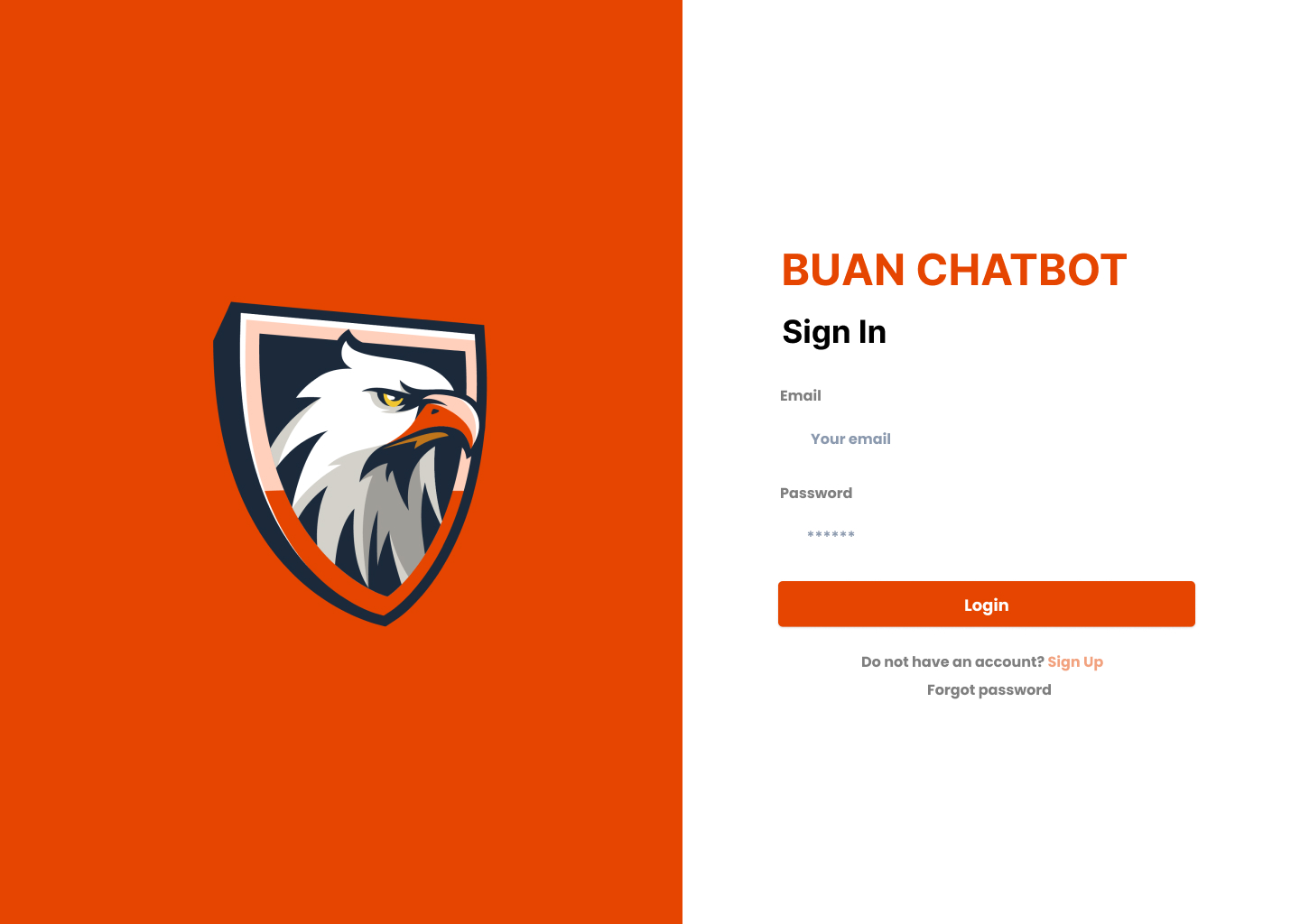
**CS673 Software Engineering** 

**Team 1 - BU Academic Navigator (BUAN)**

**Software Design Document**

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

**Revision history**

| **Version** | **Author** | **Date** | **Change** |
| --- | --- | --- | --- |
| 1.0.0 | Natthaphon Foithong | 9/18/2024 | Create Introduction, Software Architecture class diagram, UI and Security Design, and Design Pattern |
| 1.0.1 | Natasya Liew | 9/22/2024 | Proofread sections and edited Chatbot implementation sections. Added more design goals, business logic, algorithm, database design sections, and model/tools section. Added the glossary and references section. |
| 1.0.2 | Poom Chantarapornrat | 9/22/2024 | Formated and organized the entire document. Improved readability. Revised the AI Application Section |

[Introduction](#_heading=h.gjdgxs)

[Software Architecture](#_heading=h.30j0zll)

[Class Diagram](#_heading=h.1fob9te)

[UI Design (if applicable)](#_heading=h.3znysh7)

[Database Design (if applicable)](#_heading=h.2et92p0)

[Security Design](#_heading=h.tyjcwt)

[Business Logic and/or Key Algorithms](#_heading=h.3dy6vkm)

[Design Patterns](#_heading=h.1t3h5sf)

[Any Additional Topics you would like to include.](#_heading=h.4d34og8)

[References](#_heading=h.2s8eyo1)

[Glossary](#_heading=h.17dp8vu)

# 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 4.0 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 (MS SD) program. The BUAN chatbot system is designed with flexibility, scalability, and user-centricity in mind; integrating cutting-edge technologies, including React/Redux 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 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 the OpenAI’s ChatGPT 4.0 model via Langchain API. The Langchain API was also used for easy Retrieval-Augmented Generation (RAG) to provide accurate and relatable results to users.

