flexibility and scalability, making it easy to add new services later without needing to modify existing code.

### Observer Pattern

The **Observer Pattern** is applied to enable real-time updates across different components of

the chatbot system, ensuring that any change in one component (such as receiving a new message) automatically triggers updates in other dependent components (like the chat window or chat history logger).

## Use Case

In the BUAN chatbot application, various components need to remain synchronized with real-time events, such as:

- A **new message** arriving in the chat.
- A user **clearing the chat history**.
- A transcript being **emailed or printed**.

When a new message is sent, multiple observers (such as the ChatWindow, which displays the current conversation, and the ChatHistoryLogger, which saves the conversation for later use) need to update in real time. The **Observer Pattern** allows these components to be notified of changes and refresh automatically.

## Implementation

The ChatSession class acts as the **subject** in this pattern, notifying all observers when an update occurs. Observers could include:

- **ChatWindow**: Updates the UI to show new messages or changes in the conversation.
- **ChatHistoryLogger**: Logs new messages or changes to the chat history, ensuring that conversations are properly cached and can be retrieved later.

This pattern keeps components loosely coupled, meaning they do not need to know about each other's internal workings, making the system easier to maintain and extend.

## Models & Tools

### Chatbot Model

For our project, we decided to use **ChatGPT-4o mini** via the Langchain API as the chatbot's underlying large language model (LLM) architecture. Our chatbot serves two primary purposes:

1. **Course Recommendations for the Upcoming Semester:**
  - **Objective:** Based on a student's academic program, the courses they have already taken, program prerequisites, and the number of courses they wish to take, the chatbot will recommend courses for the upcoming semester.
  - **Flow:**
    - Users are required to log in using their email before interacting with the chatbot.
    - Once logged in, the system will gather input parameters such as the student's program, academic interests, courses taken, and preferences for the number of courses.
    - The system will then run a decision tree algorithm to analyze these inputs and generate personalized course recommendations.
    - The result of this algorithm will be stored in a variable called response, which will then be passed to the ChatGPT-4o model via the Langchain API.
    - ChatGPT-4o will formulate a user-friendly reply based on the response, providing the student with the recommended courses.
2. **Course Information Lookup:**
  - **Objective:** Allow students to query the chatbot for specific course details.
  - **Flow:**
    - After receiving course recommendations, users can ask the chatbot for more information about any of the recommended courses or any other course offered by the program.
    - The chatbot will respond with details pulled from the program and course data stored in the CSV files, such as course descriptions, prerequisites, credit hours, and related information.

In both scenarios, after completing the interaction, the user can end the chat session. All chat history will be cached and associated with the user's account, enabling them to retrieve or share the chat history in the future for reference or printing.

By integrating **ChatGPT-4o** via Langchain, our system provides robust and intelligent academic advising capabilities that deliver personalized recommendations and course insights in real time.

## Tools Utilized

1. **JWT Authentication:** We implemented JWT (JSON Web Tokens) for secure user authentication. This method allows for stateless authentication, improving the user experience by enabling users to remain logged in across sessions without needing to constantly re-enter their credentials.
2. **Fetch API:** The Fetch API is employed for making network requests to the server. It allows our application to request and retrieve data from the backend efficiently, handling both requests for course recommendations and information lookups.
3. **Flask API:** Our backend is built using Flask, a lightweight web framework for Python. The Flask API serves as the intermediary between the chatbot and the database, managing requests for course data and user authentication.
4. **FastAPI:** We also utilize FastAPI, a modern web framework for building APIs with Python. FastAPI is known for its high performance, built-in data validation, and automatic generation of OpenAPI documentation. It enables asynchronous programming, making our API calls faster and more efficient, especially under load.
5. **FAISS:** Facebook AI Similarity Search. It is a library that allows developers to quickly search for embeddings of text documents that are similar to each other and feed the documents to the model for more contextual and meaningful responses.
6. **Cucumber Testing:** We use Cucumber for behavior-driven development (BDD) testing. This tool allows us to write acceptance tests in plain language, facilitating communication between developers, testers, and non-technical stakeholders. It helps ensure our application meets business requirements and performs as expected.
7. **Automation:** Automation tools and scripts are integrated into our workflow to streamline various processes, including testing, deployment, and continuous integration. Automation reduces human error and increases efficiency, allowing developers to focus on building features.
8. **JIRA:** We utilize JIRA for project management and tracking tasks. It helps organize work into manageable segments, assign responsibilities, and monitor progress, ensuring that all team members are aligned on goals and deadlines.
9. **GitHub:** GitHub serves as our version control system, where all project code is stored, tracked, and managed. It facilitates collaboration among team members through features like branching, pull requests, and issue tracking.
10. **Axios:** Axios is a promise-based HTTP client used in our frontend for making requests to the Flask and FastAPI backends. It simplifies the process of handling asynchronous requests and allows for easy integration of error handling and response data management.
11. **Captcha:** We integrated CAPTCHA into the user authentication process to add an extra layer of security. This helps prevent automated attacks and ensures that only legitimate users can access the system.
12. **Langchain:** Langchain is the framework we use to connect our chatbot with the ChatGPT model. It provides the necessary tools and abstractions to manage interactions, enabling us to build robust conversational agents.
13. **Docker:** Docker is used to containerize our application, ensuring that it runs consistently across different environments. This makes deployment easier and enhances scalability, allowing our application to handle varying loads.

14. **AWS:** We deploy our application on Amazon Web Services (AWS), leveraging its robust infrastructure for hosting, storage, and computing. AWS allows us to scale our resources dynamically based on demand and provides various services that enhance our application's capabilities.

## 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-4o, 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-4o 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-4o:** 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.

**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.