1. *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.

1. *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.

1. *Security and User Privacy:*

Implement secure user authentication using Okta and 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.

1. *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.

1. *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.

1. *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.

1. *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.

1. *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.

1. *CI/CD Deployment and Scalability:*

The application will be containerized using Docker, ensuring consistent environments across development, testing, and production. GitHub Actions will be employed to establish the continuous integration and continuous deployment (CI/CD) pipelines, automating the building, testing, and deployment processes. The application will be deployed on AWS, leveraging cloud-based infrastructure to provide flexibility, scalability, and reliability in the production environment.

1. *Multi-language Integration:*

This project demonstrates the power of integrating multiple programming languages and technologies, including React/Redux 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.

1. *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, 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, Redux, 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. Okta handles authentication, and the system is deployed using Docker and AWS for CI/CD.

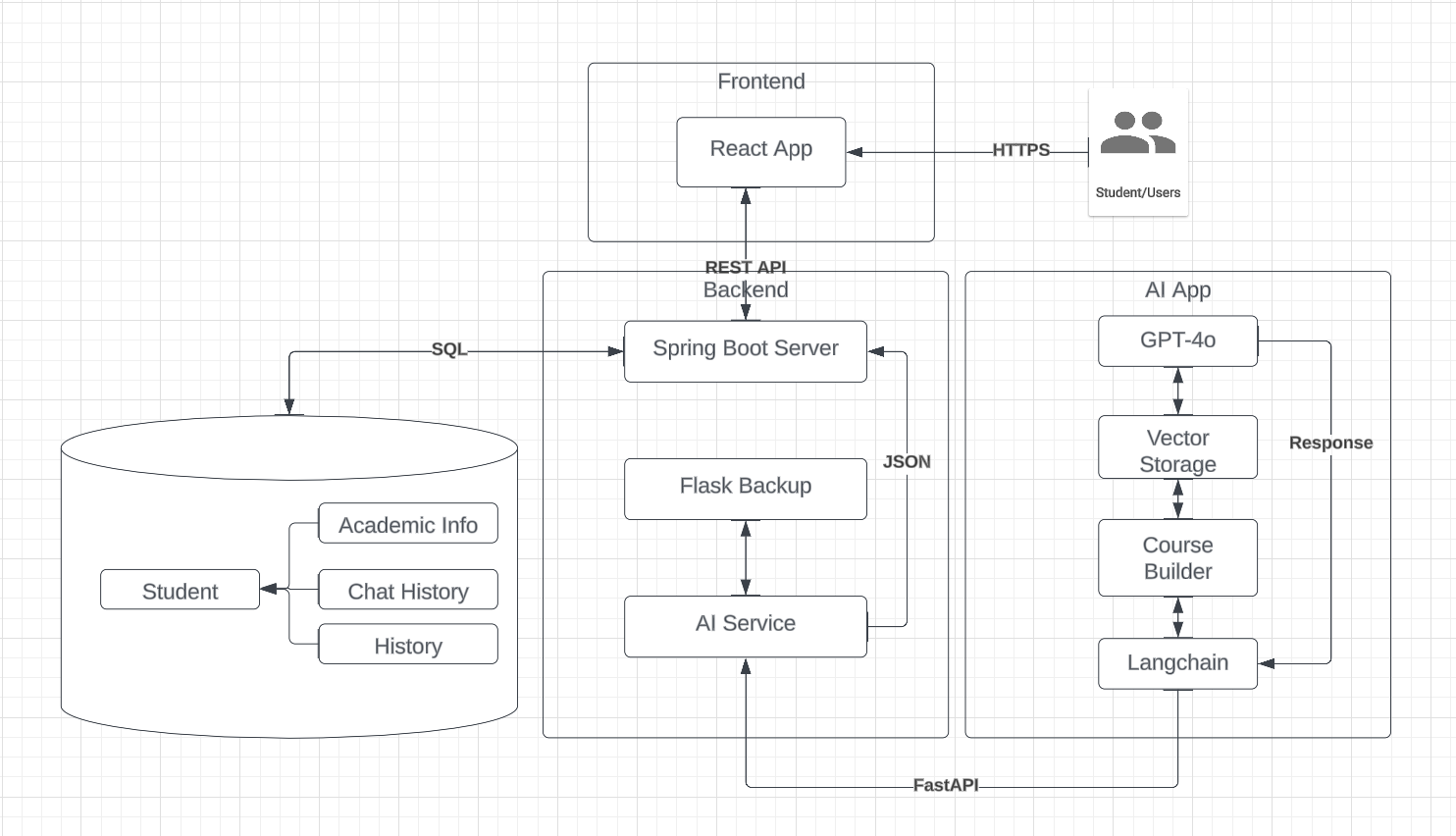


Figure 1. System Architecture Diagram

**Front-end Layer** – *(React, Redux, Fetch API)*

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.
* *Axios Service –* Handles communication between the frontend and backend via RESTful API calls.
* *Chat History Sharing and Emailing –* Allows users to email or print their chat history.
* *Caching –* Cookies or local storage are used to cache chat data and user preferences for faster loading.
* *User Authentication –* Okta authentication ensures that the user is logged in securely.

**Frameworks and Libraries**

* *React.js* –For building responsive and interactive UIs on the web.
* *Axios –*For HTTP requests to communicate with the backend.
* *Redux –*For state management in larger components.
* *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 Okta including token verification.
* Data Layer – communicates with PostgreSQL database

**Frameworks and Libraries**

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

**AI Application Layer** – (Python, Langchain, FastAPI)This application layer is responsible for handling AI/chat responses to user inputs and consists of several key components. It integrates a large language model (GPT-4) using LangChain to manage complex queries and answer course-related questions by interacting with external data sources. Course and program data from Boston University is collected via a Python script, stored in CSV format, and embedded for the model to generate accurate responses. The data is stored in a Chroma vector database, accessible via Pinecone API, which serves as the document retriever within the LangChain workflow. To provide personalized course recommendations, a custom tree algorithm generates schedules 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. The stack also utilizes essential libraries and frameworks like Pinecone for vector storage and Pandas for data manipulation.

**Components**

* *LLM Application*

This component integrates a large language model (GPT-4o) to handle all the responses in natural language. Furthermore, we use Langchain to handle more complex queries such as answering the course description. LangChain is a framework designed to simplify building applications with large language models (LLMs). It allows developers to integrate LLMs with external data sources, manage complex workflows, and create Retrieval-Augmented Generation (RAG) systems for more contextually accurate responses. LangChain also provides tools for prompt engineering, memory management, document handling, and custom component creation, enabling efficient data interaction and decision-making within applications.

* *Data Mining*  
  In this project, we made a deliberate decision to avoid using web scraping techniques to collect course and program data from the Boston University MET website. This decision was driven not only by the desire to maintain the integrity of the website and ensure compliance with relevant data usage policies but also not wanting to burden the university IT team’s existing technical challenges in integrating with the university’s new Registration Management System (MS) with other online platforms. Given these limitations, we determined that it was in the best interest of the group to rely on internally compiled data. As a result, we utilized a pre-existing CSV file created by a team member, containing comprehensive information on courses and programs. This dataset is embedded within a large language model (LLM) framework, enabling accurate responses to user queries regarding academic courses and programs. By adopting this approach, we ensured data consistency, minimized potential system disruptions, and adhered to ethical standards in data collection.
* Vector Storage  
  To enable the model to access course and program data, we store it in a vector database. We chose Chroma for this purpose, with access provided via the Pinecone API. LangChain’s seamless integration with Pinecone simplifies this process. The vector database is then incorporated as the document retriever, which is linked into the main LLM chain for efficient data retrieval and question answering.
* *Course Builder Algorithm*  
  One user story requires generating a recommended course schedule, a complex task that LLMs may struggle with due to potential hallucinations. To address this, we developed a custom tree algorithm that generates a list of recommended courses based on the program, area of interest, and the number of courses previously taken. This structured output is then used by the model to provide natural language responses, ensuring accuracy and clarity in its recommendations.
* *API Creation*  
  After building the application using Python, LangChain, and vector storage, we need to expose it as an API for the backend and frontend teams to use. We chose FastAPI as our API framework due to its speed and efficiency in handling asynchronous requests. FastAPI simplifies the process of building RESTful APIs by automatically generating OpenAPI documentation and interactive Swagger UI, making it easy for developers to test and explore the endpoints. Its support for data validation through Pydantic ensures robust input handling, and the framework's performance rivals Node.js and Go, making it ideal for high-performance applications. Additionally, 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.
* *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.
* *Pinecone –*A vector database service designed to store and retrieve high-dimensional data, perfect for integrating with LLMs to support similarity search and real-time document retrieval.
* *Pandas –*A versatile Python library for data manipulation and analysis, offering powerful data structures like DataFrames to efficiently process and clean data for further use in the application.

**Database Layer** –(PostgreSQL)

The PostgreSQL database store user data, course history, and chat logs. The database ensures that all user interactions and queries are persisted for future retrieval, analysis, and personalized recommendation. The backend (Spring Boot) interacts with the PostgreSQL database via JDBC (Java Database Connectivity) to store and retrieve user data and chat logs.

**Components**

* *PostgreSQL* –Used for persisting chat history, Student information, and course histories. It provides structured relational storage for conversations between the user and the chatbot.
* *Redis* (optional) – Can be used for caching frequently accessed data like course lists or user chat history.

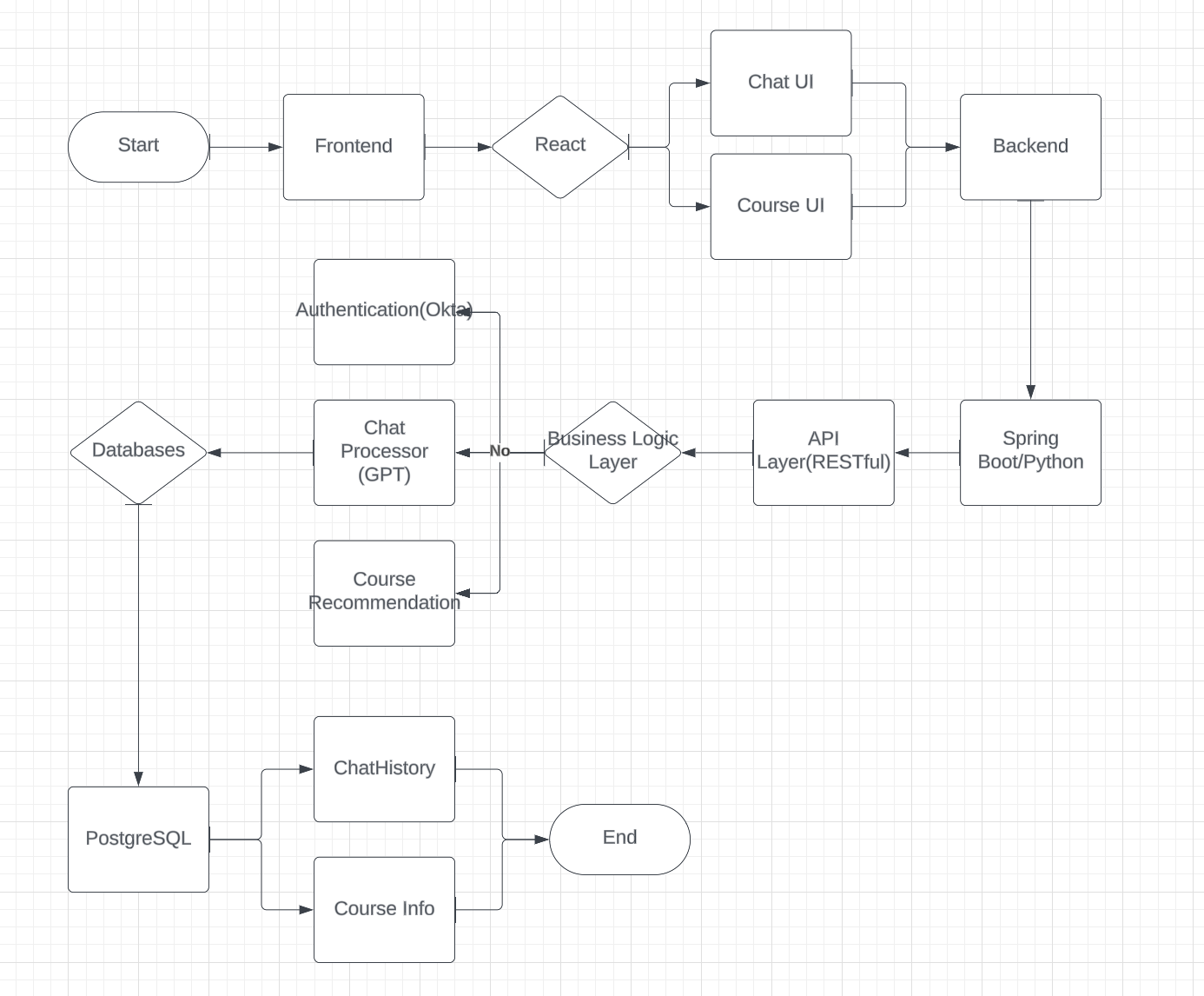


Figure 2. An Example of Data Communication Flow Across the Program

# Class Diagram

# 

Figure 3. Application Class Diagram

**Frontend Components**

*ChatPage*

* Attributes: chatHistory, userInput
* Methods: sendMessage(), emailChatHistory(), printChatHistory(), shareChatHistory()

*Button*

* Attributes: type, onClickAction
* Methods: trigger()

**Backend Components**

*ChatService*

* Methods: processMessage(), saveChatHistory()

*EmailService*

* Methods: sendEmail(email, chatHistory)

*ShareService*

* Methods: generateShareLink(chatHistory)

*PrintService*

* Methods: formatForPrint(chatHistory)

**Database Collections (PostgreSQL)**

*ChatHistoryLog*

* Attributes: log\_id, user\_id, created\_at, entry

*Programs*

* Attributes: program\_id, program\_name, core\_courses, elective\_course, total\_credits

*Courses*

* Attributes: course\_id, course\_name, credits, description, prerequisites

*Prerequisites*

* Attributes: course\_id, required\_courses

**Relationships between Classes**

*Frontend*

* ChatPage (Has-A) -> EmailButton
* ChatPage (Has-A) -> ShareButton
* ChatPage (Has-A) -> PrintButton

*Backend*

* ChatPage (Uses) -> ChatService
* ChatService (Interacts-With) -> ChatHistoryLog (PostgreSQL)
* ChatService (Calls) -> EmailService
* ChatService (Calls) -> ShareService
* ChatService (Calls) -> PrintService

*Database*

* ChatHistoryLog (Stores chat data)
* Programs, Courses, and Prerequisites are related through a One-To-Many relationship.

# 

# UI Design

1. **Layout Design**

The overall design layout follows a clean, Two-column Two-row structure:

*Left column (Side panel)*–For actions like ending the chat, emailing, sharing, or printing the chat history and logout.

Right column (Chat Interface) – For the conversation area and sending messages.

1. **Key UI Components**

*Chat Interface*

* Chat Window: The main chat window occupies the majority of the screen.
* Message Bubbles: User and Bot messages will appear in distinct bubbles, with user messages aligned to the right and bot responses aligned to the left.
* Chat Scroll: The window will have vertical scrolling to allow users to scroll through the chat history.
* Time Stamps (Optional): Each message will have a timestamp underneath the message bubble for tracking the conversation flow.
* Input Field: At the bottom of the chat window, there will be a text input field where users can type messages. It will include the following features:
* Text Box: A wide field for typing messages.
* Send Button: A button to send messages.
* Grammarly Check (Optional): An additional optional grammar-checking feature to provide real-time feedback on message content.

*Action Buttons*

* End Chat Button: Allows the user to close the current chat session
  + Function: Ends the conversation and logs the user out of the system.
  + Design: Simple “Logout” icon.
* Email Chat History Button: Lets users send the chat transcript to an email.
  + Function: Opens a modal where the user can enter an email address, then emails the chat history as an attachment.
  + Design: “Envelope” icon button.
* Share Chat Button: Generates a shareable link for the conversation.
  + Function: Upon click, the button will generate a unique URL link, which can be copied to share the chat history with others.
  + Design: “Share” icon button.
* Print Chat History Button: Prepares the chat transcript for printing.
  + Function: Opens a printable view of the chat, allowing the user to print the chat history.
  + Design: “Printer” icon button.
* Logout Button: Allows the user to end their session and return to the login page.

*User Input Validation*

* Input Suggestions: Auto-suggestions based on course names or major as the user types.
* Error Handling: If a user inputs an invalid course or an error occurs, the chatbot will prompt a helpful message to correct the action.

*ChatBot Modes and User Profile Panel (Optional)*

* 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.

1. **Colors and Typography**

*Color Palette*

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

*Typography*

* *Outfit*

1. **Responsive Design** (Optional)

*Mobile view*

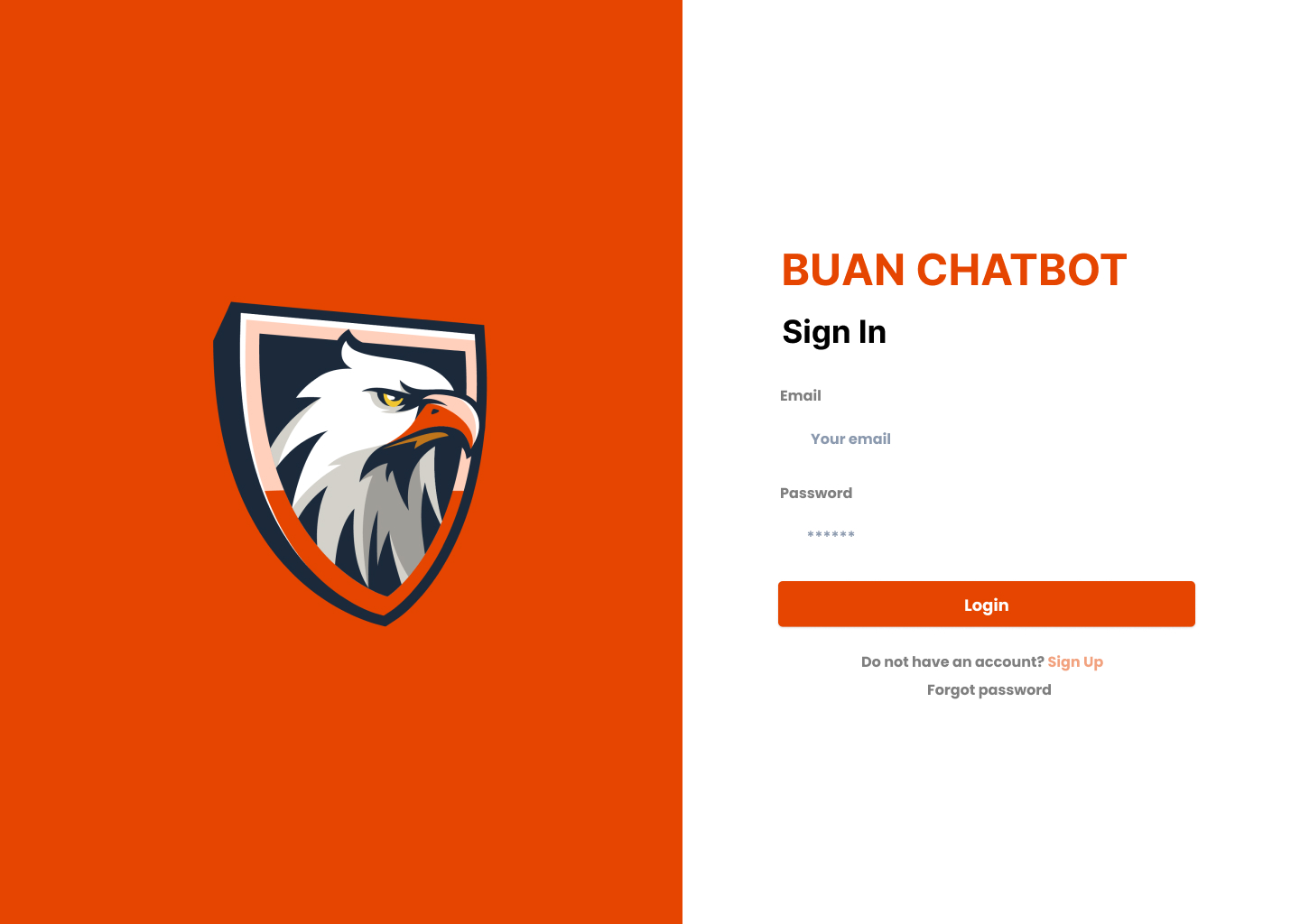
* + The chat interface collapses into a single-column view.
  + Action buttons are represented as icons, a toolbar or dropdown.
  + The input field remains sticky at the bottom for easy access.

1. **Navigation Flow**

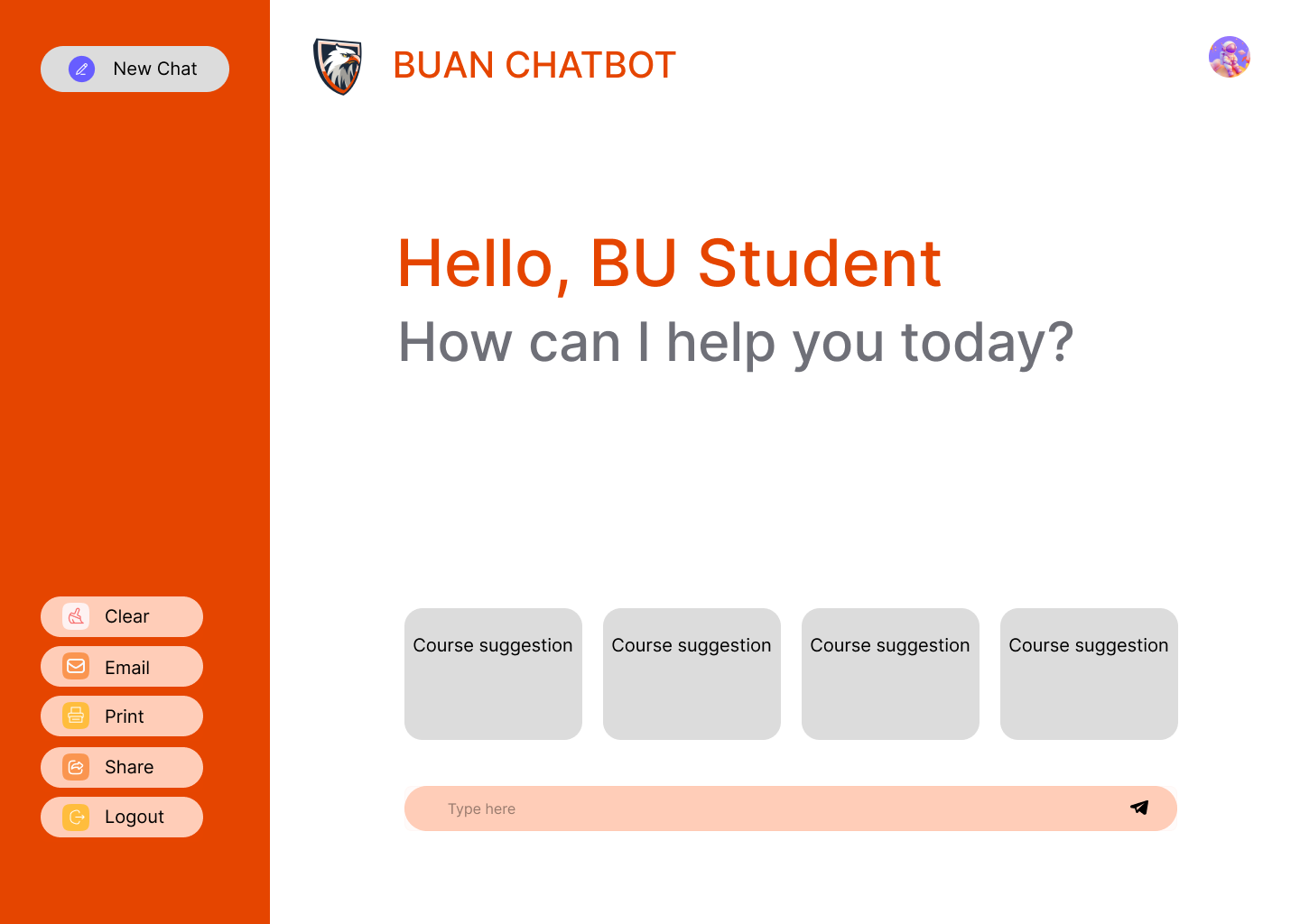
* *Login Page –*The user logs in with their email or BU login credentials via Okta authentication.
* *Welcome Page –*Once logged in, users are greeted with a welcome message from the chatbot, asking if they need help with course selection or course information.
* *Chat Interface –*Users engage with the chatbot, and action such as emailing, sharing, and printing can be taken during or after the conversation.
* *End Chat –*After completing the session, users can click “End Chat” to clear chat and logout.

1. **Examples of User Interfaces**

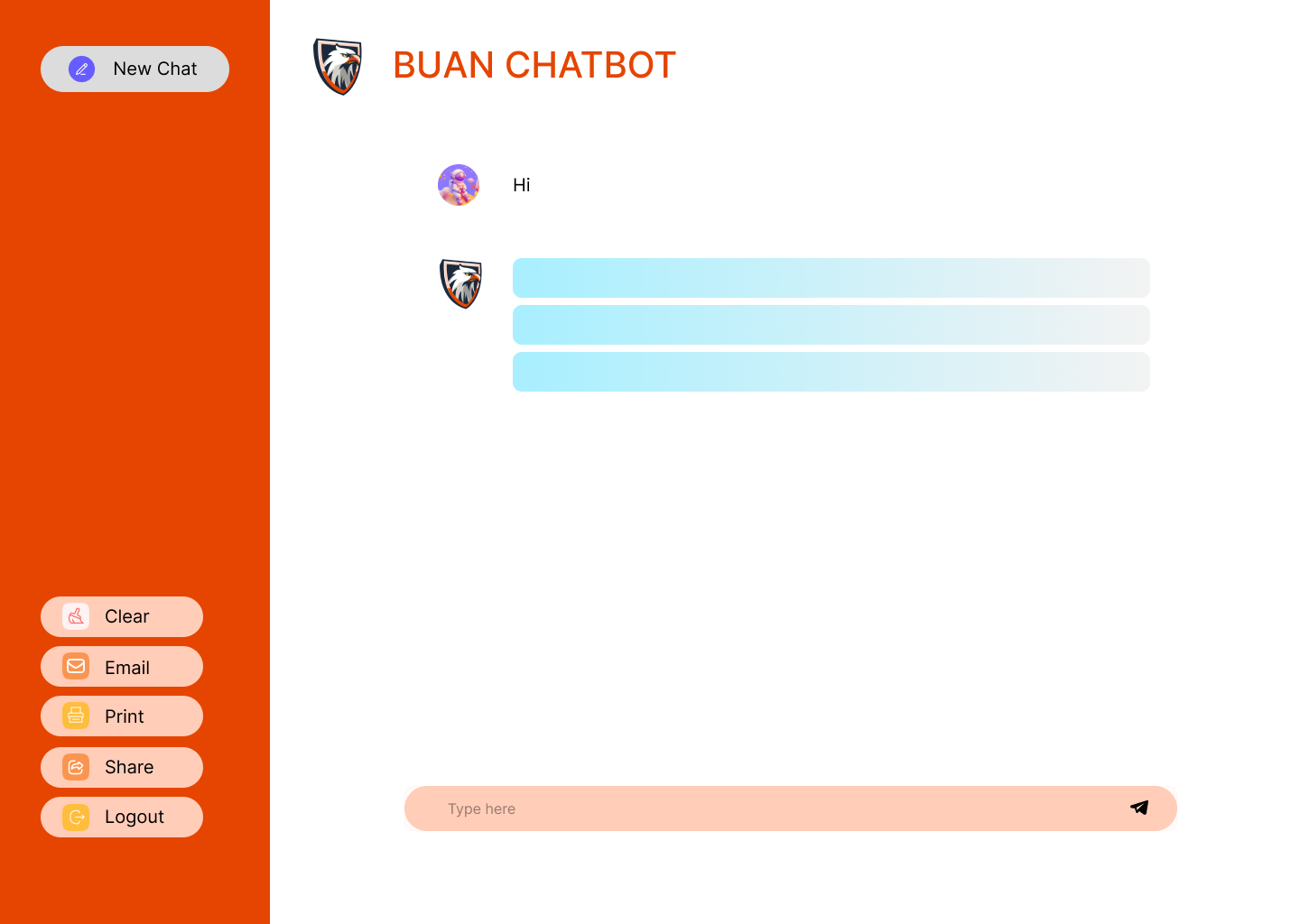
* The sign-in page

****

* The home page (chat screen)

****

* Response Loading Animation

****

Please see all UI designs in the Figma link below.<https://www.figma.com/design/gjNG1bADwnFxgDqclMwQVQ/Chat-AI-Bot---673ONE---final-mockup?node-id=0-1&t=hKYhsbJV2d2LeDvM-1>

# 

# Database Design

***Chatbot Implementation***

Using a PostgreSQL database, we created three collections: Programs, Courses, Prerequisites. Within the Programs collection, we have the attributes: program\_id (primary key), program\_name (string), core\_courses (array of course IDs), elective\_courses (array of course IDs), total\_credits (integer). Next within the Courses collection, we have the attributes: course\_id (primary key), course\_name (string), credits (integer), description (string), and prerequisites (array of prerequisite course IDs). Last, we have the Prerequisites collection, a bridge entity that contains the attributes: course\_id (from the Courses collection) and required\_courses (array of course IDs from the Courses collection).

***Chat History Caching Implementation***

Also using a PostgreSQL database, we created a collection called `ChatHistoryLog` which contains the attributes: log\_id (primary key; serial), user\_id (string) which is the email used by the user upon login authentication, created\_at (datetimestamp), entry (string) containing the chat history log entry.

**BU Course Registration Integration** (Optional/Future Work)

With the condition that BU MET approve us to have access of their Student Management System database, we can put the data into a data lake and add the Student collection with the attributes: \_id (primary key), name (string), completed\_courses (array of course IDs), registered\_program (program\_id), into the PostgreSQL database mentioned above and retrain the model for this additional information. But doing this means that our original architecture design must be altered.

For all IDs above, we will you the natural keys from the BU MET CS department. In other words, if the class is Software Engineering, the course ID will be ‘673’, while MS in Software Development will be ‘MSSD.’

# 

# Security Design

#### **User Authentication**

* *Okta Integration* – The chatbot system uses **Okta** for secure user authentication. This ensures that only authorized users (students, faculty, etc.) can log into the system using **single sign-on (SSO)** mechanisms.
* *SonarQube* – an open-source platform for continuous code quality inspection, detecting bugs, vulnerabilities, and code smells in software projects.

#### **Sensitive Data Handling**

* *Minimization –* The system adheres to the principle of data minimization, only collecting and storing the minimum amount of personal information necessary for functionality.
* *Chat Logs –* Chat history is retained in the system for a predefined period (e.g., 30 days) and then automatically deleted unless users opt to save or export it via email or other options.
* *Personal Identifiable Information (PII) –* Any PII stored in the system, such as names or email addresses, is encrypted to prevent unauthorized access.

#### **Email Security**

* *Email Chat History* – When users choose to email their chat history, the system:
  + - Encrypts the email contents and attachments using **SSL/TLS** before sending.
    - Ensures that only the intended recipient can decrypt and access the chat history.

#### **Sharing Security**

* *Unique Links –* When users share their chat history via a link, the system generates **time-limited unique URLs**. After a certain period (e.g., 24 hours), the link expires and becomes unusable to prevent unauthorized access.

#### **Secure Session Cookies**

* *Session Cookies –* Session cookies are marked with the **Secure** and **HttpOnly** flags to ensure they are transmitted over HTTPS and are inaccessible via JavaScript.

#### **Session Timeout**

* *Automatic Logout –* The system includes an automatic **session timeout** feature, logging users out after a period of inactivity to prevent unauthorized access.

# Business Logic and/or Key Algorithms

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

## 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.

## 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.

## 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.

## (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.

## 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 ChatServiceFactory class is responsible for creating instances of these service classes depending on the user's chosen options (e.g., CourseRecommendationService or CourseInfoService). This factory ensures that new services can be introduced in the future without altering the application's core structure.

For example:

* If a user selects the course recommendation option, ChatServiceFactory 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 4.0** 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 4.0 model via the Langchain API.
     + ChatGPT 4.0 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 4.0** via Langchain, our system provides robust and intelligent academic advising capabilities that deliver personalized recommendations and course insights in real time.

# 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/*](https://developer.okta.com/docs/reference/api-overview/)

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

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

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

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

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

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

***Foithong, N.*** *(2024, September 18). Chat AI Bot - 673ONE. Figma. Retrieved from* [*https://www.figma.com/design/gjNG1bADwnFxgDqclMwQVQ/Chat-AI-Bot---673ONE?node-id=0-1&node-type=canvas*](https://www.figma.com/design/gjNG1bADwnFxgDqclMwQVQ/Chat-AI-Bot---673ONE?node-id=0-1&node-type=canvas)

***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 4.0 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 4.0*: 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.

*Redux*: A state management library for JavaScript applications, often used with React.

*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.